OpenMP
advanced topics

Mitsuhisa Sato
RIKEN R-CCS
and University of Tsukuba

Agenda

- Trends of Multicore processor
- What's OpenMP
- Advanced topics
  - MPI/OpenMP Hybrid Programming
    - Programming for Multi-core cluster
  - OpenMP 3.0 (2007, approved)
    - Task parallelism
  - OpenACC (2012)
    - For GPU, by NVIDIA, PGI, Cray, ...
  - OpenMP 4.0 (2013, released)
    - Accelerator extension
    - SIMD extension
    - Task dependency description

What's OpenMP?

- Programming model and API for shared memory parallel programming
  - It is not a brand-new language.
  - Base-languages(Fortran/C/C++) are extended for parallel programming by directives.
  - Main target area is scientific application.
  - Getting popular as a programming model for shared memory processors as multi-processor and multi-core processor appears.

- OpenMP Architecture Review Board (ARB) decides spec.
  - Initial members were from ISV compiler venders in US.
  - Oct. 1997 Fortran ver.1.0 API
  - Oct. 1998 C/C++ ver.1.0 API
  - Latest version, OpenMP 3.0

- http://www.openmp.org/

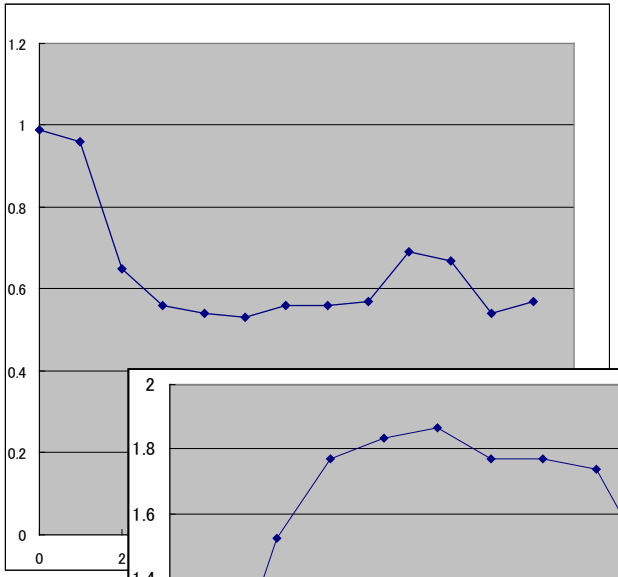
What about performance?

- OpenMP really speedup my problem?!
- It depends on hardware and problem size/characteristics
- Esp. problem sizes is an very important factor
  - Trade off between overhead of parallelization and grain size of parallel execution.
- To understand performance, …
  - How to lock
  - How to exploit cache
  - Memory bandwidth

Laplace performance

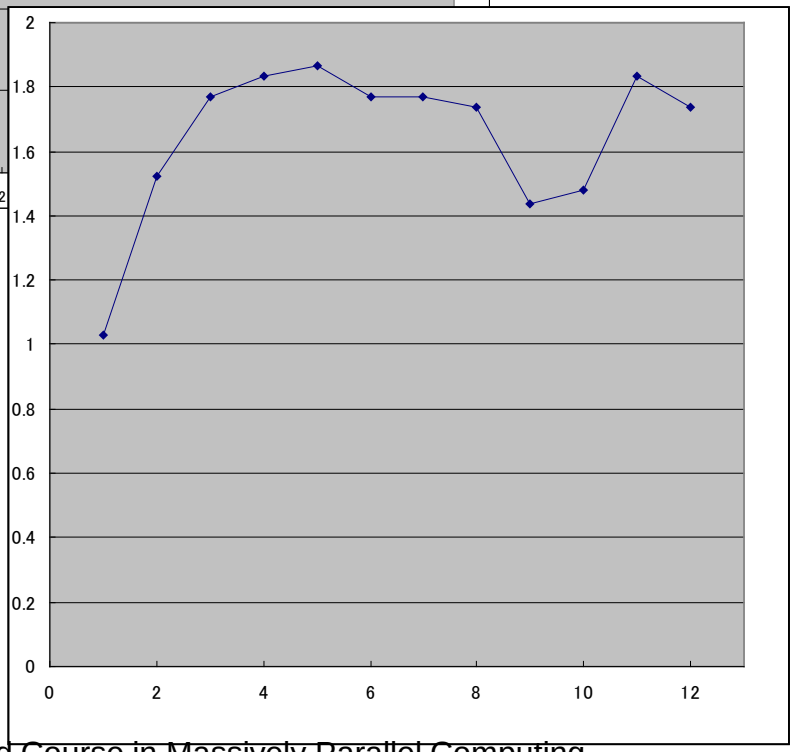
AMD Opteron quad , 2 socket

XSIZE=YSIZE=1000

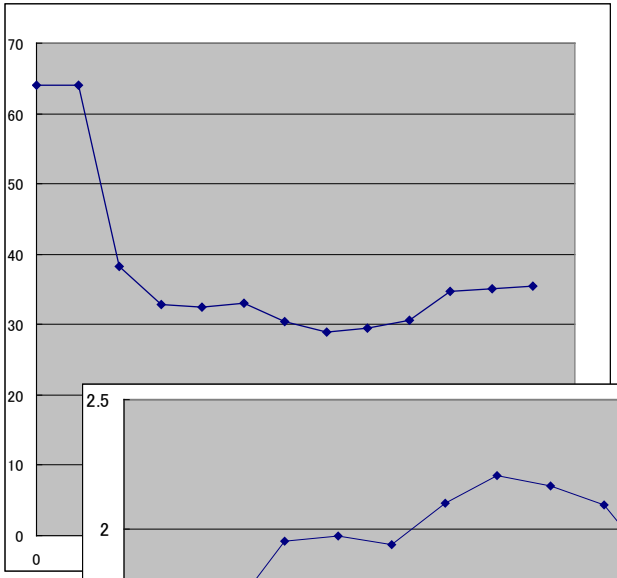


Exec time

speedup

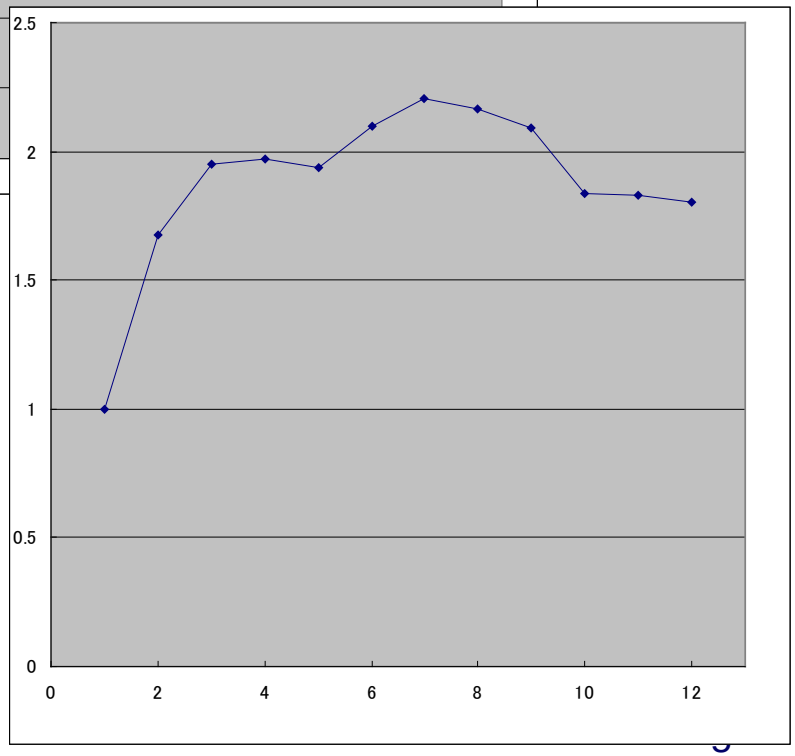


XSIZE=YSIZE=8000



Exec time

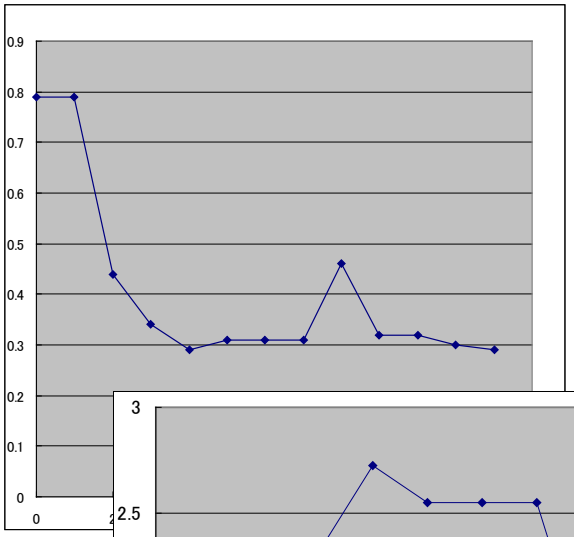
speedup



Laplace performance

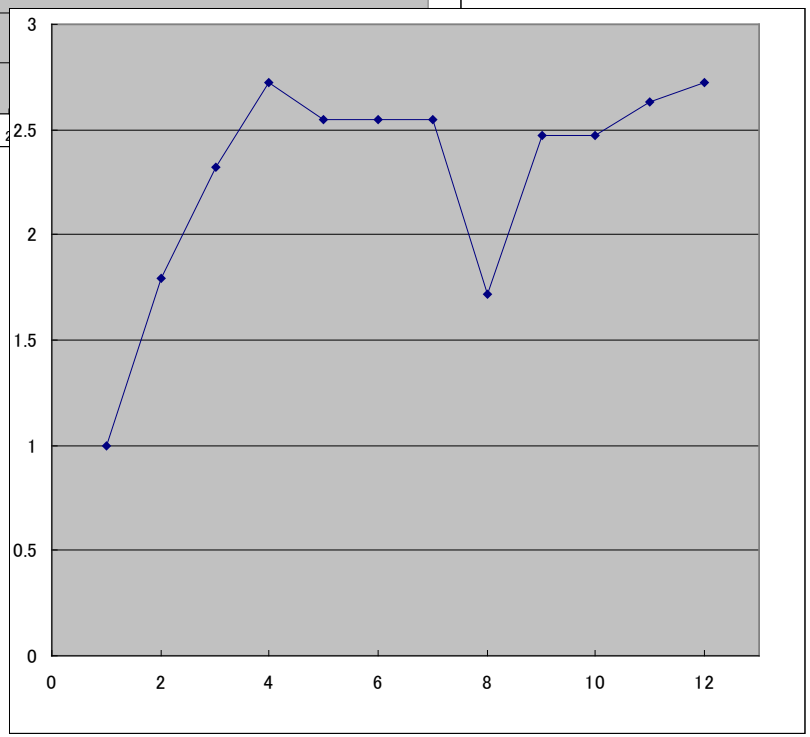
Core i7 920 @ 2.67GHz, 2 socket

XSIZE=YSIZE=1000

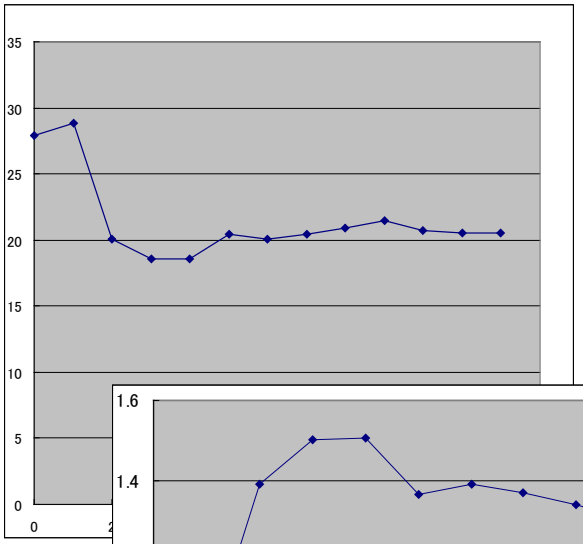


Exec time

speedup

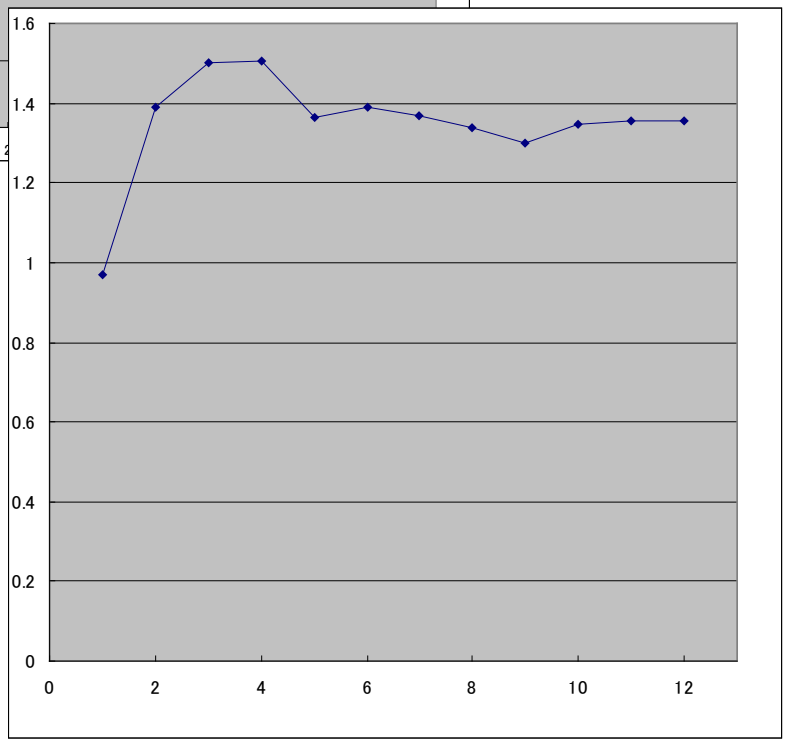


XSIZE=YSIZE=8000



Exec time

speedup

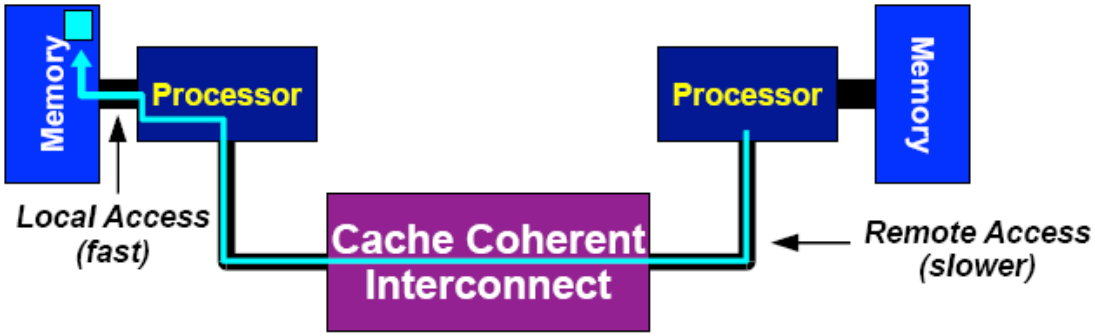


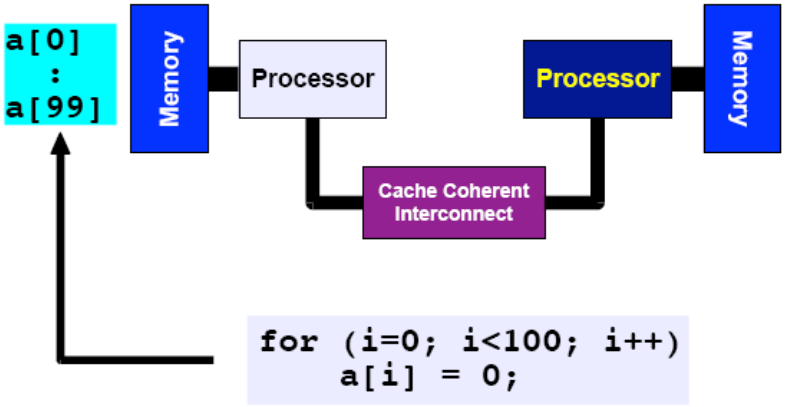
The Myth

"OpenMP Does Not Scale"

- *The transparency of OpenMP is a mixed blessing*
  - *Makes things pretty easy*
  - *May mask performance bottlenecks*
- *In the ideal world, an OpenMP application just performs well*
- *Unfortunately, this is not the case*
- *Two of the more obscure effects that can negatively impact performance are* ***cc-NUMA behavior*** *and* ***False Sharing***
- *Neither of these are restricted to OpenMP, but they are important enough to cover in some detail here*

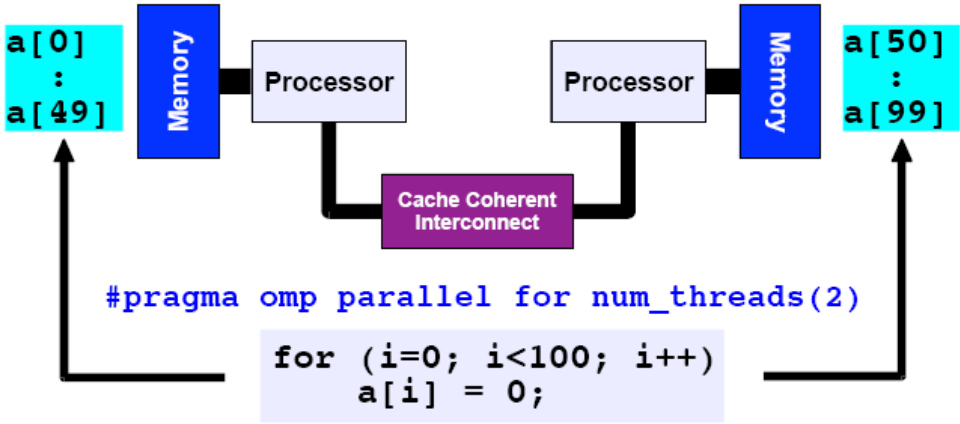
CC-NUMA and first touch






```
for (i=0; i<100; i++)
    a[i] = 0;
```

First Touch
All array elements are in the memory of the processor executing this thread

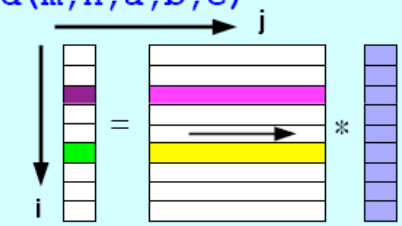


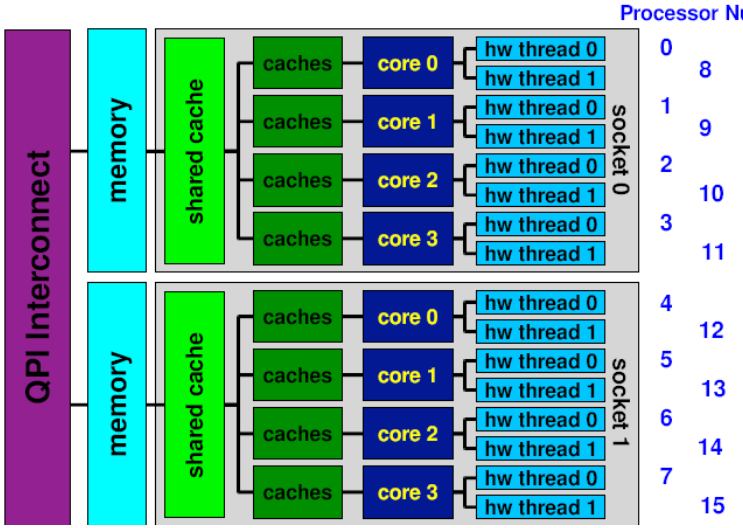

```
#pragma omp parallel for num_threads(2)
for (i=0; i<100; i++)
    a[i] = 0;
```

First Touch
Both memories each have "their half" of the array

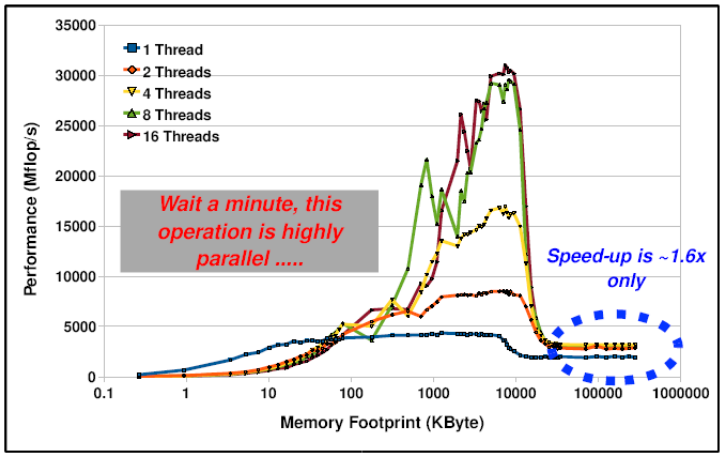
First touch

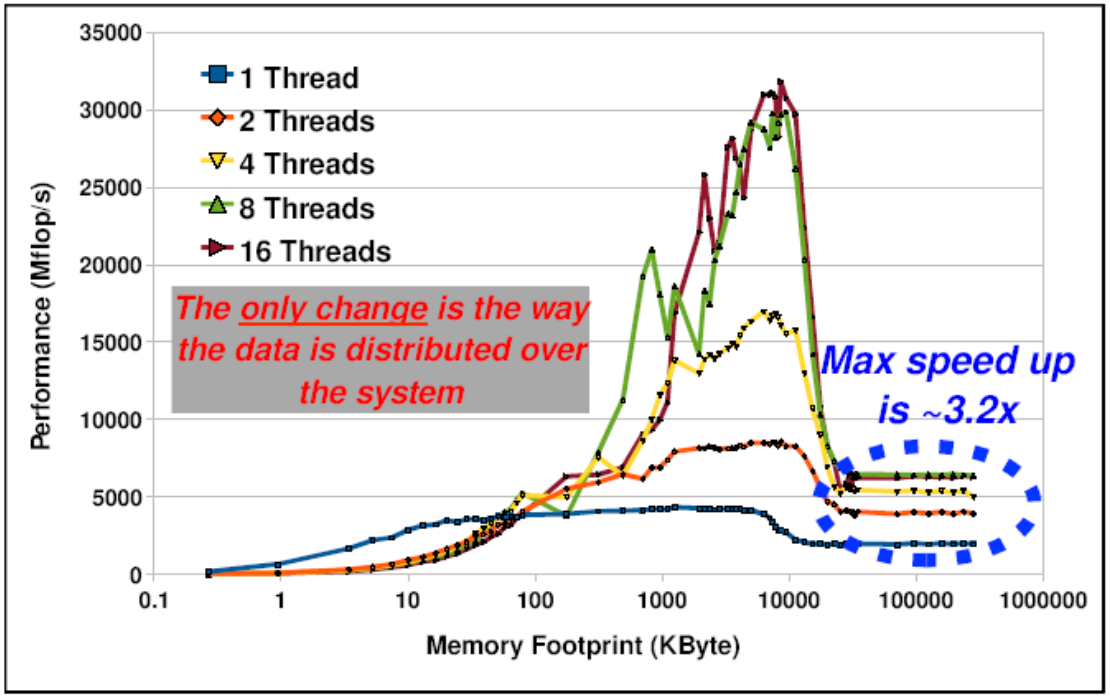
```
#pragma omp parallel for default(none) \
        private(i,j) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
  a[i] = 0.0;
  for (j=0; j<n; j++)
     a[i] += b[i][j]*c[j];
 }
```





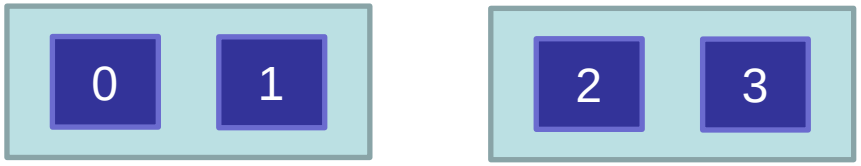
2 socket Nehalem





CPU affinity

- In multi-core multi-socket system, how the thread assigned to cores is called CPU affinity, but in OpenMP it is not standardized. (OpenMP 4.0 specifies thread affinity by proc_bind in close)
- In GCC, GOMP_CPU_AFFINITY specifies the order of assigned cores.
  - GOMP_CPU_AFFINITY="0 1 2 3" or "0 2 1 3"
- In Intel compiler,
  - KMP_AFFINITY=compact or scatter



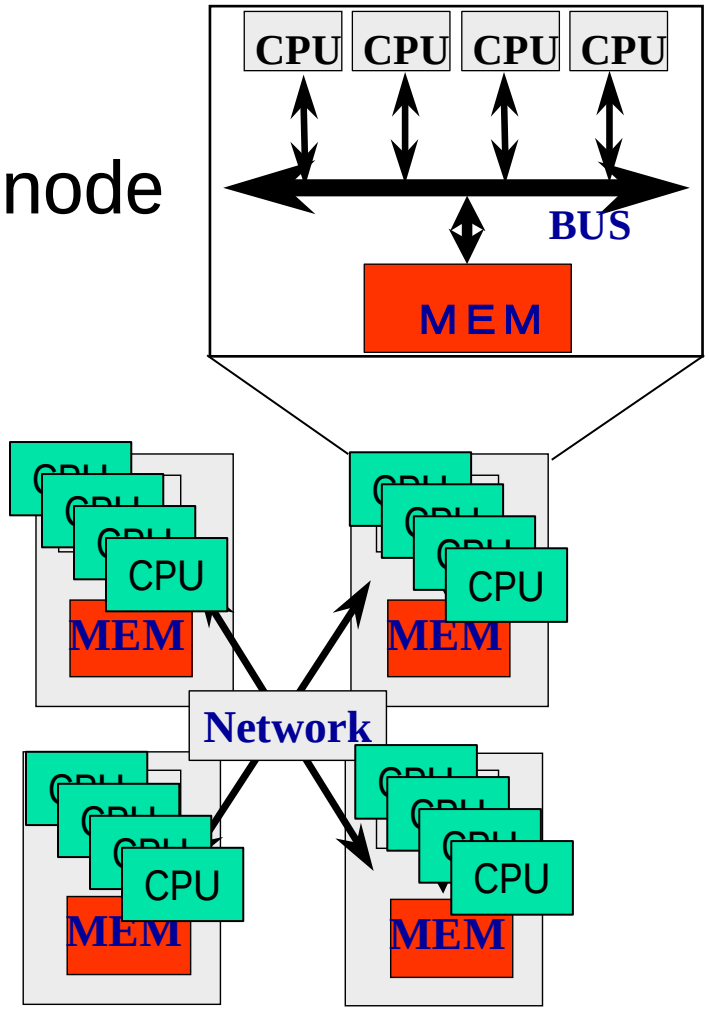
Advanced topics

- MPI/OpenMP Hybrid Programming
  - Programming for SMP (multicore) cluster
- OpenMP 3.0
  - Approved in 2007
  - Task
- OpenMP 4.0
  - Approved in 2013
  - Accelerator device extension

MPI-OpenMP hybrid programming
How to use multi-core cluster

- Flat MPI: Run MPI process in core (CPU)
  - Many MPI processes
  - Only MPI programming is needed

- MPI-OpenMP hybrid
  - Use MPI between nodes
  - Use OpenMP in node
  - Save number of MPI process, resulting in saving memory. Important in large-scale system
  - Cost: Need two (MPI-OpenMP) programming
  - Sometimes OpenMP performance is worse than MPI



Thread-safety of MPI

- Use MPI_ MPI_Init_thread to get info about thread-safety
- MPI_THREAD_SINGLE
  - A process has only one thread of execution.
- MPI_THREAD_FUNNELED
  - A process may be multithreaded, but only the thread that initialized MPI can make MPI calls.
- MPI_THREAD_SERIALIZED
  - A process may be multithreaded, but only one thread at a time can make MPI calls.
- MPI_THREAD_MULTIPLE
  - A process may be multithreaded and multiple threads can call MPI functions simultaneously.

Update in OpenMP3.0

- The concept of “task” is introduced:
  - An entity of thread created by Parallel construct and Task construct.
  - Task Construct & Taskwait construct
- Interpretation of shared memory consistency in OpenMP
  - Definition of Flush semantics
- Nested loop
  - Collapse clauses
- Specify stack size of thread.
- constructor, destructor of private variables in C++

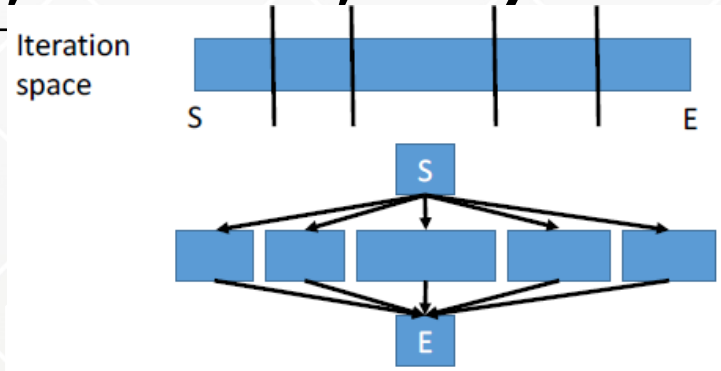
“Classic” OpenMP

- Mainly using parallel loop “parallel do/for” for data parallelism
- Fork-Join model

こんな感じで、OK!

Just Like this!

```
#pragma omp parallel for
reduction(+:s)
   for(i=0; i<1000;i++) s+= a[i];
```



Task in OpenMP

- Available starting with OpenMP 3.0 (2008)
- A task is an entity composed of
  - Code to be executed
  - Data environment (inputs to be used and outputs to be generated)
  - A location (stack) where the task will be executed (a thread)

- Allowing the application to explicitly create tasks provide support for different execution models
  - More elastic as now the threads have a choice between multiple existing tasks
  - Require scheduling strategies to be more powerful
  - Move away from the original fork/join model of OpenMP constructs

Task in OpenMP – history –

- **OpenMP 3.0 → May 2008**
  - Task support (useful to parallelize recursive function calls)
- **OpenMP 3.1 → July 2011**
  - Taskyield construct
  - Extension of atomic operations
- **OpenMP 4.0 → July 2013**
  - SIMD constructs
  - PROC_BIND and places
  - Device constructs (for GPU/accelerator)
  - Task dependencies
- **OpenMP 4.5 → November 2015**
  - Taskloop constructs
  - Task priority

Directives for task

- **Task directive:**
  - Each encountering thread/task creates a new task
  - Tasks can be nested

C/C++
```
#pragma omp task [clause]
... Structured block ...
```

Fortran
```
!$omp task [clause]
... Structured block ...
!$omp end task
```

- **Taskwait directive: Task barrier**
  - Encountering task is suspended until child tasks are complete
  - Applies only to direct children, not descendants!

C/C++
```
#pragma omp taskwait
```

- **OpenMP barrier**
  - All tasks created by any thread of the current *Team* are guaranteed to be completed at barrier exit

An example of Task - recursive Fibonacci program -

- Task enables to parallelize recursive function calls
- Starting with single thread surrounded by "parallel" directive
- Task directive creates a task to execute a function call
- Taskwait directive to wait children

```
int fib(int n)   {
   if (n < 2) return n;
   int x = fib(n - 1);
   int y = fib(n - 2);
   return x+y;
}
```

```
int main(int argc,
         char* argv[])
{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

```
int fib(int n)   {
   if (n < 2) return n;
int x, y;
#pragma omp task shared(x)
{
   x = fib(n - 1);
}
#pragma omp task shared(y)
{
   y = fib(n - 2);
}
#pragma omp taskwait
   return x+y;
}
```

Optimized versions

- **Control granularity**
  - Skip the OpenMP overhead once a certain n is reached

```
int fib(int n)   {
   if (n < 2) return n;
int x, y;
#pragma omp task shared(x) \
   if(n > 30)
{
   x = fib(n - 1);
}
#pragma omp task shared(y) \
   if(n > 30)
{
   y = fib(n - 2);
}
#pragma omp taskwait
   return x+y;
}
```

```
int fib(int n)   {
   if (n < 2) return n;
   if (n <= 30)
      return serfib(n);
int x, y;
#pragma omp task shared(x)
{
   x = fib(n - 1);
}
#pragma omp task shared(y)
{
   y = fib(n - 2);
}
#pragma omp taskwait
   return x+y;
}
```

Taken from IWOMP Tutorial: 30th September 2015,
"Advanced OpenMP Tutorial – Tasking", by Christian Terboven, Michael Klemm

Example of Task Constructs

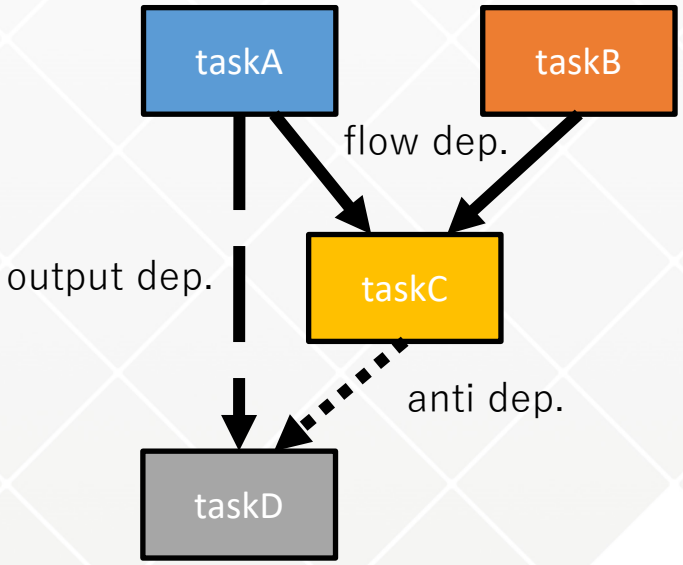
```
struct node {
        struct node *left;
        struct node *right;
};

void postorder_traverse( struct node *p ) {
        if (p->left)
                #pragma omp task // p is firstprivate by default
                postorder_traverse(p->left);
        if (p->right)
                #pragma omp task // p is firstprivate by default
                postorder_traverse(p->right);
        #pragma omp taskwait
        process(p);
}
```

Task dependency in OpenMP 4.0 (2013)

- **Task directive in OpenMP4.0 creates a task with dependency specified "depend" clause with dependence-type in, out, or inout and variable**
  - Flow dependency: in after out (taskA to taskC, taskB to taskC)
  - Anti dependency: out after in (taskC to taskD)
  - Output dependency: out after out (taskA to taskD)
  - If there are no dependency, tasks are executed immediately in parallel
- **NOTE: The task dependency depends on the order of reading and writing to data based on the sequential execution.**

```
#pragma omp parallel
#pragma omp single
{
#pragma omp task depend(out:A)
   A = 1;                    /* taskA */
#pragma omp task depend(out:B)
   B = 2;                    /* taskB */
#pragma omp task depend(in:A, B) depend(out:C)
   C = A + B;                /* taskC */
#pragma omp task depend(out:A)
   A = 2;                    /* taskD */
}
```



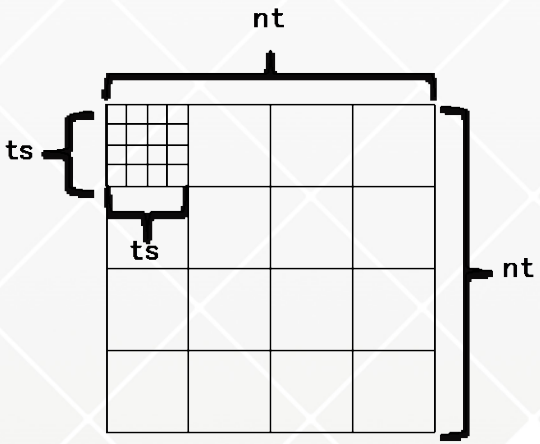
Example: Block Cholesky Factorization

- **Block Cholesky Factorization consists of**
  - potrf: This kernel performs the Cholesky factorization of a diagonal (lower triangular) tile
  - trsm: a triangular solve applies an update to an off-diagonal tile
  - syrk: symmetric rank-k update applies updates to a diagonal tile
  - gemm: matrix multiplication applies updates to an off-diagonal tile

```
void cholesky(const int ts, const int nt, double* A[nt][nt])
{
   for (int k = 0; k < nt; k++) {
      x_potrf(A[k][k], ts, ts);
      for (int i = k + 1; i < nt; i++)
         x_trsm(A[k][k], A[k][i], ts, ts);
      for (int i = k + 1; i < nt; i++) {
         for (int j = k + 1; j < i; j++)
            x_gemm(A[k][i], A[k][j], A[j][i], ts, ts);
         x_syrk(A[k][i], A[i][i], ts, ts);
      }
   }
}
```

A[nt][nt][ts*ts]

In right example
A[4][4][16]



Question: How do you parallelize this in OpenMP?

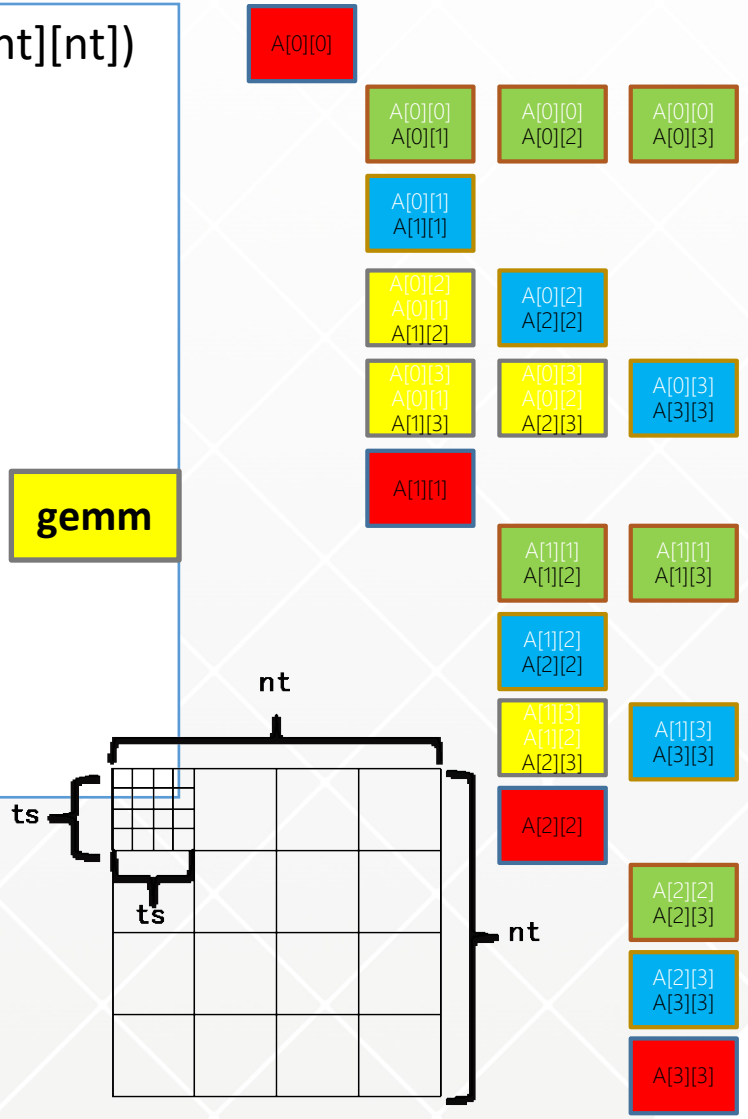
```
void cholesky(const int ts, const int nt, double* A[nt][nt])
{
   for (int k = 0; k < nt; k++) {
      x_potrf(A[k][k], ts, ts);
      for (int i = k + 1; i < nt; i++)
         x_trsm(A[k][k], A[k][i], ts, ts);
      for (int i = k + 1; i < nt; i++) {
         for (int j = k + 1; j < i; j++)
            x_gemm(A[k][i], A[k][j], A[j][i], ts, ts);
         x_syrk(A[k][i], A[i][i], ts, ts);
      }
   }
}
```

potrf

trsm

gemm

syrk



Question: How do you parallelize this in OpenMP?

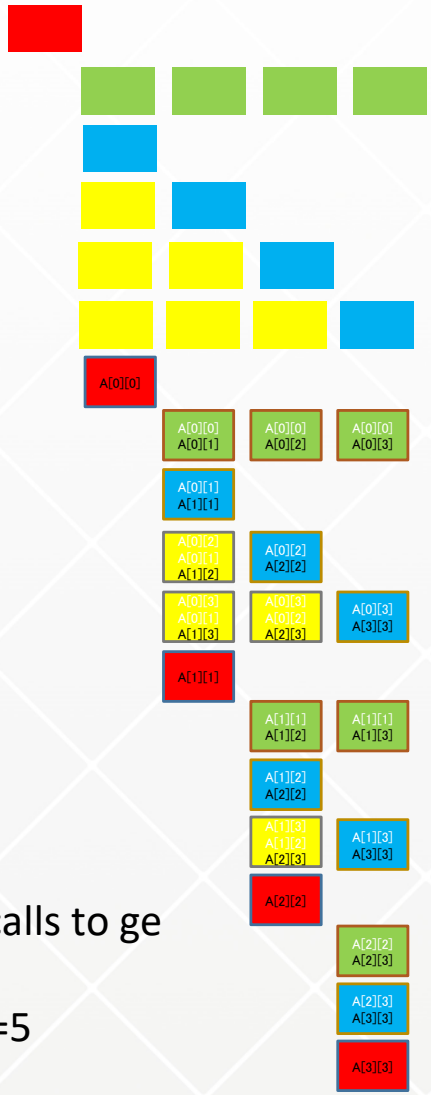
```
void cholesky(const int ts, const int nt, double* A[nt][nt])
{
   for (int k = 0; k < nt; k++) {
      x_potrf(A[k][k], ts, ts);
      for (int i = k + 1; i < nt; i++)
         x_trsm(A[k][k], A[k][i], ts, ts);
      for (int i = k + 1; i < nt; i++) {
         for (int j = k + 1; j < i; j++)
            x_gemm(A[k][i], A[k][j], A[j][i], ts, ts);
         x_syrk(A[k][i], A[i][i], ts, ts);
      }
   }
}
```

potrf

trsm

gemm

syrk



Note:
As ts increases, the calls to ge
mm is increasing!
(right, the case of ts=5

An answer: using OpenMP “classic” parallel loop

```
void cholesky(const int ts, const int nt, double* A[nt][nt])
{
  int i;
    for (int k = 0; k < nt; k++) {
        x_potrf(A[k][k], ts, ts);
#pragma omp parallel for
        for (int i = k + 1; i < nt; i++) {
            x_trsm(A[k][k], A[k][i], ts, ts);
        }
        for (int i = k + 1; i < nt; i++) {
#pragma omp parallel for
          for (int j = k + 1; j < i; j++) {
                x_gemm(A[k][i], A[k][j], A[j][i], ts, ts);
            }
        }
#pragma omp parallel for
        for (int i = k + 1; i < nt; i++)
          x_syrk(A[k][i], A[i][i], ts, ts);
    }
}
```

These loops can be collapsed

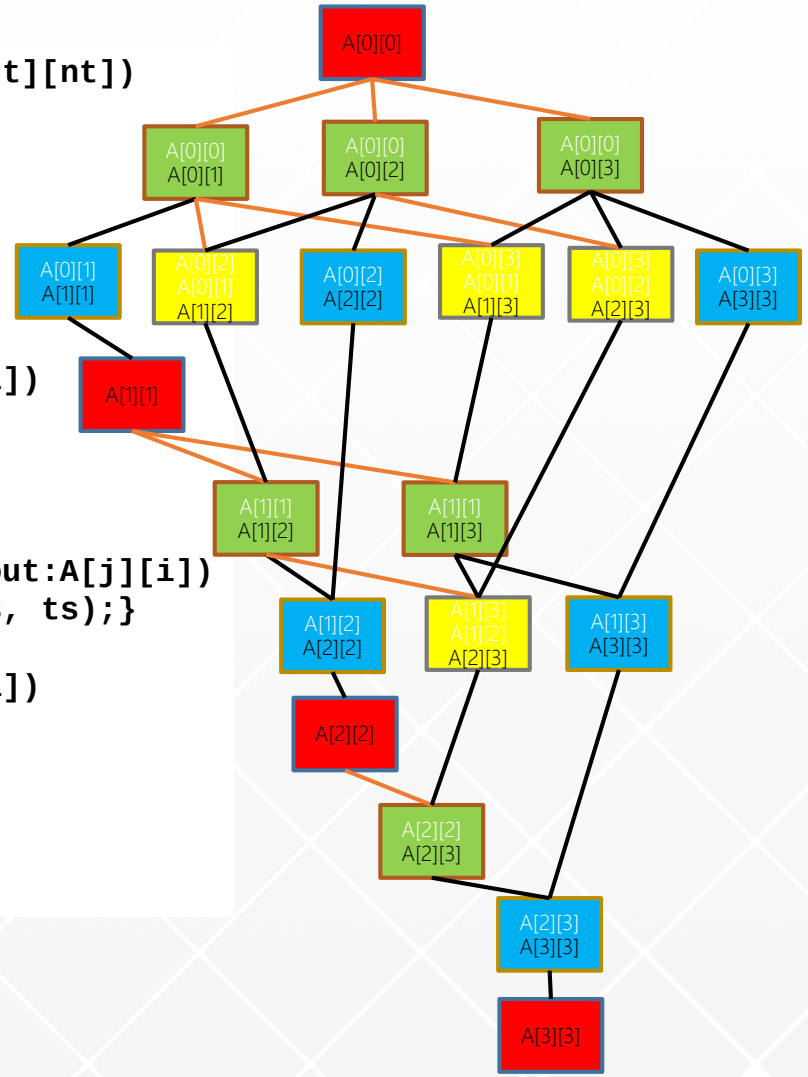
Syrk can be called separately

An example using tasks

```
void cholesky(const int ts, const int nt, double* A[nt][nt])
{
#pragma omp parallel
#pragma omp single
    for (int k = 0; k < nt; k++) {
#pragma omp task depend(out:A[k][k])
{        x_potrf(A[k][k], ts, ts); }
        for (int i = k + 1; i < nt; i++) {
#pragma omp task depend(in:A[k][k]) depend(out:A[k][i])
{            x_trsm(A[k][k], A[k][i], ts, ts); }
        }
        for (int i = k + 1; i < nt; i++) {
            for (int j = k + 1; j < i; j++) {
#pragma omp task depend(in:A[k][i], A[k][j]) depend(out:A[j][i])
{                x_gemm(A[k][i], A[k][j], A[j][i], ts, ts);}
            }
#pragma omp task depend(in:A[k][i]) depend(out:A[i][i])
{            x_syrk(A[k][i], A[i][i], ts, ts);}
        }
    }
#pragma omp taskwait
}
```

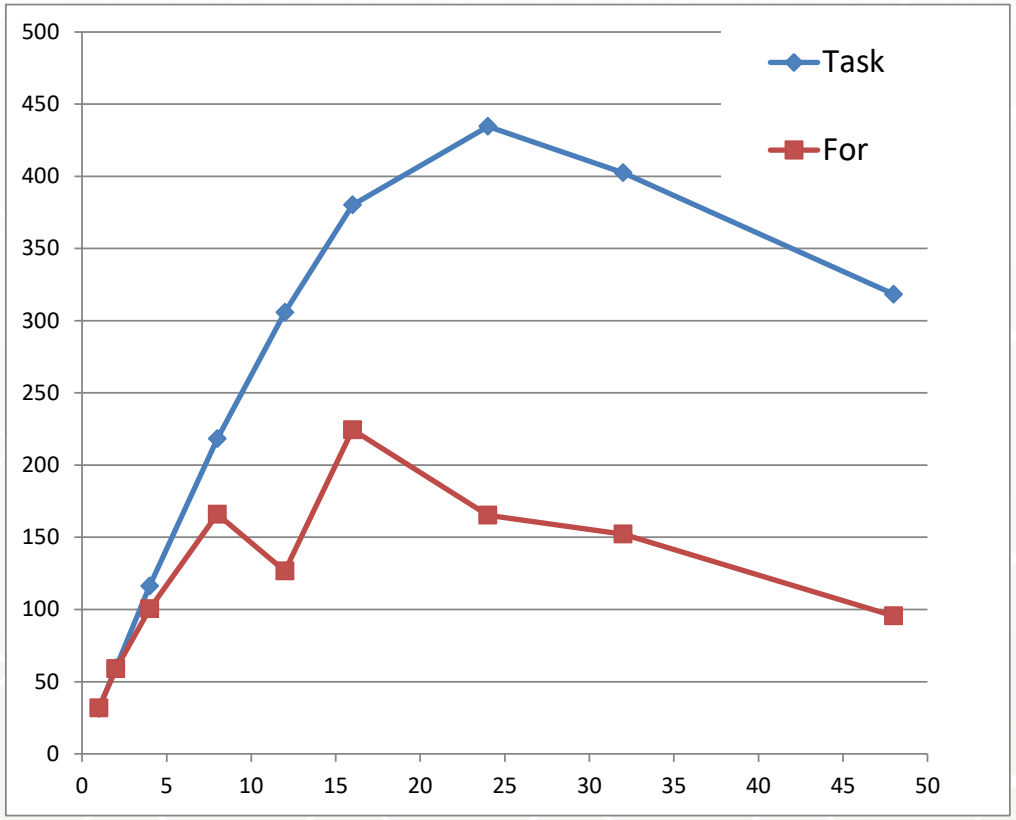
What's task graph created?

```
void cholesky(const int ts, const int nt, double* A[nt][nt])
{
#pragma omp parallel
#pragma omp single
    for (int k = 0; k < nt; k++) {
#pragma omp task depend(out:A[k][k])
{        x_potrf(A[k][k], ts, ts); }
        for (int i = k + 1; i < nt; i++) {
#pragma omp task depend(in:A[k][k]) depend(out:A[k][i])
{            x_trsm(A[k][k], A[k][i], ts, ts); }
        }
        for (int i = k + 1; i < nt; i++) {
            for (int j = k + 1; j < i; j++) {
#pragma omp task depend(in:A[k][i], A[k][j]) depend(out:A[j][i])
{                x_gemm(A[k][i], A[k][j], A[j][i], ts, ts);}
            }
#pragma omp task depend(in:A[k][i]) depend(out:A[i][i])
{            x_syrk(A[k][i], A[i][i], ts, ts);}
        }
    }
#pragma omp taskwait
}
```



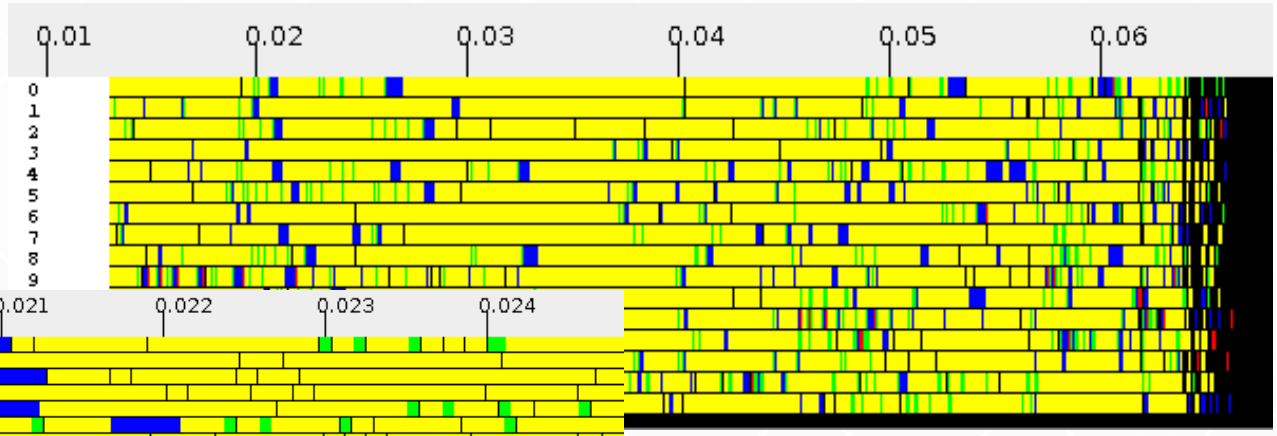
How about performance?

- **Platform: Intel Haswell** E5-2680V3 2.50 GHz, 12core 24 threads x 2 sockets
- Problem size : 4096 x 128

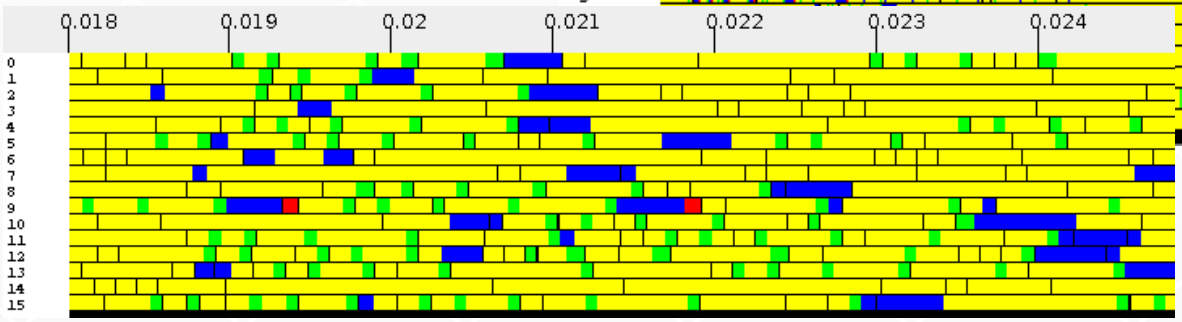


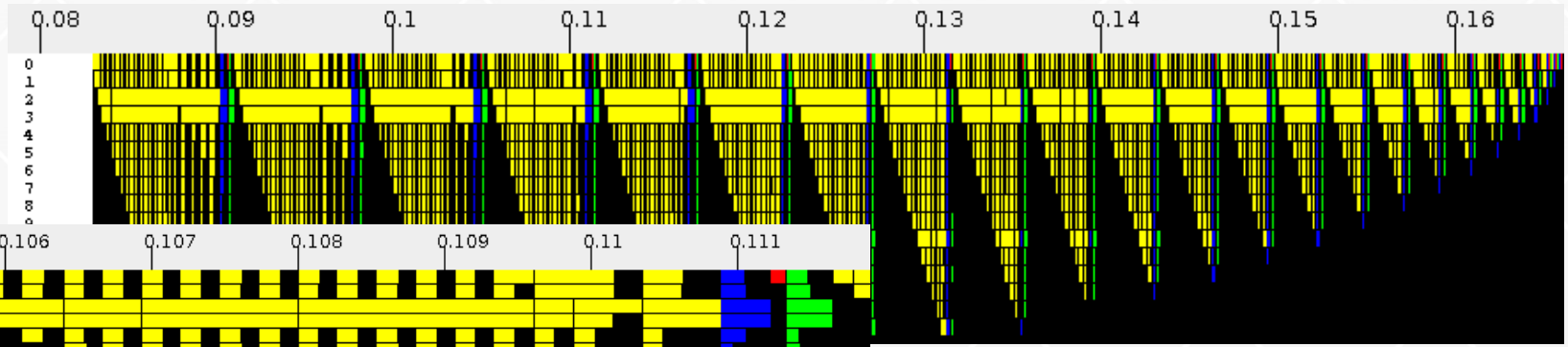
Taking a look closer at execution profile!

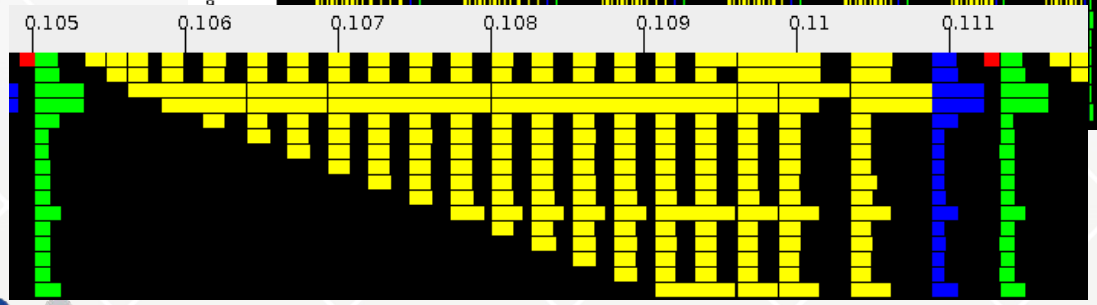
note: #threads = 16



By using Task







By Parallel Loop

This is not complete tutorial, …

- **Data scope rules …**
- **New clauses**
  - final clause
  - Mergable clause
- **Taskyield**
- **Taskloop**
  - Parallelize a loop using OpenMP tasks
- **Taskgroup**
  - Specifies a wait on completion of child tasks and their descendent tasks

“MPI+X” for exascale?

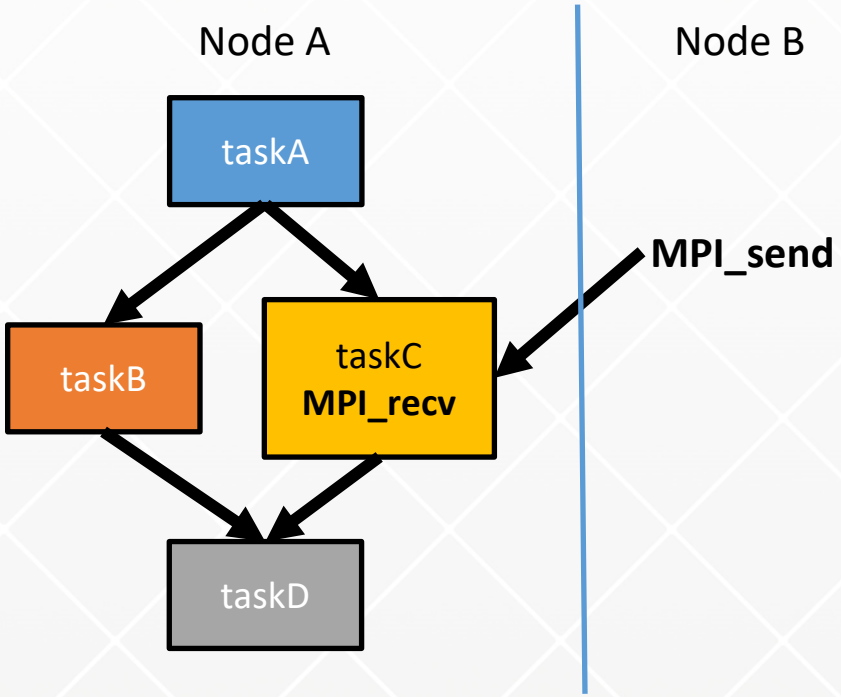
- **X is OpenMP!**
- **“MPI+Open” is now a standard programming for high-end systems.**

- **Questions:**
  - X is OpenMP task !!!
  - Then, what is a role of OpenMP?

“MPI task” in OmpSs approach

- **OmpSs (BSC) proposed an approach to make a block containing MPI calls (“MPI task”) a task in OpenMP task programming.**
  - Advantage: It enables overlapping with computation and communications, and hiding communication latency.

```
#pragma omp parallel
 #pragma omp single
                {
    #pragma omp task depend(out:A)
  A = ...;                          /* taskA */
   #pragma omp task depend(in A, out B)
   B = foo(A)                       /* taskB */
#pragma omp task depend(in:A) depend(out:C)
  {  MPI_recv(....);   /* communication */
                C = goo(A, ...);
 }                                  /* taskC */
     #pragma omp task depend(in B, C)
  D = B + C                         /* taskD */
                }
```



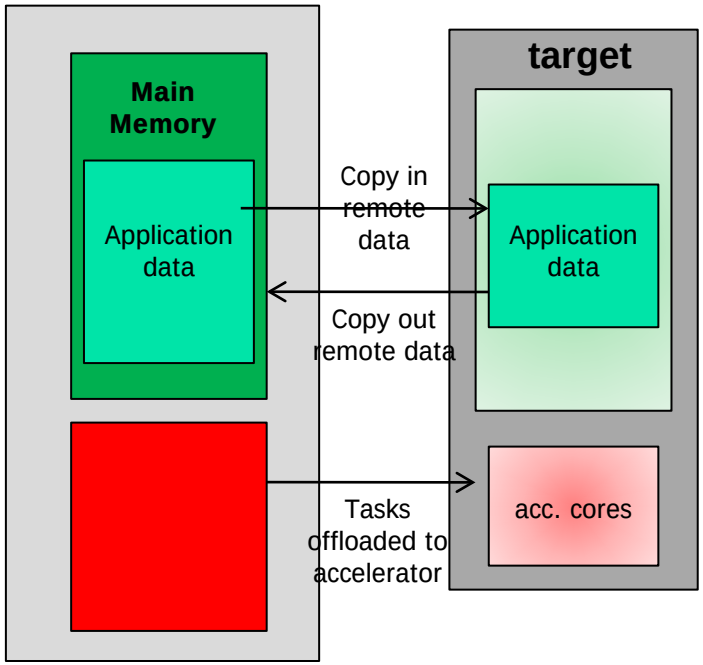
Task B and communication in task C can be overlap
Note: In this case, MPI_Irecv can do the same thing.

OpenMP 4.0

- Released July 2013
  - http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf
  - A document of examples is expected to release soon
- Changes from 3.1 to 4.0 (Appendix E.1):
  - *Accelerator: 2.9*
  - *SIMD extensions: 2.8*
  - *Places and thread affinity: 2.5.2, 4.5*
  - *Taskgroup and dependent tasks: 2.12.5, 2.11*
  - *Error handling: 2.13*
  - *User-defined reductions: 2.15*
  - *Sequentially consistent atomics: 2.12.6*
  - *Fortran 2003 support*

Accelerator (2.9): offloading

- Execution Model: Offload data and code to accelerator
- *target* construct creates tasks to be executed by devices
- Aims to work with wide variety of accs
  - GPGPUs, MIC, DSP, FPGA, etc
  - A target could be even a remote node, intentionally




```
#pragma omp target
{
/* it is like a new task
* executed on a remote device */
{
```

target and map examples

```
void vec_mult(int N)
{
   int i;
   float p[N], v1[N], v2[N];
   init(v1, v2, N);
   #pragma omp target map(to: v1, v2) map(from: p)
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

```
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   #pragma omp target map(to: v1[0:N], v2[:N]) map(from: p[0:N])
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

“MPI+X” for exascale?

- **X is OpenMP!**
- **“MPI+Open” is now a standard programming for high-end systems.**
  - I’d like to celebrate that OpenMP became “standard” in HPC programming

- **Questions:**
  - Then, What’s the style OpenMP?

Challenges of Programming Languages/models for exascale computing

- **Multithreading/multitasking models for manycore node**
  - “Manycore” is inevitable for higher node performance
  - Multitasking is useful to overlap comm with computation.
    - May fill a gap between node performance and comm. performance.
  - PGAS for a programming model of simple efficient one-sided communication for “manycore”.
- **Strong Scaling in node**
  - SIMD & Accelerator (GPU)
  - Complex memory hierarchy
- **Workflow and Fault-Resilience**
- **(Power-aware)**