Shared Memory and OpenMP

Alan Sussman, Department of Computer Science
Announcements

• Assignment 1 is due on: March 7, 11:59 pm ET
  • Good-faith attempt of each assignment is required
  • Questions?

• Quiz 1 was due today, Feb. 23, at 11AM
  • You should be able to see your score, and correct answers, in ELMS
  • Questions?
Shared memory programming

• All entities (threads) have access to the entire address space
• Threads “communicate” or exchange data by sharing variables
• User has to manage data conflicts
OpenMP

• OpenMP is an example of a shared memory programming model

• Provides on-node parallelization

• Meant for certain kinds of programs/computational kernels
  • Ones that use arrays and loops

• Potentially easy to implement an application in parallel with small code changes
OpenMP

• OpenMP is a language extension and library that enables parallelizing C/C++/Fortran code

• Programmer uses compiler directives and library routines to indicate parallel regions in the code and how to parallelize them

• Compiler converts code to multi-threaded code

• Fork/join model of parallelism
Fork-join parallelism

- Single flow of control
- Master thread spawns worker threads

https://en.wikipedia.org/wiki/OpenMP
Race conditions when threads interact

• Unintended sharing of variables can lead to race conditions

• Race condition: program outcome depends on the scheduling order of threads
  • Defined as one or more threads accessing a memory location with at least one of them performing a write, and without proper synchronization

• How can we prevent data races?
  • Use synchronization
  • Change how data is stored
OpenMP pragmas

-Pragma: a compiler directive in C or C++
- Mechanism to communicate with the compiler
- Compiler may ignore pragmas

```
#pragma omp construct [clause [clause] ... ]
```
Hello World in OpenMP

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

int main(void)
{
    #pragma omp parallel
    printf("Hello, world.\n");
    return 0;
}
```

- **Compiling:** `gcc -fopenmp hello.c -o hello`
- **Setting number of threads:** `export OMP_NUM_THREADS=2`
Parallel for

- Directs the compiler that the immediately following for loop should be executed in parallel

```c
#pragma omp parallel for [clause [clause] ... ]
for (i = init; test_expression; increment_expression) {
    ...
    do work
    ...
}
```
Parallel for example

```c
int main(int argc, char **argv)
{
    int a[100000];

    #pragma omp parallel for
    for (int i = 0; i < 100000; i++) {
        a[i] = 2 * i;
    }

    return 0;
}
```
Parallel for execution

• Master thread creates worker threads
• All threads divide iterations of the loop among themselves
Number of threads

• Use environment variable

```c
export OMP_NUM_THREADS=X
```

• Use `void omp_set_num_threads(int num_threads)`
  • Set the number of OpenMP threads to be used in parallel regions

• `int omp_get_num_procs(void);`
  • Returns the number of available processors/cores
  • Can be used to decide the number of threads to create
Data sharing defaults

• Most variables are shared by default
• Global variables are shared
• Exception: loop index variables are private by default
• Stack variables in function calls from parallel regions are also private to each thread (thread-private)
saxpy (single precision a*x+y) example

```c
#pragma omp parallel for
for (int i = 0; i < n; i++) {
    z[i] = a * x[i] + y[i];
}
```
Overriding defaults using clauses

• Specify how data is shared between threads executing a parallel region

• private(list)

• shared(list)

• default(shared | none)

• reduction(operator: list)

• firstprivate(list)

• lastprivate(list)

https://www.openmp.org/spec-html/5.0/openmp-su106.html#x139-5540002.19.4
private clause

• Each thread has its own copy of the variables in the list
• Private variables are uninitialized when a thread starts
• The value of a private variable is unavailable to the master thread after the parallel region has been executed
default clause

- Determines the data sharing attributes for variables for which this would be implicitly determined otherwise
Anything wrong with this example?

val = 5;

#pragma omp parallel for private(val)
for (int i = 0; i < n; i++) {
    ... = val + 1;
}

The value of val will not be available to threads inside the loop
Anything wrong with this example?

```c
#pragma omp parallel for private(val)
for (int i = 0; i < n; i++) {
    val = i + 1;
}
printf(“%d\n”, val);
```

The value of val will not be available to the master thread outside the loop.
firstprivate clause

- Initializes each thread’s private copy to the value of the master thread’s copy, on entry to the parallel section

```c
val = 5;

#pragma omp parallel for firstprivate(val)
for (int i = 0; i < n; i++) {
    ... = val + 1;
}
```
lastprivate clause

• Writes the value belonging to the thread that executed the last iteration of the loop to the master’s copy

• Last iteration determined by sequential order

```c
#pragma omp parallel for lastprivate(val)
for (int i = 0; i < n; i++) {
    val = i + 1;
}
printf("%d\n", val);
```
reduction(operator: list) clause

• Reduce values across private copies of a variable

• Operators: +, -, *, &, |, ^, &&, ||, max, min

#pragma omp parallel for reduction(+: val)
for (int i = 0; i < n; i++) {
    val += i;
}

printf("%d\n", val);

https://www.openmp.org/spec-html/5.0/openmpsu107.html#x140-5800002.19.5
Loop scheduling

- Assignment of loop iterations to different worker threads
- Default schedule tries to balance iterations among threads
- User-specified schedules are also available
User-specified loop scheduling

• Schedule clause

```
schedule (type[, chunk])
```

• type: static, dynamic, guided, runtime

• static: iterations divided as evenly as possible (#iterations/#threads)
  • chunk < #iterations/#threads can be used to interleave threads

• dynamic: assign a chunk size block to each thread
  • When a thread is finished, it retrieves the next block from an internal work queue
  • Default chunk size = 1
Other schedules

• guided: similar to dynamic but start with a large chunk size and gradually decrease it for handling load imbalance between iterations

• auto: scheduling delegated to the compiler

• runtime: use the OMP_SCHEDULE environment variable

Calculate the value of \( \pi = \int_0^1 \frac{4}{1 + x^2} \)

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

    n = 10000;
    h   = 1.0 / (double) n;
    sum = 0.0;

    for (i = 1; i <= n; i += 1) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x * x));
    }
    pi = h * sum;
    ...
}
```
Calculate the value of \( \pi = \int_0^1 \frac{4}{1 + x^2} \)

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

    n = 10000;
    h = 1.0 / (double) n;
    sum = 0.0;

    #pragma omp parallel for firstprivate(h) private(x) reduction(+: sum)
    for (i = 1; i <= n; i += 1) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x * x));
    }
    pi = h * sum;
    ...
}
```
Parallel region

• All threads execute the structured block

```c
#pragma omp parallel [clause [clause] ... ]
  structured block
```

• Number of threads can be specified just like the parallel for directive
Synchronization

• Concurrent access to shared data may result in inconsistencies
• Use mutual exclusion to avoid that
• critical directive
• atomic directive
• Library lock routines

critical directive

• Specifies that the code is only to be executed by one thread at a time

```c
#pragma omp critical [(name)]
structured block
```
atomic directive

- Specifies that a memory location should be updated atomically

```c
#pragma omp atomic
expression
```
GPGPUs

- GPGPU: General Purpose Graphical Processing Unit
- Many slower cores

OpenMP on GPUs

• **target**: run on accelerator / device

```c
#pragma omp target teams distribute parallel for
for (int i = 0; i < n; i++) {
    z[i] = a * x[i] + y[i];
}
```

• **teams distribute**: creates a team of worker threads and distributes work amongst them