FP3C

A hierarchical programming model for integration of parallel components with a scientific workflow toward and beyond petascale computing

Miwako TSUJI

Center for Computational Sciences

University of Tsukuba

FP3C project

- JST-ANR "Framework and Programming for Post Petascale Computing (FP3C)" project
    - a collaborative project of French and Japanese researches
    - 4 Japanese Universities (Univ. of Tsukuba, etc..)
    - 6 French Universities & Research Centers
  - We investigates 4 main tasks
    - **Programming models**
    - Accelerating technology
    - Runtime technology
    - Parallel algorithm and library

  we are now starting to combine these tasks

AGENDA

- Introduction
- Background
- FP2C
  - New programming model
  - Framework for Post-Petascale Computing
- Experiments
- Conclusion

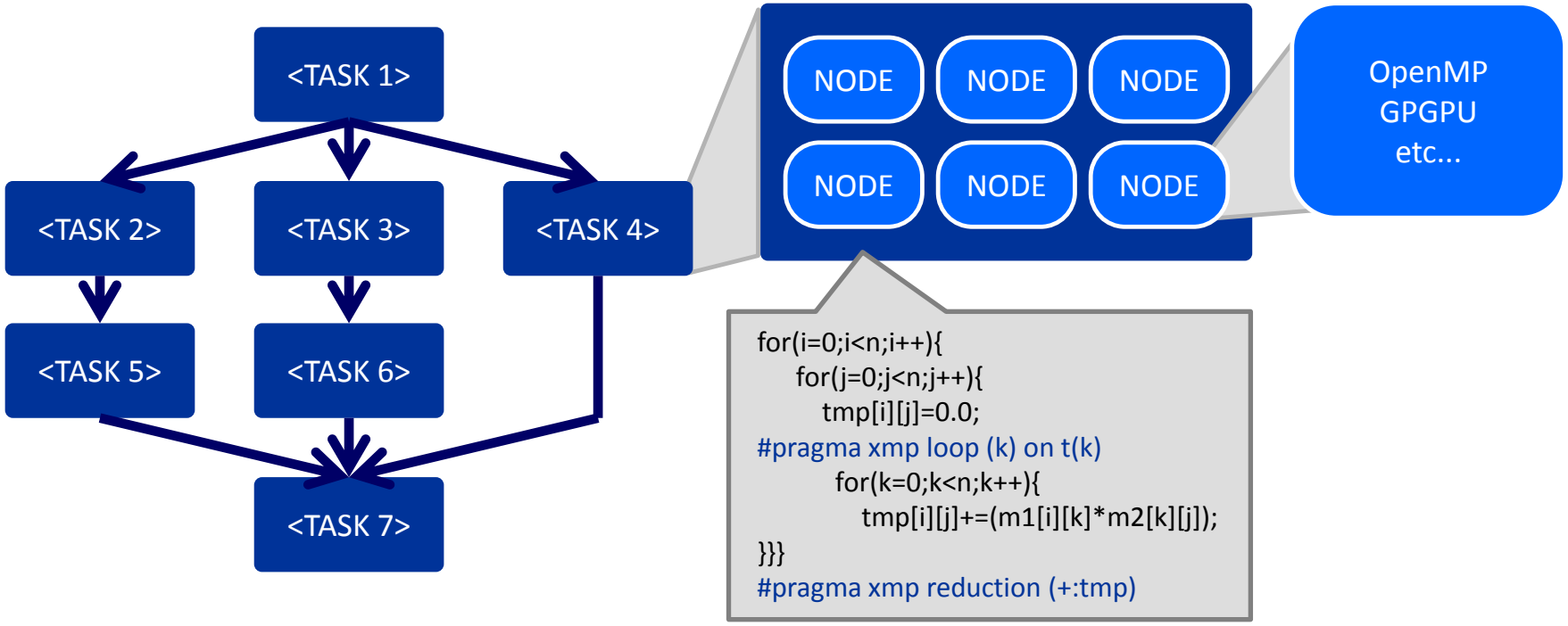
Introduction

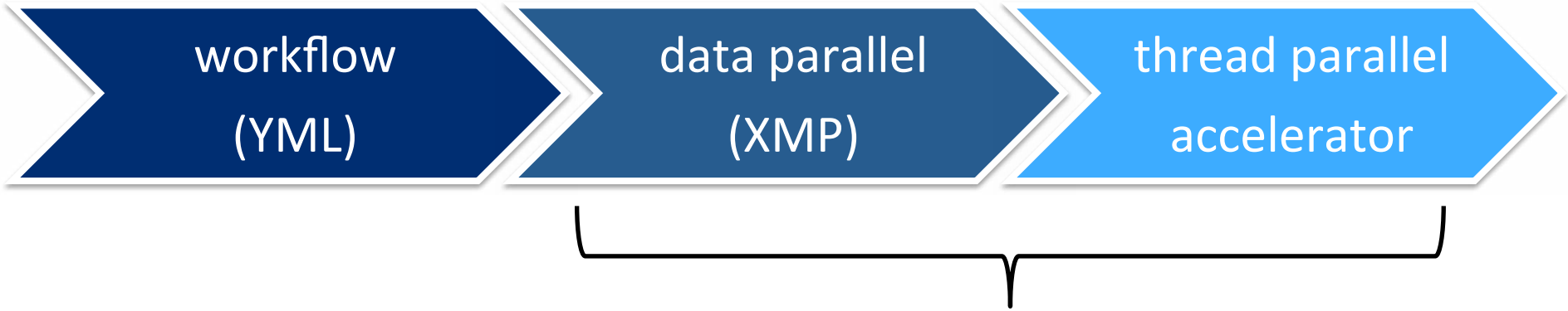
- Post-Petascale and Exascale supercomputers
- The future systems would be
  - large cluster (“K” has 864 racks), cluster of clusters
  - Network topology that creates “gang” of nodes (multi-dimensional Torus)
  - multi/many-core processors
  - accelerators
    - GPGPU, Cell, FPGA, etc..
  - Manage MPI process should be hard, even if we adopt MPI +OpenMP hybrid

Architecture of post-peta and exascale machines will be organized in a hardware hierarchy
New programming model for the huge & hierarchical architectures

FP3C

FP2C Framework for Post-Petascale Computing





FP3C

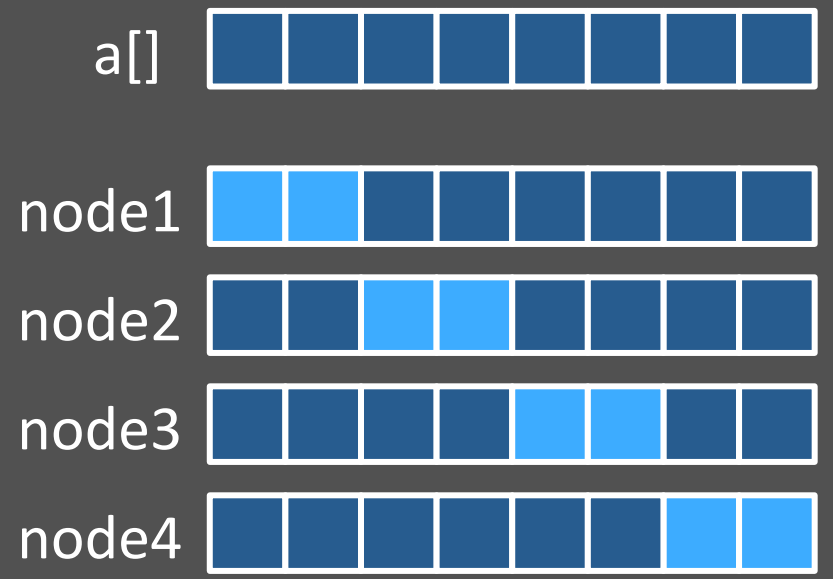
Background XcalableMP (XMP) *http://www.xcalablemp.org/*

- Directive-based language extension for scalable and performance-aware parallel programming
- It will provide a base parallel programming model and a compiler infrastructure to extend the base languages by directives.
- XMP-C source code

  → C 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

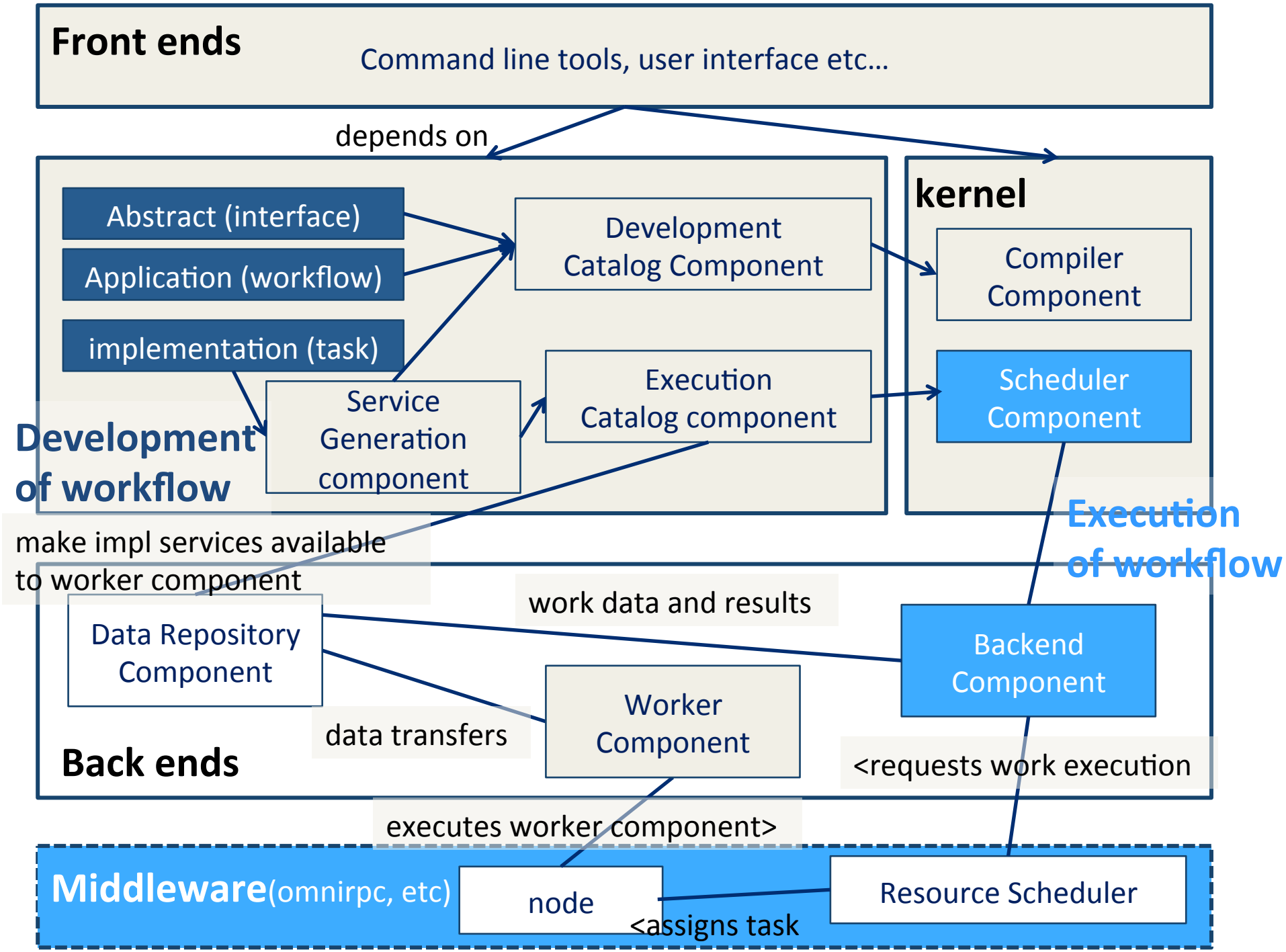
YML *http:// yml.prism.uvsq.fr/*

- A workflow programming environment
- P)
  - face
  - Application
    - High level graph description language called YvetteML can be used to describe workflow

```
<?xml version="1.0" ?>
<application name="test1" >
    <description>
        First tutorial example application
    </description>
    <params>
    </params>
    <graph>
        compute test(result, 1.0, 2.0);        # result contains 3.0
        compute test(result, result, result);  # result contains 6.0

    </graph>
</application>
```

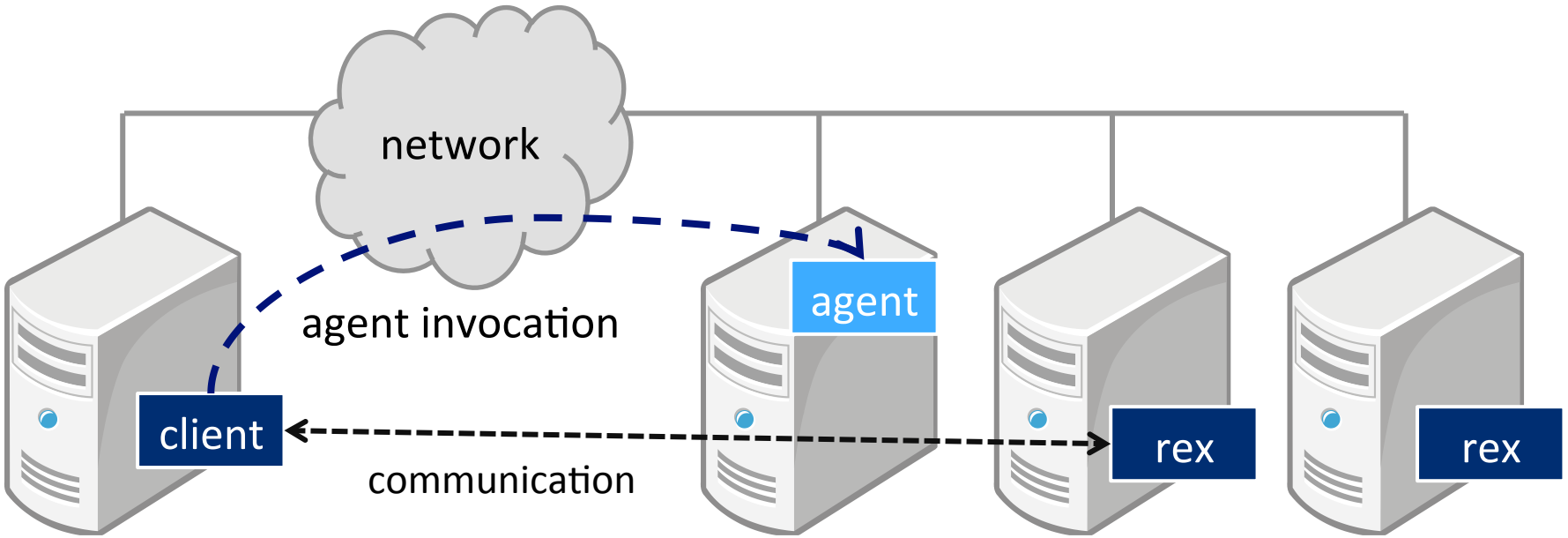
FP3C


Front ends
Command line tools, user interface etc…
depends on
Abstract (interface)
Application (workflow)
implementation (task)
Development
Catalog Component
kernel
Compiler
Component
Execution
Catalog component
Scheduler
Component
Service
Generation
component
Development
of workflow
Execution
of workflow
make impl services available
to worker component
work data and results
Data Repository
Component
Backend
Component
Worker
Component
data transfers
Back ends
<requests work execution
executes worker component>
Middleware(omnirpc, etc)
node
Resource Scheduler
<assigns task

Background

OmniRPC *http://www.omni.hpcc.jp/OmniRPC/*

- One of Middlewares of YML
- GridRPC (Remote Procedure Call)
- master-worker parallel program is supported
- remote programs (rex) executed by exec, rsh and ssh

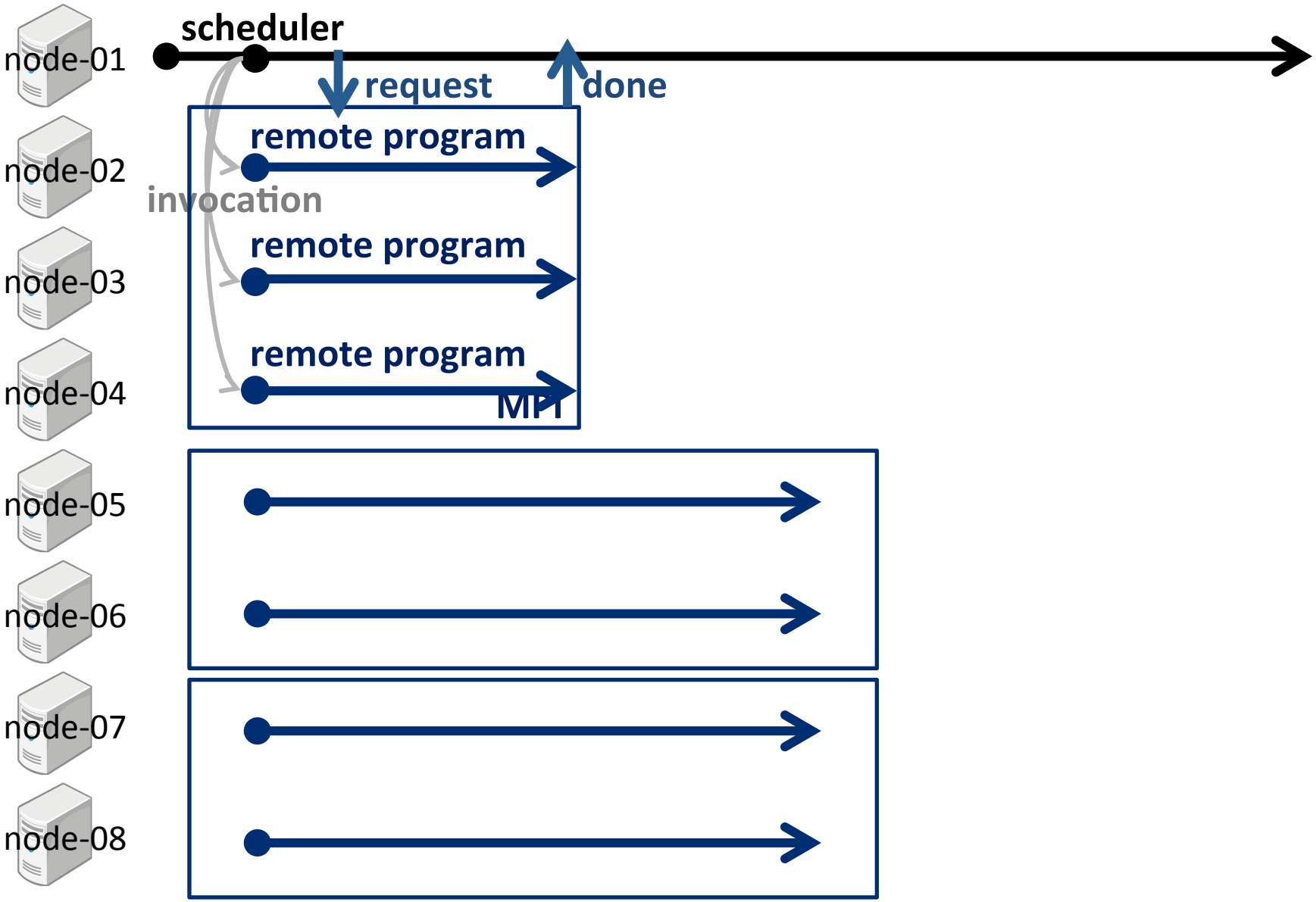


FP2C

- A new programming model "Framework for Post-Petascale Computing (FP2C)" has been proposed
- Integrate parallel components written in XMP into a YML-workflow
- Post peta architecture may be hierarchical one & Network topology may create "gang" of nodes
  - YML
    - works at top level of the hierarchical architecture
    - manages components among (loosely) connected network
  - XMP
    - works at middle (data parallel) and bottom (accelerator, thread)
    - hybrid (data-thread parallel) programming among nodes densely connected
- Provides different levels and strategies of hierarchical programming for several applications and architectures

FP3C

Overview of FP2C



FP3C

How to realize FP2C?

- OmniRPC extension for MPI
  - library linked with yml_scheduler
  - invoke remote programs, send requests to them
  - resource management
- MPIBackend
  - to use OmniRPC-MPI
- yml_compiler & yml_scheduler extension
  - # of procs is taken into account
- Implementation component extension
  - data mapping information of XMP
- Implementation component (task) generator extension
- Distributed data exchange inside a parallel component support
  - NFS, direct data exchange between tasks [todo]

OmniRPC extension & MPI-Backend

- Parallel remote programs are executed by MPI_Comm_spawn provided by MPI-2
- New API is developed to use the function, MPI_Comm_spawn
  - arguments
    - the name of stub
    - the number of nodes
- Requests & signals are exchanged via MPI
  - the original OmniRPC used TCP sockets
  - portability
- OmniRPC is incorporated into yml_scheduler as a library
- 2 pthreads
  - main: invoke remote programs, send requests to them
  - sub : listen & receive end signal of a tasks from remote programs, post-processing of the tasks
  - both : resource management of remote nodes

FP3C

OmniRPC extension & MPI-Backend (cont.)

- Resource management
  - The number of nodes are taken into account in addition to the number of components.
  - If the number of empty nodes is not enough, idle remote programs are killed until

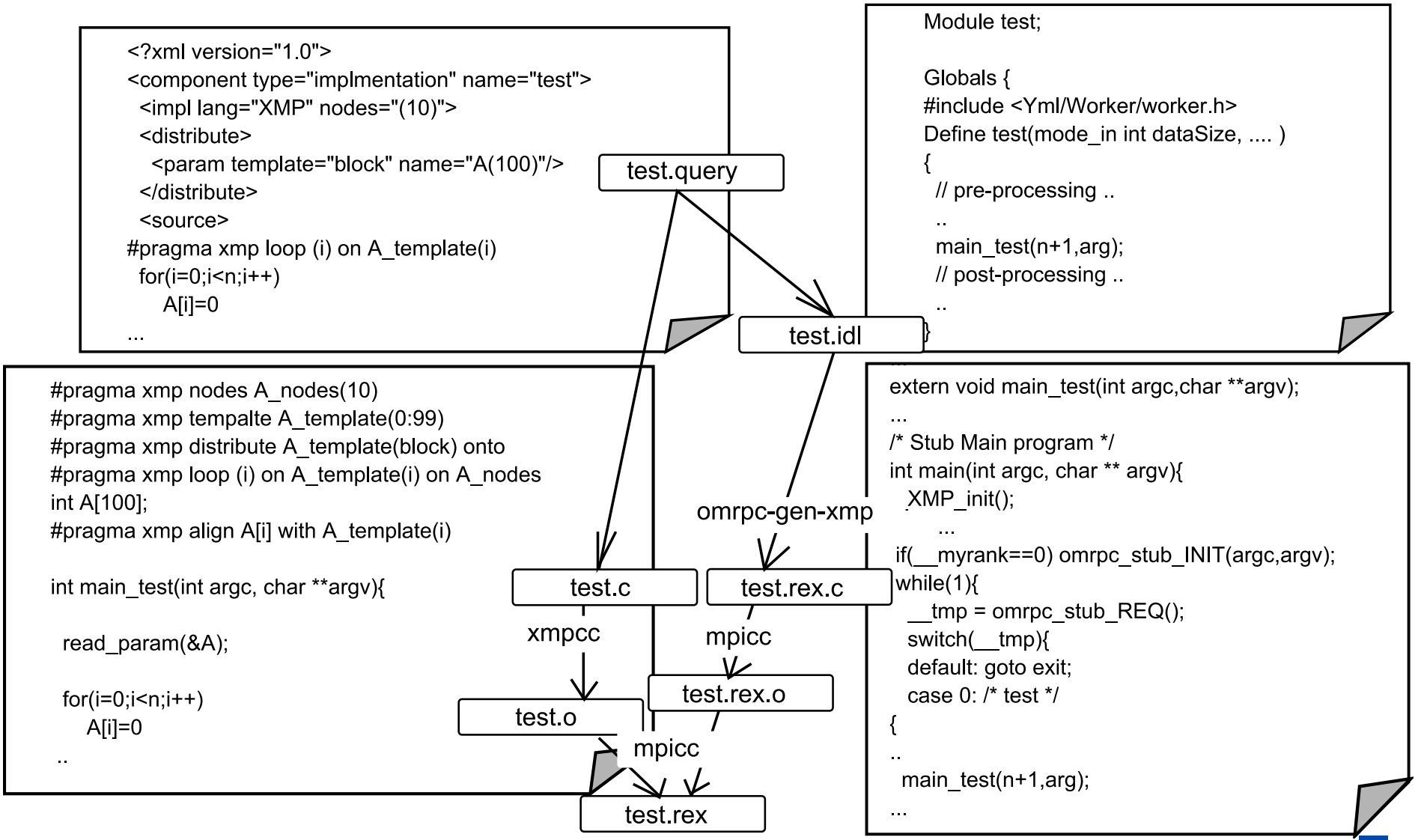
    the # of available nodes >= the # of required nodes
  - A remote program is reused if and only if it contains a required implementation component and the number of nodes are equal to the required one

Implementation component extension

```
<?xml version="1.0"?>
<component type="impl" name=“sample” abstract=“sample">
<impl lang="XMP" nodes="CPU:(2,2)">
<distribute>
<param template="block,block" name="A(512,512)" align="[i][j]:(j,i)"/>
</distribute>
<source>
  int i,j;
#pragma xmp loop (i,j) on A_template(j,i)
  for(i=0;i<512;i++){
    for(j=0;j<512;j++){
      A[i][j]=0.0;
    }}
</source>
<footer />
</impl>
</component>
```

FP3C

Implementation component generator extension



FP3C

yml_compiler & yml_scheduler extension

- yml_compiler translate YvetteML description of an application into an oriented graph of a workflow
- In the translation, the extended yml_compiler incorporates information of required computational resouces (# of nodes) for each implementation component
- yml_scheduler uses the information

Distributed data exchange support

```
Worker_import_data()
{
    MPI_File_open(MPI_COMM_WORLD, VarName,MPI_MODE_RDONLY, MPI_INFO_NULL, descriptor, status);
    MPI_File_seek(descriptor,VarSize/NbProc*Rank, MPI_SEEK_SET, status); //Block distribution
    MPI_File_read(descriptor,VarSize/NbProc, code, status);
    MPI_File_close(descriptor,status);
}

/* XMP Computations */

Worker_export_data()
{
    MPI_File_open(MPI_COMM_WORLD, VarName,MPI_MODE_RDONLY, MPI_INFO_NULL, descriptor, status);
    MPI_File_seek(descriptor,VarSize/NbProc*Rank, MPI_SEEK_SET, status); //Block distribution
    MPI_File_write(descriptor,VarSize/NbProc, code, status);
    MPI_File_close(descriptor,status);
}
```

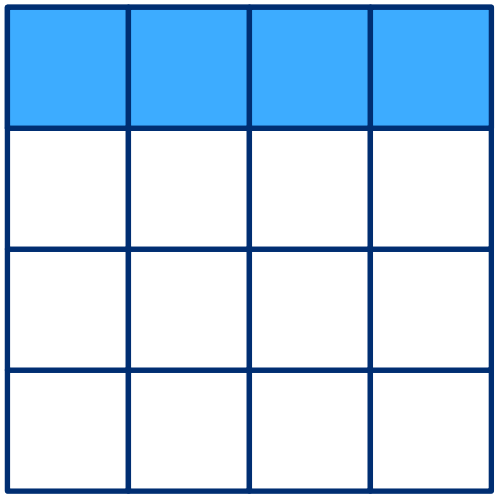
- Generated at compile-time with the parameter distribution information
- Data are read/write by each process
- Collective I/O

FP3C

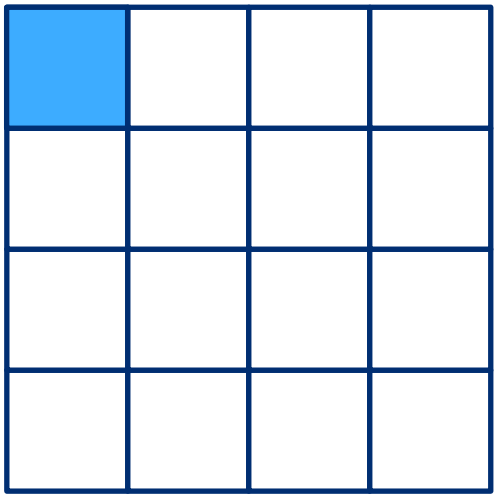
Experiments : Block Gauss-Jordan

```
Input: A (partitioned into p × p blocks)
Output: B = A^-1
  For k = 0 to p − 1
    B_kk = A^-1_kk
    For i = k + 1 to p − 1 (1)
      A_ki = B_kk × A_ki
    End For
    For i = 0 to p − 1 (2)
      If (i ≠ k)
        B_ik = −A_ik × B_kk
      End If
      If (i < k)
        B_ki = B_kk × B_ki
      End If
    End For
    For i = 0 to p − 1 (3)
      If (i ≠ k)
        For j = k + 1 to p − 1
          A_ij = A_ij − A_ik × A_kj
        End For
        For j = 0 to k − 1
          B_ij = B_ij − A_ik × B_kj
        End For
      End If
    End For
  End For
```

A matrix is divided into some blocks.



matrix A (input)



matrix B (output) = A^{-1}

$A_{ki} = B_{kk} \hat{a} A_{ki}$

compute prodMat(A[k][i], B[k][k])

FP3C

Experiments : Block Gauss Jordan

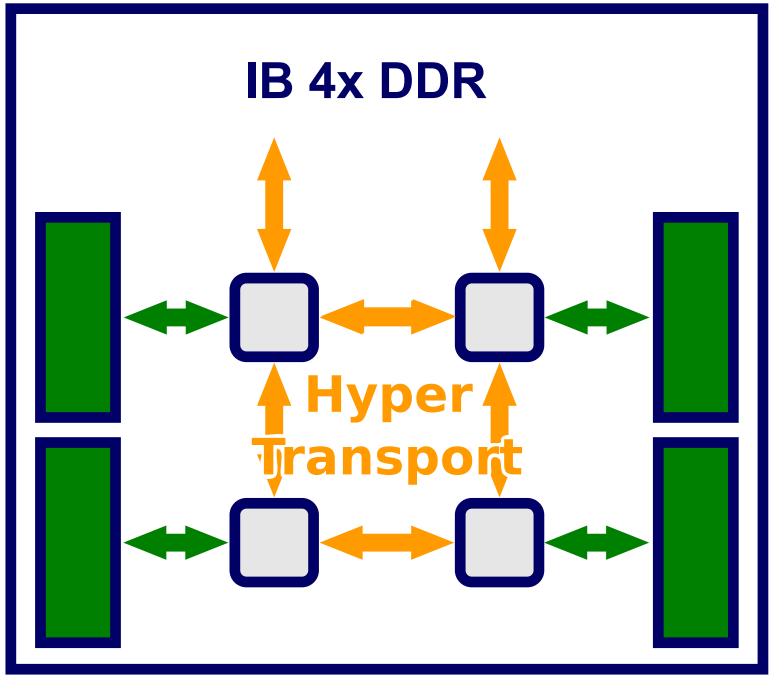
application component

```
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
```

FP3C

Experiments : T2K-Tsukuba



AMD Opteron
8350 Barcelona
Quad-core
2.3GHz

8GB
DDR2-667 memory

Node:

Opteron Barcelona B8000 CPU
2.3GBHz x 4FLOP/c x 4core x 4socket
= 147.2 GFLOPS/node
8x4=32GB memory/node

16cores in a node.
Flat MPI

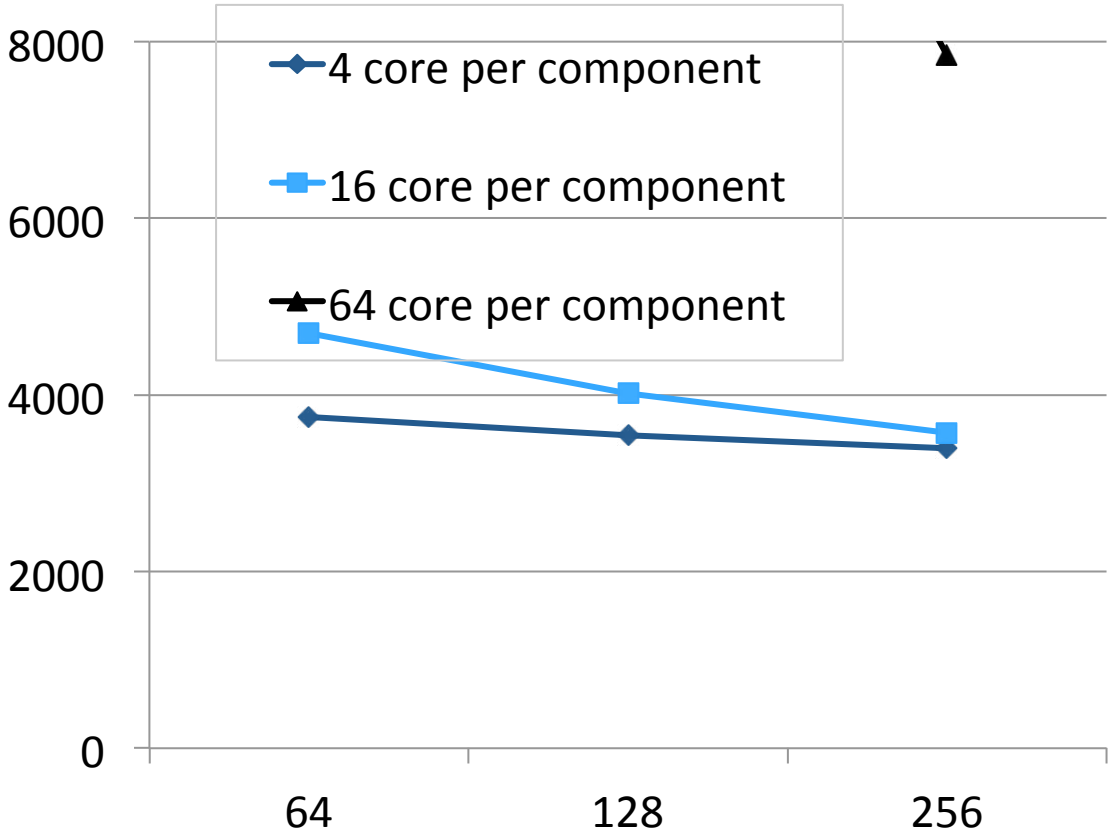
System:

648 nodes
95.3 TFLOPS/system
20.8 TB memory/system
800 TB Raid-6 Luster cluster file system
Fat-tree
full-bisection interconnection
Quad-rail of InfiniBand

Experiments

- 16,384 x 16, 384 matrix
  - 16x16 blocks (each block size is 1024x1024)
  - 32x32 blocks (each block size is 512x512)
- 64, 128, 256, cores in total used in a workflow
  - for each component, 4, 16 and 64 cores
    - in these experiments, each component is flat-MPI

Result: 32x32 blocks (512x512 each)



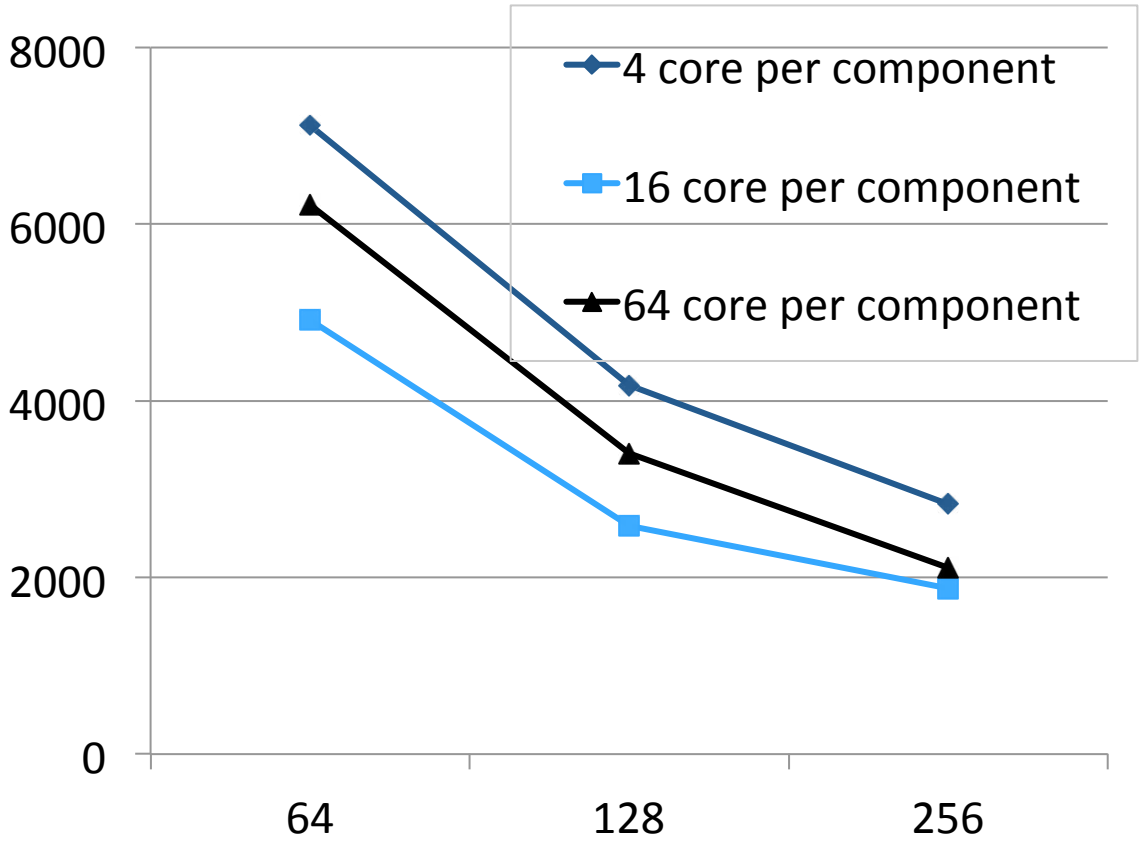
bad scalability

30,000 component calls

FP3C

Result: 16x16 blocks (1024x1024 each)

16 cores per each component is best



64 cores per component
too much, inter-node communication

4 cores per component
cache?
L2 0.5M/core
L3 2MB/socket

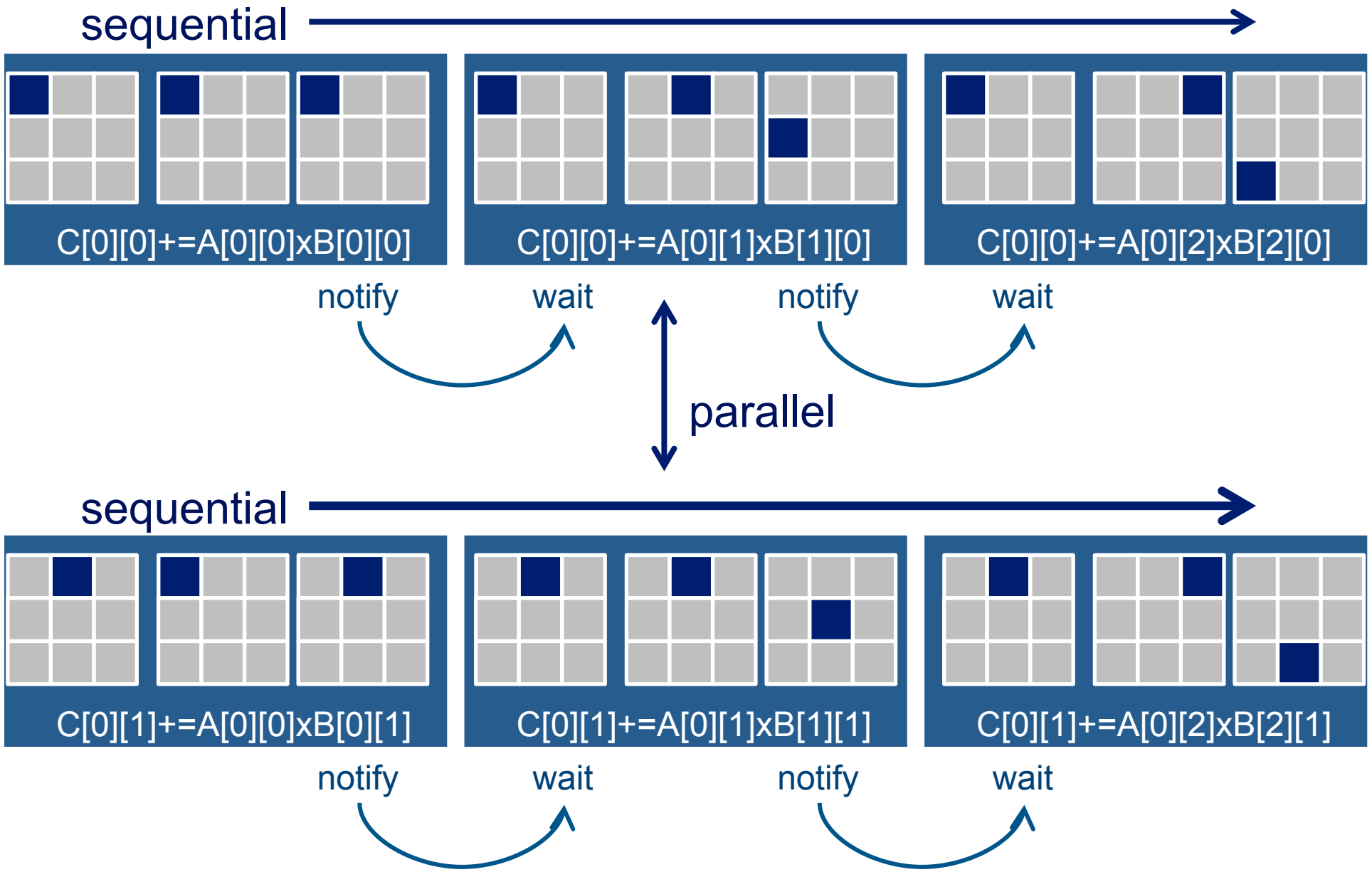
tasks < available resources

FP3C

DEMO

- block DGEMM

Block-DGEMM (YML)



Block-DGEMM (YML)

```
par(k:=0;blockcount-1) do
    par(i:=0;blockcount-1) do
        par(j:=0;blockcount-1) do
          if(k gt 0) then
            wait(step[i][j][k-1]);
          endif
          compute dgemm(A[i][k],B[k][j],C[i][j]);
          notify(step[i][j][k]);
        enddo
    enddo
enddo
```

Block-DGEMM (XMP)

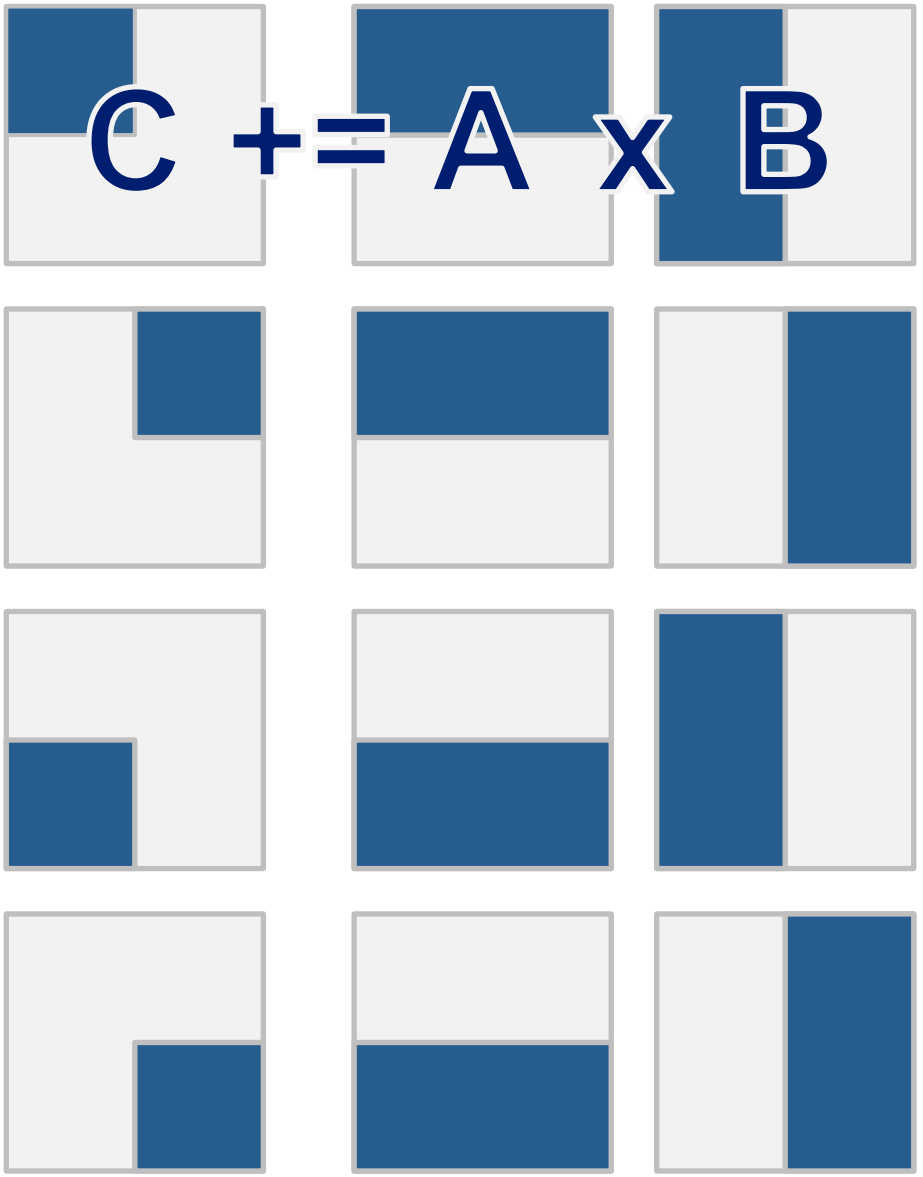
```
#pragma xmp align A[i][j] with t(j,i)
#pragma xmp align B[i][j] with t(j,i)

#pragma xmp shadow A[0:0][*]
#pragma xmp shadow B[*][0:0]

...

#pragma xmp reflect  A,B

#pragma xmp loop (i,j) on t(j,i)
  for(i=0;i<n;i++){
    for(j=0;j<n;j++){
      dtmp=0.0;
      for(k=0;k<n;k++){
        dtmp+=A[i][k]*B[k][j];
      }
      C[i][j]+=dtmp;
    }
  }
```



Conclusion

- YML and XMP has been integrated to create FP2C:
  - Parallel components are made easily by XMP
  - YML manages the parallel components
  - *http:// yml.prism.uvsq.fr*
- Implementation
  - OmniRPC extension to support MPI and XMP
  - YML implementation component extension for distributed arrays
  - XMP component generator
  - Data exchange through shared file system based on data distribution

Future work

- Direct data exchange between components
- Accelerating technology
  - *XMP*-dev (*StarPU*)
- Parallel library
  - YML supports libraries (BLAS, user-defined, etc..)
  - XMP provides interface of parallel library