Toward Intelligent Linear Algebra Methods and Multi-Level Programming Paradigms for Extreme Computing

Serge G. Petiton

With the collaboration of several researchers

University Lille 1, Sciences and Technologies, and
CNRS (CRIStAL Laboratory and Maison de la Simulation Paris-Saclay)

Outline

- **Introduction**
- Krylov subspace auto-tuned restarted methods
- Asynchronous Unite-and-Conquer methods
- Multilevel programming paradigm : Graph of components/PGAS
- What Intelligent Krylov methods for extreme computing?
- Conclusion

Extreme computing (Distributed and Parallel)
some correlated goals

- Minimize the global computing time,
- Accelerate the convergence (and analysis at runtime)
- Minimize the number of communications (optimized Ax, asynchronous comp, communication compiler and mapper,….)
- Minimize the number of longer size scalar products,
- Minimize memory space, cache optimization….
- Select the best sparse matrix compressed format,
- Mixed arithmetic
- Unite and Conquer methods
- Minimize energy consumption
- Resilience
- …….

These criteria are some of the requirements for future Exascale computing and beyond

Several optimizations are not possible at compile time and have to be decided at runtime:
Auto-tuning, smart-tuning, ….

*The goal of this talk is to illustrate that we would need **intelligent linear algebra** methods to create the next generation of High Performance numerical software, associated with adapted **programming paradigms** allowing the end-user to **give expertise***

Outline

- Introduction
- **Krylov subspace auto-tuned restarted methods**
  - **Auto-tunings at runtime for Krylov methods**
  - Subspace size
  - Incomplete orthogonalisation
  - Restart strategies
  - Sparse formats
  - Energy consumption
- Asynchronous Unite-and-Conquer methods
- Multilevel programming paradigm : Graph of components/PGAS
- What Intelligent Krylov methods for extreme computing?
- Conclusion

GMRES example : about memory space, dot products and sparse matrix vector multiplication

1 matrix vector multiplication

m, subspace size

J scalar product

1. *Start*: Choose x_0 and compute $r_0 = f - Ax_0$ and $v_1 = r_0/\|r_0\|$.
2. *Iterate*: For $j = 1, 2, \cdots, m$ do:
   $h_{i,j} = (Av_j, v_i), i = 1, 2, \cdots, j,$
   $\hat{v}_{j+1} = Av_j - \sum_{i=1}^{j} h_{i,j} v_i,$
   $h_{j+1,j} = \|\hat{v}_{j+1}\|$, and
   $v_{j+1} = \hat{v}_{j+1}/h_{j+1,j}$.
3. *Form the approximate solution*:
   $x_m = x_0 + V_m y_m$, where y_m minimizes $\|\beta e_1 - \bar{H}_m y\|$, $y \in R^m$.
4. *Restart*:
   Compute $r_m = f - Ax_m$; if satisfied then stop
   else compute $x_0 := x_m$, $v_1 := r_m/\|r_m\|$ and go to 2.

Subspace computation : $O(m^3)$

Memory space :
sparse matrix : nnz elements
Krylov basis vectors : n **m**
Hessenberg matrix : **m m**

Scalar products, at j fixed:
Sparse Matrix-vector product : n of size C
Orthogonalization : j of size n

m, the subspace size, may be auto-tuned at runtime to minimize the space memory occupation and the number of scalar product, with better or approximately same convergence behaviors.

GMRES : about memory space and dot products

1. *Start*: Choose x_0 and compute $r_0 = f - Ax_0$ and $v_1 = r_0/\|r_0\|$.
2. *Iterate*: For $j = 1, 2, \cdots, m$ do:
   $h_{i,j} = (Av_j, v_i), i = 1, 2, \cdots, j,$
   $\hat{v}_{j+1} = Av_j - \sum_{i=1}^{j} h_{i,j} v_i,$
   $h_{j+1,j} = \|\hat{v}_{j+1}\|$, and
   $v_{j+1} = \hat{v}_{j+1}/h_{j+1,j}.$
3. *Form the approximate solution*:
   $x_m = x_0 + V_m y_m$, where y_m minimizes $\|\beta e_1 - \bar{H}_m y\|$, $y \in R^m$.
4. *Restart*:
   Compute $r_m = f - Ax_m$; if satisfied then stop
   else compute $x_0 := x_m$, $v_1 := r_m/\|r_m\|$ and go to 2.

Incomplete orthogonalization (Y. Saad): i.e. i= from max(1,j-q) to j
q>0. Then, J-q+1 bands on the Hesseberg matrix.

Memory space :

- sparse matrix : nnz (i.e. < C n) elements
- Krylov basis vectors : n m
- Hessenberg matrix : m m

Scalar products, at j fixed:

- Sparse Matrix-vector product : n of size C
- Orthogonalization : m of size n

m, the subspace size, may be auto-tuned at runtime to minimize the space memory occupation and the number of scalar product, with better or approximately same convergence behaviors. The number of vectors othogonalized with the new one may be auto-tuned at runtime. The subspace size may be large!

Outline

- Introduction
- **Krylov subspace auto-tuned restarted methods**
  - Auto-tunings at runtime for Krylov methods
  - **Subspace size**
  - Incomplete orthogonalisation
  - Restart strategies
  - Sparse formats
  - Energy consumption
- Asynchronous Unite-and-Conquer methods
- Multilevel programming paradigm : Graph of components/PGAS
- What Intelligent Krylov methods for extreme computing?
- Conclusion

Previous works, subspace auto-tuning algorithms

- Subspace size : different auto-tuning at runtime
  - Subspace size increase, until a fixed limit [Katagiri00][Sosonkina96],..
  - Subspace size decrease, until a fixed limit [Baker09],…..
  - Restart Trigger [Zhang04], restart when stagnation is detected.
- Orthogonalization : no auto-tuning at runtime
  - Prior to execution : [Jia94]

Remark, *in general*:

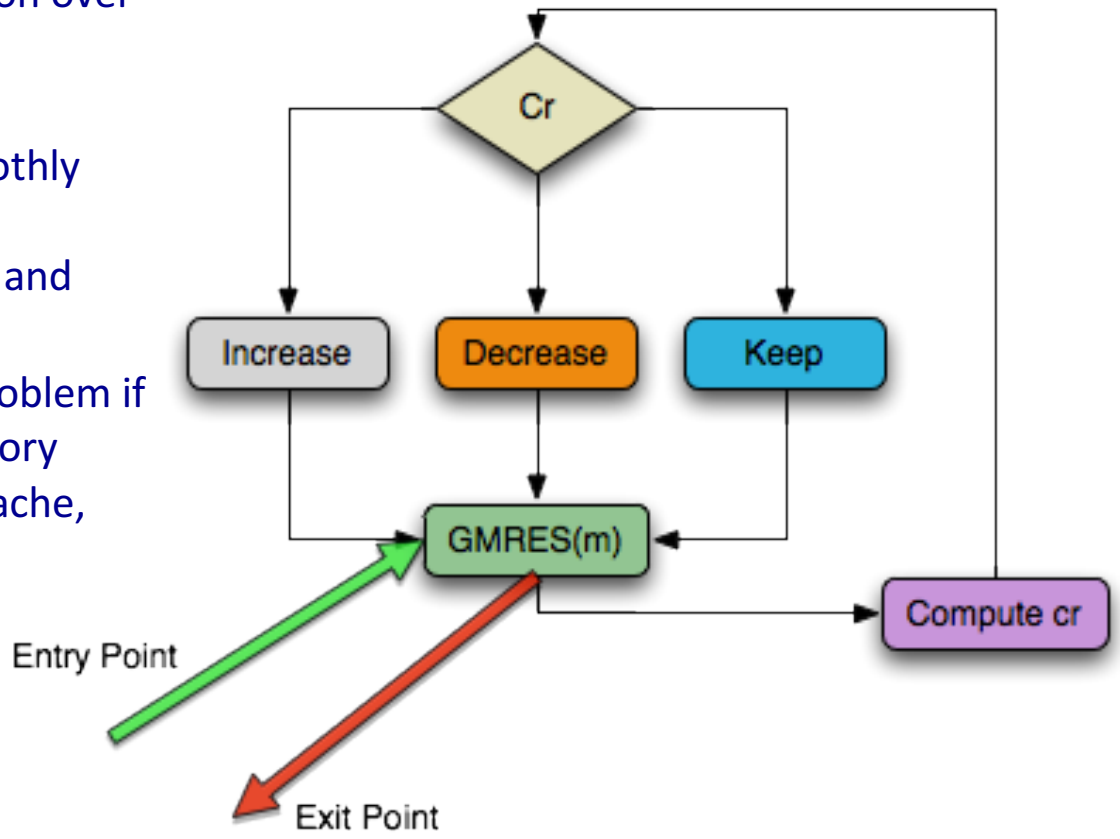
- Greater subspace size -> better convergence/long restart, less iterations
- Smaller subspace size -> slow convergence, stagnation, short restart, more iteration
- Choice of m is mandatory.

For difficult problems we have to use a large subspace size to reach convergence : the numerical stability and the quality of the orthogonalization are crucial.

Auto-Tuning Algorithms

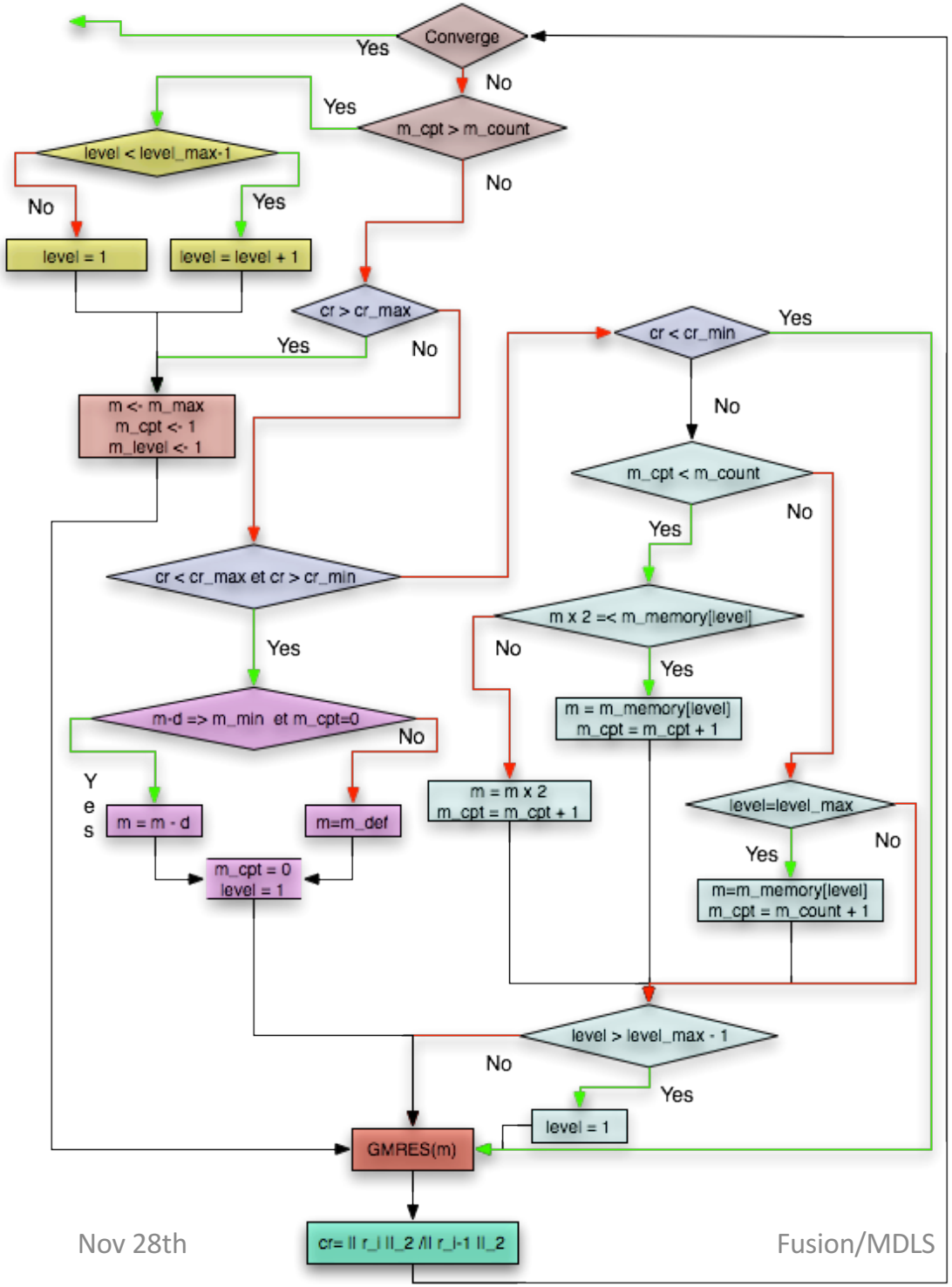
with Pierre-Yves Aquilenti (TOTAL)

- Subspace size
  - Evaluate convergence progression over some iterations.
  - Decrease if convergence are monotonous or if they are smoothly slowing (approximately same convergence but minimize time and space)- **Cr medium**
  - Increase if convergence stall (problem if we increase too much the memory space), Track memory levels : Cache, RAM, Nodes. **Cr low**
  - Do nothing if **Cr high**



Cr = norm2 (r_i) / norm2(r_{i_1})

Easy to implement using libraries (both for GMRES and Arnoldi method for example)



Parameters

d : number of steps between successive decreases,

m_min: minimum subspace size value,

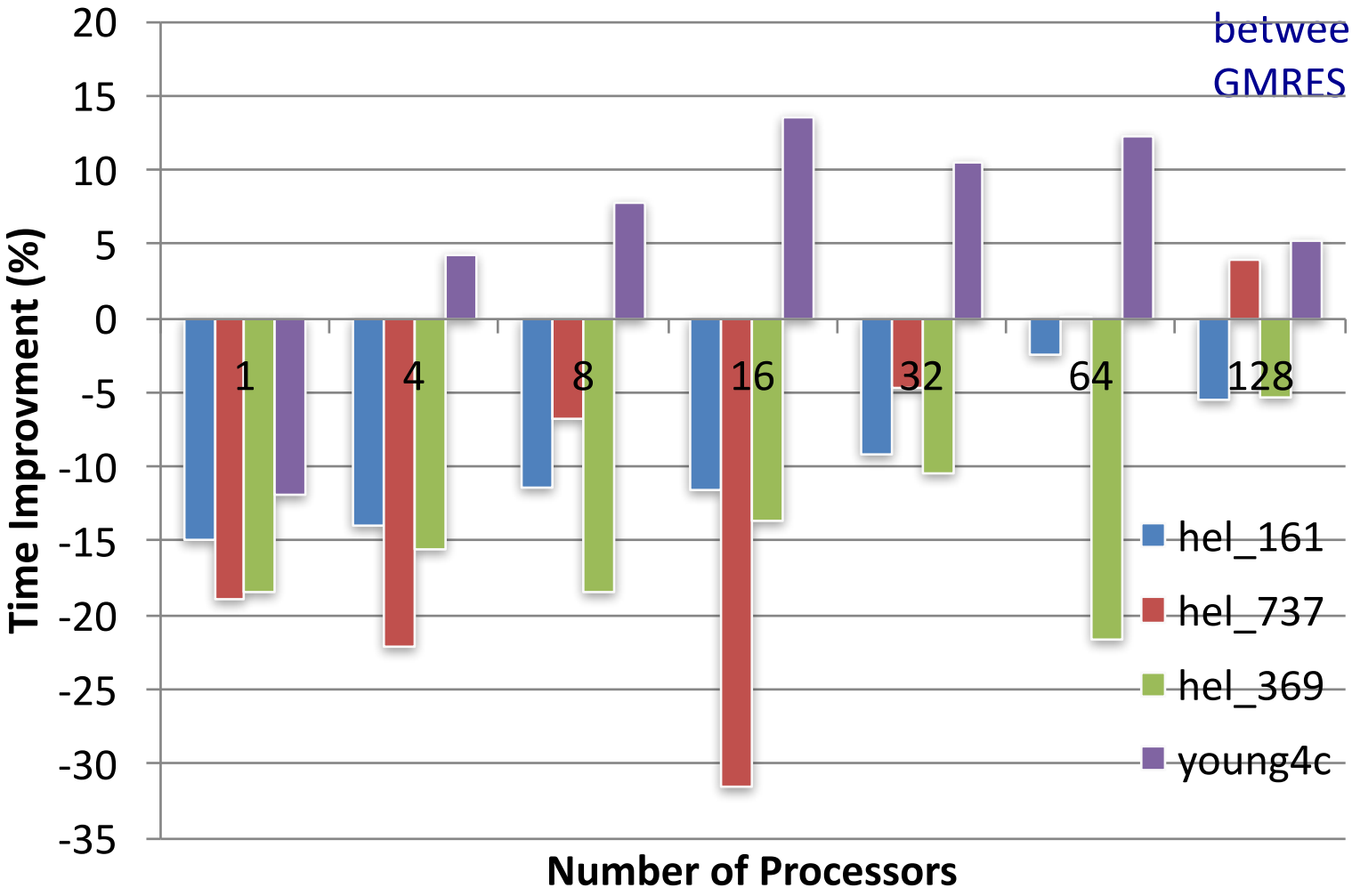
m_max : maximum subspace size value,

m_counts : number of successive soft increase before intending “special” Increases,

m_memory[ ] : array containing subspace size values for hardware increase

- Our algorithm compared to no auto-tuning

Simple Strategy, decrease or keep the subspace size between successives GMRES restarts



Less is better

Auto-Tune Restarted GMRES With Caches
with Pierre-Yves Aquilenti (TOTAL)

- We define levels of GMRES restart parameter auto-tuning increase depending on levels of memory

Cache L1	Cache L2	Cache L3	RAM	SWAP
4	10	20	200	1000

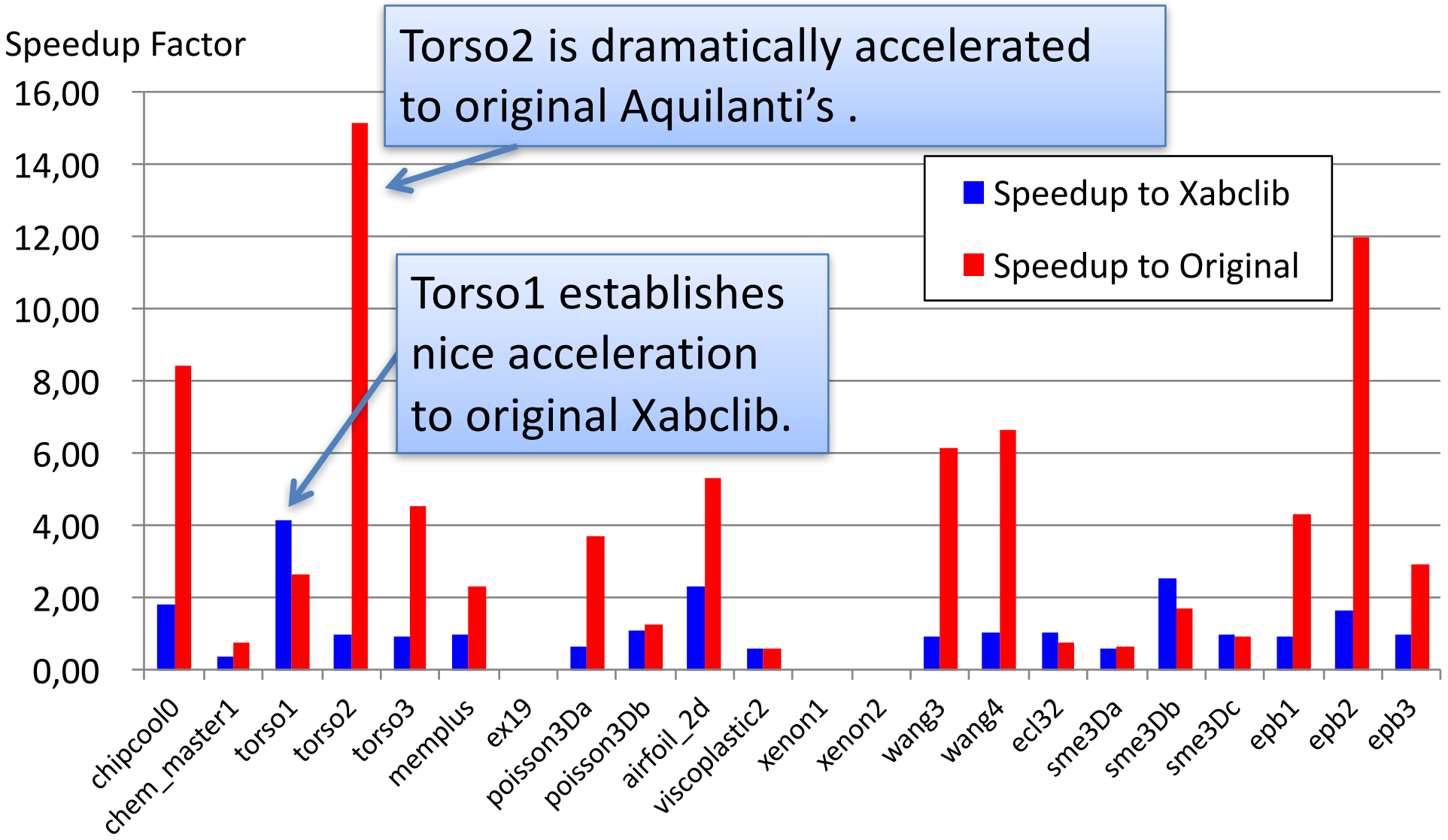
$$(nnz + 3n + m(m+1) + n(m+1)) \times SizeOfScalar \geq MemoryBytesLevel$$

$$\frac{MemoryBytesLevel}{SizeOfScalar} - nnz - 4n = m^2 + m(n+1)$$

nnz = number of non zeros of the matrix
m = subspace size / restart parameter
n = matrix size

With Pierre-Yves Aquilenti (TOTAL) and Takahiro Katagari (U. Tokyo)

16 Threads on the T2K Open Supercomputer (1 node), Xabclib_GMRES V1.00



Outline

- Introduction
- **Krylov subspace auto-tuned restarted methods**
  - Auto-tunings at runtime for Krylov methods
  - Subspace size
  - **Incomplete orthogonalisation**
  - Restart strategies
  - Sparse formats
  - Energy consumption
- Asynchronous Unite-and-Conquer methods
- Multilevel programming paradigm : Graph of components/PGAS
- What Intelligent Krylov methods for extreme computing?
- Conclusion

Incomplete orthogonalization Auto-Tuning

Complete orthogonalisation : we orthogonalise with all the previous computed vectors of the basis, i.e. at step k, we ortogonalise with k vectors, which generates k scalar product at step k.

Incomplete orthogonalisation : we orthogonalize with only min(k,q) previous computed vectors of the basis, i.e. at step k, we ortogonalise only with min(k,q) vectors, q < m. DQGMRES : [Saad '94], DQGMRES : [Wu '97] IGMRES : [Brown '86][Jia '07]

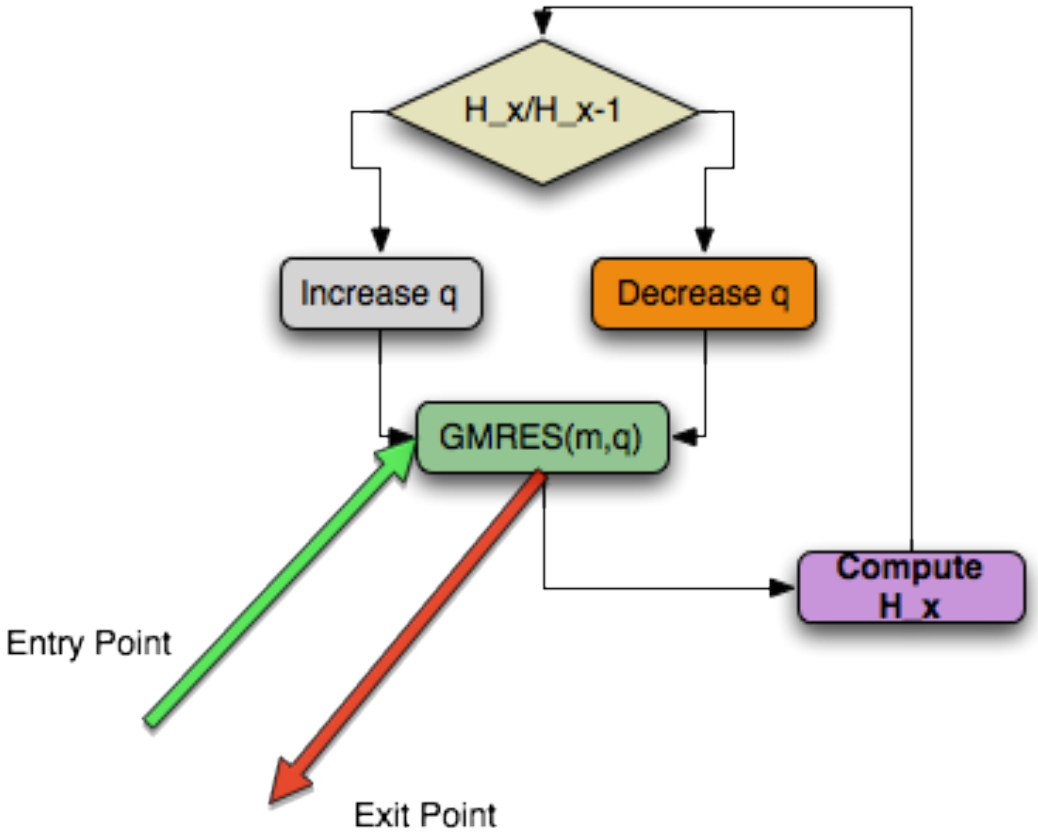
Then, we have only q scalar product at step k (for k > or equal to q).

Complete ortogonalization : k scalar product for k fixed
Incomplete orthogonalization : q scalar product for k fixed, q < k

We may then save k-q scalar products, for q < k, and, then, several synchronized communications .

Even, if the number of iterations may be a little larger, we minimize a lot of long globa communications generated by scalar products.

Incomplete orthogonalization algorithm at runtime (with P.-Y. Aquilenti)

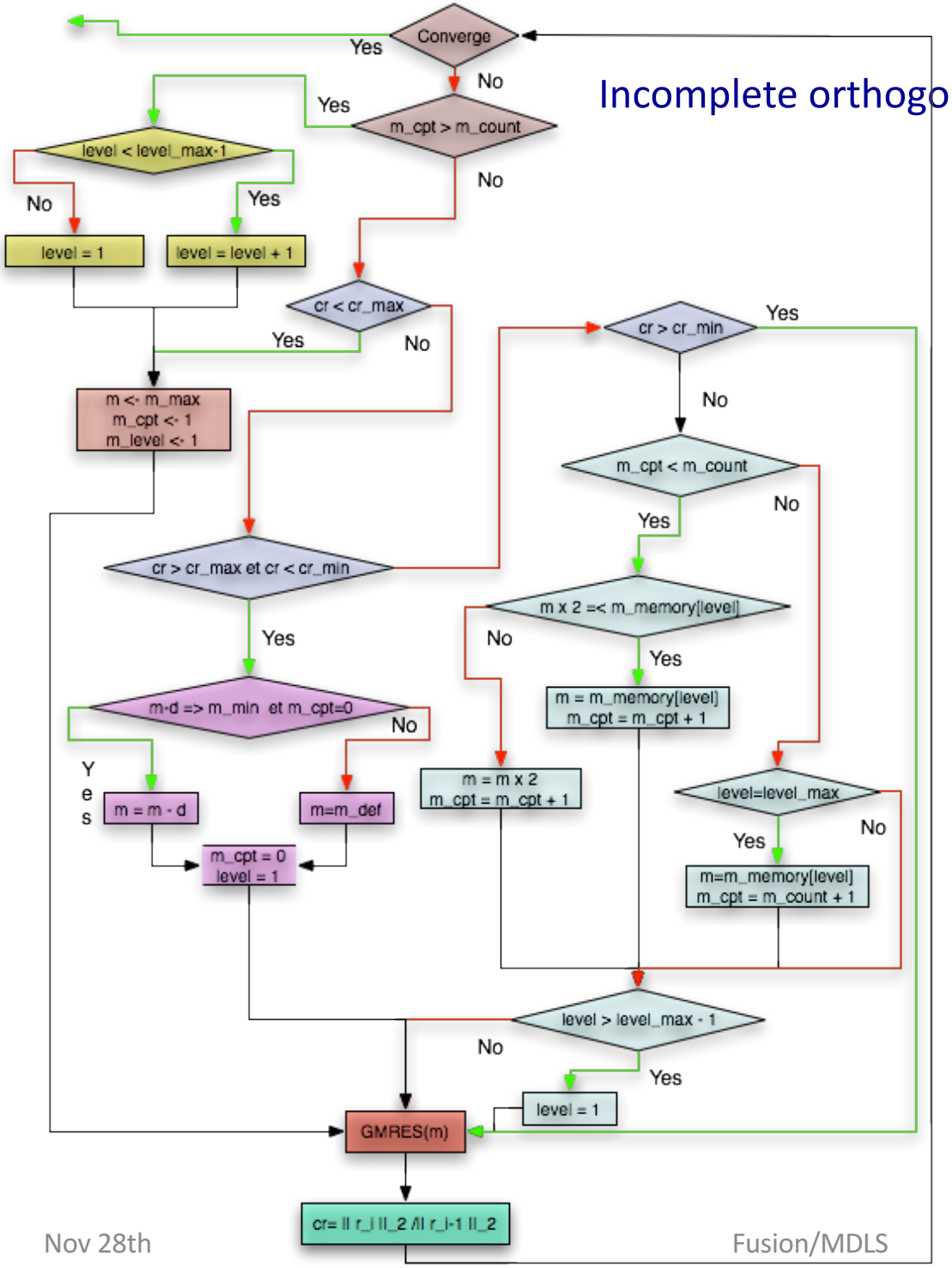


- Evaluate iteration costs in time vs. Convergence
- Decrease number of orthogonalized vectors q if ratio convergence/(time iteration) decrease

A complex heuristic-based algorithm : With respect to the variation of the residual between restarts, we change the number q of vectors concerned by the orthogonalization

Still, a lot of researches to achieve to optimize this algorithm.

Incomplete orthogonalization Auto-tuning Algorithm



q_{min}= minimum number of vector to orthogonalize

q_{max}= maximum number of vector to orthogonalize, typically = m the gmres subspace size

T_x = time of the x^{th} restart

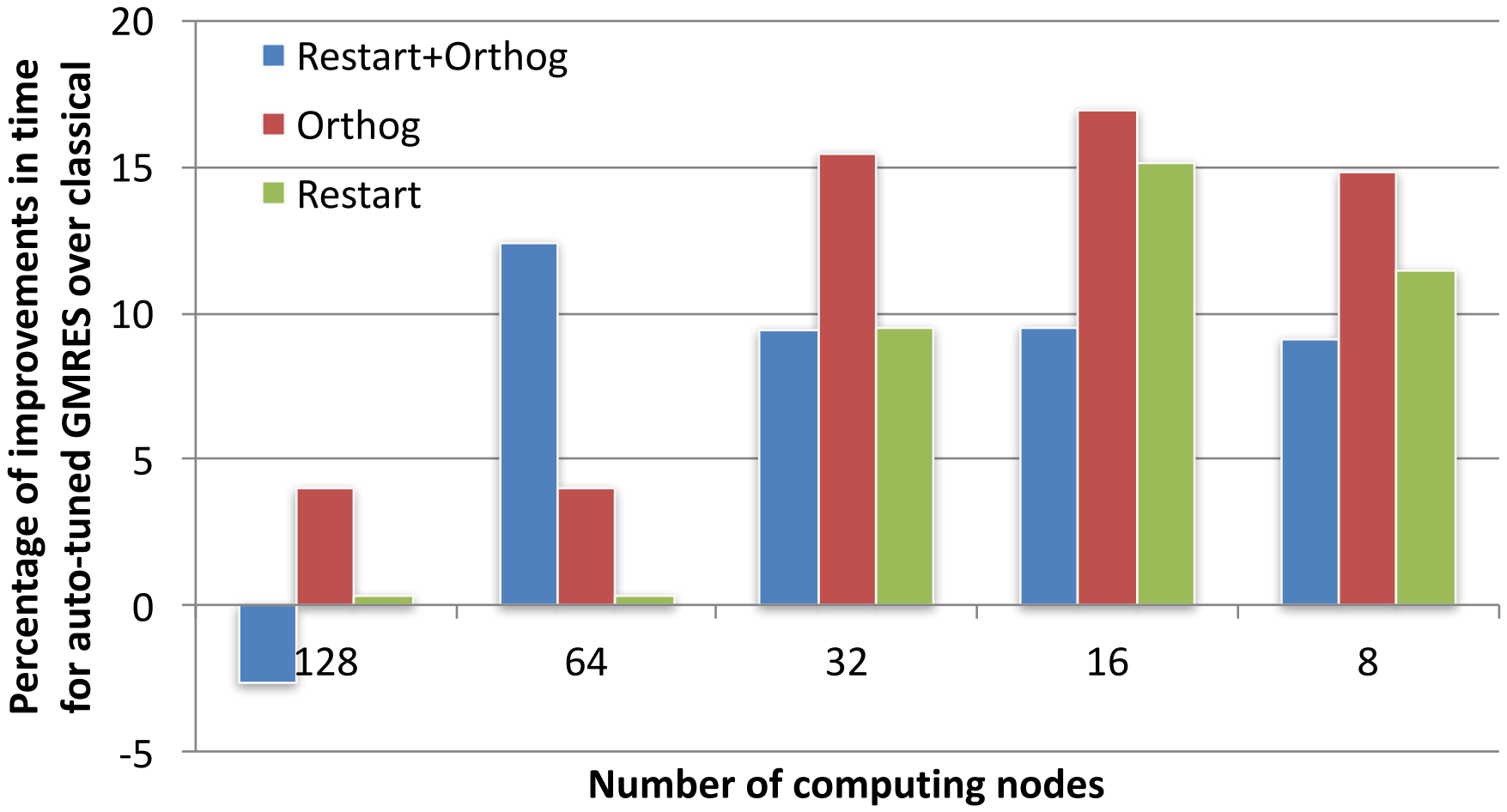
N_x = norm of the residual variation, equal to the norm of the duration of the x^{th} restart minus the duration of the $x\text{-}1^{th}$ restart

H_x = relative variation
$= N_x / T_x$

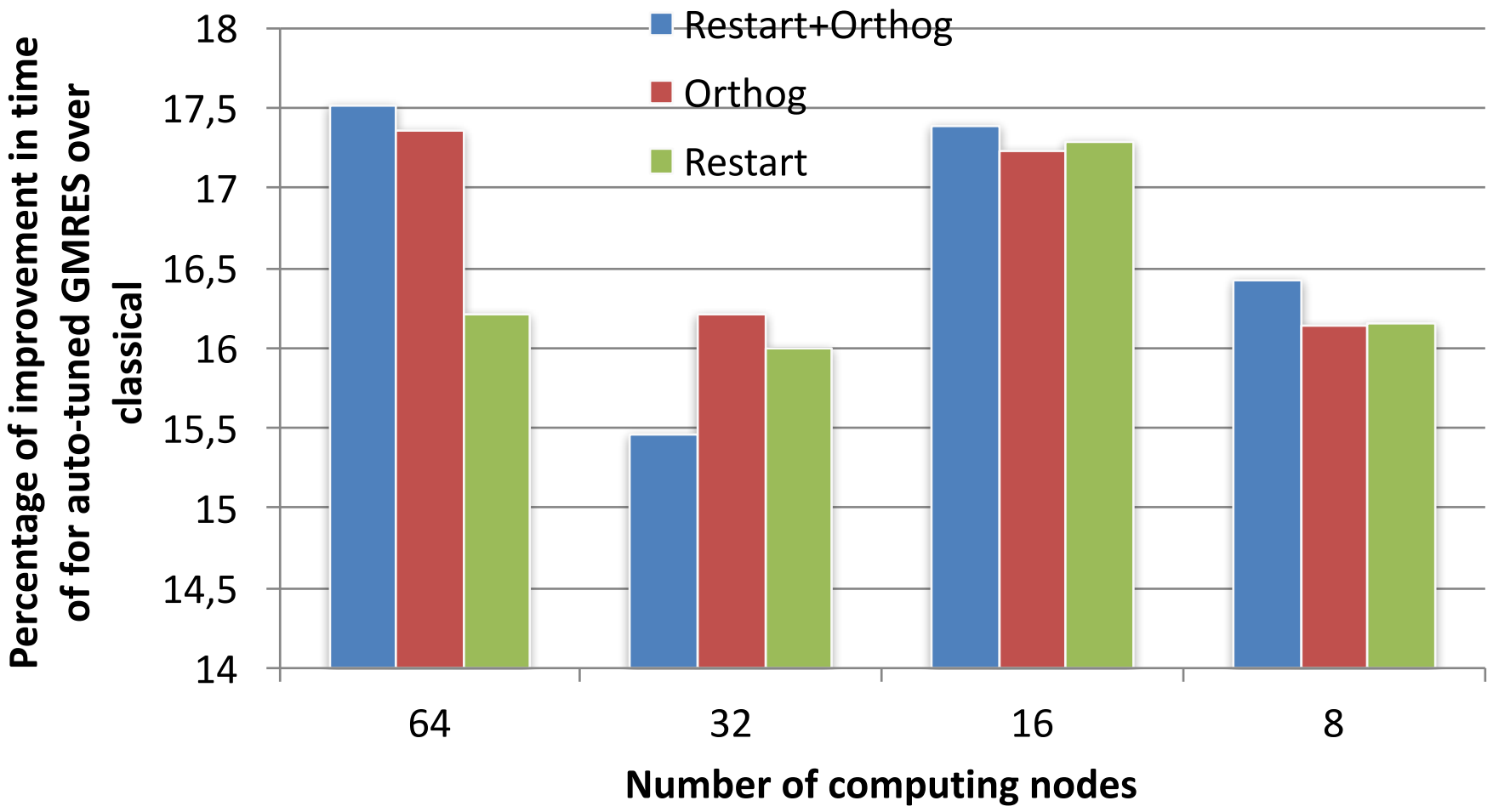
Heuristic = ratio of the relative variation between restart x and x-1, equal H_x / H_{x-1}

Results : Industrial Case (TOTAL)

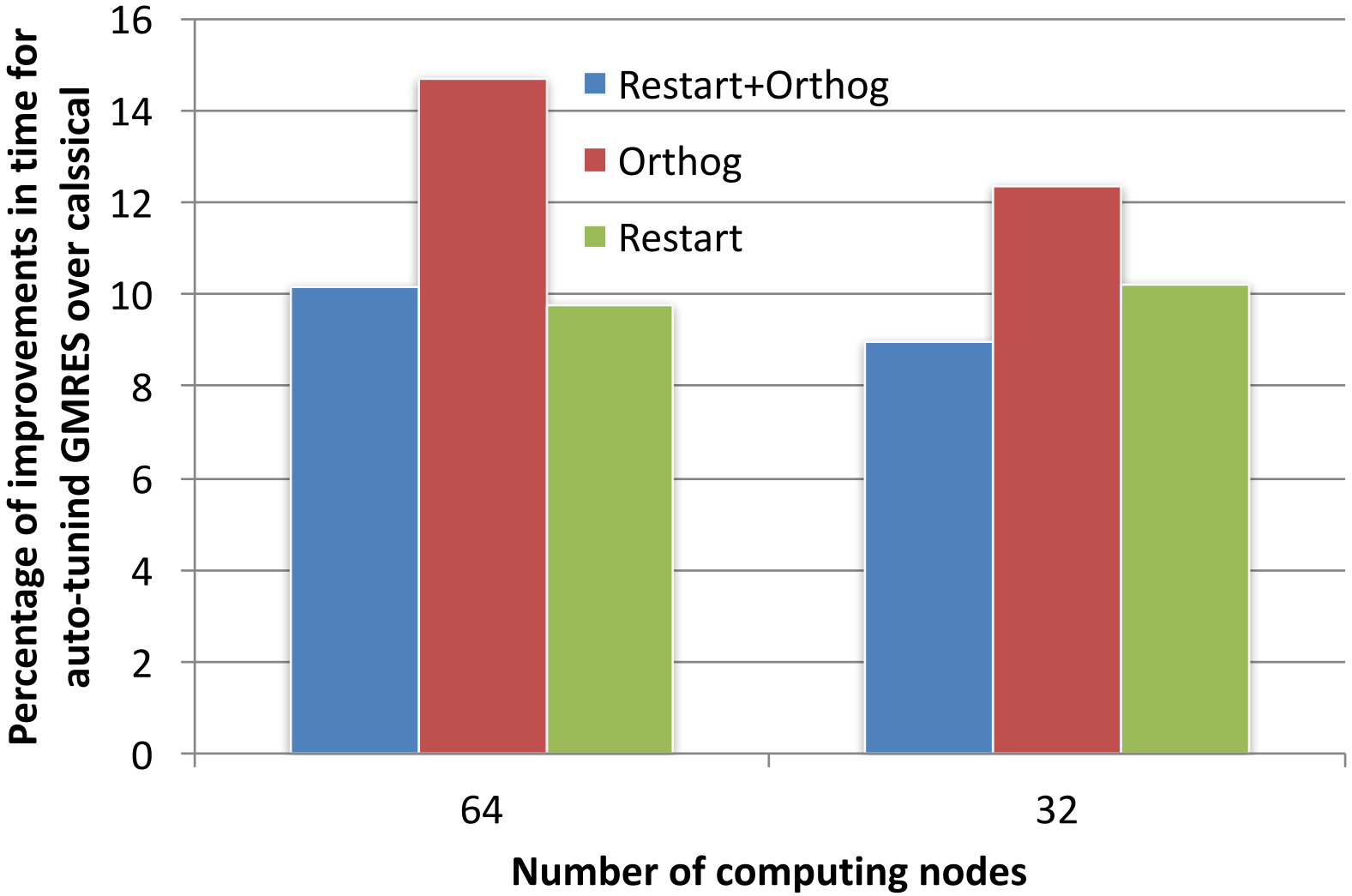
number of unknown = (119 x 119 x 115), 3Hz, m=10



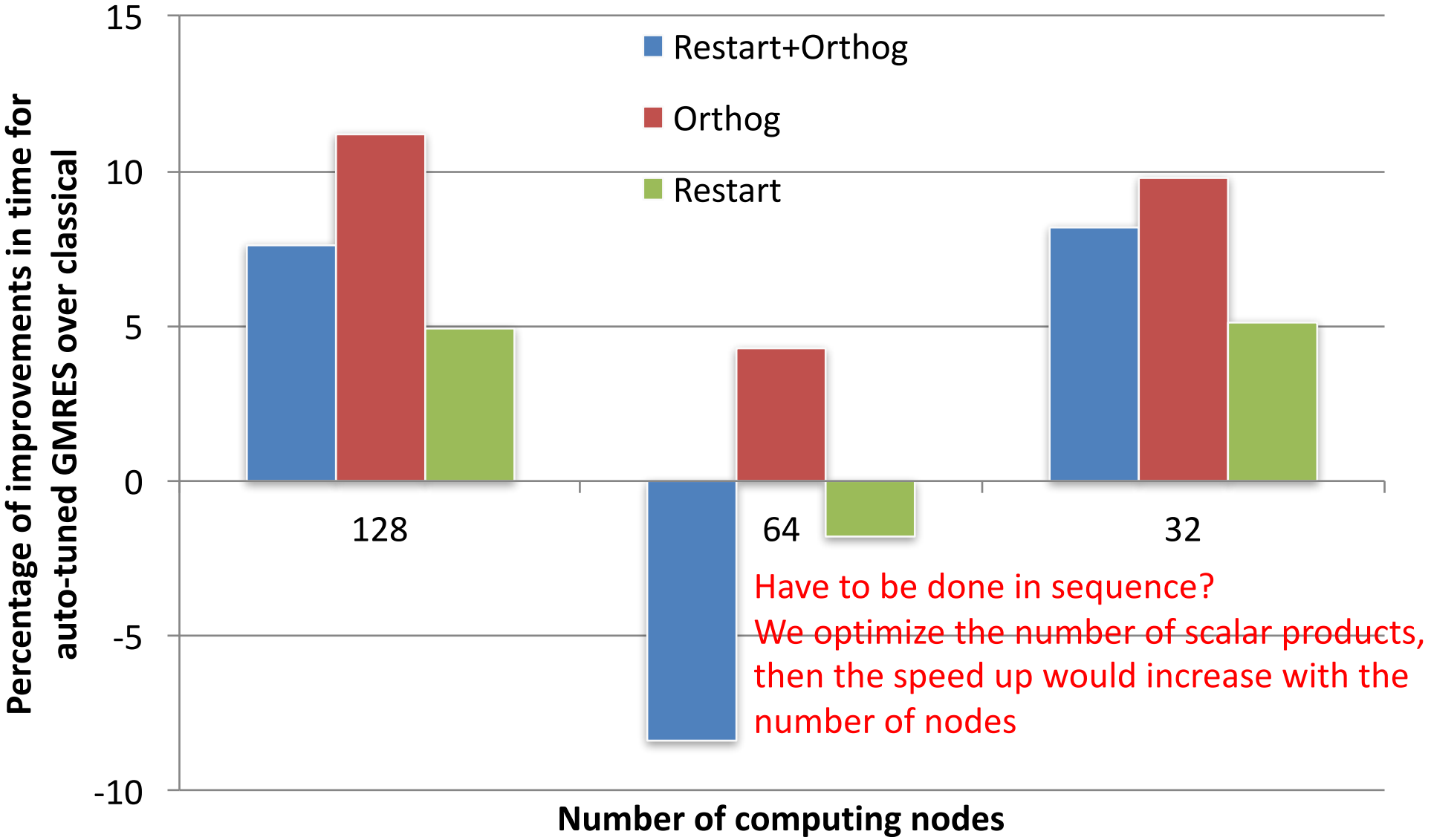
number of unknown = (119 x 119 x 115), 3 Hz, m=30



number of unknown = (183 x 183 x 191), 5 Hz, m=10



number of unknown = (335 x 327 x 383), 5Hz, m=30



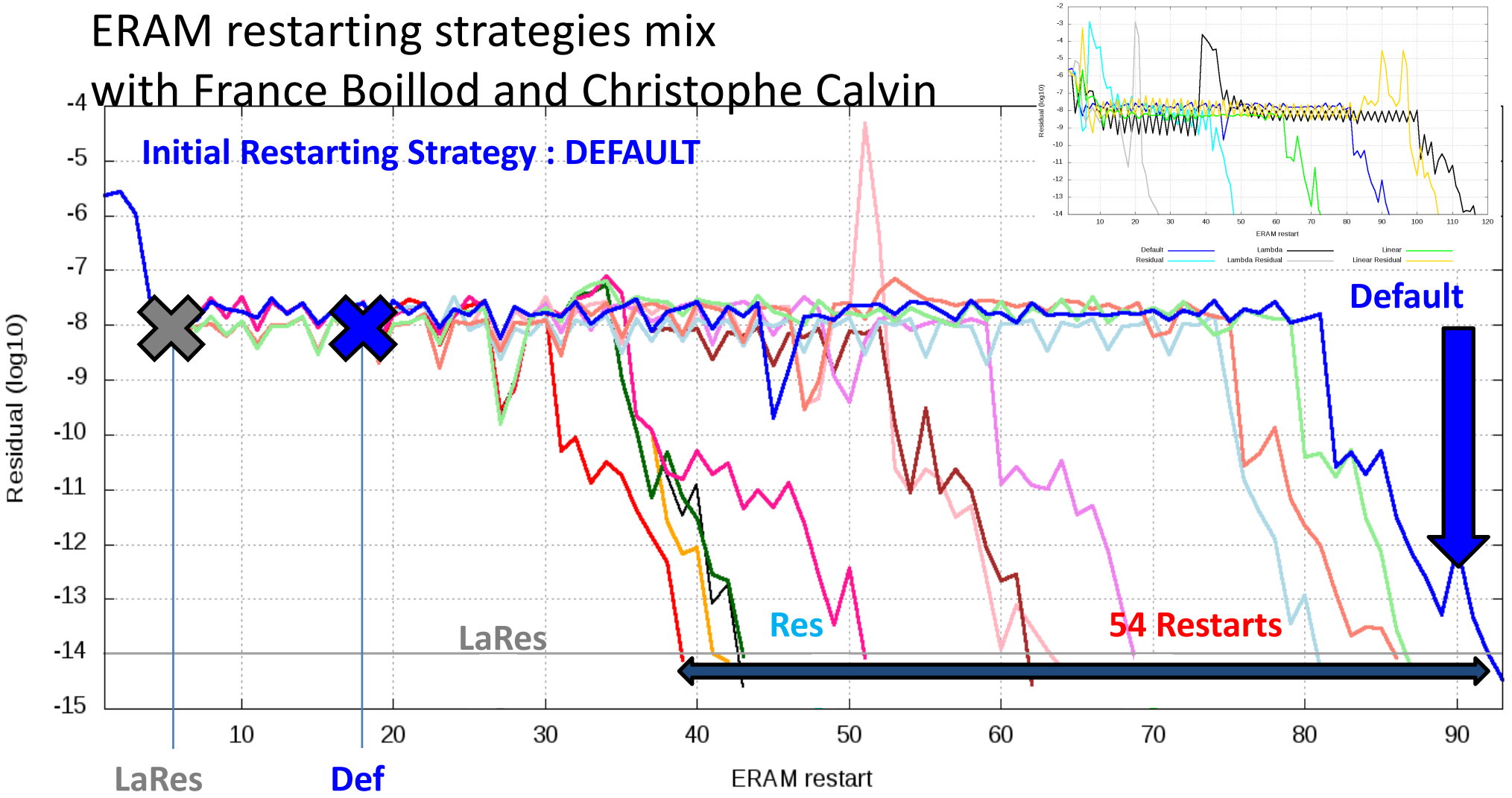
Have to be done in sequence?

We optimize the number of scalar products, then the speed up would increase with the number of nodes

Outline

- Introduction
- **Krylov subspace auto-tuned restarted methods**
  - Auto-tunings at runtime for Krylov methods
  - Subspace size
  - Incomplete orthogonalisation
  - **Restart strategies**
  - Sparse formats
  - Energy consumption
- Asynchronous Unite-and-Conquer methods
- Multilevel programming paradigm : Graph of components/PGAS
- What Intelligent Krylov methods for extreme computing?
- Conclusion

ERAM restarting strategies mix
with France Boillod and Christophe Calvin

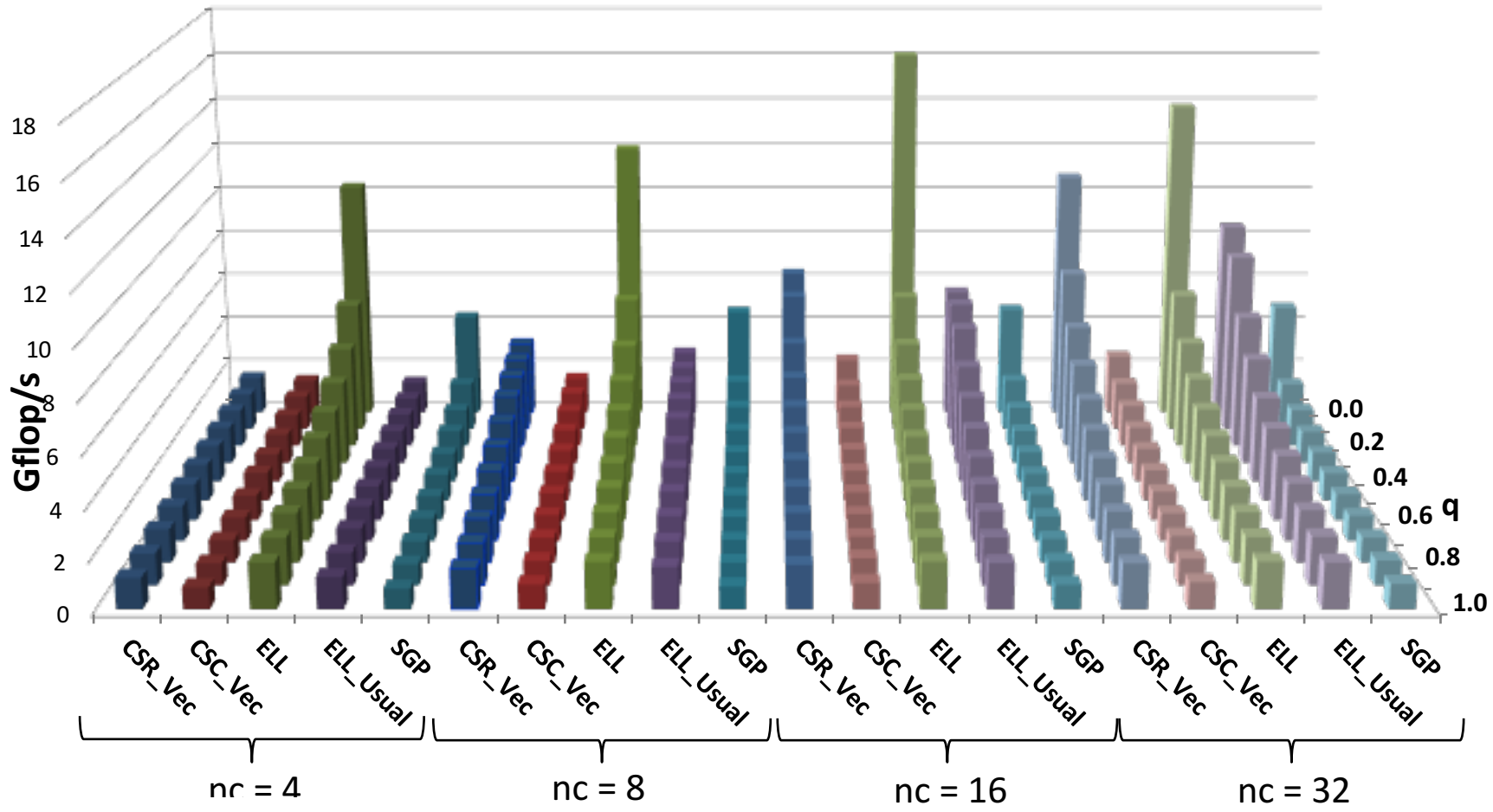


4 eigenpairs,
m=15, CGSr
Bayer04 Matrix
- N=20545
- nnz=85537
- 1 Intel i5-2430M

Outline

- Introduction
- **Krylov subspace auto-tuned restarted methods**
  - Auto-tunings at runtime for Krylov methods
  - Subspace size
  - Incomplete orthogonalisation
  - Restart strategies
  - **Sparse formats**
  - Energy consumption
- Asynchronous Unite-and-Conquer methods
- Multilevel programming paradigm : Graph of components/PGAS
- What Intelligent Krylov methods for extreme computing?
- Conclusion

Performance of SpMV

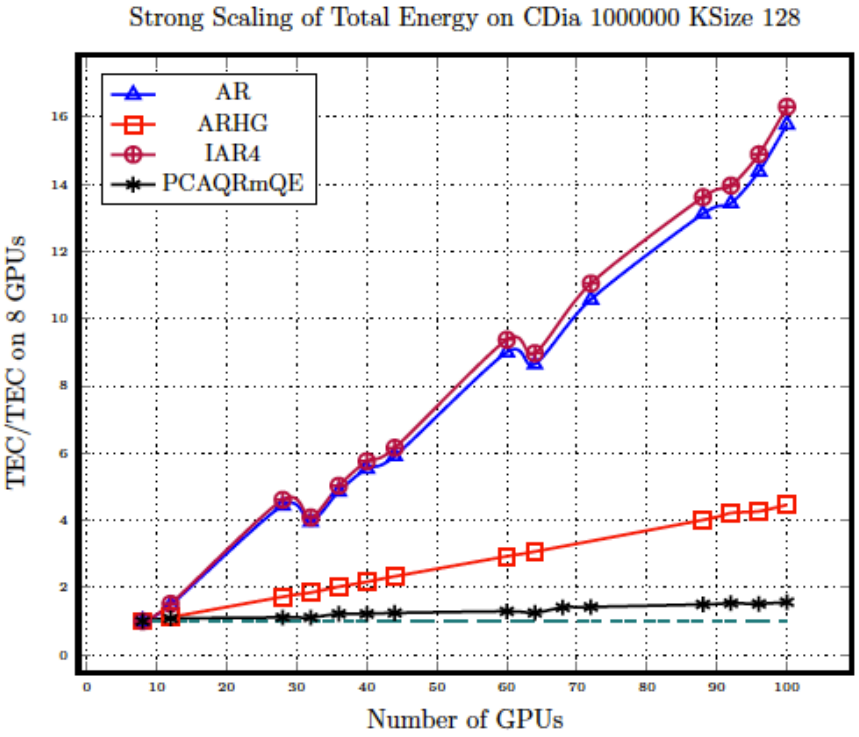


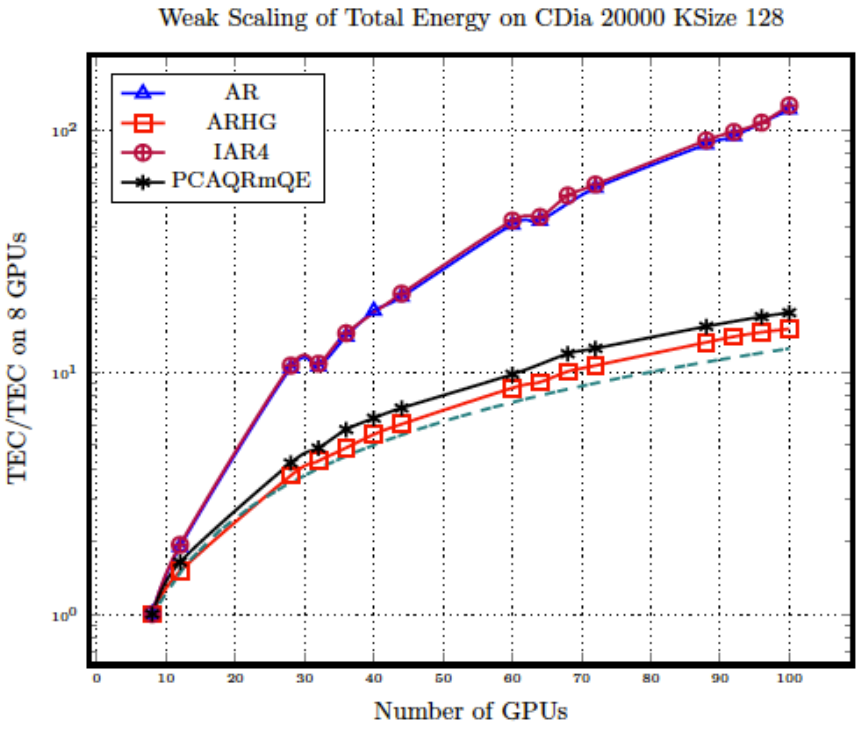
Outline

- Introduction
- **Krylov subspace auto-tuned restarted methods**
  - Auto-tunings at runtime for Krylov methods
  - Subspace size
  - Incomplete orthogonalisation
  - Restart strategies
  - Sparse formats
  - **Energy consumption**
- Asynchronous Unite-and-Conquer methods
- Multilevel programming paradigm : Graph of components/PGAS
- What Intelligent Krylov methods for extreme computing?
- Conclusion

Scalability of TEC

With Langshi Chen PhD, Lille 1 and MDLS, 201





- ArnoldiHG and PCAQRmQE have good scalability of TEC
- Communication is important to scalability of energy consumption.

Then, for a given method:

Several parameter have to be optimized at runtime :

- Subspace size
- Incomplete orthogonalisation
- Restart strategies
- Energy consumption
- Sparse format compression
- Orthogonalisation algorithm
- Preconditionners
- Mixed arithmetic
- Others ….

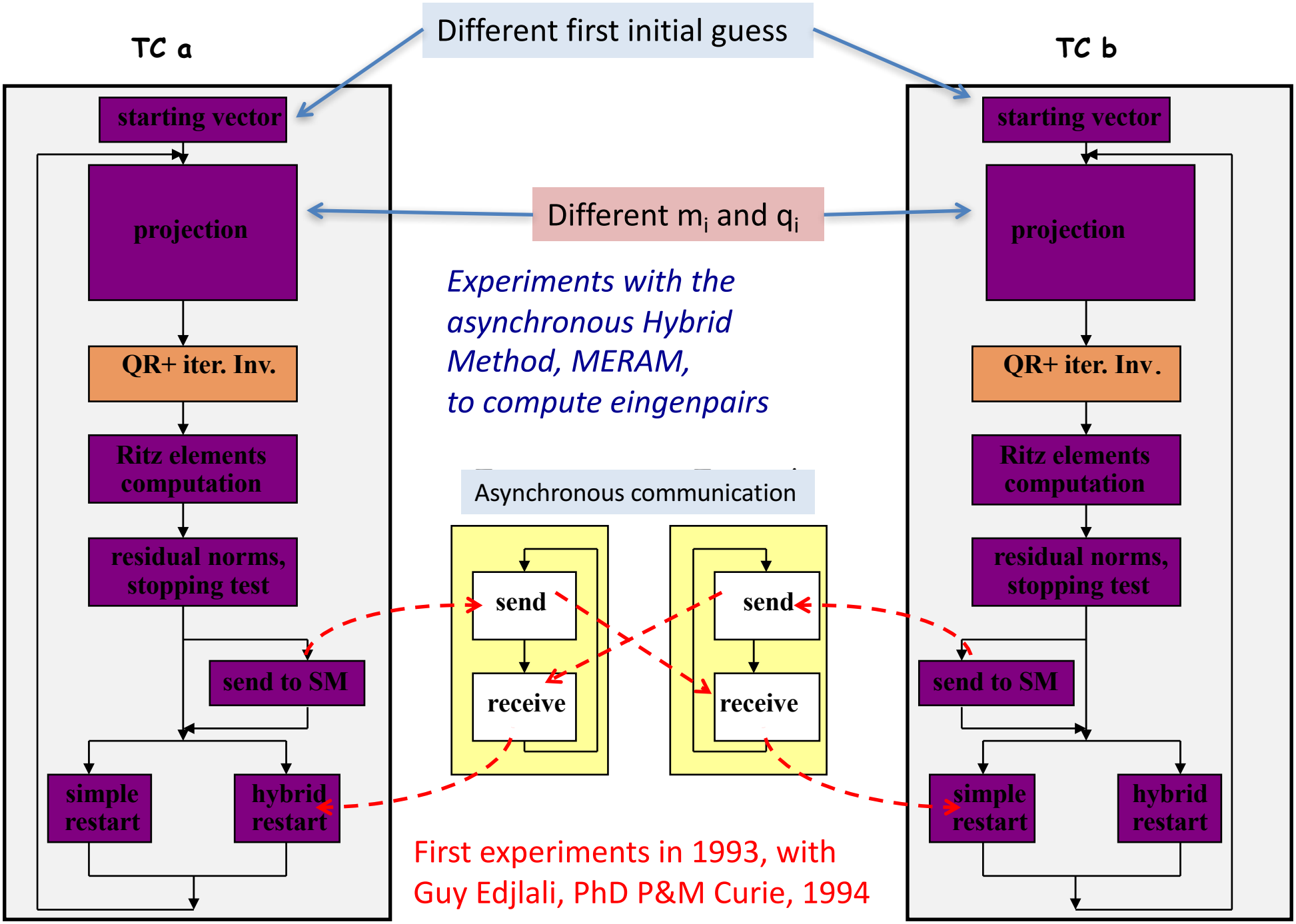
Nevertheless, we have to be able to evaluate the convergence, the stability and others importants criteria at runtime, and some learning may be introduced

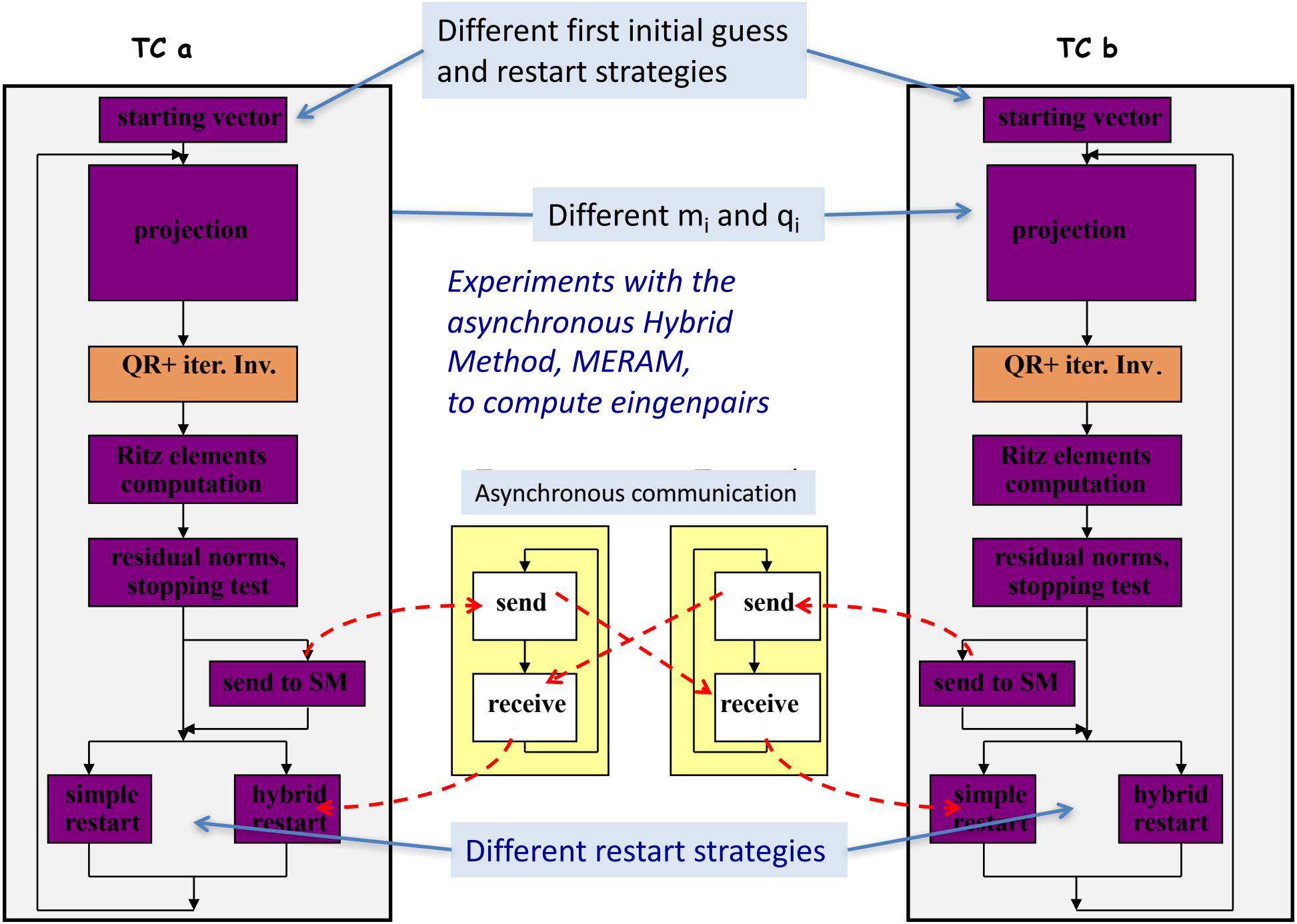
Deep learning technics may be used to learn and big analytics may help to compare with past experiments

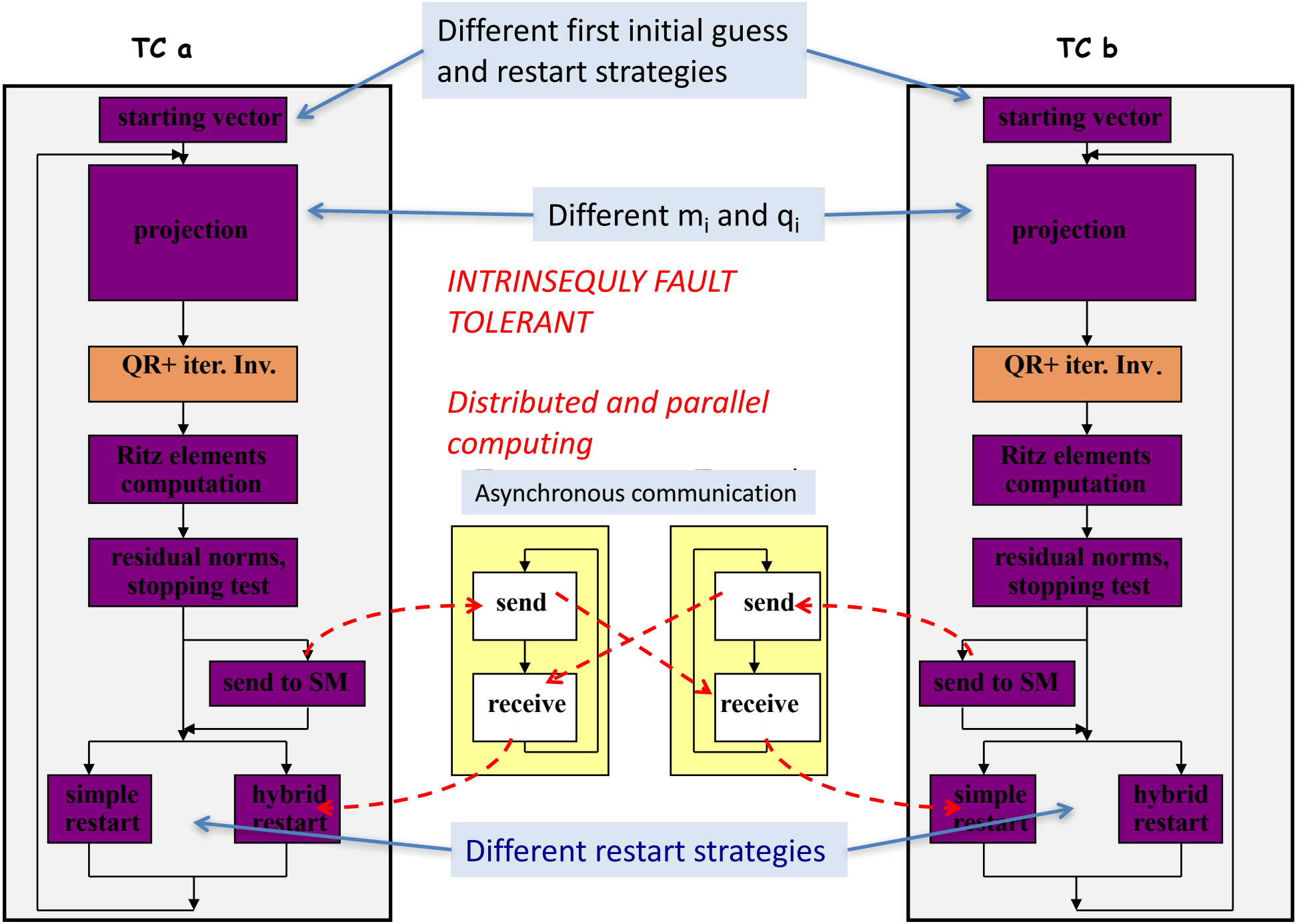
Unite-and-conquer methods would generate new potential problems, and information.

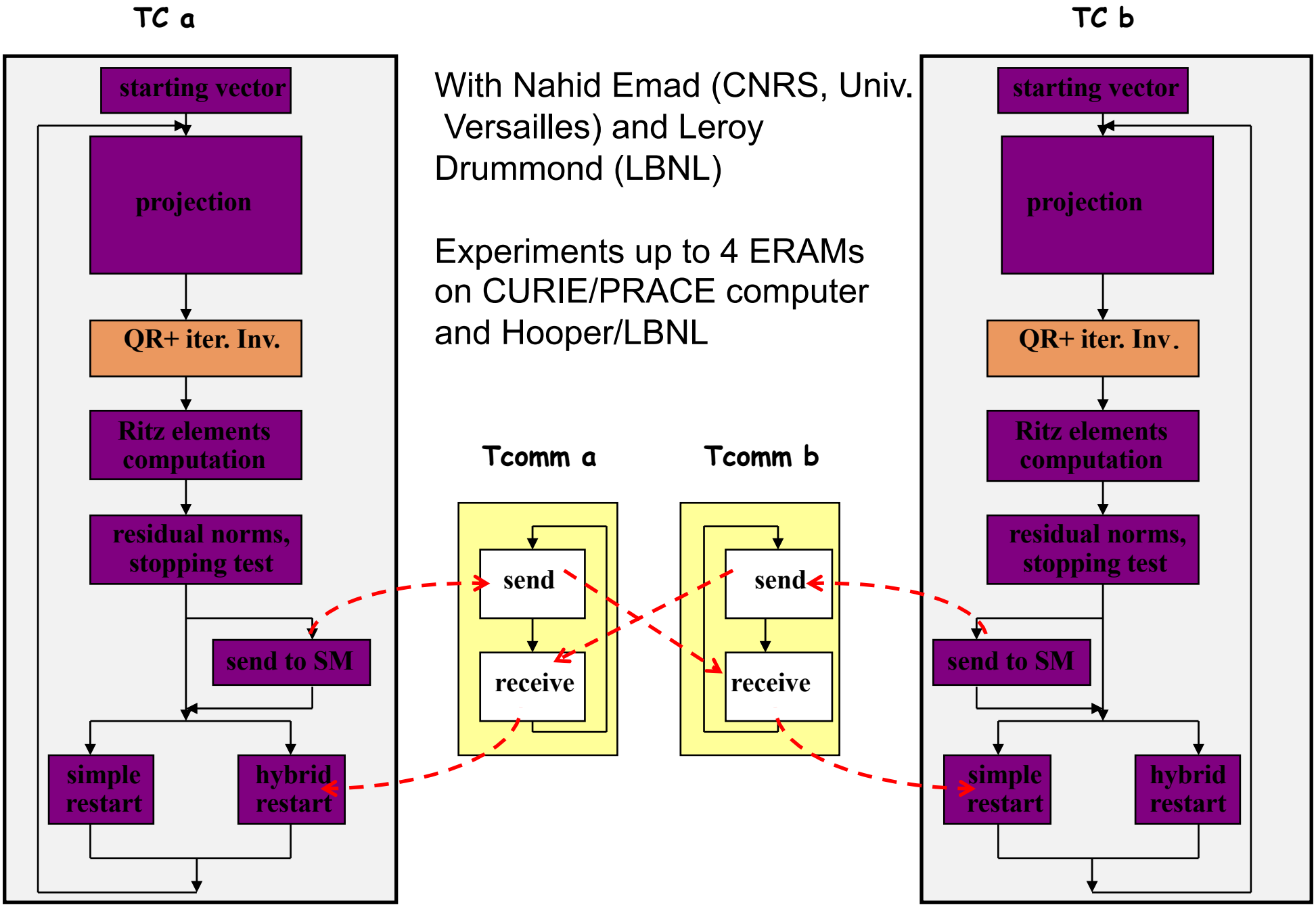
Outline

- Introduction
- Krylov subspace auto-tuned restarted methods
- **Asynchronous Unite-and-Conquer methods**
- Multilevel programming paradigm : Graph of components/PGAS
- What Intelligent Krylov methods for extreme computing?
- Conclusion


Different first initial guess
TC a
TC b
starting vector
projection
QR+ iter. Inv.
Ritz elements computation
residual norms, stopping test
send to SM
simple restart
hybrid restart
Different m_i and q_i
Experiments with the asynchronous Hybrid Method, MERAM, to compute eingenpairs
Asynchronous communication
send
receive
send
receive
starting vector
projection
QR+ iter. Inv.
Ritz elements computation
residual norms, stopping test
send to SM
simple restart
hybrid restart
First experiments in 1993, with Guy Edjlali, PhD P&M Curie, 1994


TC a
Different first initial guess and restart strategies
TC b
starting vector
projection
QR+ iter. Inv.
Ritz elements computation
residual norms, stopping test
send to SM
simple restart
hybrid restart
Different m_i and q_i
Experiments with the asynchronous Hybrid Method, MERAM, to compute eingenpairs
Asynchronous communication
send
receive
send
receive
starting vector
projection
QR+ iter. Inv.
Ritz elements computation
residual norms, stopping test
send to SM
simple restart
hybrid restart
Different restart strategies


TC a
TC b
Different first initial guess and restart strategies
starting vector
projection
QR+ iter. Inv.
Ritz elements computation
residual norms, stopping test
send to SM
simple restart
hybrid restart
Different m_i and q_i
INTRINSEQULY FAULT TOLERANT
Distributed and parallel computing
Asynchronous communication
send
receive
send
receive
starting vector
projection
QR+ iter. Inv.
Ritz elements computation
residual norms, stopping test
send to SM
simple restart
hybrid restart
Different restart strategies



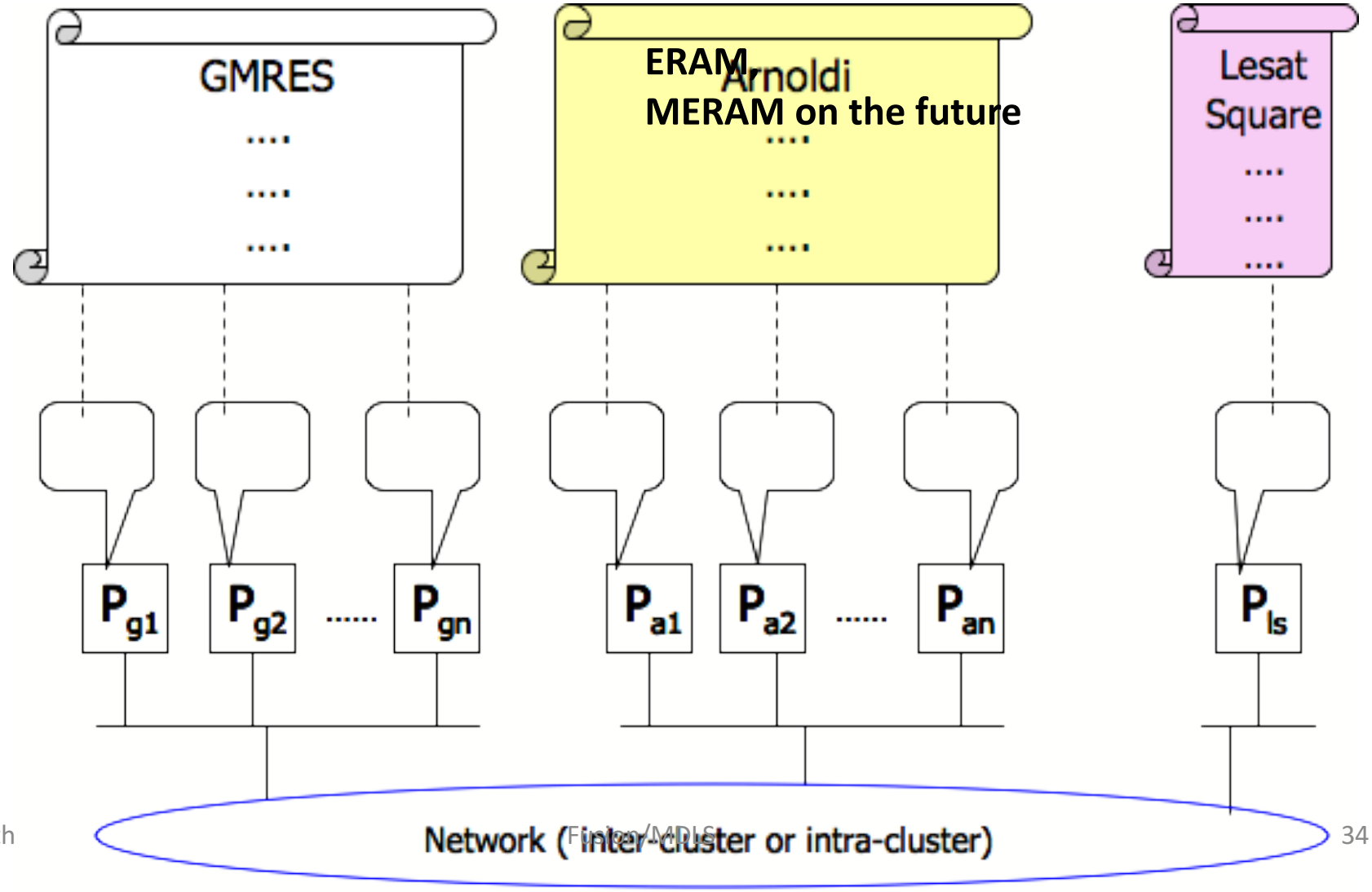
With Nahid Emad (CNRS, Univ. Versailles) and Leroy Drummond (LBNL)

Experiments up to 4 ERAMs on CURIE/PRACE computer and Hooper/LBNL

Asynchronous Iterative Restarted Methods

Collaboration with He Haiwu and Guy Bergère (U. Lille 1, CNRS)
and Ye Zhang (Hohai Univ. Nanjing) , Salim Nahi (Maison de la simulation),
and Pierre-Yves Aquilenti (TOTAL), Xinzhe Wu (Lille 1 and MDLS)

Experiments on several supercomputers, or networks of clusters/supercomputers)
We are beginning experiment on Tianhe 2 and planed some on the #1 in Japan

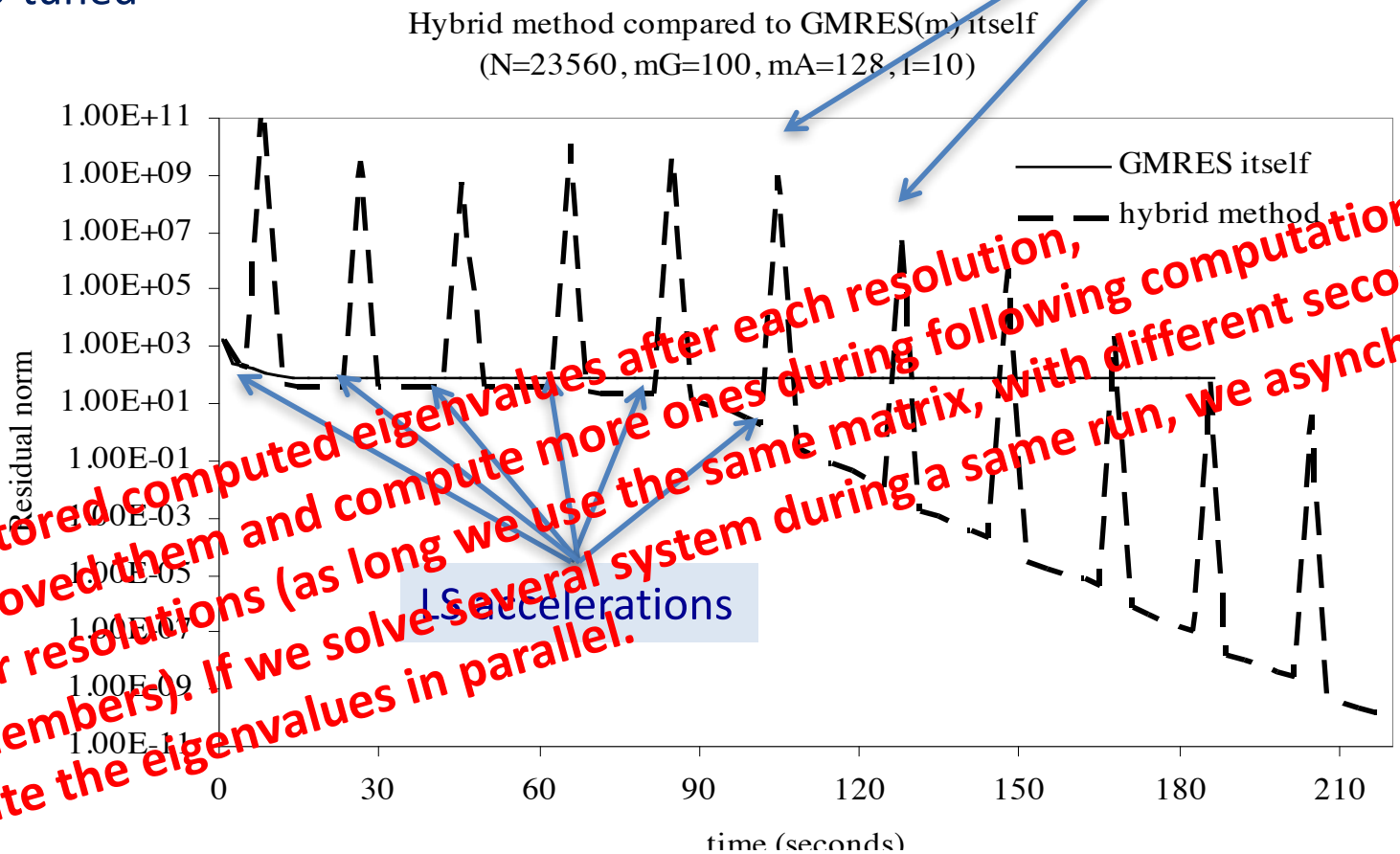


Numeric Results and Analysis

advantage over GMRES (difficult convergence case)

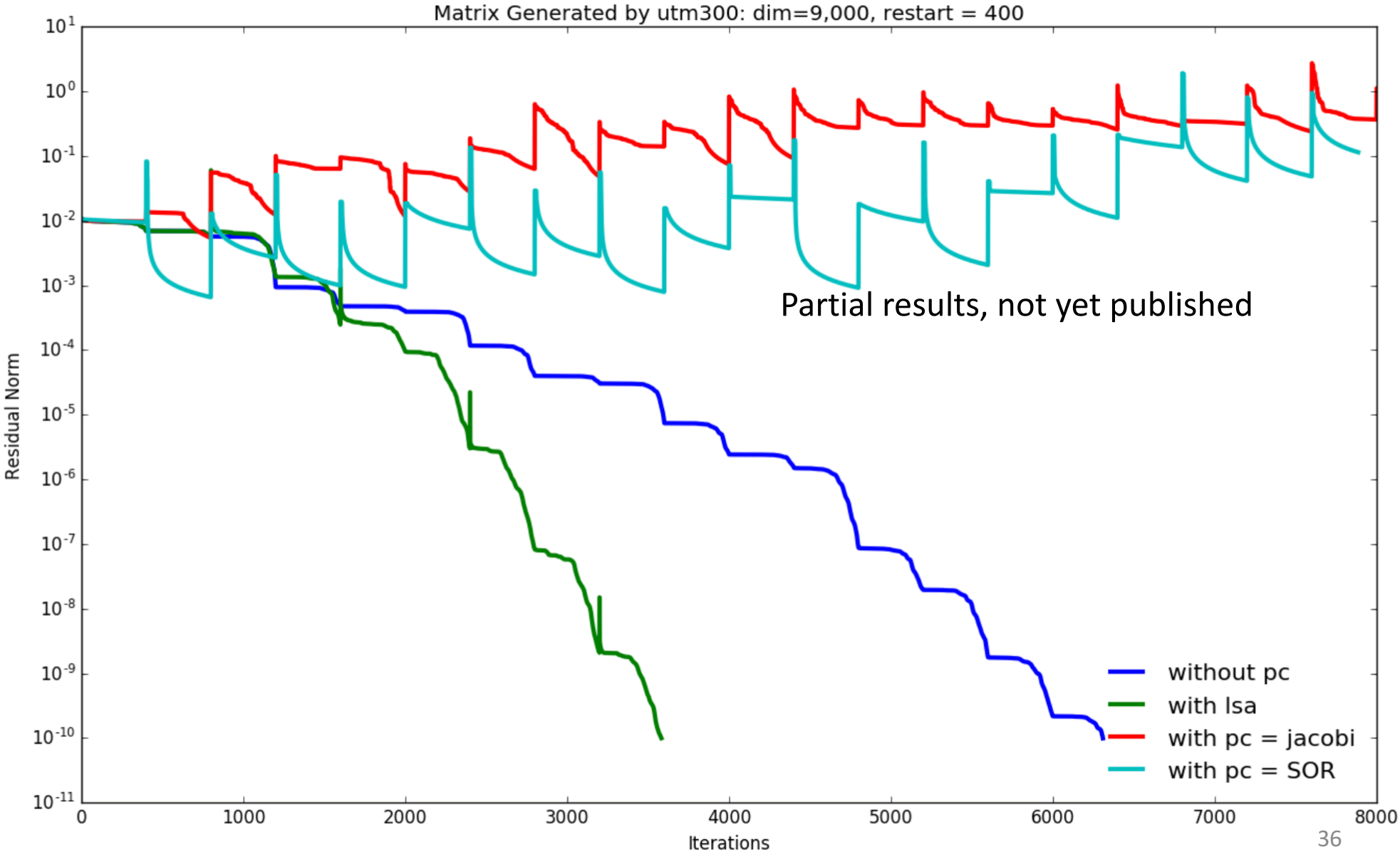
Degrees of polynomial,
number of iterations between two LS “accelerations”,….
may be auto-tuned

Well- known behaviours



We may stored computed eigenvalues after each resolution, and improved them and compute more ones during following computation for other resolutions (as long we use the same matrix, with different second hand members). If we solve several system during a same run, we asynchronously compute the eigenvalues in parallel.

Last results, last week on a Bull supercomputer in Reims (Romeo), with Xinzhe Wu
We also change the number of processor to compute ERAM and GMRES at runtime



Then

Criteria of each methods have to be tuned at runtime

The information, and learning, extracted on each method have to be exchanged asynchronously with the others in order to improve all the global convergence and numerical behaviors

Each run would generate information which may be used for following computation (ex : approximated eigenvalues for GMRES-ERAM/LS, ….)

But, existing programming paradigms are not well-adapted

And end-users/scientists may help with their expertise

A new generation of numerical methods has to be invented!

Outline

- Introduction
- Krylov subspace auto-tuned restarted methods
- Asynchronous Unite-and-Conquere methods
- **Multilevel programming paradigm : Graph of components/PGAS**
- What Intelligent Krylov methods for extreme computing?
- Conclusion

Toward graph of parallel tasks/components

*Supercomputers as distributed **and** parallel platforms*

- 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. unite-and-conquer 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.
- We have to allow end-user to give expertise

Some elements on YML (since 2000)

- 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]

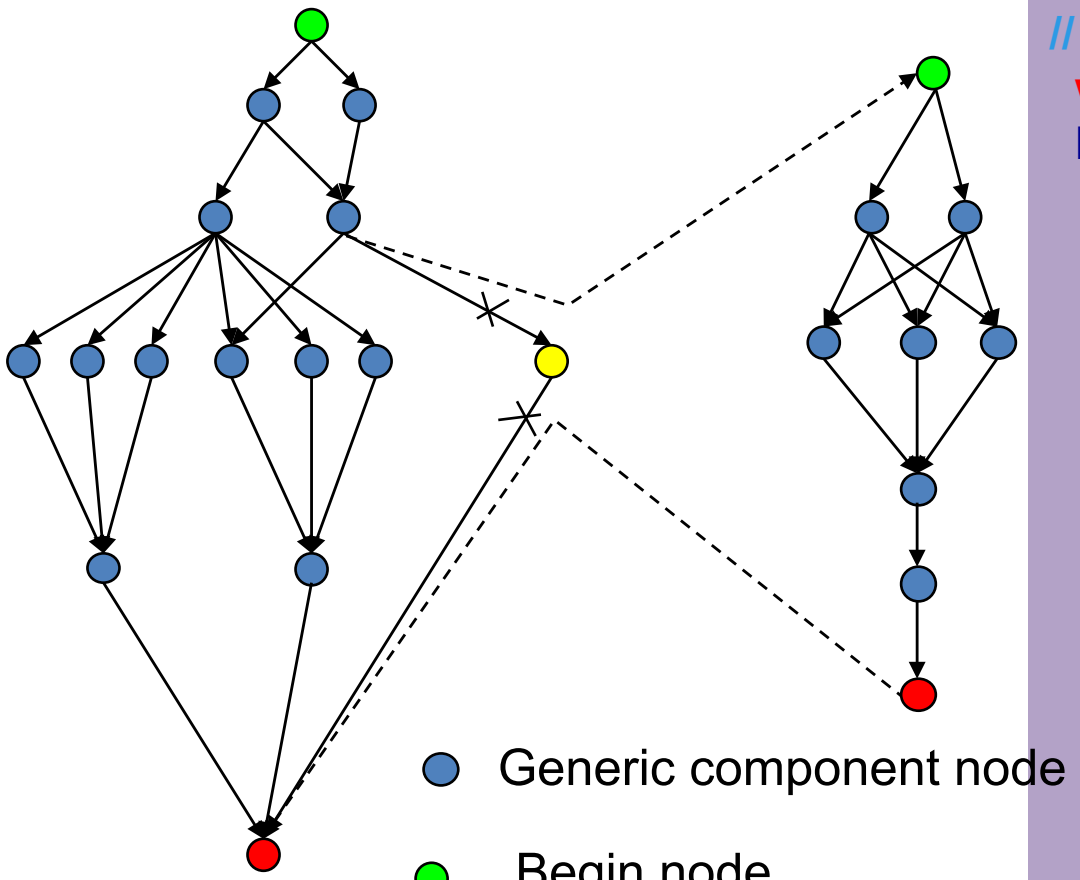
- Components approach
  - Defined in XML
  - **Three types** : Abstract, Implementation (in FORTRAN, C or C++;XMP,..), Graph (Parallelism)
  - Reuse and Optimized

- 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 France , Belgium, Ireland, Japan (T2K, K, FX10), China, Tunisia, USA (LBNL, TOTAL-Houston).

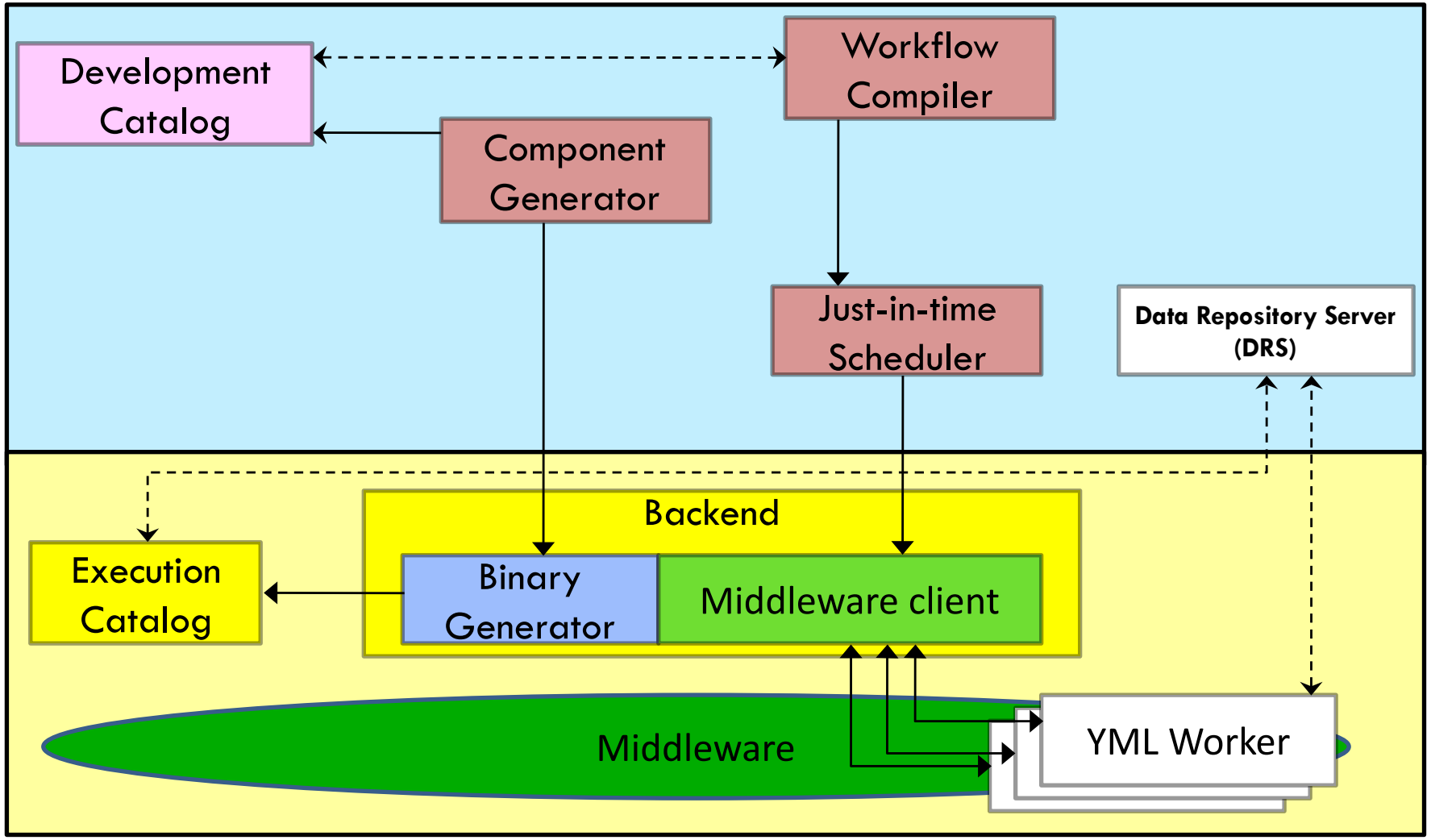
- Experiment on both supercomutes, Grid (Gird5000) and P2P (100 PCs in Lille, 100 PC in Orsay, and 4 cluserts in Japon, launch from a SC INRIA boooth a few years ago)

Graph (n dimensions) of components/tasks) YML




```
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
```

YML Architecture



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>
```

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>
```

End users may add some expertise here (we'll see 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>
```

Several possible implementation components for each abstract one, using different languages

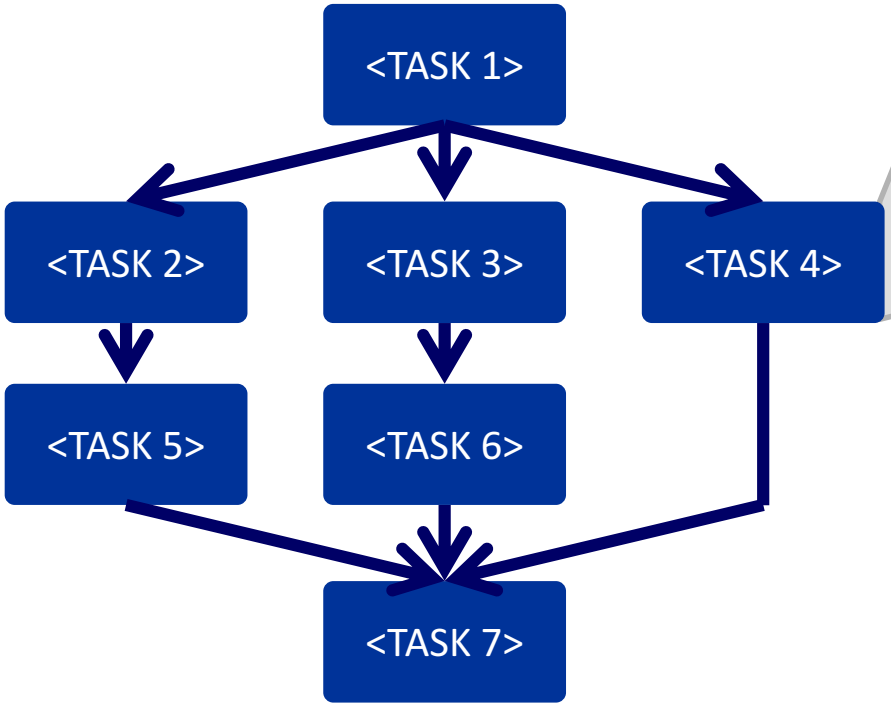
Graph component of Block Gauss-Jordan Method

```
<?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
```

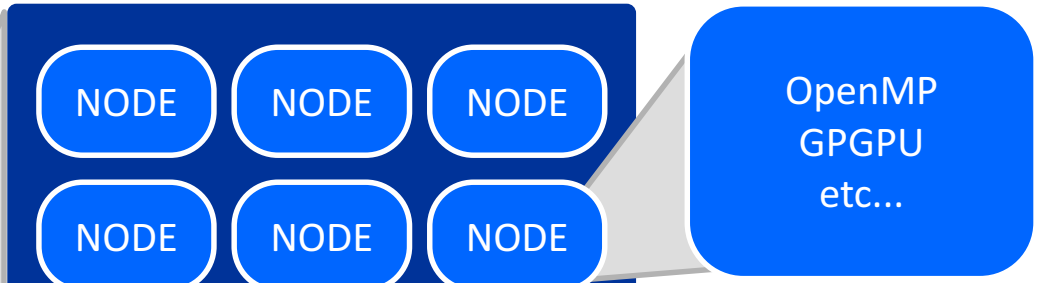
```
#Step3
            par( i:= 0;blockcount - 1)
            do
                  if (i neq k) then
                        if (k neq blockcount - 1) then
                        #step 3.1
                              par (j:=k + 1;blockcount - 1)
                              do
                                    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>
```

Multi-Level Parallelism Integration: YML-XMP

N dimension graphs available



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

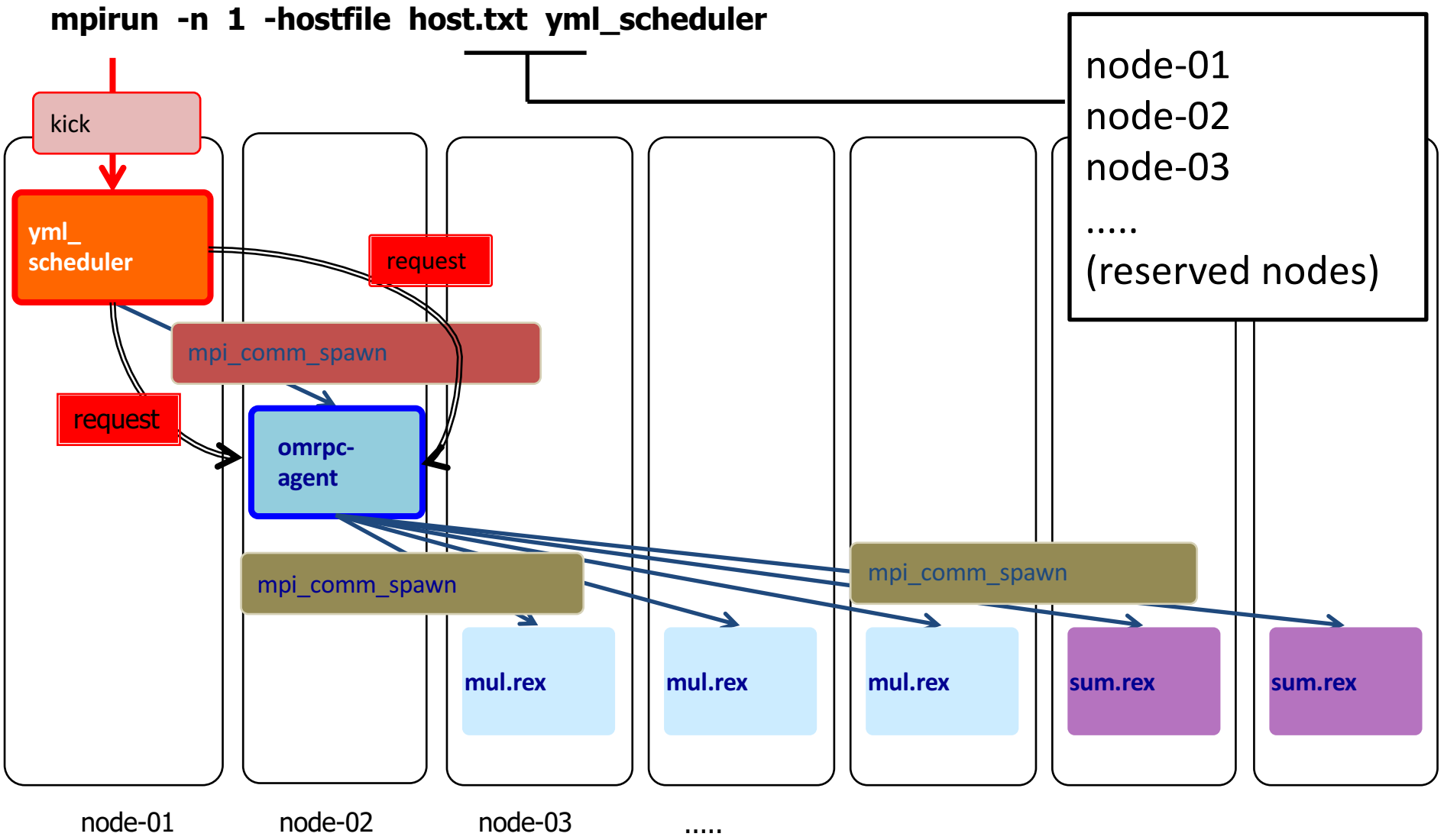



```
for(i=0;i<n;i++){
    for(j=0;j<n;j++){
       tmp[i][j]=0.0;
#pragma xmp loop (k) on t(k)
         for(k=0;k<n;k++){
            tmp[i][j]+=(m1[i][k]*m2[k][j]);
}}}
#pragma xmp reduction (+:tmp)
```

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

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



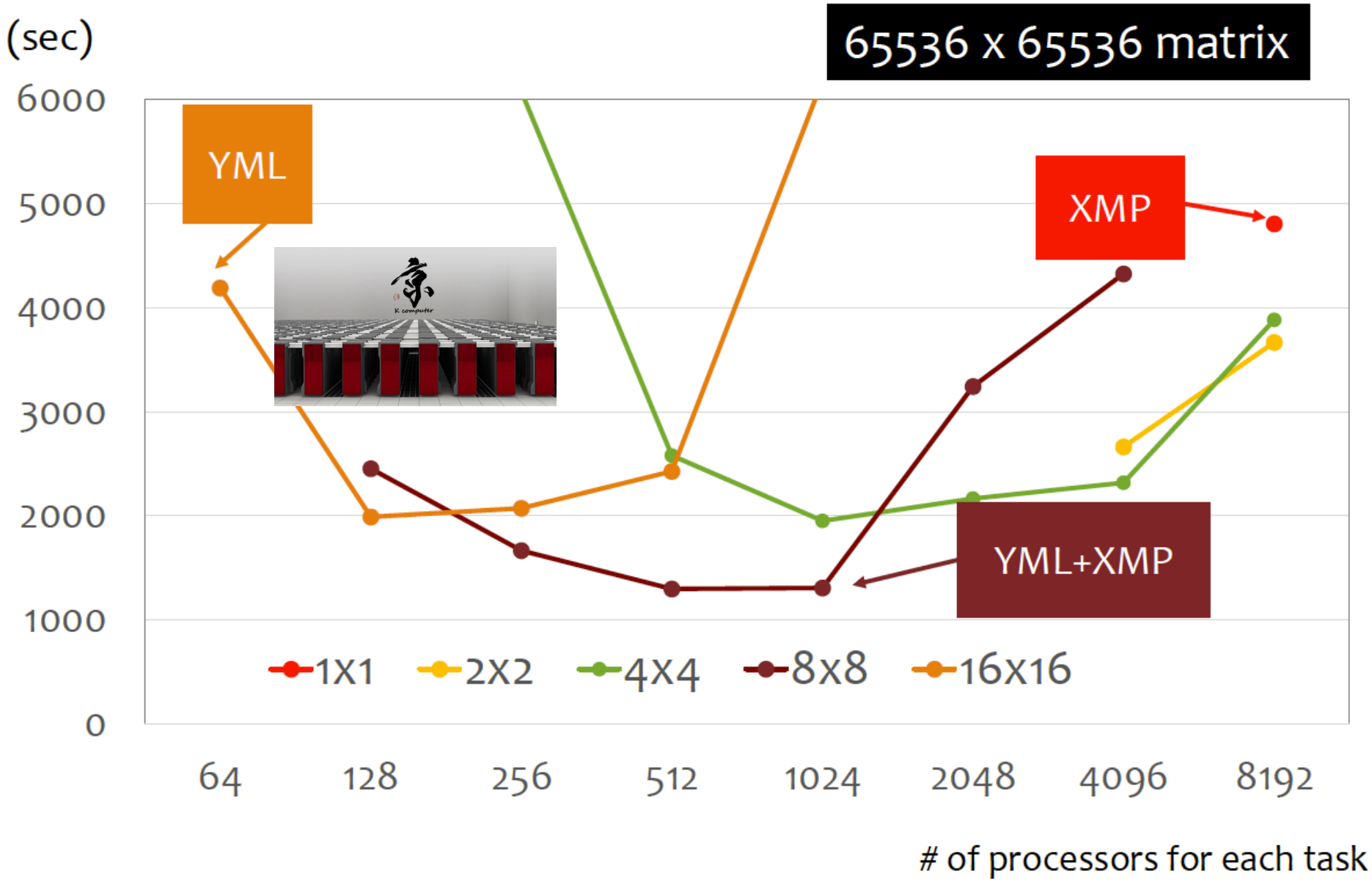
Implementation Component Extension

- 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>
```

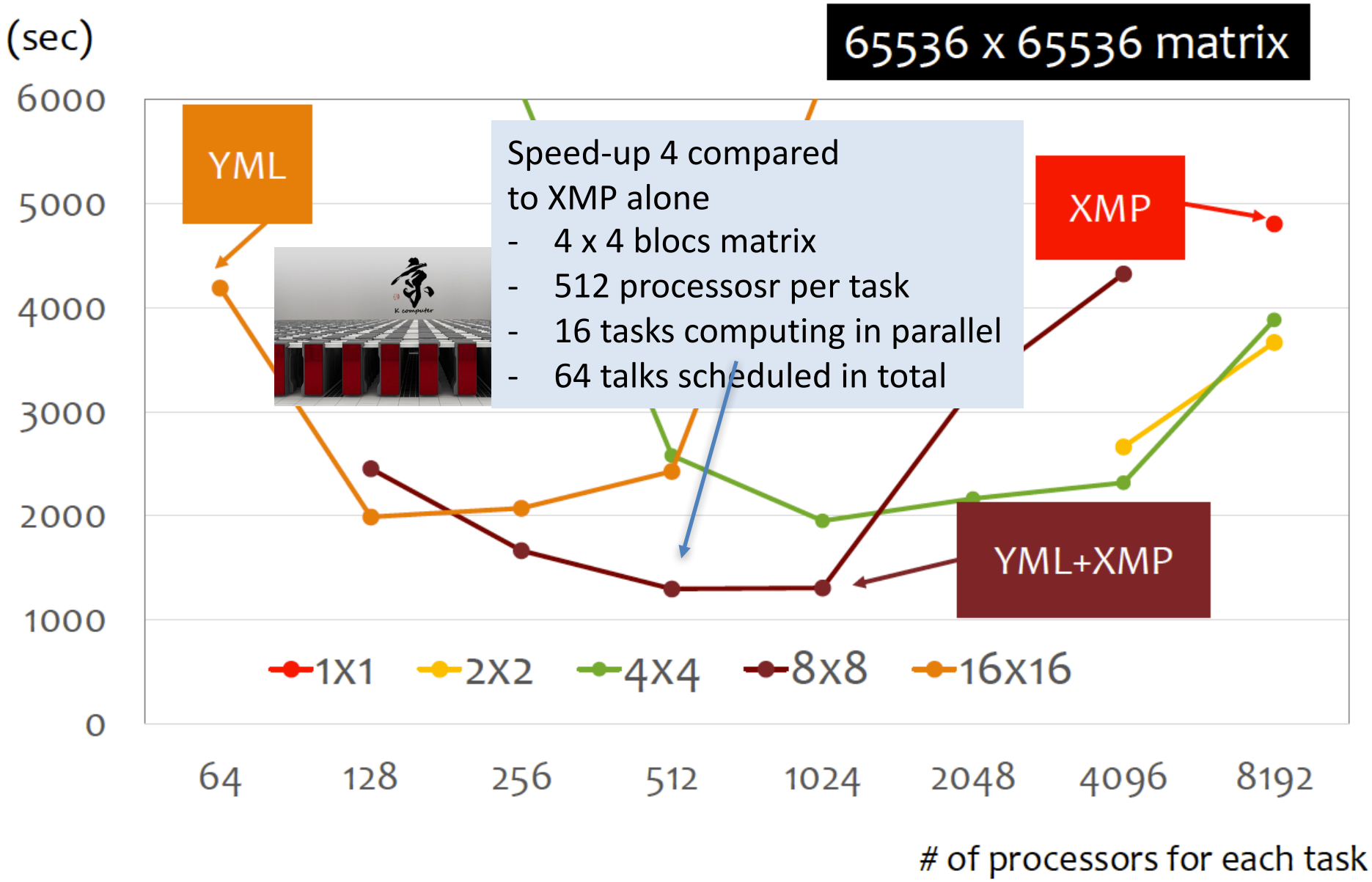
Information for XMP

Experiments (2) BGJ on K-Computer



Slide written by Miwako TSUJI, RIKEN/AICS

Experiments (2) BGJ on K-Computer



Slide written by Miwako TSUJI, RIKEN/AICS

Then,

Such programming paradigm (with some scheduled adaptations) is well-adapted for multi-level programming and unite-and-conquer methods

We may also consider the I/O and other data movements using some software as AIDOS (and others works done with DDN and TOTAL) as we have both the data and the control flow Graphs which allow data migration anticipations

We have a fault tolerent version of YML (talk at a SC15 workshop) developed mainly at AICS

For the same “task”, we have different components and the end-users may give some expertise which will may be used at any level of the software stack and at runtime by the scheduler, the middleware and others….

We may use this to allow the end-users to give expertise for future numerical methods

Outline

- Introduction
- Krylov subspace auto-tuned restarted methods
- Asynchronous Unite-and-Conquer methods
- Multilevel programming paradigm : Graph of components/PGAS
- **What Intelligent Krylov methods for extreme computing?**
- Conclusion

Interface between different languages

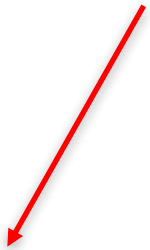
We already saw that :

```
<?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>
```

Implementation Component and smart-tuning associated to a language and an implemntation

- Range of parameters to tuned
- Expertise from end-users
- Learning

LL(1) languages may allow end-users to give expertise



```
<?xml version="1.0"?>
<component type="impl" name=   " GMRES_Tuned " abstract=   " GMRES_Tuned"
      description= "Example">
      <range m = {15,100} />
      <Algo_tuning = is method_1 in libX if size larger than 1000, is method_2 otherwise />
      <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>
```

Abstract Component associated with the method

```
<?xml version="1.0" ?>
<component type="abstract"    name= »GMRES_Tuned"  description=“restarted
    GMRES method" >
    <Smart_tunings>
    <param name=“m” type=“subspace_size” />
    <param name=“q”, type=“orthogonalization_parameter />
    </Smart_tunings>
    <params>
     <param    name="matrixA"  type="Matrix" mode="in" />
     <param    name="matrixV"    type="Matrix" mode="out" />
     <param    name="size"     type="integer" mode="in" />
    </params>
</component>
```

Future (allowing to change the graph at runtime depending of the result):

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

Abstract Component

```
<?xml version="1.0" ?>
<component type="abstract"    name= »GMRES_Tuned"  description=“restarted
    GMRES method" >
    <Smart_tunings>
    <param name=“m” type=“subspace_size” />
    <param name=“q”, type=“orthogonalization_parameter />
    </Smart_tunings>
    <params>
     <param    name="matrixA"  type="Matrix" mode="in" />
     <param    name="matrixV"    type="Matrix" mode="out" />
     <param    name="size"     type="integer" mode="in" />
    </params>
</component>
```

Future :

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

Implementation Component and smarrt-tuning

- Range of parameters to tuned
- Expertise from end-users
- Learning

LL(1) languages may allow end-users to give expertise

```
<?xml version="1.0"?>
<component type="impl" name=   " GMRES_Tuned " abstract=   " GMRES_Tuned"
        description= "Example">
        <range m = {15,100} />
        <Algo_tuning = is method_1 if size larger than 1000, is method_2 otherwise />
        <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>
```

Tuning for extreme computing

- Each parameters of each method may be auto-tuned,
- Each modification of a parameter in one method have to be analyse by the others methods
- We have to analyse the convergence, the efficiency of each iteration, the energy consumed, the accuracy, the stability variation,….

We propose high level programming paradigms which allow the end-user and/or the applied mathematician to give some expertise (range of the subspace size, dominant eigenvalue clustering, condition number, ….).

YML is such a programming langage (we have a virtual machine with tutorial and documentation, send me email : serge.petiton@univ-lille1.fr)

We have to use components and high level software strategies to propose tools and language for future numerical analysis methods, who will have to take decisison at runtime : to smart-tune but also tp chose methods and preconditionning, for example

Nevertheless, the important challenge is to propose new intelligent methods for such programming paradigms

Outline

- Introduction
- Krylov subspace auto-tuned restarted methods
- Asynchronous Unite-and-Conquere methods
- Multilevel programming paradigm : Graph of components/PGAS
- What Intelligent Krylov methods for extreme computing?
- **Conclusion**

Conclusion

- Auto-tunning at runtime and unite-and-conquer methods are important propositions to develop methods to announced extreme computing machine ; leading to “intelligent linear algebra”
- High level programming paradigms are required
- We propose both a framework based on multi-level programming and programming paradigm, and allowing end-users/scientists to give expertise
- Introducing learning into numerical methods would be an important improvements
- International collaboration are important to be able to evaluate and improve those approaches.

After HPC, High Performance Artificial Intelligence,
High-Performance Data Analytics,…..

Next step, more general :
High Performance Intelligent Computing