Intermediate OpenMP

Mobeen Ludin

What we learned so far about OpenMP?

- A standard Application Programming Interface (API) for writing shared memory parallel applications in C, C++, and Fortran

- Where to find more information, examples, exercises:
  - OpenMP Web-page: http://www.openmp.org
  - Lawrence Livermore NL: https://computing.llnl.gov/tutorials/openMP/

Components of OpenMP

Directives:

- Parallel region
- work sharing constructs
- Tasking
- Synchronization
- Data-sharing attributes

Library Routines:

- Number of threads
- Threads ID
- Dynamic thread adjustment
- Nested Parallelism
- Schedule
- Active Levels
- Thread limit
- Nesting level
- Ancestor thread
- Team Size
- Wallclock timer
- Locking

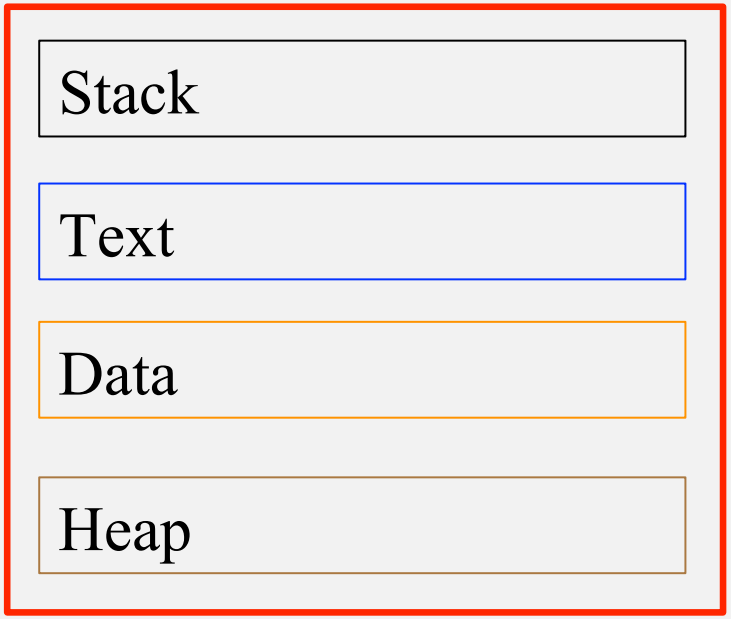
Environment Variables:

- Number of threads
- Scheduling type
- Dynamic thread adjustment
- Nested parallelism
- Stacksize
- Idle thread
- Active levels
- Thread limit

Difference between a process and a thread

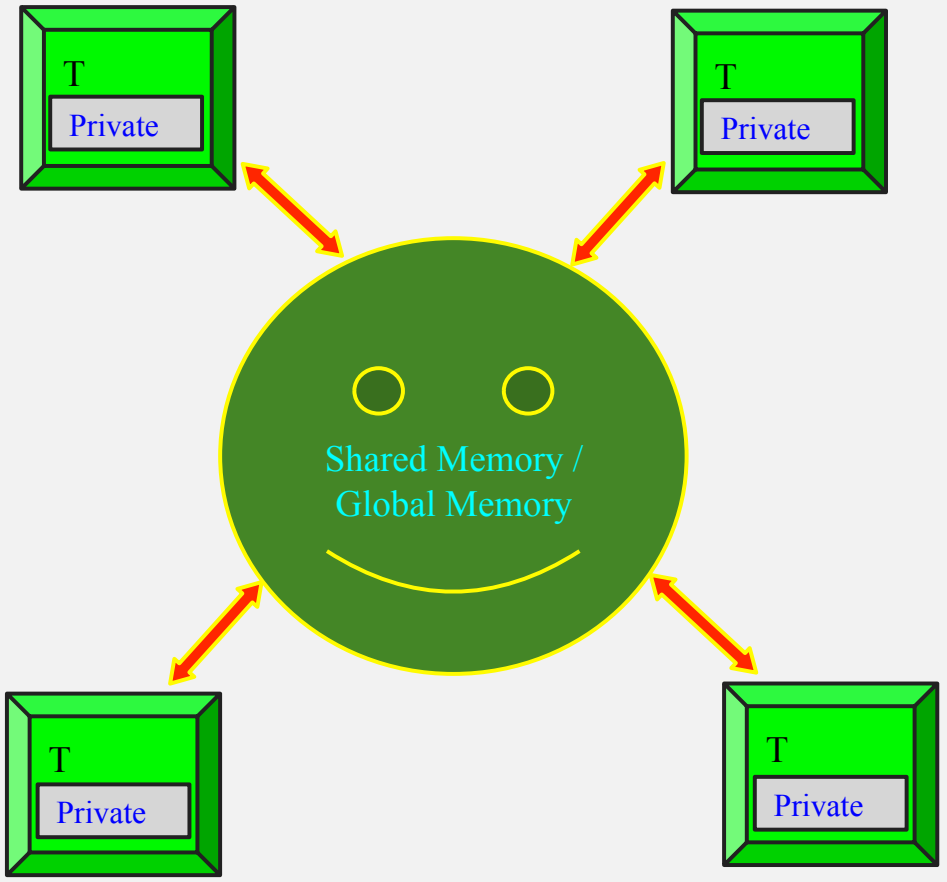
Process and Thread are two unit of executions that are not the same in the sense of executing environment.

- ❖ Compiled program requites CPU to execute its instructions
- ❖ Requires its own memory space for storing its execution environment
  - ➢ Text segment: Storing program code
  - ➢ Heap: Storing global data
  - ➢ Stack: Storing local data



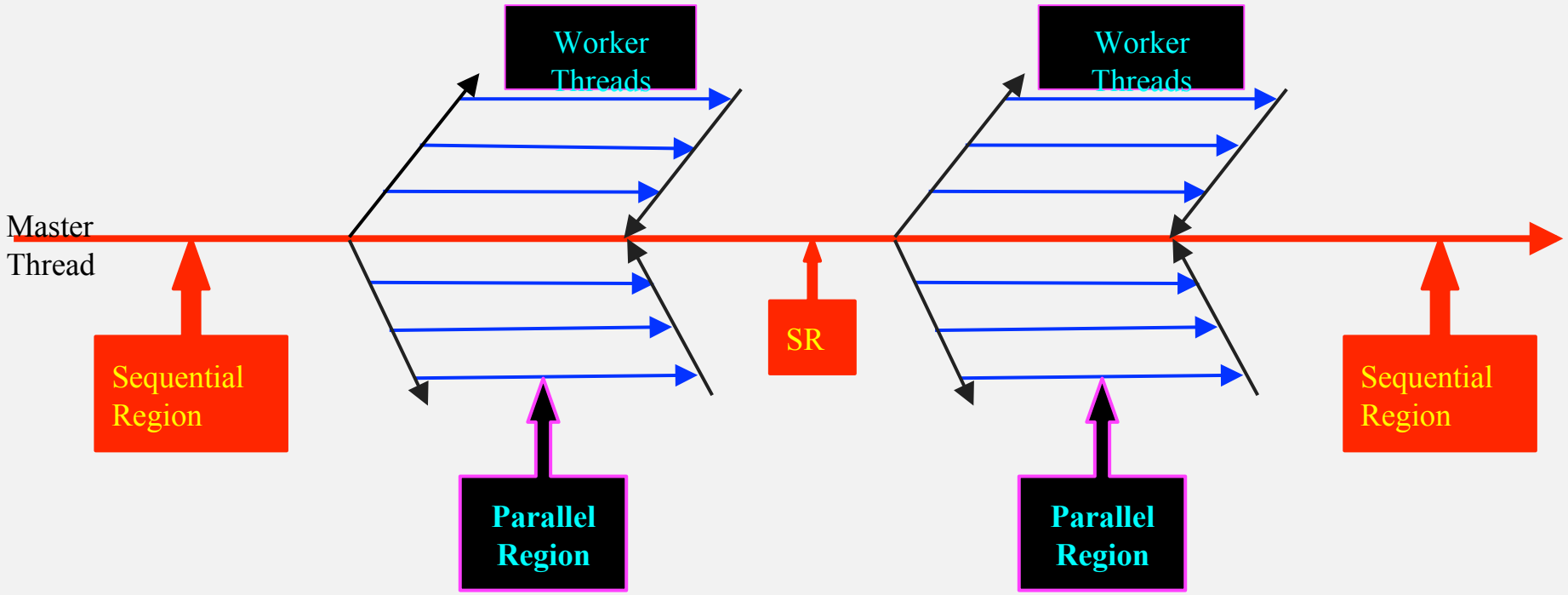
OpenMP memory model

- All thread have access to the same **globally shared memory**
- Data can be **shared** or **private**
- Shared data is accessible by all threads
- Private data can only be accessed by the thread that owns it
- Data transfer is **transparent** to the programmer
- **Synchronization** takes place, but its mostly implicit

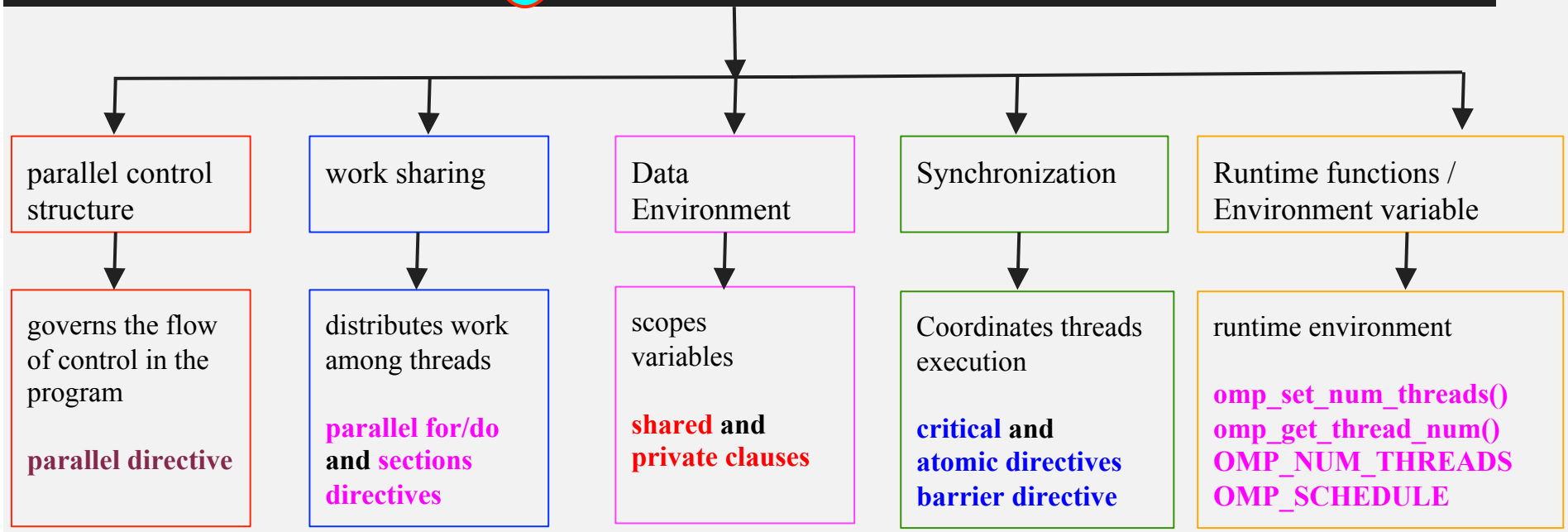


OpenMP execution model

- **Fork-Join Parallelism:**
  - **Master** thread spawns team of threads as needed
  - Parallelism added incrementally until performance goals are met



OpenMP Core Elements



- The above categorized constructs are the core elements of OpenMP.
- All the categories and their elements are the same in C/C++ and Fortran

OpenMP programming model

- **Directives:** OpenMP directives in C/C++ are based on the `#pragma` compiler directives. The directive itself consist of directive name followed by a clause
  - `#pragma omp directive_name [clause list]`
  - Example: `#pragma omp parallel`
    - OpenMP programs execute serially until they encounter the **parallel** directive.
    - This directive is responsible for creating group of threads.
    - The exact number of thread can be specified using **environment variable** or at **runtime using OpenMP functions,** or **clause**.
    - The main thread that encounters the `parallel` directive becomes the **master** thread of this group of threads and is assigned the **thread id 0**

Directive format:

- **C/C++ directives are case sensitive, Fortran is case insensitive**
- **C/C++ Syntax:**
  - `#pragma omp directive [clause [clause] ...]`
  - **Brake lines: use** `\`
- **Fortran Syntax:**
  - `sentinel directive [clause [[,] clause]...]`
  - The **sentinel** is one of the following:
    - `!$OMP or C$OMP or *$OMP` **(fixed format)**
    - `!$OMP` **(free format)**
  - **Brake lines: use the language syntax (`&`)**

Parallel directive example:

```
#include <stdio.h>
#include <omp.h>
int main(){
#pragma omp parallel
     printf("Hello from thread %d, of nthreads %d: \n",
 omp_get_thread_num(), omp_get_num_threads());
}
```

Compile: $ `gcc -fopenmp omp_hello_world_omp.c –o omp_helloworld`
BW Compile: $ `cc omp_hello_world.c -o omp_hello_world`
Run: $ `export OMP_NUM_THREADS=4 && ./omp_hello_world`
Output:

Hello from thread 3, of nthreads 4:
Hello from thread 2, of nthreads 4:
Hello from thread 0, of nthreads 4:
Hello from thread 1, of nthreads 4:

What are OpenMP Clauses for?

- The **clause [list]** is used to specify **conditional parallelization**, **number of threads**, **and data handling**

  - **Conditional parallelization:** The clause `if(scalar expression)` determines whether the parallel construct results in creation of threads. **Only one if clause can be used with a parallel directive.**

  - **Degree of concurrency:** The **clause** `num_threads(integer expression)` specifies the number of threads that are created **by the parallel directive**.

The “if()” clause and “num_threads() clause example:

```
int i; double area = 0.0;
  // Serial segment of the code is here
#pragma omp parallel if (n > 100) num_threads(16)
{ // Start of parallel region
  for (i=0; i<n; i++)
        x[i] += y[i];
} // End of parallel region
```

- This program will only execute in parallel when if expression evaluates to true. Otherwise it will just run in serial
- Overhead of fork/join is high
- If a loop is small, you don't want to parallelize.
- 12 threads each gets 1 block, but last four threads gets each 2 blocks
  - doing more work than some other threads. (make the blocks smaller for equal work load)

Specifying concurrent tasks in OpenMP

The parallel directive can be used in conjunction with other directives to specify concurrency across iterations and tasks.

OpenMP provides two directives (**for and sections**) to specify concurrent iterations

The for directive:

```
#pragma omp parallel for
{
 for(i=1; i<n; i++)
     b[i]=(a[i]+a[i-1])/2.0
}
```

Assigning Iterations to Threads

Parallel loop in C/C++ and Fortran:

// C/C++ OpenMP code:

```
void example(int n, float *a,
 float *b)
{
 int i;
#pragma omp parallel for
 for(i=1; i<n; i++)
     b[i]=(a[i]+a[i-1])/2.0
}
```

!Fortran OpenMP code:

```
SUBROUTINE EXAMPLE(N, A, B)
         INTEGER I, N
    REAL B(N), A(N)
!$OMP PARALLEL DO
         DO I=2,N
            B(I) = (A(I) + A(I-1))/2.0
         ENDDO
!$OMP END PARALLEL DO
         END SUBROUTINE EXAMPLE
```

Assigning iterations to threads: using schedule clause

- The schedule clause of the for directive deals with the assignment of the iterations to the threads. Syntax: `schedule(scheduling_class [, parameter])`
  - `schedule(static [, chunk-size ])`
    - Distribute the work evenly or in chunk size units specified
    - Pre-determined and predictable amount of work between each iterations
    - compile time
  - `schedule(dynamic [, chunk-size ])`
    - Distribute the work on available threads in chunk size specified
    - When no idea how long each iterations will take.
    - most work is done runtime
  - `schedule(guided [, chunk-size ])`
    - Variation of dynamic starting from large chunks sizes and exponentially going down to chunk size
  - `schedule(auto)`
    - Compiler do whatever the heck you thinks is best to get some performance
    - supported only in the newer version of OpenMP
  - `schedule( runtime )`
    - the environment variable `OMP_SCHEDULE` which is one of the static, dynamic, guided or an appropriate pair like:
      - `export OMP_SCHEDULE="static,500"`

How OpenMP threads interact with each other?

- **OpenMP** is a multi-threading, shared address model
  - Threads **communicate** by sharing variables

- Unintended sharing of data causes **race condition**
  - **Race condition:** when the program's outcome changes as the threads are scheduled differently

- To control race condition:
  - Use **synchronization** to protect data conflicts

- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization

Race Condition Exercise

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>   // For OpenMP
int main(int argc, char **argv) {
    int i, j, tID;
   printf("There is something wrong with this example FIX IT\n”);
#pragma omp parallel private(i, j)
 {
   for(i = 0; i < 1000; i++)
     for(j = 0; j < 1000; j++)
       tID = omp_get_thread_num();
   printf("Thread %d : My value of tid (thread id) is %d \n",
 omp_get_thread_num(), tID);
 }
   printf("\n Did you figure out what is wrong? \n");
  printf("\nHint: do a # comparison\n”);
}
```

OpenMP clauses (cont.): Data handling/sharing

- In an OpenMP program data need to be **labeled**
- There are two ways one could label data in OpenMP
  - **Shared:** There is only one instance of the data
    - All the threads can read and write the date simultaneously, unless **protected** through a specific OpenMP construct
    - All changes made are available to all threads
      - But not necessarily immediately, unless inforced
  - **Private:** Each thread has its own copy of the data
    - No other threads can have `R/W` access to this data
    - Changes only **visible** to the thread that owns the data

private and shared clauses

- **`shared ( list )`**
  - ➢ Data is accessible by all the threads in the team
  - ➢ All threads access the same address space

- **`private ( list )`**
  - ➢ No changes associated with the original object
  - ➢ All references are to the local object
  - ➢ Values are undefined on entry and exit

Variable Initialization

- **`firstprivate(list)`**

All variables in the list are initialized with the value the original object had before entering the parallel construct

- **`lastprivate(list)`**

The thread that executes the sequentially last iteration or section updates the value of the objects in the list

Initialization example:

```
#include <stdio.h>
#include <omp.h>
int main (void)
{
    int i = 10;
#pragma omp parallel firstprivate(i)
    {
        printf("thread %d: i = %d \n",
  omp_get_thread_num(), i);
        i = 1000 +
  omp_get_thread_num();
    }
    printf("i = %d \n", i);
    return 0;
}
```

Runtime Experiment:

Compile: `$ gcc -fopenmp firstprivate.c -o firstprivate`

Output: `$ export OMP_NUM_THREADS=4 && ./firstprivate`

```
thread 0: i = 0
thread 1: i = -1
thread 3: i = 0
thread 2: i = 0
i = 10
```

Output when `i` is used as private:

`private(i)`

```
thread 1: i = 10
thread 3: i = 10
thread 0: i = 10
thread 2: i = 10
i = 10
```

Output when `i` is initialized as firstprivate:

`firstprivate(i)`

reduction(operator:list) clause

Reduction clause performs a reduction on the variables appear in its list.

Syntax:

```
reduction ( [operator | intrinsic] ) : list )
```

- ➢ Reduction variable(s) must be shared variables
- ➢A reduction is defined as:

```
reduction ( operator : list )
```

A simple OpenMP example

```
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>   // For OpenMP
#define NUM_THREADS 2
static long num_steps = 100000;
double step;
void main (){
 int i;
 double x, pi, sum = 0.0;
 step = 1.0/(double) num_steps;
 omp_set_num_threads (NUM_THREADS);
 #pragma omp parallel for reduction(+:sum) private(x)
 for (i=1;i<= num_steps; i++){
       x = (i-0.5)*step;
       sum = sum + 4.0/(1.0+x*x);
  }
 pi = step * sum;
 printf(" pi is %f \n",pi);
}
```

Sources:

http://www.cs.uiuc.edu/homes/snir/PPP/
https://computing.llnl.gov/tutorials/openMP/
MIT Course:
http://ocw.mit.edu/courses/earth-atmospheric-and-planetary-sciences/12-950-parallel-programming-for-multicore-machines-using-openmp-and-mpi-january-iap-2010/
http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf
http://openmp.org/wp/
https://www.cac.cornell.edu/VW/OpenMP/threads.aspx
https://source.ggy.bris.ac.uk/wiki/OpenMP