In-Class “Guerrilla” Development of MPI Examples

Steve Lantz
Computing and Information Science 4205
www.cac.cornell.edu/~slantz

Guerrilla Development?

- guer ril´la (*n.*) - A member of an irregular, usually indigenous military or paramilitary unit operating in small bands in occupied territory to harass and undermine the enemy, as by surprise raids.
  The American Heritage Dictionary of the English Language, Fourth Edition

- “Guerrilla Development - Software development in an environment unsupportive of the effort. The adversity can take the form of active management encumbrances (ridiculous budget/time/staffing constraints, and the like).... Guerrilla Developers find a way to win, despite the odds, and work to prevent their project from becoming a Death March.”
  http://www.artima.com/weblogs/viewpost.jsp?thread=5414

- “The Guerrilla Experience means total immersion in social coding. Multiple instructors keep you engaged throughout the entire learning process collaborating, competing, and coding.” – DevelopMentor (a Microsoft partner)

What’s the Best Way to Receive a Message From Any of Several Possible Sources?

- **MPI_Probe and MPI_Recv**
- **MPI_Irecv and MPI_Waitany**
- **MPI_Recv(…MPI_ANY_SOURCE…)**

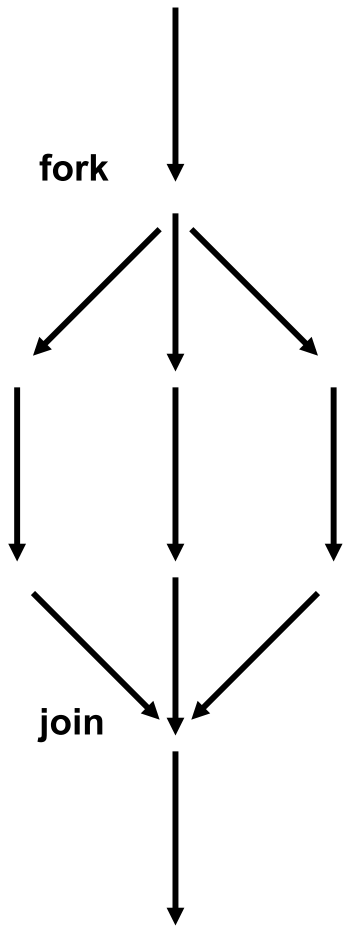
Shared Memory Programming Using Basic OpenMP

What is Shared Memory Programming?

- **Physically: processors in a computer share access to the same RAM**
- **Virtually: threads running on the processors interact with one another via shared variables in the common address space of a single process**
- **Making performance improvements to serial code tends to be easier with multithreading than with message passing parallelism**
  - Message passing usually requires the code/algorithm to be redesigned
  - Multithreading allows incremental parallelism: take it one loop at a time
- **Clusters today are commonly made up of multiple processors per compute node; using OpenMP with MPI is a strategy to improve performance at the two levels of shared and distributed memory**

Fork & Join

- Multi-threaded programming is the most common shared-memory programming methodology.
- A serial code begins execution. The process is the master thread or only executing thread.
- When a parallel portion of the code is reached the Master thread can "fork" more threads to work on it.
- When the parallel portion of the code has completed, the threads "join" again and the master thread continues executing the serial code.



What is OpenMP?

- **Every system has its own threading libraries**
  - Can be quite complicated to program
  - Generally not portable
  - Often optimized to produce the absolute best performance
- **OpenMP has emerged as a standard method for shared-memory programming**
  - Similar to the way MPI has become the standard for distributed-memory programming
  - Codes are portable
  - Performance is usually good enough
- **Consists of: compiler directives, API calls, environment variables**
- **Compiler support**
  - C, C++ & Fortran
  - Microsoft, Intel (icc -openmp), and GNU (gcc -fopenmp)

How Does OpenMP Work, Typically?

- **“for” loops often point to a naturally parallel section of the code**
- **Using compiler directives, we show the compiler where it can automatically parallelize a loop for us**
  - In C & C++ these directives are called “pragmas” (pragmatic information)
  - OpenMP pragmas start with #pragma omp
  - Pragmas go in front of for loops to tell the compiler that they can be parallelized
- **Loops that can be parallelized have some requirements**
  - Run time system must know how many iterations will be executed
  - Loops cannot have logic that allow them to exit early: break, return, exit etc.
  - Loops must have “canonical shape”

```
for (i = start; i   < end;    i++           )
    <=                ++i
                      >         i--
    >=                --i
                      i+=inc
                      i-=inc
                      i=i+inc
                      i=i-inc
```

Program omp_dp1.c – 1 of 2

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

int main(int argc, char **argv )
 {
  int i,workers;
  int *a,*b,*c;
  int mult=256000;
  double start, end;

  workers = omp_get_max_threads();
  printf(“%d parallel workers”,workers);
  a = malloc(workers*mult*sizeof(int));
  b = malloc(workers*mult*sizeof(int));
  c = malloc(workers*mult*sizeof(int));

  start = omp_get_wtime();
```

Program omp_dp1.c – 2 of 2

```
 #pragma omp parallel for
 for (i=0;i<workers*mult;i++)
  {
   a[i]=i;
   b[i]=i;
   c[i]=i;
   a[i] = b[i] + c[i];
  }
 end = omp_get_wtime();
 printf ("Parallel Loop took %.04f seconds\n",end-start);
 start = omp_get_wtime();
 for (i=0;i<workers*mult;i++)
  {
   a[i]=i;
   b[i]=i;
   c[i]=i;
   a[i] = b[i] + c[i];
  }
 end = omp_get_wtime();
 printf ("Serial Loop took %.04f seconds\n",end-start);
 return 0;
}
```

Comparison of icc and gcc on an OpenMP Code

```
-sh-3.1$ icc -openmp -o i_omp_dp1 omp_dp1.c
omp_dp1.c(18): (col. 3) remark: OpenMP DEFINED LOOP WAS
PARALLELIZED.
omp_dp1.c(29): (col. 3) remark: LOOP WAS VECTORIZED.
omp_dp1.c(19): (col. 3) remark: LOOP WAS VECTORIZED.
-sh-3.1$ ./i_omp_dp1
8 parallel workers
Parallel Loop took 0.0096 seconds
Serial Loop took 0.0179 seconds
-sh-3.1$
-sh-3.1$ gcc -fopenmp -o omp_dp1 omp_dp1.c
-sh-3.1$ ./omp_dp1
8 parallel workers
Parallel Loop took 0.0106 seconds
Serial Loop took 0.0403 seconds
-sh-3.1$ gcc -fopenmp -o omp_dp1 -O3 omp_dp1.c
-sh-3.1$ ./omp_dp1
8 parallel workers
Parallel Loop took 0.0088 seconds
Serial Loop took 0.0179 seconds
```

Scheduling Loops
#pragma omp parallel for schedule(sched_type)

Domain Decomposition Analogs

- **schedule(static)** *= block [usually this is the default]*
  - statically allocate (Total Iterations/Total Threads) contiguous iterations per thread
- **schedule(static, chunk)** *= block cyclic*
  - Interleaved allocation of "chunks" (chunk = number of iterations) to each thread

Master-Worker Analogs

- **schedule(dynamic)** *= fine-grained master-worker*
  - Iterations dynamically allocated to each thread, 1 at a time
- **schedule(dynamic, chunk)** *= coarse-grained master-worker*
  - Iterations dynamically allocated to each thread, 1 chunk at a time

Unique to OpenMP

- **schedule(guided)** *= coarse- to fine-grained master-worker*
- **schedule(guided, chunk)** *(fill in the cracks)*
  - Dynamic allocation of iterations using "guided self-scheduling"
  - Large chunks shrink exponentially to a minimum size chunk (1 if not specified)
  - First chunk goes like *n/np*; subsequent chunks go like *(n-ndone)/np*

Basic OpenMP Functions

- **omp_get_num_procs**
```
int procs = omp_get_num_procs()  //number of CPUs in machine
```
- **omp_get_num_threads**
```
int threads = omp_get_num_threads()  //# of active threads
```
- **omp_get_max_threads**
```
printf("%d threads will be started\n",omp_get_max_threads());
```
- **omp_get_thread_num**
```
printf("Hello from thread id %d\n",omp_get_thread_num());
```
- **omp_set_num_threads** – *query this setting with omp_get_max_threads*
```
omp_set_num_threads(procs * atoi(argv[1]));
```

Program omp_helloworld.c – 1 of 2

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

int main(int argc, char **argv )
 {
  char host[80];
  int procs,i,a,b,c,t,m;

  if (gethostname(host,sizeof(host)) != 0)
   {
    printf("ERROR getting hostname\n");
    return 30;
   }
  printf("Hello from %s\n", host);
  // Determine the number of physical processors
  procs = omp_get_num_procs();
  printf("%s has %d processors\n",host,procs);
```

Program omp_helloworld.c – 2 of 2

```
  /* Determine max threads defined by default through
     through the OMP_NUM_THREADS environment variable */
  printf("OMP_NUM_THREADS = %d\n",omp_get_max_threads());
  printf("OMP_DYNAMIC = ");
  if (!omp_get_dynamic())
   {
    printf("FALSE\n");
    // Set the number of threads, it will be exact
    printf("Setting NUM_THREADS = %d\n",procs * atoi(argv[1]));
    omp_set_num_threads(procs * atoi(argv[1]));
   }
  else printf("TRUE\n");
  /* Report maximum number of threads that can be created, the
     actual number will be determined dynamically at runtime */
  printf("Maximum number of threads = %d\n",omp_get_max_threads());
  #pragma omp parallel
  printf("Hello from thread id %d\n",omp_get_thread_num());
  return 0;
}
```

OpenMP Environment Variables

- **OMP_DYNAMIC**
  - Enables (if TRUE) or disables (if FALSE) dynamic loop scheduling
  - Use of omp_set_dynamic tends to be useful if your code includes the OpenMP nowait clause. When nowait is used then subsequent parallel sections may be arrived at prior to having the full complement of threads available for processing. If the system is directed to use 4 threads and if prior sections include the nowait clause then it is possible, as an example, to arrive at a parallel for loop with 3 of the 4 threads ready for execution. If dynamic is 0 then 4 threads are allocated with 1 of the thread execution range blocked until the prior code on the busy thread completes. If dynamic is .not. 0 (e.g. 1) then the runtime system could elect to distribute the processing amongst the 3 available threads.
- **OMP_NUM_THREADS**
  - Sets the number of threads to use during parallel execution
  - Is roughly equivalent to “mpiexec -n”, but pertains only to a single process on a single machine
  - Can be overridden by a call to omp_set_num_threads

One More OpenMP Environment Variable: OMP_SCHEDULE

- **schedule(runtime)**
  - **Schedule type chosen at runtime based on the value of the OMP_SCHEDULE environment variable**
  - **Example:**

```
Set OMP_SCHEDULE="dynamic,10"
```

Private Variables
private clause

- Tells compiler to allocate a private copy of a variable for each thread
- Without it, all threads would clobber the same shared memory location
  - Result would be nondeterministic

```
start = omp_get_wtime();
h = 1.0 / (double) n;
area = 0.0;
#pragma omp parallel for private(x)
for (i = 1; i <= n; i++)
 {
  x = h * ((double)i - 0.5);
  area += (4.0 / (1.0 + x*x));
 } // for
pi = h * area;
end = omp_get_wtime();
```

Critical Sections
#pragma omp critical

- **Forces threads to be mutex (mutually exclusive)**
  - Only one thread at a time executes the given code section
- **Here, keeps two threads from updating "area" at once**
  - The += operator is not atomic (load, add, store)
  - Results are non-deterministic due to a "race condition"
  - With a critical section , operation becomes atomic in effect

Wrong

```
start = omp_get_wtime();
h = 1.0 / (double) n;
area = 0.0;
#pragma omp parallel for private(x)
for (i = 1; i <= n; i++)
 {
  x = h * ((double)i - 0.5);
  area += (4.0 / (1.0 + x*x));
 } // for
pi = h * area;
end = omp_get_wtime();
```

Right

```
start = omp_get_wtime();
h = 1.0 / (double) n;
area = 0.0;
#pragma omp parallel for private(x)
for (i = 1; i <= n; i++)
 {
  x = h * ((double)i - 0.5);
  #pragma omp critical
  area += (4.0 / (1.0 + x*x));
 } // for
pi = h * area;
end = omp_get_wtime();
```

Reductions
reduction(<operator>:variable)

- Reduction clause can be added to #pragma omp parallel for
- Works like MPI_Reduce
- Reduction operators for C & C++:

Operator	Meaning	Types	Initial Value
+	Sum	float,int	0
*	Product	float,int	1
&	Bitwise and	int	all bits 1
|	Bitwise or	int	0
^	Bitwise exclusive or	int	0
&&	Logical and	int	1
||	Logical or	int	0

```
start = omp_get_wtime();
h = 1.0 / (double) n;
area = 0.0;
#pragma omp parallel for private(x) reduction(+:area)
for (i = 1; i <= n; i++)
 {
  x = h * ((double)i - 0.5);
  area += (4.0 / (1.0 + x*x));
 } // for
pi = h * area;
end = omp_get_wtime();
```

firstprivate & lastprivate clauses

- **"firstprivate" tells compiler to create private variables with the same initial value as that of the Master thread.**
  - **firstprivate variables are initialized ONCE/thread**

```
X[0] = complex_function()
#pragma omp parallel for private(j) firstprivate(x)
for (i=0; i<n; i++)
 {
  for (j=1; j<4; j++)  x[j] = g(i,x[j-1]);
  answer[i] = x[1] -x[3];
 }
```

- **"lastprivate" tells compiler to copy back to the Master thread the private copy executed on the "sequential last iteration" of a loop. (This is the last iteration that would be executed if the code were serial)**

```
#pragma omp parallel for private(j) lastprivate(x)
for (i=0; i<n; i++)
 {
  x[0] = 1.0;
  for (j=1; j<4; j++)  x[j] = x[j-1] * (i+1);
  sum_of_powers[i] = x[0] + x[1] + x[2] + x[3];
 }
n_cubed = x[3];
```

Note: examples from pages 412-413 in the text

single Pragma

- Code section will be executed by just a single thread

```
void main(int argc, char **argv )
 {
  int i,a,b;

  //Determine how many threads set by OMP_NUM_THREADS environment variable
  printf("%d threads possible\n",omp_get_max_threads());
  #pragma omp parallel private(i)
  for (i=0;i<4;i++)
    {
     #pragma omp single
     printf("Currently %d threads in use\n",omp_get_num_threads());
     printf("thread %d working on i=%d\n",omp_get_thread_num(),i);
     a = i;
     b = 2 * i;
     a += b;
    }
 }
```

OpenMP Internal Control Variables (ICVs) and How to Set Them

#pragma omp clause, if used	**overrides call to API routine** [related query]	**overrides setting of environment variable**	**overrides initial value of**
(none)	**omp_set_dynamic()** omp_get_dynamic()	**OMP_DYNAMIC**	***dyn-var***
num_threads	**omp_set_num_threads()** omp_get_max_threads()	**OMP_NUM_THREADS**	***nthreads-var***
(none)	**omp_set_nested()** omp_get_nested()	**OMP_NESTED**	***nest-var***
schedule	**omp_set_schedule()** omp_get_schedule()	**OMP_SCHEDULE**	***run-sched-var***

Row-Major (C) vs. Column-Major (Fortran)

Assume the following Matrix a[i][j]:

A11 A12 A13

A21 A22 A23

A31 A32 A33

C Loop:

```
for (i=1; i<=3; i++)
 for (j=1;j<=3;j++)
  a[i][j]
```

Row-Major (C) Stores the values in memory

A11 A12 A13 A21 A22 A23 A31 A32 A33 ---> Higher addresses

j index changes faster then the i index

Column-Major (Fortran) Stores the values in memory

A11 A21 A31 A12 A22 A32 A13 A23 A33 ---> Higher addresses

i index changes faster than the j index

Inverting Loops Example

- Columns can be updated simultaneously (not rows)
- Inverting the i & j loops reduces the number of fork & joins
- Consider how transformation affects the cache-hit rate

```
for (i=2; i<=m; i++)
  for (j=1;j<=n;j++)
    a[i][j] = 2 * a[i-1][j]
```

Becomes:

```
#pragma parallel for private(i)
for (j=1;j<=n;j++)
 for (i=2; i<=m; i++)
  a[i][j] = 2 * a[i-1][j]
```

```
A11 A12 A13
A21 A22 A23
A31 A32 A33
```

Note: examples from page 417 in the text

Consider how array is referenced in Memory

```
for (i=2; i<=m; i++)
  for (j=1;j<=n;j++)

A21 & A11  i=2, j=1 to 3
A22 & A12
A23 & A13
A31 & A21  i=3, j=1 to 3
A32 & A22
A33 & A23
```

```
A11 A12 A13
A21 A22 A23
A31 A32 A33
```

```
for (j=1;j<=n;j++)
 for (i=2; i<=m; i++)

A21 & A11  j=1 i=2 to 3
A31 & A21
A22 & A12  j=2 i=2 to 3
A32 & A22
A23 & A13  j=3 i=2 to 3
A33 & A23
```

Note: examples from page 417 in the text

Conditional Execution of Loops
if clause

- **Initiating a *parallel for* adds to overhead due to thread forking**
- **Only makes sense to do a *parallel for* if loop's trip count (n) is high**
- **Use the *if* clause to avoid parallel execution when n is too small**

```
start = omp_get_wtime();
h = 1.0 / (double) n;
area = 0.0;
#pragma omp parallel for private(x) reduction(+:area) if(n > 5000)
for (i = 1; i <= n; i++)
 {
  x = h * ((double)i - 0.5);
  area += (4.0 / (1.0 + x*x));
 } // for
pi = h * area;
end = omp_get_wtime();
```

Parallelizing montepi.c with OpenMP

- Use omp_get_num_threads, omp_get_thread_num
- Separate parallel and for pragmas

Functional Parallelism Example

$$\begin{tikzcd}
\alpha \arrow[dr] & & \beta \arrow[dl] & \\
& \gamma \arrow[dr] & & \delta \arrow[dl] \\
& & \varepsilon &
\end{tikzcd}$$

```
v = alpha();
W = beta();
x = gamma(v,w);
y = delta();
printf(“%6.2f\n”, epsilon(x,y));
```

Functional Parallelism with OpenMP, 1
parallel sections, section pragmas

```
#pragma omp parallel sections
  {
    #pragma omp section //optional
    v = alpha();
    #pragma omp section
    w = beta();
    #pragma omp section
    y = delta();
  }
x = gamma(v,w);
printf("%6.2f\n", epsilon(x,y));
```

Functional Parallelism with OpenMP, 2
separating parallel from sections

```
#pragma omp parallel
  {
    #pragma omp sections
      {
        #pragma omp section //optional
        v = alpha();
        #pragma omp section
        w = beta();
      }
    #pragma omp sections
      {
        #pragma omp section //optional
        x = gamma(v,w);
        #pragma omp section
        y = delta();
      }
  }
printf(“%6.2f\n”, epsilon(x,y));
```