Welcome Back!

Intro to OpenMP #2

Questions on hands-on?

Brief Recap

- OpenMP works on shared memory machines to simple parallelize pieces of otherwise serial codes.
- OpenMP works mainly through sets of compiler directives, marked with *#pragma omp* ... in C.
- Always pay attention to how variables are used (private, shared, reduction...). Beware of race conditions.
- Let's look at a couple of key slides from yesterday to refresh our memories.

MFOMP Loop II

Code:

```
//omp_simple_loop2.c
#include <stdio.h>
#include <omp.h>

int main(int argc, char *argv[])
{

#pragma omp parallel default(none)
  {
    int mythread=omp_get_thread_num();
#pragma omp for
  for (int i=0;i<15;i++)
    printf("Process %d gets i=%d\n",mythread,i);

  }
}
```

Behaves same as previous version, but we have now saved the repeated calls to omp_get_thread_num().

Output:

```
[sievers@tpb4 omp-intro]$ ./omp_simple_loop2
Process 3 gets i=12
Process 3 gets i=13
Process 3 gets i=14
Process 0 gets i=0
Process 0 gets i=1
Process 0 gets i=2
Process 0 gets i=3
Process 1 gets i=4
Process 1 gets i=5
Process 1 gets i=6
Process 1 gets i=7
Process 2 gets i=8
Process 2 gets i=9
Process 2 gets i=10
Process 2 gets i=11
[sievers@tpb4 omp-intro]$
```

Parallel ndot - Atomic Reduction

Code:

```
NType ndot_atomic_reduce(int n, NType *x, NType *y)
     // This version of ndot should be OK.
{
  NType tot=0;
#pragma omp parallel shared(x,y,n,tot)
  {
    NType mytot=0;
#pragma omp for
    for (int i=0;i<n;i++)
      mytot+=x[i]*y[i];

#pragma omp atomic
    tot+=mytot;
  }
  return tot;
}
```

Output:

```
[sievers@tpb4 omp-intro]$ setenv OMP_NUM_THREADS 1
[sievers@tpb4 omp-intro]$ ./omp_ndot_race
Dot product is     3.3333e+20 (vs     3.3333e+20) for n=10000000.  Took   9.3732e-02 seconds.
[sievers@tpb4 omp-intro]$ setenv OMP_NUM_THREADS 8
[sievers@tpb4 omp-intro]$ ./omp_ndot_race
Dot product is     3.3333e+20 (vs     3.3333e+20) for n=10000000.  Took   3.6198e-02 seconds.
[sievers@tpb4 omp-intro]$
```

Now we're in business! Correct answer, ~3x faster than serial.

Load Balancing

- Sometimes not all elements of a loop have the same work. If some threads finish early and have to wait for others, we will take a performance hit.
- Giving equal amounts of work to each thread (not equal number of loop bits) is called *load balancing*.
- OpenMP supports some load balancing. The *schedule* clause added to *omp for* will change how work is shared.
- We can decide either at compile-time (*static schedule*) or run-time (*dynamic schedule*) how work will be split.

Scheduling

- By default, each thread gets a big consecutive chunk of the loop. Often, just giving each thread many smaller interleaved chunks of the problem works.
- *#pragma omp for schedule(static, m)* gives *m* consecutive loop elements to each thread instead of a big chunk.
- Sometimes we need to be more flexible. If we use *schedule(dynamic, m)*, each thread will work through m loop elements, then go to the OpenMP run-time system and ask for more.
- Load balancing (possibly) better with *dynamic*, but larger overhead than with *static*.

Example - Mandlebrot Set

- Mandlebrot set simple example of non-balanced problem.
- Defined as complex points a where $|b_\infty|$ finite, b_0=0, $b_{n+1}=b_n^2+a$. If $|b_n|$ ever gets bigger than 2, point diverges.
- To calculate, pick some n_{max}, iterate at each point a, and see which ones cross 2. Outside of set, points can diverge very quickly (2-3 iterations). Inside, we have to do lots of work (maybe 1000 per point).
- If a thread gets a chunk mostly outside of set, will be very fast. Mostly inside, very slow.

Mandlebrot Set Code

Code to run loops:

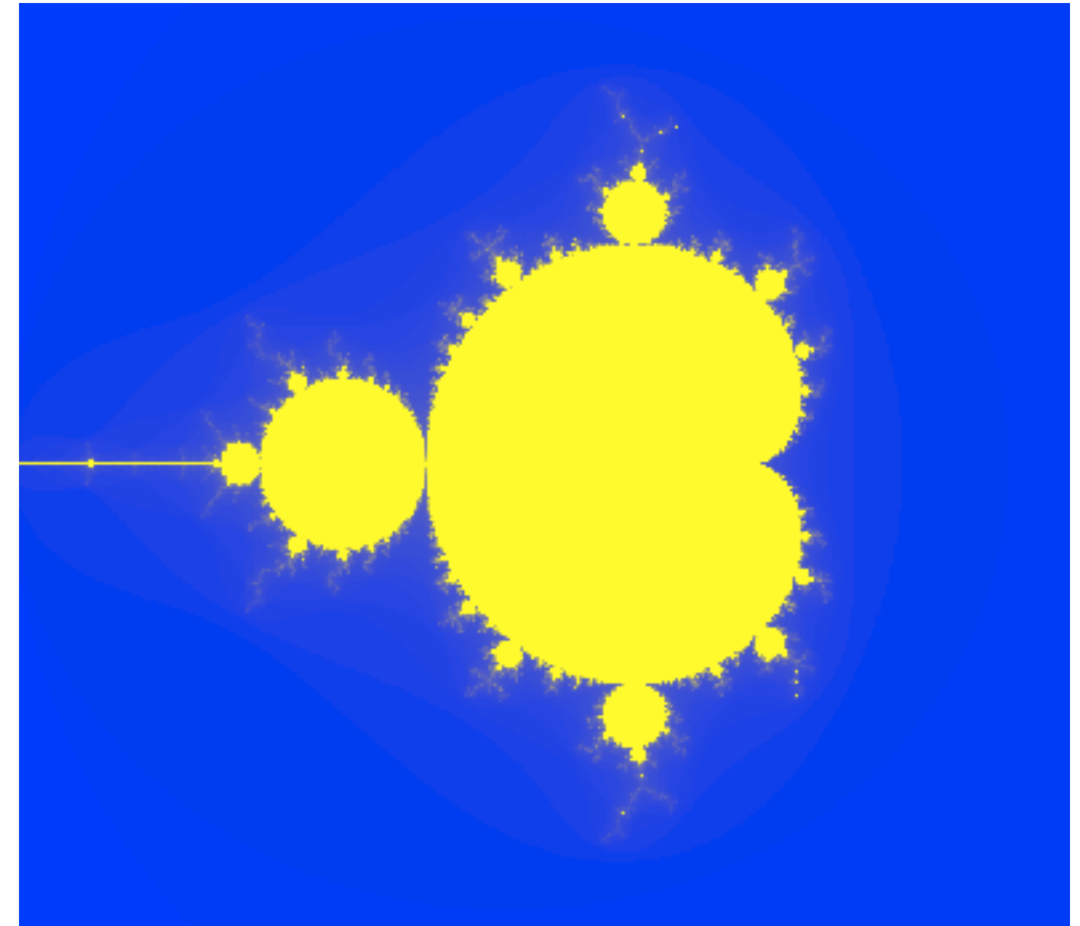
```
/*----------------------------------------------------------------------------*/
float **make_mandel_map_omp(double xmin, double xmax, double ymin, double ymax, int npix, int maxiter)
{
  float **mymap=fmatrix(npix,npix);
#pragma omp parallel for shared(mymap,npix,xmin,xmax,ymin,ymax,maxiter) default(none)
  for (int i=0;i<npix;i++)
    for (int j=0;j<npix;j++) {
      double x=((double)i)/((double)npix)*(xmax-xmin)+xmin;
      double y=((double)j)/((double)npix)*(ymax-ymin)+ymin;
      double complex a=x+I*y;
      mymap[i][j]=how_many_iter(a,maxiter);
    }
  return mymap;
}

/*----------------------------------------------------------------------------*/
```

Code to evaluate a point :

```
int how_many_iter(double complex a, int maxiter)
     /*At complex point a, figure out how many iterations it takes
       to know the point is outside of the Mandlebrot set.  Or, return maxiter
       if (as far as we know) the point is inside the set.*/
{
  int i=0;
  double complex b=a;
  double r;
  while (i<maxiter) {
    r=cimag(b)*cimag(b)+creal(b)*creal(b);
    if (r>4)
      return i;
    b=b*b+a;
    i++;
  }
  return maxiter;
}
```

Note that there is some support for complex variables in C! Not fully standard, but usually OK.

Mandlebrot Performance



```
[sievers@tpb5 omp-intro]$ setenv OMP_NUM_THREADS 1
[sievers@tpb5 omp-intro]$ ./mandel
Tock registers     2.0407e+00 seconds.
```

```
[sievers@tpb5 omp-intro]$ setenv OMP_NUM_THREADS 8
[sievers@tpb5 omp-intro]$ ./mandel
Tock registers     8.9561e-01 seconds.
```

Going from 1 core to 8 only bought a factor of 2.28, or a 28.5% efficiency. Not great.

Try Static Scheduling

Code:

```
/*------------------------------------------------------------------------------*/
float **make_mandel_map_static(double xmin, double xmax, double ymin, double ymax, int npix, int maxiter)
{
  float **mymap=fmatrix(npix,npix);
#pragma omp parallel for shared(mymap,npix,xmin,xmax,ymin,ymax,maxiter) default(none)  schedule(static,4)
  for (int i=0;i<npix;i++)
    for (int j=0;j<npix;j++) {
      double x=((double)i)/((double)npix)*(xmax-xmin)+xmin;
      double y=((double)j)/((double)npix)*(ymax-ymin)+ymin;
      double complex a=x+I*y;
      mymap[i][j]=how_many_iter(a,maxiter);
    }
  return mymap;
}

/*------------------------------------------------------------------------------*/
```

Output:

```
[sievers@tpb5 omp-intro]$ setenv OMP_NUM_THREADS 1
[sievers@tpb5 omp-intro]$ ./mandel
Tock registers     2.0407e+00 seconds.

[sievers@tpb5 omp-intro]$ setenv OMP_NUM_THREADS 8
[sievers@tpb5 omp-intro]$ ./mandel
Tock registers     4.6178e-01 seconds.
```

Static scheduling with chunk size of 4 bought us a factor of 2 in performance. At 55% of peak now.

Try Dynamic Scheduling

Code:

```
/*-----------------------------------------------------------------------------*/
float **make_mandel_map_dynamic(double xmin, double xmax, double ymin, double ymax, int npix, int maxiter)
{
  float **mymap=fmatrix(npix,npix);
#pragma omp parallel for shared(mymap,npix,xmin,xmax,ymin,ymax,maxiter) default(none)  schedule(dynamic,4)
  for (int i=0;i<npix;i++)
    for (int j=0;j<npix;j++) {
      double x=((double)i)/((double)npix)*(xmax-xmin)+xmin;
      double y=((double)j)/((double)npix)*(ymax-ymin)+ymin;
      double complex a=x+I*y;
      mymap[i][j]=how_many_iter(a,maxiter);
    }
  return mymap;
}

/*-----------------------------------------------------------------------------*/
```

Output:

```
[sievers@tpb5 omp-intro]$ setenv OMP_NUM_THREADS 1
[sievers@tpb5 omp-intro]$ ./mandel
Tock registers     2.0458e+00 seconds.

[sievers@tpb5 omp-intro]$ setenv OMP_NUM_THREADS 8
[sievers@tpb5 omp-intro]$ ./mandel
Tock registers     2.6587e-01 seconds.
```

Dynamic got us to 96% of peak.

Summary So Far:
Directives We Have Met

Start a parallel region:
#pragma omp parallel shared() private() default()

Parallelize a loop:
#pragma omp for schedule(static/dynamic, chunk)

Mark off a region only one thread can be in at a time:
#pragma omp critical

Safely update a single memory location:
#pragma omp atomic

In a parallel region, have only one process do something:
#pragma omp single

A Few More Directives
(Less commonly used)

- #pragma omp ordered - execute the loop in the order it would have run serially. Useful if you want ordered output in a parallel region. Never useful for performance.
- #pragma omp master - a block that only the master thread (thread 0) executes. Usually, #pragma omp single is better.
- #pragma omp sections - execute a list of things in parallel. In OpenMP 3, task directive (later in lecture) is more powerful.

Summary So Far II:
Style Points

If a variable is a private temporary variable inside a parallel region, try declaring it inside the region. Makes the parallel region much easier to specify, and can prevent bugs.

OpenMP supports reduction and initialization clauses. These are never necessary to use, but are very convenient and can streamline code.

You have seen how to find out how many threads exist, etc. However, in none of our examples did we use that info. I suggest that if you think you need to know how many threads you have, you are doing something wrong. Using locally declared variables, and critical regions most likely will do everything you need.

Memory Access

- Processors work on local bits of memory in their cache.
- Cache is small and fast. Main memory is big, but slow.
- There is a large *latency* in getting things from main memory - often hundreds of clock cycles. The fewer *times* we access main memory, the faster we will go.
- Computers bring in chunks of memory at a time. If you access data in contiguous memory chunks, much of it may already be in cache. Always try to do this - serial or parallel.
- C - last index is rapidly varying. Fortran first index.

Memory Access II

- Memory access is important for serial programs, but can become particularly important in OpenMP
- There is typically a limited bandwidth to main memory. If it has to be shared 2, 4, or 8 ways, it becomes especially critical to access it sensibly.
- Note on shared variables in OpenMP: If you aren't changing them, the compiler can copy the shared variable to local cache and no performance hit. *Modifying* shared variables is expensive - we have already seen this with the dot product.

Example - Matrix Multiplication

- Linear algebra a classic example.
- Matrix multiplication: C=A*B, or c[i][j]=∑a[i][k]*b[k][j]
- Different implementations can take 10-100x longer than optimal. Slowness entirely due to memory access.
- The more you do with stuff you've pulled from main memory, the faster you'll run.

Slow Multiplication

```
void matmult_slowest(NType **a, NType **b, NType **c, int n)
{
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++)
    {
      c[i][j]=0;
      for (int k=0;k<n;k++)
        c[i][j]+=a[i][k]*b[k][j];
    }
}
```

```
int main(int argc, char *argv[])
{
  pca_time tt;
  int n=500;

  NType **a=matrix(n,n);
  NType **b=matrix(n,n);
  NType **c=matrix(n,n);

  fill_random_matrix(a,n);
  fill_random_matrix(b,n);

  tick(&tt);
  matmult_slowest(a,b,c,n);
  printf("Time to multiply %d x %d matrices with slow multiplication is %9.4f\n",n,n,tocksile
  printf("Sum of elements is %14.4e\n",matrix_sum(c,n));

}
```

Output:

```
[sievers@tpb5 omp-intro]$ ./matmult_slow
Time to multiply 500 x 500 matrices with slow multiplication is    2.6004
Sum of elements is     3.1275e+07
[sievers@tpb5 omp-intro]$
```

Slow Matrix Multiplication

- What happened? For every element in C, we had to pull a fast direction from A, but a slow direction from B.
- Could change the order of the loops, making B fast, but then A would be slow.
- We pulled a slow vector for each element in C, for a total of n^2 slow column pulls.
- Could make the transpose of B, then we would always pull from the fast columns. Only have to do n slow pulls this way.
- Drawback: must make a copy of B. If B is large, can take lots of memory.

Transpose Multiplication

Code:

```
void matmult_transpose(NType **a, NType **b, NType **c, int n)
{
  NType **bt=matrix(n,n);
  matrix_transpose(b,bt,n);
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++)  {
      c[i][j]=0;
      for (int k=0;k<n;k++)
        c[i][j]+=a[i][k]*bt[j][k];
    }
  free_matrix(bt);
}
/*-----------------------------------------------------------------------------*/
NType matrix_sum(NType **a, int n)
{
  NType sum=0;
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++)
      sum+=a[i][j];
  return sum;
}
```

Output:

```
[sievers@tpb5 omp-intro]$ ./matmult_transpose
Time to multiply 500 x 500 matrices with transpose multiplication is    1.8632
Sum of elements is     3.1275e+07
[sievers@tpb5 omp-intro]$
```

Nearly 50% faster than slow version

Blocks

- Multiplication was still kind of slow. Why?
- For every column of C we calculate, we have to process *all* of B, for a total of *n* times. That's a lot of memory throughput.
- Recall $c_{ij}=\sum a_{ik}*b_{kj}$. Nowhere have we said that c_{ij}, a_{jk}, and b_{kj} are scalars. They could be blocks of the matrices. If we treat them as blocks, then we'll have to go to main memory less often.
- Say blocks are 20x20. Then I have to pull all of B each time I process a column of *blocks*. Or a total of *n*/20 times. Much less stress on system memory.

Block Multiplication

```
/*--------------------------------------------------------------------------------*/
void matmult_block(NType **a, NType **b, NType **c, int n, int bs)
{

  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++)
      c[i][j]=0;

  int nb=n/bs;
  assert(nb*bs==n);  /*fail if we don't have an exact multiple of the block size*/


  NType **myblock_1=matrix(bs,bs);
  NType **myblock_2=matrix(bs,bs);
  NType **myblock_3=matrix(bs,bs);

  for (int ib=0;ib<nb;ib++)
    for (int jb=0;jb<nb;jb++)
      for (int kb=0;kb<nb;kb++) {
        int ii=ib*bs;
        int jj=jb*bs;
        int kk=kb*bs;
        // Pull the blocks from A and B out
        for (int i=0;i<bs;i++)
          for (int j=0;j<bs;j++) {
            myblock_1[i][j]=a[i+ii][j+kk];
            myblock_2[i][j]=b[j+kk][i+jj];
          }
        //Do the block multiplication
        for (int i=0;i<bs;i++)
          for (int j=0;j<bs;j++) {
            myblock_3[i][j]=0;
            for (int k=0;k<bs;k++)
              myblock_3[i][j]+=myblock_1[i][k]*myblock_2[j][k];
          }
        //Accumulate the product into c
        for (int i=0;i<bs;i++)
          for (int j=0;j<bs;j++)
            c[i+ii][j+jj]+=myblock_3[i][j];
      }
  free_matrix(myblock_1);
  free_matrix(myblock_2);
  free_matrix(myblock_3);
}
```

Output:

```
[sievers@tpb5 omp-intro]$ ./matmult_block
Time to multiply 500 x 500 matrices with block multiplication is    1.9331
Sum of elements is     3.1275e+07
[sievers@tpb5 omp-intro]$
```

Same time as transpose, but no matrix copy and less stress on system memory.

Blocks Debrief

- Well, managed to do better in memory, calculation time was still the same.
- You may gather that writing a fast, parallel matrix multiplier isn't easy. You are right.
- People have spent a long time optimizing matrix multiplication, and gotten to 80-90% of theoretical max, using block-based algorithms (look up Goto BLAS).
- Important corollary: Think you need to code something? Don't! See if someone else has done it. For core routines, they have, and better than you will *ever* do it.
- For the same problem, Goto runs in 0.044 seconds - 40x faster.

Make sure serial performance is good before worrying about parallel!

Conditional OpenMP

- There is *always* overhead associated with starting threads, splitting work, etc. Also, some jobs parallelize better than others.
- Sometimes, overhead takes longer than 1 thread would need to do a job - e.g. very small matrix multiplies.
- OpenMP supports conditional parallelization. Add if(condition) to parallel region beginning. So, for small tasks, overhead low, while large tasks remain parallel.

Conditional OpenMP in Action

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

int main(int argc, char *argv[])
{
  int n=atoi(argv[1]);
#pragma omp parallel if (n>10)
#pragma omp single
  printf("have %d threads with n=%d\n",omp_get_num_threads(),n);
}
```

```
gpc-f101n084-$ ./conditional_if 12
have 8 threads with n=12
gpc-f101n084-$ ./conditional_if 9
have 1 threads with n=9
gpc-f101n084-$
```

First, pull an integer from the command line. Check to see if it's bigger than a number (in this case, 10). If so, start a parallel region. Otherwise, execute serially.

Controlling # of Threads

- Sometimes you might want more or fewer threads. May even want to change while running.
- Example - IBM P6 cluster. Matrix multiply runs fast with twice as many program threads as physical cores (hyperthreading). However, matrix factorizations run slower with more threads.
- omp_set_num_threads(int) sets or changes the number of threads during runtime.

omp_set_num_threads() in action

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

int main(int argc, char *argv[])
{
  //find how many physical cores we have available.
  //this is an openmp library routine.
  int max_threads=omp_get_num_procs();
  int n=atoi(argv[1]);
  //set the number of threads equal to the input value,
  //assuming it's let than max_threads
  if (n<max_threads)
    omp_set_num_threads(n);
  else
    omp_set_num_threads(max_threads);
#pragma omp parallel
#pragma omp single
  printf("Running  with %d threads for n=%d\n",omp_get_num_threads(),n);

  //now set to our guess at the max.
  omp_set_num_threads(max_threads);
#pragma omp parallel
#pragma omp single
  printf("Finished with %d threads for n=%d\n",omp_get_num_threads(),n);
}
```

We have changed the # of threads during the program. We could always change the number later on in the same code, if we so desired. Note the use of omp_get_num_procs(), a library call to detect the physical number of available processors.

```
gpc-f101n084-$ ./set_num_threads 1
Running  with 1 threads for n=1
Finished with 8 threads for n=1
gpc-f101n084-$ ./set_num_threads 4
Running  with 4 threads for n=4
Finished with 8 threads for n=4
gpc-f101n084-$ ./set_num_threads 8
Running  with 8 threads for n=8
Finished with 8 threads for n=8
gpc-f101n084-$ ./set_num_threads 1000
Running  with 8 threads for n=1000
Finished with 8 threads for n=1000
gpc-f101n084-$
```

Profiling: gprof

- You should always know where your program spends its time working.
- One way - gprof. gprof uses statistical sampling - every so often, it asks where it is in code.
- Code must be compiled with appropriate flags: -g (debug) -pg (profile).
- When run, code writes to a file by default called *gmon.out.*
- Output analyzed later by calling "gprof a.out" (if a.out is executable). That will analyze stuff in gmon.out.
- gprof will tell you which routines (or even which lines) used what fraction of the codes run time.

Code We'll Profile - fast_slow_loops.c

```c
/*==============================================================================
int main(int argc, char *argv[])
{
  long n=4096;
  double **m1=allocate_matrix(n,n);
  double **m2=allocate_matrix(n,n);
  double **msum=allocate_matrix(n,n);

  fill_matrices(n,m1,m2);
  int which_fun=1;
  if (argc>1)
    which_fun=atoi(argv[1]);
  assert(which_fun>=1);
  assert(which_fun<=3);  //only have functions one through three defined.

  switch(which_fun) {
  case 1:
    loop_one(n,m1,m2,msum);
    break;

  case 2:
    loop_two(n,m1,m2,msum);
    break;

  case 3:
    loop_three(n,m1,m2,msum);
    break;
  default:
    printf("Woops.  Unrecognized function type, shouldn't have gotten here.\n");
    break;
  }
}
```

Three different ways of element-wise matrix multiplication. Different access patterns will affect run-time. Note switch/case statements.

```c
/*---------------------------------------------------------------------------*/
void loop_one(int n, double **m1, double **m2, double **msum)
{
  double t1,t2;
  t1=omp_get_wtime();
  for (int i=0;i<n;i++)
    for (int j=0;j<n;j++)
      msum[i][j]=m1[i][j]+m2[i][j];
  t2=omp_get_wtime();
  printf("Loop one took %12.3e seconds.\n",t2-t1);
}

/*---------------------------------------------------------------------------*/
void loop_two(int n, double **m1, double **m2, double **msum)
{
  double t1,t2;

  t1=omp_get_wtime();
  for (int j=0;j<n;j++)
    for (int i=0;i<n;i++)
      msum[i][j]=m1[i][j]+m2[i][j];
  t2=omp_get_wtime();
  printf("Loop two took %12.3e seconds.\n",t2-t1);
}
/*---------------------------------------------------------------------------*/
void loop_three(int n, double **m1, double **m2, double **msum)
{
  double t1,t2;

  double *mm1=m1[0];
  double *mm2=m2[0];
  double *mmsum=msum[0];
  long nn=n*n;
  t1=omp_get_wtime();
  for ( long i=0;i<nn;i++)
    mmsum[i]=mm1[i]+mm2[i];
  t2=omp_get_wtime();
  printf("Loop three took %12.3e seconds.\n",t2-t1);
}
/*---------------------------------------------------------------------------*/
```

Profiling, cont'd

```
[sievers@tpb4 ~]$ gcc -g -pg -std=c99 -o fast_slow_loops fast_slow_loops.c -lgomp
[sievers@tpb4 ~]$ ./fast_slow_loops 1
Loop one took    3.183e-01 seconds.
[sievers@tpb4 ~]$ ./fast_slow_loops 2
Loop two took    4.412e+00 seconds.
[sievers@tpb4 ~]$ ./fast_slow_loops 3
Loop three took    2.611e-01 seconds.
```

Note - wrong memory access again kills us by nearly factor of 15.

gprof: -b=brief. It reports how much time, what percent were spent in different routines.

Because statistical, should not expect identical values run-to-run.

Look - more time spent in setting up matrices, not in doing work. Tells us what to fix.

```
[sievers@tpb4 ~]$ ./fast_slow_loops 1
Loop one took    3.179e-01 seconds.
[sievers@tpb4 ~]$ gprof -b fast_slow_loops
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 56.30      0.37     0.37        1   371.55   371.55  fill_matrices
 44.12      0.66     0.29        1   291.21   291.21  loop_one
  0.00      0.66     0.00        3     0.00     0.00  allocate_matrix


                        Call graph


granularity: each sample hit covers 2 byte(s) for 1.51% of 0.66 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.66                 main [1]
                0.37    0.00       1/1           fill_matrices [2]
                0.29    0.00       1/1           loop_one [3]
                0.00    0.00       3/3           allocate_matrix [4]
-----------------------------------------------
                0.37    0.00       1/1           main [1]
[2]     56.1    0.37    0.00       1         fill_matrices [2]
-----------------------------------------------
                0.29    0.00       1/1           main [1]
[3]     43.9    0.29    0.00       1         loop_one [3]
-----------------------------------------------
                0.00    0.00       3/3           main [1]
[4]      0.0    0.00    0.00       3         allocate_matrix [4]
-----------------------------------------------


Index by function name

   [4] allocate_matrix         [2] fill_matrices          [3] loop_one
[sievers@tpb4 ~]$
```

Tasks

- OpenMP 3.0 supports the *#pragma omp task* directive.
- A task is a job assigned to a thread. Powerful way of parallelizing non-loop problems.
- Tasks should help omp/mpi hybrid codes - one task can do communications, rest of threads keep working.
- Like all omp, tasks must be called from parallel region.
- Raises complication of nested parallelism (what happens if a parallel loop called from parallel loop?).

Tasks: test_task.c

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

int main(int argc, char *argv[])
{
  #pragma omp parallel
  #pragma omp single
  {
    printf("hello!\n");
    #pragma omp task
    {
      printf("hello 1 from %d.\n",omp_get_thread_num());
      double s=0;
      for (int i=0;i<40960;i++)
        for (int j=0;j<40960;j++)
          s+=i*j;
      printf("s is %14.4e\n",s);
    }

    #pragma omp task
    printf("hello 2 from %d.\n",omp_get_thread_num());

  }

  return 0;
}
```

Often want to start tasks from as if from serial region. Must be in parallel for tasks to spawn, so #pragma omp parallel followed by #pragma omp single very useful.

What would happen w/out #pragma omp single?

```
Macintosh-270:omp-intro sievers$ ./test_task
hello!
hello 1 from 0.
hello 2 from 1.
s is     7.0365e+17
```

Beauty of Tasks

- Some problems naturally fit into tasks that are otherwise hard to parallelize.
- Example (from standard): parallel tree processing.
- Each node has left, right pointers, process each sub-pointer with a task.
- Look how short the parallel tree is!

How would you do this problem without tasks?

```
struct node {
  struct node *left;
  struct node *right;
};
extern void process(struct node *);
void traverse( struct node *p ) {
  if (p->left)
#pragma omp task   // p is firstprivate by default
      traverse(p->left);
  if (p->right)
#pragma omp task    // p is firstprivate by default
      traverse(p->right);
  process(p);
}
```

Homework

- Homework will walk you through parallelizing the matrix multiplications, make you use gprof and optimization flags, and highlight importance of good memory access.
- Look in pca/src/openmp2 directory for Homework.txt
- Do what it says. You can directly edit the codes there, and add your answers to the Homework.txt file.