Shared Memory
&
OpenMP

Michial Green II
-
Contra Costa College
Shared Memory Architecture
Threads

- Short for “Thread of execution”
- Threads execute the statements within a program.
- Threads are usually created by a process
Distributed:

- Multiple instances of program created
- Each process has its own local memory
- One process cannot see anything local to another process
- May not even be on same node
- Memory communicated via message passing
Shared

- Only one instance runs
- Threads are created as needed within
- Each thread has local memory
- Each thread can be run by separate CPUs or
- Each thread can run on the same CPU
- Local memory can be communicated by updates to main memory
The Round Table

- Round table with one worker
  worker has:
  - Pencil
  - paper
  - calculator
  - public checkbook to balance
The Round Table

- Worker calls in 2 more helpers.
- The helpers get:
  - pencil
  - paper
  - calculator

Helpers may also look at the public checkbook!
The Round Table

- No one can see what you write on your own paper.
- But you can change the public checkbook at any time so everyone can see.
The Round Table

• Helpers finish their jobs, and then are dismissed.

• Before leaving they record their results in the public checkbook.
Fork and Join

Notice the areas with just the original master thread.
Amdahl's Law

\[ \text{SpeedUp} = \frac{1}{(1-F)+\frac{F}{N}} \]

- \( F \) = the parallelizable sections of serial code
- \( N \) = number of processors
Pseudo Parallelism?

- What if you have two threads on one processor?
- What about your operating system?
A processor can only do one thing at a time.
So it switches between jobs quickly.
Caveats

The word we hear so much:

Overhead

- Creating threads
- Communicating between threads
- Managing Memory Access
Thread Creation Overhead

- Anything else running?
- Resources present?
- Are there enough physical processors?
  - These can be expensive
  - Expense rises as the number of threads required rises
- No work until workers arrive
Thread Communication Overhead

- Do they need to talk?
- Structure scenarios for communicating
- Choosing reliable communications

- Remember time spent talking is not time spent working
Shared Memory Concerns

- Shared literally means “Shared”
  A: What's mine is mine, I do what I please
  B: What's yours is mine, I do what I please
Managing Memory Access

- Writing at the same time
- Only reading valid data
- Waiting..... waiting..... waiting....
OpenMP

• Don't confuse with OpenMPI
• MP means “multiprocessing”

• Unlike MPI, some OpenMP capabilities require compiler support, and aren't linked into your executable.
Compiling OpenMP Programs

- Call your openmp compliant compiler as normal.

  Example:

  gcc filename.c

- Add your appropriate compiler flag to enable openMP.

  GNU = -fopenmp ; Intel = -Qopenmp ; etc.

  check → openmp.org/wp/openmp-compilers/

- Gcc -g -o myprogram -fopenmp filename.c -lm
# Going Parallel

- `#include <stdio.h>`
- `using namespace std;`
- `const int thread_count = 2;`
- `void Hello(void)`
  ```c
  {
  printf("Oh well helllooooooo!\n");
  }
  ```
- `int main(void)`
  ```c
  {
  # pragma omp parallel num_threads(thread_count)
  Hello();
  return 0;
  }
  ```
Setting the number of threads

- `export OMP_NUM_THREADS = #`
- `setenv OMP_NUM_THREADS #`

- `# pragma omp parallel num_threads(thread_count)`

changes the number of threads arbitrarily
The Parallel Construct - C

• # pragma omp parallel
  – Next structured block runs in parallel
    the number of threads used are determined by OMP_NUM_THREADS
    or
    THE NUMBER OF AVAILABLE PROCESSORS
  – OMP_NUM_PROCS

• Clauses can be added to refine your approach
Parallel clauses

• num_threads(int)
  – Use: #pragma omp parallel num_threads(15)
    specifies the number of threads to run in block

• private(variablename1, variablename2,...,)
  – Use: #pragma omp parallel private(i, my_rank)
    grants all threads in block a local version of the
    specified variables, that they can manipulate
Parallel for construct

- `# pragma omp parallel for`
  
  ```c
  for(int i=0; i<limit; i++){sum+=i;}
  ```

- Assuming that sum was defined before
  - Sum is visible and shared to all threads
  - No while loops or do while loops
Parallel for Construct

• Trouble comes at the statement:
  – sum+=i;
  because the variable is shared.

• Who will update the variable and when?
Critical Sections

• You don't want multiple updates
• Make the area mutually exclusive

We could use:

```c
#pragma omp critical
{sum+=i;}
```
Critical Sections

• Bonus:
  – No need to worry about interrupts

• Caveats:
  – This area is SERIAL BY NATURE
  – Performance hit
Parallel for Construct

- We could make the variable private
  - # pragma omp parallel for private(sum)

- Now its up to us to gather the values from each thread
- This is a lot of work.
The Reduction Clause

Int sum = 0, limit = 10000000;
#pragma omp parallel for num_threads(8) \ 
  reduction(+: sum)
for(int i=0; i<limit; i++){sum += i;}

• sum+=i  remains parallel

• I no longer have to coordinate a reduction manually
Runtime Libraries

• #include <omp.h>

• Who am I, How many of us are there?
  Int omp_get_num_threads();
  int omp_get_thread_num();
#include <cstdio>
#include <omp.h>

main () {

    int nthreads, tid;

    /* Fork a team of threads with each thread having a private tid variable */

    #pragma omp parallel private(tid)
    {

        /* Obtain and print thread id */
        tid = omp_get_thread_num();

        printf("Hello World from thread = %d\n", tid);
        printf("Hello, I may appear in what seems a random spot!\n");

        /* Only master thread does this */

        if (tid == 0)
        {
            nthreads = omp_get_num_threads();
            printf("Number of threads = %d\n", nthreads);
        }
    } /* All threads join master thread and terminate */

    return 0;
}
OpenMP – Area Under the curve

• Setup OpenMP (-fopenmp, setenv)
• Defining your constants
• Determining your number of rectangles
• Fork a team to create chunks of rectangles
• Calculate areas
• Remember to use reduction on for construct
• Output result and timing