A programming paradigm for extreme computational and data science

Serge G. Petiton

serge.petiton@univ-lille1.fr

Outline

- Introduction
- YML for computational science applications
- TEZ and others tools for data science computation
- YML for computational and data science distributed and parallel computing
- Conclusion

Outline

- **Introduction**
- YML for computational science applications
- TEZ and others tools for data science computation
- YML for computational and data science distributed and parallel computing
- Conclusion

Toward graph of parallel tasks/components

- Communications have to be minimized : but all communications have not the same costs, in term or energy and time.
- Latencies between farther cores will be very time consuming : global reduction or other synchronized global operations will be really a bottleneck.
- We have to avoid large inner products, global synchronizations, and others operations involving communications along all the cores. Large granularity parallelism is required (cf. CA technics and Hybrid methods).
- **Graph or tasks/components programming allows to limit these communications only between the allocated cores to a given task/components.**
- Communications between these tasks and the I/O may be optimized using efficient scheduling and orchestration strategies(asynchronous I/O and others)
- **Distributed computing meet parallel computing**, as the future super(hyper)computers become very hierarchical and as the communications become more and more important. Scheduling strategies would have to be developed.

Toward graph of tasks/components computing and other computing levels

- Each task/component may be an existing method/software developed for a large part of the cores, but not all of them (then classical or CA methods may be eficients)
- The computation on each core may use multithread optimizations and runtime libraries
- Accelerator programming may be optimize also at this level.
- Then we have the following levels of programming and computing :
  - Graph of components, already developed or new ones,
  - Each component is run on a large part of the computer, on a large number of cores
  - On each processor, we may program accelerators,
  - On each core, we have a multithread optimisation.
- In term of programming paradigms, we propose : Graph of task (Data flow oriented)/SPMD or PGAS-like or…/Data parallelism
- We have to allow the users to give expertise to the middleware, runtime system and schedulers. Scientific end-users have to be the principal target on co-design process. Frameworks and languages have to consider them first.

Outline

- Introduction
- **YML for computational science applications**
- TEZ and others tools for data science computation
- YML for computational and data science distributed and parallel computing
- Conclusion

Some elements on YML

- YML[1] Framework is dedicated to develop and run parallel and distributed applications on Cluster, clusters of clusters, and **supercomputers** (schedulers and middleware would have to be optimized for more integrated computer – cf. “K” and OmnRPC for example).

- **Independent from systems and middlewares**
  - The end users can reused their code using another middleware
  - Actually the main system is OmniRPC[3]

- Components approach
  - Defined in XML
  - **Three types** : Abstract, Implementation (in FORTRAN, C or C++;XMP,..), Graph (Parallelism)
  - Reuse and Optimized

- The parallelism is expressed through a graph description language, named **Yvette** (*name of the river in Gif-sur-Yvette where the ASCI lab was*). **LL(1) grammar, easy to parse.**

- Deployed in France Belgium, Ireland, Japan (T2K, K), China, Tunisia, USA (LBNL, TOTAL-Houston).

Graph description language: *Yvette*

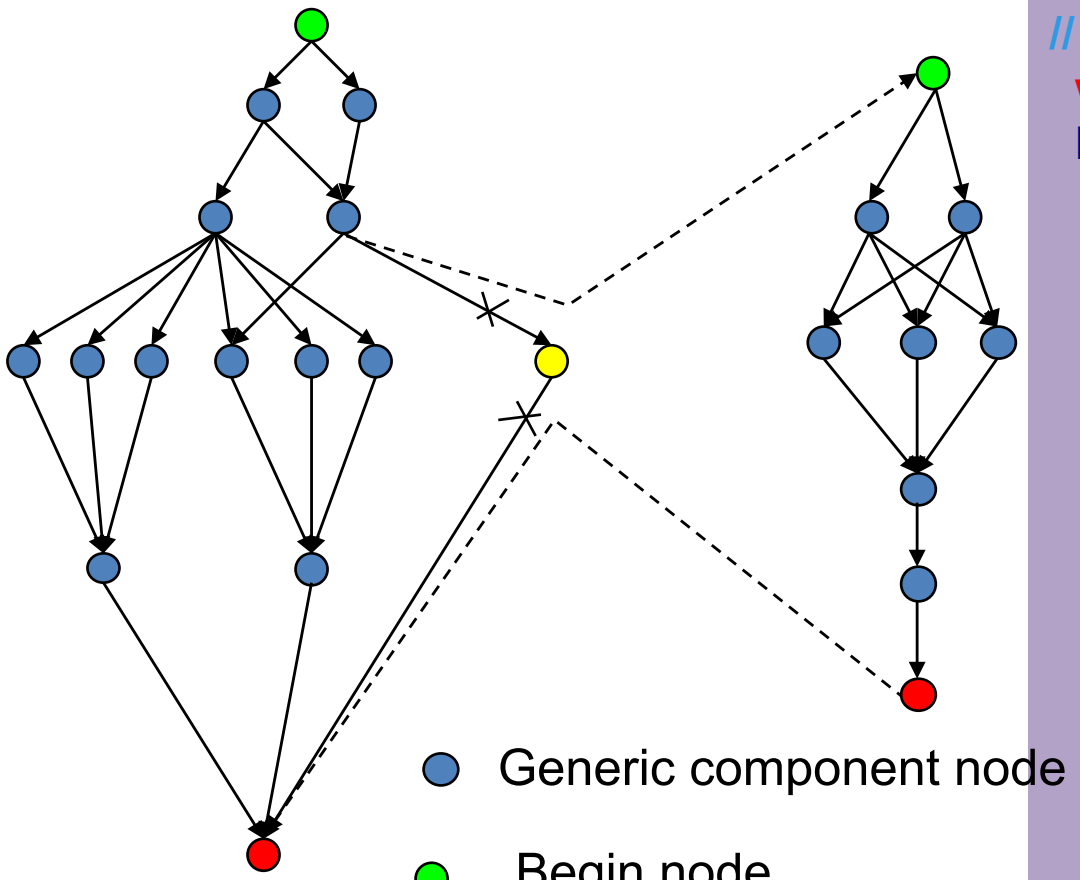
- **Language keywords**
  - Parallel sections: **par** *section1 // … // section N* **endpar**
  - Sequential Loops: **seq (i:=**begin;end**)do** … **enddo**
  - Parallel Loops: **par (i:=**begin;end**)do** … **enddo**
  - Conditionnal structure: **if** (condition) **then** … **else** … **endif**
  - Synchronization: **wait(event)** / **notify(event)**
  - Component call: **compute** NameOfComponent(args,..,..)

- **4 types de components :**
  - Abstracts
  - Graphs
  - Implementations
  - Executions

From a Taxonomy we are developing :

Graph	Granularity	Comunications	Multi-level	component	Runtime scheduler	Graph dynamic	Multi-back ends
DAG	Large	implicit	yes	yes	YML engine	No yet	Yes
General			up to 3 already				OmniRPC+ XtermWeb

Graph (n dimensions) of components/tasksYML




```
par
  compute tache1(..);
  notify(e1);
//
  compute tache2(..); migrate matrix(..);
  notify(e2);
//
  wait(e1 and e2);
  Par (i :=1;n) do
    par
       compute tache3(..);
    notify(e3(i));
    //
    if(I < n)then
       wait(e3(i+1));
       compute tache4(..);
       notify(e4);
     endif;
    //
    compute tache5(..); control robot(..);
    notify(e5); visualize mesh(…) ;
    end par
  end do par
//
  wait(e3(2:n) and e4 and e5);
  compute tache6(..);
  compute tache7(..);
end par
```

Abstract Component

```
<?xml version="1.0" ?>
<component type="abstract"    name="prodMat"  description=“Matrix
    Matrix Product" >
    <params>
     <param    name="matrixBkk"  type="Matrix" mode="in" />
     <param    name="matrixAki«   type="Matrix" mode="inout" />
     <param    name="blocksize"     type="integer" mode="in" />
    </params>
</component>
```

Future :

```
<param    name= "conv"  type= " graph_param_float" mode= "inout" />
```

Implementation Component

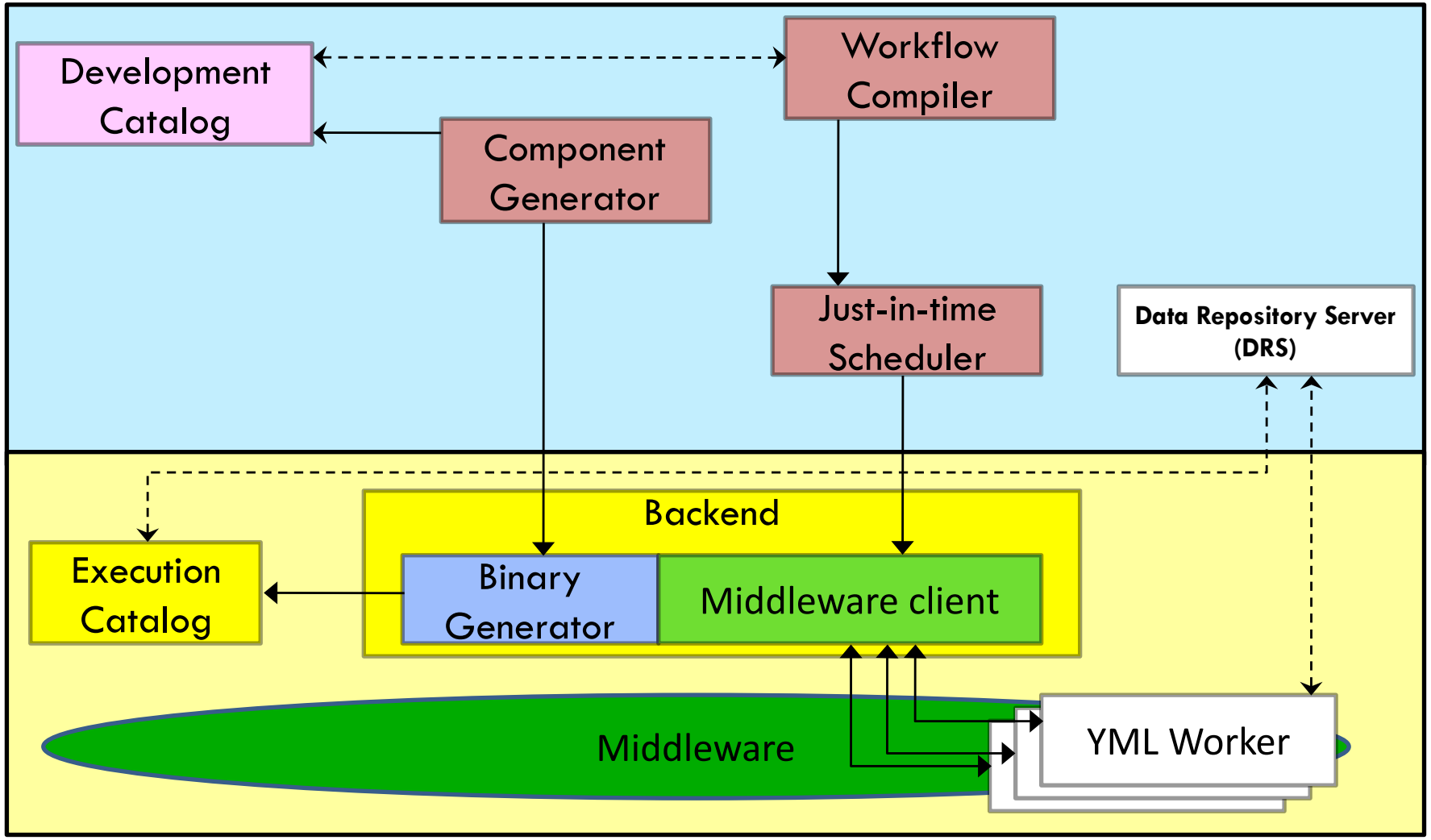
```
<?xml version="1.0"?>
<component type="impl" name="prodMat" abstract="prodMat" description="Implementation
      component of a Matrix Product">
      <impl lang="CXX">
        <header />
        <source>
        <![CDATA[
int i,j,k;
double ** tempMat;
//Allocation
  for(k = 0 ; k< blocksize ; k++)
    for (i = 0 ;i <blocksize ; i++)
         for (j = 0 ;j <blocksize ; j++)
        tempMat[i][j] = tempMat[i][j] + matrixBkk.data[i][k] * matrixAki.data[k][j];

      for (i = 0 ;i < blocksize ; i++)
              for (j = 0 ;j < blocksize ; j++)
                    matrixAki.data[i][j] = tempMat[i][j];
//Desallocation
      ]]>
              </source>
        <footer />
      </impl>
</component>
```

Graph component of Block Gauss-Jordan Method

```
<?xml version="1.0"?>
<application name="Gauss-Jordan">
<description>produit matriciel pour deux matrice carree
</description>
<graph>
blocksize:=4;
blockcount:=4;

      par (k:=0;blockcount - 1)
      do
            #inversion
            if (k neq 0) then
                  wait(prodDiffA[k][k][k - 1]);
            endif
            compute inversion(A[k][k],B[k][k],blocksize,blocksize);
            notify(bInversed[k][k]);

            #step 1
            par (i:=k + 1; blockcount - 1)
            do
                  wait(bInversed[k][k]);
                  compute prodMat(B[k][k],A[k][i],blocksize);
                  notify(prodA[k][i]);
            enddo

            par(i:=0;blockcount - 1)
            do
            #step 2.1
                  if(i neq k) then
                        wait(bInversed[k][k]);
                       compute  mProdMat(A[i][k],B[k][k],B[i][k],blocksize);
                        notify(mProdB[k][i][k]);
                  endif
            #step 2.2
                  if(k gt i) then
                        wait(bInversed[k][k]);
                        compute prodMat(B[k][k],B[k][i],blocksize);
                        notify(prodB[k][i]);
                  endif
            enddo
```

```
#Step3
            par( i:= 0;blockcount - 1)
            do
                  if (i neq k) then
                        if (k neq blockcount - 1) then
                        #step 3.1
                              par (j:=k + 1;blockcount - 1)
                              do
                                    wait(prodA[k][j]);
                                    compute
prodDiff(A[i][k],A[k][j],A[i][j],blocksize);
                                    notify(prodDiffA[i][j][k]);
                              enddo
                        endif
                        #step 3.2
                        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],blocksize);
                              enddo
                        endif
                  endif
            enddo
      enddo
   </graph>
</application>
```

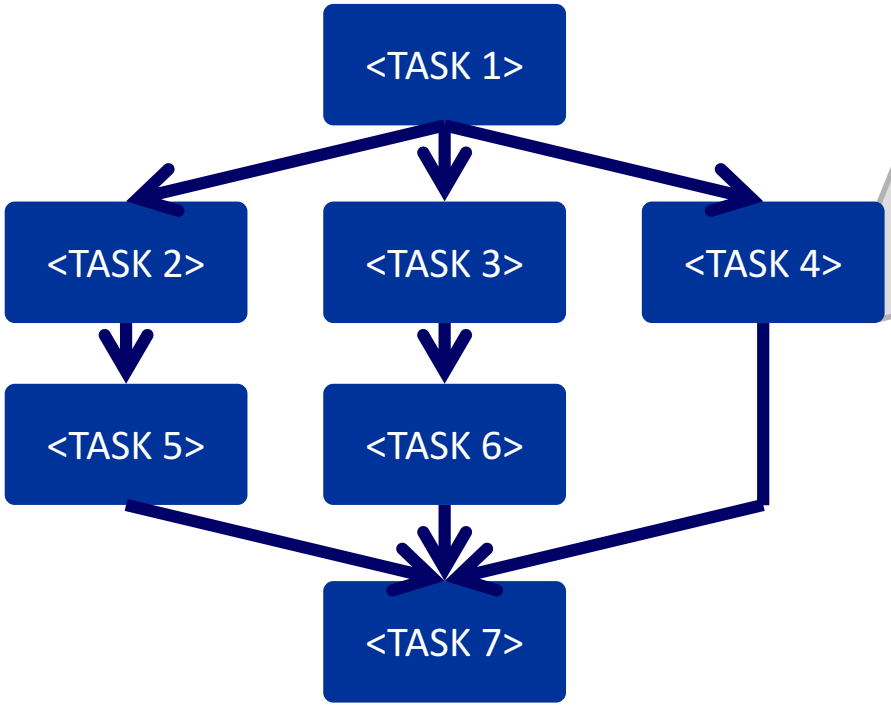
YML Architecture



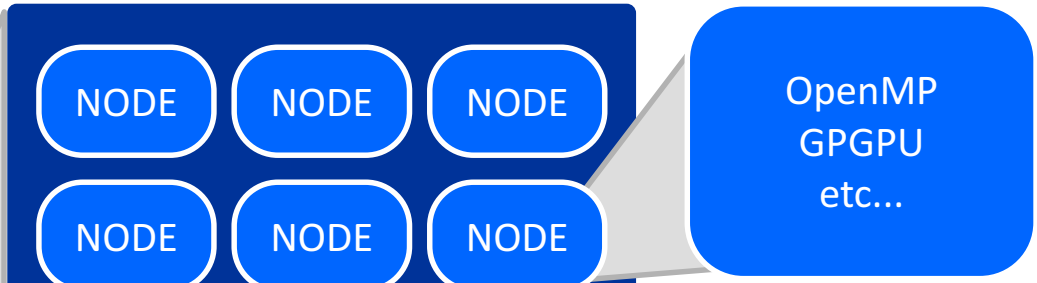
Architecture of the 1.0.5 Version

Multi-Level Parallelism Integration: YML-XMP

N dimension graphs available



YML provides a workflow programming environment and high level graph description language called YvetteML




```
for(i=0;i<n;i++){
    for(j=0;j<n;j++){
       tmp[i][j]=0.0;
#pragma xmp loop (k) on t(k)
         for(k=0;k<n;k++){
            tmp[i][j]+=(m1[i][k]*m2[k][j]);
}}}
#pragma xmp reduction (+:tmp)
```

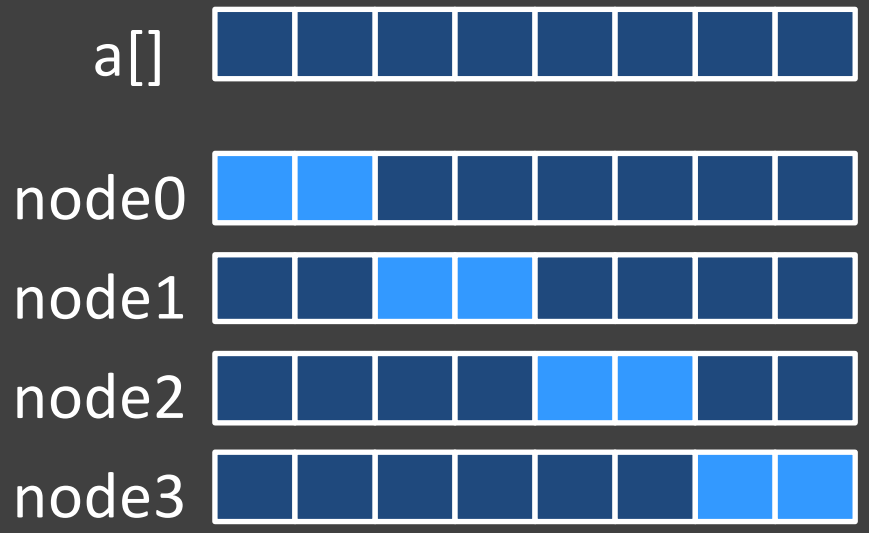
Each task is a parallel program over several nodes.
XMP language can be used to descript parallel program easily!

YML/XMP/StarPu expriments on T2K in Japan, project FP3C

XcalableMP (XMP), as example of PGAS language

- 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.
- Source (C+XMP) to source (C+MPI) compiler
- 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;
```



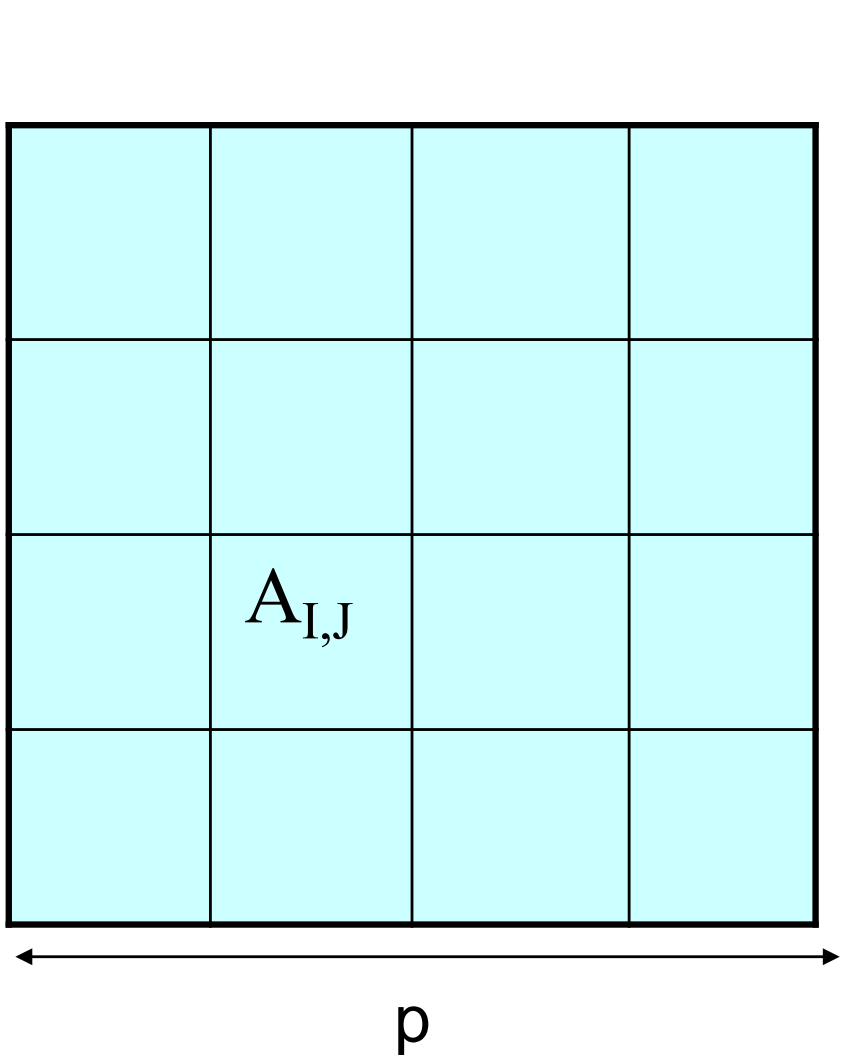
Implementation Component Extension

- Topology and number of processors are declared to be used at compile and run-time.
- Data distribution and mapping are declared
- Automatic generation for distributed language (XMP, CAF, ...)
- Used at run-time to distribute data over processes

```
<?xml version="1.0"?>
<component type="impl" name="Ex" abstract="Ex" description= "Example">
    <impl lang="XMP" nodes="CPU:(5,5)" libs=" " >
    <distribute>
        <param template=" block,block "  name="A(100,100) "  align="[i][j]:(j,i) "  />
        <param template=" block "  name="Y(100);X(100)" align="[i]:(i,*) "/>
    </distribute>
        <header />
        <source>
        <![CDATA[
      /* Computation Code */
        ]]>
      </source>
        <footer />
    </impl>
</component>
```

Scheduling

- Language for graph of task programming exists, but performance often depend of the associated middleware and scheduler : independent for the moment of the supercomputers
- Scheduling, runtime systems and middleware-OS are crucial to propose efficient programming based on graph of tasks.
- **The duration of each task has to be larger that the time to schedule the following tasks (smart scheduling would take more time…)**
- **The duration of each time has to be enough large to recover anticipated data migrations and other data movements**
- **We need the graph of control and the graph of data** to propose efficient communications optimisations and task allocations.
- We really exploit technics coming from distributed computing (on large cluster of parallel resources) adapted on supercomputers where throughputs and hierarchy are different.
- Fault tolerance, resilience may be managed by the scheduler (Miwako's talk March 12Th, Houston)



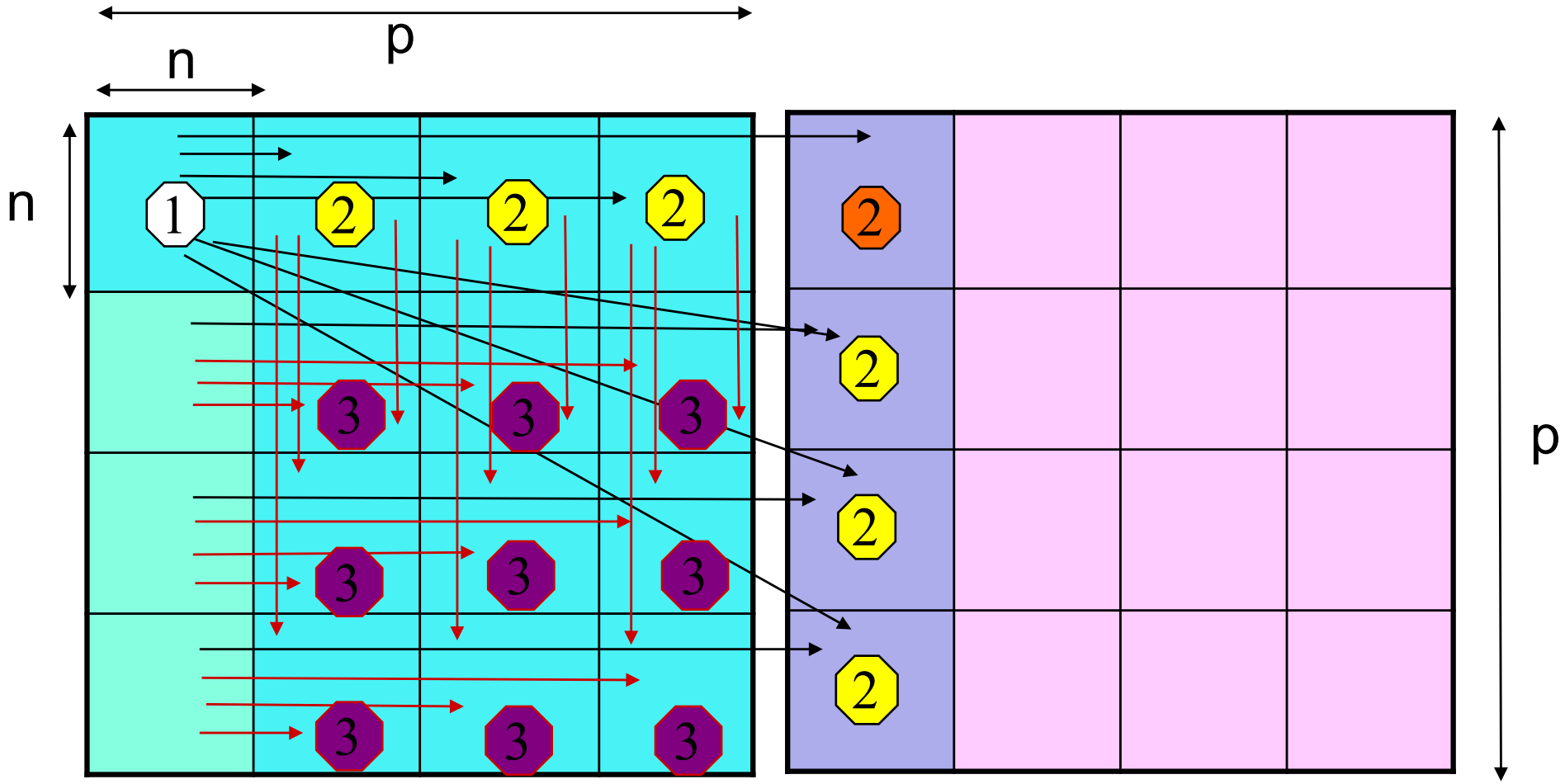
$$B = \begin{pmatrix} I & 0 & 0 & 0 \\ 0 & I & 0 & 0 \\ 0 & 0 & I & 0 \\ 0 & 0 & 0 & I \end{pmatrix}$$

A B = B A = I

Block Gauss-Jordan

Matrix size = N = p n

To invert a matrix
$2N^3$ operations
Challenge : $N = 10^6$

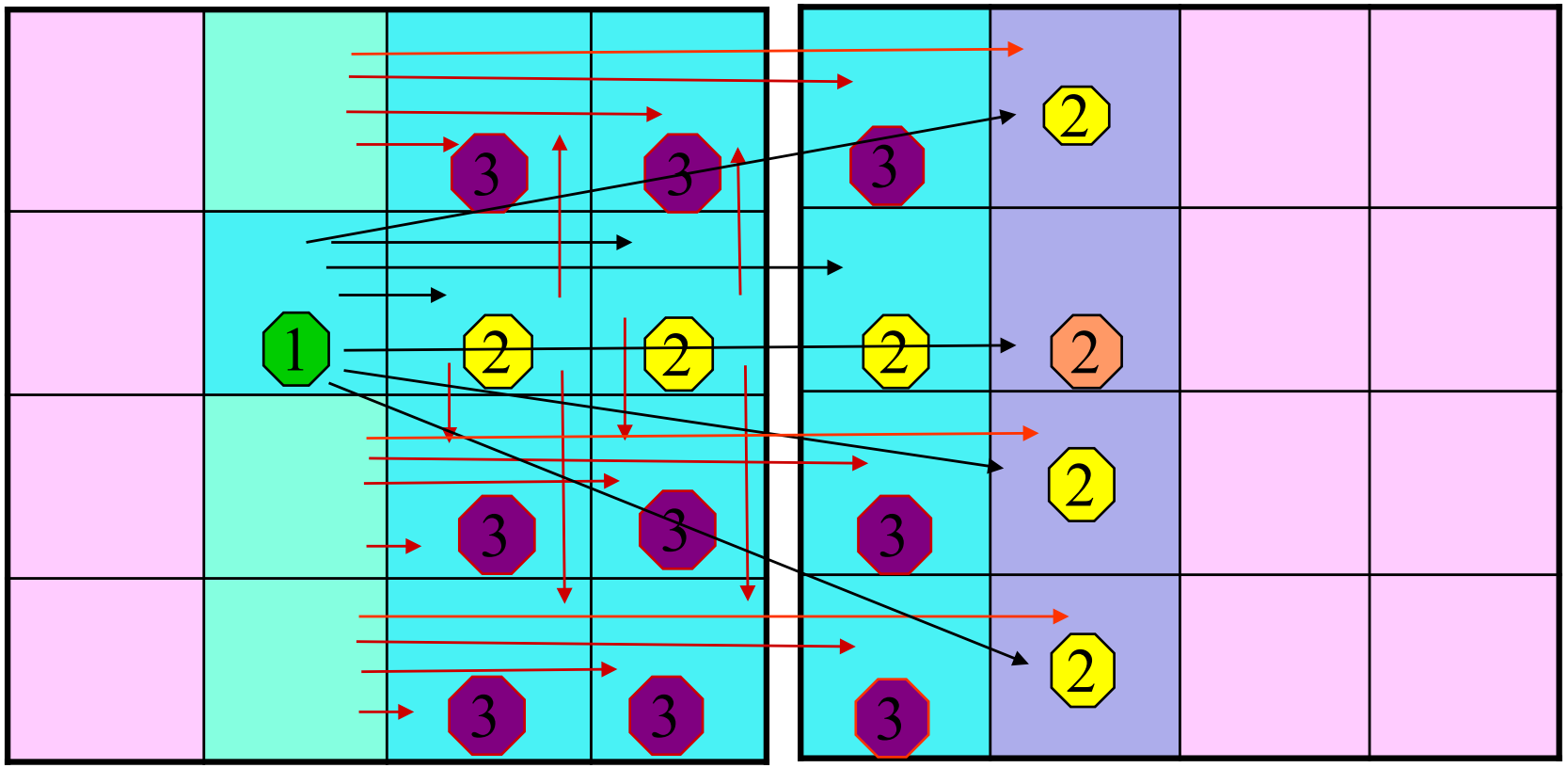


1 Element Gauss-Jordan, LAPACK, cx $=2n^3+O(n^2)$

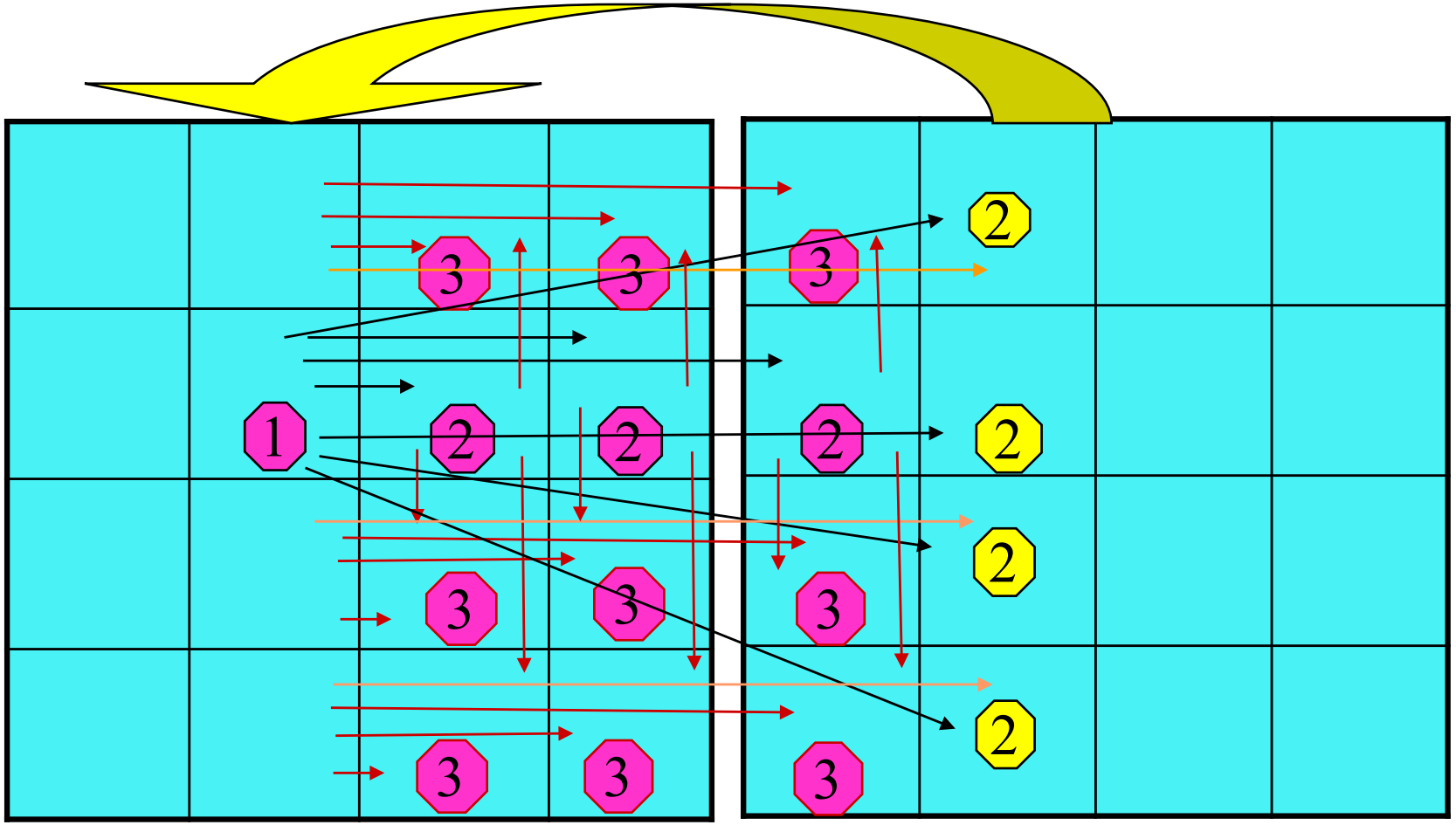
2 A = +/- A B ; BLAS3, cx = $2n^3 - n^2$, 2 A = B

3 A = A – B C ; BLAS3, cx = $2n^3$

n^2 64 bit floating point numbers



Each computing task : 1 up to 3 blocks
maximum n < (memory size of one pair) / 3
Up to $(p-1)^2$ peers



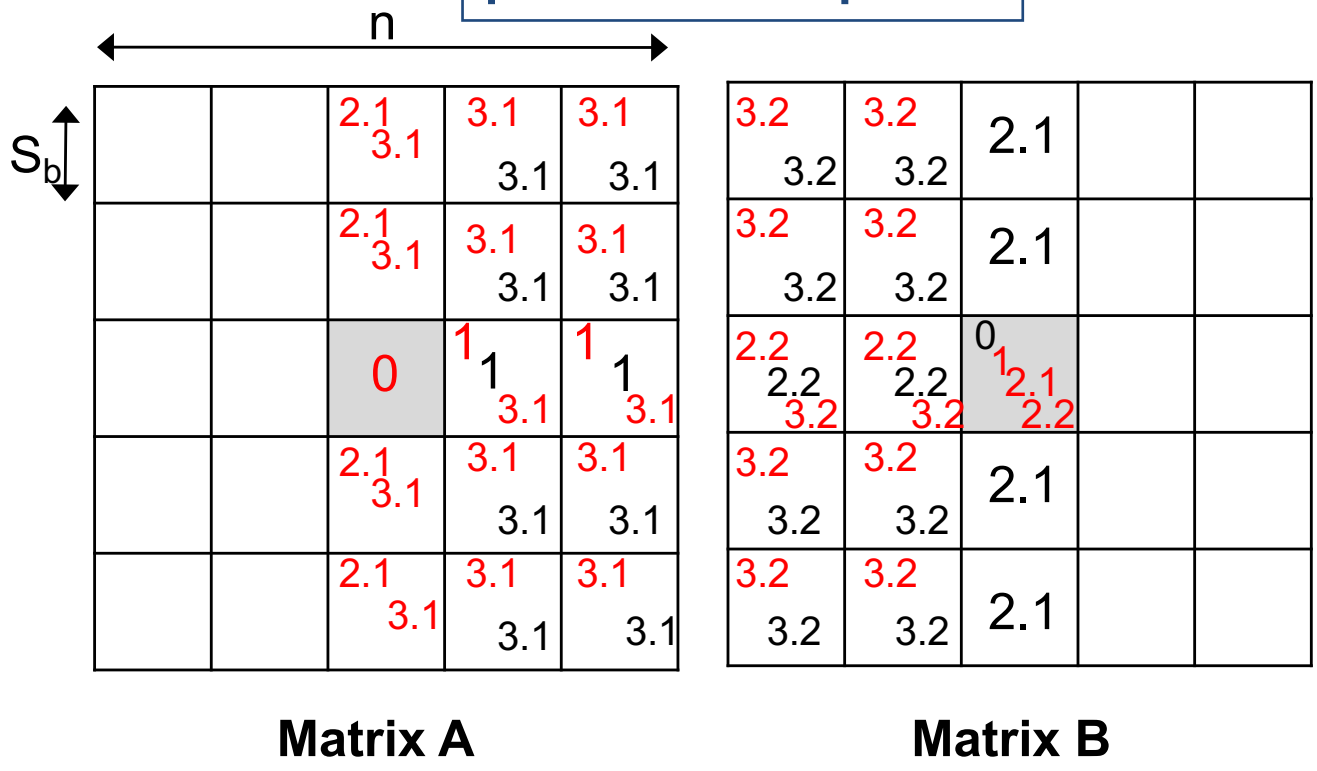
- Computation of « new » blocks on peer which minimize communications
- « update » of block at step k, on peer who updated the block a step k-1
- data send to dedicate peer ASAP

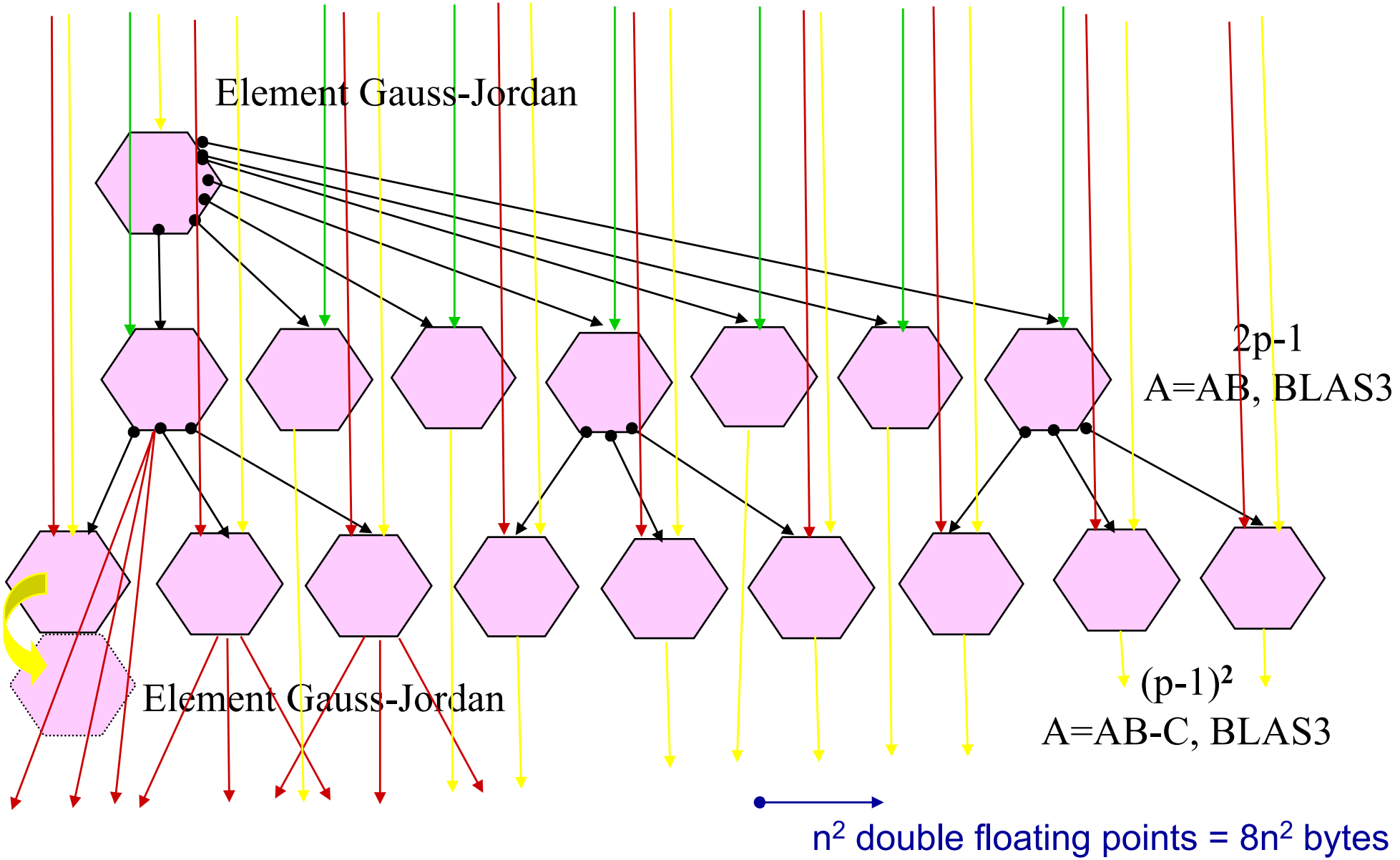
Block-based Gauss-Jordan method

Input: A (partitioned into $p \times p$ blocks)
Output: $B = A^{-1}$

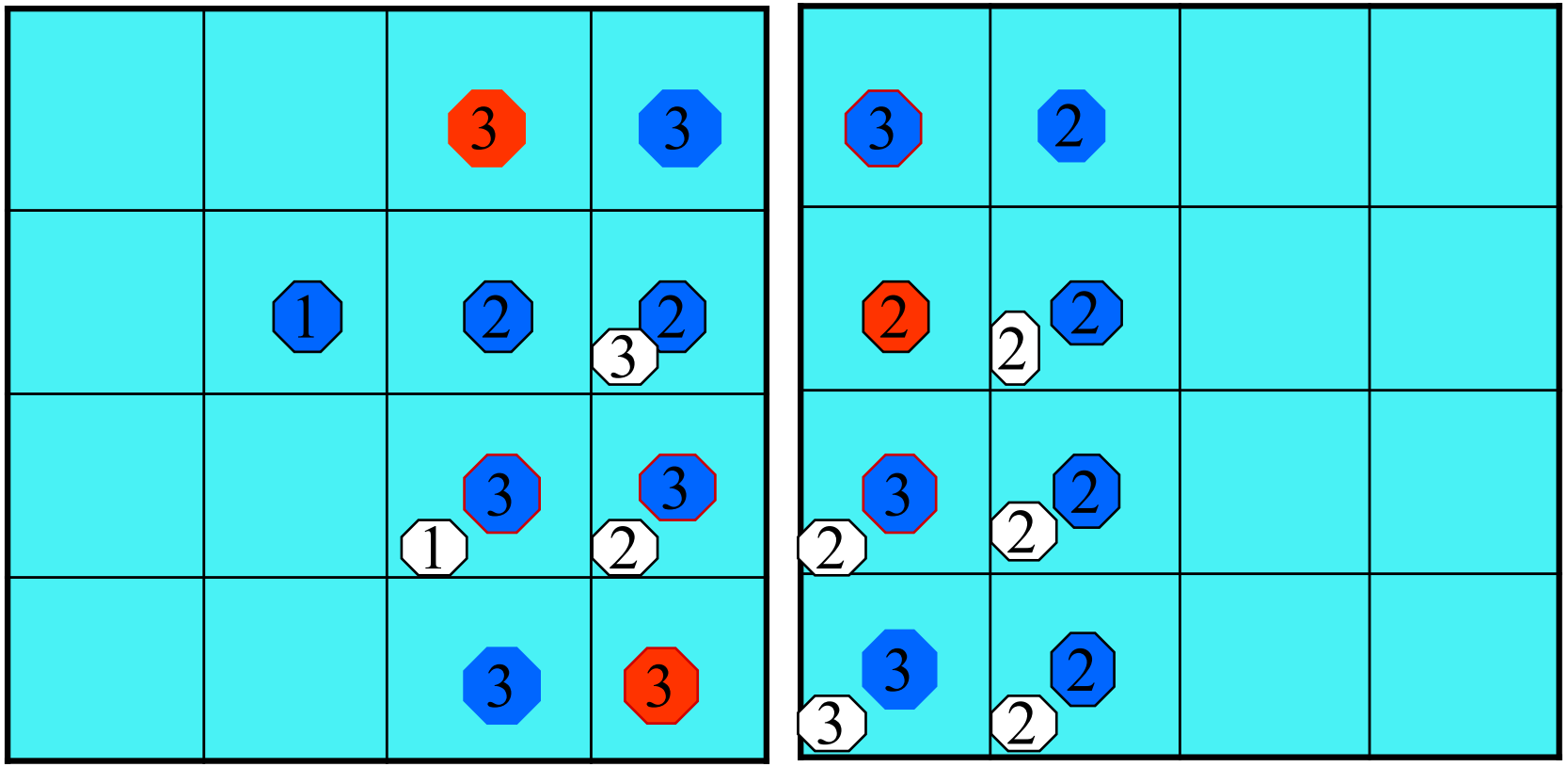
For $k = 0$ to $p - 1$
  $B_{kk} = {A^{-1}}_{kk}$
  For $i = k + 1$ to $p - 1$ **(1)**
    $A_{ki} = B_{kk} \times A_{ki}$
  End For
  For $i = 0$ to $p - 1$ **(2)**
    If $(i \neq k)$
      $B_{ik} = -A_{ik} \times B_{kk}$
    End If
    If $(i < k)$
      $B_{ki} = B_{kk} \times B_{ki}$
    End If
  End For
  For $i = 0$ to $p - 1$ **(3)**
    If $(i \neq k)$
      For $j = k + 1$ to $p - 1$
        $A_{ij} = A_{ij} - A_{ik} \times A_{kj}$
      End For
      For $j = 0$ to $k - 1$
        $B_{ij} = B_{ij} - A_{ik} \times B_{kj}$
      End For
    End If
  End For
End For

p = 5 at the step k = 2





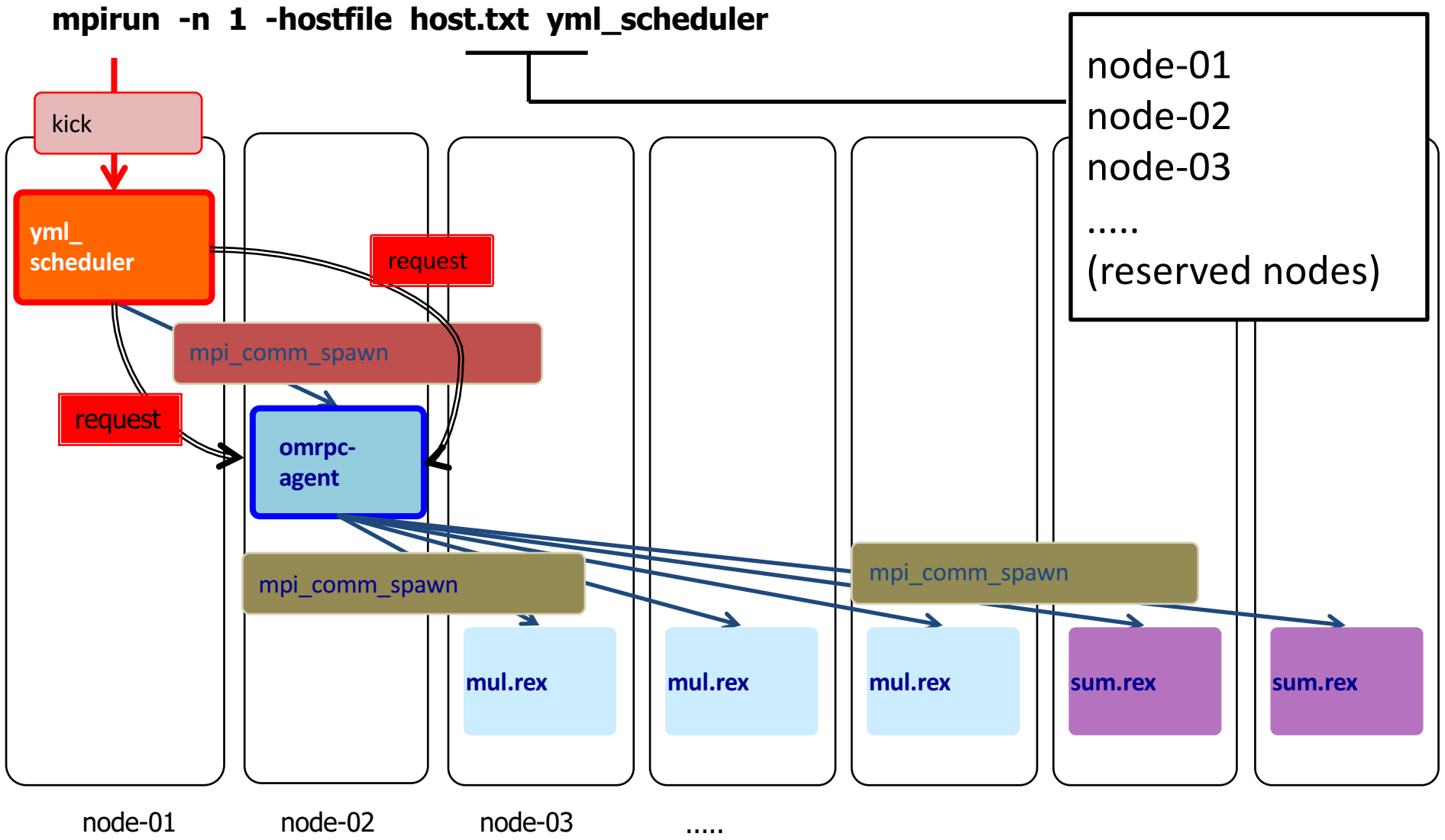
One step of the Block Gauss-Jordan method ; p=4



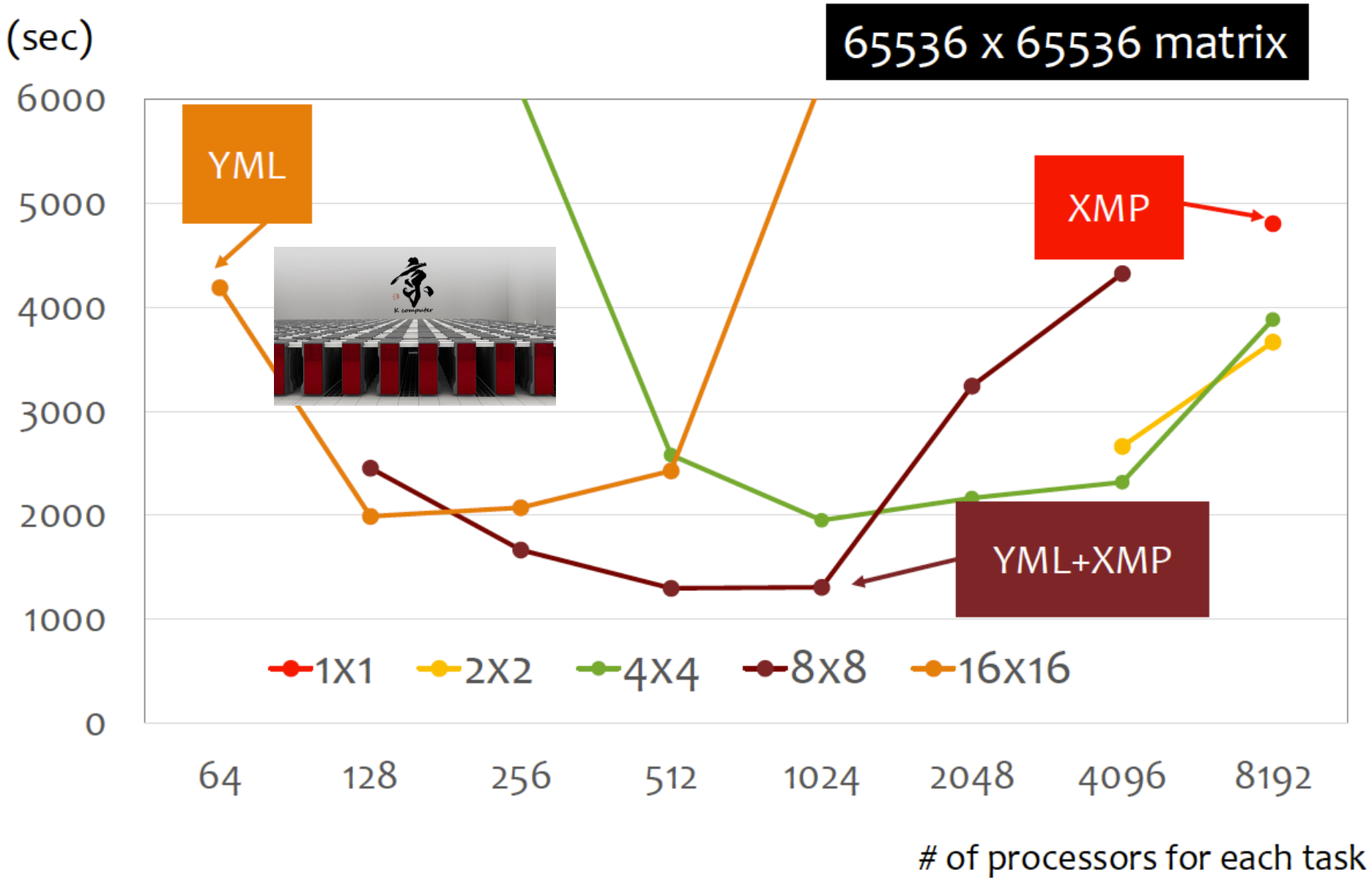
Nevertheless, we can have in parallel computing from several steps of the method.

We have to use an inter and intra steps dependency graph (3D for Block Gauss-Jordan).

FP2C : YML-XMP
on the K computer at AICS
Processes management: *OmniRPC Extension, on MPI*



Experiments (2) BGJ on K-Computer

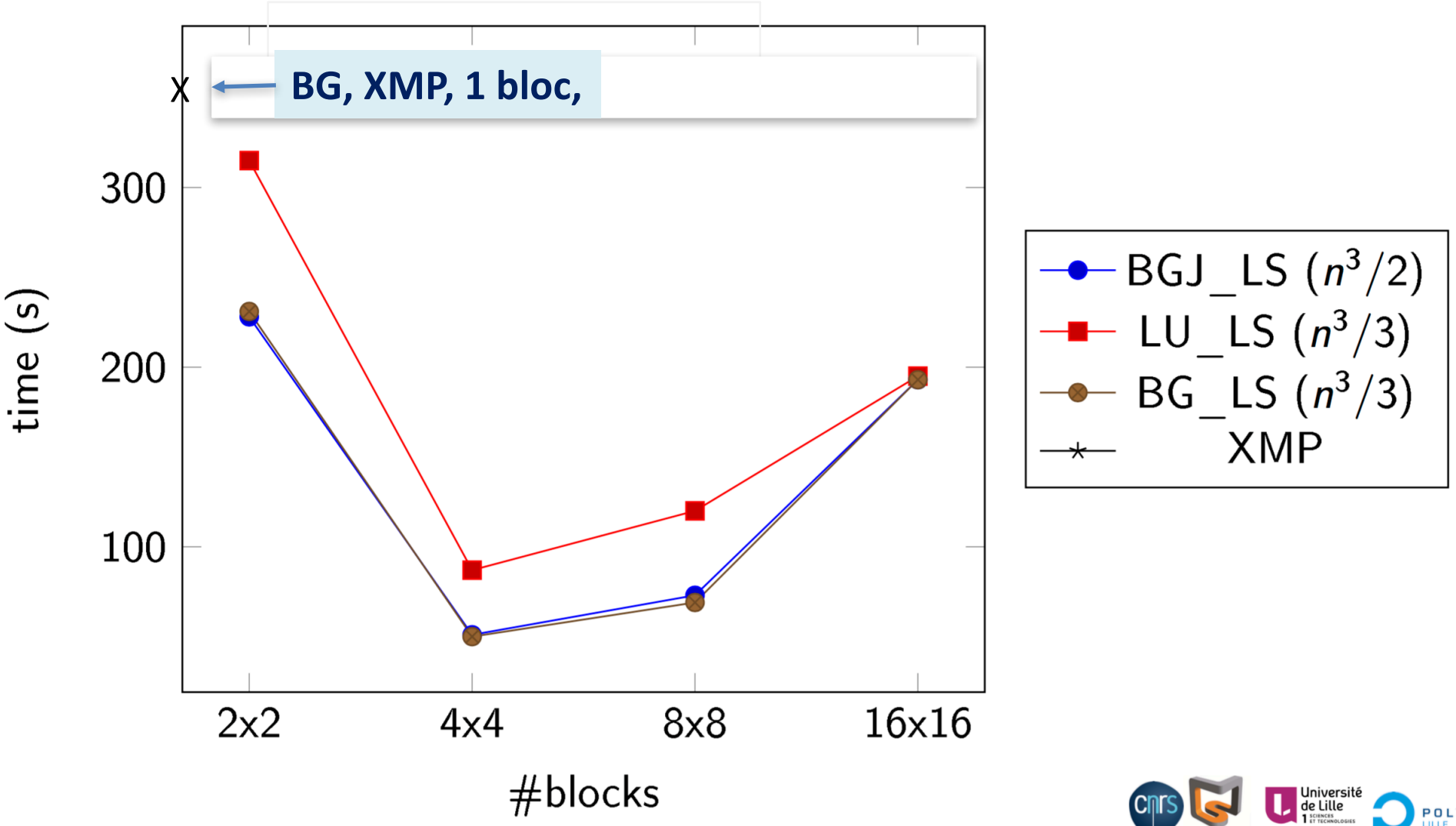


Slide written by Miwako TSUJI, RIKEN/AICS

On Poincarré, MDLS cluster
Others tests on K, Romeo

Jerome Gurhen
Master's thesis at MDLS

Figure 3: 128 procs/task



Outline

- Introduction
- YML for computational science applications
- **TEZ and others tools for data science computation**
- YML for computational and data science distributed and parallel computing
- Conclusion

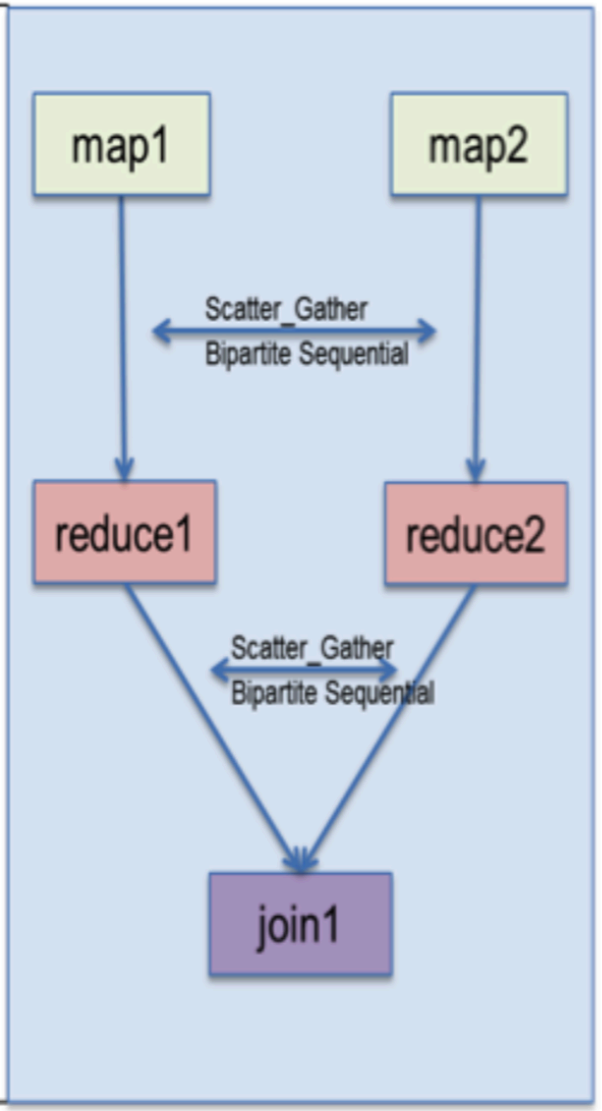
Structure of an application using Hadoop + Tez

Application = one client + one master + many slaves

- Client submits masters ; client is executed outside the cluster and may be stopped after application submission
- Master is the execution Engine (Tez) ; it handles DAG of tasks. Usually executed in a Container in Cluster.
- Slaves $\approx$ containers.
- Tasks defined into the DAG are data parallel. Usually:
  - One container contains one instance of a given task (SPMD).
  - There is one instance per data chunk
  - An instance may use many core (but multicore is most of the time useless)
- Client defines DAG in a Java-based language
- Each task is based on a I/P/O model :
  - I/O : possibility to define data movement between vertex
  - Processor is a program using DataInput and DataOutput types.

DAG Tez

```
DAG dag = new DAG();
Vertex map1 = new Vertex(MapProcessor.class);
Vertex map2 = new Vertex(MapProcessor.class);
Vertex reduce1 = new Vertex(ReduceProcessor.class);
Vertex reduce2 = new Vertex(ReduceProcessor.class);
Vertex join1 = new Vertex(JoinProcessor.class);
.......
Edge edge1 = Edge(map1, reduce1, SCATTER_GATHER,
PERSISTED, SEQUENTIAL, MOutput.class, RInput.class);
Edge edge2 = Edge(map2, reduce2, SCATTER_GATHER,
PERSISTED, SEQUENTIAL, MOutput.class, RInput.class);
Edge edge3 = Edge(reduce1, join1, SCATTER_GATHER,
PERSISTED, SEQUENTIAL, MOutput.class, RInput.class);
Edge edge4 = Edge(reduce2, join1, SCATTER_GATHER,
PERSISTED, SEQUENTIAL, MOutput.class, RInput.class);
.......
dag.addVertex(map1).addVertex(map2)
    .addVertex(reduce1).addVertex(reduce2)
    .addVertex(join1)
    .addEdge(edge1).addEdge(edge2)
    .addEdge(edge3).addEdge(edge4);
```



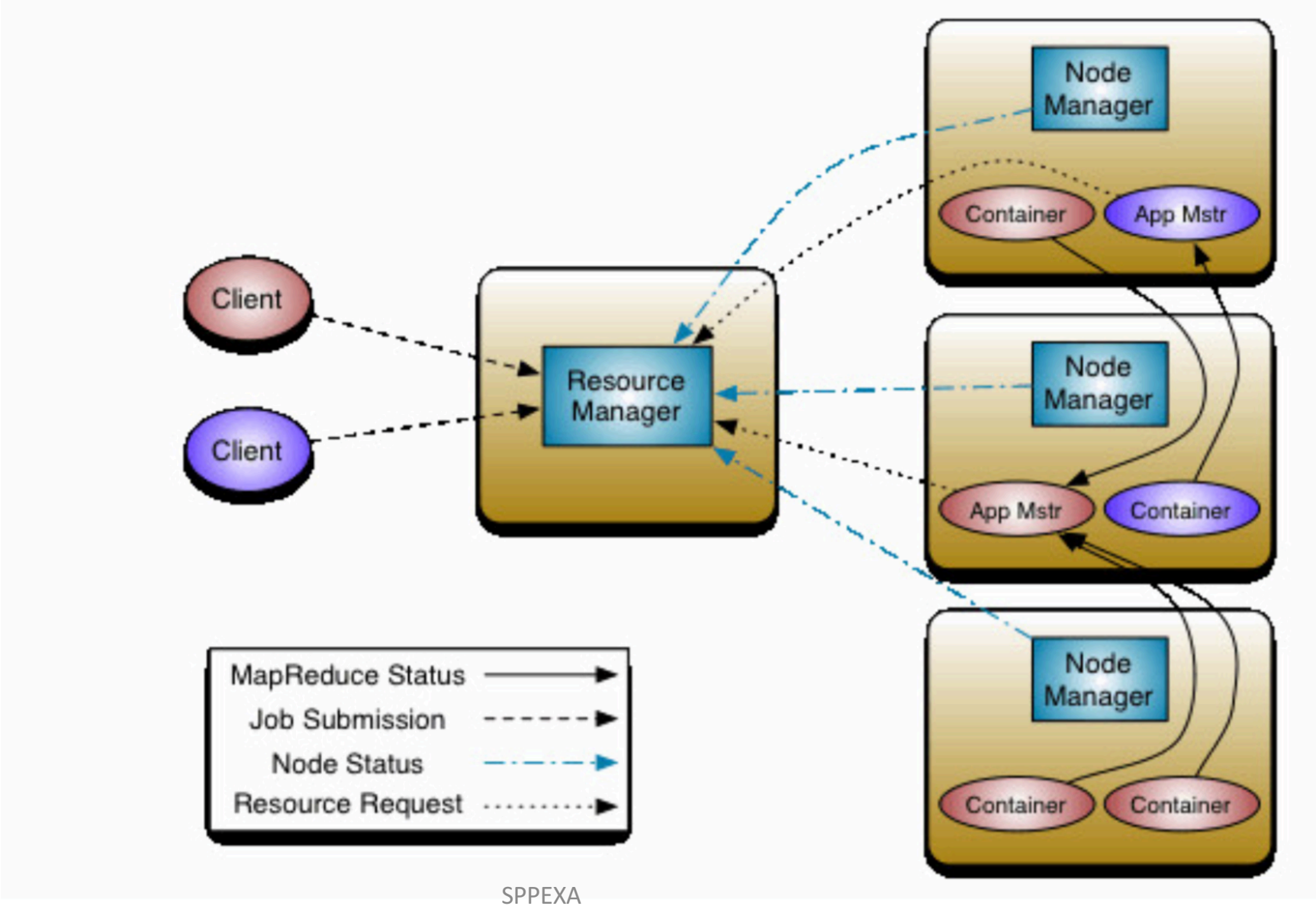
Hadoop runtime: YARN

Resource management: separating global resource management and application inner management.

- A unique Resource Manager
  - Handles client requests and fair resource allocation to users
  - Allocates (Docker-like) Containers
  - Do not consider it as a front-end on a cluster !
- For each application, an Application Master (AM) or Execution Engine (EE) is running:
  - Manages tasks (monitor, scheduling)
  - Asks RM resources and receive it as Containers
  - Running in a Container itself
- For each node, a NodeManager handles containers and interacts with RM for monitoring.

Master/Slave between RM/NM and AM/Containers, heartbeat-based communications

YARN Overview



Outline

- Introduction
- YML for computational science applications
- Data migration optimization using YML
- TEZ and others tools for data science computation
- **YML for computational and data science distributed and parallel computing**
- Conclusion

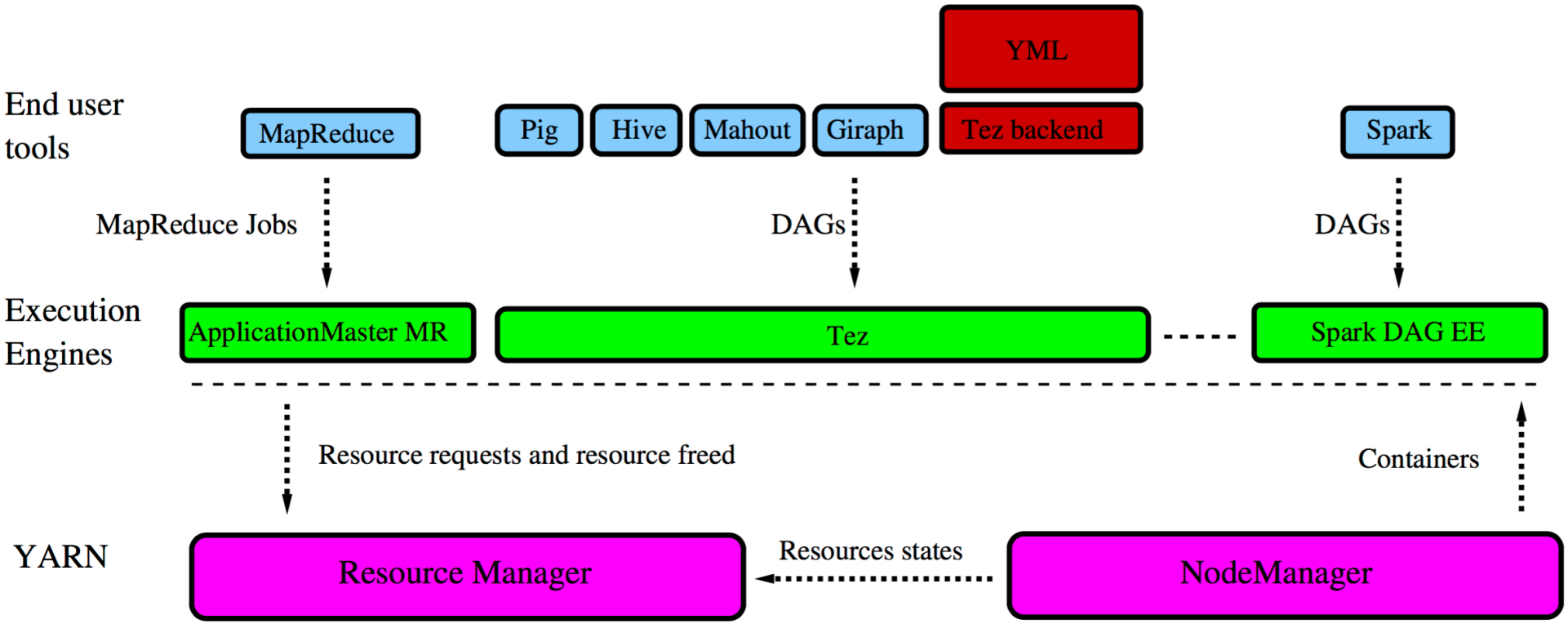
Collaboration with Laurent Bobelin

Main ideas

- Use YML as the upper level language. Enhancing it:
  - By letting abstract component definition use a new parameter type corresponding to big data : data
  - By defining a new type of impl component (hadoop) that allows YML to handle "native" Hadoop component (that will be compiled and defined using Hadoop standard dev tools)
- At the component implementation level :
  - Add a way to pass data from big data world (hadoop type component) to YML component : add a way to output data from java into data handled by YML (by giving to any java task a local access to the memory handled by YML)
  - (As data output of YML will not be big (i.e. big as in big data) there is not need to add the other way support)

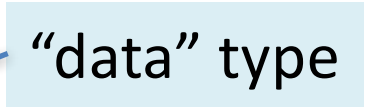
Main ideas - cont'd

- At the application setup, there is no, by doing so, major changes in actual way of doing things:
  - Hadoop-based programs are developed as any normal Tez Processor task.
  - Compilation of YML is unchanged.
- At application start up:
  - A java client will submit an application that starts Tez that will be used as Execution Engine
  - Once Tez starts, it asks for a specified amount of Containers.
  - Once containers are started, Tez start YML scheduler based on omnirpc-mpi


End user tools
MapReduce
Pig
Hive
Mahout
Giraph
YML
Tez backend
Spark
MapReduce Jobs
DAGs
DAGs
Execution Engines
ApplicationMaster MR
Tez
Spark DAG EE
Resource requests and resource freed
Containers
YARN
Resource Manager
Resources states
NodeManager

An example from YML point of view: Abstract components

Make a mean on some big data values and add it to some other value




```
<?xml version="1.0"?>
<yml-query login="XXX" password="XXX">
   <component type="abstract" name="bigmean"
  description="This component gives mean of some big data">
      <param type="real" mode="out" name="res" />
      <param type="data" mode="in" name="a" />
   </component>
</yml-query>

<yml-query login="XXX" password="XXX">
   <component type="abstract" name="add"
  description="This component add two numbers">
      <param type="real" mode="out" name="res" />
      <param type="real" mode="in" name="a" />
      <param type="real" mode="in" name="b" />
   </component>
</yml-query>
```

An example from YML point of view: Impl components

Hadoop

```
<?xml version="1.0"?>
<yml-query login="XXX" password="XXX">
    <component type="hadoop" name=bigmean_impl"
     description="An mplementation component for bigmean" abstract="bigmean"
     class="org.foo.bar.bigmeanImpl">
    </component>
</yml-query>

<?xml version="1.0"?>
<yml-query login="XXX" password="XXX">
    <component type="impl" name="add_impl"
     description="An implementation component for sum" abstract="add">
        <globals><![CDATA[
#include <stdlib.h>
]]>
        </globals>
        <source lang="CXX" libs="">
res = a + b;
        </source>
    </component>
</yml-query>
```

Outline

- Introduction
- YML for computational science applications
- TEZ and others tools for data science computation
- YML for computational and data science distributed and parallel computing
- **Conclusion**

Conclusion

Graph of components and containers programming is a potential solution for extreme computational and data science computing

Multi-level programming, including PGAS developed software, would be a solution for exascale computing

YML-XMP, YML-XACC, YML-TEZ and others solutions ”proof” the Interest of this programming paradigm, experimenting on several Example. SPPEXA/MYX project contributes to validate this programming programming

HPC + ”Data Science”” + exascale + new programming paradigm >>>

Intelligent Machine Learning (project with John Wu, LBNL)