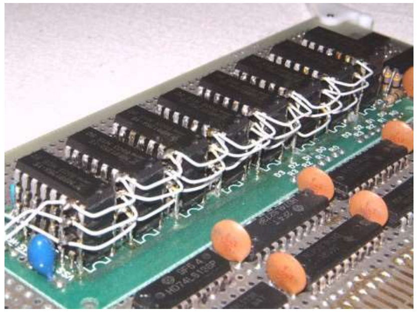
Fundamentals of MCS
Computer Architecture #3
“Parallelism”

Toshio Endo

endo@is.titech.ac.jp

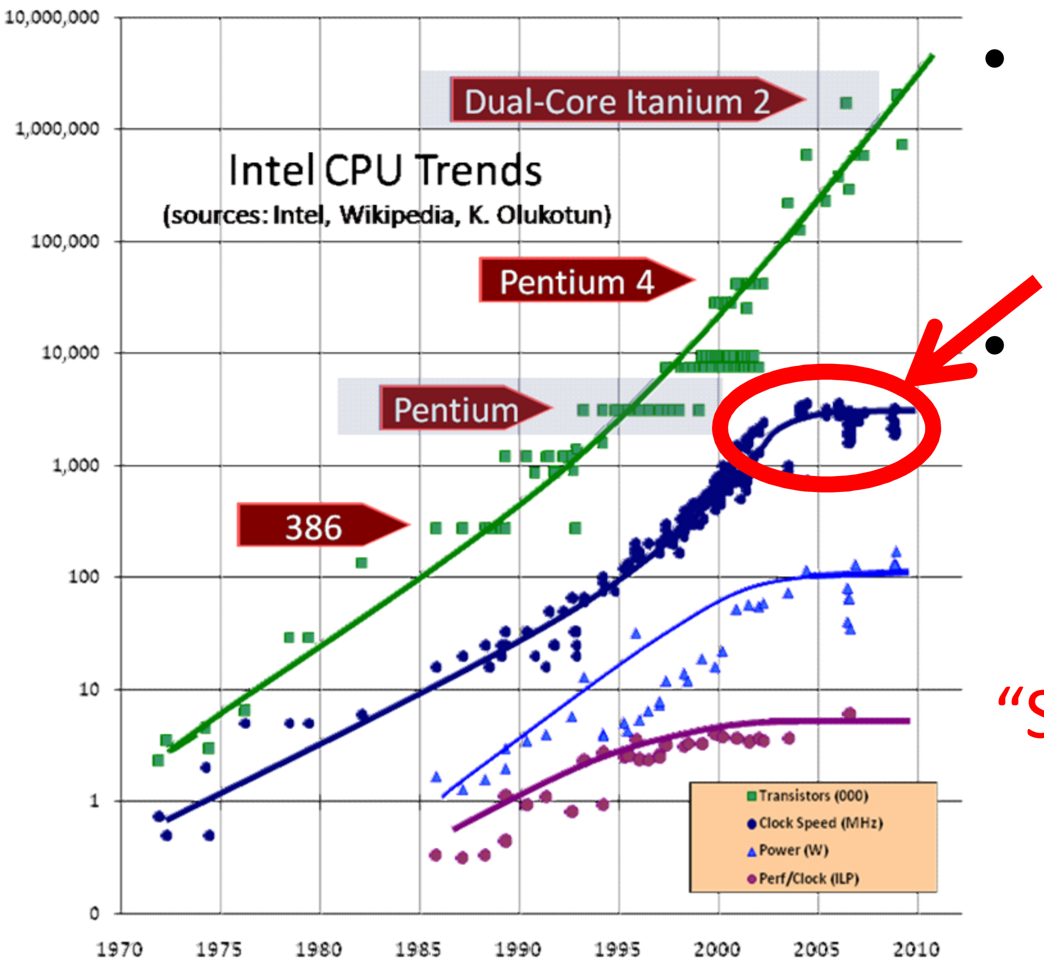
www.el.gsic.titech.ac.jp

Improvement of Processor Clock

	Around 1980		Present
CPU	2MHz → 1clock = 500ns 	x1000	2GHz → 1clock = 0.5ns 
Memory	Access time = 2000ns(?) 	x40(?)	Access time = 50ns or more 

Will processor clocks become further faster?
Will we have 2THz CPUs 30years later?

Trend of Processors

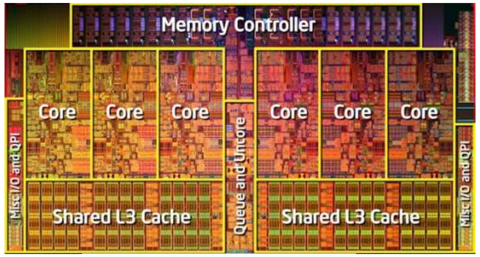


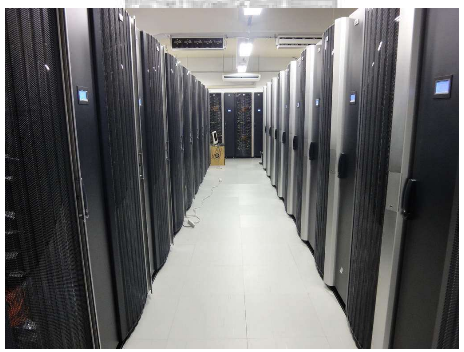
- Growth of clock speed of processors has stopped around 2003
- Instead, the number of cores is increasing
  - Even cellphones have dual-core processors

"Slow&Parallel" approach

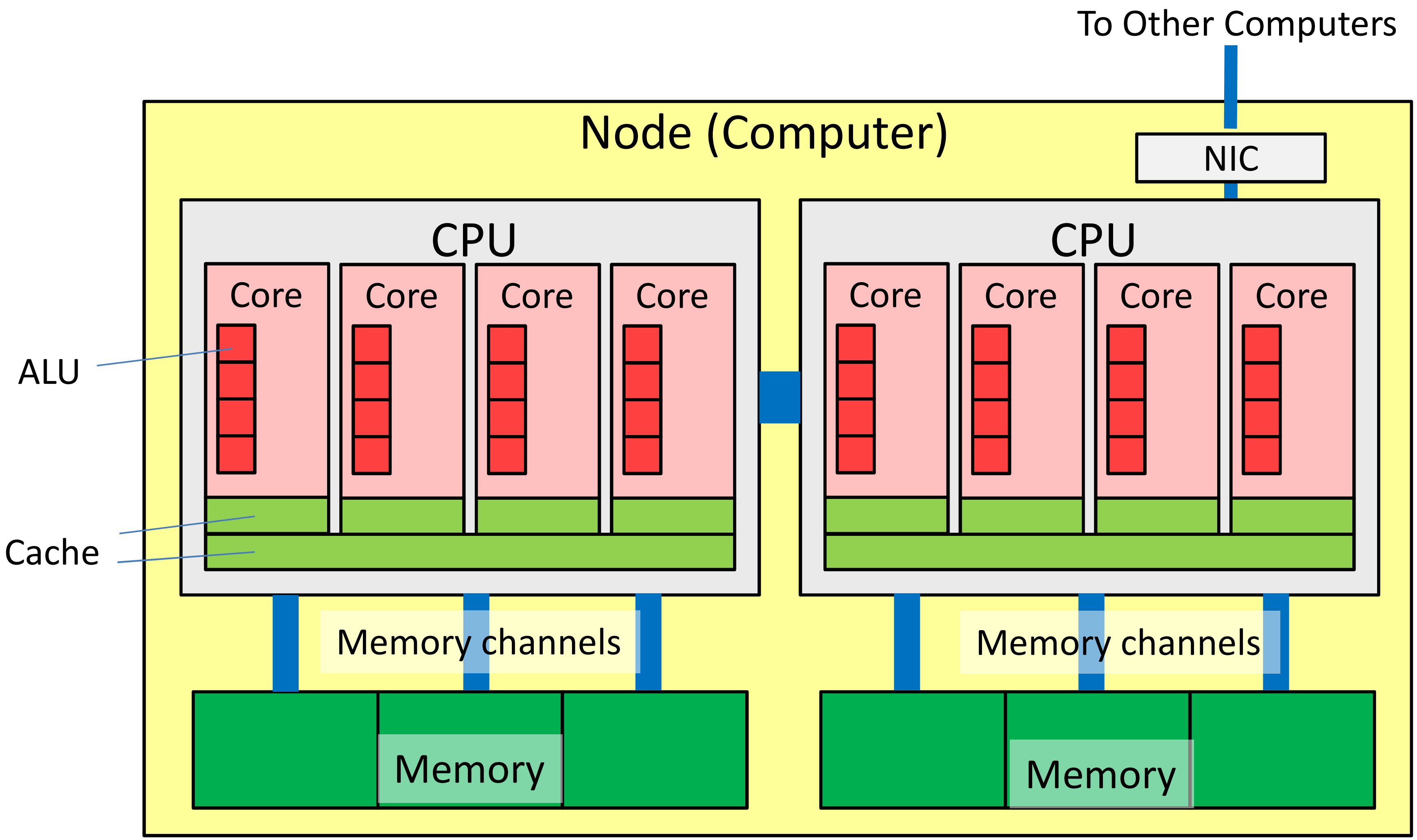
Hierarchy of Parallelism

- SIMD parallelism
  - multiple operations can be done simultaneously (SIMD)
  - MMX, SSE, AVX
- Multi-Core (Multi-CPU) parallelism
  - Multiple cores can work cooperatively
  - Pthread, Java thread, OpenMP
- Using GPU (skipped in this lecture)
  - Thousands of GPU cores
  - CUDA, OpenCL, OpenACC
- Multi-Node parallelism
  - Multiple nodes can work cooperatively
  - Socket, Hadoop, MPI

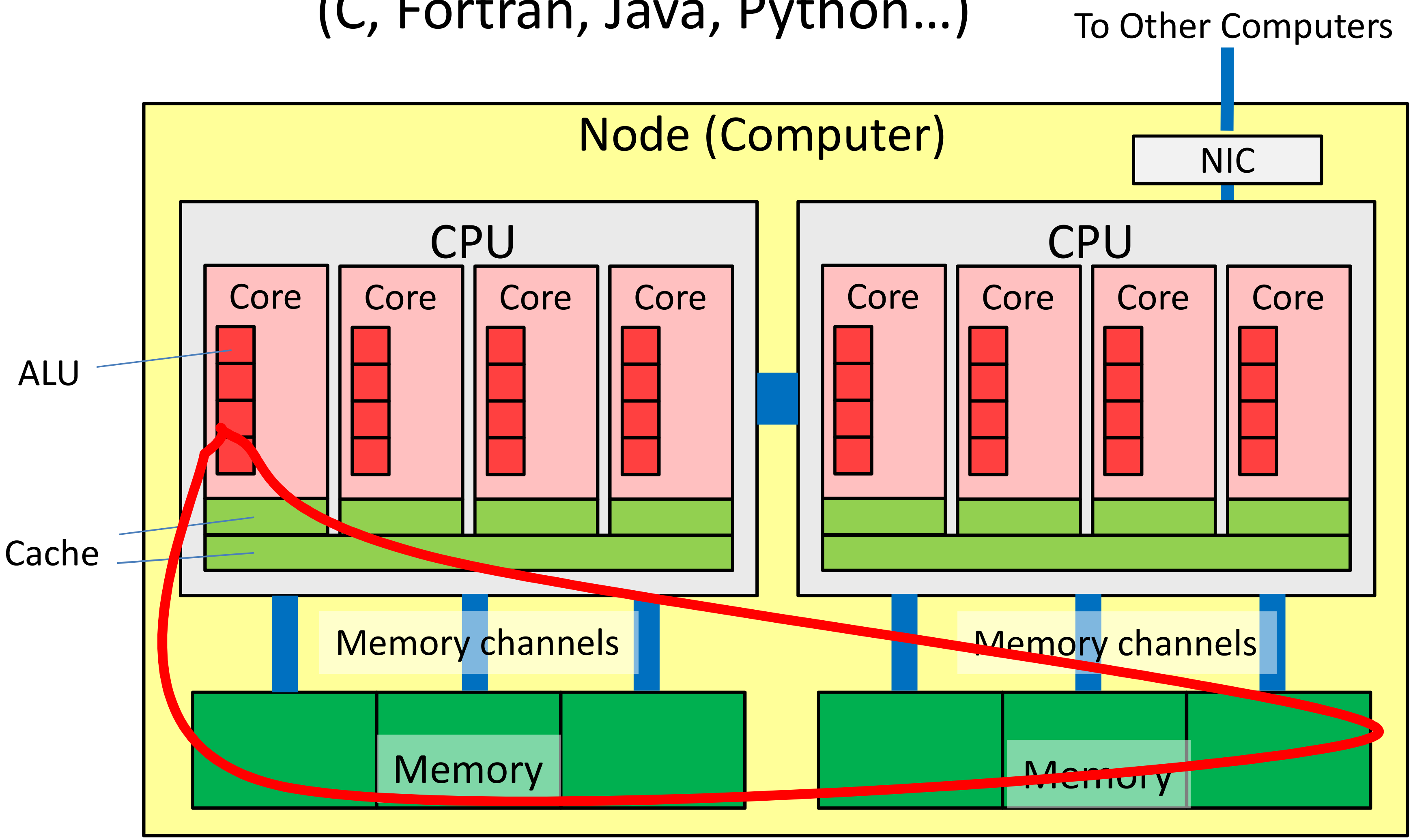




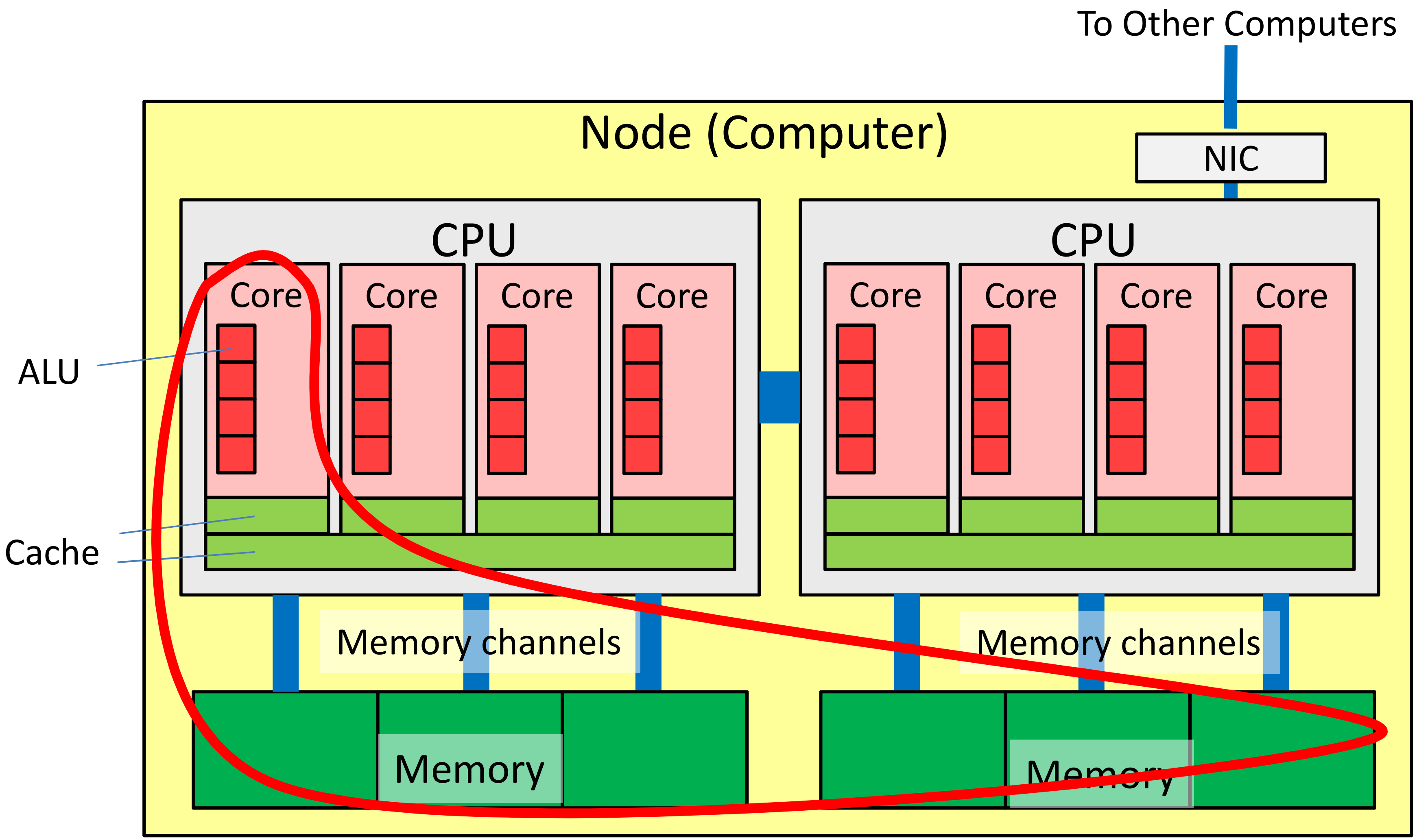
Hierarchy of Parallelism in Modern Computers



No Parallelism
With Typical Programming Languages
(C, Fortran, Java, Python…)



SIMD parallelism
(SSE, AVX…)



Harnessing Intra-core Parallelism: SIMD Programming

- SIMD = Single Instruction Multiple Data
  - Multiple operations can be done simultaneously
- Tightly coupled with CPU architecture
  - In Intel CPUs, MMX → SSE → AVX
    - TSUBAME2 nodes support SSE
- Usually written by assembly language
  - gcc and Intel compilers supports special methods called “intrinsics”
  - _mm_load_pd, _mm_mul_pd, _mm_add_pd…
  - Some recent compilers can produce SIMD instructions automatically!

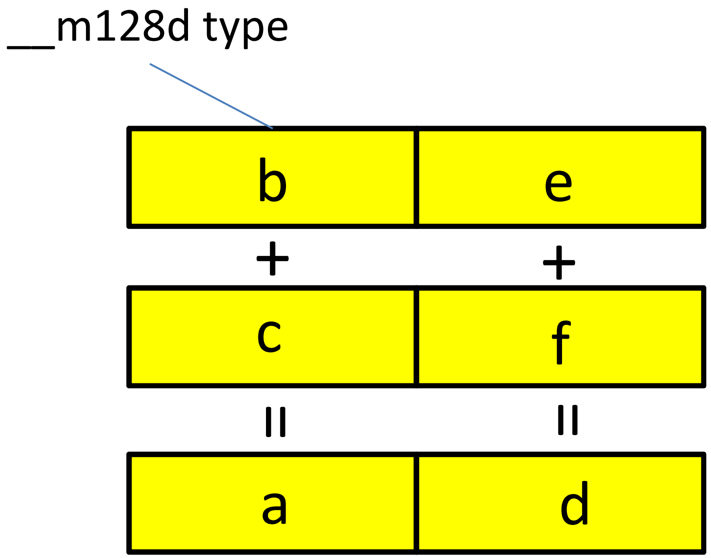
Basics of SSE

With normal operations

```
a = b+c;
d = e+f;
```

With SSE operations

```
ad = _mm_add_pd(be, cf);
```



- In SSE, 128 bit (16byte) packed type is used
  - __m128d value can contain 2 double values
  - __m128 value can contain 4 single values
- In AVX, 256 bit packed type is used

SSE Operations

Use gcc or Intel compiler

- __m128d a = _mm_load_pd(p);
  - Makes _m128d value that contains p[0], p[1]
  - Hereafter, a0, a1 mean contents of a
  - pd means “packed double”
- __m128d c = _mm_add_pd(a, b);
  - c0 = a0+b0; c1 = a1+b1;
- __m128d c = _mm_mul_pd(a, b);
  - c0 = a0*b0; c1 = a1*b1; (not dot product)
- __mm_store_pd(p, a);
  - p[0] = a0; p[1] = a1;
- Also there are “packed single” version
  - Such as __m128 a = _mm_load_ps(p);

Matrix Multiply with SSE

With normal operations

```
for (j = 0; j < n; j++) {
  for (l = 0; l < k; l++) {
    double blj = B[l+j*ldb];
    for (i = 0; i < m; i++) {
      double ail = A[i+l*lda];
      C[i+j*ldc] += ail*blj;
    } }   }
```

A working program is avaiable at
~endo-t-ac/mcs/sse/mmsse.c
In TSUBAME

With SSE operations

```
#include <emmintrin.h>
#include <xmmintrin.h>
        :
for (j = 0; j < n; j++) {
    for (l = 0; l < k; l++) {
        __m128d bv = _mm_load_pd1(&B[l+j*ldb]);
        double *ap = &A[l*lda];
        double *cp = &C[j*ldc];
        for (i = 0; i < m; i += 2) {
          __m128d av = _mm_load_pd(ap);
          __m128d cv = _mm_load_pd(cp);
          av = _mm_mul_pd(av, bv);
          cv = _mm_add_pd(cv, av);
          _mm_store_pd(cp, cv);
          ap += 2;
          cp += 2;
        }}}
```

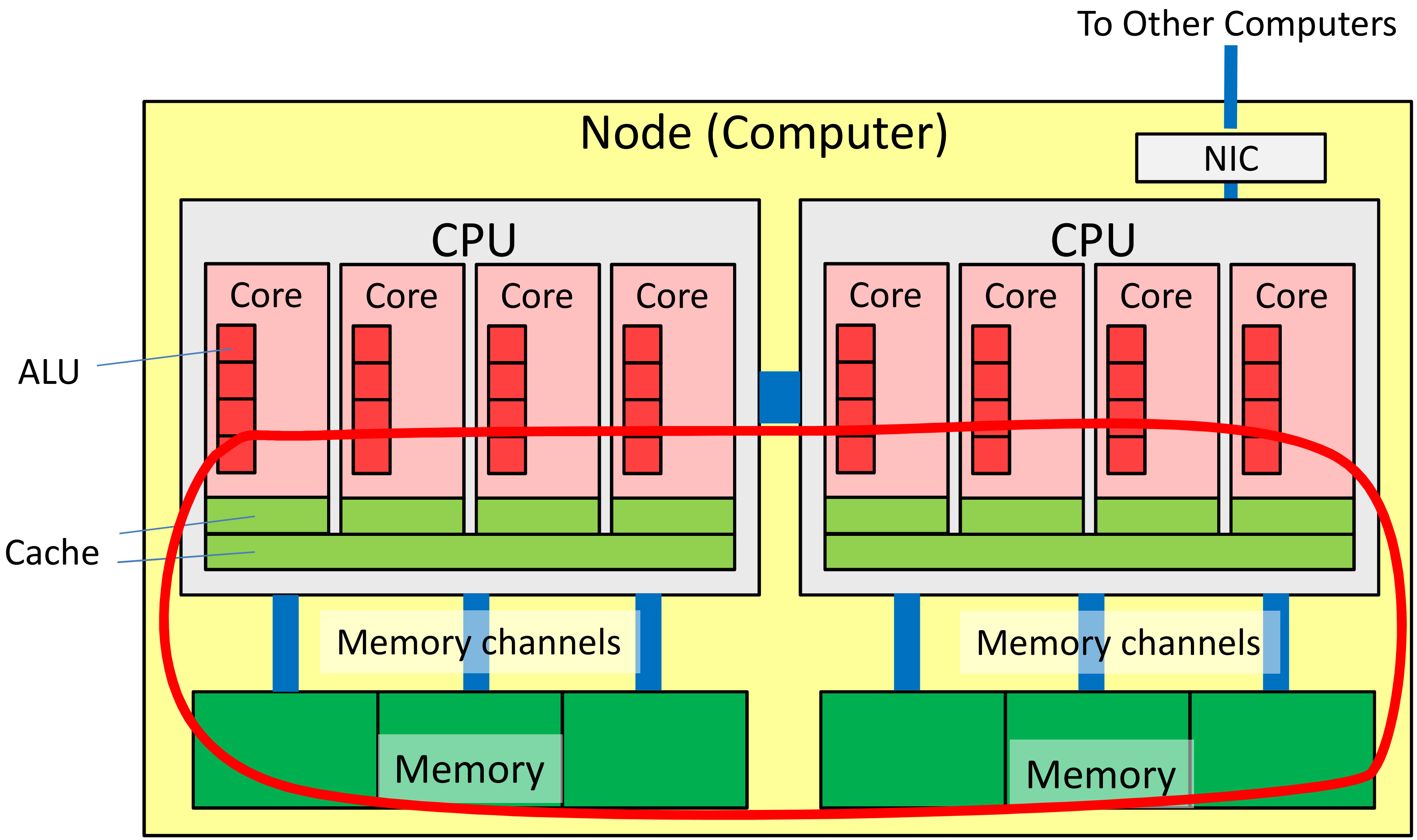
Matrix Multiply Performance

Imple	IJL	ILJ	JIL	JLI	LIJ	LJI	JLI-SSE	JLI-SSE (unroll)	GotoBLAS (1core)
Speed (Gflops)	0.25	0.12	0.25	1.92	0.12	1.65	2.82	3.71	11.8

- JLI version with SSE gets faster
- Loop unrolling improves performance
- Still slower than GotoBLAS
  - Please refer to: *K. Goto, R. Geijn: Anatomy of high-performance Matrix Multiplication, ACM TOMS 2008*

Multi-Core/Multi-CPU Parallelism

(Pthreads, Java threads, OpenMP…)



MIMD = Multiple Instruction Multiple Data

- Multiple streams of operations (threads or processes) can work cooperatively

Threads and Processes

- In a computer, several processes are running
  - Each process has distinct memory space
- In a process, one or several threads are running
  - Threads in a process share a memory space
- Threads and processes are scattered among cores by OS

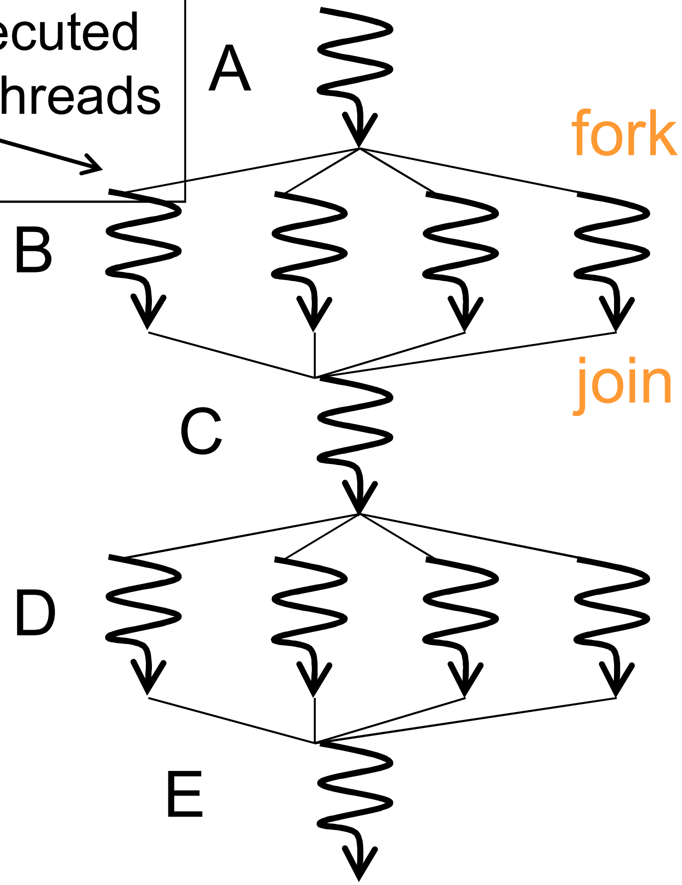
- Thread programming:
  - Pthreads, Java threads, OpenMP, Cilk…
  - OpenMP, Java threads internally uses pthreads
  - For more details, please attend "practical parallel computing" / "実践的並列コンピューティング" in the summer semester / 夏学期. (Sorry in Japanese)

Basics of OpenMP

```
#include <omp.h>

int main()
{
    A;
#pragma omp parallel
    {
            B;
    }
    C;
#pragma omp parallel
    D;
    E;
}
```

B, D are executed by multiple threads in parallel



Blocks or sentences just after #pragma omp parallel becomes "parallel regions"

Using Parallel Regions of OpenMP

- The number of threads is determined by "OMP_NUM_THREADS" environment variable
- You can write any code fragment in parallel regions
  - Unlike SIMD, each thread can execute completely different operations
- To parallelize for loop, "parallel for" idiom may be useful

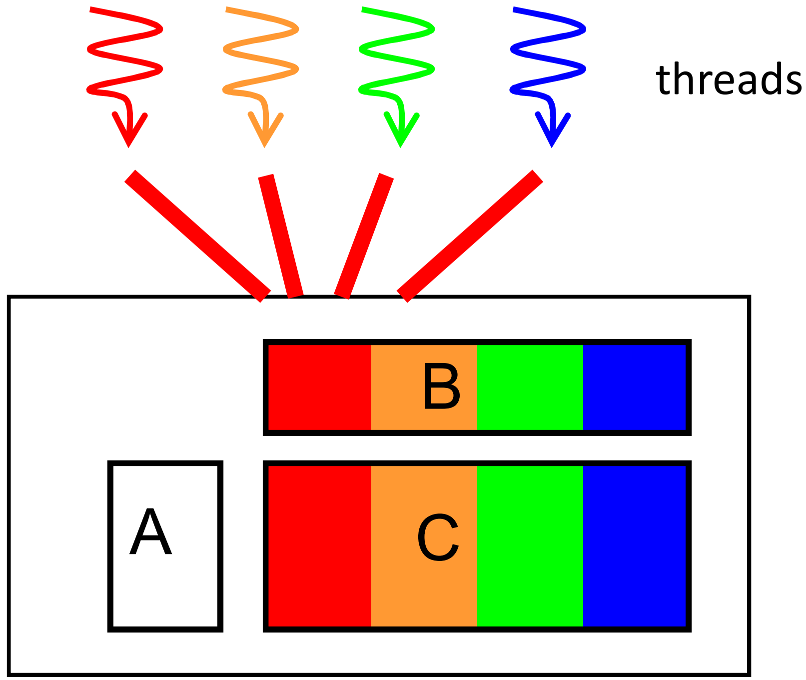
```
#pragma omp parallel for
  for (i = 0; i < 100; i++) {
     a[i] = b[i]+c[i];
  }
```

100 tasks are distributed by OMP_NUM_THREADS threads automatically

Matrix Multiply with OpenMP

- j-loop or i-loop can easily parallelized

```
#pragma omp parallel for
for (j = 0; j < n; j++) {
  for (l = 0; l < k; l++) {
    double blj = B[l+j*ldb];
    for (i = 0; i < m; i++) {
      double ail = A[i+l*lda];
      C[i+j*ldc] += ail*blj;
    } }  }
```



Matrix A is touched by all threads

- l-loop has difficulty due to dependency between operations

Performance of Matrix Multiply

1 thread

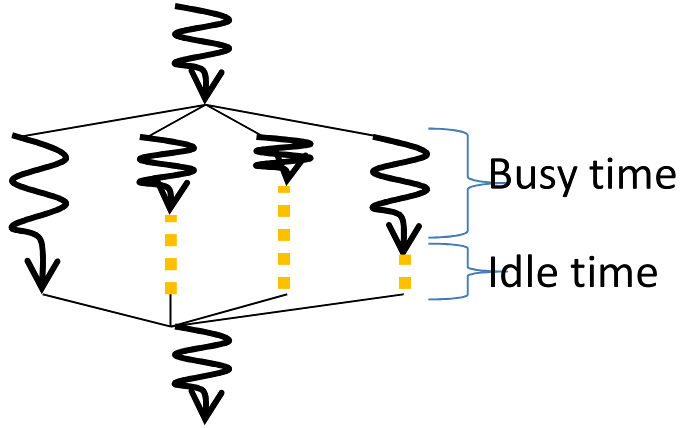
Program	JLI	JLI-SSE	JLI-SSE (unroll)	GotoBLAS
Speed (Gflops)	1.92	2.82	3.71	11.8

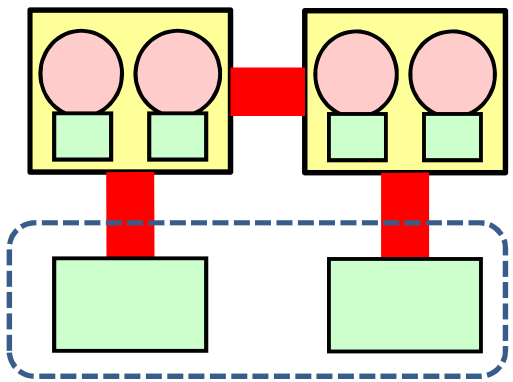
12 threads

Program	JLI	JLI-SSE	JLI-SSE (unroll)	GotoBLAS
Speed (Gflops)	20.3	25.3	26.7	119
	10.6x	9.0x	7.2x	10.1x

Why Is Speed-up <12 ?

- Load imbalance
  - If some threads have more tasks, they become bottleneck
  - It does not seems the case now
- Critical path
  - Such as mutual execution (Skipped in this lecture, sorry!)
- Contention of memory access
  - Bandwidth of memory channel is finite
- False sharing (later)

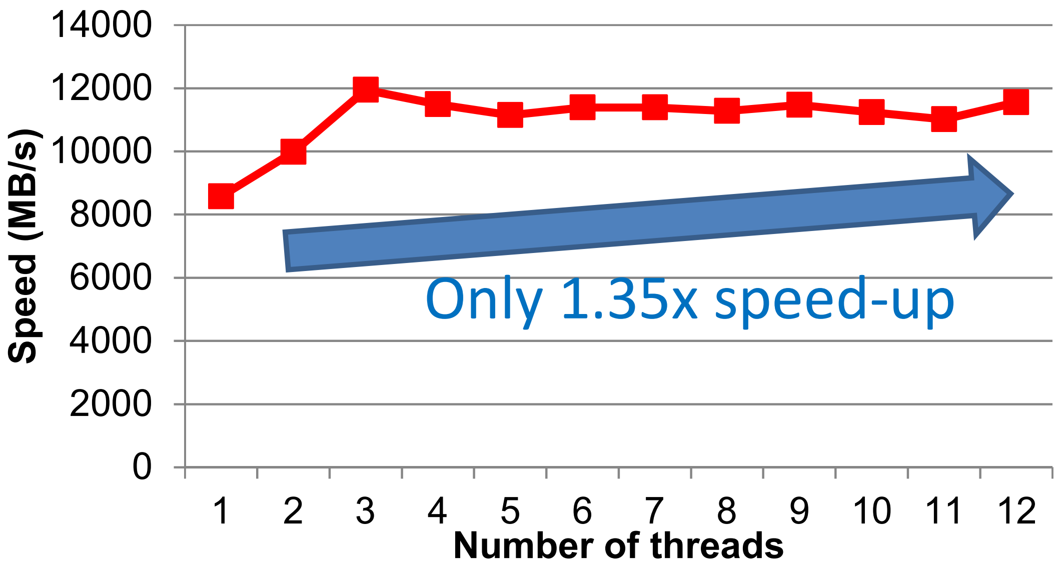




Effects of Memory Contention

To observe memory contention, memory copy program was measured

```
#pragma omp parallel for
for (i = 0; i < n; i++)  a[i] = b[i];
```

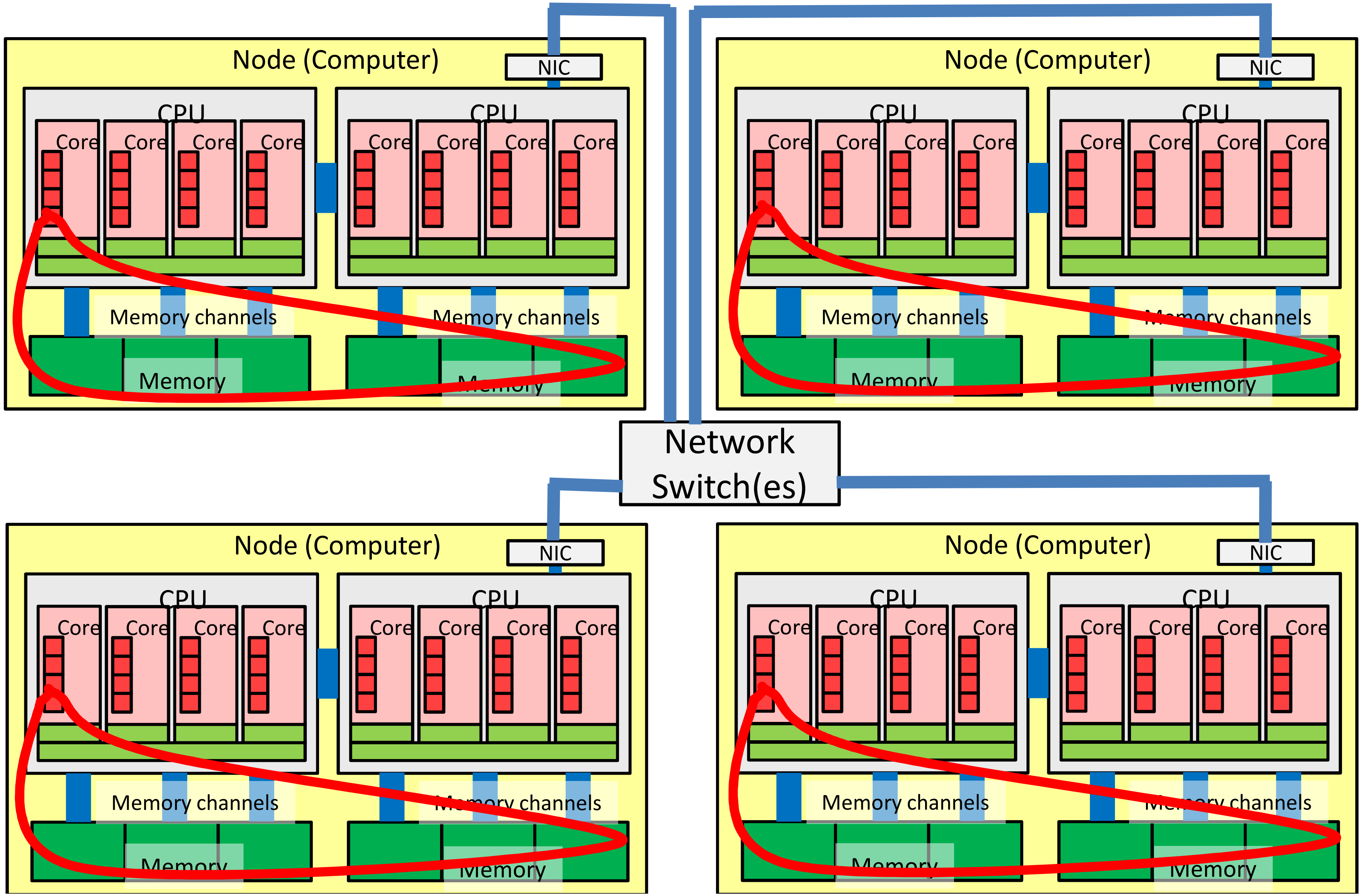


In 12 threads (12 cores) case, cost of each memory miss increases by 12/1.35=8.9 times

- Speed was calculated by (Read bytes + Write bytes) / time

Multi-Node Parallelism With Communication

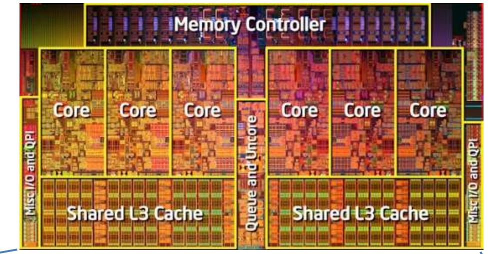
(Sockets, Hadoop, MPI…) → Appeard in Next lecture

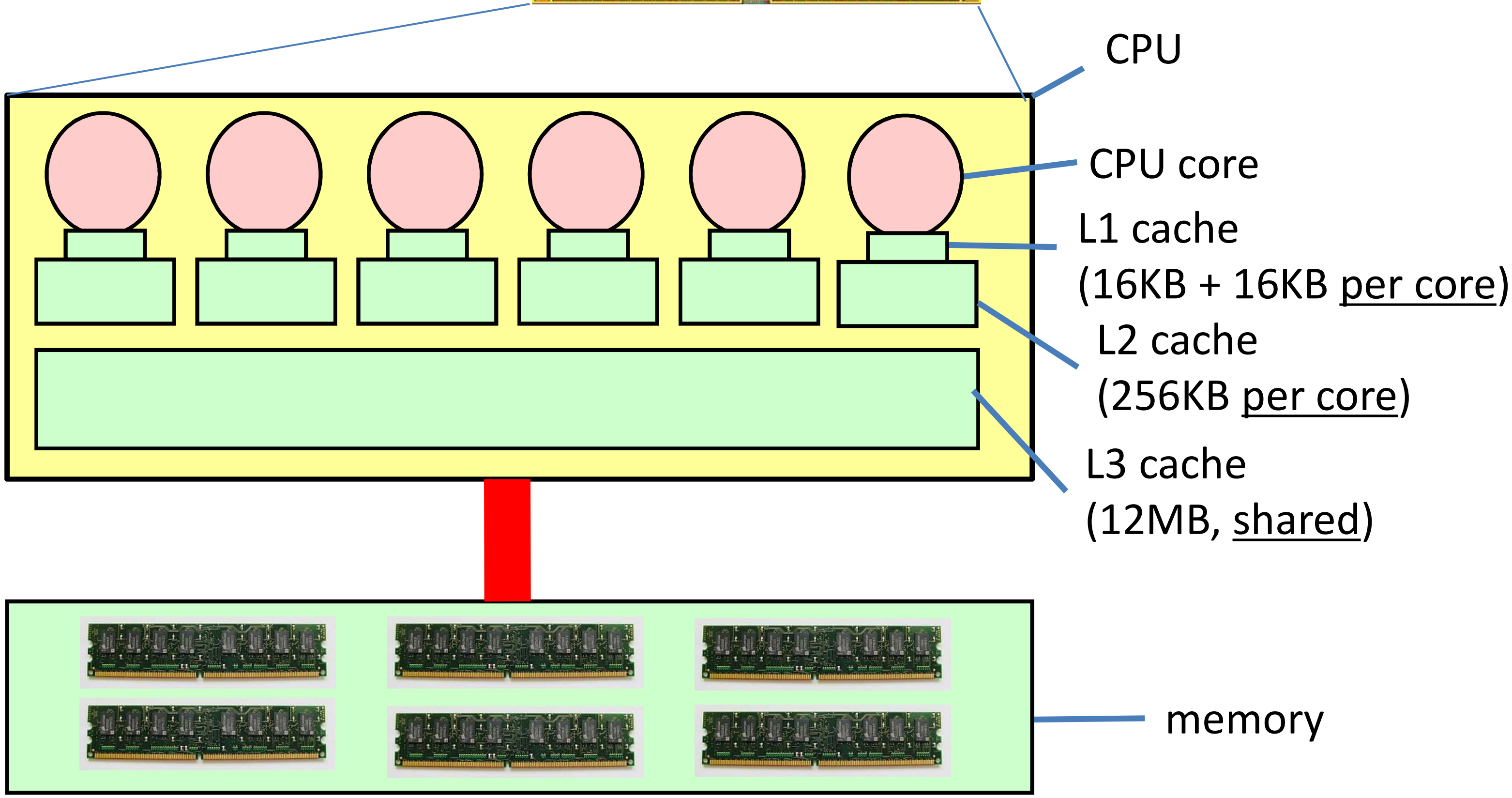


CACHE IN PARALLEL SYSTEMS

Cache of Multicore Processors

Intel Xeon X5670
(CPU used in TSUBAME2)

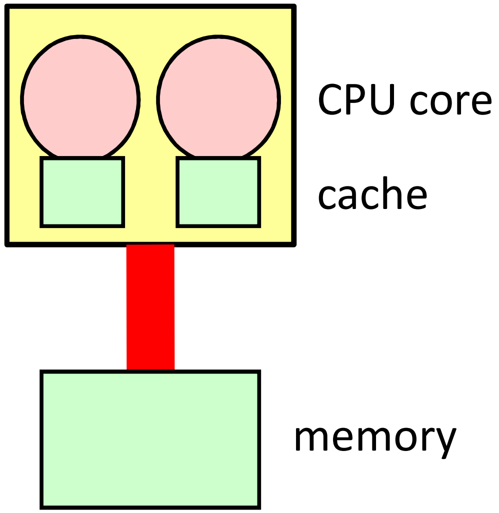




Cf) "lscpu" Linux command gives information

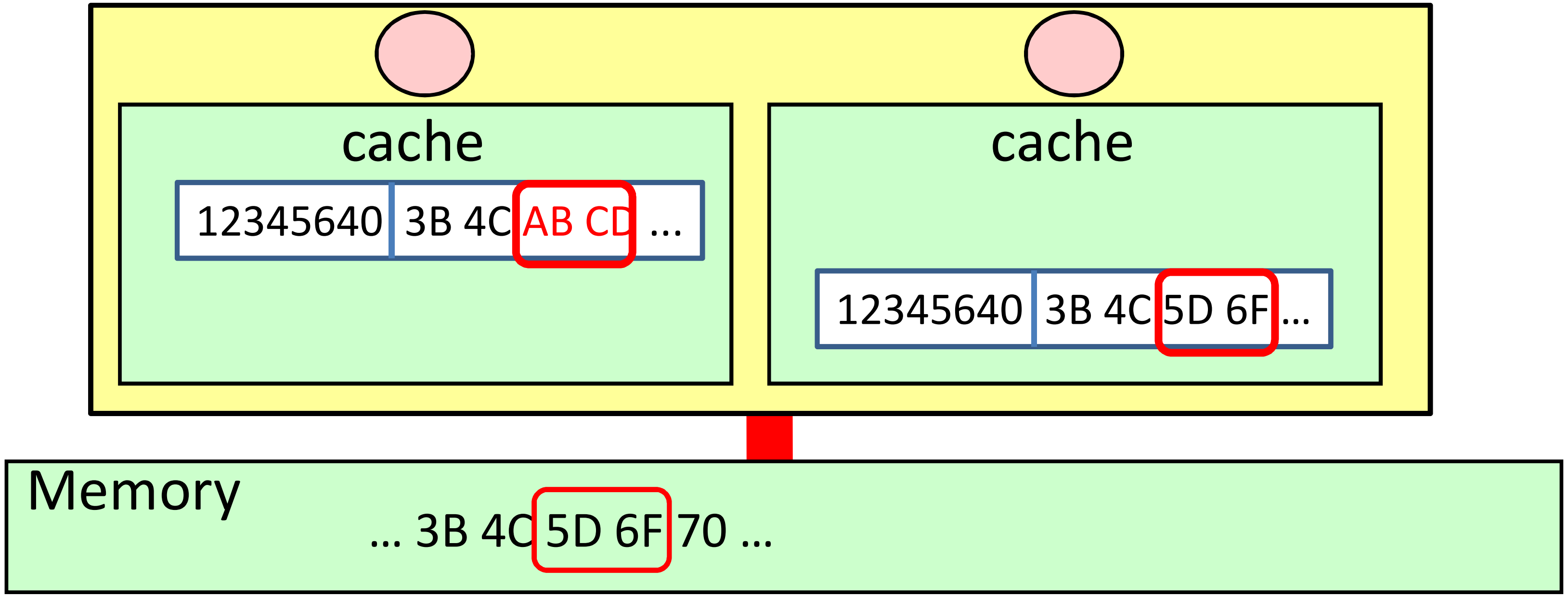
Assumptions in this Lecture

- We discuss a simplified cache model



- There is only a single level cache
- Each core has a distinct (non-shared) cache
- We assume write-back protocol
  - Written new data is reflected to main memory lazily
- We consider only sequential consistency, not relaxed consistency

Difficulty in Parallel Cache



- How can we avoid reading "wrong" data?
  - We call this "keeping consistency"
  - There is protocol to keep consistency among caches

MSI Protocol

- MSI protocol is the simplest protocol to keep consistency
- Each cache line in cache is in one of 3 modes
  - Modified
  - Shared
  - Invalid

cache

Address	Data	Mode
456789C0	23 45 67 89 … 24 35 46 57	M
12345640	3B 4C 5D 6F … … … 20	S
---	---	I
34FEDC00	11 22 33 44 … FF 00 11 22	S

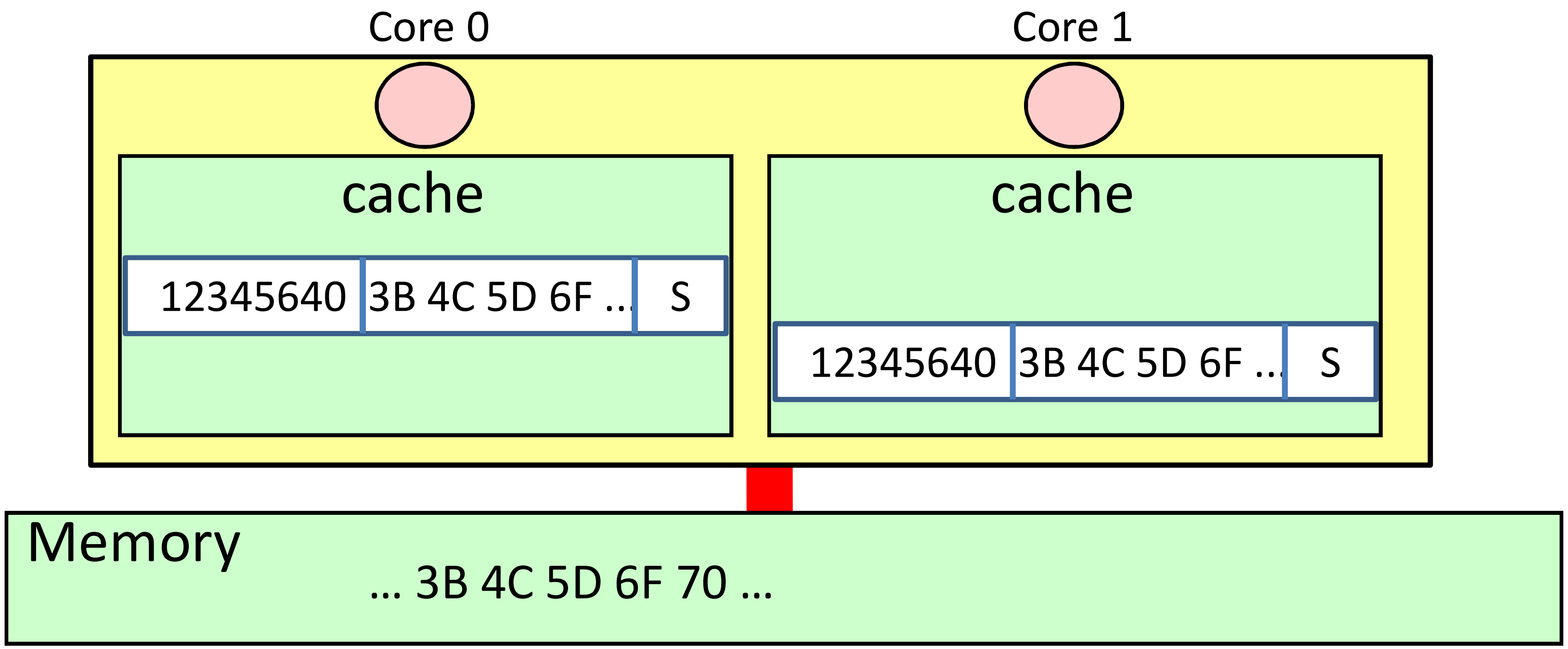
64 bytes

Modes in MSI Protocol

- Invalid:
  - This line is not used
- Shared:
  - This line may be shared by several caches
  - Contents of line is same as other cache or memory
- Modified:
  - Contents of line has been modified by CPU core
  - Contents of line may differ from memory
  - There must only line for the address among caches

Cache Behavior in MSI Protocol (1)

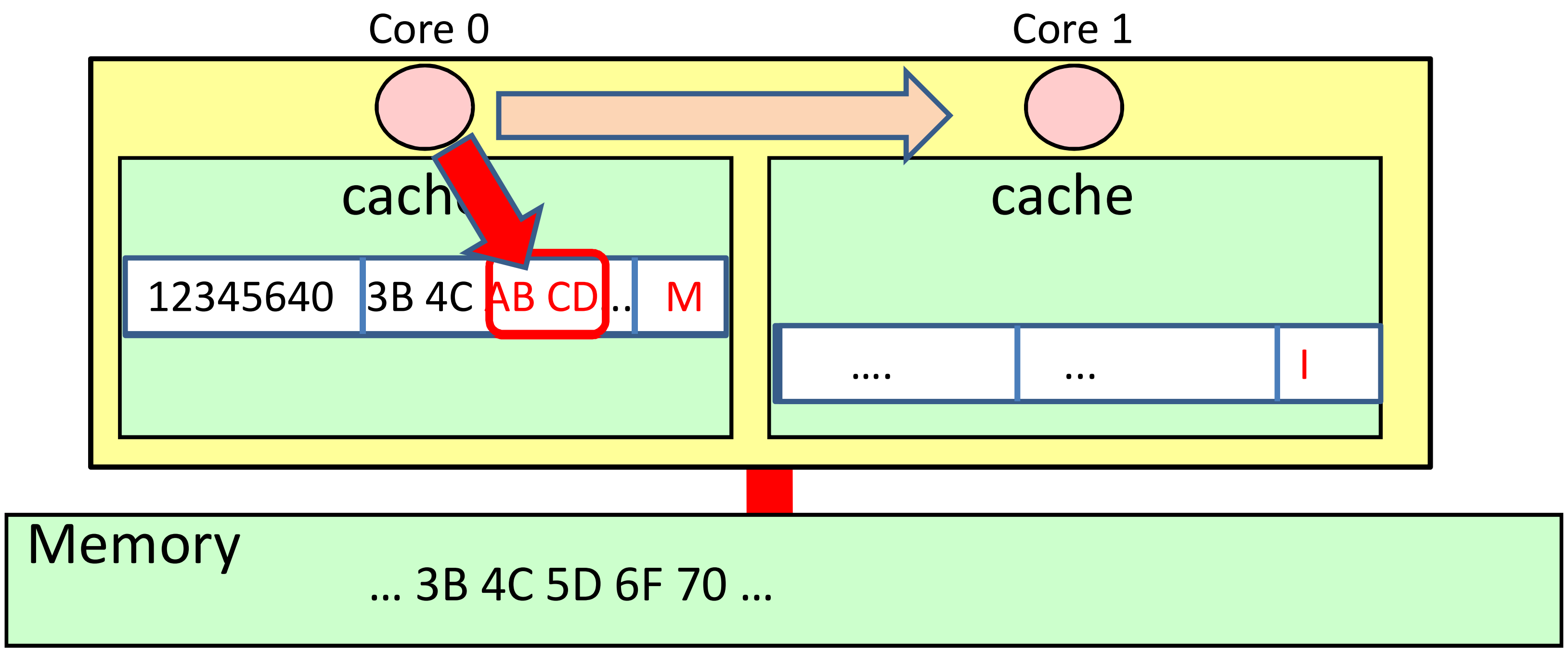
- Here line 12345640 is “Shared”



- What happens if core 0 executes “write”?

Cache Behavior in MSI Protocol (2)

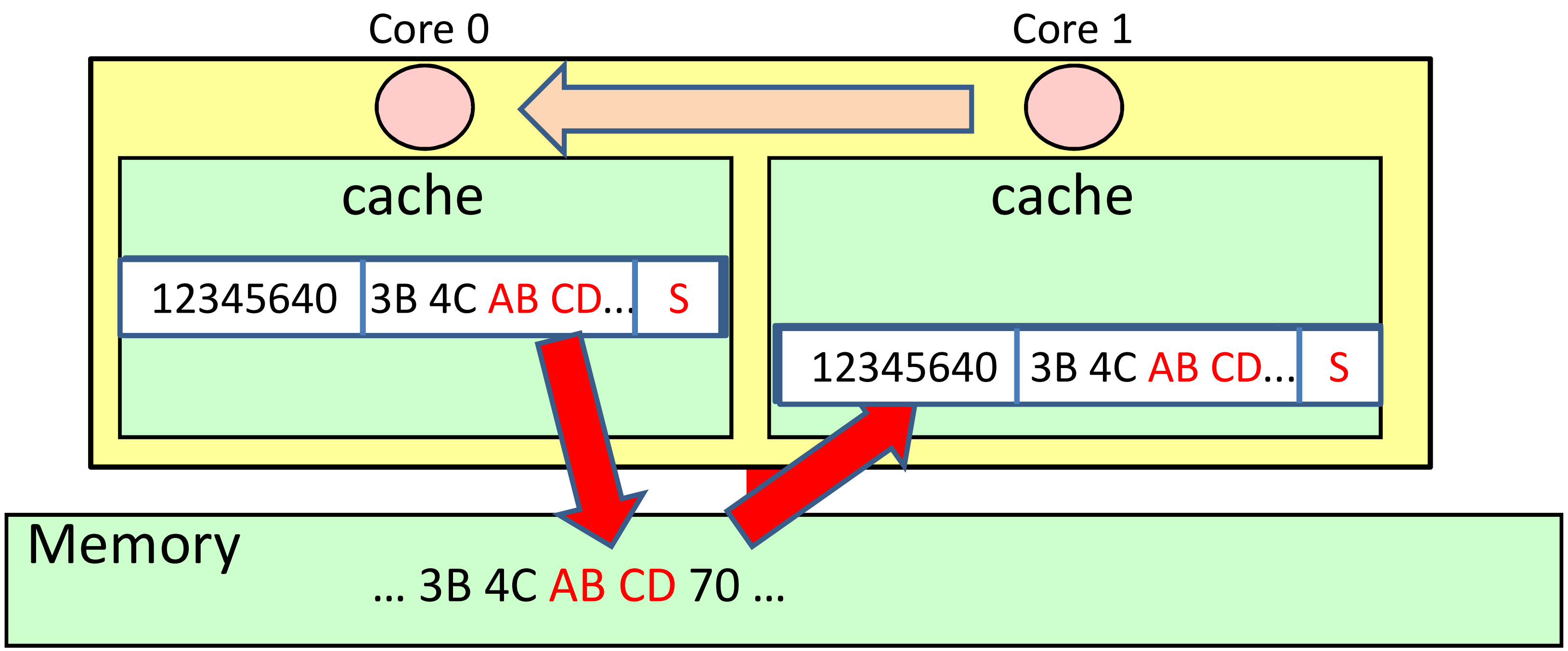
- Write to a shared line includes:
  - Make other shared line (if exist) invalid (called invalidate)
  - Make own line modified
  - Write to its own line



- What happens if core 1 executes “read”?

Cache Behavior in MSI Protocol (3)

- Read-miss (by core 1) includes:
  - If there is modified line in other cache
    - let the owner copy back the contents to memory
    - Make owner's line shared
  - Core 1 Reads from memory



- It includes 2 memory operations (>200 clocks)

Other Protocols

Extensions of MSI

- MOSI
  - Modified, Owned, Shared, Invalid
- MESI
  - Modified, Exclusive, Shared, Invalid
- MOESI
  - Modified, Owned, Exclusive, Shared, Invalid
- MESIF
  - Modified, Exclusive, Shared, Invalid, Forward

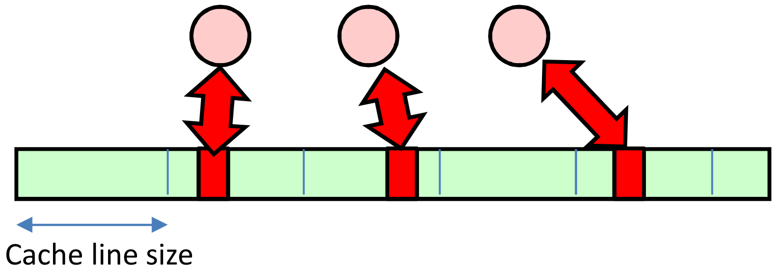
Approaches for Performance Improvement

- The key for performance is avoiding frequent operation involving memory
  - Mode-change involves memory operations → heavy
- If data is read-only, sharing is harmless
  - The shared cache line remains “S” mode
- If data is read-write (RW), it is better to make it discrete (localize)

Sharing RW Data
What is False Sharing?

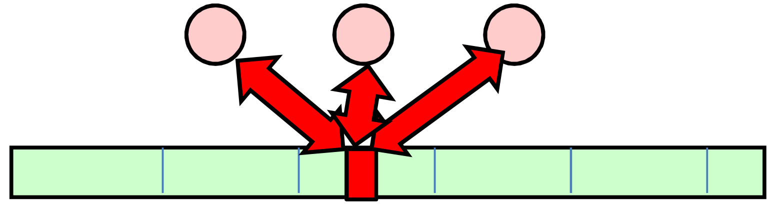
- No sharing:
  - Each core (thread) accesses distinct data on distinct cache line

  → Usually *efficient*



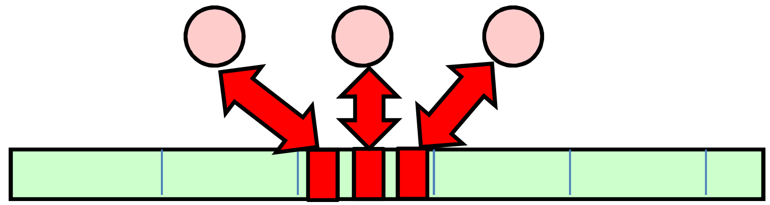
- (True) sharing:
  - Cores have access to the same data

  → *Inefficient*, due to mode change and mutual exclusion



- False sharing:
  - In programmers' intention, RW data is localized
  - But the data resides in a single cache line

  → *inefficient*, in spite of programmers' intention



Example of False Sharing

An example of
summation
per thread

With false sharing

```
int local_results[n_of_threads];
#pragma omp parallel
{
  id = omp_thread_num(); // get thread ID
  for (….) {  // many times
     local_results[id] += …; // write operations
 } }
```

- Although cores update different values, they are treated as "unique" in cache line level! → very slow

An example of
Improvement

w/o false sharing

```
#define PAD 16  // PAD*sizeof(int) should be >=64
int local_results[n_of_threads * PAD];
#pragma omp parallel
{
  id = omp_thread_num(); // get thread ID
  for (….) {  // many times
     local_results[id*PAD] += …; // write operations
 } }
```

Today’s Summary

- Being aware of parallelism is the key, if you worry about performance
  - High performance software is energy efficient!
  - Even smart phones have multi-core CPUs
- Hierarchical parallelism
  - SIMD
  - Thread programming for multi-core, multi-CPU
    - Cache algorithm is more complicated to support multi-core/CPU
  - Multi-node using network (in next lecture)
- There are many bottlenecks that degrade actual speedup
  - Memory contention, load imbalance, false sharing of cache…

Assignment for Architecture Part

- Please write a report of 2—4 pages
  - Submit via OCW, or
  - E-mail me (endo@is.titech.ac.jp)
- Choose one of problems 1, 2 or 3
- Due: Monday, February 15

- Longer than 4 pages is ok
- It is a good idea to use figures

1. Summarize one (or more) of MOSI, MESI, MOESI, MESIF protocols
   - When and how is it better than MSI protocol?

Assignment for Architecture Part (cont'd)

2. Summarize and discuss the paper

- *K. Goto, R. Geijn:*

  Anatomy of high-performance Matrix Multiplication,

  ACM TOMS 2008, 25 pages

  - Optimized matrix operations algorithm, mainly matrix multiply (GEMM), considering cache architecture
  - Partial summary is ok

Assignment for Architecture Part (cont'd)

3. Summarize and discuss the paper

3(a) R. A. van de Geijn and J. Watts:

> ***SUMMA**: scalable universal matrix multiplication algorithm,*
>
> *Concurrency - Practice and Experience, 9(4):255–274, 1997*
>
> *http://www.netlib.org/lapack/lawnspdf/lawn96.pdf*

OR

3(b) [advanced] J. Demmel, D. Eliahu, A. Fox, S. Kamil, B. Lipshitz, O. Schwartz, O. Spillinger:

> *Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication,*
>
> *IEEE IPDPS 2013, 12 pages*

- Papers about efficient matrix multiplication algorithms (Related to next lecture)
- Partial summary is OK

Next Lecture

- Feb 1 (Mon): Network
    - Network, especially in supercomputers
    - Distributed matrix multiplication algorithm
        - (Related to assignment 3)