Task parallelism and PGAS

- Trends and Challenges on Programming Models for Extreme-Scale Computing -

Mitsuhisa Sato

Deputy Project Leader and Team Leader, Architecture Development Team, Flagship 2020 Project

Team Leader, Programming Environment Research Team

RIKEN Center for Computational Science

Outline of my talk

- **Short introduction**
  - Challenges of Programming Languages/models for exascale computing

- **Agenda**
  - **Question : “MPI+X” for exascale?**
  - Task parallel programming in OpenMP
    - X is Task? MPI+Task?
  - What is PGAS (Partitioned Global Address Space) model, and advantage?
  - What PGAS for “MPI+X”?

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 efficient one-sided communication for "manycore".
- **Strong Scaling in node**
  - SIMD & Accelerator (GPU)
  - Complex memory hierarchy
- **Workflow and Fault-Resilience**
- **(Power-aware)**

“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?

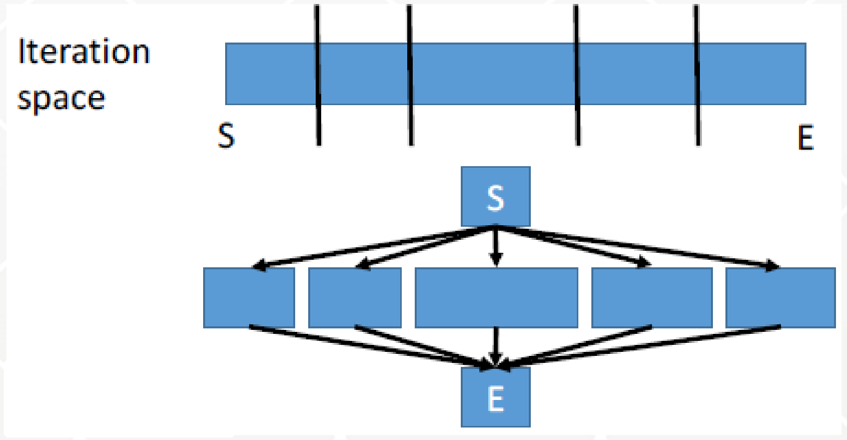
“Classic” OpenMP

- 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

RIKEN R-CCS

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

```c
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;
}
```

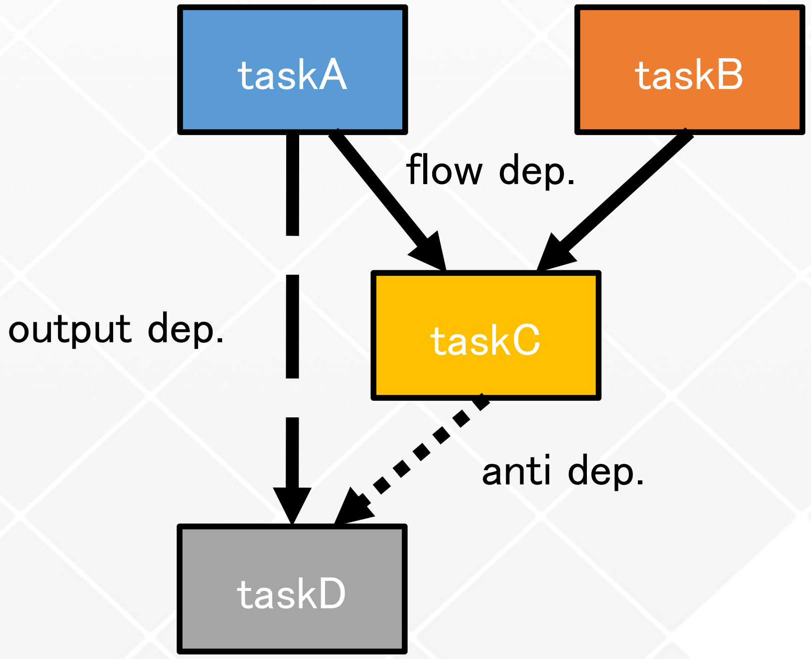
```c
int fib(int n)   {
   if (n < 2) return n;
   if (n <= 30)
      return serfib(n);
int x, y;
#pragma omp task shared(x)
{
   x = fib(n - 1);
}
#pragma omp task shared(y)
{
   y = fib(n - 2);
}
#pragma omp taskwait
   return x+y;
}
```

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

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 */
}
```



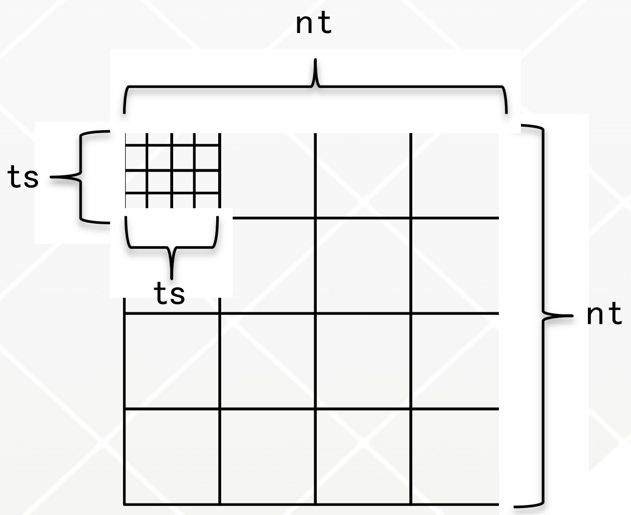
Example: Block Cholesky Factorization

- **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?

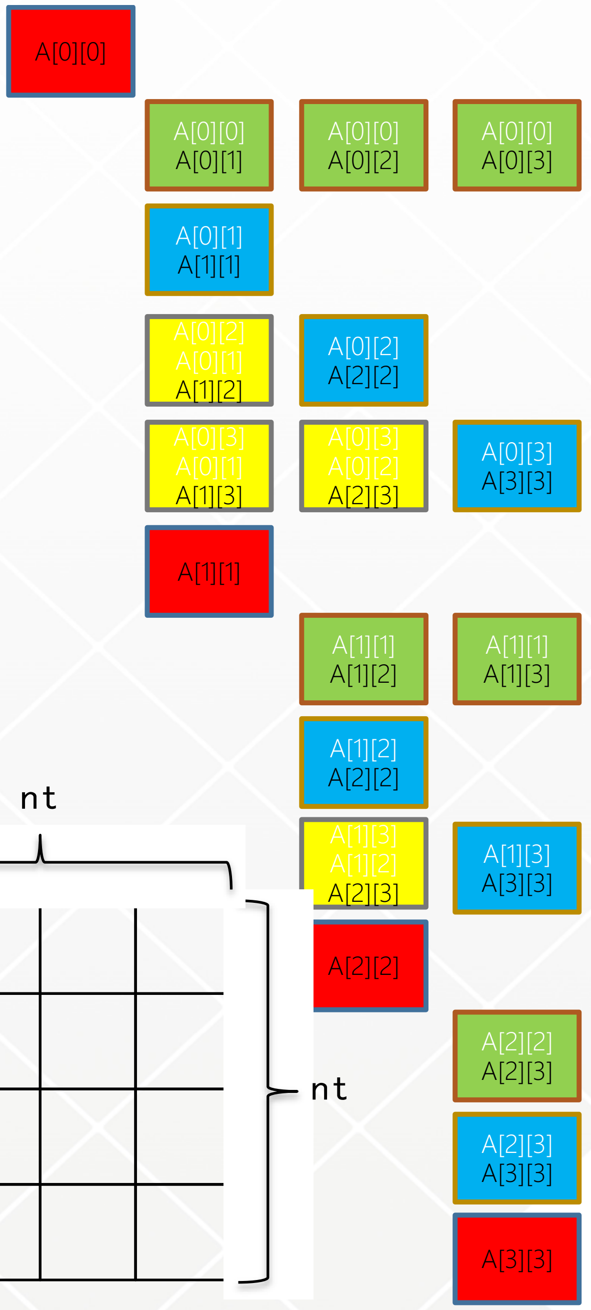
```
void cholesky(const int ts, const int nt, double* A[nt][nt])
{
   for (int k = 0; k < nt; k++) {
      x_potrf(A[k][k], ts, ts);
      for (int i = k + 1; i < nt; i++)
         x_trsm(A[k][k], A[k][i], ts, ts);
      for (int i = k + 1; i < nt; i++) {
         for (int j = k + 1; j < i; j++)
   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

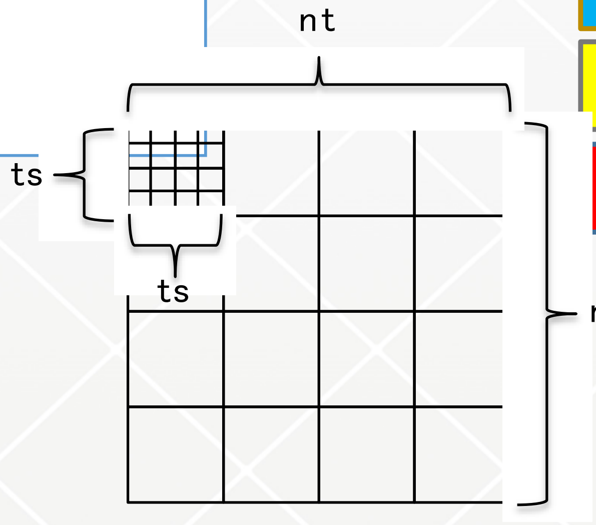
potrf

trsm

gemm

syrk





Question: How do you parallelize this in OpenMP?

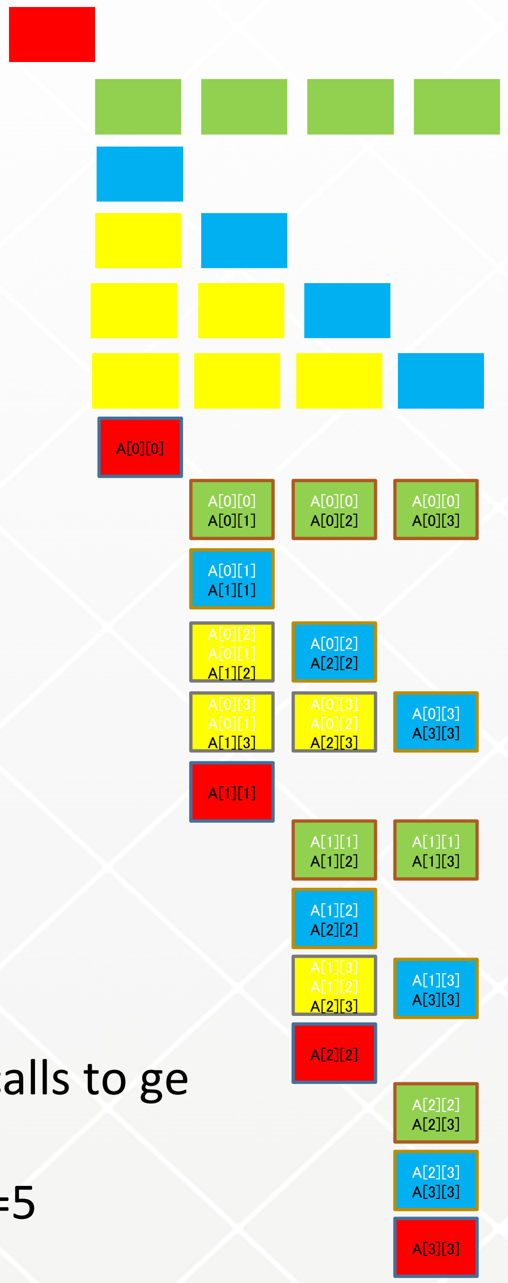
```
void cholesky(const int ts, const int nt, double* A[nt][nt])
{
   for (int k = 0; k < nt; k++) {
      x_potrf(A[k][k], ts, ts);
      for (int i = k + 1; i < nt; i++)
         x_trsm(A[k][k], A[k][i], ts, ts);
      for (int i = k + 1; i < nt; i++) {
         for (int j = k + 1; j < i; j++)
   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

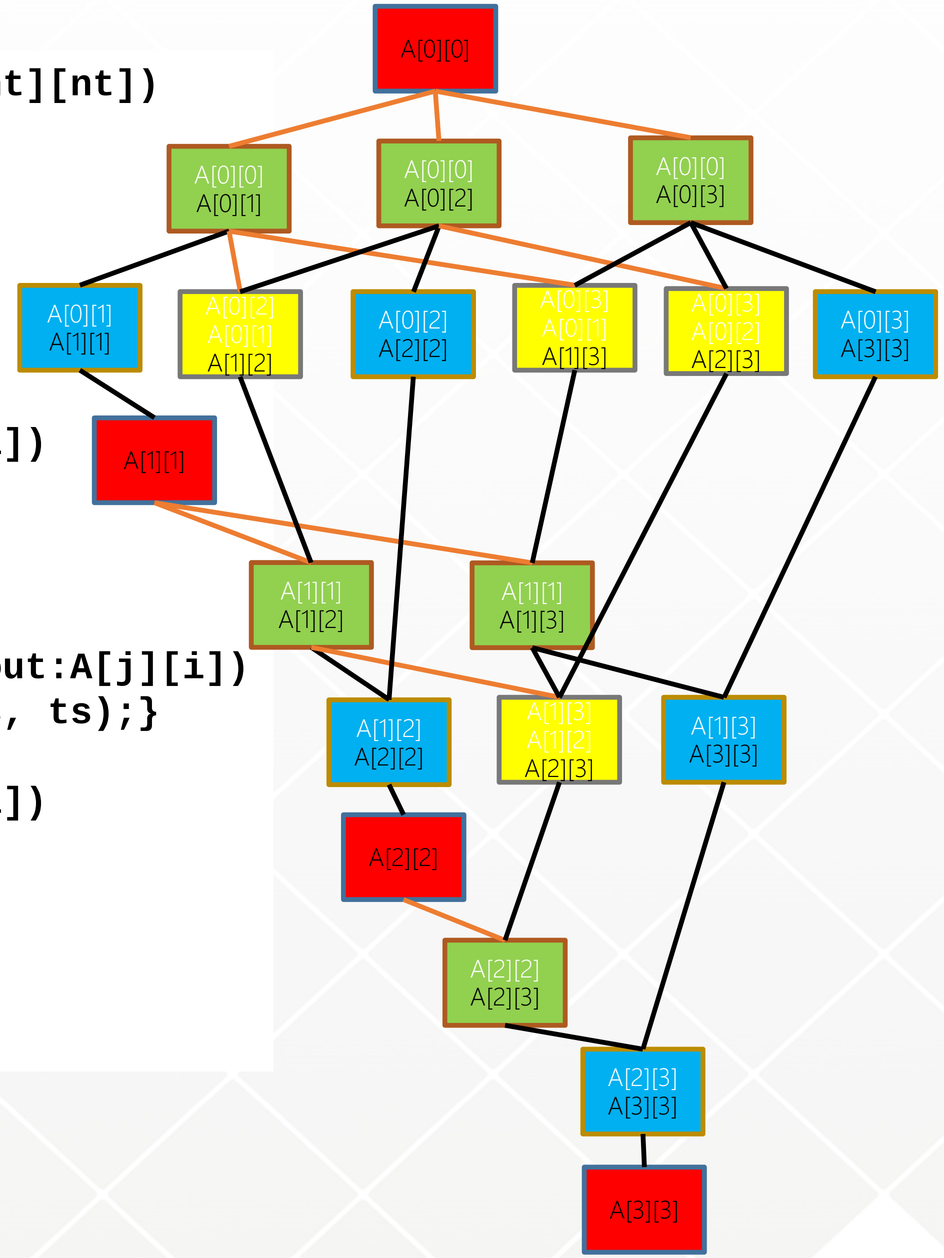
Syrk can be called separately

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
}
```

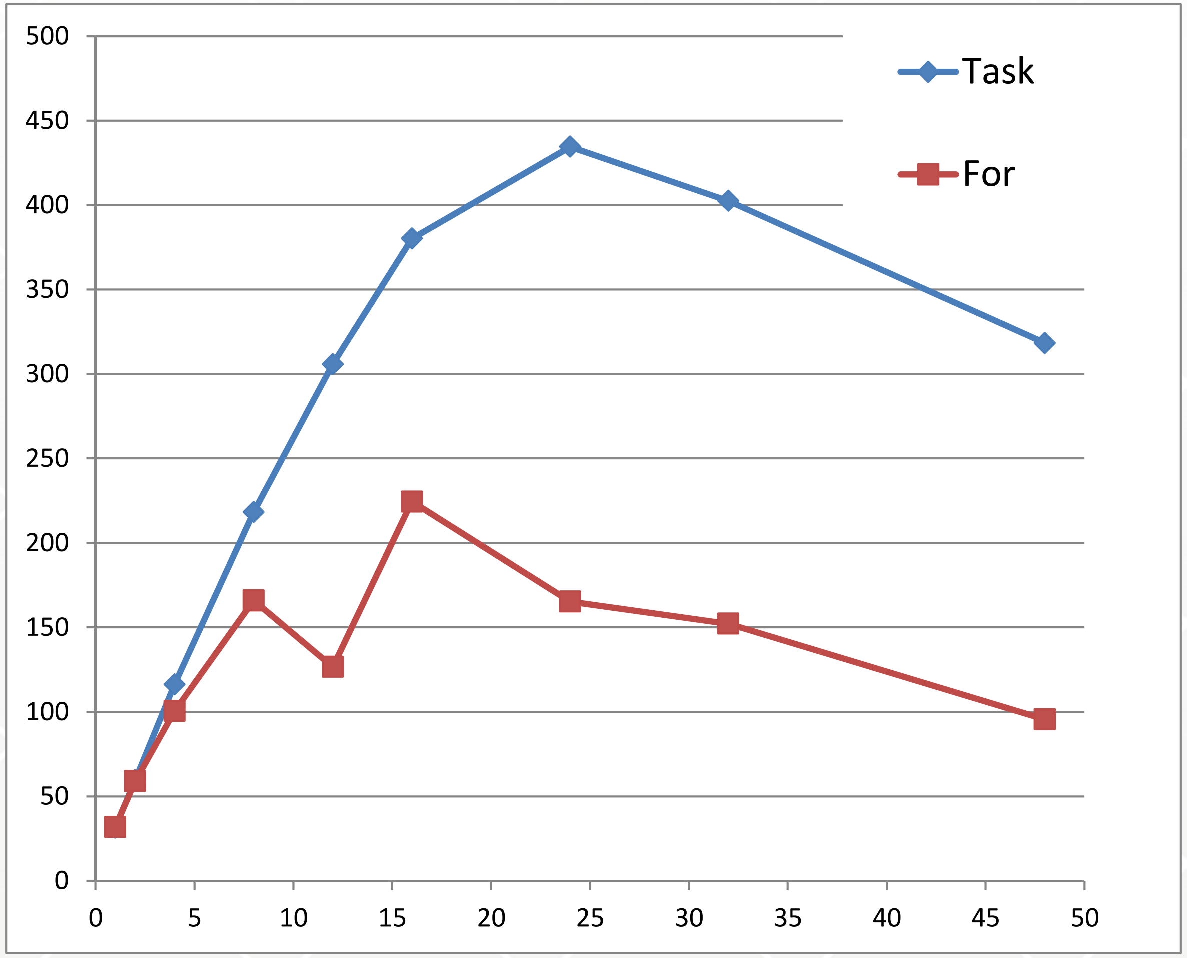
What’s task graph created?

```
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?

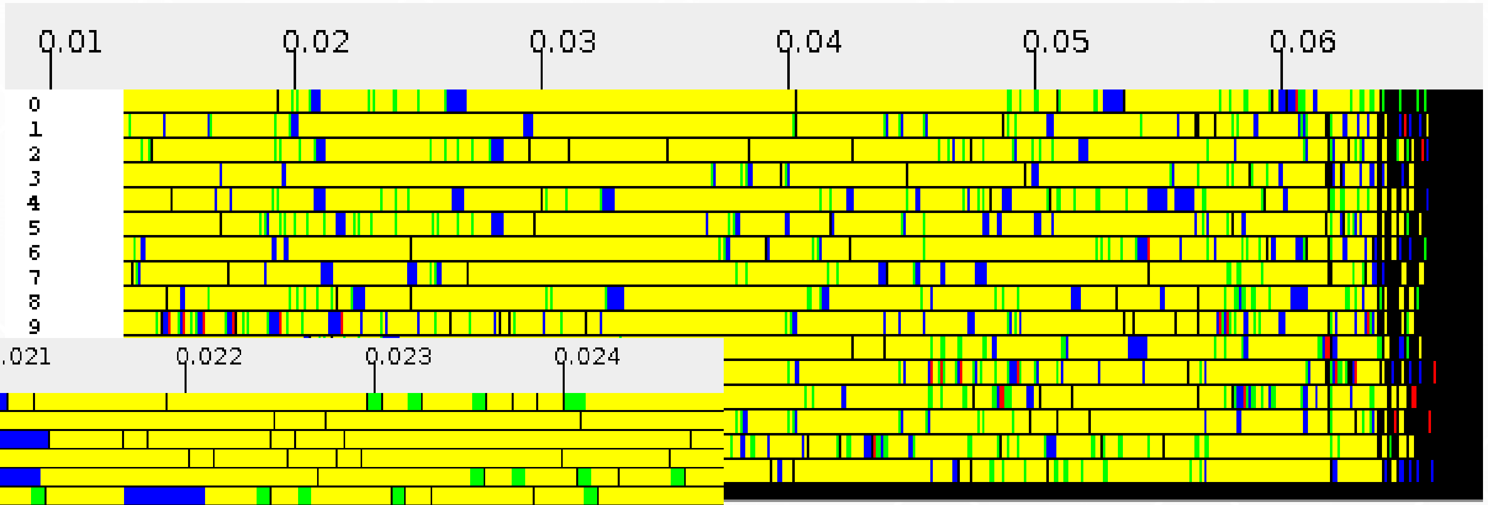
- **Platform: Intel Haswell** E5-2680V3 2.50 GHz, 12core 24 threads x 2 sockets
- Problem size : 4096 x 128

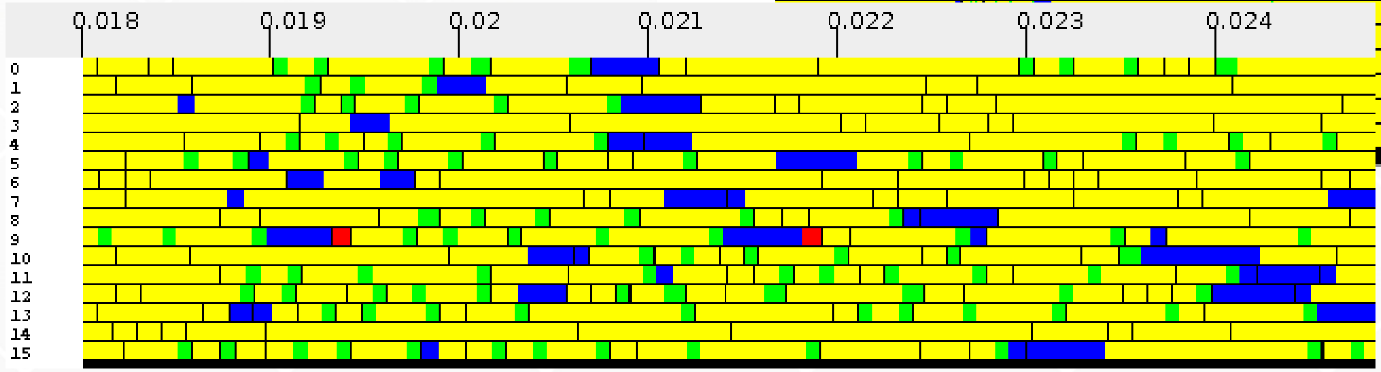


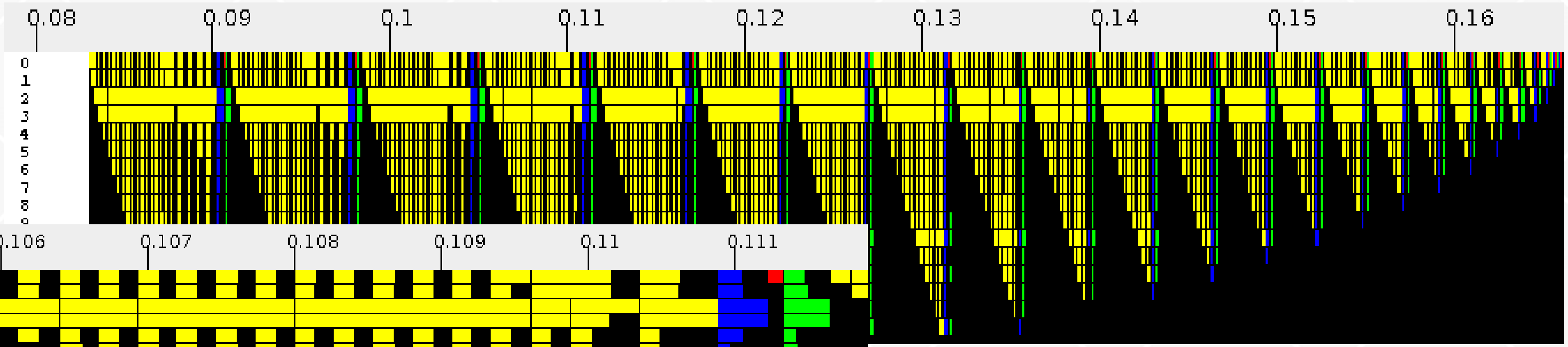
Taking a look closer at execution profile!

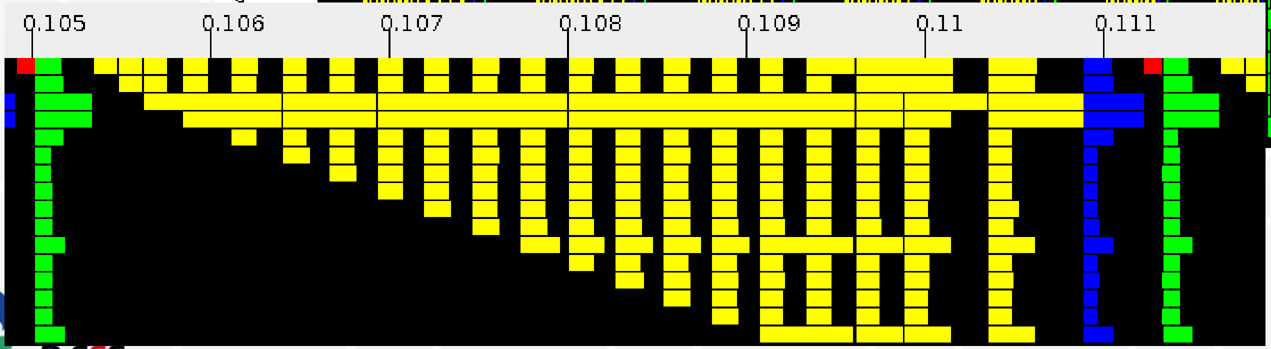
note: #threads = 16

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?

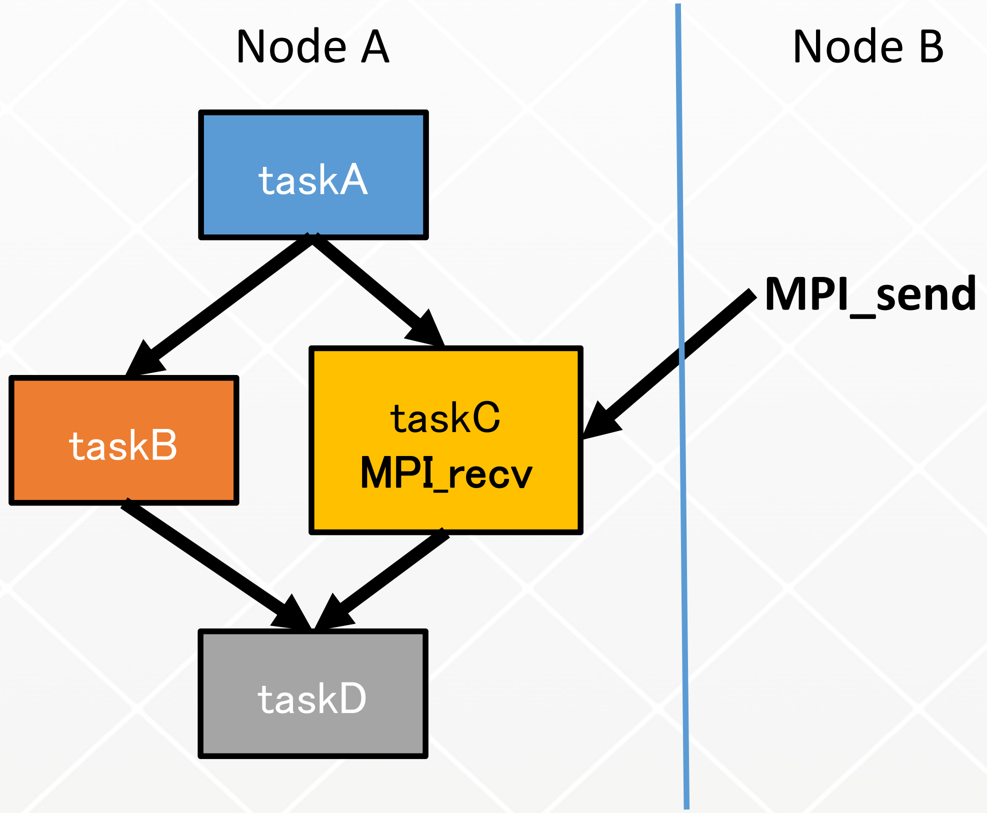
- **X is OpenMP!**
- **“MPI+Open” is now a standard programming for high-end systems.**

- **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.

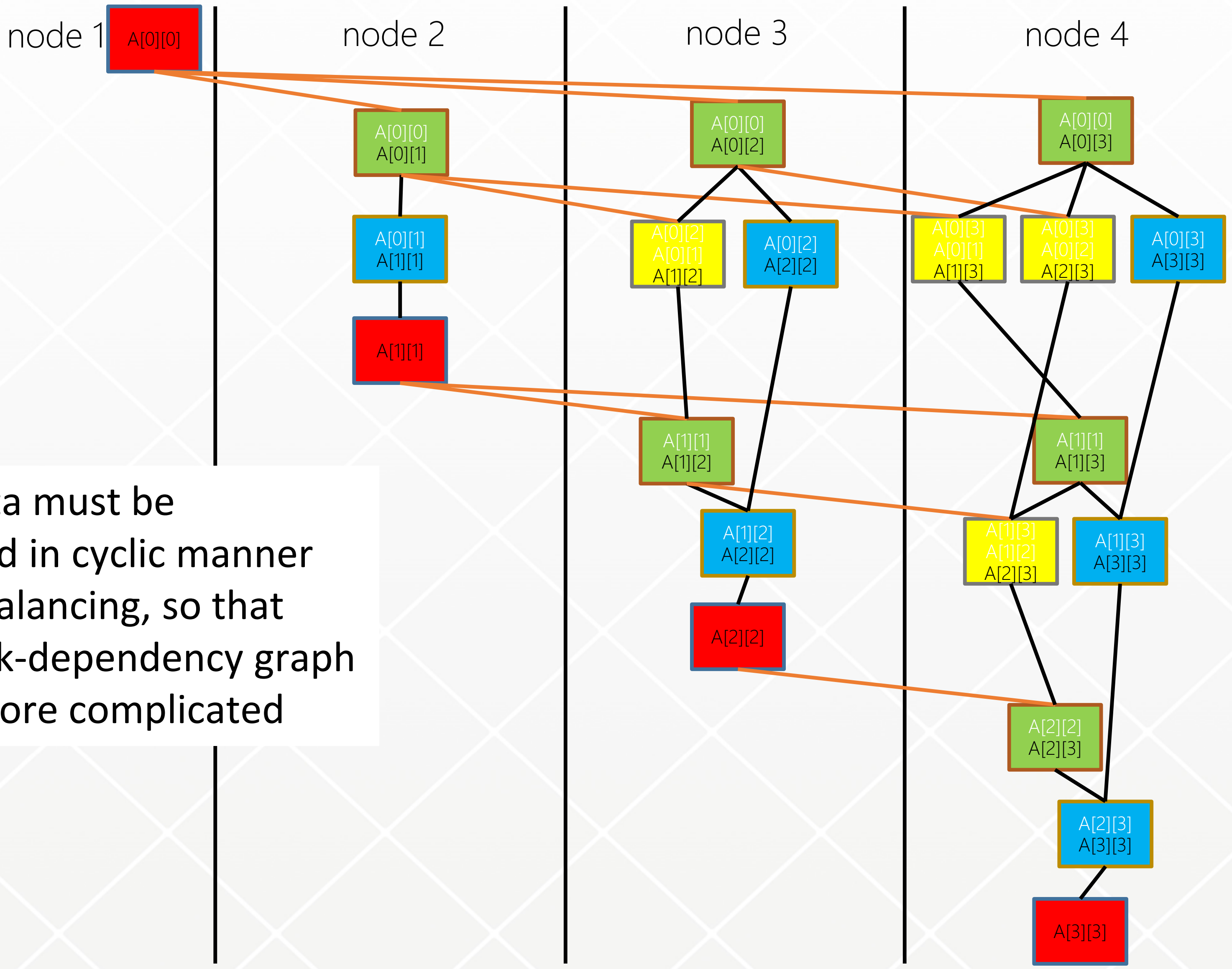
```
#pragma omp parallel
#pragma omp single
{
#pragma omp task depend(out:A)
    A = ...;                          /* taskA */
#pragma omp task depend(in A, out B)
    B = foo(A)                        /* taskB */
#pragma omp task depend(in:A) depend(out:C)
{   MPI_recv(....);    /* communication */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

A Case of Cholesky Factorization in Distributed memory

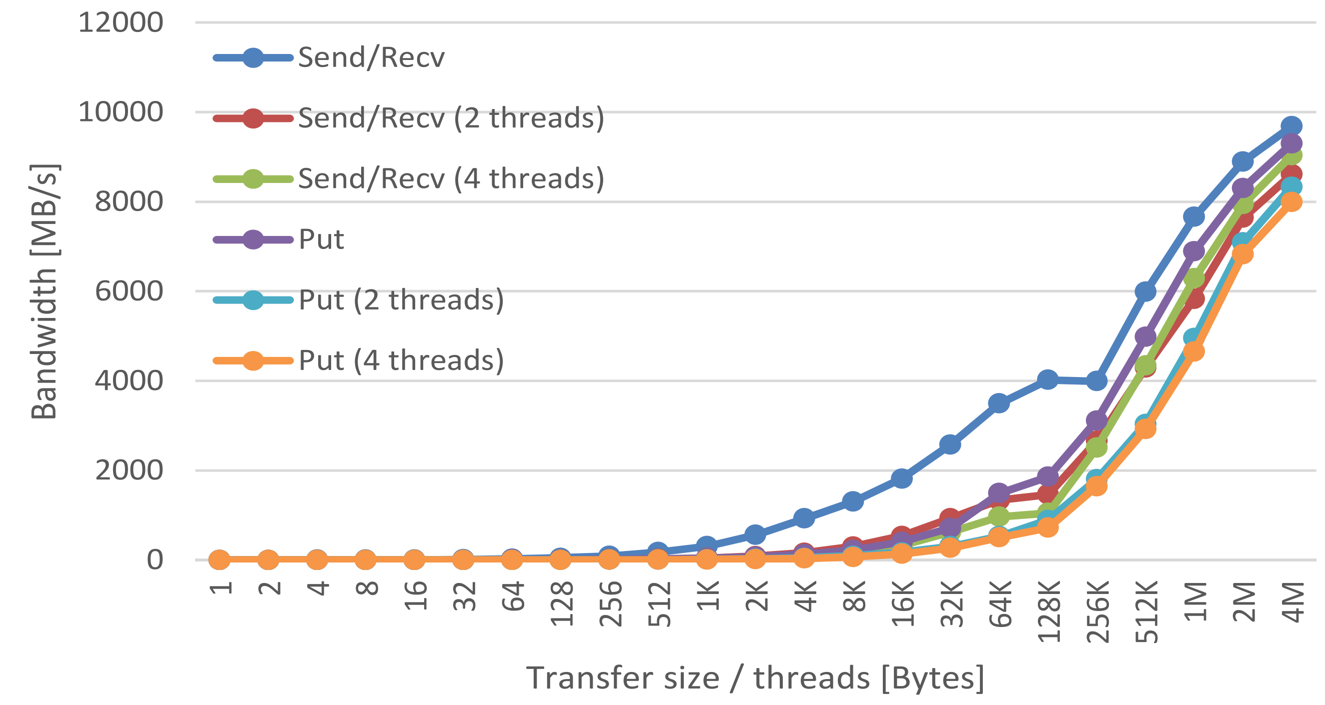
- Data dependency between nodes involves data communications.



Note: data must be distributed in cyclic manner for load balancing, so that actual task-dependency graph may be more complicated

A Problem: communication between threads in manycore

- **Aggregate comm. performance between multiple threads between nodes in KNLs**
  - The performance using “Send/Recv” may be better than that using “Put”.
  - <u>the performance of MPI_THREAD_MULTIPLE is lower than that of the single-threaded communication.</u>



- **Why MPI_THREAD_MULTIPLE is so slow?**
  - “giant-lock” for message ordering and tag matching for wild-card.
  - New proposal is being discussed: “end-point” for thread.
- <u>**Can RDMA be another “light-weight” solution for communication in multithreaded execution of manycore?**</u>

Thread-safety of MPI: How you can use MPI in tasks

- **MPI_Init_thread get supported thread-safety level**

- **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.

“MPI+X” for exascale?

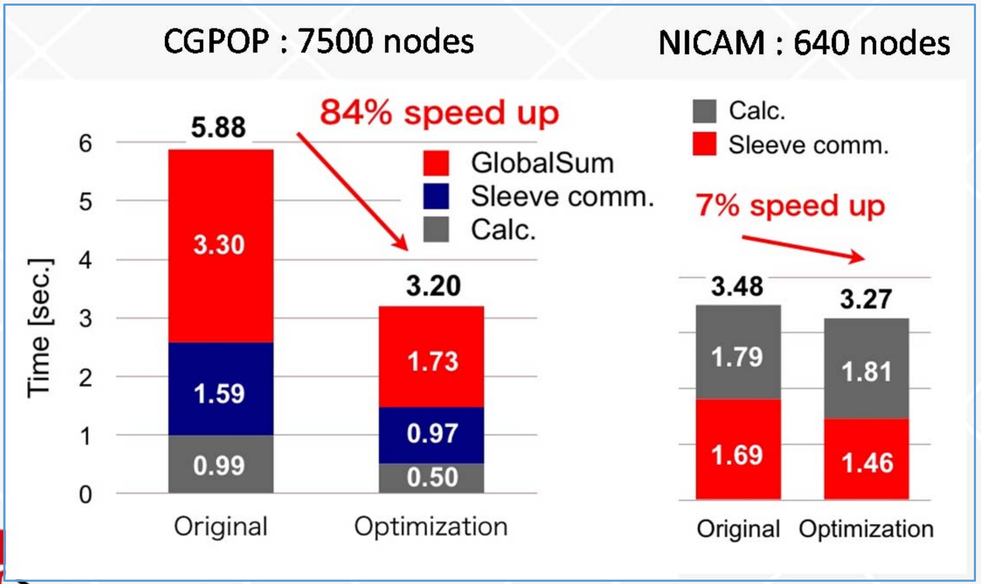
- **X is OpenMP!**
- **“MPI+Open” is now a standard programming for high-end systems.**

- **Questions:**
  - “MPI+OpenMP” is still a main programming model for exa-scale?
  - MPI has some problem on comm. in multithreaded execution (tasks) !!!
  - Can PGAS replace(beat!?) “MPI” for exascale?

    not “MPI+X”, **but “PGAS+X”!!!**

PGAS (Partitioned Global Address Space) models

- Light-weight one-sided communication and low overhead synchronization semantics.
- PAGS concept is adopted in Coarray Fortran, UPC, X10, XcalableMP.
  - XMP adopts notion Coarray not only Fortran but also “C”, as “local view” as well as “global view” of data parallelism.
- Advantages and comments
  - Easy and intuitive to describe, not noly one side-comm, but also stride comm.
  - Recent networks such as Cray and Fujitsu Tofu support remote DMA operation which strongly support efficient one-sided communication.
  - Other collective communication library (can be MPI) are required.

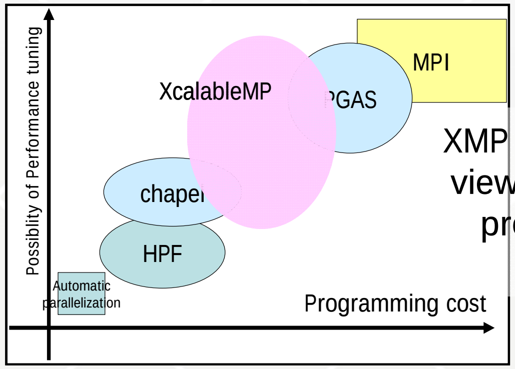


Case study of XMP on K computer
CGPOP, NICAM: Climate code

5-7 % speed up is obtained by replacing MPI with Coarray

XcalableMP(XMP) http://www.xcalablemp.org

- **What's XcalableMP (XMP for short)?**
  - A PGAS programming model and language for distributed memory , proposed by **XMP Spec WG**
  - XMP Spec WG is a special interest group to design and draft the specification of XcalableMP language. It is now organized under **PC Cluster Consortium**, Japan. Mainly active in Japan, but open for everybody.
- **Project status (as of June 2016)**
  - XMP Spec **Version 1.3** is available at XMP site. new features: mixed OpenMP and OpenACC , libraries for collective communications.
  - Reference implementation by U. Tsukuba and Riken AICS: **Version 1.2 (C and Fortran90)** is available for PC clusters, Cray XT and K computer. Source-to-Source compiler to code with the runtime on top of MPI and GasNet.
- **HPCC class 2 Winner 2013. 2014**



XMP provides a global view for data parallel program in PGAS model

- Language Features
  - Directive-based language extensions for Fortran and C for PGAS model
  - Global view programming with global-view distributed data structures for data parallelism
    - SPMD execution model as MPI
    - pragmas for data distribution of global array.
    - Work mapping constructs to map works and iteration with affinity to data explicitly.
    - Rich communication and sync directives such as "gmove" and "shadow".
    - Many concepts are inherited from HPF
  - Co-array feature of CAF is adopted as a part of the language spec for local view programming (also defined in C).

Code example

```
int array[YMAX][XMAX];

#pragma xmp nodes p(4)
#pragma xmp template t(YMAX)
#pragma xmp distribute t(block) on p
#pragma xmp align array[i][*] to t(i)

main(){
  int i, j, res;
  res = 0;

#pragma xmp loop on t(i)  reduction(+:res)
  for(i = 0; i < 10; i++)
    for(j = 0; j < 10; j++){
      array[i][j] = func(i, j);
      res += array[i][j];
    }
}
```

data distribution

add to the serial code : incremental parallelization

work sharing and data synchronization

CAF: Co-Array Fortran

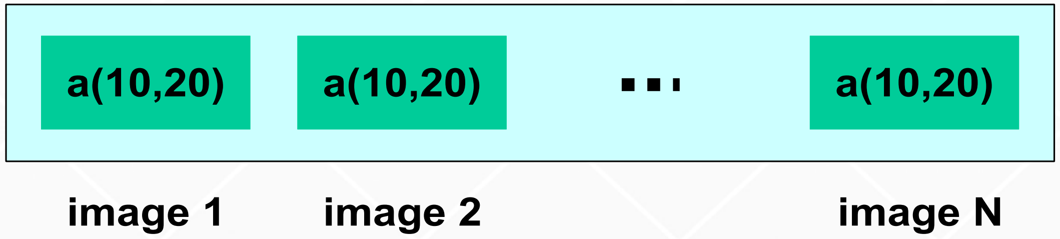
- **Global address space programming model**
  - one-sided communication (GET/PUT)
- **SPMD execution model**
- **Adopted in Fortran 2008**
- **Co-array extension**

  Each node (processor) has different "image"

  ```
  real, dimension(n)[*] :: x,y
  x(:) = y(:)[q]
  ```

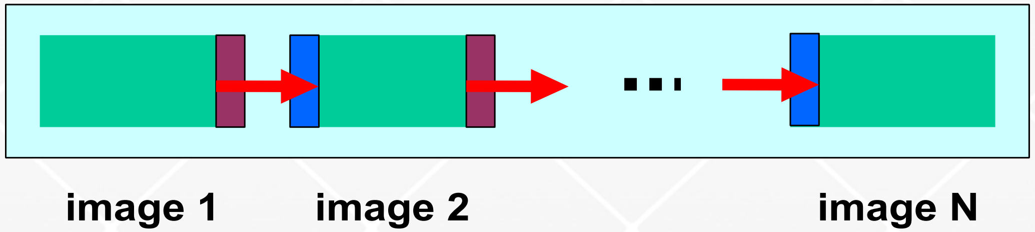
  Copy data y on image of q to local x (get)

```
integer a(10,20)[*]
```



- Coarray provides language construct for data transfer and synchronization

  Programmer can control:
  - Data distribution
  - Assignment of computation
  - Communications
- amenable to compiler-based communication optimization



```
if (this_image() > 1)
   a(1:10,1:2) =
     a(1:10,19:20)[this_image()-1]
```

Local-view XMP program: Coarray

- **XMP includes the coarray feature imported from Fortran 2008 for the local-view programming.**
  - Basic idea: data declared as *coarray* can be accessed by remote nodes.
  - Coarray in XMP/Fortran is fully compatible with Fortran 2008.

```
real b(8)[*]

if (xmp_node_num() == 1) then
  a(:) = b(:)[2]
```

b is declared as a coarray.

Node 1 *gets* b from node 2.

- **Coarrays can be used in XMP/C.**
  - The subarray notation is also available as an extension.

- Declaration

```
float b[8]:[*];
```

- Put

```
a[0:3]:[1] = b[3:3];
```

puts b to node 1.

- Get

```
a[0:3] = b[3:3]:[2];
```

gets b from node 2.

- Synchronization

```
void xmp_sync_all(int *status)
```

PGAS and remote memory access (RMA)/one-sided comm.

- PGAS is a programming model relating to distributed memory system with a shared address space that distinguishes between local (cheap) and remote (expensive) memory access.
  - Easy and intuitive to describe remote data access, for not only one side-comm, but also stride comm.
- RMA is a mechanism (operation) to access data in remote memory by giving address in (shared) address space.
  - RDMA is a mechanism to directly access data in remote memory without involving the CPU or OS at the destination node.
  - Recent networks such as Cray and Fujitsu Tofu support remote DMA operation which strongly support efficient one-sided communication.
- PGAS is implemented by RMA providing light-weight one-sided communication and low overhead synchronization semantics.
- For programmers, both PGAS and RMA are programming interfaces and offer several constructs such as remote read/write and synchronizations.
  - MPI3 provides several RMA (one-sided comm.) APIs as library interface.

“Compiler-free” approaches for PGAS

- (Language: UPC, CAF, Chapel, XcalableMP)
- Library approach: MPI3 RMA, OpenShmem, GlobalArray, …
- C++ Template approach: UPC++, DASH, …
- This approach may increase portability, clean separation from base compiler optimization, … but sometimes hard to debug in C++ template…

- But, approach by compiler will give:
  - New language, or language extension provides easy-to-use and intuitive feature resulting in better productivity.
  - Enable compiler analysis for further optimization: removal of redundant sync and selection of efficient communication, etc, …
  - But, in reality, compiler-approach is not easy to be accepted for deployment, and support many sites, …

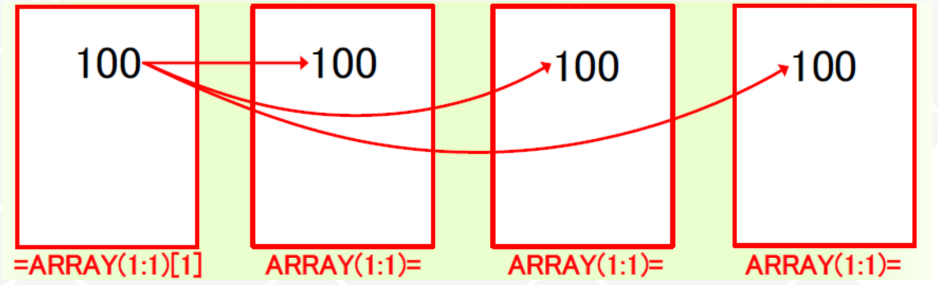
Should everything be written in PGAS?

- MPI broadcast operation can be written in CAF easily, …
- But, is it efficient?

```
REAL(8),DIMENSION(:),ALLOCATABLE :: ARRAY
 ・・・
CALL MPI_COMM_SIZE(MPI_COMM_WORLD,NIMG,IERR)
CALL MPI_COMM_RANK(MPI_COMM_WORLD,ID,IERR)
 ・・・
CALL MPI_BCAST(ARRAY, MSGSIZE, MPI_REAL8, 0,
MPI_COMM_WORLD,IERR)
```

```
REAL(8),DIMENSION(:),CODIMENSION[:],ALLOCATABLE :: ARRAY
 ・・・
NIMG= NUM_IMAGES()
ID= THIS_IMAGE()
 ・・・
SYNC ALL
IF(ID /= 1) THEN
ARRAY(1:MSGSIZE) = ARRAY(1:MSGSIZE)[1]
END IF
SYNC ALL
```

- Should use “co_broadcast” in CAF.
- Sophisticated collective communication libraries of “matured” MPI are required
- Obviously, PGAS need to **collaborate** with MPI.

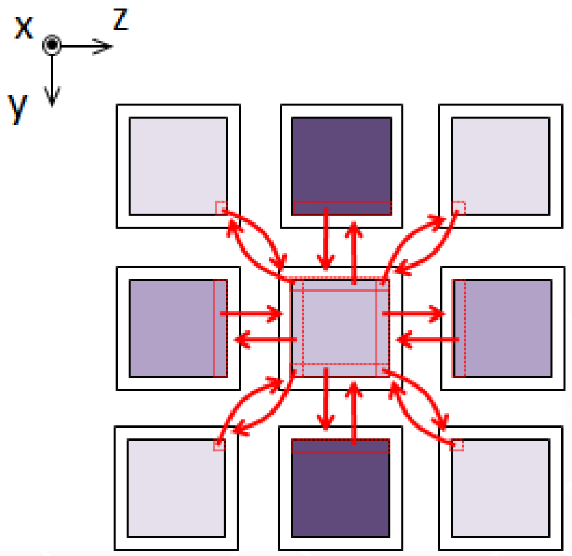


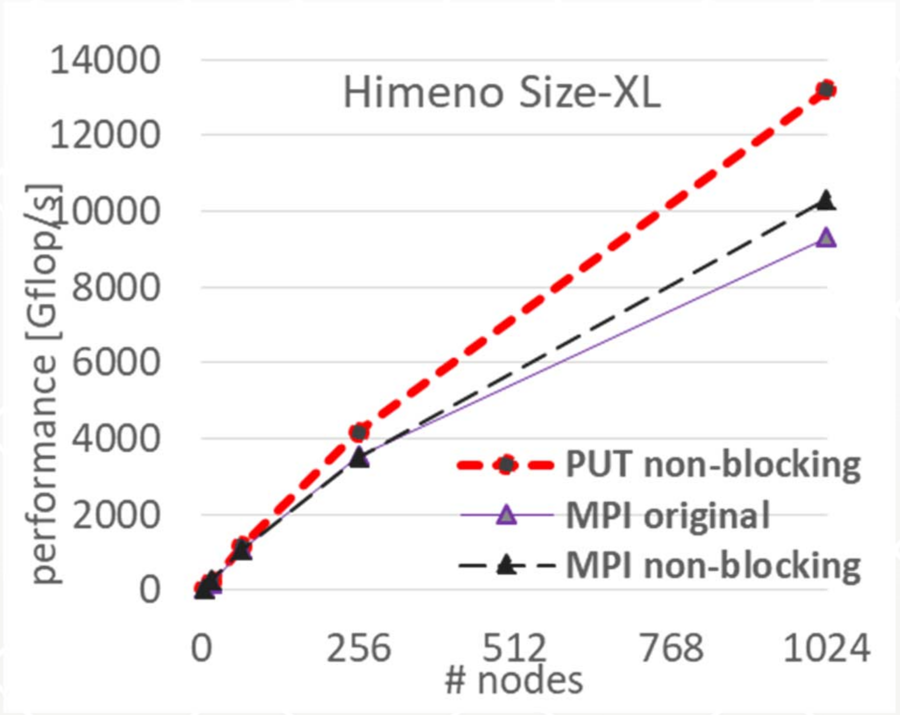
Can PGAS replace MPI? / Can PGAS be faster than MPI?

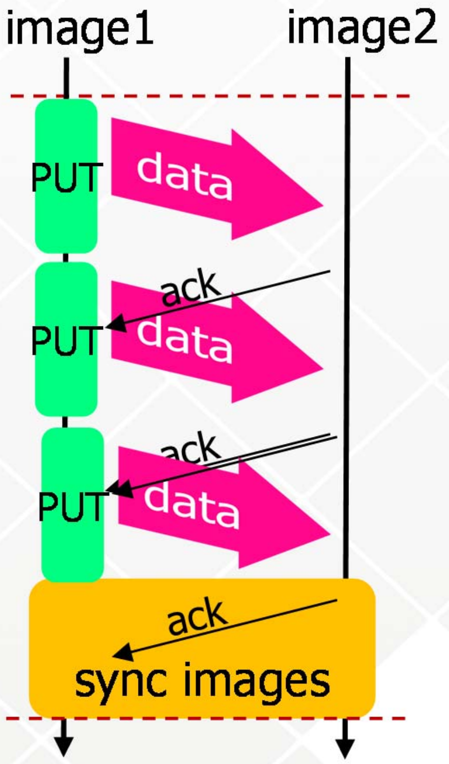
- **Advantages of RMA/RDMA Operations**
  - (Note: Assume MPI RMA is an API for PGAS)
  - multiple data transfers can be performed with a single synchronization operation
  - Some irregular communication patterns can be more economically expressed
  - Significantly faster than send/receive on systems with hardware support for remote memory access
    - Recently, many kinds of high-speed interconnect have hardware support for RDMA, including Infiniband, … as well as Cray and Fujitsu.
- **Simpler than MPI in the context of multithread env.**
  - It should be proven … not yet.

Case study: stencil communication

- Typical communication pattern in domain-decomposition.
- Advantage of PGAS: Multiple data transfers with a single synchronization operation at end
- PUT non-blocking outperforms MPI in Himeno Benchmark!
  - Don’t wait ack before sending the next data (by FJ-RDMA)







NOTE: The detail of this results is to be presented in HPCAisa 2018: Hidetoshi Iwashita, Masahiro Nakao, Hitoshi Murai, Mitsuhisa Sato, “A Source-to-Source Translation of Coarray Fortran with MPI for High Performance”

MPI RMA as a underlying comm. layer for PGAS?

- **“MPI is too low and too high API for communication”. (Prof. Marc Snir, JLESC 7th WS)**
  - MPI RMA APIs offer their PGAS model rather than “primitives” for other PGAS.
- **In case of our XMP Coarray implementation:**
  - Using “passive target”
  - MPI flush operation and synchronization do not match to implement “sync_images”.
  - Complex “window” management to expose the memory as a coarray.
  - (We need more study for better usage of MPI RMA)
  - Fujitsu RDMA interface is much faster in K-computer.

“MPI+X” for exascale?

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

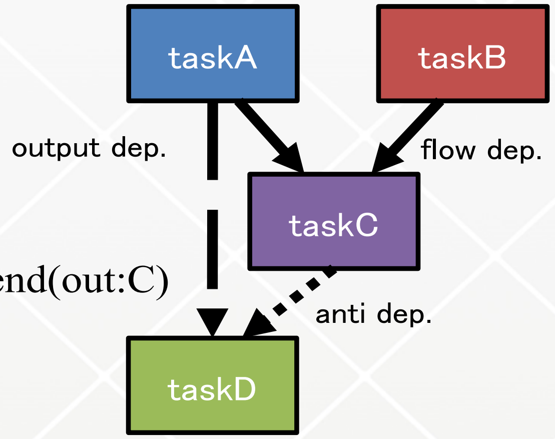
- **Questions:**
  - “MPI+OpenMP” is still a main programming model for exa-scale?
  - PGAS can beat “MPI” in some cases.
  - PGAS seems good in multithreading.

“PGAS +OpenMP task” ??!!

Task in OpenMP and extension to between nodes

- Task directive in OpenMP4.0 creates a task with dependency specified "depend" clause
- The task dependency depends on the order of reading and writing to data based on the sequential execution.
  - OpenMP multi-tasking model cannot be applied to tasks running in different nodes since threads of each nodes are running in parallel.
  - In OmpSs, interactions between nodes are described through the MPI task that is executing MPI communications (MPI task).
  - Otherwise, run the same task program in all nodes to resolve dep. (StarPU)

```
#pragma omp parallel
#pragma omp single
{
   int A, B, C;
#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 = 3;        /* taskD */
}
```



- Flow dependency: The flow dependency occurs between dependence-type out and in with same variables, similar to read after write (RAW) consistency. It is shown in between taskA and taskC with variable *A*, or taskB and taskC with variable *B*.
- Anti dependency: The anti dependency occurs between dependence-type in and out with same variables, similar to write after read (WAR) consistency. It is shown in between taskC and taskD with variable *A*.
- Output dependency: The output dependency occurs between dependence-type out and out with same variables, similar to write after write (WAW) consistency. It is shown in between taskA and taskC with variable *A*.

Multitasking in XMP: our proposal

- **For Intra-node**
  - Tasklet directive creates a task with dependency specified "in/out/inout" clause in sequential order
  - Same as in OpenMP4.0 and OMPss
- **For Inter-node**
  - Describe communications by PGAS operations (Coarray and gmove)
  - Annotated by get/put and get_ready/put_ready clauses to specify dependency.

`#pragma xmp tasklet` *tasklet-clause[, tasklet-clause[, ...]]* on {*node-ref* | *template-ref*}
(structured-block)

where *tasklet-clause* is :
{`in` | `out` | `inout`} (*variable[, variable, ...]*)
or
{`put` | `get`} (*tag*)
or
{`put_ready` | `get_ready`} (*variable*, {*node-ref* | *template-ref*}, *tag*)

`#pragma xmp taskletwait` *[on {node-ref | template-ref}]*

Put operation in tasklet

- put_ready clause: indicates that the specified data may be written by the associated PUT operation
  - This clause has the dependence-type out for the specified data on a node since its values are overwritten by the remote node.
- put clause: indicates that the PUT operation may be performed in the associated structured block.
  - At the beginning of the block, the task waits to receive the post notification with the tag by the put_ready clause to indicates that the data is exposed in the target node for the PUT operations.

- When output dependencies for the data are satisfied before executing the block, the clause exposes the data for the PUT operation from the specified set of nodes by sending the post notifications to these nodes, starting the PUT operations eventually in remote nodes. Then, it waits until remote operations are done. When the task receives the completion notification of the PUT operation, the block is immediately scheduled.
- When the post notification is received, the task is scheduled to execute the calculation and PUT operation in the block. When the execution of the block is finished, the data written by the PUT operation is flushed and the completion notification is sent to the node matched by the tag.

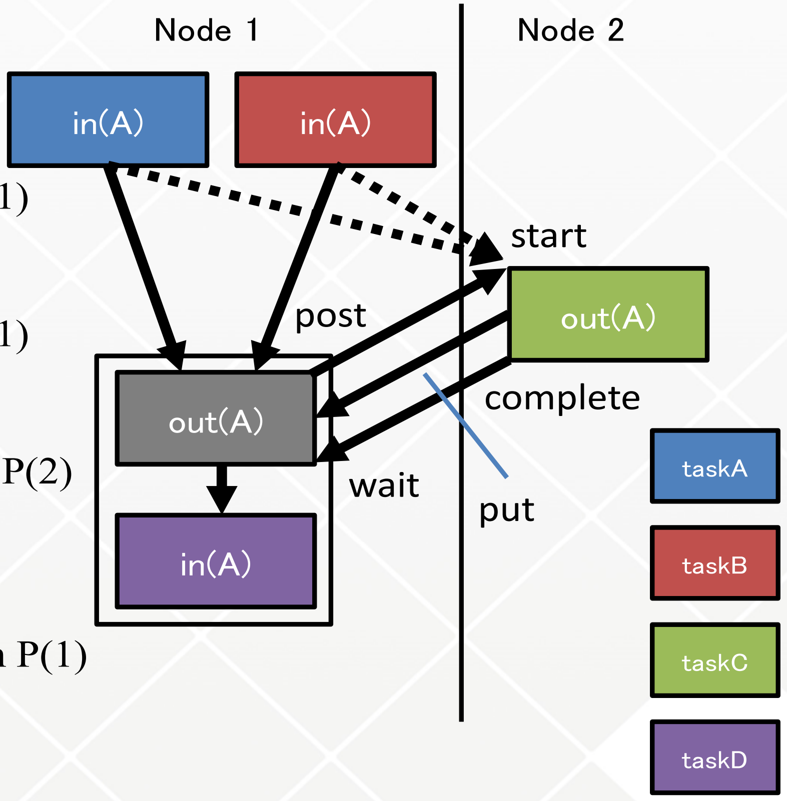
```
#pragma xmp nodes P(2)
   int A:[*], B, C, D, tag;

#pragma xmp tasklet in(A) out(B) on P(1)
   B = A;        /* taskA */

#pragma xmp tasklet in(A) out(C) on P(1)
   C = A;        /* taskB */

#pragma xmp tasklet out(A) put(tag) on P(2)
   A:[1] = 1;    /* taskC */

#pragma xmp tasklet in(A) out(D) ¥
                 put_ready(A, P(1), tag) on P(1)
   D = A;        /* taskD */
```



Summary: “MPI+X” for exascale?

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

PGAS (RDMA) +
MPI(collective) +
OpenMP task (+data parallel)
+ accelerator(GPU) offloading

We may need more unified and sophisticated programming model?