

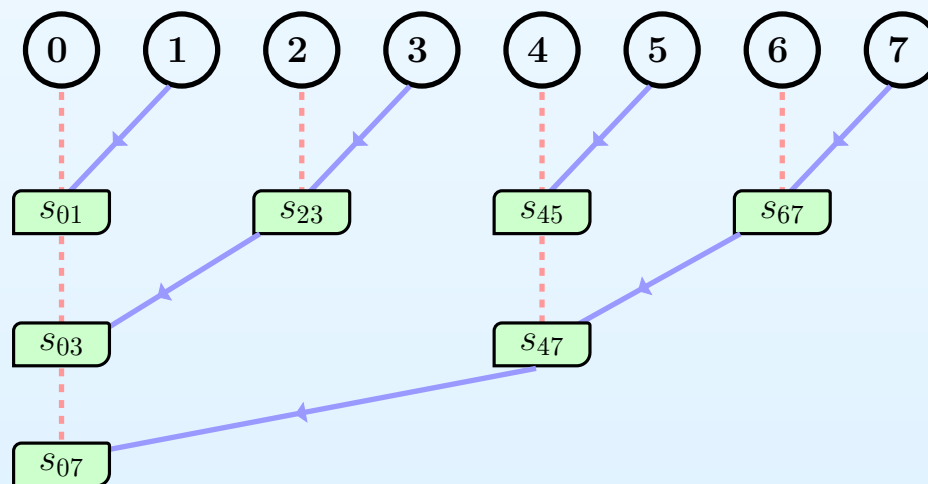


# Understanding Parallel Applications

Xiaoxu Guan

*High Performance Computing, LSU*

*May 31, 2018*





# 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  $\simeq 40$  hours (wall-clock time)
- (c) Charged CPU hours are  $2,500 \times 40 = 0.1$  M SUs
- (d) It is about  $100,000/24/365 \simeq 11$  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

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

**Data parallelism**

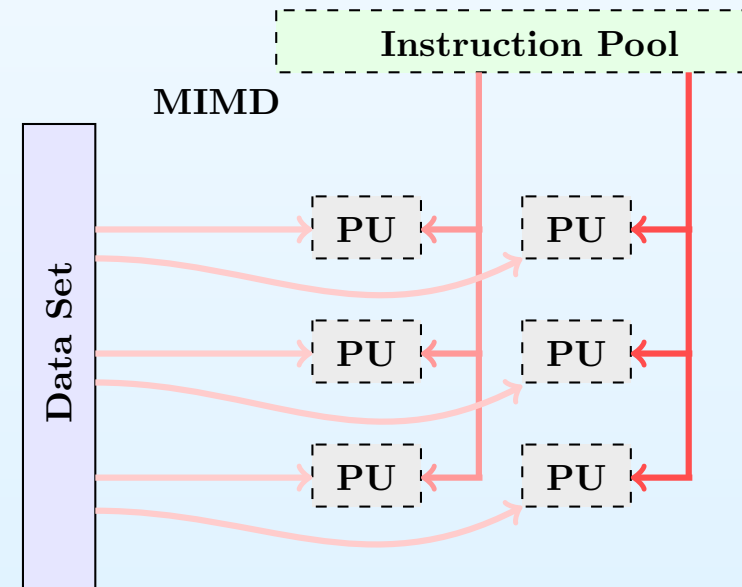
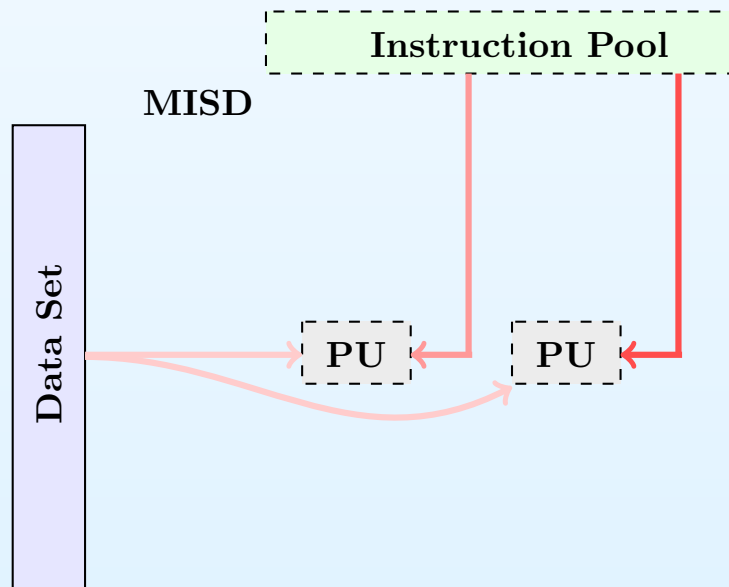
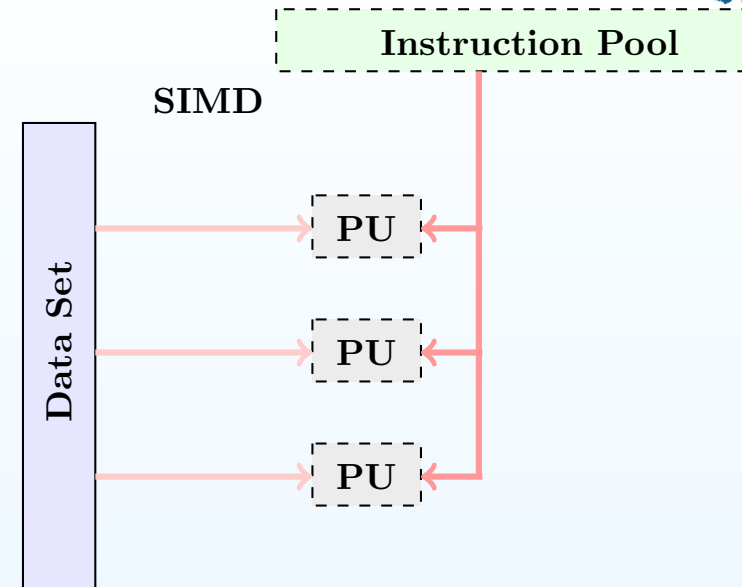
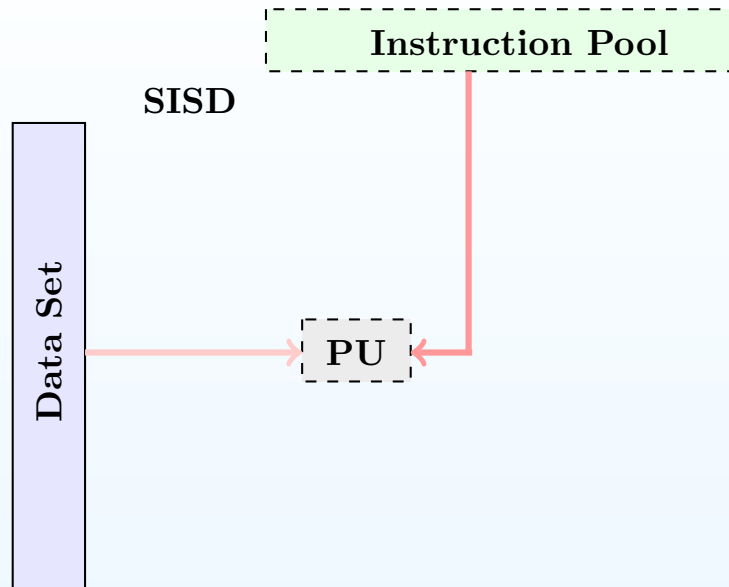
```
1 { for c(i) = a(i) + b(i) }
2 { for d(j) = sin(a(j)) }
```

**Task parallelism**

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



# Parallel computing





# 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

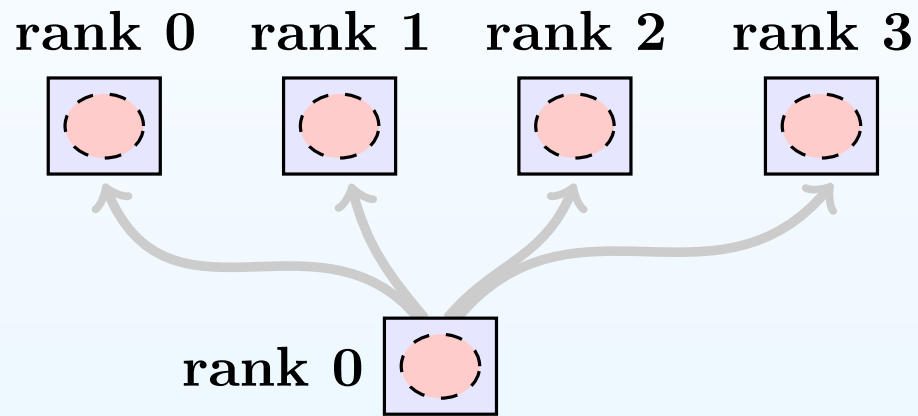
- Requirements for parallel computing;
  - How does **MPI** meet these requirements?
    - **Specify parallel execution** – single program on multiple data (**SPMD**) and tasks;
    - **Data communication** – two- and one-side communication (explicit or implicit **message passing**);
    - **Synchronization**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



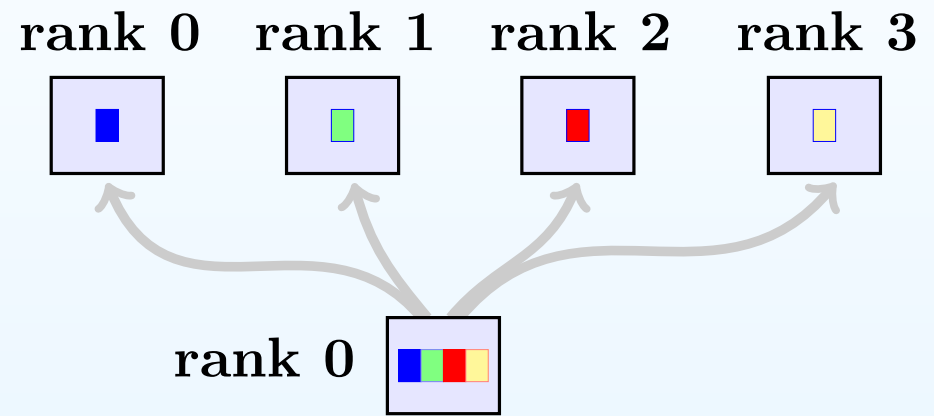
# MPI collective communication

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

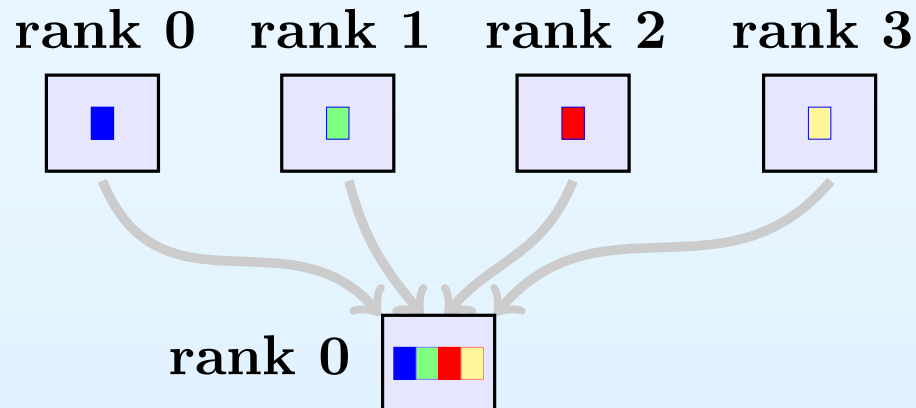
## Broadcast



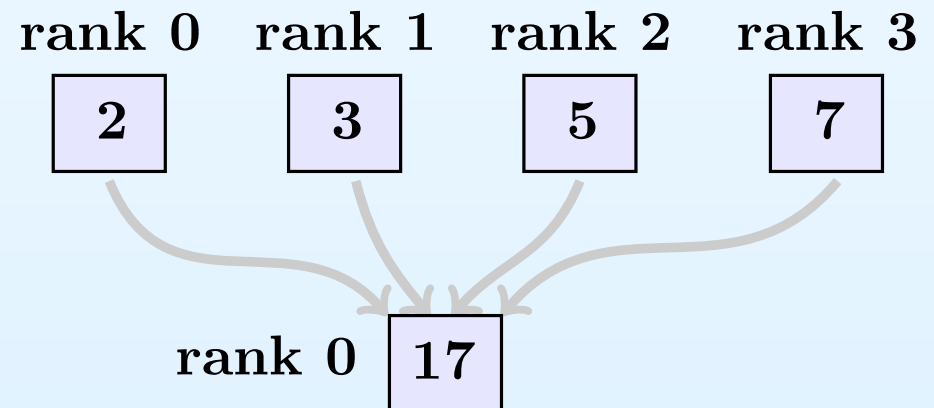
## Scatter



## Gather



## Reduction





## 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**;



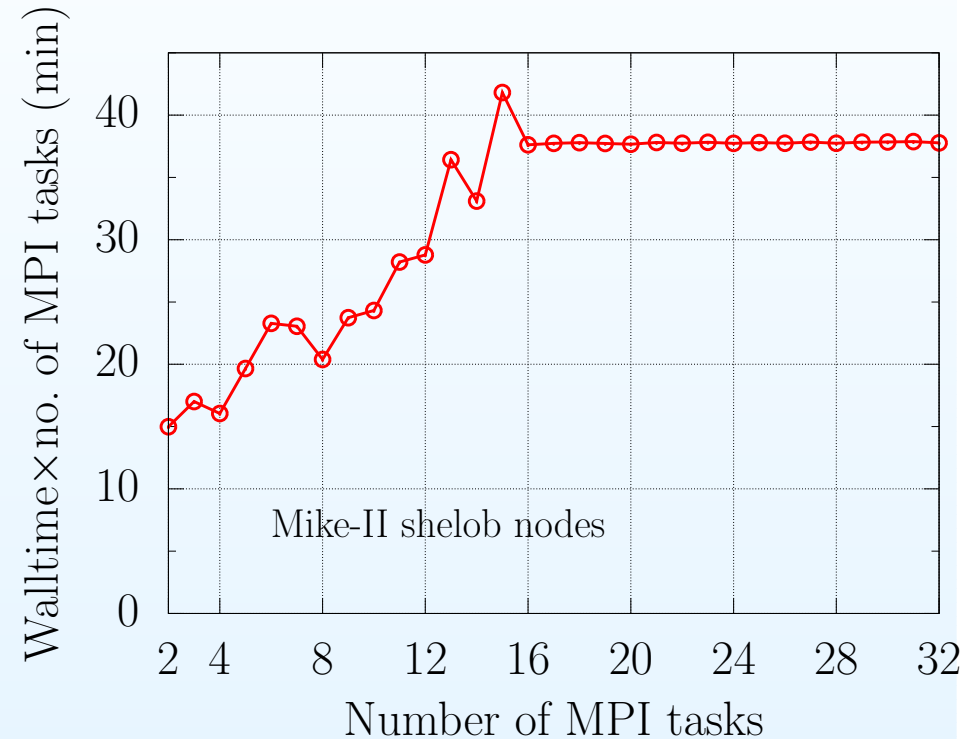
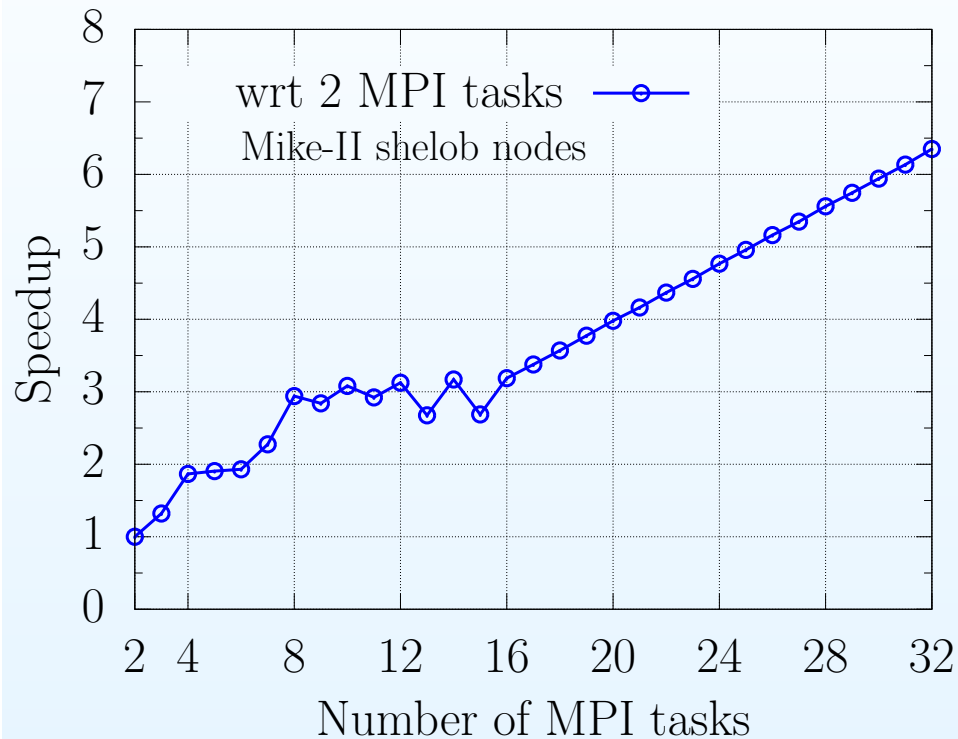
## MPI examples on multiple nodes

- **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;



- 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
```

```
1 Starting CG solver ...      mpicxx/mpicc/mpif90
2 Initial Residual = 501.001
3 ...
4 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 ...      mpicxx/mpicc/mpif90
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;

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

```
1 # 23 cores on 2 Mike-II regular nodes.
2 Total:
3 Total CG Time: 2194.6
4 Total CG Flops: 1.68522e+12
5 Total CG Mflops: 767.89
6 Time per iteration: 10.973
7 Total Program Time: 2365.55
```

- 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;

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

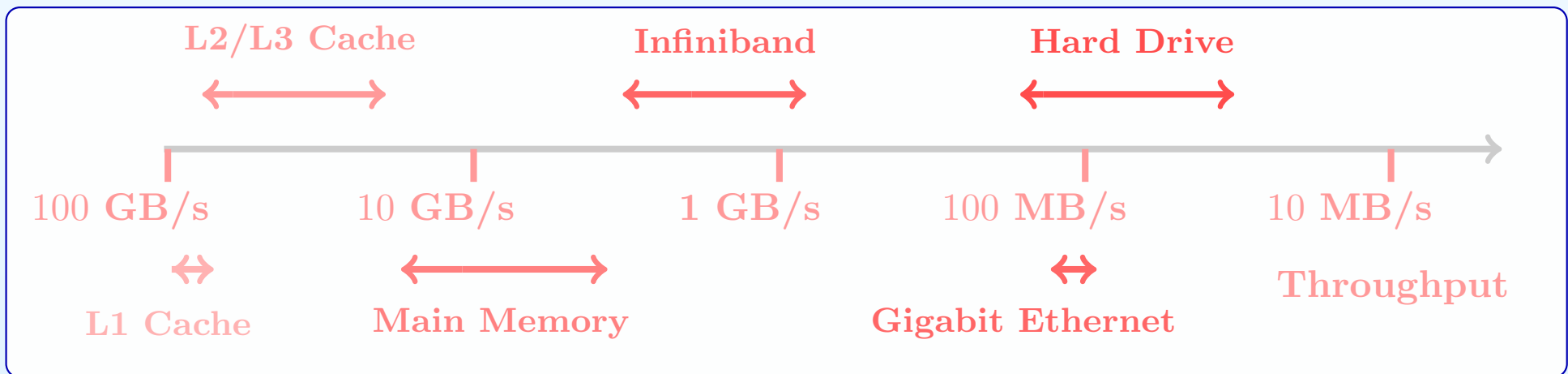
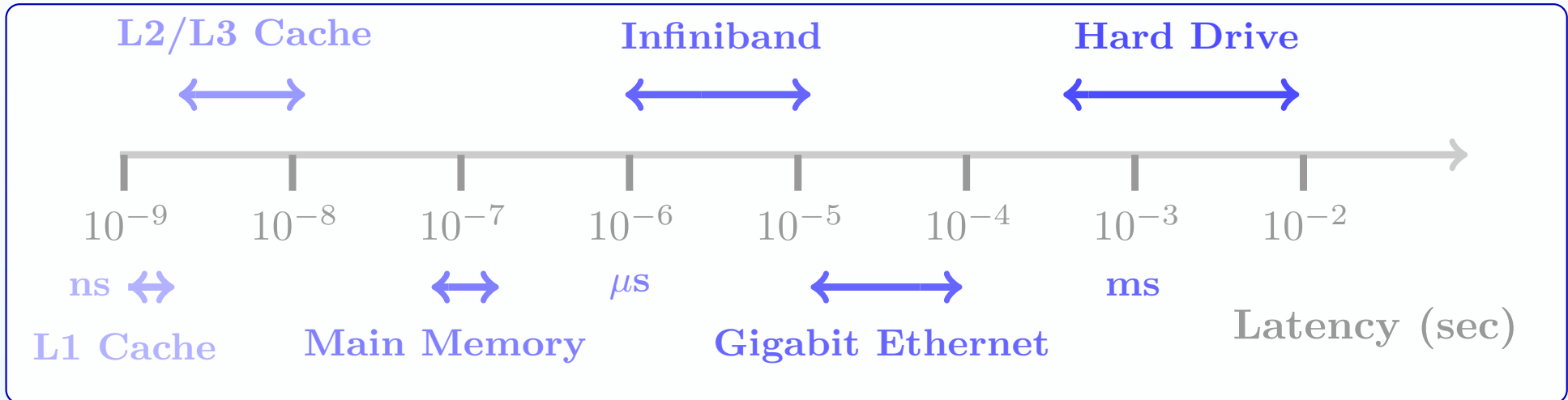
```
1 # 23 cores on 2 Mike-II regular node.
2 # 12 on 1st node, 11 on 2nd node.
3 Total:
4 Total CG Time: 104.151
5 Total CG Flops: 1.68522e+12
6 Total CG Mflops: 16180.6
7 Time per iteration: 0.520753
8 Total Program Time: 179.608
```

- Note that it is fine to have a little swapping (~20 MB here);



# Latency and throughput

## Latency and throughput matter





## 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 \
   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;

```
1 $ 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);

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

1 #		[time]	[calls]	<%mpi>	<%wall>
2 #	MPI_Allreduce	324.365	13024	77.06	9.13
3 #	MPI_Send	38.2421	75072	9.09	1.08
4 #	MPI_Init	29.3108	32	6.96	0.83
5 #	MPI_Wait	28.3825	75072	6.74	0.80
6 #	MPI_Bcast	0.363768	64	0.09	0.01
7 #	MPI_Allgather	0.163873	96	0.04	0.00
8 #	MPI_Irecv	0.0918336	75072	0.02	0.00
9 #	MPI_Comm_size	0.0051572	6720	0.00	0.00
10 #	MPI_TOTAL	420.925	245536	100.00	11.85

- 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;
- **MPVAPICH2:**
  - `-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;

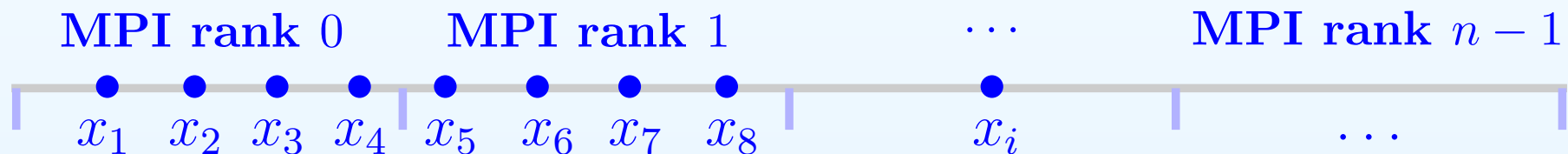


# Hybrid model

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

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

## • Pure MPI:



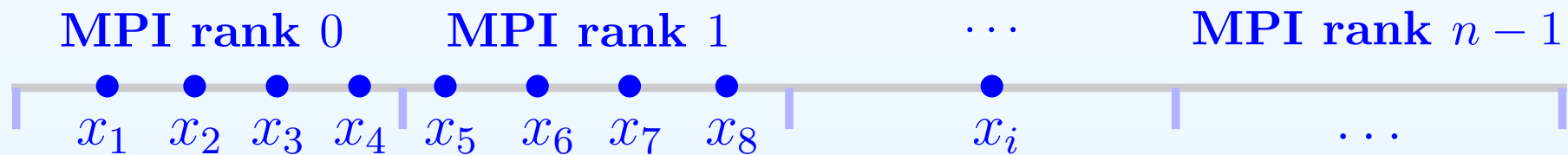


# Hybrid model

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

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

- Pure MPI:



**MPI\_REDUCE(...)**  $\Rightarrow$  result.

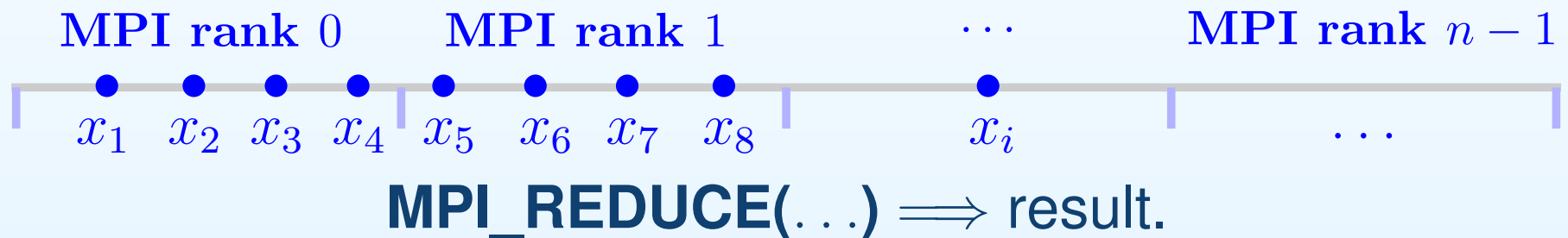


# Hybrid model

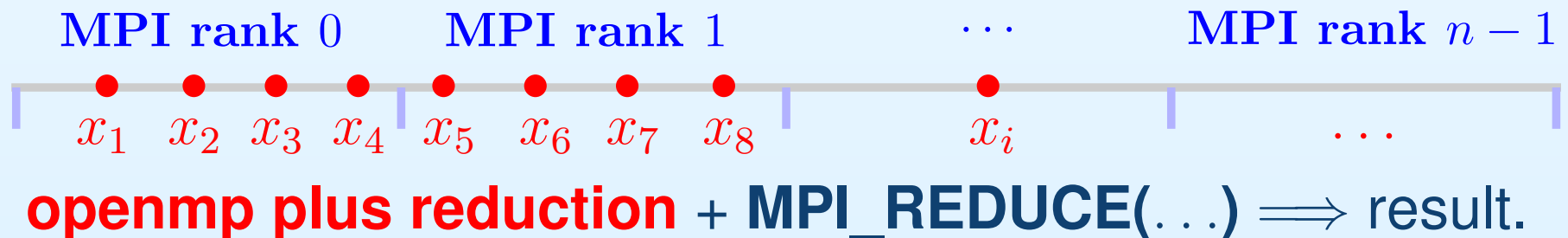
$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

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

- Pure MPI:



- Hybrid MPI+OpenMP:





## Hybrid model

Fortran  
Pure MPI

```
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)

```

**Fortran  
Hybrid**

- 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**:

```

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

```

Fortran  
Hybrid

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

No. of MPI tasks	No. of threads	Wall time (sec)
16	1	0.45986
8	2	0.46088
4	4	0.46389
2	8	0.46021
1	16	0.45919



## Hybrid model

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

```
top - ..., 1 user,load average: 186.71,84.11,32.83
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

PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
64761 xiaoxu 20 0 203m 3472 2868 R 1.3 0.0 0:00.04 mpi_openmp_pi_f
64762 xiaoxu 20 0 203m 3388 2800 R 1.3 0.0 0:00.04 mpi_openmp_pi_f
64763 xiaoxu 20 0 203m 3392 2804 R 1.3 0.0 0:00.04 mpi_openmp_pi_f
64764 xiaoxu 20 0 203m 5436 2804 R 1.3 0.0 0:00.04 mpi_openmp_pi_f
```

- 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:
  - (1) **Compute**-bound (aka. “**CPU**”-bound);
  - (2) **Cache**-bound;
  - (3) **Memory**-bound;
  - (4) **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?
  - (1) you're the developer of the application;
  - (2) 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)  $\times$  **Operation Intensity** (FLOP/byte)

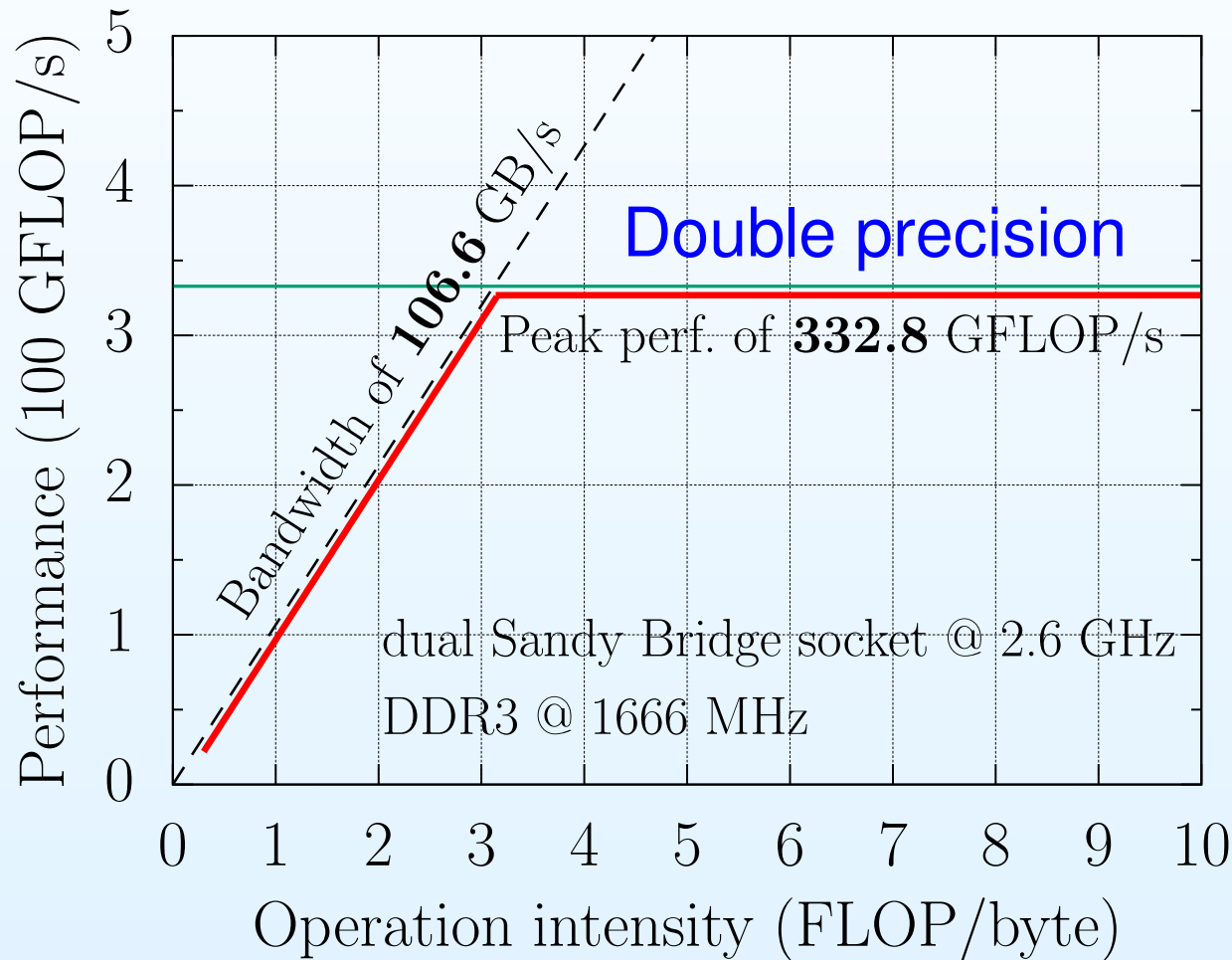
$$y(\text{FP Perf.}) = k(\text{BW.}) \times (\text{OI.})$$

- 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, ...;



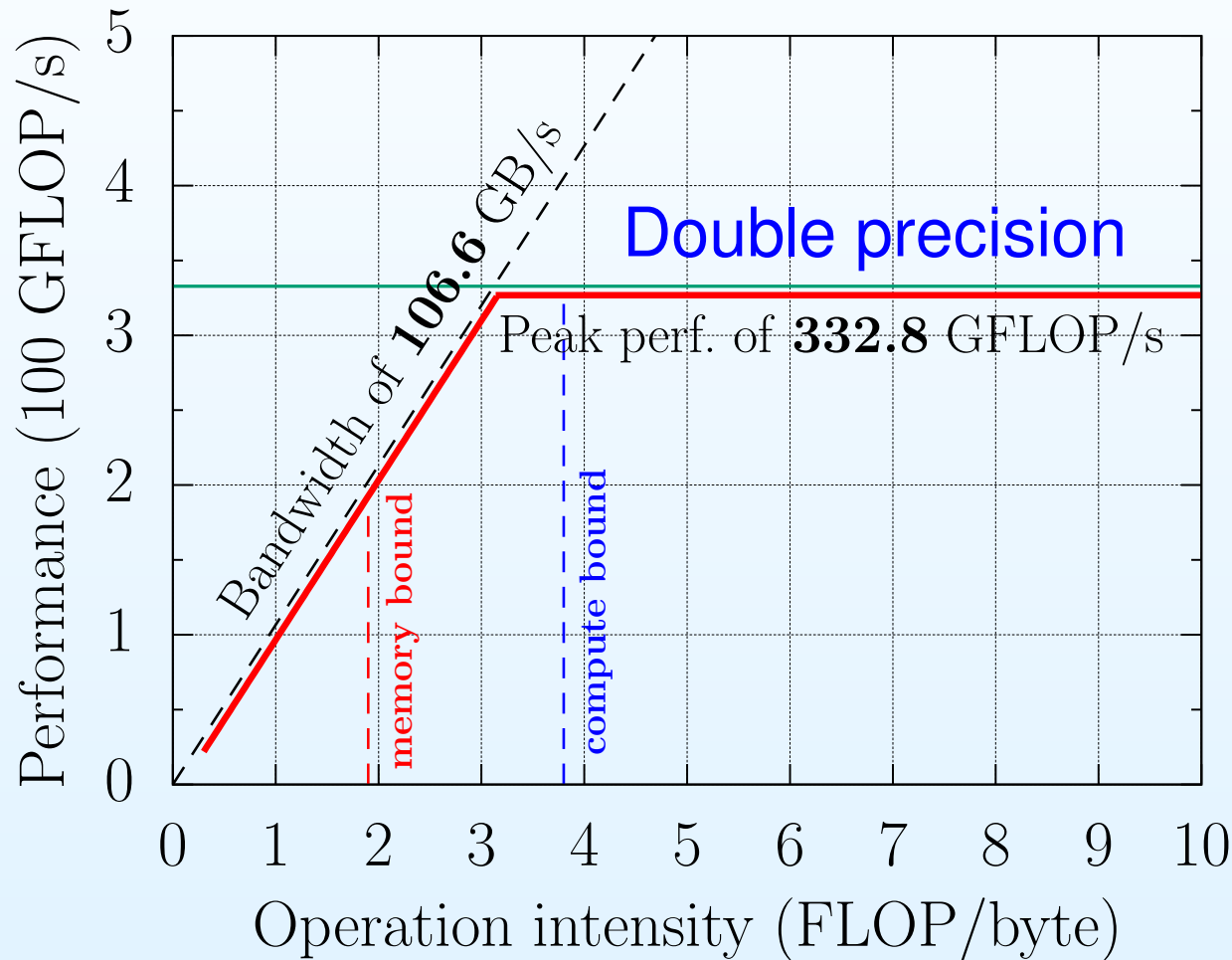
### Roofline model

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:

$OI \ll$

3.2 FLOP/byte

Compute bound:

$OI \gg$

3.2 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 **Ol.** 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);

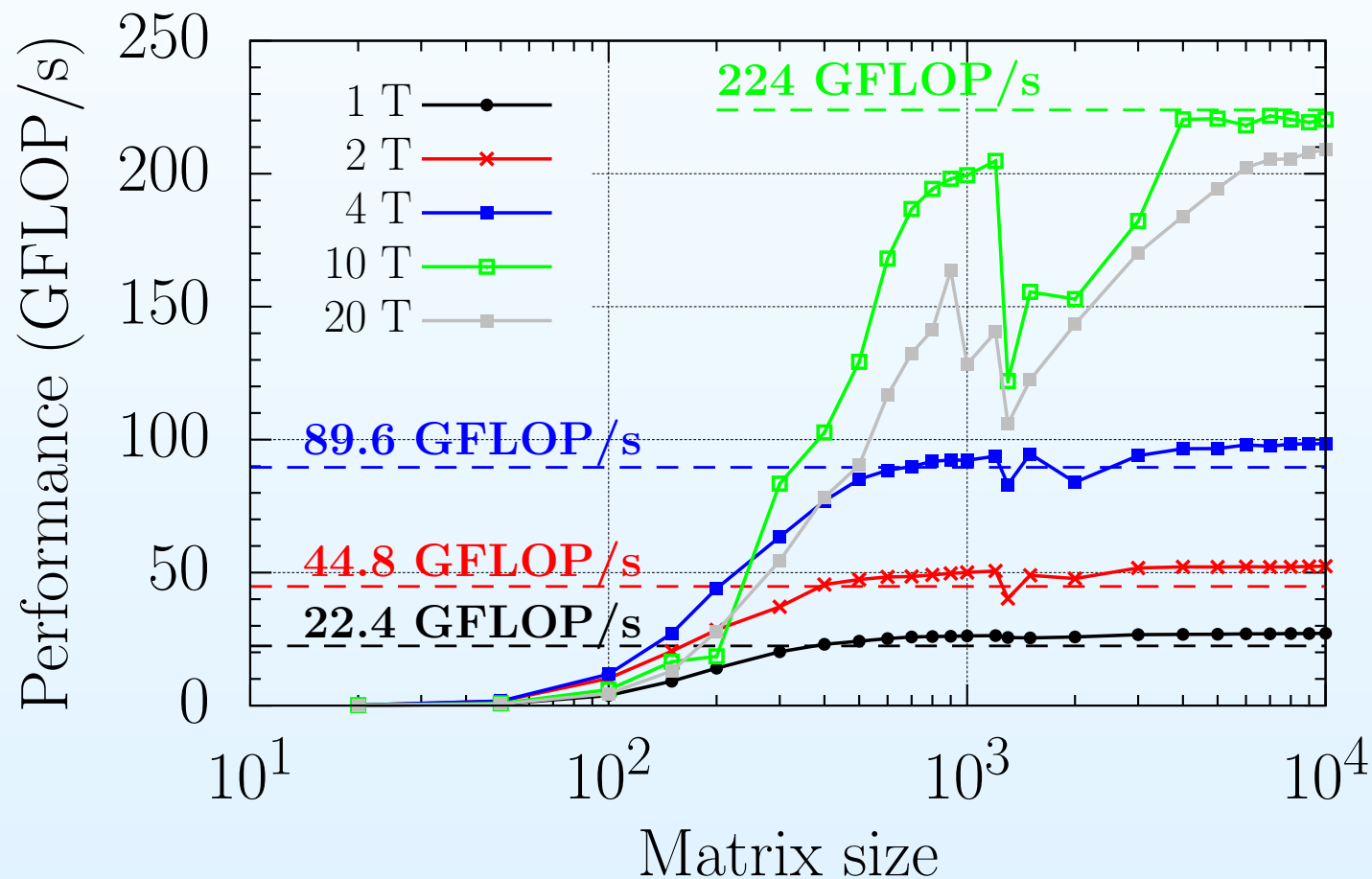
```
1  const int nsize = 10000;
2  const int iteration = 20;
3  // allocate the matrices.
4  // initialize the matrices.
6  for (k=0; k<iteration; k++)    //  $C = A \times B$ .
7  { cblas_dgemm(CblasRowMajor, CblasNoTrans, \
8    CblasNoTrans, nsize, nsize, nsize, \
9    alpha, matrix_a, nsize, matrix_b, nsize, \
10   beta, matrix_c, nsize); }
11  perf = 2.0 * dsize * dsize * dsize * (double) \
12  (iteration) / elapsed_time / 1.e+6;
```

C/C++



## 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  $1,000 \times 1,000$ ?

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

- 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**:

```
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
```

Fortran

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



## Memory bound

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

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

<sup>†</sup> 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 `mkldomatcopy`;

```

1  for (k=0; k<iteration; k++)
2      mkldomatcopy('R', 'T', nsize, nsize, \
3      alpha, matrix_a, nsize, matrix_b, nsize);

```

C/C++

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

Machine	4,000	20,000	40,000
<b>SuperMIC</b>	23.93	21.22	18.68
<b>SuperMIC</b> <sup>†bigmem</sup>	17.96	18.01	18.08
<b>QB2</b> <sup>†k40</sup>	20.96	18.05	15.45

<sup>†k40</sup> 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?



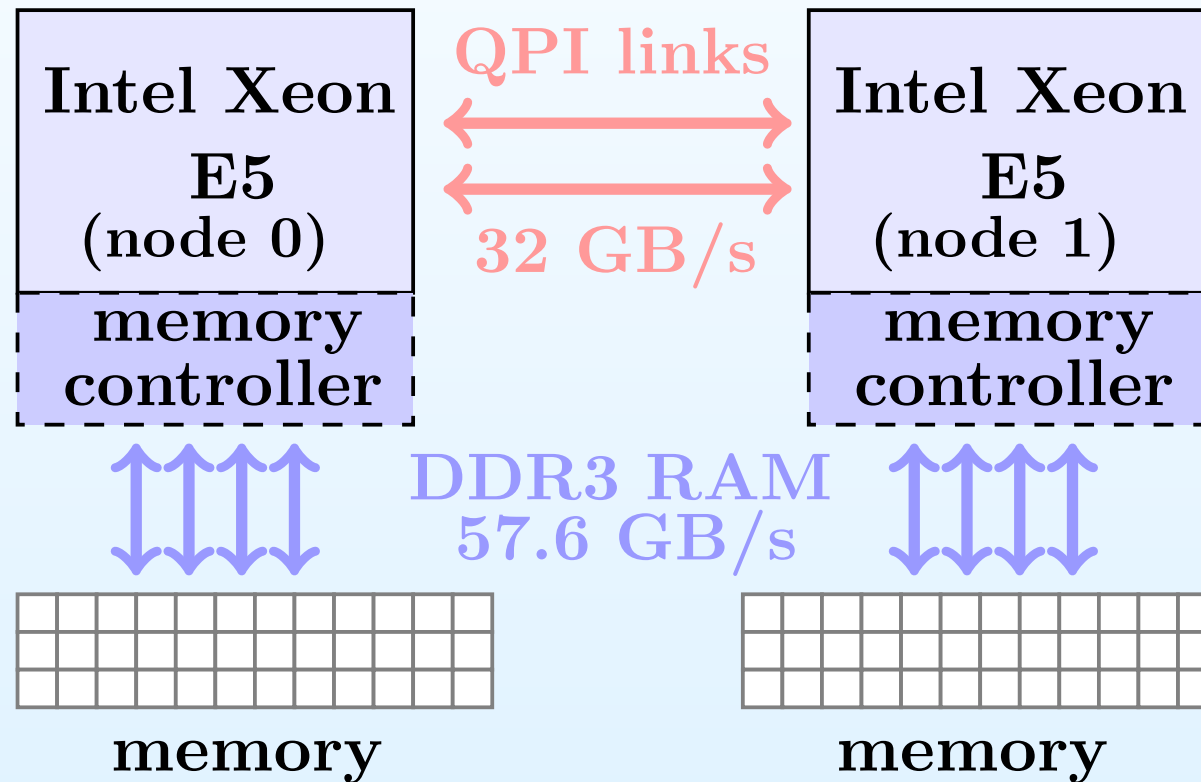
## Socket and processor level

- 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

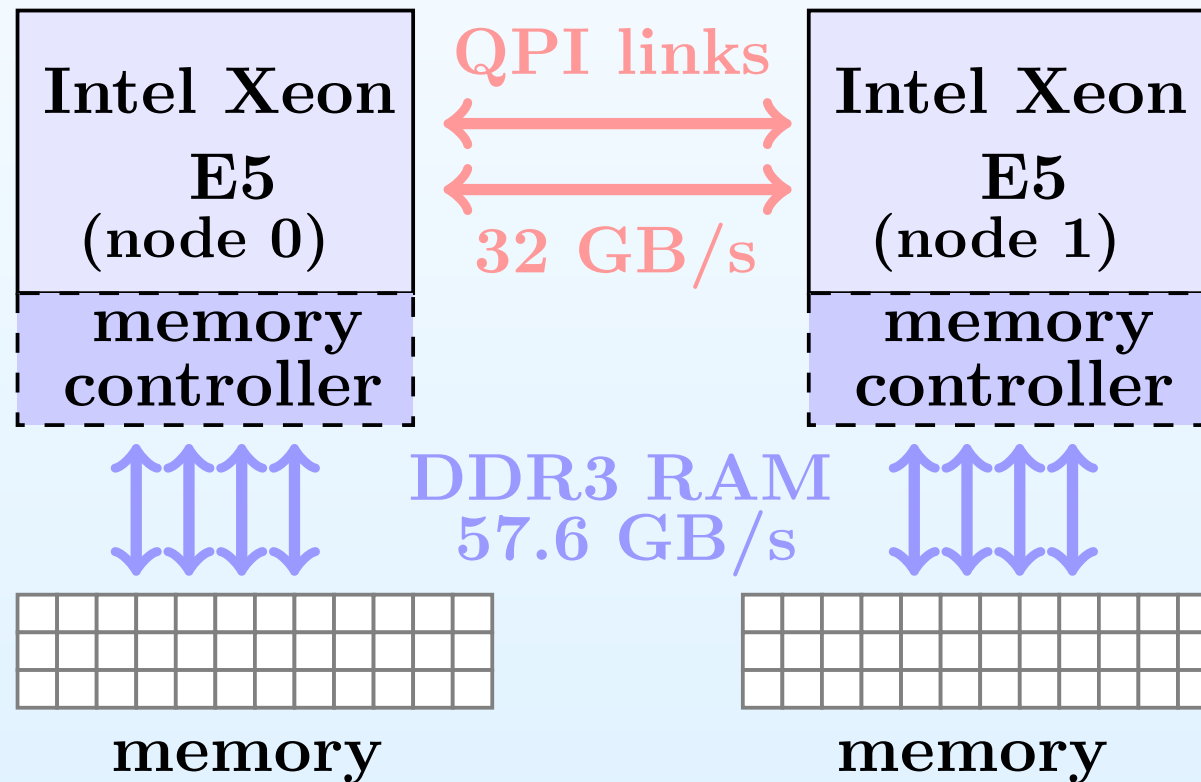
- The **NUMA** (non-uniform memory access) architecture;
- The key point in **NUMA** is about **shared memory**;
- Furthermore, it has been implemented as **ccNUMA** (cache coherent NUMA);





## 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 `numctl` 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 --physcpubind
```



## Socket and processor level

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

1 Measuring idle latencies (in ns)...

2                      Numa node

3 Numa node            0            1            # DDR3 1866 MHz

4            0            72.3        123.0        # SuperMIC reg. node

5            1            123.5        72.9        # NUMA/UMA = 1.7

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:

1 Measuring idle latencies (in ns)...

2                      Numa node

3 Numa node            0                      1            # DDR4 2400 MHz

4                      0            87.2            128.6    # SuperMIC bigmem node

5                      1            129.8            87.9    # NUMA/UMA = 1.5

1 Bandwidths are in GB/sec

2 Using Read-only traffic type

3                      Numa node

4 Numa node            0                      1            # DDR4 2400 MHz

5                      0            67.78            23.49    # SuperMIC bigmem node

6                      1            23.41            67.94    # UMA/NUMA = 2.9



## Socket and processor level

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

1 Measuring idle latencies (in ns)...

2                      Numa node

3 Numa node            0            1            # DDR3 1866/1600 MHz

4            0            71.4            122.9            # QB2 reg. node

5            1            123.6            71.5            # NUMA/UMA = 1.7

1 Bandwidths are in GB/sec

2 Using Read-only traffic type

3                      Numa node

4 Numa node            0            1            # DDR3 1866/1600 MHz

5            0            53.46            25.02            # QB2 reg. node

6            1            25.03            46.82            # UMA/NUMA = 2.2



## Socket and processor level

1 Measuring idle latencies (in ns)...

2                      Numa node            # QB2 bigmem node, 1.6

3 Numa node	0	1	2	3
4            0	129.4	202.1	192.0	200.9
5            1	202.2	130.4	199.6	194.2
6            2	196.4	196.0	129.0	193.4
7            3	201.4	195.9	191.4	128.2

1 Bandwidths are in GB/sec

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

3                      Numa node            # QB2 bigmem node, 4.2

4 Numa node	0	1	2	3
5            0	53.52	12.65	12.68	12.44
6            1	12.70	54.39	12.65	12.65
7            2	12.48	12.50	53.71	12.66
8            3	12.63	12.52	12.71	54.37



# Core level parallelism



## Core level (vectorization)

Instruction set	Register width	Processor
SSE	128-bit	Pentium (1997)
SSE2	128-bit	Pentium III (1999)
AVX	256-bit	Xeon Sandy Bridge (2011)
AVX	256-bit	AMD Bulldozer (2011)
AVX2	256-bit	Xeon Haswell (2013)
AVX2	256-bit	Xeon Broadwell (2014)
AVX2	256-bit	AMD Carrizo (2015)

- 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;

```
1 // vectorized or non-vectorized loop      v0 C/C++
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<nsize; i++)
9             vector_a[i] = vector_a[i] + vector_b[i] \
10            + 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;

```
1 // vectorized or non-vectorized loop      v0 C/C++
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<nsize; i++)
9             vector_a[i] = vector_a[i] + vector_b[i] \
10            + 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;

```
1 // vectorized or non-vectorized loop      v0 C/C++
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<nsize; i++)
9             vector_a[i] = vector_a[i] + vector_b[i] \
10            + vector_c[i] + vector_d[i] + vector_e[i];
```

- Record the speedup of `vec/no-vec` with varying `nsize`;
- `nsize` = 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/C++

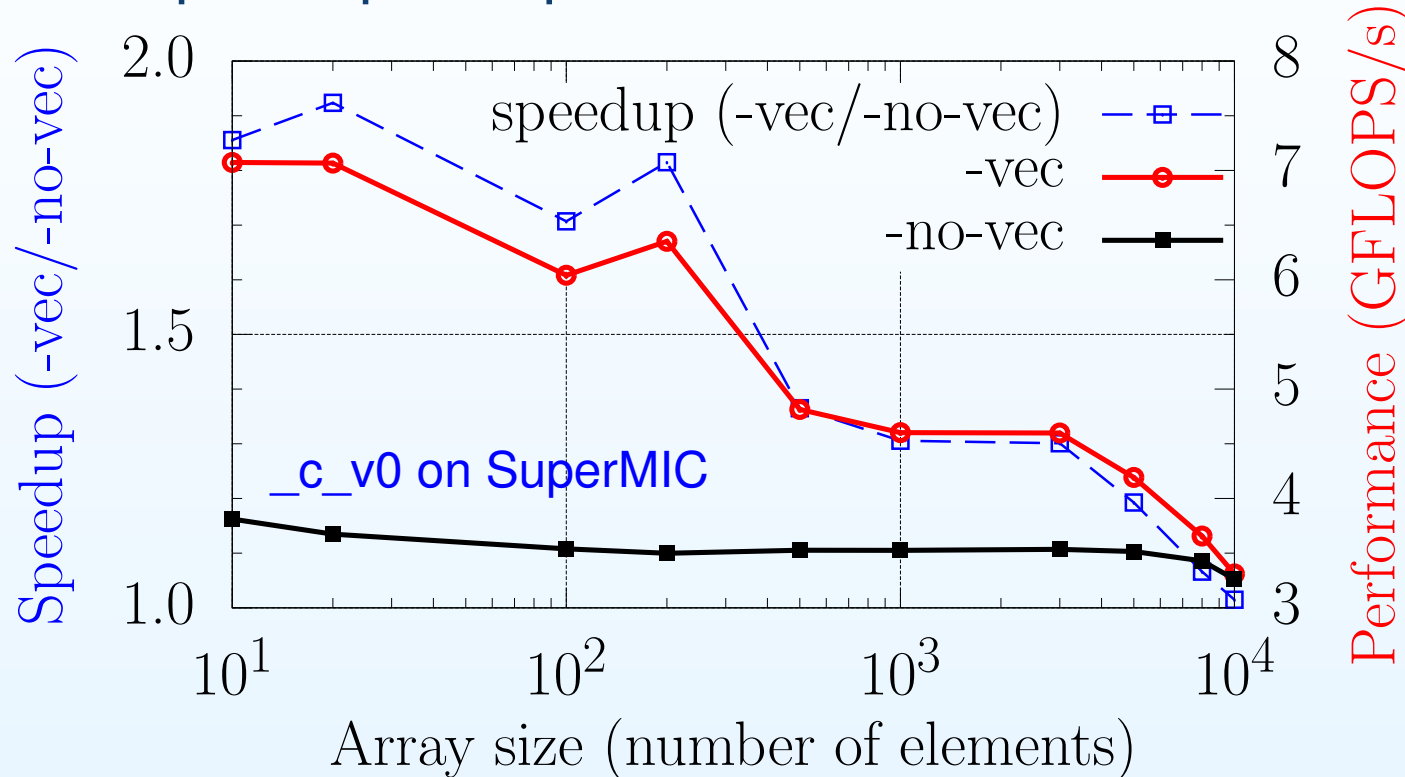
```
..._v0.c(52): (col. 3) remark: LOOP WAS VECTORIZED  
..._v0.c(78): (col. 4) remark: LOOP WAS VECTORIZED  
..._v0.c(77): (col. 4) remark: loop was not  
vectorized: not inner loop
```

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



## Core level (vectorization)

- Check the speedup and performance:



- 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;

```
1 // vectorized or non-vectorized loop      v1 C/C++
2 int main (int argc, char *argv[])
3     ...
4     nosize = atoi(argv[1]);
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] \
10        + 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;

**v1 C/C++**

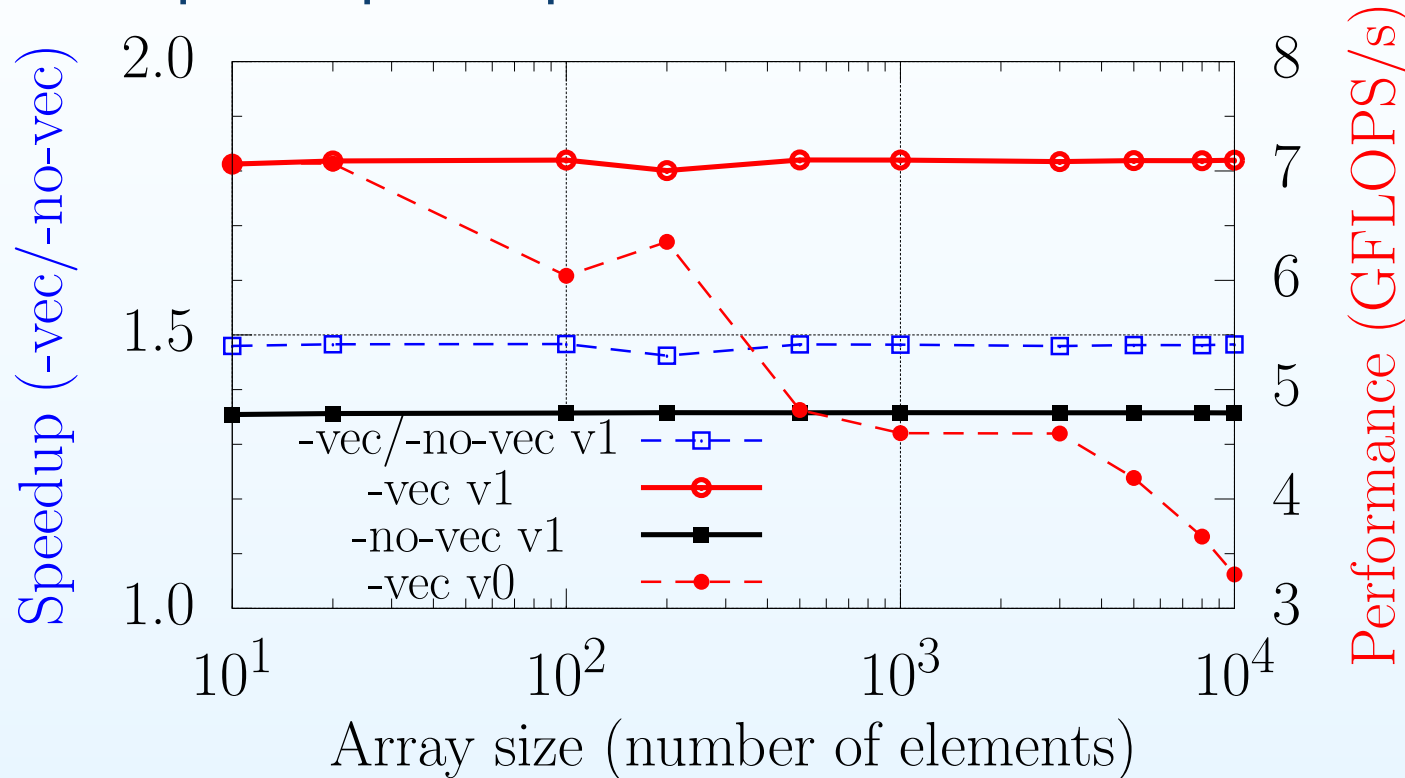
```
..._v1.c(50): (col. 3) remark: LOOP WAS VECTORIZED  
..._v1.c(75): (col. 4) remark: PERMUTED LOOP WAS  
                    VECTORIZED  
..._v1.c(76): (col. 4) remark: loop was not  
                    vectorized: not inner loop
```

- **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,  $2.5/8 \simeq 31\%$  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:
  - **MPI**:  $\sim 10-1000\times$ ;
  - **OpenMP**:  $\sim 10-40\times$ ;
  - **Memory affinity** on multiple-socket:  $\sim 2-4\times$ ;
  - **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](mailto:sys-help@loni.org)