Multi-SPMD Programming Paradigm for Extreme Computing

Miwako TSUJI

RIKEN AICS, JAPAN

Agenda

INTRODUCTION

Multi SPMD Programming model

Overview

Background

Experiments

Collaborations with

numerical library group

accelerator group

Fault Tolerance in the Multi SPMD

CONCLUSION

FP3C Framework and Programming for Post Petascale Computing

- September. 2010 – March. 2014
- Various research fields and their integration
  - Programming model and programming language design
  - Runtime libraries
  - Accelerator
  - Algorithm and mathematical libraries
  - etc…

Agenda

INTRODUCTION

Multi SPMD Programming model

Overview

Background

Experiments

Collaborations with

numerical library group

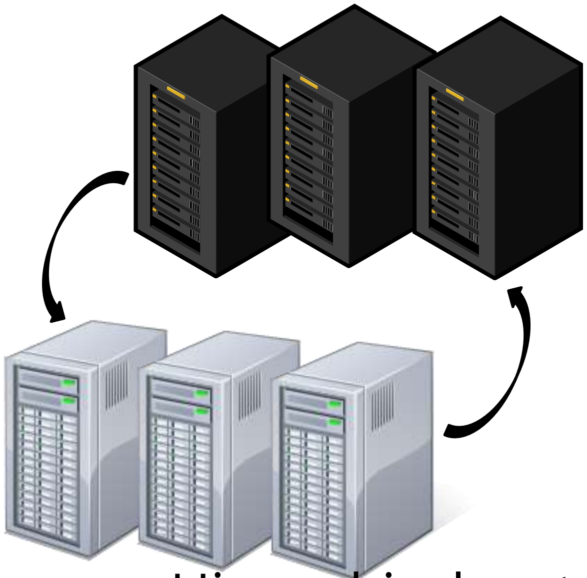
accelerator group

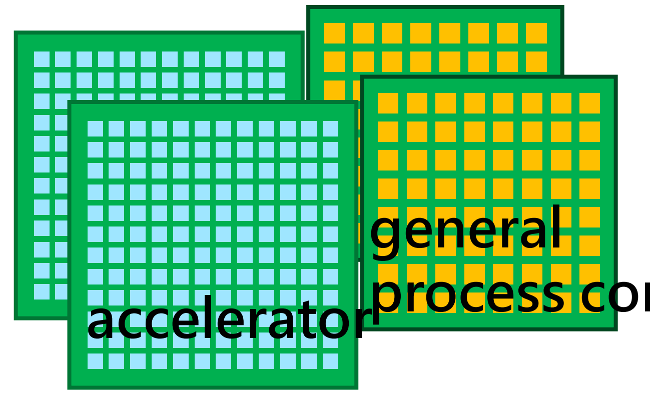
Fault Tolerance in the Multi SPMD

CONCLUSION

AICS

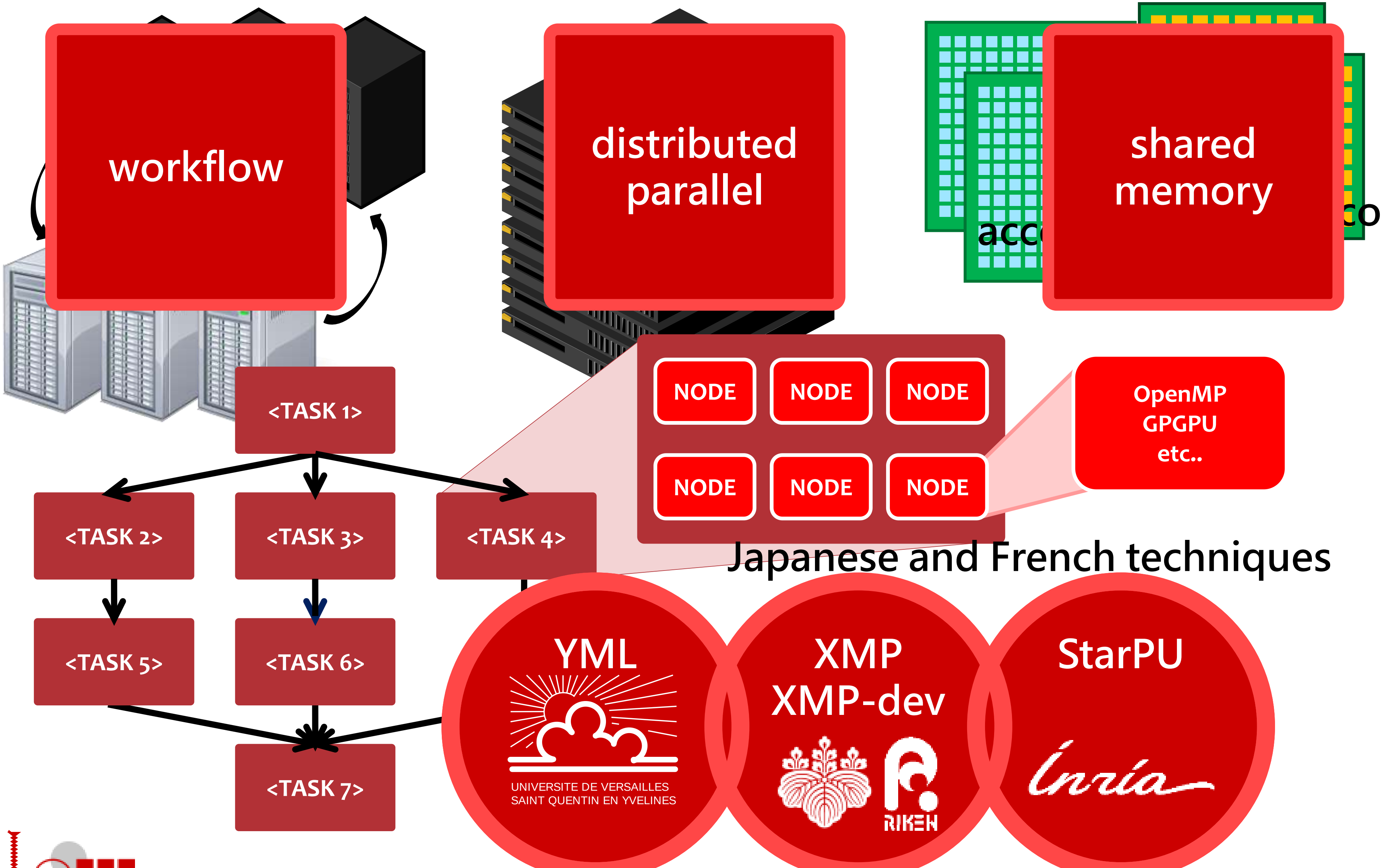
Multi-SPMD Programming MODEL



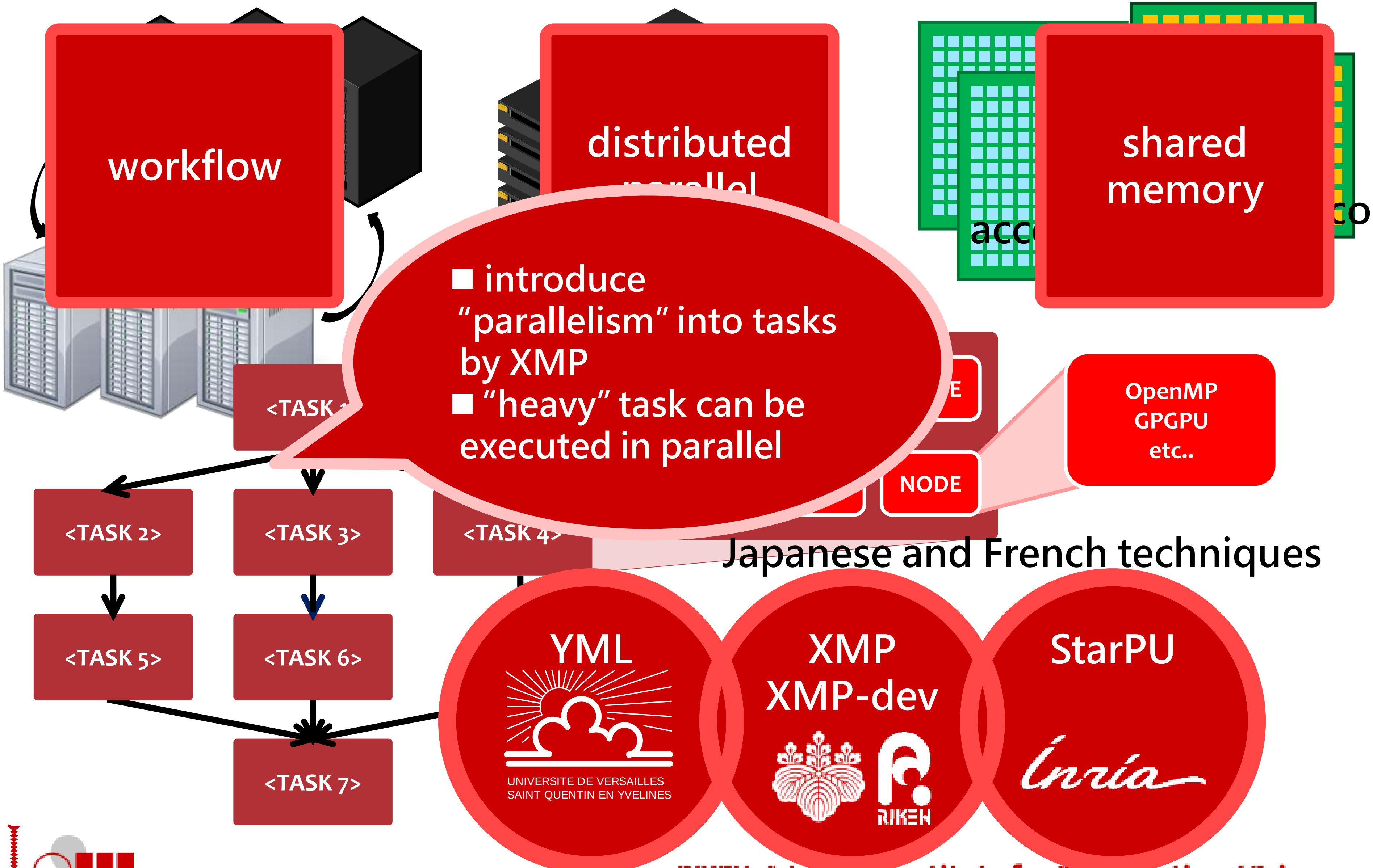


- Hierarchical systems
  - A node may consist of many general cores and accelerator cores
  - A group of nodes tightly connected
  - A system consists of groups of nodes / a cluster of clusters
- Multi-programming methodologies across multi-architectural levels
- Software had been developed to execute applications based on this programming model

Multi-SPMD Programming MODEL

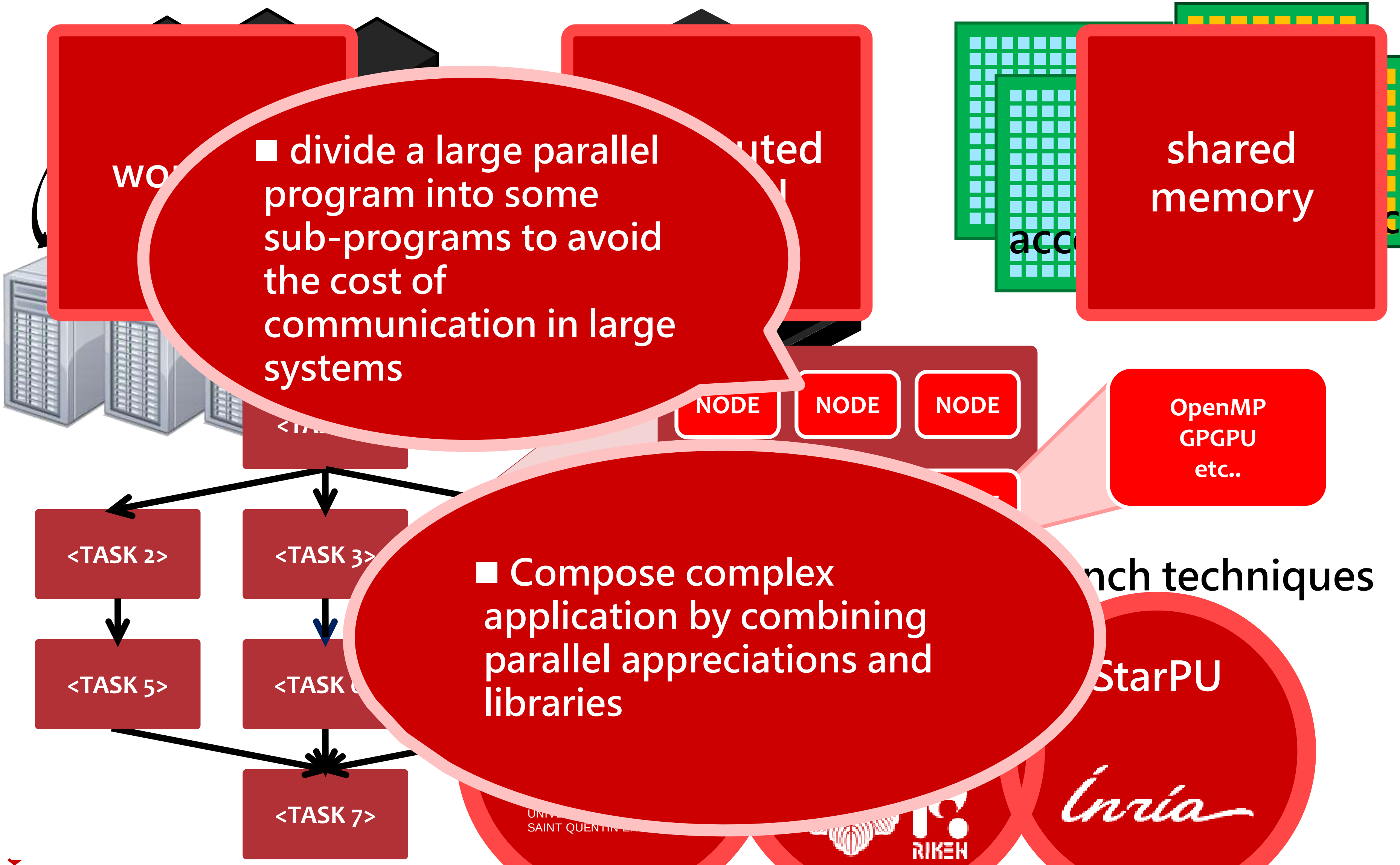


Multi-SPMD Programming MODEL



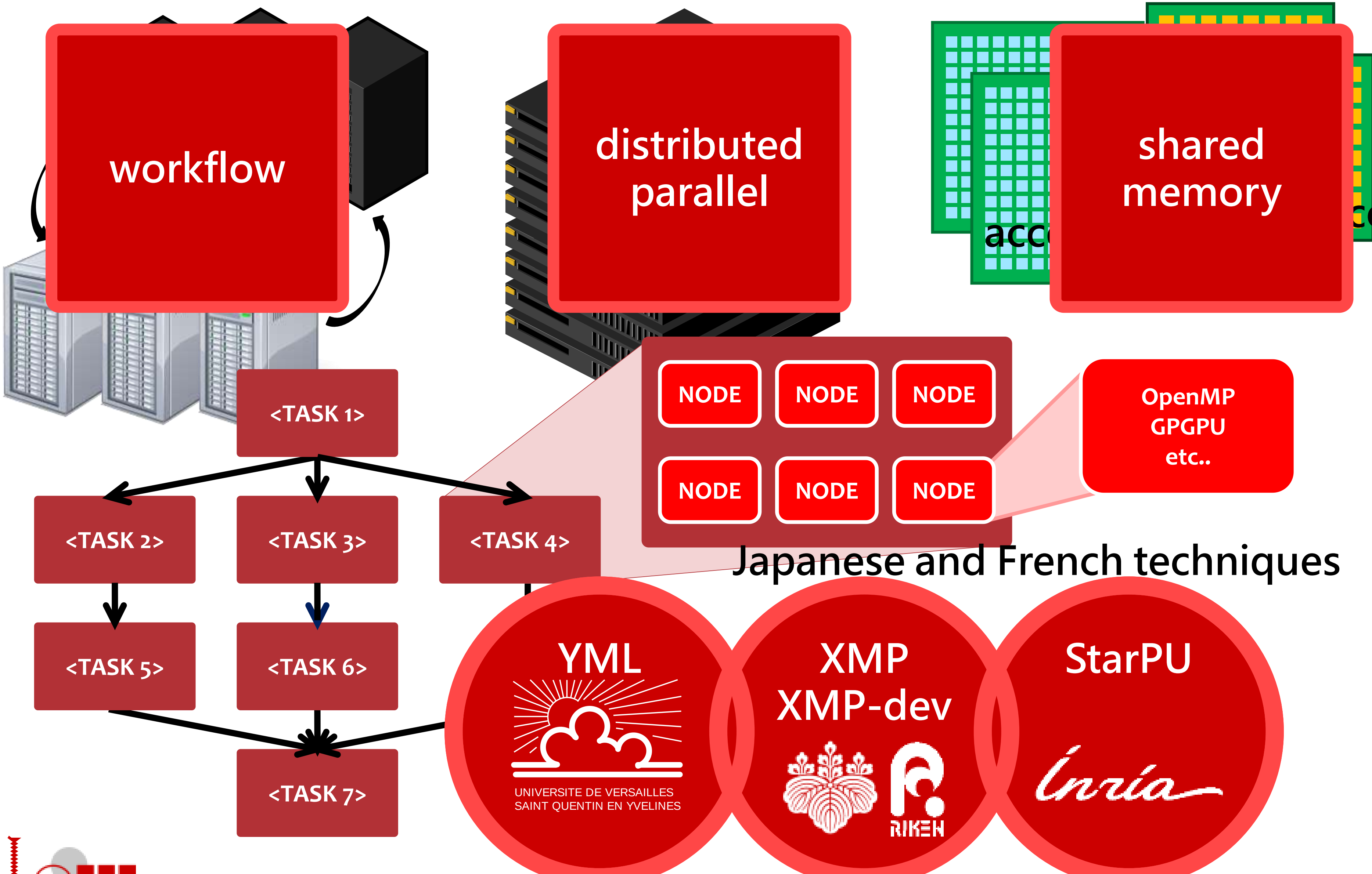
Multi-SPMD Programming MODEL

- divide a large parallel program into some sub-programs to avoid the cost of communication in large systems



- Compose complex application by combining parallel appreciations and libraries

Two cores: YML and XMP



Background

XcalableMP (XMP)

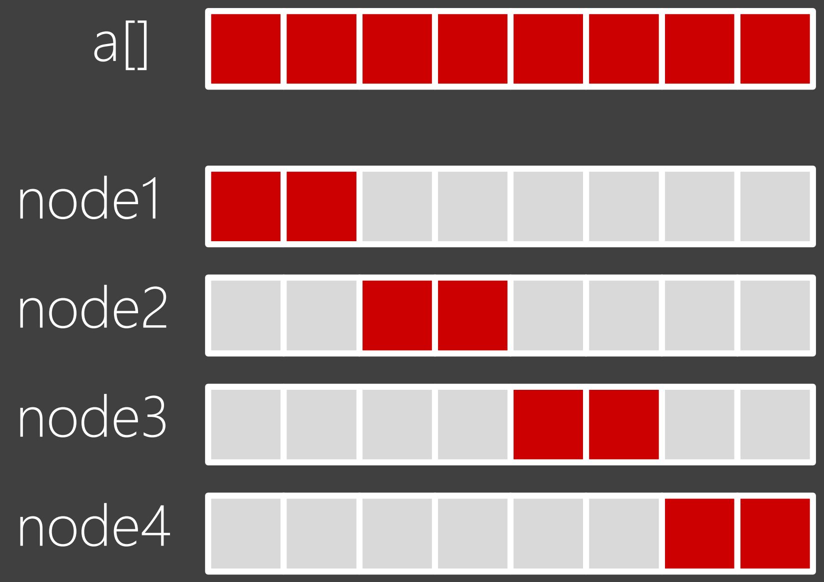
http://www.xcalablemp.org/

- Directive-based language extension for scalable and performance-aware parallel programming
- In XMP project, we have been developing a reference implementation of XMP compiler.
- XMP source code

  → C (or fortran) source code with XMP runtime library calls (MPI).
- Data mapping & Work mapping using template

```
#pragma xmp nodes p(4)
#pragma xmp template t(0:7)
#pragma xmp distribute t(block) onto p
int a[8];
#pragma xmp align a[i] with t(i)

int main(){
#pragma xmp loop on t(i)
  for(i=0;i<8;i++)
    a[i] = i;
```



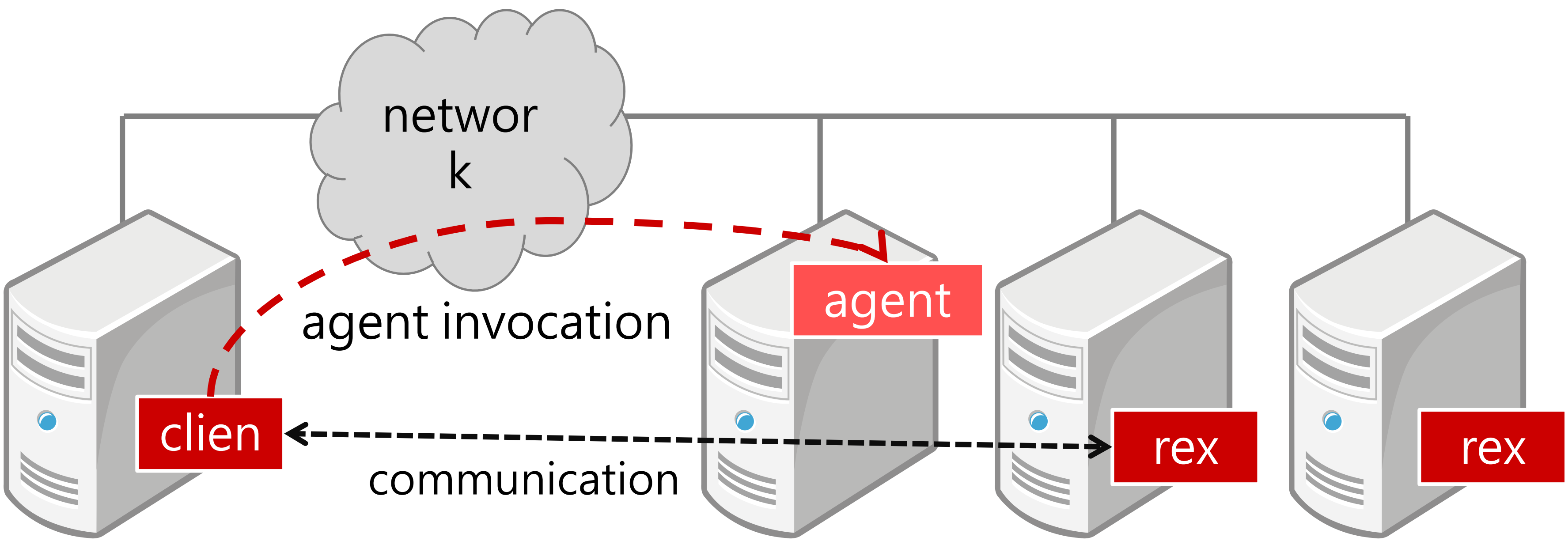
Background YML

http:// yml.prism.uvsq.fr/

- A workflow programming environment
  - Component generator
  - Workflow Compiler
  - Scheduler
    - Middleware : OmniRPC (Cluster) and XtreamWeb (P2P)
- Components
  - Abstract
    - definition of interface
  - Implementation
    - description of a remote program with a specific interface
    - C++ is supported.
    - We also support XMP!
  - Application
    - High level graph description language called YvetteML can be used to describe workflow

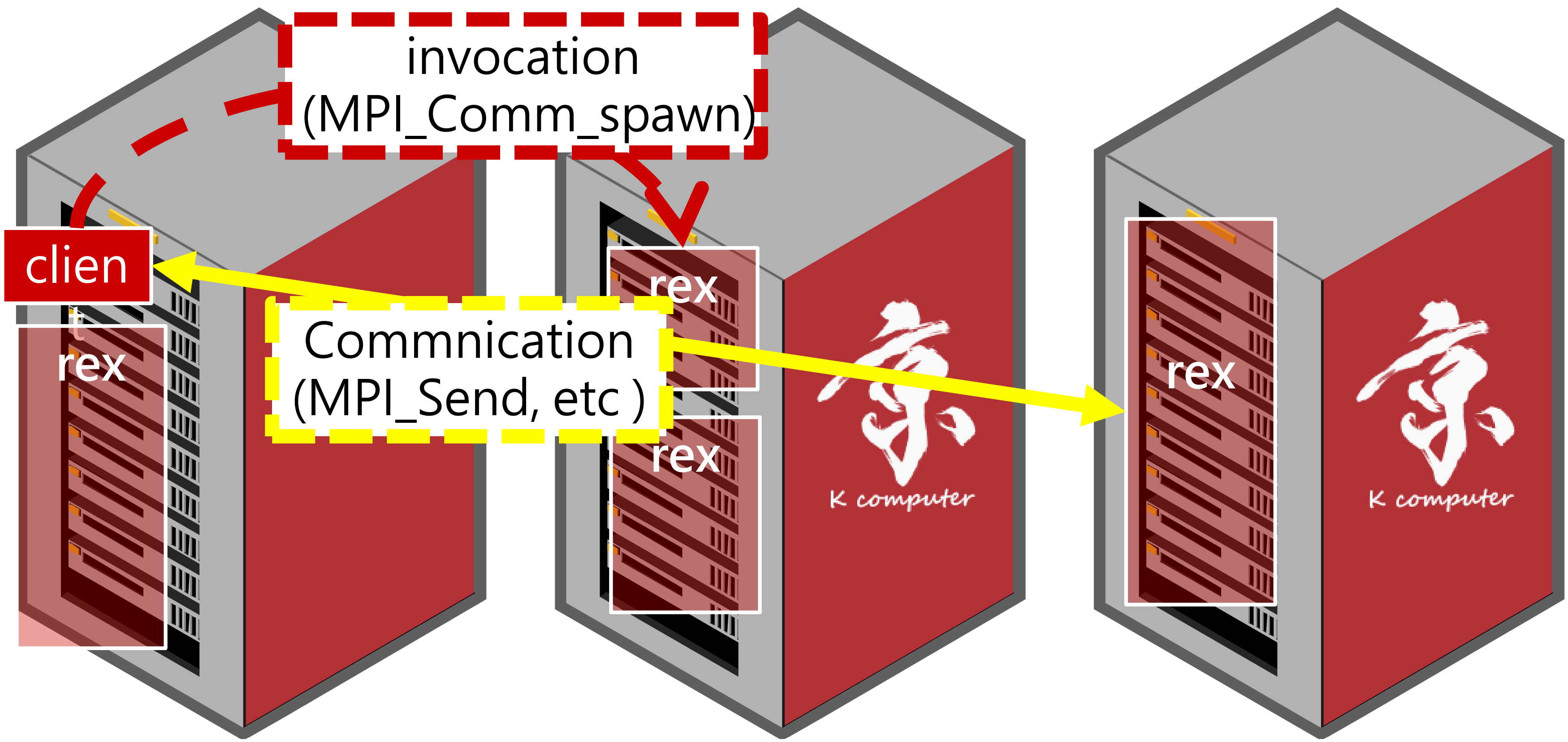
OmniRPC (Middleware)

- Mitsuhisa Sato, Motonari Hirano, Yoshio Tanaka, Satoshi Sekiguchi, "OmniRPC: A Grid RPC Facility for Cluster and Global Computing in OpenMP". Proc. of WOMPAT 2001, pp. 130-136, 2001.
- GridRPC (Remote Procedure Call)
- master-worker parallel program is supported
- remote programs (rex) executed by exec, rsh and ssh



OmniRPC-MPI (Middleware)

- OmniRPC extension for clusters
  - Remote programs can be executed in parallel



How to develop applications

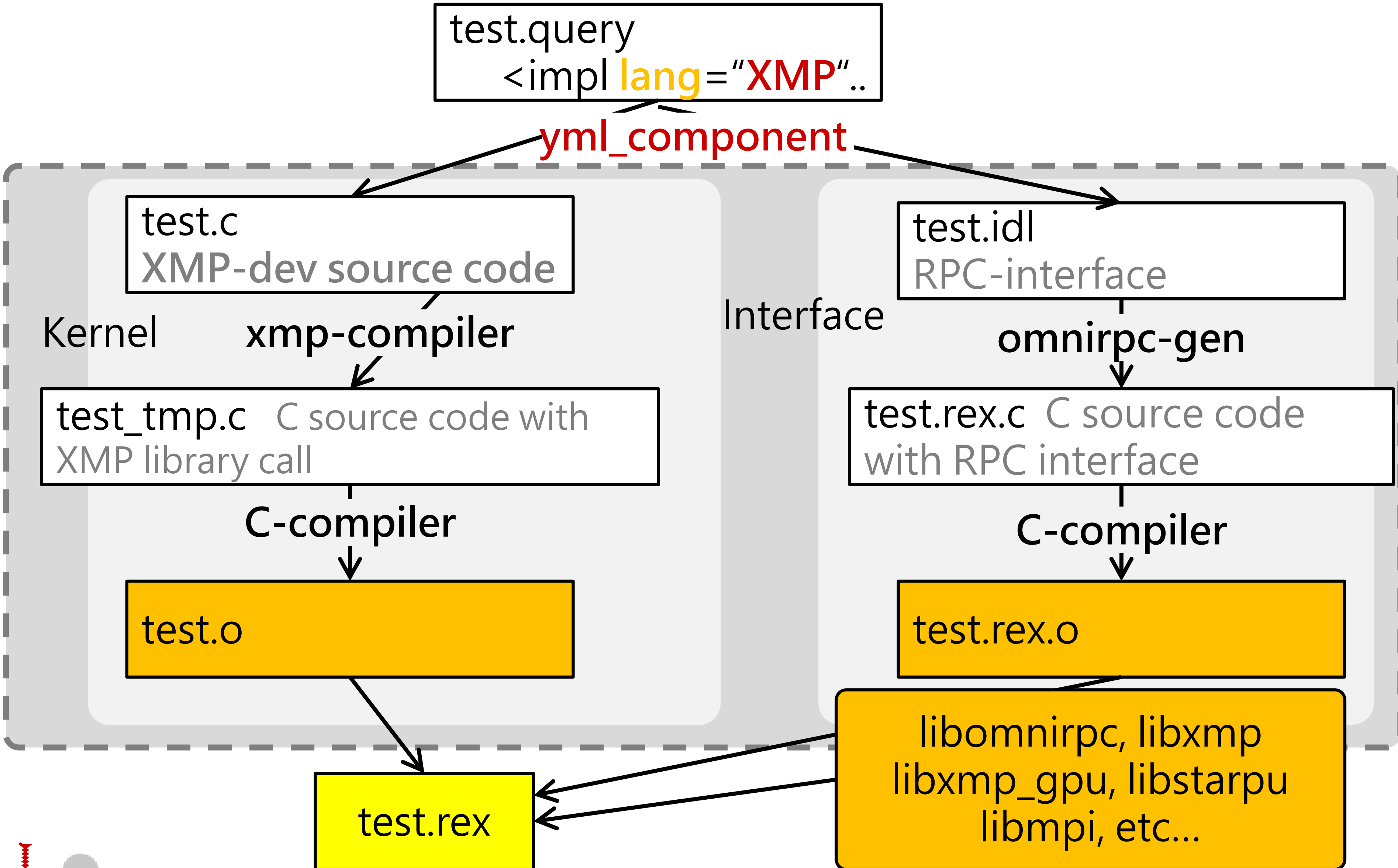
- Task development
  - Define interface (input/output) of a task
  - Define procedure of a task
    - C++, XMP, XMP-dev/StarPU, XMP for Fortran, MPI
      (The original YML supported only C++, parallel programming was not supported)

- Workflow development
  - Define dependency between tasks
    - YvetteML
  - Compile the definition into directed acyclic graph
    - yml_compiler
    - interpreted by yml_scheduler

Task development

```
<?xml version="1.0"?>
<component type="impl" name=“sample" abstract=“sample">
<impl lang="XMP" nodes="CPU:(16)" >
<templates>
<template name="t" format="block" size=“256"/>
</templates>
<distribute>
<param template="t"  name="A(256)" align="[i]:(i)"/>
<param template="t"  name="B(256)" align="[i]:(i)"/>
</distribute>
<source>
<![CDATA[
int i;
#pragma xmp loop (i) on t(i)
 for(i=0;i<256;i++){
   B[i] = A[i]*A[i];
.......
```

Task (Remote Program) Generator



Workflow Description in YvetteML

```
par
  A[i][j] is initialized at random
  B[i][j] is initialized as an unit matrix
endpar

par
  par(k:=0;count-1)
  do
    if (k neq 0) then
      wait(prodDiffA[k][k][k-1]);
    endif
    compute inversion(A[k][k],B[k][k]);
    notify(bInversed[k][k]);
    if (k neq count-1) then
      par (i:=k+1; count-1)
      do
        wait(bInversed[k][k]);
        compute prodMat(B[k][k],A[k][i]);
        notify(prodA[k][i]);
      enddo
    endif
    wait(bInversed[k][k]);
    par(i:=0;count-1)
    do
      if(i neq k) then
        compute mProdMat(A[i][k],B[k][k],B[i][k]);
        notify(mProdB[k][i][k]);
      endif
      if(k gt i) then
        compute prodMat(B[k][k],B[k][i]);
        notify(prodB[k][i]);
      endif
    enddo
    par(i:= 0;count-1)
    do
      if (i neq k) then
        if (k neq count - 1) then
          par (j:=k + 1;count-1)
          do
            wait(prodA[k][j]);
            compute prodDiff(A[i][k],A[k][j],A[i][j]);
            notify(prodDiffA[i][j][k]);
          enddo
        endif
        if (k neq 0) then
          par(j:=0;k-1)
          do
            wait(prodB[k][j]);
            compute prodDiff(A[i][k],B[k][j],B[i][j]);
          enddo
        endif
      endif
    enddo
  enddo
endpar
```

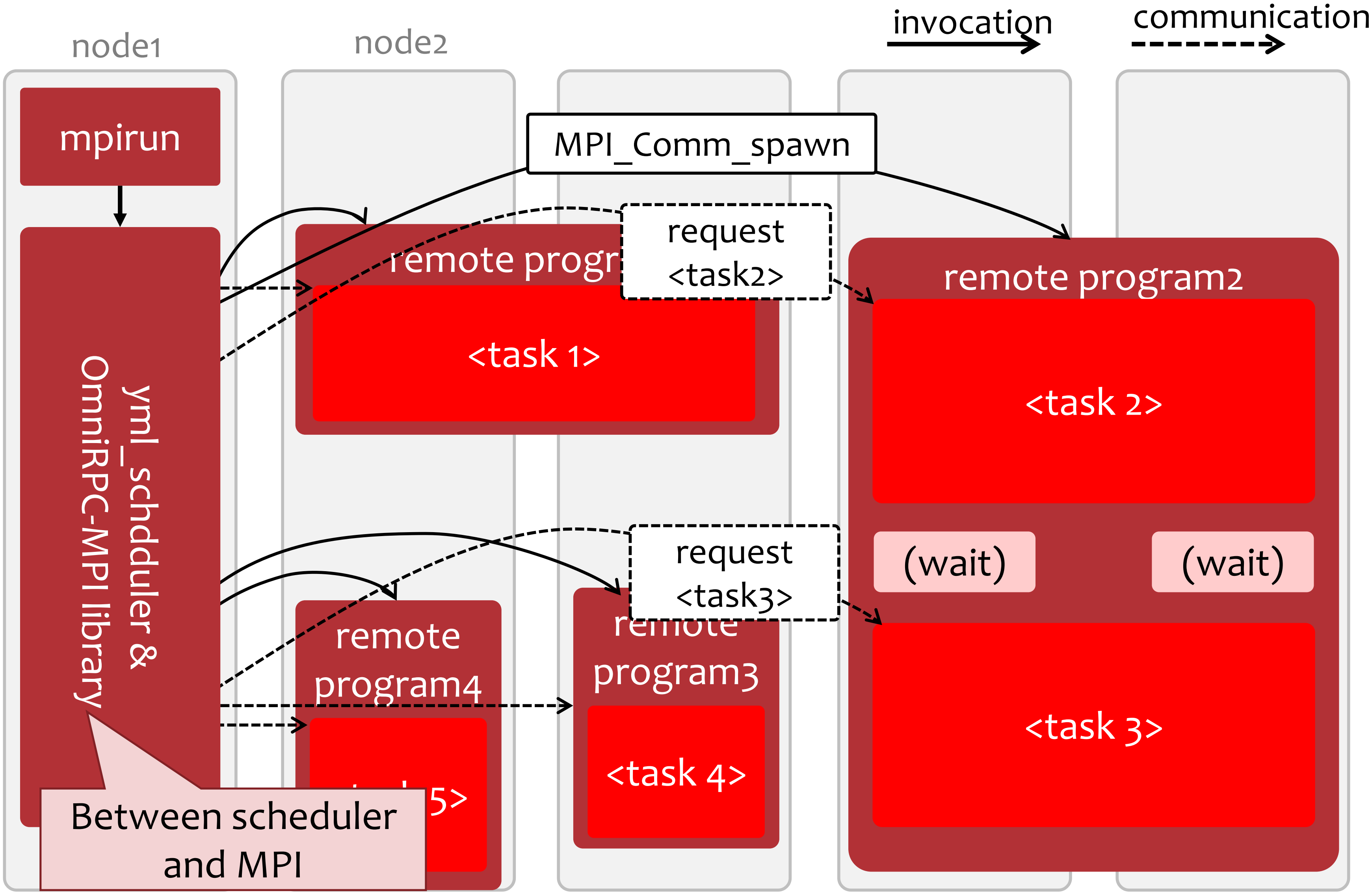
call task

notify-wait (dependency)

Parallel Execution

YvetteML: simple workflow language

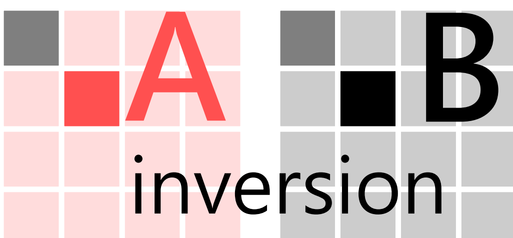
Execute an application

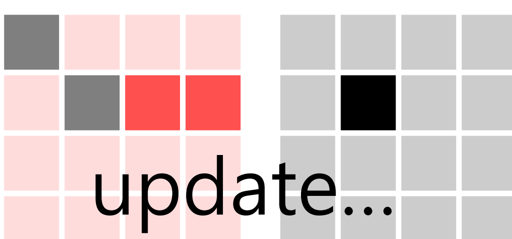


http://www.aics.riken.jp/jp/k/system.html#kcomputerslide

Experiment (1)

- Block Gauss Jordan
- B=A^{^-1}
  - Compute the inversion of a matrix by computing the inversion of a block and updating other blocks repeatedly





Computation node specs	CPU	SPARC64™ VIIIfx 2GHz
	Performance	128 GF (16 GF x 8 cores)
	Memory	16GB
Number of racks		864
Number of nodes		82,944
Network		Tofu Interconnect (6D Mesh/Torus)
Peak performance		10.62 PF
Total memory capacity		1.26 PB
File system		Fujitsu Exabyte File System (FEFS)
Storage		30 PB

Experiment (1) Block Gauss Jordan on K

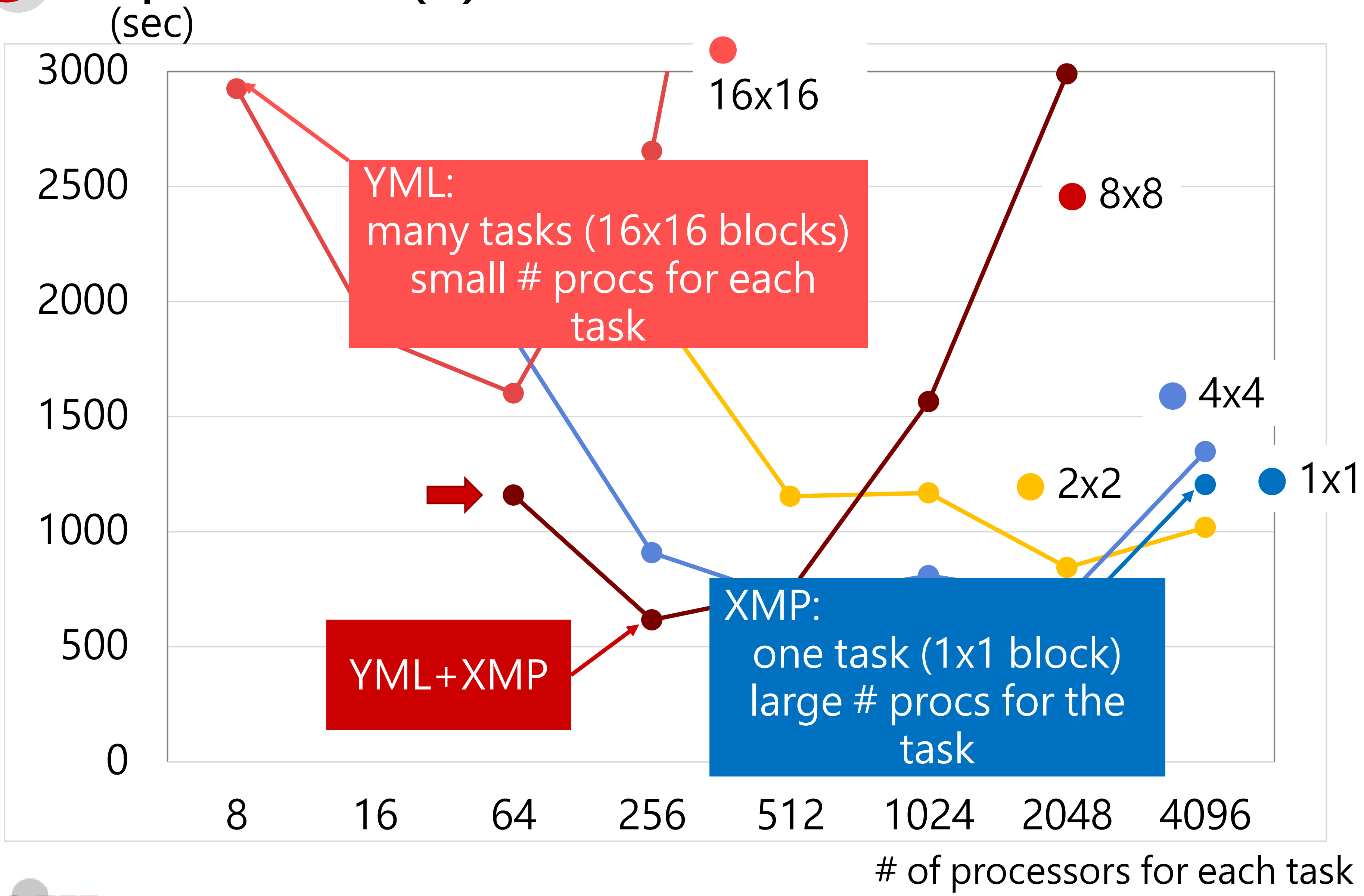
- Investigate different levels of hierarchical parallelism
  - the total size of matrix is fixed, but the number of blocks is varied
  - the total number of processes for a workflow is fixed, but the number of processes for each task is varied.

    ↓
  - “1x1 blocks & all processes for a task” ≒ distributed parallel program
  - A small # of processes for a task ≒ traditional workflow (the original YML)
- 32,768 x 32, 768 matrix

blocks	1x1	2x2	4x4	8x8	16x16
block size	32768	16384	8192	4096	2048

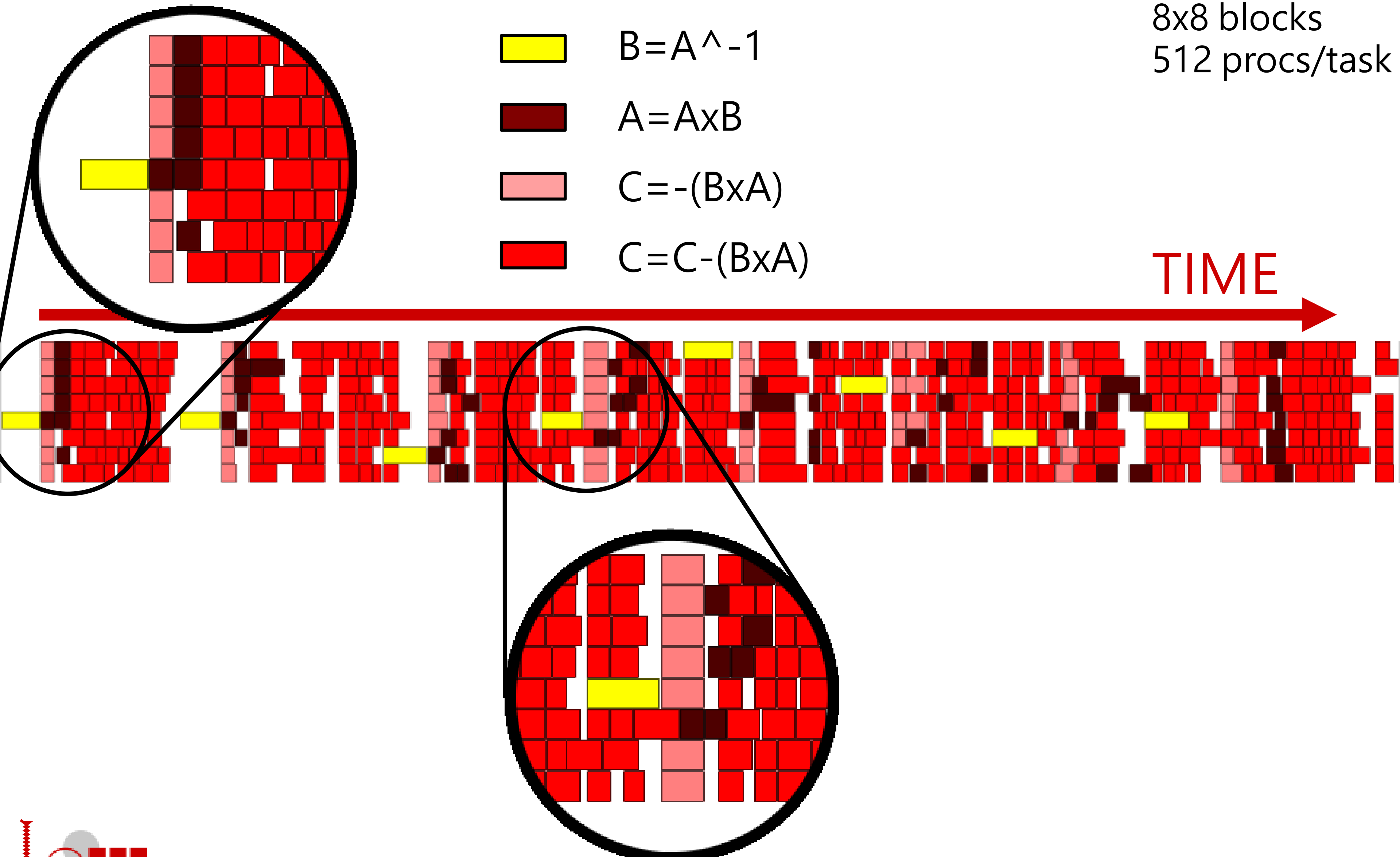
- 4096 processes for a workflow
  - 8~4096 processes for a task
  - (If 512 processes for a task, at most 8 tasks can be executed at the same time)

Experiment (1) Block Gauss Jordan on K



Experiment (1) Block Gauss Jordan on K

8x8 blocks
512 procs/task



Agenda

INTRODUCTION

Multi SPMD Programming model

Overview

Background

Experiments

Collaborations with

numerical library group

accelerator group

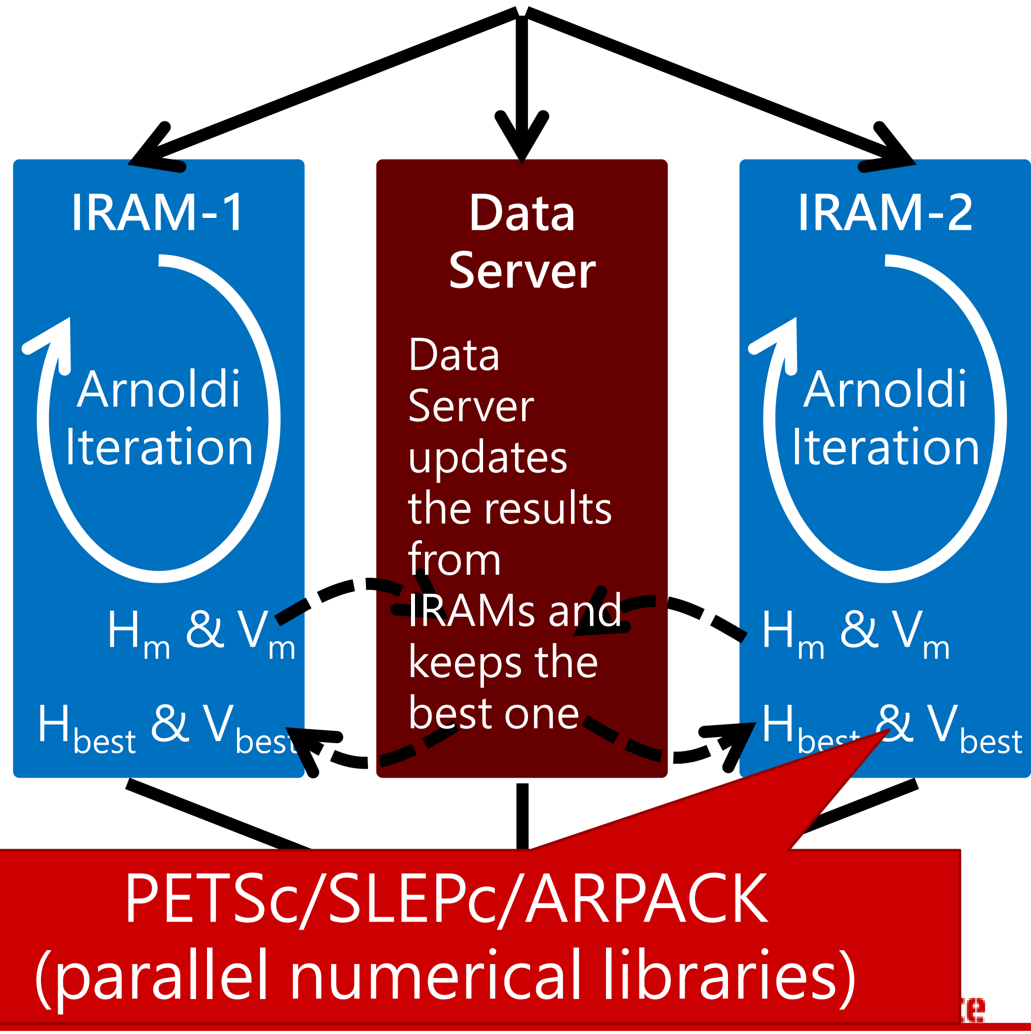
Fault Tolerance in the Multi SPMD

CONCLUSION

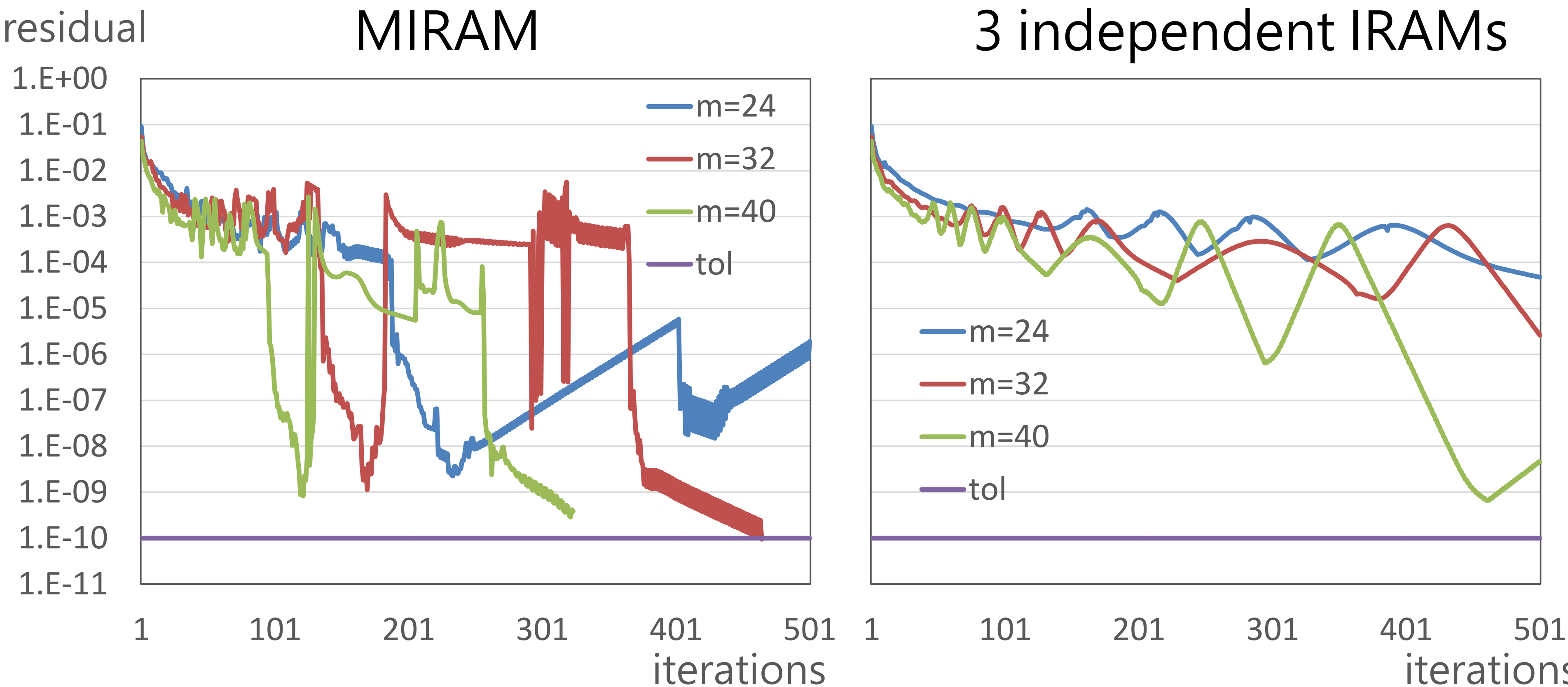
AICS

MIRAM Multiple Implicitly Restarted Arnodi Method

- IRAM (Implicitly Restarted Arnodi Method)
  - Iterative methods to obtain eigen pair of a matrix
- MIRAM
  - hybrid iterative method
  - invokes several IRAMs with different parameters
  - exchanges information between IRAMs to speedup convergence
- Schenk/nlpkkt240 (UF Sparse Matrix Collection) rows x cols 27,993,600^2 # of non-zeros 760,648,352
- **K-computer**



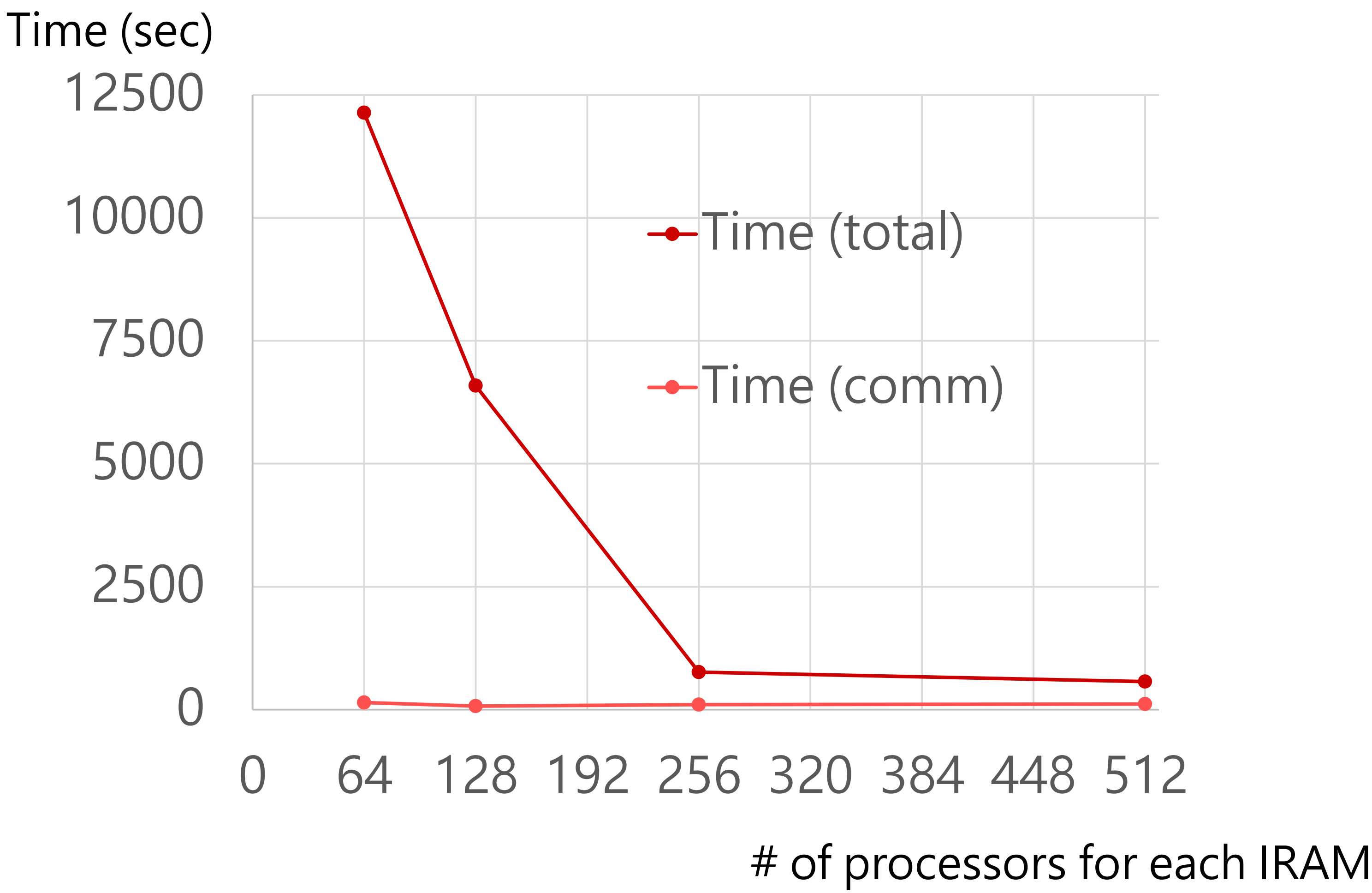
MIRAM: Speedup Convergence



We can reduce the number of iterations!

MIRAM: Speedup Execution Time

64, 128, 256, 512 cores for each IRAM on K-Computer



MIRAM Summary

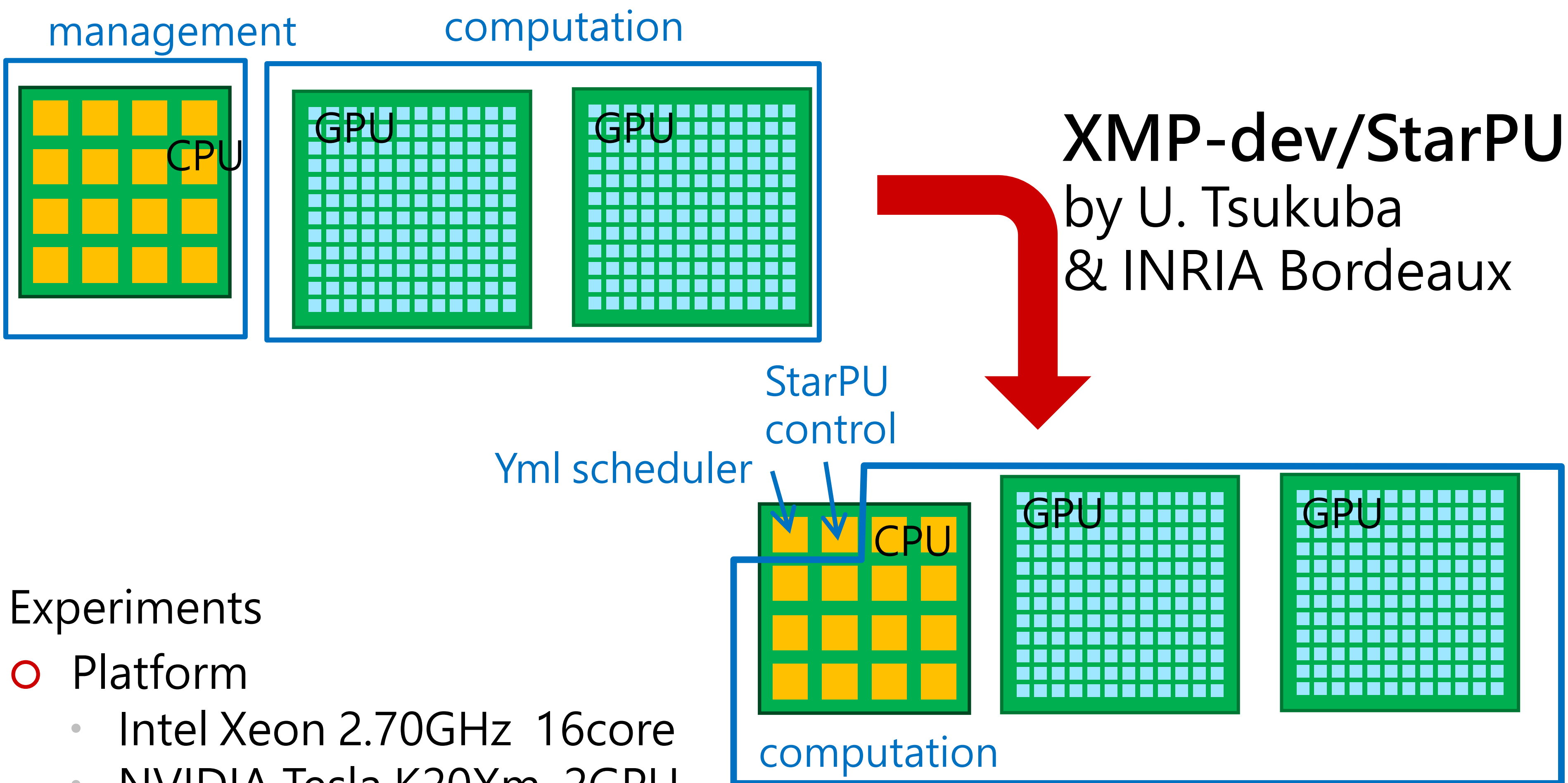
Two different speedups based on
Two different programming models

Workflow	• Reduce # of iterations until convergence
Distributed Parallel	• Reduce the Execution time for each iteration

XMP/StarPU

- Developed by Accelerator group (U. Tsukuba, INRIA Bordeaux)
  - StarPU
    - A Unified Runtime System for Heterogeneous Multicore Architectures
    - Task-sharing between CPU and GPU
  - XMP
    - extended to write such task-sharing based on StarPU
- YML/XMP/StarPU for heterogeneous systems
  - allows to write tasks with XMP/StarPU

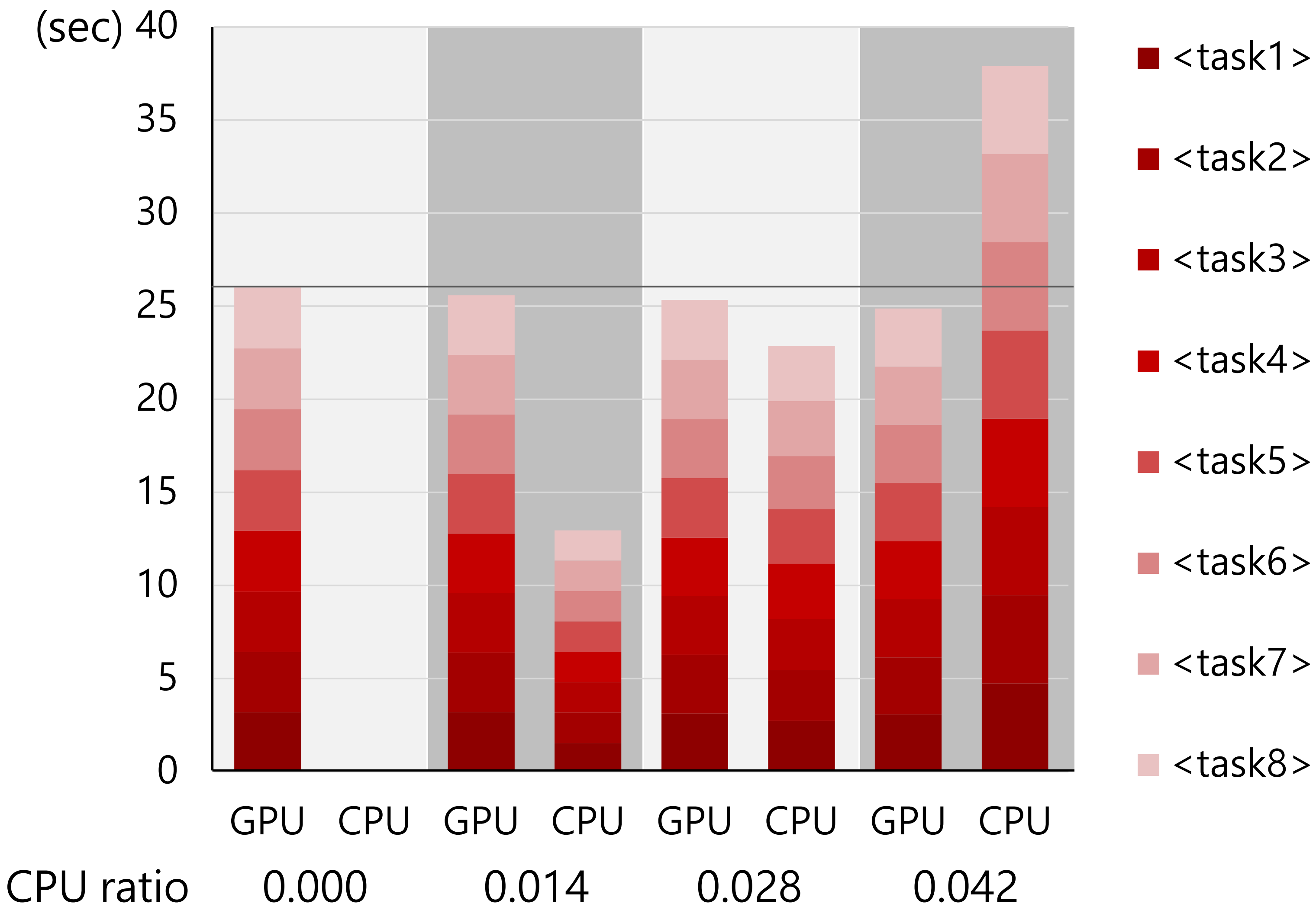
YML+XMP+StarPU



Experiments

- Platform
  - Intel Xeon 2.70GHz 16core
  - NVIDIA Tesla K20Xm 2GPU
- Block DGEMM (2x2 blocks)
  - 10000 x 10000 matrix (-> 5000x5000 block)

Experiments Block DGEMM with YML+XMP-dev+StarPU



Agenda

INTRODUCTION

Multi SPMD Programming model

Overview

Background

Experiments

Collaborations with

numerical library group

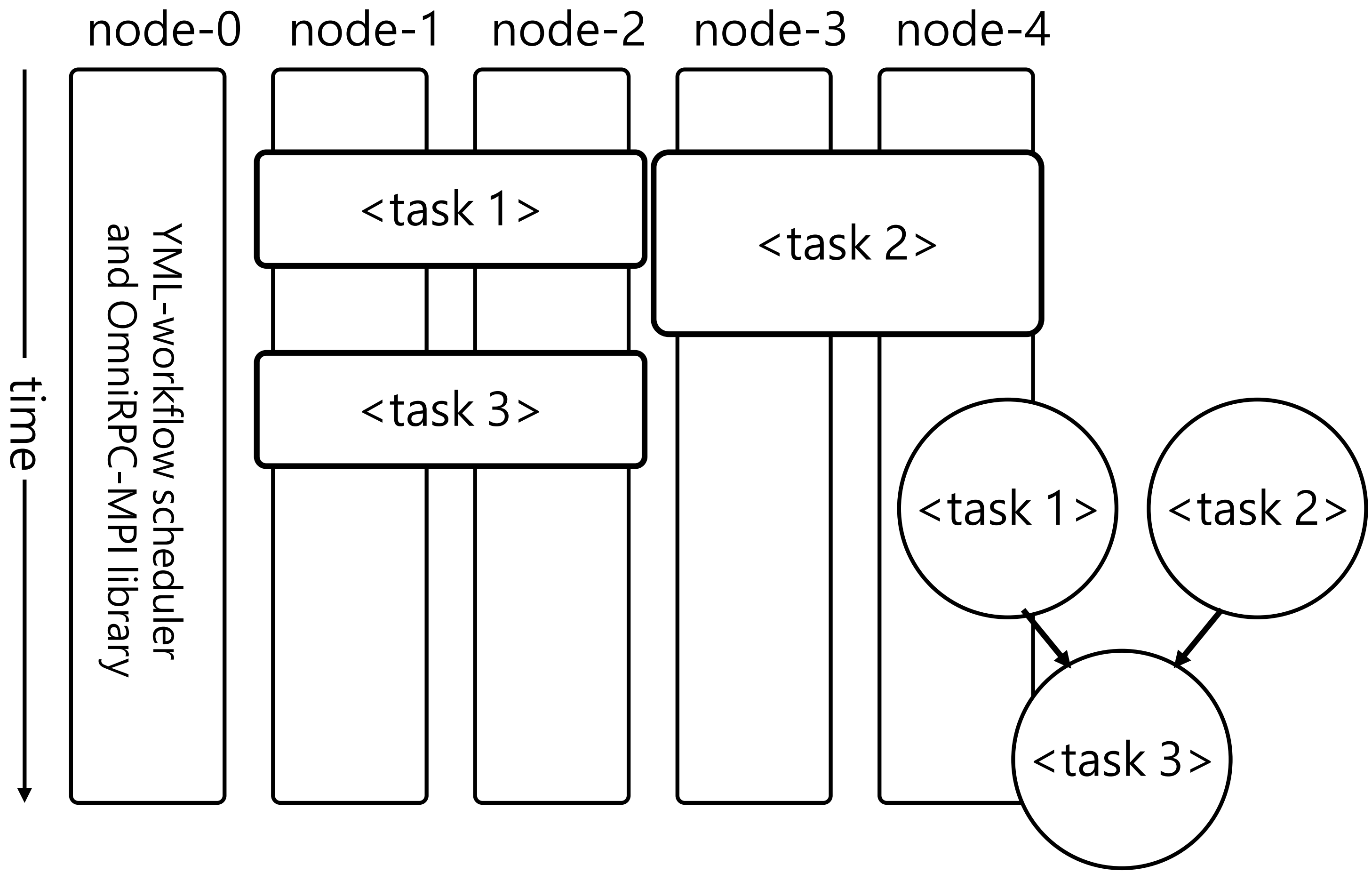
accelerator group

Fault Tolerance in the Multi SPMD

CONCLUSION

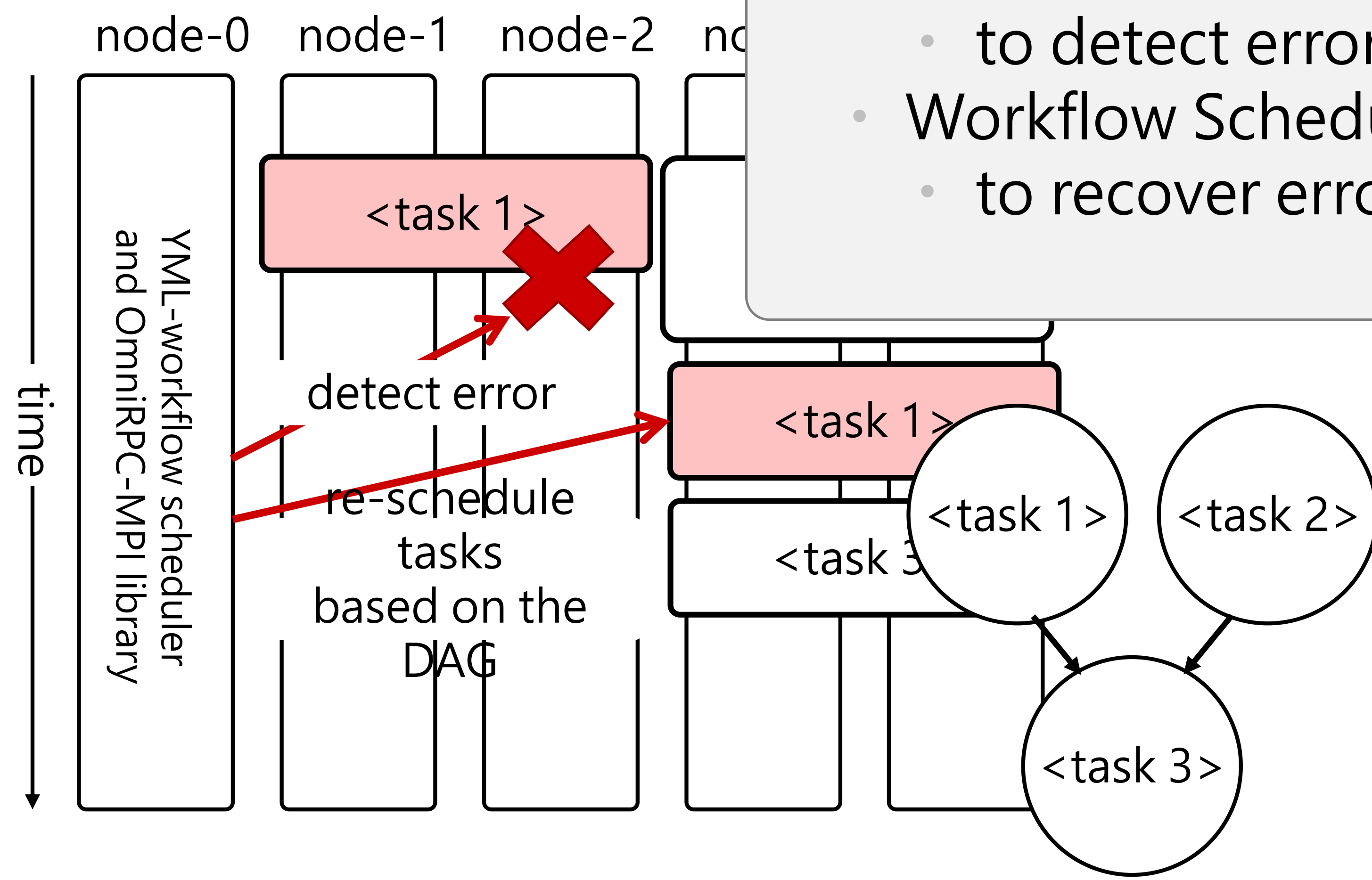
AICS

Fault Tolerance in YML/XMP



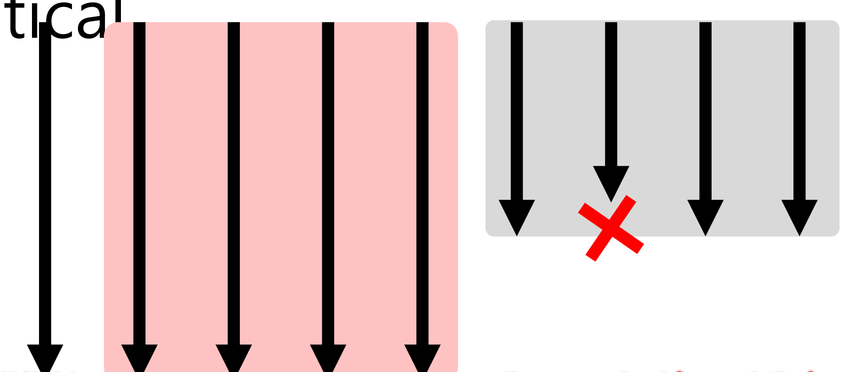
Fault Tolerance in YML/XMP

- We have extend
  - Middleware
    - to detect errors
  - Workflow Scheduler
    - to recover errors



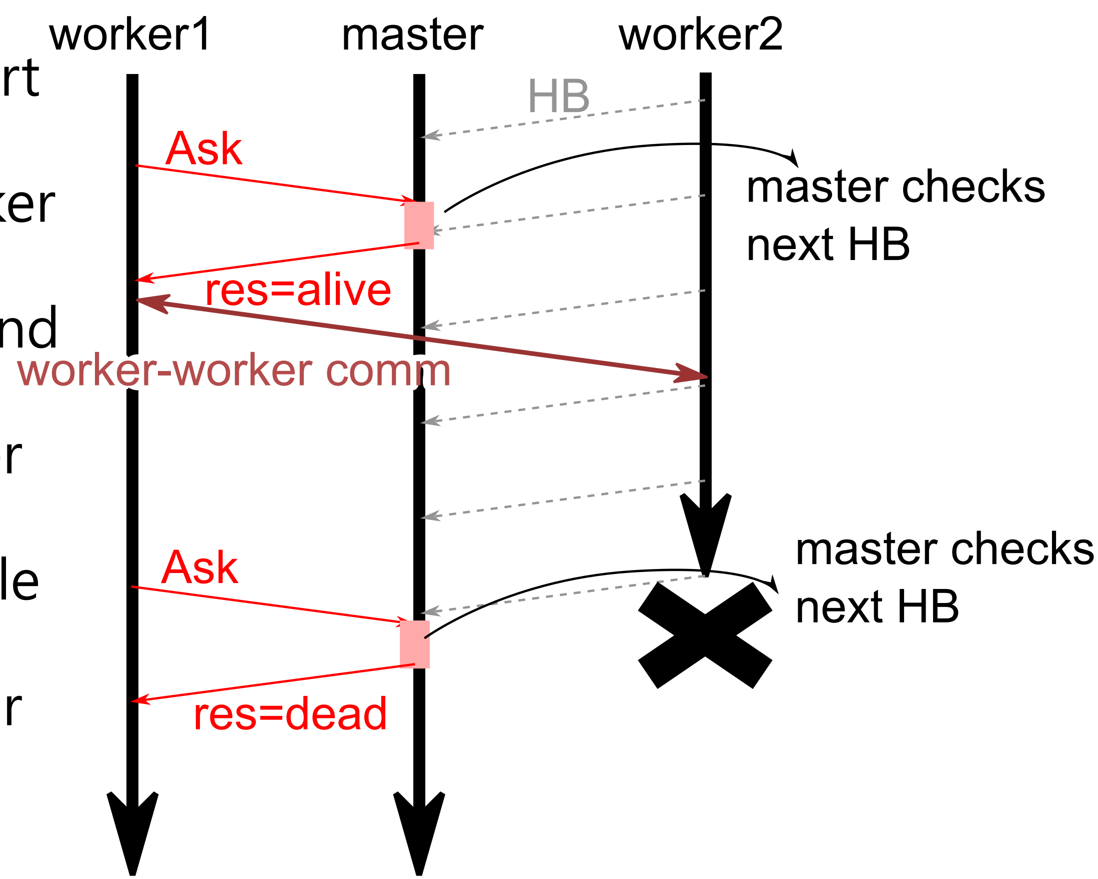
OmniRPC-MPI to OmniRPC-MPI-FT

- OmniRPC-MPI-FT
  - extension of OmniRPC-MPI to realize fault tolerance
- Assumption (a new job scheduler proposed [Mutai et al.2013])
  - there is an error in a node used by a worker program, all the other processes in the worker program are stopped. These processes are not available until the job is finished. On the other hand, the processes in other worker programs and master program can continue.
  - An error in a master is critical



OmniRPC-MPI to OmniRPC-MPI-FT

- Implementation
  - Error detection using Heart Beat (HB) messages
  - API to ask whether a worker is dead or not
    - OmniRpcMpiCheckHandle(void *hd);
      - master checks worker availability
    - OmniRpcMpiAskHandleAlive(int id);
      - worker checks worker availability



Workflow Scheduler

- YML workflow scheduler
  - sends requests to execute tasks to the middleware (OmniRPC-MPI library) based on the DAG of a workflow application
- YML workflow scheduler for FT
  - if an error is reported by the middleware, then remove it from the request-list and return main loop
  - The main loop executes the req again.

```
Yml::Core::SchedulerTask
*MpiBackend::retrieveImpl(void){
 for(i=0;i<NUMBER_REQUESTS;i++){
  if(OmniRpcProbe(req[i])==success){
   remove the req[i] from the request list
   return task[i];
  }else if(OmniRpcProbe(req[i])==fail){
   remove the req[i] from the request
list
   set the status of task[i] error
   return task[i];
  }else{
   // req[i] is in execution
   // retrieveImpl do nothing
  }
 }
 return 0;
}
```

Experiments --Environment

- The overhead of the fault detection
- The ability to find a failure and to recover from the failure
- The elapsed time when error(s) occur.

- 65 nodes
  - 1 node for YML workflow scheduler
  - 64 nodes (1024 processes) for worker-programs (tasks)

FX10 @ AICS	
CPU	FUJITSU SPARC64IXfx 16core 1.65 GHz
Memory	32GB/s, 85GB/s
Compiler	Fujitsu Compiler 1.2.1

Experiments -- Test Problem (Block-Gauss-Jordan)

- Compute an inversion of a matrix by inversions of a block of the matrix and the updates of other blocks based on the inversions.

- We can control the hierarchical parallelism levels easily by FP2C
  - Fix the matrix size (20480) total number of processes(1024)
  - Change the size of blocks and the number of processes for each task (block)

	64	256	512	1024
20480^2 (1^2 blocks)	-	-	-	
10240^2 (2^2 blocks)	-			
5120^2 (4^2 blocks)				
2560^2 (8^2 blocks)				

≒Simple SPMD

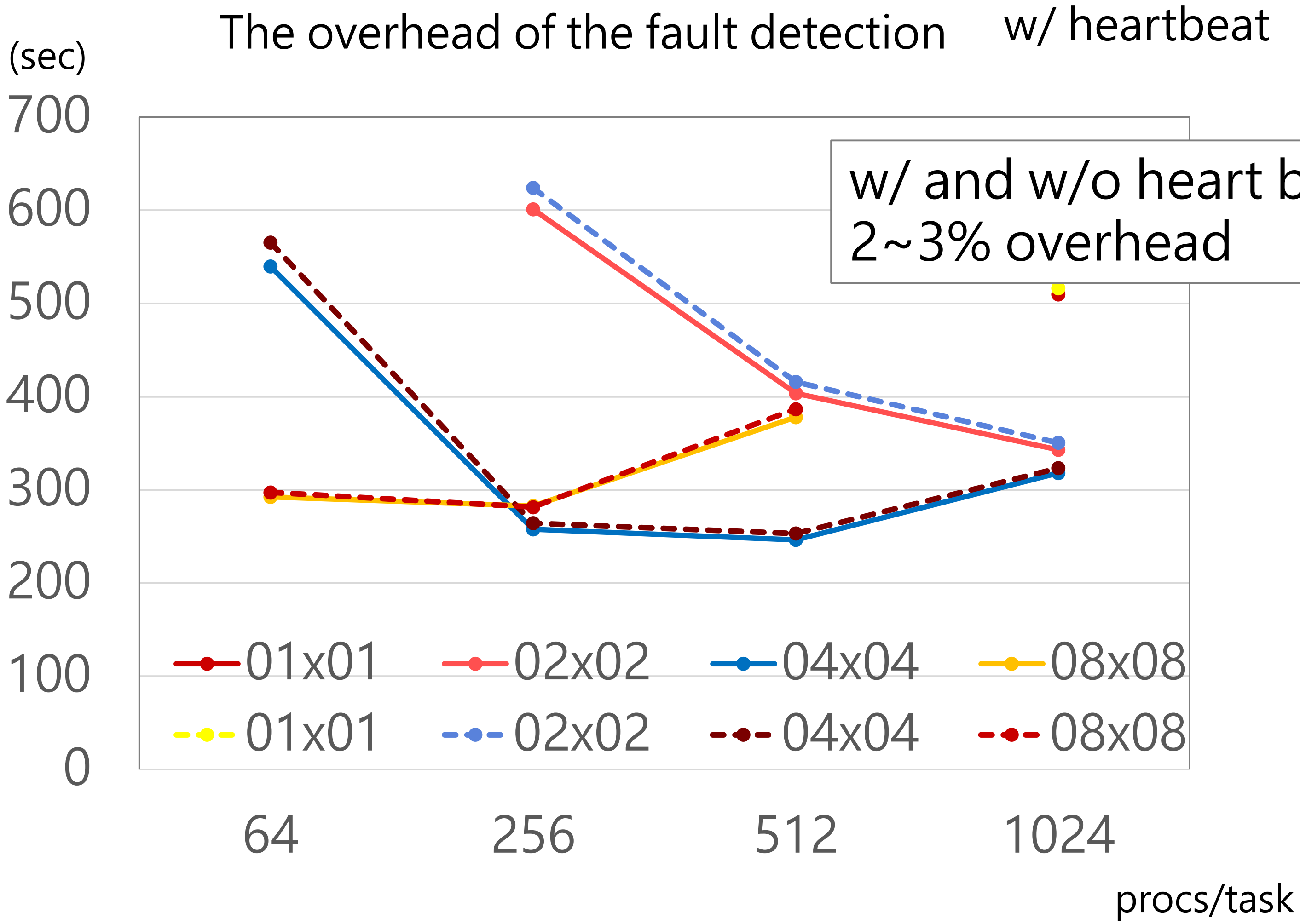
2 tasks can be executed simultaneously

≒ Simple workflow

4 tasks can be executed simultaneously

Experiments -- Results (w/o Error)

w/o heartbeat ¯ solid
w/ heartbeat - - dotted

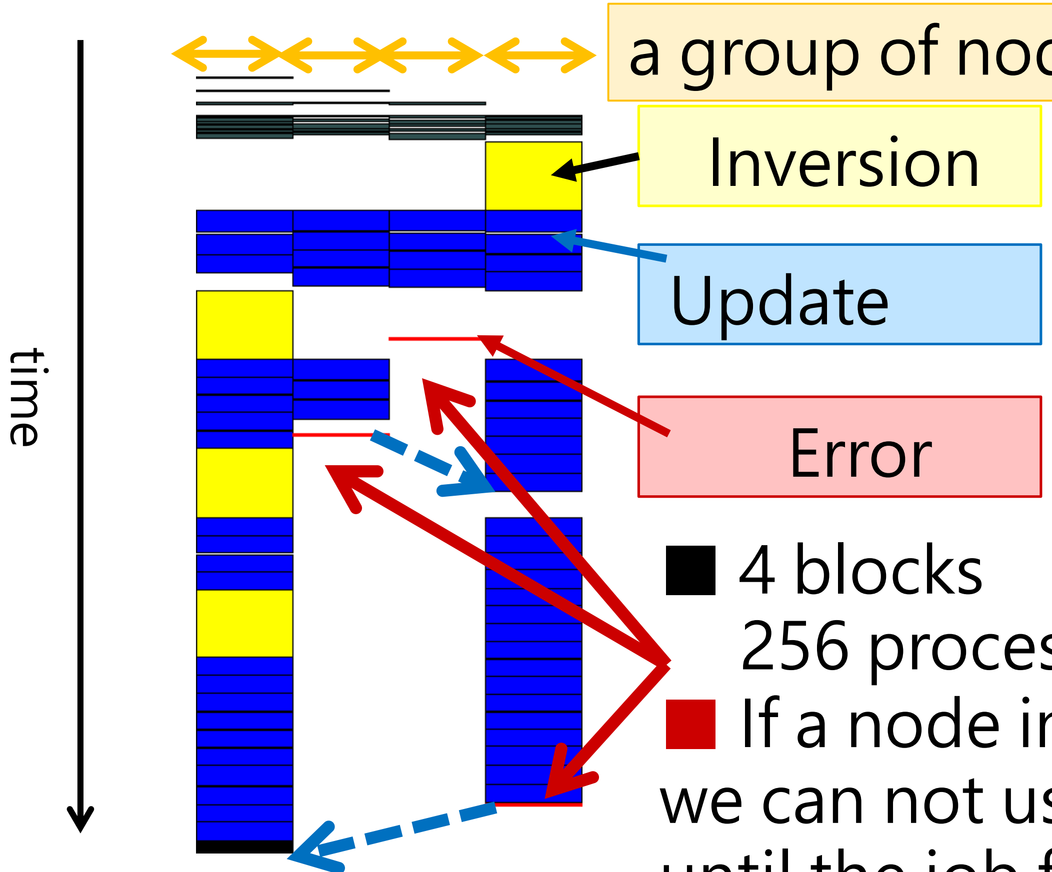


Experiments -- Error Scenarios

- The ability to find a failure an to recover from the failure

- Difficult to encounter a real error
- Stop a process in worker programs randomly based on several MTBFs
  - 12.5, 25, 50 hours
- 10 times for each of (MTBF, procs/task, # of blocks) combinations

Experience - Timeline (observed in an experiment)

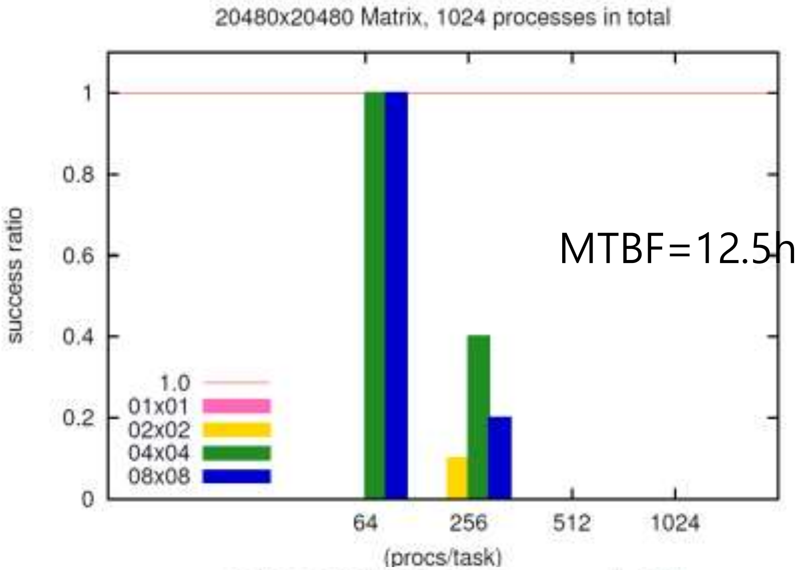


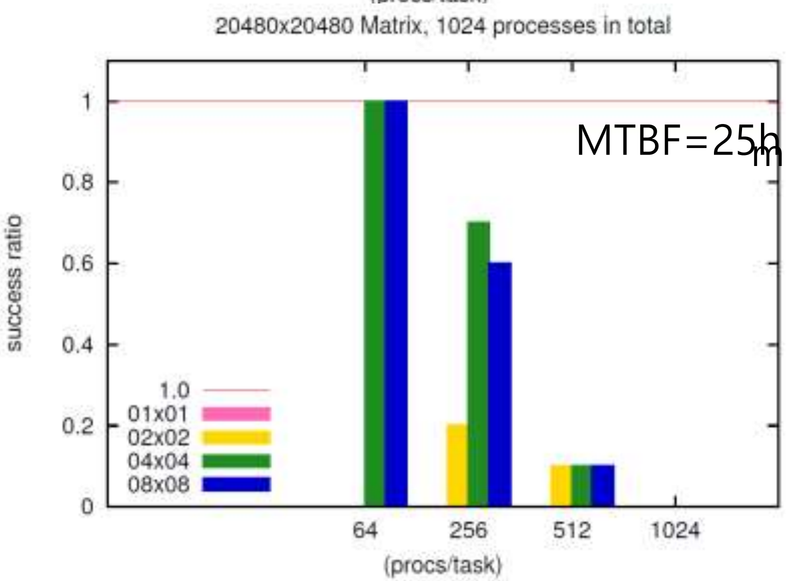
- 4 blocks
  256 processes for each task
- If a node in a group fails, we can not use the group until the job finishes

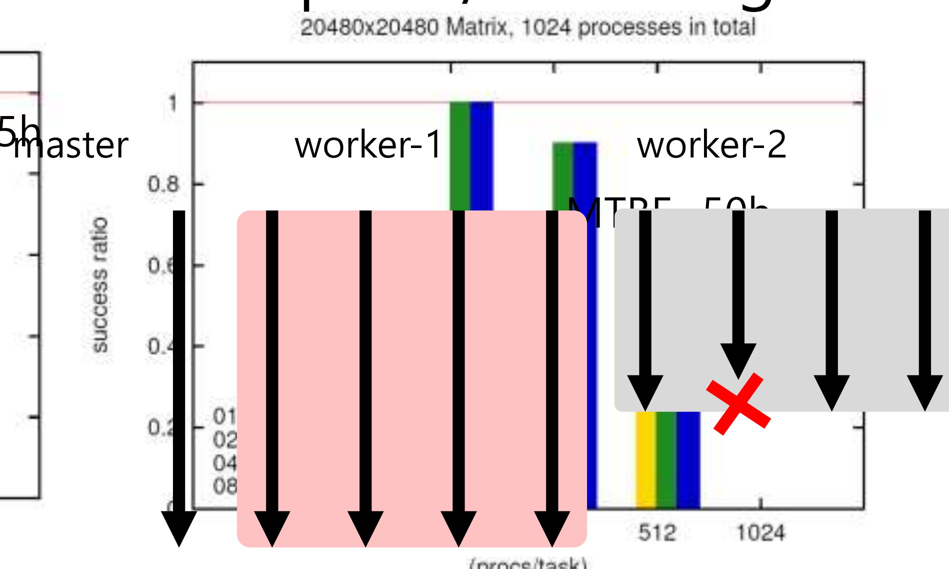
- **The tasks failed are re-executed on another group**

Experience -- Completion ratio for each MTBFs

- 1x1 block, 1024 procs/task (simple XMP programming model) always fails when there is an error
- (no room to re-schedule the task after error
- many small blocks and small # of procs/task are good

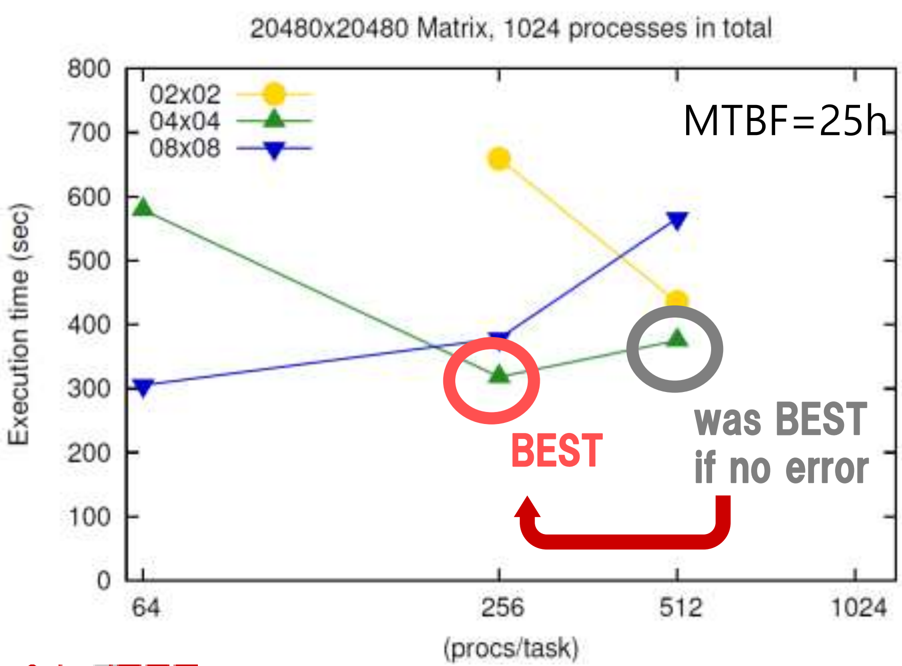


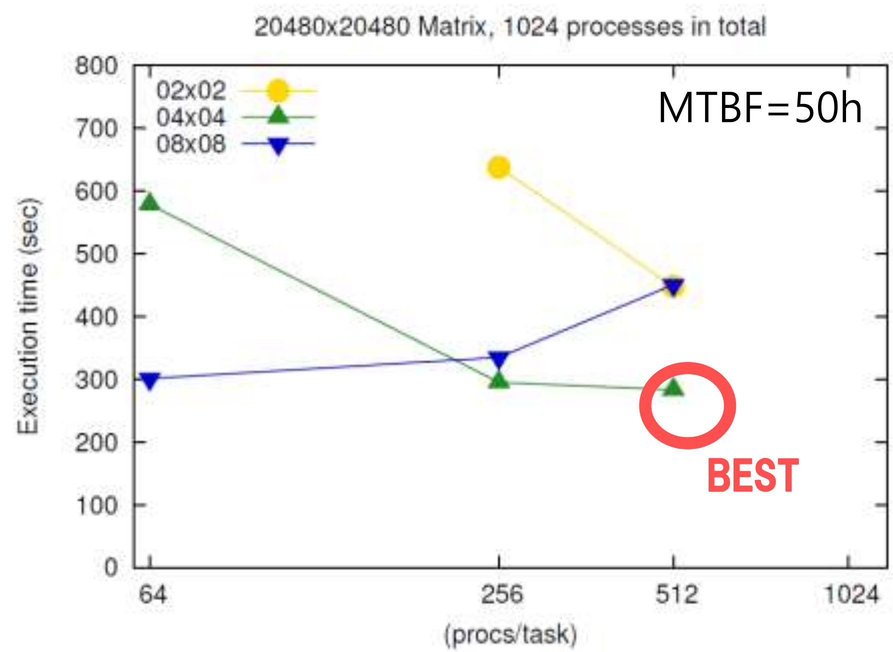




Experience -- Execution time ratio w/ error

- Execution time when there is **at least one error**
  - ignore the "lucky" case that an application is completed without any error
  - ignore the "unlucky" case that an application is note completed
- Execution time increases
  - 12% average, **3% min**, 19% max





Experience -- Summary

- The overhead to detect error (HB messages) is only 2~3%
- The overhead to detect an error(s) and complete application (even where there is an error(s) varies from 3-19%.
  - We can reduce it by controlling appropriate decomposition of computational resources for the multi SPMD programming model
  - The control is easy(!), if you use our programming tool
- We've find that the best combination of SPMD and workflow depends on MTBF
  - Again, we can control it easily by using our "multi-SPMD" programming model

Agenda

INTRODUCTION

Multi SPMD Programming model

Overview

Background

Experiments

Collaborations with

numerical library group

accelerator group

Fault Tolerance in the Multi SPMD

CONCLUSION

CONCLUSION

- FP3C
  - multi-SPMD programming model
  - multi-SPMD programming model + numerical algorithm
  - multi-SPMD programming model + XMP/StarPU
- After FP3C,
  - multi-SPMD programming model + fault tolerance
- Future work
  - collaboration with MDLS (MOU)
  - application side
    - TOTAL