Introduction to High Performance Computing

Alejandro Cárdenas-Avendaño

OpenMP

OpenMP quick review

In code:

- In C++, you add lines starting with **#pragma omp**
  - This parallelizes the subsequent code block.

When compiling:

- To turn on OpenMP support in g++, add the **-fopenmp** flag to the compilation and link commands

When running:

- The environment variable **OMP_NUM_THREADS** determines how many threads will be started in an OpenMP parallel block.

Example-OpenMP V4.2

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

int main ()
{
  int   i, j, n;
  long double a[3000], b[3000], result;

  /* Some initializations */
  n = 3000;

  result = 0.0;
  for (i=0; i < n; i++)
    #pragma omp parallel for default(none) private(j,a,b) shared (i,n,result)
    for (j=0;j<n;j++)
          {
        a[i] = log(i+1)*pow(cos(i)*tan(j),2);
        b[i] = sqrt(j)*pow(sin(i)*tan(i),2);
          #pragma omp atomic update
          result += pow((a[i]*a[i] + b[i]*b[i]),0.5);
          }

  printf("Final result= %Lf\n",result);
}
```

OpenMP atomic construct:

- Most hardware has support for atomic instructions (indivisible so cannot get interrupted)
- Small subset, but load/add/store usually in it.
- Not as general as critical
- Much lower overhead.
- #pragma omp atomic [read|write|update|capture]

Example-OpenMP V5

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

int main ()
{
  int   i, j, n;
  long double a[3000], b[3000], result;

  /* Some initializations */
  n = 3000;

  result = 0.0;
  for (i=0; i < n; i++)
    #pragma omp parallel for default(none) private(j,a,b) shared (i,n) reduction(+:result)
    for (j=0;j<n;j++)
          {
        a[i] = log(i+1)*pow(cos(i)*tan(j),2);
        b[i] = sqrt(j)*pow(sin(i)*tan(i),2);
          result += pow((a[i]*a[i] + b[i]*b[i]),0.5);
          }

  printf("Final result= %Lf\n",result);
}
```

```
$ g++-8 -O0 -o omploopvec5
omploopvec5.cpp -fopenmp
$ export OMP_NUM_THREADS=2
$ ./omploopvec5
Final result= 128235433739.279927
real   0m1.406s
```

```
Serial One:
Final result= 128235433739.279927
real   0m2.387s
```

```
$ export OMP_NUM_THREADS=4
$ ./omploopvec5
Final result= 128235433739.279937
real   0m0.940s
```

```
$ export OMP_NUM_THREADS=1
$ ./omploopvec5
Final result= 128235433739.279937
real   0m2.626s
```

OpenMP Reduction Operations

- What we did is quite common, taking a bunch of data and summing it to one value: **reduction**
- OpenMP supports this using **reduction variables**.
- When declaring a variables as reduction variables, private copies are made (much as for private variables), which are combined at the end of a parallel region through some operation (*+, *, min, max*).

Load Balancing

Load Balancing in OpenMP

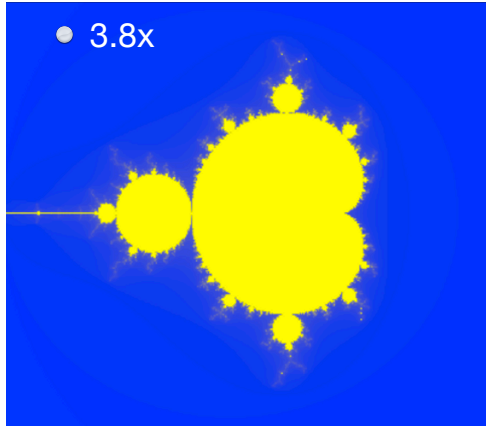
- So far every iteration of the loop had the same amount of work.
- Not always the case.
- Sometimes cannot predict beforehand how unbalanced the problem is

OpenMP has work sharing constructs that allow you do statically or dynamically balance the load.

Example - Mandelbrot Set

- Example of non-balanced problem.
- Based on a mapping in complex plane:
  - $b_{n+1} = b_n^2 + a$
- Mandelbrot set is boundary between diverging points *a* ($b_0 = 0 \Rightarrow \|b_\infty\| = \infty$)and converging points ($\|b_\infty\| = \infty$).

  Note: if $\|b_n\| > 2$, point diverges.
- Calculation:
  - iterate for each point a in square, see if $\|b_n\| > 2$.
  - n<nmax, then blue, else yellow.
- On the outside points diverge quickly. Inside points: lots of work.



Scheduling constructs in OpenMP

- Default: each thread gets a big consecutive chunk of the loop. Often better to give each thread many smaller interleaved chunks.
- Can add schedule clause to omp for to change work sharing.
- We can decide either at compile-time (static schedule) or run-time (dynamic schedule) how work will be split.
- ***#pragma omp parallel for schedule(static, m)*** gives *m* consecutive loop elements to each thread instead of a big chunk.
- With '**schedule(dynamic, m)'**, each thread will work through m loop elements, then go to the OpenMP run-time system and ask for more.
- Load balancing (possibly) better with dynamic, but larger overhead than with static.

Scheduling constructs in OpenMP

Static

```
schedule(static):
****************
                ****************
                                ****************
                                                ****************
```

```
schedule(static, 4):
****            ****            ****            ****
    ****            ****            ****            ****
        ****            ****            ****            ****
            ****            ****            ****            ****
```

```
schedule(static, 8):
********                        ********
        ********                        ********
                ********                        ********
                        ********                        ********
```

Dynamic

```
schedule(dynamic):
*   ** **  * * *  *      *  *    **   *  *  * *       *  *   *
  *       *     *    * *     * *   *    *        * *   *    *
 *       *    *     * *   *   *     *  *       *  *  *  *  *   *
   *  *     *    * *    * *    *    *    ** *  *   *     *   *
```

```
schedule(dynamic, 1):
    *    *    *       *   *    * *  *  *         *  * *  * *
*  *  *   * *     *  * * *    * *      *   ***  *   *         *
 *   *  *  *  *    ** *    *      *  *  * *   *  *   *   *
  *    *     *  **        *  * *    *          *  *    *  * *  *
```

```
schedule(dynamic, 4):
            ****                    ****                    ****
****            ****    ****            ****        ****
    ****            ****    ****            ****        ****
        ****                    ****            ****
```

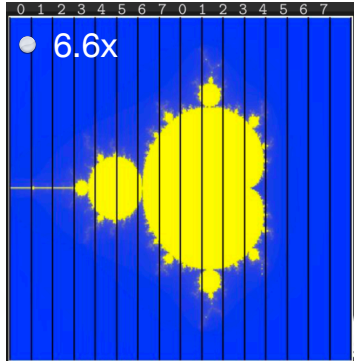
```
schedule(dynamic, 8):
                ********                                        ********
                        ********        ********
********                        ********        ********
        ********
```

- We parallelized a for loop with **64** iterations and we used **four threads** to parallelize the for loop.
- Each **row** of stars in the examples represents a thread. Each **column** represents an iteration.

Mandelbrot: Schedule

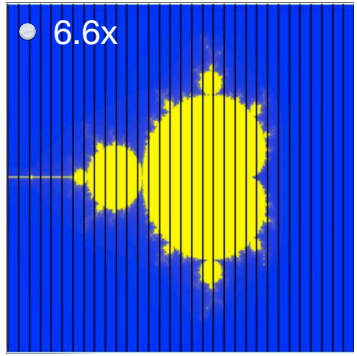
- Can change the chunk size different from ~ N/nthreads
- In this case, more columns – work distributed a bit better.
- Now, for instance, thread 7 gets both a big work chunk and a little one.

```
#pragma omp parallel for schedule(static,25)
```



- Break up into many pieces and hand them to threads when they are ready.
- Dynamic scheduling.
- Increases overhead, decreases idling threads. Can also choose chunk size.

```
#pragma omp parallel for schedule(dynamic)
```



More. . .

There are many more features to OpenMP not discussed here.

- Collapsed loops
- Tasks
- Tasks with dependencies
- Nested OpenMP parallelism
- Locks
- SIMD
- Thread affinities
- Compute devices (e.g. graphics cards, Phi)

Parallelization concepts and OpenMP

Parallelization concepts

A few lectures ago, we learned about different approached to parallelism, but this week, and next week, we'll talk mostly about technical details.

Let's step back a moment and see how these fit together.

Let's consider:

1. Data parallel computations
2. Task parallel computations
3. Others

1. Data parallel computations

- In these cases, the same action is performed on different data.
- Many, if not most, scientific computations falls in this category.
- For instance, the diffusion of robots on a ring had the same computation for each grid point in each time step.
- If the data fits in memory, OpenMP is a great option to use to parallelize such codes.
- This isn't quite 'embarrisingly parallel' because each timestep depends on the results of the previous time step, i.e., there are data dependencies, just not within a single time step.
- If the data does not fit in memory, or the number of cores per node is insufficient, we can turn to distributed computing with MPI (next lectures)

2. Task parallel

- Different things done on same or different data.
- This is the case with several independent jobs, but still **requires scheduling** to get proper load balance. E.g.
  - Workloads like the ones we use gnu parallel for,
  - or, on a, larger scale, the job scheduler for the computing clusters.
- Often, the tasks are not independent, but part of a pipeline or workflow.
- OpenMP has this capability to scheduling tasks with dependencies, but we won't have time to cover it.

Other cases

- Reduction computations

Think of summing up numbers to a single number. The input data is all independent, but the output is not.

- Both OpenMP and MPI have the capability to do reductions.
- Broadcast or recursive computations.

Here the input data has dependencies, or a single source, but the output is not. Eg random number generators, reading data from (non parallel) file systems.

If simple copy, it's like a reverse reduction. If OpenMP, you just use a shared variable. In MPI, we'll need to broadcast explicitly.

If not, need **new algorithm**. E.g. How would you parallelize a random number generator?

References

- SciNet Education (https://support.scinet.utoronto.ca/education/help/help_about.php)
- International HPC Summer School 2018 Lectures
- ARCHER UK National Supercomputing Service Training Courses (http://www.archer.ac.uk/training/)
- Muna, D & Price-Whelan, A. SciCoder Workshop. scicoder.org
- Peter Norvig, “What to demand from a Scientific Computing Language”, Mathematical Sciences Research Institute, 2010.