Understanding Parallel Applications

Xiaoxu Guan

High Performance Computing, LSU

May 30, 2017
Overview

- Parallel applications and programming on shared-memory and distributed-memory machines
- We follow the parallelism methodology from top to bottom
- Heterogeneous and homogeneous systems
- Models of parallel computing
- Multi-node level: MPI
- Single-node level: MPI/OpenMP
- Hybrid model: MPI + OpenMP
- Compute-bound and memory-bound applications
- Socket and Processor level: NUMA and affinity
- Core level: SIMD (pipeline and vectorization)
- Summary
Parallel computing

- Parallel computing means a lot;
- It almost covers everything in the HPC community;
- Many programming languages support parallel computing:
  - Fortran, C, and C++;
  - Matlab, Mathematica;
  - Python, R, Java, Hadoop, . . . ;
  - Parallel tools: GNU parallel, parallel shells, . . . ;
- They support parallel computing at very different levels through a variety of mechanisms;
- From embarrassment computing to parallel computing that needs extensively data communication;
- Beyond the language level: parallel filesystems: lustre, and the fabric network: Ethernet and Infiniband;
Parallel computing

- Why parallel or concurrency computing?
- Goes beyond the single-core capability (memory and flops per unit time), and therefore increases performance;
- **Reduces** wall-clock time, and saves energy;
- Finishes those **impossible** tasks in my lifetime;
- Handles larger and larger-scale problems;

Consider a production MPI job:

(a) Runs on 2,500 CPU cores
(b) Finishes in \( \approx 40 \) hours (wall-clock time)
(c) Charged CPU hours are \( 2,500 \times 40 = 0.1 \) M SUs
(d) It is about \( \frac{100,000}{24/365} \approx 11 \) years on 1 CPU core!

- Is parallel computing really necessary?
Parallel computing

- Why parallel or concurrency computing?
- Goes beyond the single-core capability (memory and flops per unit time), and therefore increases performance;
- **Reduces** wall-clock time, and saves energy;
- Finishes those **impossible** tasks in my lifetime;
- Handles larger and larger-scale problems;
- **There is no free lunch, however!**
- Different techniques other than serial coding are needed;
- Effective parallel algorithms in terms of performance;
- Increasing flops per unit time or throughput is one of our endless goals in the HPC community;
- **Think in parallel**;
- Start parallel programming as soon as possible;
Parallel computing

- *Our goal* here is to “Understanding Parallel Applications”;  
- This is no simple and easy way to master parallel computing;  
- Evolving software stack and architecture complexity;  
- HPC is one of essential tools in my research;  
- And *my goal* is to advance scientific progress;  
- I’m not the code developer, *what can I do*?  
- I have been a programmer for years, *is there anything else I should be concerned*?  
- Besides, “Understanding Parallel Applications” requires basic knowledge of the *hardware*;  
- Provide you a concrete introduction to *parallel computing* and *parallel architecture*;  
- Focus on *performance* and *efficiency* analysis;
Parallel computing

- Parallel computing can be viewed from different ways;
- **Flynn**’s taxonomy: *execution* models to achieve parallelism
  - SISD: single instruction, single data;
  - MISD: multiple instruction, single data;
  - SIMD: single instruction, multiple data;
  - MIMD: multiple instructions, multiple data (or tasks);
  - SPMD: single program, multiple data;
- **Memory** access and *programming* model:
  - **Shared memory**: a set of cores that can access the common and shared physical memory space;
  - **Distributed memory**: No direct and remote access to the memory assigned to other processes;
  - **Hybrid**: they are not exclusive;
Parallel computing

- Parallel computing can be viewed from different ways;
- **Flynn**’s taxonomy: *execution* models to achieve parallelism
  - SISD: single instruction, single data;
  - MISD: multiple instruction, single data;
  - SIMD: single instruction, multiple data;
  - MIMD: multiple instructions, multiple data (or tasks);
  - SPMD: single program, multiple data;

- Model of workload breakup: data and task parallelism

```plaintext
1 for i from imin to imax, do
2 c(i) = a(i) + b(i)
3 end do
```

Data parallelism

```plaintext
1 { for c(i) = a(i) + b(i) }
2 { for d(j) = sin(a(j)) }
```

Task parallelism

- All the levels of **parallelism** found on a production cluster;
Multi-node level parallelism

MPI applications on distributed-memory systems
Multi-node level parallelism

- On a **distributed-memory** system:
  - Each node has its own **local** memory;
  - There is **no** physically **global** memory;
  - **Message passing**: send/receive message through network;

- **MPI** (Message Passing Interface) is a default programming model on DM systems in HPC user community;
- **MPI-1** started in 1992. The current standard is **MPI 3.x**.
- MPI standard is **not** an IEEE or ISO standard, but a **de facto** standard in HPC world;
- Don’t be confused between MPI implementations and MPI standard;
- **MPICH, MVAPICH2, OpenMPI, Intel MPI, ...**;
Multi-node level parallelism

- Requirements for parallel computing;
- How does MPI meet these requirements?
  - Specify parallel execution – single program on multiple data (SPMD) and tasks;
  - Data communication – two- and one-side communication (explicit or implicit message passing);
  - Synchronization – synchronization functions;

1. Expose and then express parallelism;
2. Must exactly know the data that need to be transferred;
3. Management of data transfer;
4. Manually partition and decompose;
5. Difficult to program and debug (deadlocks, . . .);
Multi-node level parallelism

- Requirements for parallel computing;
- How does MPI meet these requirements?
  - Specify parallel execution – single program on multiple data (SPMD) and tasks;
  - Data communication – two- and one-side communication (explicit or implicit message passing);
  - Synchronization – synchronization functions;

(6) **SPMD**: All processes (MPI tasks) run the same program. They can store different data but in the same variable names because of distributed memory location. Each process has its own memory space;

(7) **Less** data communication, **more** computation;
**MPI collective communication**

- **Collective** communications: synchronization, data movement, and collective computation;

- **Broadcast**: Rank 0 sends data to all other ranks.
- **Scatter**: Rank 0 distributes data to all other ranks.
- **Gather**: All ranks combine data into a single output at rank 0.
- **Reduction**: All ranks compute a single output value, typically a sum or product, at rank 0.
MPI examples on multiple nodes

- Use Intel MPI (impi), MVAPICH2, and OpenMPI on Mike-II;
- impi: better performance on Intel architecture;
- It also supports diagnostic tools to report MPI cost;

**Example 1:** the open source miniFE code

1. It is a part of the miniapps package;
2. It is written in C++;
3. It mimics the unstructured finite element generation, assembly, and solution of a 3D physical domain;
4. It can be thought as the kernel part in many science and engineering problems;
5. Output the performance in FLOPS, walltime, and MFLOP/s;
MPI examples on multiple nodes

- **Benchmark** your parallel applications;
- The baseline info is important for further **tuning**;
- It also allows us to determine the **optimal** settings to run the application more efficiently;
  - Have a better understanding of your **target machine**;
  - Set up a **non-trivial** case (or maybe an artificial test case, if multiple production runs are not feasible);
  - Know how large your workload is in the test case and make it **measurable**;
  - Set up the correct **MPI** run-time environment, if necessary;
  - Be aware of the issues with **high load**, memory usage, and **intensive swapping**;
  - Any computational “experiments” should be **reproducible**;
  - Tune only **one** of the multiple control knobs at a given time;
**MPI examples on multiple nodes**

- Load **Intel MPI** (+impi-4.1.3.048-Intel-13.0.0);
- Run the pre-built **miniFE.x** on 1 or 2 nodes;

![Graph showing Speedup and Walltime](image)

- The base info with 1 MPI task is not always available;
- On 2 nodes, the max FP perf. is **23.8 GFLOP/s (3.6%)**;
- Is it a compute-bound or memory-bound application?
MPI examples on multiple nodes

- Load Intel MPI (+impi-4.1.3.048-Intel-13.0.0);
- Run the pre-built `miniFE.x` on 2 nodes;

```
1 $ mpirun -np 32 ./miniFE.x nx=500

1 Starting CG solver ... mpiicpc/mpiicc/mpiifort
2 Initial Residual = 501.001
3 ...
4 Final Resid Norm: 0.00397271
```

- Check the `yaml` log:

```
1 # 32 cores on Mike-II regular nodes.
2 Total:
3 Total CG Time: 77.6081
4 Total CG Flops: 1.68522e+12
5 Total CG Mflops: 21714.4
6 Time per iteration: 0.38804
7 Total Program Time: 110.087
```
MPI examples on multiple nodes

- Load MVAPICH2 (+mvapich2-1.9-Intel-13.0.0);
- Run the pre-built miniFE.x on 2 nodes;

```
1 $ mpirun -np 32 ./miniFE.x nx=500
2 Starting CG solver ...          mpicxx/mpicc/mpif90
3 Initial Residual = 501.001
4 ...                              
5 Final Resid Norm: 0.00393607      
```

- Check the yaml log:

```
1 # 32 cores on Mike-II regular nodes.
2 Total:
3 Total CG Time: 79.0407
4 Total CG Flops: 1.68522e+12
5 Total CG Mflops: 21320.9
6 Time per iteration: 0.395203
7 Total Program Time: 104.769
```
**MPI examples on multiple nodes**

- **Load** OpenMPI-1.6.2 (+openmpi-1.6.2-Intel-13.0.0);
- **Run the pre-built** miniFE.x **on 2 nodes**;

```
1 $ mpirun -np 32 ./miniFE.x nx=500
```

```
1 Starting CG solver ...
2 Initial Residual = 501.001
3 ...
4 Final Resid Norm: 0.00393607
```

- **Check the yaml log:**

```
1 # 32 cores on 2 Mike-II regular nodes.
2 Total:
3 Total CG Time: 221.005
4 Total CG Flops: 1.68522e+12
5 Total CG Mflops: 7625.23
6 Time per iteration: 1.10503
7 Total Program Time: 324.937
```
MPI examples on multiple nodes

- The **same** performance with Intel **MPI** and **MVAPICH2**;
- OpenMPI-1.6.2 seems much **slower** than the ones above;

  (1) **High** average load > 100 per node;
  (2) Control the number of **OpenMP** threads;

```
1  $ OMP_NUM_THREADS=1 \ 
   mpirun -np 32 ./miniFE.x nx=500
```

1  # 32 cores on 2 Mike-II regular nodes.
2 Total:
3 Total CG Time:  104.758
4 Total CG Flops:  1.68522e+12
5 Total CG Mflops: 16086.7
6 Time per iteration:  0.523792
7 Total Program Time: 182.978

(3) After that, the performance difference is \(\sim 1.33 \times\);
MPI examples on multiple nodes

- Use OpenMPI-1.6.2, but reduce MPI tasks to 23;

```
$ OMP_NUM_THREADS=1 \nmpirun -np 23 ./miniFE.x nx=500
```

- That’s too bad: $20 \times$ slower! What happened with \(-np 23\)?
- Memory footprint is $\sim 46$ GB with \(nx=500\);
- Load imbalance: (1) wrt process or MPI task, (2) wrt node;
- Intense swapping and large swap space in use ($\gg 10$ GB);
MPI examples on multiple nodes

- Use OpenMPI-1.6.2, but reduce MPI tasks to 23;
- There are 16 MPI tasks on the 1st node, while the rest of the 7 tasks on the 2nd node – load imbalance wrt nodes;
- Swapping mechanism was triggered differently;

```
$ OMP_NUM_THREADS=1 \ 
  mpirun -np 23 -npernode 12 ./miniFE.x nx=500
```

1 # 23 cores on 2 Mike-II regular node.
2 # 12 on 1st node, 11 on 2nd node.
3 Total:
4 Total CG Time: 104.151
5 Total CG Flops: 1.68522e+12
6 Total CG Mflops: 16180.6
7 Time per iteration: 0.520753
8 Total Program Time: 179.608

- Note that it is fine to have a little swapping (∼20 MB here);
# Latency and throughput

**Latency and throughput matter**

<table>
<thead>
<tr>
<th>L2/L3 Cache</th>
<th>Infiniband</th>
<th>Hard Drive</th>
<th>Latency (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-9}$ ns</td>
<td>$10^{-6}$ μs</td>
<td>$10^{-3}$ ms</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L1 Cache</th>
<th>Main Memory</th>
<th>Gigabit Ethernet</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^{-8}$ ns</td>
<td>$10^{-7}$ ns</td>
<td>$10^{-4}$ ms</td>
<td>1000 MB/s</td>
</tr>
<tr>
<td>$100$ GB/s</td>
<td>$10$ GB/s</td>
<td>$10$ MB/s</td>
<td></td>
</tr>
</tbody>
</table>
MPI examples on multiple nodes

- No need to specify a machine file explicitly in the 3 cases;
- Try OpenMPI-1.6.5 (+openmpi-1.6.5-Intel-13.0.0);

```
1 $ OMP_NUM_THREADS=1 \n   mpirun -np 32 ./miniFE...x nx=500
```

1 # 32 cores on 2 Mike-II regular nodes.
2 Total:
3 Total CG Time: ≫ 74 minutes
4 Total CG Flops: 1.68522e+12
5 Total CG Mflops: ???
6 Time per iteration: ???
7 Total Program Time: ≫ 74 minutes

- Too bad, again: all tasks piled up on 1st node and 2nd is idle;
- Load imbalance wrt node;
- Intense swapping and large swap space in use (≫ 23 GB);
MPI examples on multiple nodes

- Use OpenMPI-1.6.5 (+openmpi-1.6.5-Intel-13.0.0);
- Specify a machine file explicitly;

```bash
$ OMP_NUM_THREADS=1 \ 
mpirun -np 32 -machinefile $PBS_NODEFILE \ 
./miniFE.x nx=500
```

1 # 32 cores on 2 Mike-II regular nodes.
2 Total:
3 Total CG Time: 213.942
4 Total CG Flops: 1.68522e+12
5 Total CG Mflops: 7876.99
6 Time per iteration: 1.06971
7 Total Program Time: 280.768

- After that, the MPI tasks were properly mapped on 2 nodes;
- Still $1.6 \times$ slower than OpenMP-1.6.2-Intel-13.0.0
  (Total CG Mflops: 12659.4);
**MPI examples on multiple nodes**

- Load Intel MPI (+impi-4.1.3.048-Intel-13.0.0);
- Diagnostic facilities (the log `stats.ipm`);

```bash
1$ I_MPI_STATS=ipm mpirun -np 32 ./miniFE.x nx=500
```

<table>
<thead>
<tr>
<th>#</th>
<th>[time]</th>
<th>[calls]</th>
<th>&lt;%mpi&gt;</th>
<th>&lt;%wall&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>324.365</td>
<td>13024</td>
<td>77.06</td>
<td>9.13</td>
</tr>
<tr>
<td>3</td>
<td>38.2421</td>
<td>75072</td>
<td>9.09</td>
<td>1.08</td>
</tr>
<tr>
<td>4</td>
<td>29.3108</td>
<td>32</td>
<td>6.96</td>
<td>0.83</td>
</tr>
<tr>
<td>5</td>
<td>28.3825</td>
<td>75072</td>
<td>6.74</td>
<td>0.80</td>
</tr>
<tr>
<td>6</td>
<td>0.363768</td>
<td>64</td>
<td>0.09</td>
<td>0.01</td>
</tr>
<tr>
<td>7</td>
<td>0.163873</td>
<td>96</td>
<td>0.04</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>0.0918336</td>
<td>75072</td>
<td>0.02</td>
<td>0.00</td>
</tr>
<tr>
<td>9</td>
<td>0.0051572</td>
<td>6720</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>10</td>
<td>420.925</td>
<td>245536</td>
<td>100.00</td>
<td>11.85</td>
</tr>
</tbody>
</table>

- Overhead of MPI communication;
MPI examples on multiple nodes

- Number of MPI tasks needs to **match** the nodes’ capacity;
- Pinning MPI tasks (ranks) to CPU cores;
- Properly distribute MPI tasks on multiple nodes;
- Run-time control:
  - **Intel MPI:**
    - `-hostfile <filename>`: specifies the host names on which MPI job runs (same as `-f`);
    - `-ppn <number>`: specifies no. of tasks per node;
  - **MVAPICH2:**
    - `-hostfile <filename> (-f)`: same as `impi`;
    - `-ppn <number>`: same as `impi`;
  - **Open MPI:**
    - `-hostfile <filename> (-machinefile)`: see the above;
    - `-npernode <number>`: specifies no. of tasks per node;
    - `-npersocket <number>`: specifies no. of tasks per socket;
Hybrid model

distributed-memory plus shared-memory systems
Hybrid model

- Except **inter-node** MPI communication, no essential difference between single- and multiple-node MPI jobs;
- Faster **intra-node** data communication within a node;
- More examples on a shared-memory systems;
- Here we focus on **MPI+OpenMP**:

  **Example 2**: calculation of $\pi$

- **MPI** takes care of **inter-node** communication, while **intra-node** parallelism is achieved by **OpenMP**;
- **MPI**: coarse-grained parl.; **OpenMP**: fine-grained parl.;
- Each MPI process can spawn multiple threads;
  - May reduce the memory usage on node level;
  - Good for accelerators or coprocessors;
  - It is hard to outperform a pure MPI job;
Hybrid model

\[ \pi = \int_0^1 \frac{4}{1 + x^2} \, dx \]

\[ \pi \simeq \frac{1}{N} \sum_{i=1}^{N} \frac{4}{1 + x_i^2}, \quad x_i = \frac{1}{N} \left( i - \frac{1}{2} \right), \quad i = 1, \ldots, N \]

- Pure MPI:

  MPI rank 0  MPI rank 1  \ldots  MPI rank \( n - 1 \)

  \[ x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8, \ldots, x_i, \ldots \]
Hybrid model

\[ \pi = \int_0^1 \frac{4}{1 + x^2} dx \]

\[ \pi \simeq \frac{1}{N} \sum_{i=1}^{N} \frac{4}{1 + x_i^2}, \quad x_i = \frac{1}{N} (i - \frac{1}{2}), \quad i = 1, \ldots, N \]

- Pure MPI:

  
<table>
<thead>
<tr>
<th>MPI rank 0</th>
<th>MPI rank 1</th>
<th>...</th>
<th>MPI rank n - 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_1 )</td>
<td>( x_2 )</td>
<td>...</td>
<td>( x_8 )</td>
</tr>
<tr>
<td>( x_4 )</td>
<td>( x_5 )</td>
<td>...</td>
<td>( x_{i-4} )</td>
</tr>
<tr>
<td>( x_3 )</td>
<td>( x_6 )</td>
<td>...</td>
<td>( x_i )</td>
</tr>
<tr>
<td>( x_2 )</td>
<td>( x_7 )</td>
<td>...</td>
<td>( x_{i+1} )</td>
</tr>
<tr>
<td>( x_1 )</td>
<td>( x_8 )</td>
<td>...</td>
<td>( x_{i+2} )</td>
</tr>
</tbody>
</table>

\text{MPI\_REDUCE}(\ldots) \implies \text{result.}
Hybrid model

\[ \pi = \int_0^1 \frac{4}{1 + x^2} dx \]

\[ \pi \simeq \frac{1}{N} \sum_{i=1}^{N} \frac{4}{1 + x_i^2}, \quad x_i = \frac{1}{N}(i - \frac{1}{2}), \quad i = 1, \ldots, N \]

- **Pure MPI:**

  - MPI rank 0
  - MPI rank 1
  - \ldots
  - MPI rank \( n - 1 \)

  \[ x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad x_7 \quad x_8 \quad x_i \quad \ldots \]

  \[ \text{MPI}_\text{REDUCE}(\ldots) \implies \text{result}. \]

- **Hybrid MPI+OpenMP:**

  - MPI rank 0
  - MPI rank 1
  - \ldots
  - MPI rank \( n - 1 \)

  \[ x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6 \quad x_7 \quad x_8 \quad x_i \quad \ldots \]

  openmp plus reduction + \[ \text{MPI}_\text{REDUCE}(\ldots) \implies \text{result}. \]
Hybrid model

```fortran
1 2
3 do i = istart, iend ! same var. diff. values
4   xi = h * (dble(i)-0.5_idp)
5   tmp = 1.0_idp + xi * xi
6   fsum = fsum + 1.0_idp / tmp
7 end do
8 fsum = 4.0_idp * h * fsum
9 call MPI_REDUCE(fsum,pi,1,..., &
10   MPI_SUM,0,MPI_COMM_WORLD,ierr)
```

- **SPMD**: Each MPI task runs the **same** program and holds the **same** variable names;
- Due to the **distinct** memory space, the **same** variable (istart and iend) may hold **different** values;
Hybrid model

1  !$omp parallel do private(i,xi,tmp), &
2       reduction(+:fsum)
3  do i = istart, iend  !same var.diff.values
4     xi = h * (dble(i)-0.5_idp)
5     tmp = 1.0_idp + xi * xi
6     fsum = fsum + 1.0_idp / tmp
7  end do
8  fsum = 4.0_idp * h * fsum
9  call MPI_REDUCE(fsum,pi,1, ..., &
10     MPI_SUM,0,MPI_COMM_WORLD,ierr)

- Add the OpenMP directive/pragma to parallelize the loop;
- Make the partial sum (fsum) a **reduction** variable with **plus** operation;
- The **MPI_REDUCE** is the same as before at the **outer** level;
Hybrid model

- **Hybrid MPI+OpenMP:**

```fortran
1  !$omp parallel do private(i,xi,tmp), &
2      reduction(+:fsum)
3  do i = istart, iend  ! same var. diff. values
4      xi = h * (dble(i)-0.5_idp)
5      ...
```

- **On Mike-II using impi-4.1.3.048, \( N = 2 \times 10^9 \):**

<table>
<thead>
<tr>
<th>No. of MPI tasks</th>
<th>No. of threads</th>
<th>Wall time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1</td>
<td>0.45986</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>0.46088</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>0.46389</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>0.46021</td>
</tr>
<tr>
<td>1</td>
<td>16</td>
<td>0.45919</td>
</tr>
</tbody>
</table>
Hybrid model

- How many OpenMP threads and MPI tasks are needed?
- What happens if `OMP_NUM_THREADS=16 mpirun -np 16 ...`?

```
[Top output]
Tasks: 813 total, 88 running, 725 sleeping, 0 stopped, 0 zombie
Cpu(s): 95.2%us, 2.1%sy, 0.0%ni, 2.7%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 32815036k total, 16993228k used, 15821808k free, 48676k buffers
Swap: 100663292k total, 45556k used, 100617736k free, 13629192k cached
```

- Again, **high load** issues per node and should prevent;
- Don’t oversubscribe the node resources;
- **MPI+OpenMP** turns out to be **MPI × OpenMP**;
Compute-bound and memory-bound applications
Where are the bottlenecks?

- A lot of factors can slow down your applications;
- In terms of execution units and a variety of bandwidths, we have:
  - Compute-bound (aka. “CPU”-bound);
  - Cache-bound;
  - Memory-bound;
  - I/O-bound;
- For a given application, how do we know it is compute-bound or memory-bound?
- Why do we need to know this and what is the benefit of it?
  - you’re the developer of the application;
  - you’re the user of the application;
Where are the bottlenecks?

- A lot of factors can **slow** down your applications;
- Parallel algorithms, bandwidths, overhead, . . .;
- Once a datum is fetched from the **memory**, on average how many **arithmetic** operations do we need to perform on that datum to keep the execution units busy?

**FP Performance** (GFLOP/s) =

Memory BW (GB/s) × Operation Intensity (FLOP/byte)

\[ y(\text{FP Perf.}) = k(\text{BW.}) \times (\text{Oi.}) \]

- However, the **max** performance **cannot** go beyond the theoretical **peak** performance;
Where are the bottlenecks?

- A lot of factors can **slow** down your applications;
- Parallel algorithms, bandwidths, overhead, . . .

![Graph showing operation intensity vs performance with Double precision.](image)

**Peak perf. of 332.8 GFLOP/s**

**Bandwidth of 106.6 GB/s**

**Roofline model**

*Williams et al. Commun. ACM 52, 65 (2009).*
Where are the bottlenecks?

- A lot of factors can **slow** down your applications;
- Parallel algorithms, bandwidths, overhead, . . .

---

**On Mike-II**

**Memory bound:**

\[ \text{OI} \ll 3.2 \text{ FLOP/byte} \]

**Compute bound:**

\[ \text{OI} \gg 3.2 \text{ FLOP/byte} \]
Where are the bottlenecks?

- A lot of factors can **slow** down your applications;
- Parallel algorithms, bandwidths, overhead, . . .;
- On average, for each DP FP number an application needs at least **25 FLOPs** to be **compute bound**;
- What can we learn from the **roofline** model?
  - It is **not uncommon** to see that there are many applications performing at a level of much less than **30 GFLOP/s** (10%);
- These applications are typically **memory** bound;
- We need to **increase** the **OI**. per data fetching;
- **Reuse** the data in **caches** as much as possible;
- Use well developed and optimized libraries: **MKL** routines on Intel CPUs and **ACML** on AMD CPUs;
- Link your **top-level** applications to the optimized libraries;
Compute bound

- On SuperMIC (Ivy Bridge at 2.8 GHz), the theoretical peak performance is 22.4 GFLOP/s per core;
- Benchmark **MKL DGEMM** routine (matrix-matrix products);

```c++
const int nsize = 10000;
const int iteration = 20;
// allocate the matrices.
// initialize the matrices.
for (k=0; k<iteration; k++) // C = A × B.
{
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, nsize, nsize, nsize, alpha, matrix_a, nsize, matrix_b, nsize, beta, matrix_c, nsize);
}
perf = 2.0 * dsize * dsize * dsize * (double) (iteration) / elapsed_time / 1.e+6;
```
Compute bound

- On SuperMIC (Ivy Bridge at 2.8 GHz), the theoretical peak performance is 22.4 GFLOP/s per core;
- Benchmark MKL DGEMM routine (matrix-matrix products);
Compute bound

- How does the attainable performance improve with respect to the matrix size?
- How does the attainable performance improve with respect to the thread count?
- What happens around the matrix size of $1,000 \times 1,000$?

<table>
<thead>
<tr>
<th>No. of threads</th>
<th>Attainable perf. (GFLOP/s)</th>
<th>Peak perf. (GFLOP/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(matrix size $10^4 \times 10^4$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>27.16</td>
<td>22.4</td>
</tr>
<tr>
<td>2</td>
<td>52.41</td>
<td>44.8</td>
</tr>
<tr>
<td>4</td>
<td>98.46</td>
<td>89.6</td>
</tr>
<tr>
<td>10</td>
<td>220.3</td>
<td>224.0</td>
</tr>
<tr>
<td>20</td>
<td>209.0</td>
<td>448.0</td>
</tr>
</tbody>
</table>

- Turbo boost mode at higher frequency;
Memory bound

- Does the **roofline** model tell us the whole story?
- The **MKL** `DGEMM` routine is **compute bound**;
- Consider the other scenario: what happens if my code does **not** have too many **FP** operations?
- We need a quantity like the **memory bandwidth** (MB/s or GB/s) to benchmark the code, instead of FLOP/s;
- Consider the out-of-place **matrix transposition**:

```fortran
1  do i = 1, nsize
2    do j = 1, nsize
3      matrix_out(i,j)= matrix_inp(j,i)
4    end do
5  end do
```

- **Throughput** (GB/s) = $2N^2/(2^{30}T_{\text{walltime}})$;
Memory bound

- Intel Xeon processors on SuperMIC, Mike-II, QB2, and Philip;

<table>
<thead>
<tr>
<th>Machine</th>
<th>CPU Family</th>
<th>CPU Freq.</th>
<th>LLC</th>
<th>DDR Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td>SuperMIC</td>
<td>E5 v2 2680</td>
<td>2.8 GHz</td>
<td>25 MB</td>
<td>1866 MHz</td>
</tr>
<tr>
<td>SuperMIC†</td>
<td>E5 v4 2690</td>
<td>2.6 GHz</td>
<td>35 MB</td>
<td>2400 MHz</td>
</tr>
<tr>
<td>QB2</td>
<td>E5 v2 2680</td>
<td>2.8 GHz</td>
<td>25 MB</td>
<td>1866 MHz</td>
</tr>
<tr>
<td>QB2†</td>
<td>E7 v2 4860</td>
<td>2.6 GHz</td>
<td>30 MB</td>
<td>1066 MHz</td>
</tr>
<tr>
<td>Mike-II</td>
<td>E5 v1 2670</td>
<td>2.6 GHz</td>
<td>20 MB</td>
<td>1600 MHz</td>
</tr>
<tr>
<td>Mike-II†</td>
<td>E7 4870</td>
<td>2.4 GHz</td>
<td>30 MB</td>
<td>1066 MHz</td>
</tr>
<tr>
<td>Philip</td>
<td>X5570</td>
<td>2.93 GHz</td>
<td>8 MB</td>
<td>1333 MHz</td>
</tr>
</tbody>
</table>

† on SuperMIC’s and QB2’s bigmem nodes, or Mike-II’s bigmemtb nodes.

- Different Xeon processors on bigmem or bigmemtb nodes to support large memory;
Memory bound

- Matrix transposition: MKL routine `mkl_domatcopy`;

```c
for (k=0; k<iteration; k++)
    mkl_domatcopy('R', 'T', nsize, nsize, \
    alpha, matrix_a, nsize, matrix_b, nsize);
```

- Benchmark the throughput (GB/s): 10 threads with `numactl`

<table>
<thead>
<tr>
<th>Machine</th>
<th>4,000</th>
<th>20,000</th>
<th>40,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>SuperMIC</td>
<td>23.93</td>
<td>21.22</td>
<td>18.68</td>
</tr>
<tr>
<td>SuperMIC + bigmem</td>
<td>17.96</td>
<td>18.01</td>
<td>18.08</td>
</tr>
<tr>
<td>QB2 + k40</td>
<td>20.96</td>
<td>18.05</td>
<td>15.45</td>
</tr>
</tbody>
</table>

†k40 configured at 1600 MHz.

- Both memory bandwidth and latency contribute to the throughput;
Memory and compute bound

- **Memory-bound by nature**: increase throughput;
- **Memory-bound due to implementation**:
  1. Optimize the algorithm and code to reuse the data in caches: spatial and temporal reuse;
  2. It is possible to convert memory-bound to compute-bound code;
  3. Mixed heavy arithmetic parts and non-FP operations;
  4. Why most applications fall in the memory-bound category?
  5. Know memory architecture better;
  6. Changing compiler may be helpful;
  7. Prior to optimizing the “hotspot”, identify if it is compute-bound or memory-bound;
Socket and processor level
within a socket or a processor
Socket and processor level

- Within a node, several processors can be connected together to form a **multi-processor** system;
- This is called a **socket**: two-socket or four-socket systems;
- The Intel Xeon processors **Sandy Bridge** (v1), **Ivy Bridge** (v2), and **Broadwell** (v4) on SuperMIC, Mike-II, and QB2;
- Connection through the Intel **QPI** (QuickPath Interconnect), while AMD uses **HyperTransport** technology;
- It can be thought of a **point-to-point** interconnection between multiple-processors;
- Not only implemented as **links** between processors, but also used to connect a processor and the **I/O hub**;
- How does this affect **parallelism** at the application or code execution level?
Socket and processor level

- The **NUMA** (non-uniform memory access) architecture;
- The key point in **NUMA** is about **shared memory**;
- Furthermore, it has been implemented as **ccNUMA** (cache coherent NUMA);
Socket and processor level

- The **NUMA** (non-uniform memory access) architecture;
- The key point in **NUMA** is about shared memory;
- Furthermore, it has been implemented as **ccNUMA** (cache coherent NUMA);

![Diagram showing Intel Xeon E5 nodes with QPI links and memory controllers connected to memory]

Intel Xeon E5 (node 0)

**QPI links**

32 GB/s

Intel Xeon E5 (node 1)

**DDR3 RAM**

57.6 GB/s

**memory**
Socket and processor level

- Each processor is connected to its **own** RAM via the memory controller;
- Due to the **QPI** links, CPU cores in a processor (**node 0**) can access the RAM connected to the other processor (**node 1**);
Socket and processor level

- Why the **NUMA** matters?
- Focus on how an array was allocated and **initialized** on shared-memory system;
- “**First Touch**” policy – memory **binding** or **affinity**;
- Bandwidth differences in **local** and **remote** memory access;
- It may have significant impact on code performance;
- If it plays a role in application’s **performance**, are there any ways to **control** it?
- Linux provides a wonderful tool **numctl** that allows us to
  (1) run processes with a memory **placement policy** or specified scheduling;
  (2) set the processor **affinity** and memory **affinity** of a process;
Socket and processor level

- With `numactl` we can
  1. run processes with a memory placement policy or specified scheduling;
  2. set the processor affinity and memory affinity of a process;

# Lists the available cores: same as `-H`.
$ numactl --hardware

# Ensures memory is allocated only on specific nodes.
$ numactl --membind

# Ensures specified command and its child processes execute only on the specified node.
$ numactl --cpunodebind

# Ensures a specified command and its child processes execute only on the specified processor.
$ numactl --phycpubind
Socket and processor level

- Memory latency between **UMA** cores and **NUMA** cores;
- On **SuperMIC** 2-socket regular node and 2-socket bigmem node:

|   | Measuring idle latencies (in ns)...
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Numa node</td>
</tr>
<tr>
<td>2</td>
<td>Numa node 0 1 # DDR3 1866 MHz</td>
</tr>
<tr>
<td>3</td>
<td>0 72.3 123.0 # SuperMIC reg. node</td>
</tr>
<tr>
<td>4</td>
<td>123.5 72.9 # NUMA/UMA = 1.7</td>
</tr>
</tbody>
</table>

1 Bandwidths are in GB/sec
2 Using Read-only traffic type
3 Numa node
4 Numa node 0 1 # DDR3 1866 MHz
5 0 55.86 25.43 # SuperMIC reg. node
6 1 25.48 50.23 # UMA/NUMA = 2.2
### Socket and processor level

- Memory latency between **UMA** cores and **NUMA** cores;
- On SuperMIC 2-socket regular node and 2-socket bigmem node:

<table>
<thead>
<tr>
<th></th>
<th>Measuring idle latencies (in ns)...</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><em>Numa node</em></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><em>Numa node</em> 0 1 # DDR4 2400 MHz</td>
</tr>
<tr>
<td>4</td>
<td>0 87.2 128.6 # SuperMIC bigmem node</td>
</tr>
<tr>
<td>5</td>
<td>1 129.8 87.9 # NUMA/UMA = 1.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Bandwidths are in GB/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><em>Using Read-only traffic type</em></td>
</tr>
<tr>
<td>2</td>
<td><em>Numa node</em></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><em>Numa node</em> 0 1 # DDR4 2400 MHz</td>
</tr>
<tr>
<td>5</td>
<td>0 67.78 23.49 # SuperMIC bigmem node</td>
</tr>
<tr>
<td>6</td>
<td>1 23.41 67.94 # UMA/NUMA = 2.9</td>
</tr>
</tbody>
</table>
Socket and processor level

- Memory latency between **UMA** cores and **NUMA** cores;
- On **QB2** the 2-socket regular node and 4-socket bigmem node:

<table>
<thead>
<tr>
<th></th>
<th>Measuring idle latencies (in ns)...</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Numa node</td>
</tr>
<tr>
<td>3</td>
<td>Numa node 0 1</td>
</tr>
<tr>
<td>4</td>
<td>0 71.4 122.9</td>
</tr>
<tr>
<td>5</td>
<td>1 123.6 71.5</td>
</tr>
</tbody>
</table>

Bandwidths are in GB/sec

<table>
<thead>
<tr>
<th></th>
<th>Using Read-only traffic type</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Numa node</td>
</tr>
<tr>
<td>4</td>
<td>Numa node 0 1</td>
</tr>
<tr>
<td>5</td>
<td>0 53.46 25.02</td>
</tr>
<tr>
<td>6</td>
<td>1 25.03 46.82</td>
</tr>
</tbody>
</table>

Memory latency (in ns) between UMA and NUMA cores:

- **QB2** reg. node: 71.4 - 122.9
- **NUMA/UMA** = 1.7

Bandwidths (in GB/sec):

- **QB2** reg. node: 53.46 - 25.02
- **UMA/NUMA** = 2.2
### Socket and processor level

#### Measuring idle latencies (in ns)...

<table>
<thead>
<tr>
<th>Numa node</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>129.4</td>
<td>202.1</td>
<td>192.0</td>
<td>200.9</td>
</tr>
<tr>
<td>1</td>
<td>202.2</td>
<td>130.4</td>
<td>199.6</td>
<td>194.2</td>
</tr>
<tr>
<td>2</td>
<td>196.4</td>
<td>196.0</td>
<td>129.0</td>
<td>193.4</td>
</tr>
<tr>
<td>3</td>
<td>201.4</td>
<td>195.9</td>
<td>191.4</td>
<td>128.2</td>
</tr>
</tbody>
</table>

Bandwidths are in GB/sec

- Using Read-only traffic type
- DDR3 1600/1066 MHz

<table>
<thead>
<tr>
<th>Numa node</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>53.52</td>
<td>12.65</td>
<td>12.68</td>
<td>12.44</td>
</tr>
<tr>
<td>1</td>
<td>12.70</td>
<td>54.39</td>
<td>12.65</td>
<td>12.65</td>
</tr>
<tr>
<td>2</td>
<td>12.48</td>
<td>12.50</td>
<td>53.71</td>
<td>12.66</td>
</tr>
<tr>
<td>3</td>
<td>12.63</td>
<td>12.52</td>
<td>12.71</td>
<td>54.37</td>
</tr>
</tbody>
</table>
Core level parallelism
### Core level (vectorization)

<table>
<thead>
<tr>
<th>Instruction set</th>
<th>Register width</th>
<th>Processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSE</td>
<td>128-bit</td>
<td>Pentium (1997)</td>
</tr>
<tr>
<td>SSE2</td>
<td>128-bit</td>
<td>Pentium III (1999)</td>
</tr>
<tr>
<td>AVX</td>
<td>256-bit</td>
<td>Xeon Sandy Bridge (2011)</td>
</tr>
<tr>
<td>AVX</td>
<td>256-bit</td>
<td>AMD Bulldozer (2011)</td>
</tr>
<tr>
<td>AVX2</td>
<td>256-bit</td>
<td>Xeon Haswell (2013)</td>
</tr>
<tr>
<td>AVX2</td>
<td>256-bit</td>
<td>Xeon Broadwell (2014)</td>
</tr>
<tr>
<td>AVX2</td>
<td>256-bit</td>
<td>AMD Carrizo (2015)</td>
</tr>
</tbody>
</table>

- Compiler and assembler support of AVX:
  - (1) GCC higher than v4.6;
  - (2) Intel compiler suite higher than v11.1;
  - (3) PGI compilers since 2012;
- Linux kernel version higher than 2.6.30 to support AVX;
Core level (vectorization)

- Why vectorization matters?
- Vector width keeps increasing from 128-bit to 256-bit, even to 512-bit on KNC and KNL;
- Take the advantage of longer vector register width;
- Each register in the 256-bit AVX can hold up to four 64-bit (8-byte) DP floating point numbers, or eight SP numbers;

(1) For additions or products, it is preferable to operate four pairs of DP numbers, or eight pairs of SP numbers with a single instruction;
(2) By comparison, the vectorization (AVX) can deliver the max speedup of 4 for DP or 8 for SP;
(3) Improvement for SP operations is always doubled compared to DP;
Core level (vectorization)

- **Vectorization** works in such a way so that the execution units execute a *single* instruction on multiple data *simultaneously* (in parallel) on a *single* CPU core (**SIMD**);
- Enabling vectorization in your applications will “potentially” improve performance;
- Typically vectorization can be attributed to **data** parallelism;
Core level (vectorization)

- Intel compilers support **auto-vectorization** for `-O2` or higher;
- Compile the following code with `-vec` and `-no-vec` flags;

```c
1 // vectorized or non-vectorized loop
2 const int nsize = 20;
3 const int kitemax = 10000000;
4 // allocate and initialize vectors.
5 ...
6 // sum over all vector elements
7 for (k=0; k<kitemax; k++)
8 for (i=0; i<nosize; i++)
9    vector_a[i] = vector_a[i] + vector_b[i] + vector_c[i] + vector_d[i] + vector_e[i];
```

- Add `#pragma simd` or `#pragma vector` right above the **inner** loop, and see what happens;
Core level (vectorization)

- Intel compilers support **auto-vectorization** for `-O2` or higher;
- Compile the following code with `-vec` and `-no-vec` flags;

```c
// vectorized or non-vectorized loop
const int nsize = 20;
const int kitemax = 10000000;

// allocate and initialize vectors.
...

// sum over all vector elements
for (k=0; k<kitemax; k++)
for (i=0; i<nosize; i++)
vector_a[i] = vector_a[i] + vector_b[i] + vector_c[i] + vector_d[i] + vector_e[i];
```

- `-vec` (-O2): 0.113 sec; `-no-vec` (-O1): 0.226 sec with 1 thread;
- Does the speedup remain the **same** if we use more threads?
Core level (vectorization)

- Intel compilers support **auto-vectorization** for `-O2` or higher;
- Compile the following code with `-vec` and `-no-vec` flags;

```c
1    // vectorized or non-vectorized loop
2    const int nsize = 20;
3    const int kitemax = 10000000;
4    // allocate and initialize vectors.
5    ...
6    // sum over all vector elements
7    for (k=0; k<kitemax; k++)
8        for (i=0; i<nosize; i++)
9            vector_a[i] = vector_a[i] + vector_b[i] + vector_c[i] + vector_d[i] + vector_e[i];
```

- Record the speedup of `vec/no-vec` with varying `nosize`;
- `nosize = 20, 200, 500, 1000, 3000, and 5000 (1 thread);`
Core level (vectorization)

- Let's take a look at which loop is vectorized and which is not:
  
  turn `-vec-report3` on;

  
  ```
  .._v0.c(52): (col. 3) remark: LOOP WAS VECTORIZED
  .._v0.c(78): (col. 4) remark: LOOP WAS VECTORIZED
  .._v0.c(77): (col. 4) remark: loop was not
                  vectorized: not inner loop
  ```

- Everything is expected. We know that the **inner loop** is a good candidate for vectorization.
Core level (vectorization)

- Check the speedup and performance:

  ![Graph showing speedup and performance with respect to array size]

- A speedup of $\sim 2$ for small data and $\sim 1$ for large data;
- Significant improvement over the non-vectorized loops;
- The max performance is about $31\%$ of the peak performance (22.4 GFLOP/s) with one thread on SuperMIC;
Core level (vectorization)

• Can we do better?
• Make `nosize` unknown at compilation time (v1), so the compiler may choose a different optimization technique;

```
// vectorized or non-vectorized loop
int main (int argc, char *argv[])
...
nosize = atoi(argv[1]);
...
// sum over all vector elements
for (k=0; k<kitemax; k++)
for (i=0; i<nosize; i++)
  vector_a[i] = vector_a[i] + vector_b[i] + vector_c[i] + vector_d[i] + vector_e[i];
```
Core level (vectorization)

- Can we do **better**?
- Make **nosize unknown** at compilation time (**v1**), so the compiler may choose a different optimization technique;

```c
_v1.c(50): (col. 3) remark: LOOP WAS VECTORIZED
_v1.c(75): (col. 4) remark: PERMUTED LOOP WAS VECTORIZED
_v1.c(76): (col. 4) remark: loop was not vectorized: not inner loop
```

- **Confused**?!
- The compiler is smart enough to **permute** (swap) the **inner** and **outer** loops, and vectorize the “**inner**” (the ordinary **outer**) loop;
Core level (vectorization)

- Again, the speedup and performance:

- Significant improvement for the **large** data size;
- The relative performance (**-vec/-no-vec**) may be lower (small data);
- The performance of **-no-vec** is also improved;
Core level (vectorization)

- On SuperMIC (Ivy Bridge at 2.8 GHz), a simple estimate shows we achieved \( \sim 2.5 \) DP FLOP/cycle (v1);
- Both Sandy Bridge and Ivy Bridge support up to 8 DP FLOP/cycle (4 add and 4 mul);
- Thus, \( 2.5/8 \approx 31\% \) of the peak performance;
- Can we improve it?
  - Loop was already vectorized;
  - Contiguous memory access;
  - Memory affinity?
  - Reuse the data in cache?
  - FP execution units are not saturated;
  - ...
Summary

• Performance scales on different levels:
  o **MPI**: $\sim 10-1000 \times$;
  o **OpenMP**: $\sim 10-40 \times$;
  o **Memory affinity** on multiple-socket: $\sim 2-4 \times$;
  o **Vectorization**: $\sim 4-8 \times$;

• **Compute-bound** and **memory-bound** applications;

• Bottlenecks in most parallel applications;

• Memory **hierarchy** and **throughput**;

• Performance killers: **High** load, load **imbalance** issues, and
  **intensive** swapping, . . . ;

Questions?

sys-help@loni.org