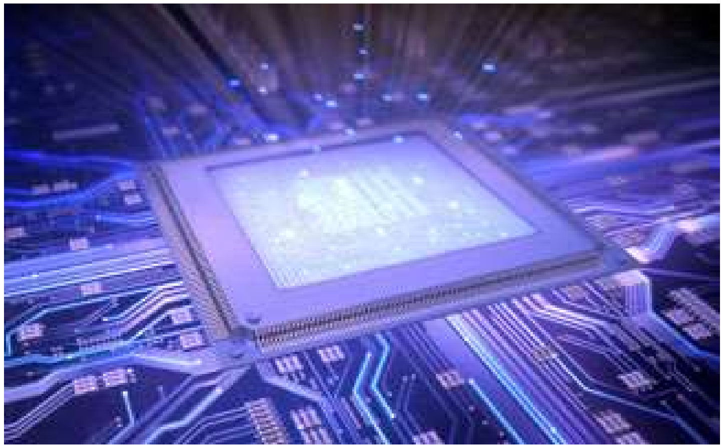
Parallel Computing

OpenMP – Last Touch

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some slides from here are adopted from:
- Yun (Helen) He and Chris Ding Lawrence Berkeley National Laboratory
- Yao-Yuan Chuang



Performance

- Easy to write OpenMP but hard to write an efficient program!
- 5 main causes of poor performance:
  - Sequential code
  - Communication
  - Load imbalance
  - Synchronisation
  - Compiler (non-)optimisation.

Sequential code

- Amdahl's law: Limits performance.
- Need to find ways of parallelising it!
- In OpenMP, all code outside of parallel regions and inside MASTER, SINGLE and CRITICAL directives is sequential.
  - This code should be as small as possible.

Communication

- On Shared memory machines, communication = increased memory access costs.
  - It takes longer to access data in main memory or another processor's cache than it does from local cache.
- Memory accesses are expensive!
- Unlike message passing, communication is spread throughout the program.
  - Much harder to analyse and monitor.

Caches and coherency

- Shared memory programming assumes that a shared variable has a unique value at a given time.
- Caching means that multiple copies of a memory location may exist in the hardware.
- To avoid two processors caching different values of the same memory location, caches must be kept *coherent*.
- Coherency operations are usually performed on the cache lines in the level of cache closest to the shared inclusive cache/memory

False sharing

- Cache lines consist of several words of data.
- What happens when two processors are both writing to different words on the same cache line?
  - Each write will invalidate the other processors copy.
  - Lots of remote memory accesses.
- Symptoms:
  - Poor speedup
  - High, non-deterministic numbers of cache misses.
  - Mild, non-deterministic, unexpected load imbalance.

Matrix-vector multiplication

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

$$\begin{array}{|c|c|c|c|}
\hline
a_{00} & a_{01} & \cdots & a_{0,n-1} \\
\hline
a_{10} & a_{11} & \cdots & a_{1,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{i0} & a_{i1} & \cdots & a_{i,n-1} \\
\hline
\vdots & \vdots & & \vdots \\
\hline
a_{m-1,0} & a_{m-1,1} & \cdots & a_{m-1,n-1} \\
\hline
\end{array}
\quad
\begin{array}{|c|}
\hline
x_0 \\
\hline
x_1 \\
\hline
\vdots \\
\hline
x_{n-1} \\
\hline
\end{array}
\quad = \quad
\begin{array}{|c|}
\hline
y_0 \\
\hline
y_1 \\
\hline
\vdots \\
\hline
y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} \\
\hline
\vdots \\
\hline
y_{m-1} \\
\hline
\end{array}$$

```
for (i = 0; i < m; i++) {
   y[i] = 0.0;
   for (j = 0; j < n; j++)
      y[i] += A[i][j]*x[j];
}
```

Matrix-vector multiplication

```
#  pragma omp parallel for num_threads(thread_count) \
      default(none) private(i, j) shared(A, x, y, m, n)
   for (i = 0; i < m; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i][j]*x[j];
   }
```

Run-times and efficiencies
of matrix-vector multiplication
(times are in seconds)

	Matrix Dimension m x n					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Matrix-vector multiplication

```
#  pragma omp parallel for num_threads(thread_count) \
      default(none) private(i, j) shared(A, x, y, m, n)
   for (i = 0; i < m; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i][j]*x[j];
   }
```

This worse performance, relative to 8000x8000 is mainly due to cache performance

Even though the number of operations is the same!

	Matrix Dimension m x n					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
Threads	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Matrix-vector multiplication

```
#   pragma omp parallel for num_threads(thread_count) \
       default(none) private(i, j) shared(A, x, y, m, n)
    for (i = 0; i < m; i++) {
       y[i] = 0.0;
       for (j = 0; j < n; j++)
          y[i] += A[i][j]*x[j];
    }
```

Far more write-misses than the other two.

Threads	Matrix Dimension m x n					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Matrix-vector multiplication

```
#  pragma omp parallel for num_threads(thread_count) \
      default(none) private(i, j) shared(A, x, y, m, n)
   for (i = 0; i < m; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i][j]*x[j];
   }
```

Far more read-misses than the other two.

Threads	Matrix Dimension m x n					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Data affinity

- Data is cached on the processors which access it.
  - Must reuse cached data as much as possible.
- Write code with good *data affinity*:
  - Ensure the same thread accesses the same subset of program data as much as possible.
- Try to make these subsets large, contiguous chunks of data.
  - Will avoid false sharing and other problems.
- The manner in which the memory is accessed by individual threads has a major influence on performance
  - If each thread accesses a distinct portion of data consistently through the program, the threads will probably make excellent use of memory.
  - This improvement includes good use of thread-local cache.

Load imbalance

- Load imbalance can arise from both communication and computation.
- Worth experimenting with different scheduling options
  - **runtime** clause is handy here
- If none are appropriate, may be best to do your own scheduling!

Synchronisation

- Barriers can be very expensive
- Avoid barriers via:
  - *Careful* use of the NOWAIT clause.
  - Parallelise at the outermost level possible.
    - May require re-ordering of loops /indices.
  - Choice of CRITICAL / ATOMIC / lock routines may impact performance.

Compiler (non-)optimisation

- Sometimes the addition of parallel directives can inhibit the compiler from performing sequential optimisations.

- Symptoms:
  - 1-thread parallel code has longer execution and higher instruction count than sequential code.

- Can sometimes be cured by making shared data private, or local to a routine.

Performance Tuning

- My code is giving me poor speedup. I don't know why. What do I do now?
- A:
  - Say "this machine/language is a heap of junk"
  - Give up and go back to your laptop
- B:
  - Try to classify and localise the sources of overhead.
    - What type of problem is it and where in the code does it occur
  - Fix problems that are responsible for large overheads first.
  - Iterate

Performance Tuning: Timing the OpenMP Performance

- A standard practice is to use a standard operating system command.
- For example

$time ./a.out

  - The “real”, “user”, and “system” times are then printed after the program has finished execution.
  - For example

```
$ time .program.exe
real    5.4   Elapsed time
user    3.2 ┐
            ├ CPU time
sys     1.0 ┘
```

  - These three numbers can be used to get initial information about the performance.

Performance Tuning: Timing the OpenMP Performance

- A common cause for the difference between the wall-clock time of 5.4 seconds and the CPU time is a processor sharing too high a load on the system.
- If sufficient processors are available (i.e., not being used by other users), your elapsed time should be less than the CPU time.
- The omp_get_wtime() function provided by OpenMP is useful for measuring the elapsed time of blocks of source code.

Performance Tuning: Avoid Parallel Regions in Inner Loop

- Another common technique to improve the performance is to move parallel regions out of the innermost loops.
- Otherwise, we repeatedly incur the overheads of the parallel construct.
- By moving the parallel construct outside of the loop nest, the parallel construct overheads are minimized.

Performance Tuning: Overlapping Computation and I/O

- This helps avoid having all but one processors wait while the I/O is handled.
- A general rule for MIMD parallelism in general is to overlap computation and communications so that the total time taken is less that the sum of the times to do each of these.
- However, this general guideline might not always be possible.

Hybrid OpenMP and MPI

MPI vs. OpenMP

- **Pure MPI Pro:**
  - Portable to distributed and shared memory machines.
  - Scales beyond one node
  - No race condition problem
- **Pure MPI Con:**
  - Difficult to develop and debug
  - High latency, low bandwidth
  - Explicit communication
  - Large granularity
  - Difficult load balancing

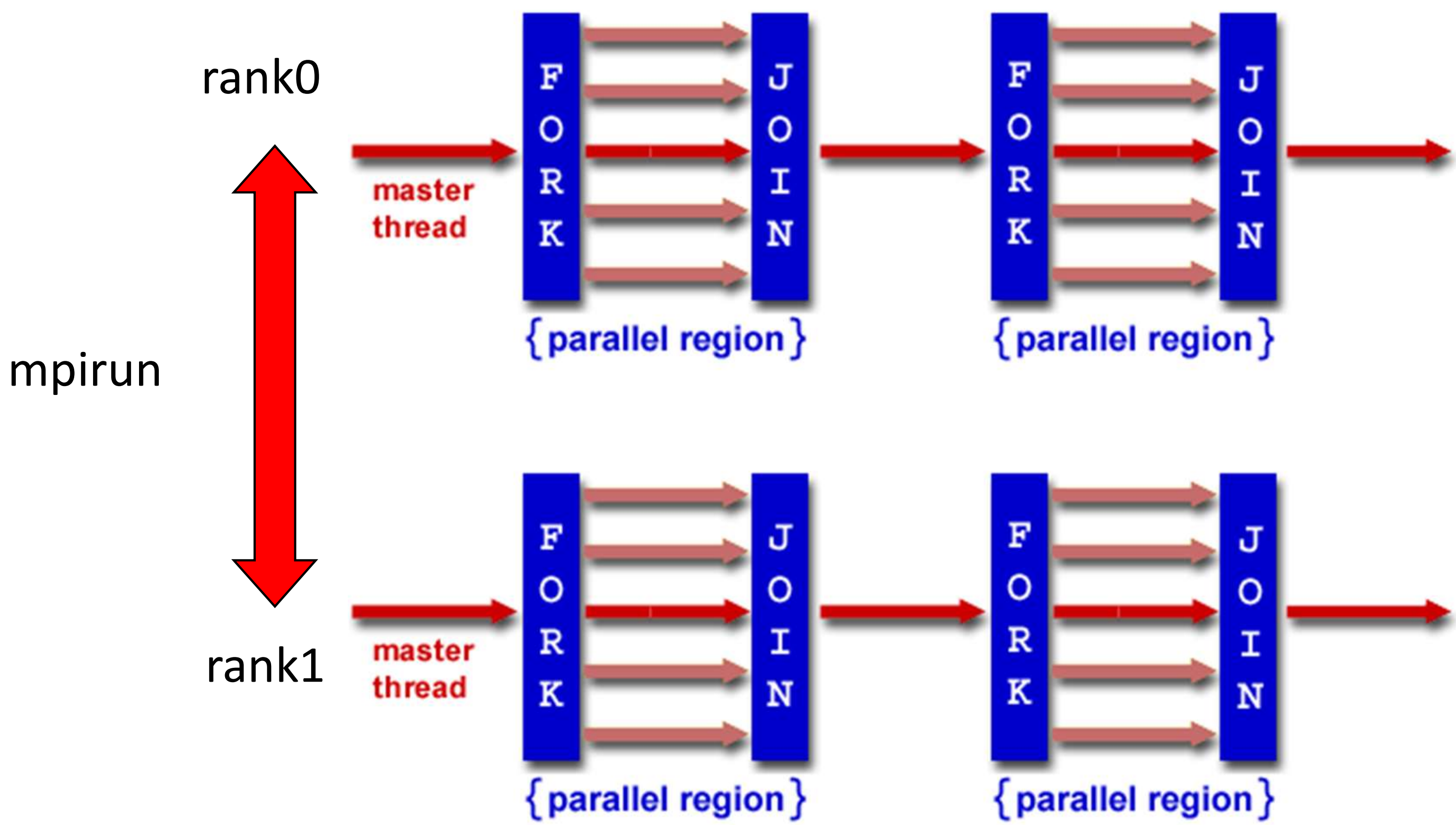
- **Pure OpenMP Pro:**
  - Easy to implement parallelism
  - Low latency, high bandwidth
  - Implicit Communication
  - Dynamic load balancing
- **Pure OpenMP Con:**
  - Only on shared memory machines
  - Scale within one node
  - Possible race condition problem
  - No specific thread order

Why Hybrid?

- Hybrid MPI/OpenMP paradigm is the software trend for clusters of SMP architectures, supercomputers (although GPUs are usually also used here), … .
- Elegant in concept and architecture: using MPI across nodes and OpenMP within nodes. Good usage of shared memory system resource (memory, latency, and bandwidth).
- Avoids the extra communication overhead with MPI within node.
- OpenMP adds fine granularity and allows increased and/or dynamic load balancing.
- Some problems have two-level parallelism naturally.
- Could have better scalability than both pure MPI and pure OpenMP.

Hybrid Parallelization Strategies

- **From sequential code:** decompose with MPI first, then add OpenMP.
- Simplest and least error-prone way is:
  - Use MPI outside parallel region.
  - Allow only master thread to communicate between MPI tasks.



omphello.c

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

int main(int argc, char *argv[]) {
  int iam = 0, np = 1;

  #pragma omp parallel private(iam, np)
  {
      np = omp_get_num_threads();
      iam = omp_get_thread_num();
     printf("Hello from thread %d out of %d\n", iam, np);
  }
}
```

mpihello.c

```
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);

  printf("Process %d on %s out of %d\n", rank,
          processor_name, numprocs);

  MPI_Finalize();
}
```

Hybrid MPI + OpenMP: mixhello.c

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

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int iam = 0, np = 1;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);

  #pragma omp parallel default(shared) private(iam, np)
  {
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hello from thread %d out of %d from process %d out of %d on %s\n",
           iam, np, rank, numprocs, processor_name);
  }

  MPI_Finalize();
}
```

Compile and Execution

```
mpicc -fopenmp  mixhello.c

mpiexec -n x ./a.out
```

Conclusions

- Always keep in mind the 5 reasons of poor performance

If:

- you have a machine with several nodes
- The problem at hand has two levels of parallelism

Then:

- consider hybrid OpenMP + MPI