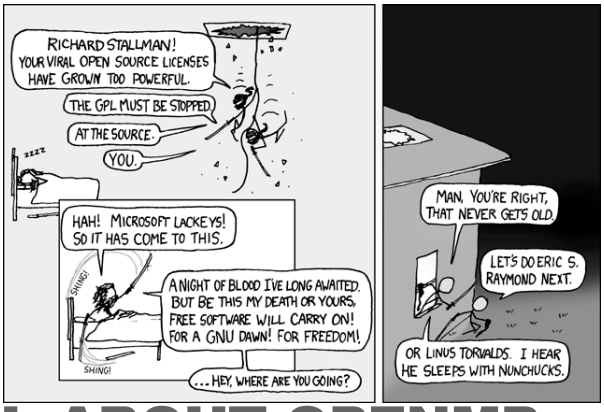
Advanced Crash Course in Supercomputing: OpenMP

Rebecca Hartman-Baker
Oak Ridge National Laboratory
hartmanbakrj@ornl.gov

© 2004-2009 Rebecca Hartman-Baker. Reproduction permitted for non-commercial, educational use only.

Outline

I. About OpenMP
II. OpenMP Directives
III. Data Scope
IV. Runtime Library Routines and Environment Variables
V. Using OpenMP



I. ABOUT OPENMP

Source: http://xkcd.com/225/

NCCS

About OpenMP

- Industry-standard shared memory programming model
- Developed in 1997
- OpenMP Architecture Review Board (ARB) determines additions and updates to standard

NCCS

Advantages to OpenMP

- Parallelize small parts of application, one at a time (beginning with most time-critical parts)
- Can express simple or complex algorithms
- Code size grows only modestly
- Expression of parallelism flows clearly, so code is easy to read
- Single source code for OpenMP and non-OpenMP – non-OpenMP compilers simply ignore OMP directives

NCCS

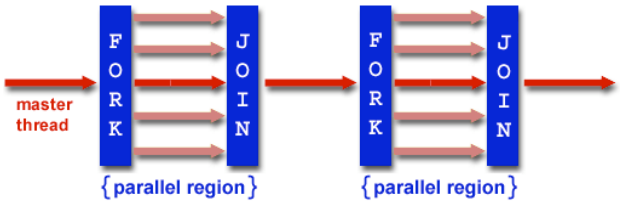
OpenMP Programming Model

- Application Programmer Interface (API) is combination of
  - Directives
  - Runtime library routines
  - Environment variables
- API falls into three categories
  - Expression of parallelism (flow control)
  - Data sharing among threads (communication)
  - Synchronization (coordination or interaction)

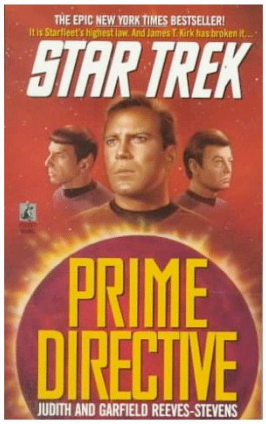
NCCS

Parallelism

- Shared memory, thread-based parallelism
- Explicit parallelism (parallel regions)
- Fork/join model



Source: https://computing.llnl.gov/tutorials/openMP/



II. OPENMP DIRECTIVES

Star Trek: Prime Directive **by Judith and Garfield Reeves-Stevens, ISBN 0671744666**

II. OpenMP Directives

- Parallel
- Loop
- Sections
- Synchronization

NCCS

OpenMP Directives: Parallel

- A block of code executed by multiple threads
- Syntax:

```
#pragma omp parallel private(list)\
  shared (list)
{
    /* parallel section */
}
```

NCCS

Simple Example

```
#include <stdio.h>
#include <omp.h>
int main (int argc, char *argv[]) {
  int tid;
  printf("Hello world from threads:\n");
  #pragma omp parallel private(tid)
  {
   tid = omp_get_thread_num();
   printf("<%d>\n", tid);
  }
  printf("I am sequential now\n");
  return 0;
}
```

NCCS

Output (Simple Example)

Output 1

```
Hello world from
  threads:
<0>
<1>
<2>
<3>
<4>
I am sequential now
```

Output 2

```
Hello world from
  threads:
<1>
<2>
<0>
<4>
<3>
I am sequential now
```

Order of execution is scheduled by OS!!!!!!

NCCS

OpenMP Directives: Loop

- Iterations of the loop following the directive are executed in parallel
- Syntax:
  - `#pragma omp for schedule(type [,chunk]) \ private(list) shared(list) nowait`

```
{
  /* for loop */
}
```

  - *type* = {`static`, `dynamic`, `guided`, `runtime`}
  - If `nowait` specified, threads do not synchronize at end of loop

NCCS

Which Loops Are Parallelizable?

Parallelizable

- Number of iterations known upon entry, and does not change
- Each iteration independent of all others
- No data dependence

Not Parallelizable

- Conditional loops (many while loops)
- Iterator loops (e.g., iterating over a `std::list<...>` in C++)
- Iterations dependent upon each other
- Data dependence

NCCS

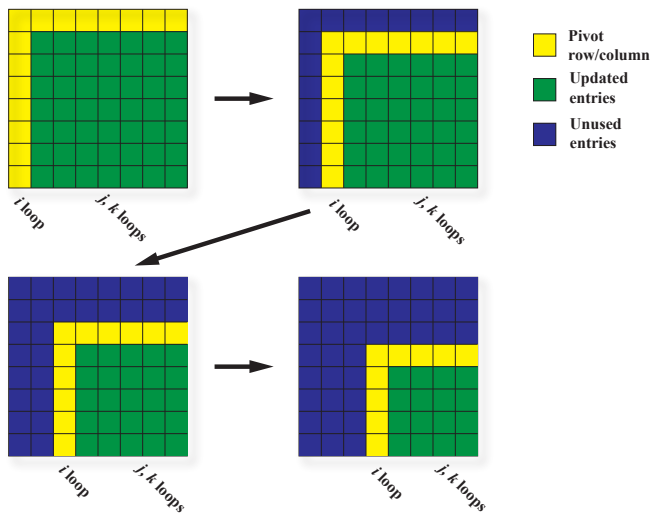
Example: Parallelizable?

```
/* Gaussian Elimination (no pivoting):
   x = A\b                                  */

for (int i = 0; i < N-1; i++) {
  for (int j = i; j < N; j++) {
    double ratio = A[j][i]/A[i][i];
    for (int k = i; k < N; k++) {
      A[j][k] -= (ratio*A[i][k]);
      b[j] -= (ratio*b[i]);
    }
  }
}
```

NCCS

Example: Parallelizable?



NCCS

Example: Parallelizable?

- Outermost Loop (`i`):
  - `N-1` iterations
  - Iterations depend upon each other (values computed at step `i-1` used in step `i`)
- Inner loop (`j`):
  - `N-i` iterations (constant for given `i`)
  - Iterations can be performed in any order
- Innermost loop (`k`):
  - `N-i` iterations (constant for given `i`)
  - Iterations can be performed in any order

NCCS

Example: Parallelizable?

```
/* Gaussian Elimination (no pivoting):
   x = A\b                                  */

for (int i = 0; i < N-1; i++) {
#pragma omp parallel for
  for (int j = i; j < N; j++) {
    double ratio = A[j][i]/A[i][i];
    for (int k = i; k < N; k++) {
      A[j][k] -= (ratio*A[i][k]);
      b[j] -= (ratio*b[i]);
    }
  }
}
```

Note: can combine `parallel` and `for` into single `pragma` line

NCCS

OpenMP Directives: Loop Scheduling

- Default scheduling determined by implementation
- Static
  - ID of thread performing particular iteration is function of iteration number and number of threads
  - Statically assigned at beginning of loop
  - Load imbalance may be issue if iterations have different amounts of work
- Dynamic
  - Assignment of threads determined at runtime (round robin)
  - Each thread gets more work after completing current work
  - Load balance is possible

NCCS

Loop: Simple Example

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000
int main ()  {
  int i, chunk;
  float a[N], b[N], c[N];
  /* Some initializations */
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
  chunk = CHUNKSIZE;
  #pragma omp parallel shared(a,b,c,chunk) private(i)
  {
    #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
      c[i] = a[i] + b[i];
  }  /* end of parallel section */
  return 0;
}
```

NCCS

OpenMP Directives: Sections

- Non-iterative work-sharing construct
- Divide enclosed sections of code among threads
- Section directives nested within sections directive
- Syntax

```
#pragma omp sections
{
  #pragma omp section
  /* first section */
  #pragma omp section
  /* next section */
}
```

NCCS

Sections: Simple Example

```
#include <omp.h>
#define N     1000
int main () {
  int i;
  double a[N], b[N], c
   [N], d[N];
  /* Some initializations
   */
  for (i=0; i < N; i++) {
    a[i] = i * 1.5;
    b[i] = i + 22.35;
  }
```

```
  #pragma omp parallel shared(a,b,c,d)
   \ private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
        for (i=0; i < N; i++)
          c[i] = a[i] + b[i];
      #pragma omp section
        for (i=0; i < N; i++)
         d[i] = a[i] * b[i];
     }  /* end of sections */
   }  /* end of parallel section */
return 0;
}
```

NCCS

OpenMP Directives: Synchronization

- Sometimes, need to make sure threads execute regions of code in proper order
  - Maybe one part depends on another part being completed
  - Maybe only one thread need execute a section of code
- Synchronization directives
  - Critical
  - Barrier
  - Single

NCCS

OpenMP Directives: Synchronization

- Critical
  - Specifies section of code that must be executed by only one thread at a time
  - Syntax

    ```
    #pragma omp critical [name]
    ```

  - Names are global identifiers – critical regions with same name are treated as same region
- Single
  - Enclosed code is to be executed by only one thread
  - Useful for thread-unsafe sections of code (e.g., I/O)
  - Syntax

    ```
    #pragma omp single
    ```

NCCS

OpenMP Directives: Synchronization

- Barrier
  - Synchronizes all threads: thread reaches barrier and waits until all other threads have reached barrier, then resumes executing code following barrier
  - Syntax

```
#pragma omp barrier
```

  - Sequence of work-sharing and barrier regions encountered must be the same for every thread

III. VARIABLE SCOPE

Angled spotting scope. Source: http://www.spottingscopes.us/angled-scope-328.jpg

Variable Scope

- By default, all variables shared except
  - Certain loop index values – private by default
  - Local variables and value parameters within subroutines called within parallel region – private
  - Variables declared within lexical extent of parallel region – private

NCCS

Default Scope Example

```
void caller(int *a, int n) {
int i,j,m=3;
#pragma omp parallel for
for (i=0; i<n; i++) {
  int k=m;
  for (j=1; j<=5; j++) {
    callee(&a[i], &k, j);
  }
}
void callee(int *x, int *y, int
   z) {
  int ii;
  static int cnt;
  cnt++;
  for (ii=1; ii<z; ii++) {
    *x = *y + z;
  }
}
```

Var	Scope	Comment
a	shared	Declared outside parallel construct
n	shared	same
i	private	Parallel loop index
j	shared	Sequential loop index
m	shared	Declared outside parallel construct
k	private	Automatic variable/parallel region
x	private	Passed by value
*x	shared	(actually a)
y	private	Passed by value
*y	private	(actually k)
z	private	(actually j)
ii	private	Local stack variable in called function
cnt	shared	Declared static (like global)

NCCS

Variable Scope

- Good programming practice: explicitly declare scope of all variables
- This helps you as programmer understand how variables are used in program
- Reduces chances of data race conditions or unexplained behavior

NCCS

Variable Scope: Shared

- Syntax
  - `shared(list)`
- One instance of shared variable, and each thread can read or modify it
- WARNING: watch out for multiple threads simultaneously updating same variable, or one reading while another writes
- Example

```
#pragma omp parallel for shared(a)
for (i = 0; i < N; i++) {
  a[i] += i;
}
```

NCCS

Variable Scope: Shared – Bad Example

```
#pragma omp parallel for shared(n_eq)
for (i = 0; i < N; i++) {
  if (a[i] == b[i]) {
    n_eq++;
  }
}
```

- `n_eq` will not be correctly updated
- Instead, put `n_eq++;` in critical block (slow) or introduce private variable `my_n_eq`, then update `n_eq` in critical block after loop (faster)

NCCS

Variable Scope: Private

- Syntax
  - `private(list)`
- Gives each thread its own copy of variable
- Example

```
#pragma omp parallel private(i, my_n_eq)
{
  #pragma omp for
  for (i = 0; i < N; i++) {
    if (a[i] == b[i])  my_n_eq++;
  }
  #pragma omp critical (update_sum)
  {
    n_eq+=my_n_eq;
  }
}
```

NCCS

Another Solution for Sum

```
#pragma parallel for reduction
  (+:n_eq)
for (i = 0; i < N; i++) {
  if (a[i] == b[i]) {
    n_eq = n_eq+1;
  }
}
```

IV. RUNTIME LIBRARY ROUTINES AND ENVIRONMENT VARIABLES

Mt. McKinley National Monument, July, 1966. Source: National Park Service Historic Photograph Collection, http://home.nps.gov/applications/hafe/hfc/npsphoto4h.cfm?Catalog_No=hpc-001845

OpenMP Runtime Library Routines

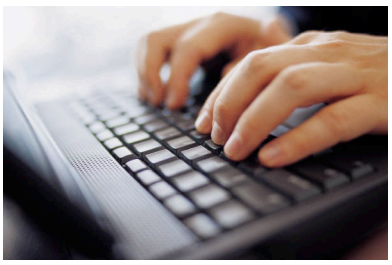
- `void omp_set_num_threads(int num_threads)`
  - Sets number of threads used in next parallel region
  - Must be called from serial portion of code
- `int omp_get_num_threads()`
  - Returns number of threads currently in team executing parallel region from which it is called
- `int omp_get_thread_num()`
  - Returns rank of thread
  - `0 ≤ omp_get_thread_num() < omp_get_num_threads()`

NCCS

OpenMP Environment Variables

- Set environment variables to control execution of parallel code
- `OMP_SCHEDULE`
  - Determines how iterations of loops are scheduled
  - E.g., `setenv OMP_SCHEDULE "guided, 4"`
- `OMP_NUM_THREADS`
  - Sets maximum number of threads
  - E.g., `setenv OMP_NUM_THREADS 4`

NCCS



V. USING OPENMP

NCCS

Conditional Compilation

- Can write single source code for use with or without OpenMP
- Pragmas are ignored
- What about OpenMP runtime library routines?
  - `_OPENMP` macro is defined if OpenMP available: can use this to conditionally include `omp.h` header file, else redefine runtime library routines

NCCS

Conditional Compilation

```
#ifdef _OPENMP
  #include <omp.h>
#else
  #define omp_get_thread_num() 0
#endif
…
int me = omp_get_thread_num();
…
```

NCCS

Running Programs with OpenMP Directives

- May need special compiler options (e.g., for PGI compilers, use `-mp=nonuma` flag)
- May need to set environment variables in batch scripts (e.g., on Jaguar, include definition of `OMP_NUM_THREADS` in script)
- Example: to run on 64 quad-core nodes on Jaguar, add the following to your script requesting 256 procs:

```
export OMP_NUM_THREADS=4
aprun -n 64 -N 1 myprog
```

NCCS

Bibliography/Resources: OpenMP

- Chapman, Barbara, Gabrielle Jost, and Ruud van der Pas. (2008) *Using OpenMP*, Cambridge, MA: MIT Press.
- Kendall, Ricky A. (2007) *Threads R Us*, http://www.nccs.gov/wp-content/training/scaling_workshop_pdfs/threadsRus.pdf
- LLNL OpenMP Tutorial, https://computing.llnl.gov/tutorials/openMP/

NCCS