cse5441 - parallel computing

Open MP

OpenMP -vs- pthreads

- both use global shared memory paradigm
- both require rigorous analysis of variable scope and sharing
- both methods operate at the "thread" level
- pthreads design objective: to give programmer maximal control
- OpenMP design objective: to provide semi-automatic parallelization without change to existing code
- performance is highly variable, and implementation specific

context:
global variables – with respect to a parallel region

what is OpenMP?

essentially:

- a set of compiler directives
- a set of library routines
- aims to implement best practices from 20 years of threads experience

available for C, C++ and Fortran

OpenMP core syntax

- **Most of the constructs in OpenMP are compiler directives:**

  #pragma omp *construct* *[clause [clause]…]*

- **Example**

  #pragma omp parallel num_threads(4)

- **Function prototypes and types in the file: #include <omp.h>**

- **Most OpenMP constructs apply to a "structured block".**

- **Structured block: a block of one or more statements surrounded by "{ }", with one point of entry at the top and one point of exit at the bottom.**

  (It's OK to have an exit() within the structured block . . .
  . . . but all will exit.)

hello world ☺

```
#include  <omp.h>
void main()
{
   #pragma  omp  parallel
   {
      int ID = 0;
      printf(“ hello(%d) ”, ID);
      printf(“ world(%d) \n”, ID);
   }
}
```

compile openMP programs on stdlinux with -fopenmp

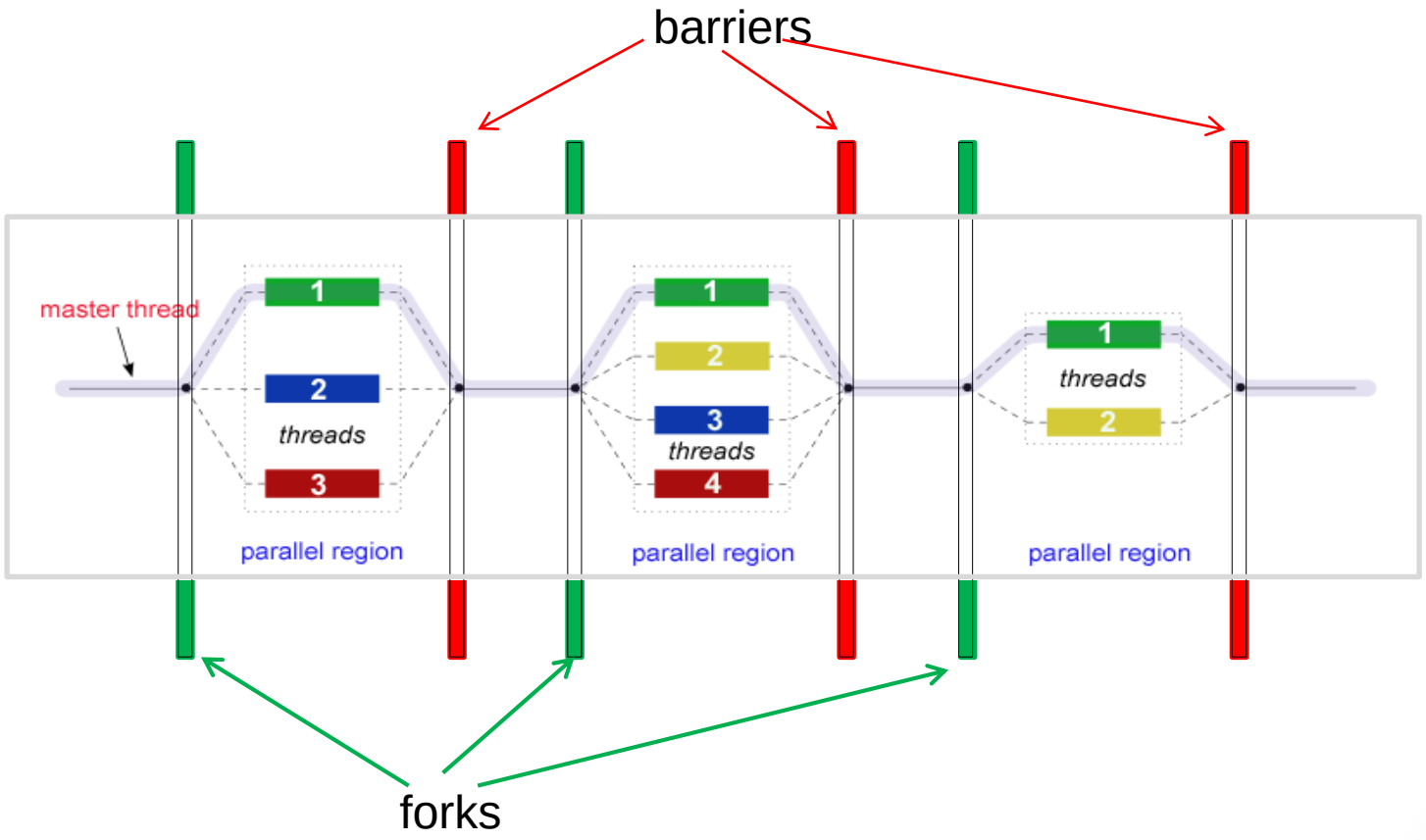
hello world ☺

```
#include  <omp.h>
void main()
{
   #pragma  omp  parallel
   {
      int ID = omp_get_thread_num();
      printf(“ hello(%d) ”, ID);
      printf(“ world(%d) \n”, ID);
   }
}
```

thread interaction

- multi-threaded global shared memory model
- thread communication by shared variables (shared memory)
- unintended sharing causes race conditions
  - inconsistent and unpredictable output
  - avoided with synchronization and serial segments

fork – join parallelism



note: parallel regions “can” be nested:
https://docs.oracle.com/cd/E19059-01/stud.10/819-0501/2_nested.html

https://computing.llnl.gov/tutorials/openMP/images/fork_join2.gif

parallel region data distribution

"workload distribution"

cyclic distribution

```
double   A[1000];

omp_set_num_threads(4);
#pragma omp parallel
{
   int t_id = omp_get_thread_num();
   for (int i = t_id; i < 1000; i += omp_num_threads())
   {
      A[i] = 0;
   }
}
```

block distribution

```
double   A[1000];

omp_set_num_threads(4);
#pragma omp parallel
{
   int t_id     = omp_get_thread_num();
   int b_size = 1000 / omp_num_threads();
   for (int i = t_id * b_size; i < (t_id+1) * b_size; i ++)
   {
      A[i] = 0;
   }
}
```

we will be
finessing
boundary
conditions . . .

OpenMP requesting threads

```
double A[1000];

#pragma omp parallel num_threads(my_choice)
{
   int t_id = omp_get_thread_num();
   for (int i = t_id; i < 1000; i += omp_num_threads())
   {
      A[i] = 0;
   }
}
```

implicit barrier

```
{
    each thread will
    execute the code
    within the block
}
```

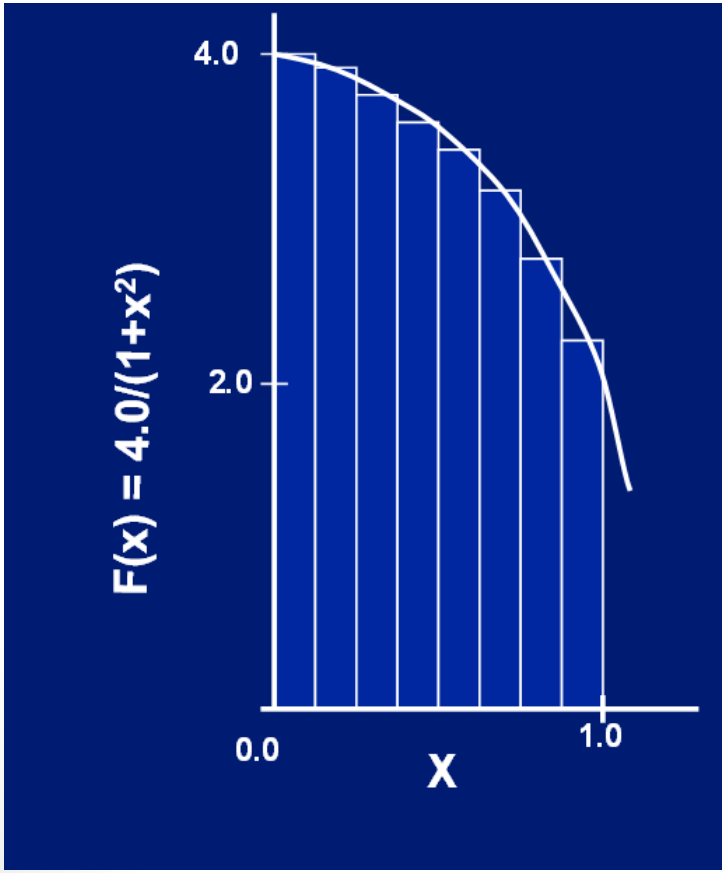
Tim Mattson & Larry Meadows

- Principal Engineers for Intel
- active part of OpenMP architecture review board
- provided content on many slides
- http://openmp.org/mp-documents/omp-hands-on-SC08.pdf
- (also on Carmen …)

the software professional must design for
a specific OpenMP version

numerical integration

background



Mathematically:

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

Which can be approximated by:

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

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

https://software.intel.com/sites/default/files/m/d/4/1/d/8/1252a.gif

serial pi

example 1

```
int        num_steps = 100000;
double     step;

void 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;
}
```

http://1.bp.blogspot.com/-M3m7__wJBQU/T_MbywTCdmI/AAAAAAAAB1Q/YnWe5odlWdg/s1600/cereal+and+pie.png

SPMD pattern with OpenMP

- **Single Program Multiple Data**

- each thread runs same program
  - selection of data, or braching conditions, based on thread id

- in OpenMP implementations:
  - parallelize loops
  - query thread_id and num_threads
  - partition input data among threads

"workload distribution"

it's your turn … parallel pi

exercise 1

using:

- #pragma omp parallel
- (void) omp_set_num_threads(int)
- (int) omp_get_num_threads()
- (int) omp_get_thread_num()

create a parallel version of pi
by parallelizing the *for* loop

pay close attention to variables which will be shared

parallel assignment race condition

sum = sum + 4.0/(1.0+x*x);

```
load_register     1, @sum
set_register      2, 4.0
set_register      3, 1.0
load_register     4, @x
multiply_into     5, 4, 4
add_into          4, 3, 5
divide_into       3, 2, 4
add_into          2, 1, 3
store             2, @sum
```

parallel pi

exercise 1

```
#include    <omp.h>
int         num_steps = 100000;
double      step;
#define     NUM_THREADS       2

void main ()
{
int         i,  nthreads;
double      pi = 0.0,  sum[NUM_THREADS];

    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
```

another way to request ^ a number of 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;

        sum[ id ] = 0.0;
        for ( i = id; i < num_steps;  i += nthrds)
        {
            x = (i+0.5)*step;
            sum[id] += 4.0/(1.0+x*x);
        }
    }
    for( i = 0; i < nthreads; i++)
    {
        pi += sum[i] * step;
    }
}
```

<- example promotion

^ partition method cyclic distribution

<- implicit barrier

this loop is serial ->

a “reduction”

parallel updates – cache considerations

```
sum[id] += 4.0/(1.0+x*x);

sum[id] = sum[id] + 4.0/(1.0+x*x);
```

false sharing

when updating:

- consider multi-processors with local caches
- consecutive elements assigned to different threads will likely go to other caches
- write will update local cache, but all other local caches would be marked invalid

cyclic -vs- block distribution

```
double A[1000];

omp_set_num_threads(4);
#pragma omp parallel
{
   int t_id = omp_get_thread_num();
   for (int i = t_id; i < 1000; i += omp_num_threads())
   {
       sum[id]  +=  4.0/(1.0+x*x);
   }
}
```

```
double A[1000];

omp_set_num_threads(4);
#pragma omp parallel
{
   int t_id     = omp_get_thread_num();
   int b_size = 1000 / omp_num_threads();
   for (int i = (t_id-1) * b_size; i < t_id * b_size; i ++)
   {
       sum[id]  +=  4.0/(1.0+x*x);
   }
}
```

critical code regions

```
#pragma omp parallel
{
float      B;
int        i, id, nthrds;

   id       = omp_get_thread_num();
   nthrds = omp_get_num_threads();
   for( i = id;  i < MAX;  i + nthrds)
   {
       B = big_job(i);
       #pragma omp critical
       {
           consume (B, res);
       }
   }
}
```

it's your turn … parallel pi

exercise 2

using:

- #pragma omp parallel
- (void) omp_set_num_threads(int)
- (int) omp_get_num_threads()
- (int) omp_get_thread_num()
- #pragma omp critical
- block workload distribution

create a parallel version of pi which:

- does not exhibit false sharing
- does not use an array of sums

parallel pi - no false sharing

exercise 2

```
int             num_steps = 100000;
double          step;
#define         NUM_THREADS         2

void main ()
{
int             i,  nthreads;
double          pi = 0.0;

    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS);
```

no array, no false sharing ->

```
    #pragma omp parallel
    {
    int         i, id,nthrds;
    double      x, sum;
```

<- sum is now local

```
        id      = omp_get_thread_num();
        nthrds = omp_get_num_threads();
        if (id == 0) nthreads = nthrds;

        sum  = 0.0;
        for ( i = id; i < num_steps;  i += nthrds)
        {
            x = (i+0.5)*step;
            sum += 4.0/(1.0+x*x);
        }
        #pragma  omp  critical
        {
            pi  +=  sum * step;
        }
    }
}
```

^ each thread adds its partial sum one thread at a time

named critical code regions

goto label1;

branching into critical region prohibited

```
#pragma omp parallel
{
float      B;
int        i, id, nthrds;

   id       = omp_get_thread_num();
   nthrds = omp_get_num_threads();
   for( i = id;  i < MAX;  i + nthrds)
   {
       B = big_job(i);
       #pragma  omp  critical  consume_it
       {
           consume_1 (B, res);
label 1:   consume_2(B, upd);
           consume_3(res, upd);
       }
   }
}
```

atomic code regions

```
int         x;

#pragma omp parallel
{
int         i, id, nthrds, y;

    id      = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    for( i = id;  i < MAX;  i + nthrds)
    {
        y = something(fun);
        /* Protect against race conditions */
        #pragma omp atomic
        {
            x += y;
        }
     }
}
```

list of OMP atomic supported operations:
https://www.ibm.com/support/knowledgecenter/SSGH2K_13.1.2/com.ibm.xlc1312.aix.doc/compiler_ref/prag_omp_atomic.html

cse5441 - parallel computing

Open MP

OpenMP loop worksharing

```
#pragma omp parallel
{
   // id      = omp_get_thread_num();
   // nthrds = omp_get_num_threads();
   // for( i = id;  i < MAX;  i + nthrds)
   #pragma omp for
   for( i = 0;  i < MAX;  i++)
   {
       B = big_job(i);
   }
}
```

combined parallel / worksharing construct

```
#pragma omp parallel for
{
   for( i = 0; i < MAX; i++)
   {
       B = big_job(i);
   }
}
```

OpenMP reduction

How do we handle this case?:

```
double    avg = 0.0;
double    A[SIZE];
#pragma  omp  parallel  for
for (int i = 0; i < SIZE; i++)
{
    avg += A[ i ];
}

avg = avg / SIZE;
```

We can use the clause:

reduction(*op* : *list*)

OpenMP reduction

```
double    avg = 0.0;
double    A[SIZE];
#pragma  omp  parallel  for  reduction( + : avg )
for (int i = 0; i < SIZE; i++ )
{
    avg += A[ i ];
}

avg = avg / SIZE;
```

parallel region

- each variable in *list* is made thread local
- each *list* variable is initialized appropriate to *op*
- local *list* variables combined according to *op* at end of loop
- combined value exported to original “global” variable

it's your turn … parallel pi

exercise 3

- **Example**

  #pragma omp parallel num_threads(4)

using:

- #pragma omp parallel for
- reduction (*op* : *list*)
- num_threads (int)
- 8 threads

create a parallel version of pi which is as close to serial pi as possible

parallel pi

exercise 3

serial pi

```
int         num_steps = 100000;
double      step;


void 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;
}
```

parallel pi

```
int         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) \
                num_threads(8)
    for ( i = 0; i < num_steps;  i++)
    {
        x = (i+0.5)*step;
        sum += 4.0/(1.0+x*x);
    }
    pi  +=  sum * step;
}
```

line continuation v

parallelize, and reduce into sum →

manage number of threads →

cse5441 - parallel computing

Open MP

additional
topics

setting num_threads at run time

```
$g++ -fopenmp my_omp_program.cc
$ a.out 4 <my_data_file

. . .

int main( int argc, char* argv[ ] )
{
    nt = atoi( argv[ 1 ] );

. . .

#pragma omp parallel num_threads(nt)
{

. . .
```

<- pragma supports variables

OpenMP synchronization: barriers

```
#pragma omp parallel private(id)
{
   id=omp_get_thread_num();
   A[id] = big_calc1(id);
   #pragma omp barrier

   #pragma omp for
   for(i=0; i<N; i++)
   {
      C[i]=big_calc3(i,A);
   }

   #pragma omp for nowait
   for(i=0; i<N; i++)
   {
      B[i]=big_calc2(C, i);
   }
   A[id] = big_calc4(id);
}
```

explicit barrier →

implicit barrier at end of parallel region →

no barrier! **nowait** cancels barrier creation →

NOTE:
no implicit barriers at the beginning of pragmas

what could happen following nowait clause?

OpenMP synchronization: master

```
#pragma omp parallel
{
    do_many_things();

    #pragma omp master
    {
        reset_boundaries();
    }

    do_many_other_things();
}
```

multiple threads of control →

only **master thread** → executes this region

← no implicit barrier here

multiple threads of control →

OpenMP synchronization: single

```
#pragma omp parallel
{
    do_many_things();

    #pragma omp single
    {
        reset_boundaries();
    }

    do_many_other_things();
}
```

multiple threads of control →

multiple threads of control →

a single thread is chosen → to execute this region

← implicit barrier

OpenMP synchronization: simple mutex locks

```
omp_lock_t   lck;
omp_init_lock(&lck);

#pragma omp parallel
{
   do_many_things();
   omp_set_lock(&lck);
      cout <<  “here i am alone\n”;
   omp_unset_lock(&lck);
   do_many_other_things ();
}
omp_destroy_lock(&lck);
```

multiple threads of control →

wait here for your turn … →

multiple threads of control →

requested number of threads: I really mean it …

```
void main()
{
int  nthreads;

    omp_set_dynamic(0);
    omp_set_num_threads( omp_get_num_procs() );
    do_some_things();

    #pragma omp parallel
    {
        int id = omp_get_thread_num();
        #pragma  omp  single
        {
            nthreads = omp_get_num_threads();
            if ( nthreads  !=  what_you_want )
            {
                cout << “arrrgh!” << endl;
                exit(fail);
            }
        do_many_more_things();
```

NOTE: older versions use omp_num_procs()

OpenMP data environment

remember, young Jedi:

- global variables (meaning declared outside the scope of a parallel region) are **shared** among threads unless explicitly made private
- automatic variables declared within parallel region scope are **private**
- stack variable declared in functions called from within a parallel region are **private**

data sharing attributes

#pragma omp parallel private(x)

- each thread receives its own **uninitialized** variable x
- the variable x falls out-of-scope after the parallel region
- a global variable with the same name is unaffected **(3.0 and later)**

#pragma omp parallel firstprivate(x)

- x must be a global-scope variable
- each thread receives a **by-value copy** of x
- the local x's fall out-of-scope after the parallel region
- the base global variable with the same name is unaffected

(3.0 and later)

#pragma omp parallel lastprivate(x)

- x must be a global-scope variable
- each thread receives a **by-value copy** of x
- the base global variable of the same name is set to the last value of x for the last thread to finish

OpenMP sections

```
#pragma omp parallel
{
    . . .

    #pragma omp sections
    {
    #pragma omp section
        X_calculation();
    #pragma omp section
        y_calculation();
    #pragma omp section
        z_calculation();
    }
    . . .

}
```

multiple threads of control → each section assigned to a different thread

by default:
extra threads are idled

with *nowait*:
extra threads continue ahead

OpenMP schedule clause

The schedule clause determines how loop iterators are mapped onto threads:

#pragma omp parallel for schedule(static [, chunk])

- chunking computed at compile time
- fixed-sized chucks assigned (alternating) to num_threads
- typical default is: chunk = iterations / num_threads
- set chunk = 1 for cyclic distribution

#pragma omp parallel for schedule(dynamic [, chunk])

- run-time scheduling (with associated overhead)
- each thread grabs “chunk” iterations off queue until all iterations have been scheduled
- good load-balancing for uneven workloads

OpenMP schedule clause

The schedule clause determines how loop iterators are mapped onto threads:

#pragma omp parallel for schedule(guided[, chunk])

- threads dynamically grab blocks of iterations
- chunk size starts relatively large, to get all threads busy with good amortization of overhead
- subsequently, chunk size is reduced to produce good workload balance
- high overhead, best for very uneven workload loops

#pragma omp parallel for schedule(runtime)

- schedule and chunk size taken from environment variable or from runtime library routines

OpenMP key env variables

unix environment variables of note:

OMP_NUM_THREADS
- set to the desired default number of threads
- still just a request and system can override

OMP_SCHEDULE
- “schedule [, chunk]”

cse5441 - parallel computing

Open MP