Latency-avoiding and Fault-Tolerant Algorithms for Dense Linear Algebra and Petascale Architectures

Julien Langou
Department of Mathematical Sciences
University of Colorado at Denver
julien.langou@cudenver.edu
(visiting at University of Tennessee)

ISC07: 06/28/07 – Algorithms for Petascale Computers – Are we ready?

- Focus attention on application developers need
  - (reason I am here)
- To know if we have the good algorithms in hand, we first need to speculate on what a petascale machine will look like.
  - The algorithm/library community needs input from the hardware community.
- We will probably need some help from kernel developers, scheduling community, compiler/language community.

Summary of 2006-2007 in Dense Linear Algebra

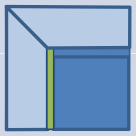
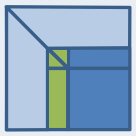
- **2007:** Two new algorithms in software available in LAPACK
  - **MRRR** (Multiple Relatively Robust Representation)
    - Symmetric Tridiagonal Eigenvalue Problem
    - $O(n^2)$ instead of $O(n^3)$
  - **THQR** (Aggressive early deflation)
    - Hessenberg Eigenvalue Problem
    - 5-10x faster

  (to come soon in ScaLAPACK)

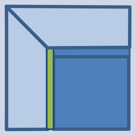
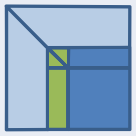
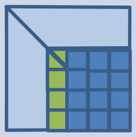
- **05-07:** New algorithms based on 2D partitionning:
  - **UTexas (van de Geijn):** SYRK, CHOL (multicore), LU, QR (out-of-core)
  - **UTennessee (Dongarra):** CHOL (multicore)
  - **HPC2N (Kågström)/IBM (Gustavson):** Chol (Distributed)
  - **UCBerkeley (Demmel)/INRIA(Grigori):** LU/QR (distributed)
  - **UCDenver (Langou):** LU/QR (distributed)

  A 3rd revolution for dense linear algebra?

A new generation of algorithms?

Algorithms follow hardware evolution along time.		
LINPACK (80's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (90's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations

A new generation of algorithms?

Algorithms follow hardware evolution along time.		
LINPACK (80's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (90's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
New Algorithms (00's) (multicore friendly)		Rely on - a DAG/scheduler - block data layout - some extra kernels

Those new algorithms
- have a very **low granularity**, they scale very well (multicore, petascale computing, …)
- **removes a lots of dependencies** among the tasks, (multicore, distributed computing)
- **avoid latency** (distributed computing, out-of-core)
- **rely on fast kernels**

Those new algorithms need new kernels and rely on efficient scheduling algorithms.

What will a petascale machine looks like.

(keep in mind that I am a math guy …)

Possible petascale machine	
1. Number of cores per nodes	10 – 100 cores
2. Performance per nodes	100 – 1,000 GFLOPs/sec
3. Number of nodes	1000-10,000 nodes
4. Latency inter-nodes	1 µsec
5. Bandwidth inter-nodes	10 Gb/sec
6. Memory per nodes	10 GB

What will a petascale machine looks like.

(aka what I learned during this conference.)

Possible petascale machine	
1. Number of cores per nodes	10 – 100 cores
2. Performance per nodes	100 – 1,000 GFLOPs/sec
3. Number of nodes	1000-10,000 nodes
4. Latency inter-nodes	1 µsec
5. Bandwidth inter-nodes	10 Gb/sec
6. Memory per nodes	10 GB

- Part I: First rule in linear algebra: Have an efficient DGEMM
  - Motivation in
    2. performance per node 5. bandwidth inter-nodes 6. memory per nodes
- Part II: Algorithms for multicore and latency avoiding algorithms for LU, QR …
  - Motivation in:
    1. Number of cores per node 2. performance per node 4. Latency inter-nodes
- Part III: Algorithms for fault tolerance
  - Motivation in:
    1. Number of cores per node 3. number of nodes

What will a petascale machine looks like.

(aka what I learned during this conference.)

Possible petascale machine	
1. Number of cores per nodes	10 – 100 cores
2. Performance per nodes	100 – 1,000 GFLOPs/sec
3. Number of nodes	1000-10,000 nodes
4. Latency inter-nodes	1 µsec
5. Bandwidth inter-nodes	10 Gb/sec
6. Memory per nodes	10 GB

- **Part I: First rule in linear algebra: Have an efficient DGEMM**
  - **Motivation in**
    2. performance per node 5. bandwidth inter-nodes 6. memory per nodes

First rule in linear algebra: Have an efficient DGEMM

- All the dense linear algebra operations rely on an efficient DGEMM (matrix Matrix Multiply)
- This is by far the easiest operation O(n^3) in Dense Linear Algebra.
  - So if we can not implement DGEMM correctly (peak performance), we will not be able to do much for the others operations.

Model.

Blocking communication	Nonblocking communication.
$perf = \dfrac{2n^3}{\dfrac{2n^3\gamma}{p} + 2\log(\sqrt{p})\dfrac{n^2}{\sqrt{p}}\beta}$	$perf = \dfrac{2n^3}{\max(\dfrac{2n^3\gamma}{p}, 2\log(\sqrt{p})\dfrac{n^2}{\sqrt{p}}\beta)}$

- γ is the time for one operation,
- β is the time to send one entry.
- various algorithms/models depending in the Bdcast algorithm used (Pipeline=SUMMA, tree=PUMMA, etc.).

Time for computation = # of ops * (time for one op) / (# of proc)

Model.

# of operations

Blocking communication	Nonblocking communication.
$perf = \dfrac{2n^3}{\dfrac{2n^3\gamma}{p} + 2\log(\sqrt{p})\dfrac{n^2}{\sqrt{p}}\beta}$	$perf = \dfrac{2n^3}{\max(\dfrac{2n^3\gamma}{p}, 2\log(\sqrt{p})\dfrac{n^2}{\sqrt{p}}\beta)}$

Time for communication = 2 * Bdcast on sqrt(p) processors of [n^2 / sqrt(p)] numbers

Time with blocking = time comp + time comm
Time with nonblocking = max(time comp , time comm)

- γ is the time for one operation,
- β is the time to send one entry.
- various algorithms/models depending in the Bdcast algorithm used (Pipeline=SUMMA, tree=PUMMA, etc.).

Model.

Blocking communication	Nonblocking communication.
$perf = \dfrac{2n^3}{\dfrac{2n^3\gamma}{p} + 2\log(\sqrt{p})\dfrac{n^2}{\sqrt{p}}\beta}$	$perf = \dfrac{2n^3}{\max(\dfrac{2n^3\gamma}{p}, 2\log(\sqrt{p})\dfrac{n^2}{\sqrt{p}}\beta)}$

- n^3 operations for n^2 communication.
- The model works fine, next slides present result from Alfredo Buttari, Jakub Kurzak, and Jack Dongarra from Utennessee.

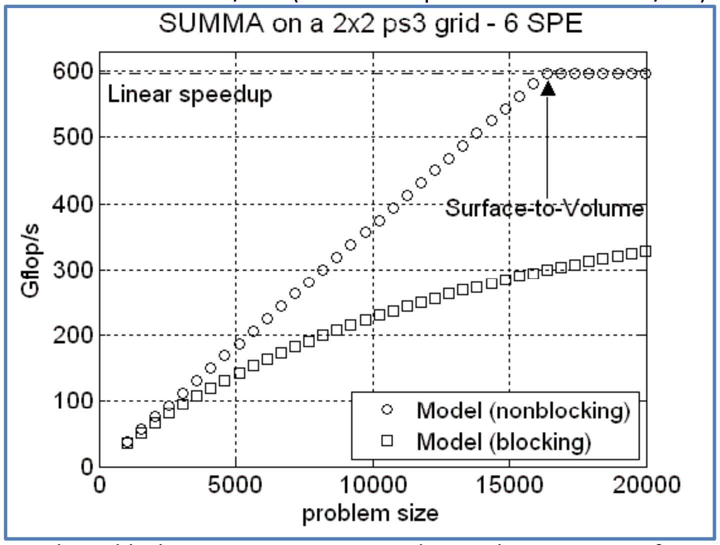
Alfredo Buttari, Jakub Kurzak, and Jack Dongarra. **Limitations of the playstation 3 for high performance cluster computing**. Technical Report UT-CS-07-597, Innovative Computing Laboratory, University of Tennessee Knoxville, April 2007.

Performance model.

Take for instance, the TN cluster of four PS3:

Bandwidth = 600 Mb/sec (with GigaBit Ethernet)

Flop rate = 149.85 GFLOPs/sec (theoretical peak is 153.6 GFLOPs/sec)



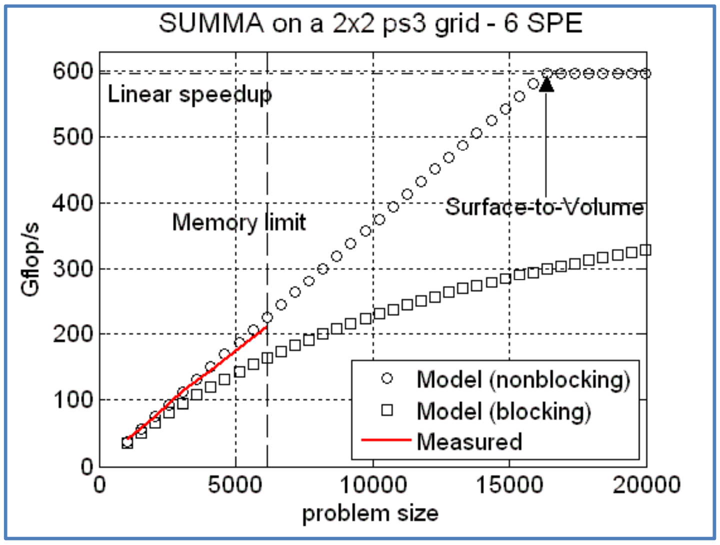
To get 90% of efficiency with nonblocking operations, one need to work on matrices of size 14,848.
To get 90% of efficiency with blocking operations, one need to work on matrices of size 146,949.

Really worse using nonblocking communication.

Alfredo Buttari, Jakub Kurzak, and Jack Dongarra. **Limitations of the playstation 3 for high performance cluster computing**. Technical Report UT-CS-07-597, Innovative Computing Laboratory, University of Tennessee Knoxville, April 2007.

Oups…

Memory for PS3 is 256MB.



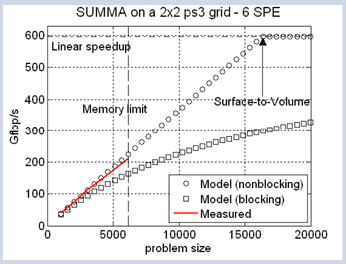
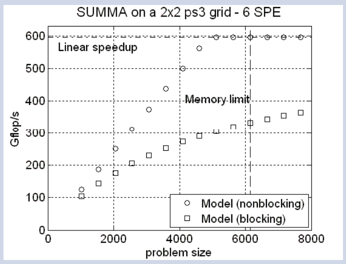
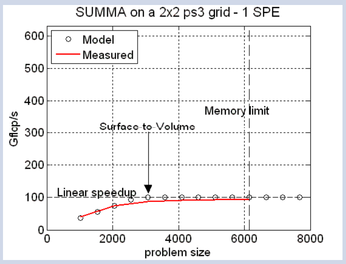
There is no way to hide the n^2 term (communication) by the n^3 term (computation). n can not get big enough. Actucally, it is the computation that are hidden by the communication. Dense Linear Algebra is stucked. There is nothing to be done.

Three ways to complain

- The network is too slow (complain to GigE)
- There is not enough memory on the nodes (complain to Sony)
- The nodes are too fast (complain to IBM)

Three Solutions

Instead of **600 Mb/sec – 258 MB – 6 SPEs** :

Increase the memory of the nodes	Increase the bandwidth of the network	Decrease the computational power of the nodes
		
600 Mb/sec – **3.3 GB** – 6 SPEs	**1.39 Gb/sec** – 258 MB – 6 SPEs	600 Mb/sec – 258 MB – **2 SPEs**

Lessons from the story.

- This story tells us that we have to be careful when choosing the triplet:
  2. performance per node
  5. bandwidth inter-nodes
  6. memory per nodes

- Come back to our petascale machine.

To perform at 99.9% of the peak on a cluster with 1 TFLOPs/sec per node (nonblock)			
Perf of a node	**Bw internodes**	**mem / node**	**(n , time)**
1 TFLOP/sec	10 Gb/sec	1GB	(n=13,000, t=5ms)
1 TFLOP/sec	5 Gb/sec	4GB	(n=26,000, t=20ms)
1 TFLOP/sec	2.5 Gb/sec	16GB	(n=52,000, t=80ms)

- The first line was in our spec of the petascale machine. This should more or less work then. So we assume MM is OK, and we can move forward to more challenging algorithms. (latency bounded for example, high granularity, …)

- **Cf. H.T Kung**. Memory Requirement for Balanced Computer Architectures. 13th International Symposium on Computer Architectures. 1986. pp. 49—54.

What will a petascale machine looks like.

Possible petascale machine	
1. Number of cores per nodes	10 – 100 cores
2. Performance per nodes	100 – 1,000 GFLOPs/sec
3. Number of nodes	1000-10,000 nodes
4. Latency inter-nodes	1 μsec
5. Bandwidth inter-nodes	10 Gb/sec
6. Memory per nodes	10 GB

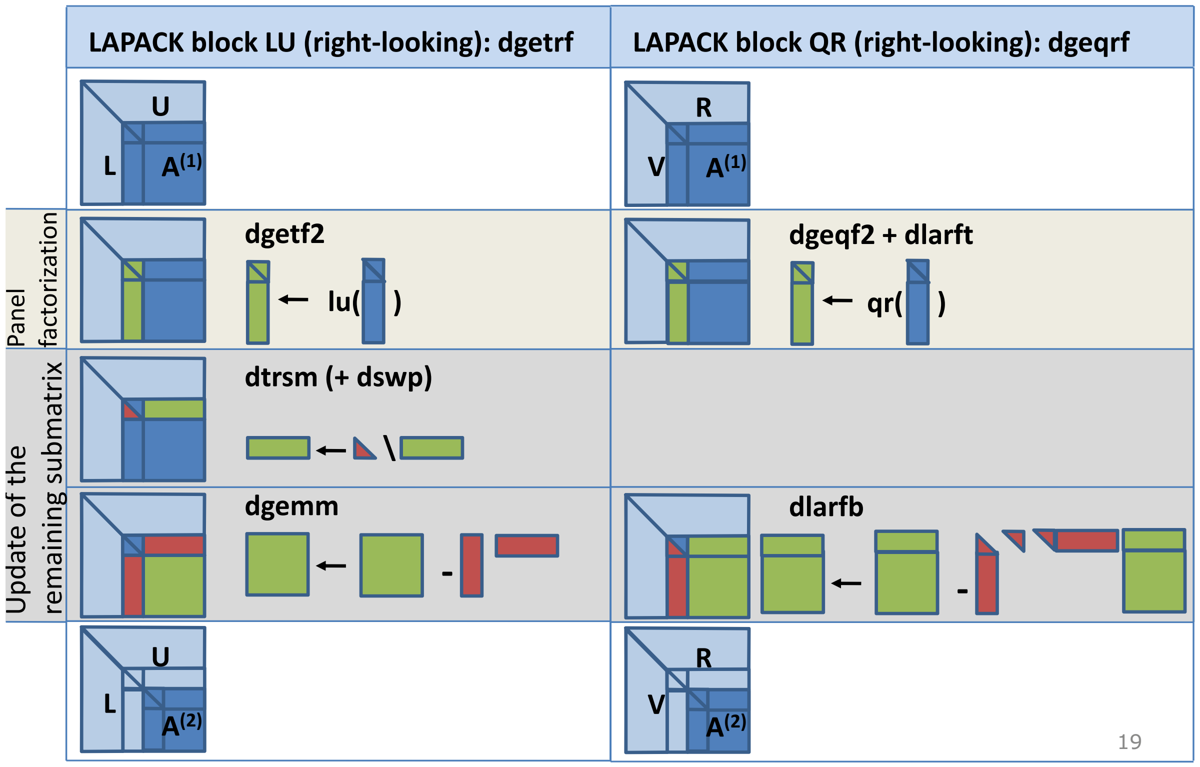
- **Part I: First rule in linear algebra: Have an efficient DGEMM**
  - **Motivation in**
    - **2. performance per node 5. bandwidth inter-nodes 6. memory per nodes**
- Part II: Algorithm for multicore / latency avoiding algorithm for LU, QR …
  - Motivation in:
    - 1. Number of cores per node 2. performance per node 4. Latency inter-nodes
- Part III: Algorithm for fault tolerance
  - Motivation in:
    - 1. Number of cores per node 3. number of nodes

What will a petascale machine looks like.

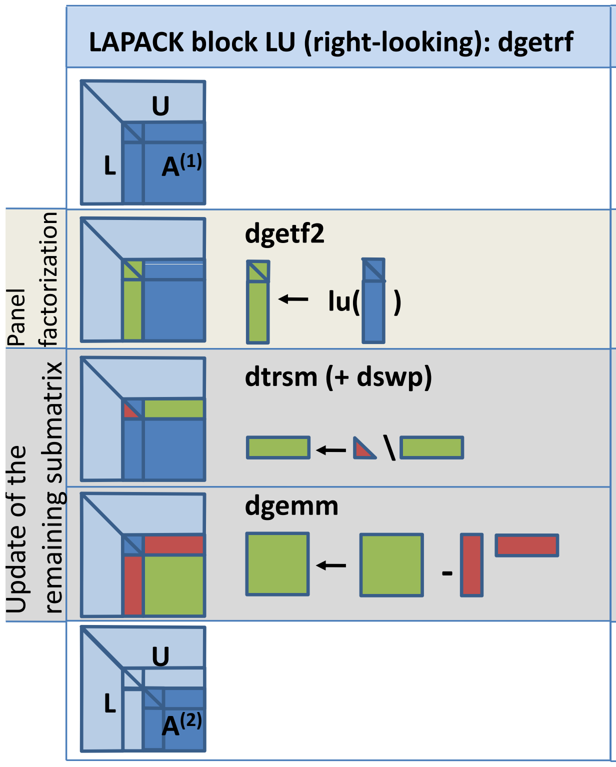
Possible petascale machine	
1. Number of cores per nodes	10 – 100 cores
2. Performance per nodes	100 – 1,000 GFLOPs/sec
3. Number of nodes	1000-10,000 nodes
4. Latency inter-nodes	1 μsec
5. Bandwidth inter-nodes	10 Gb/sec
6. Memory per nodes	10 GB

- Part I: First rule in linear algebra: Have an efficient DGEMM
  - Motivation in
    - 2. performance per node 5. bandwidth inter-nodes 6. memory per nodes
- **Part II: Algorithm for multicore / latency avoiding algorithm for LU, QR …**
  - **Motivation in:**
    - **1. Number of cores per node 2. performance per node 4. Latency inter-nodes**
- Part III: Algorithm for fault tolerance
  - Motivation in:
    - 1. Number of cores per node 3. number of nodes

Blocked LU and QR algorithms (LAPACK)


LAPACK block LU (right-looking): dgetrf
LAPACK block QR (right-looking): dgeqrf
U
L
A(1)
R
V
A(1)
Panel factorization
dgetf2
lu( )
dgeqf2 + dlarft
qr( )
Update of the remaining submatrix
dtrsm (+ dswp)
\
dgemm
-
dlarfb
-
U
L
A(2)
R
V
A(2)

Blocked LU and QR algorithms (LAPACK)


LAPACK block LU (right-looking): dgetrf
U
L
A(1)
Panel factorization
dgetf2
lu( )
Update of the remaining submatrix
dtrsm (+ dswp)
\
dgemm
-
U
L
A(2)

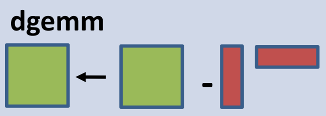
Latency bounded:
more than nb AllReduce for $n*nb^2$ ops

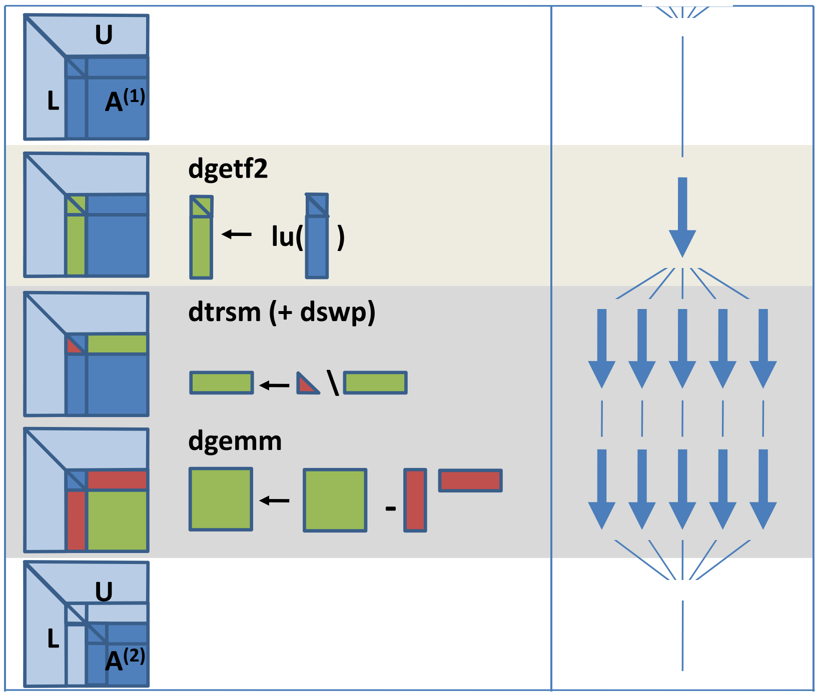
CPU - bandwidth bounded:
the bulk of the computation: n*n*nb ops
highly paralleliable, efficient and saclable.

Parallelization of LU and QR.

Parallelize the update:

- Easy and done in any reasonable software.
- This is the $2/3n^3$ term in the FLOPs count.
- Can be done efficiently with LAPACK+multithreaded BLAS

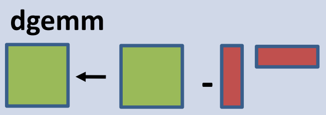




Parallelization of LU and QR.

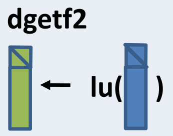
Parallelize the update:

- Easy and done in any reasonable software.
- This is the $2/3n^3$ term in the FLOPs count.
- Can be done efficiently with LAPACK+multithreaded BLAS



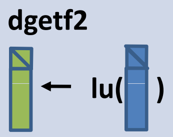
Parallelize the panel factorization:

- Not an option in multicore context ($p < 16$)
- See e.g. ScaLAPACK or HPL but still by far the slowest and the bottleneck of the computation.

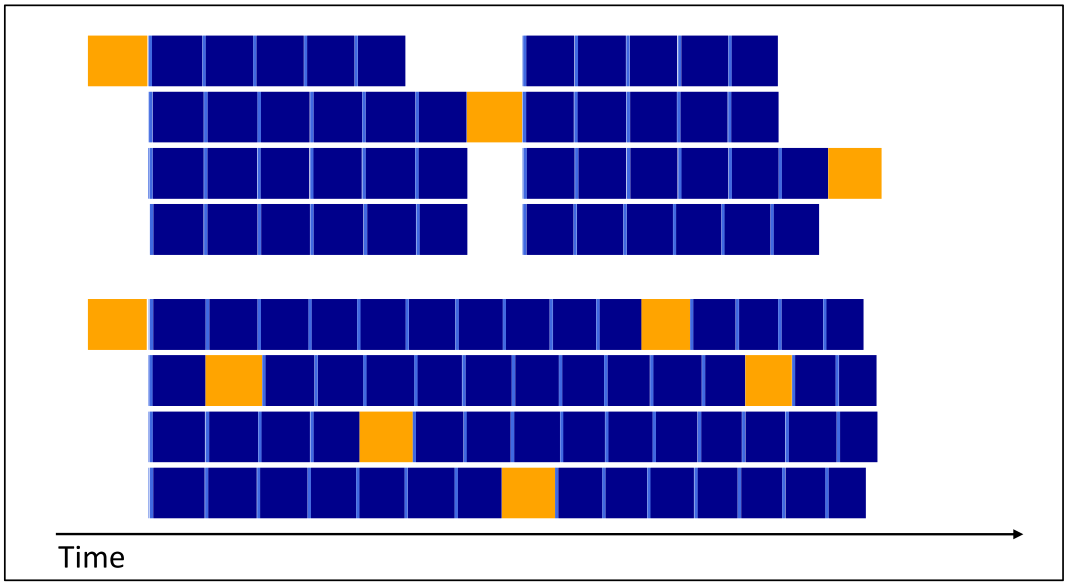


Hide the panel factorization:

- Lookahead (see e.g. High Performance LINPACK)
- Dynamic Scheduling



Hiding the panel factorization with dynamic scheduling.



Courtesy from Alfredo Buttari, UTennessee

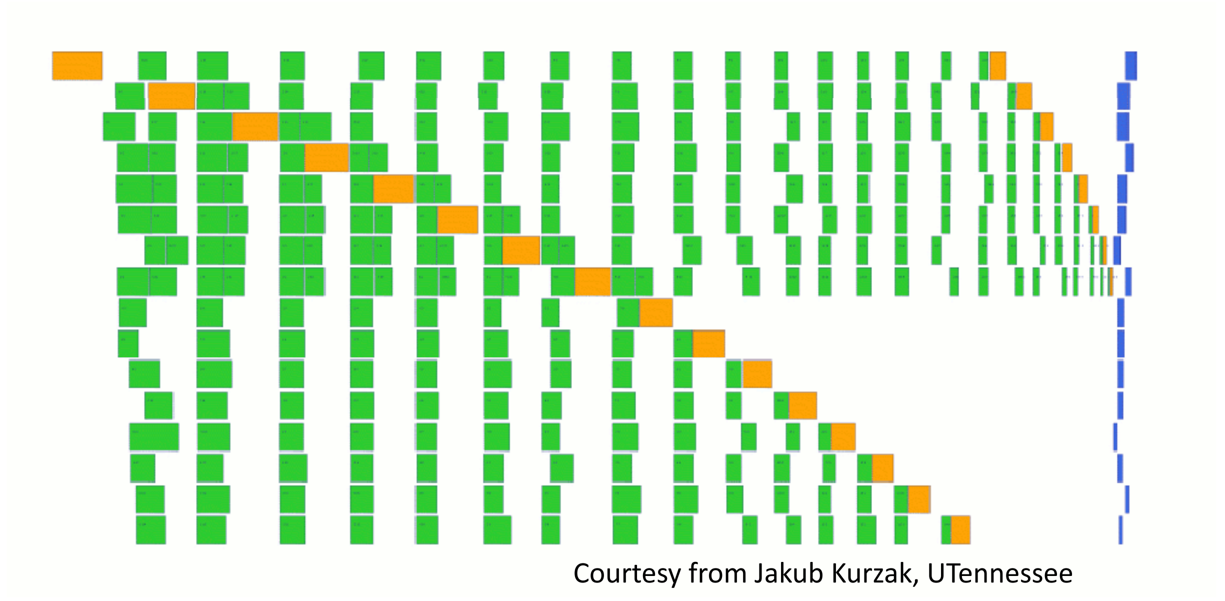
What about strong scalability?

What about strong scalability?

N = 1536

NB = 64

procs = 16


Courtesy from Jakub Kurzak, UTennessee

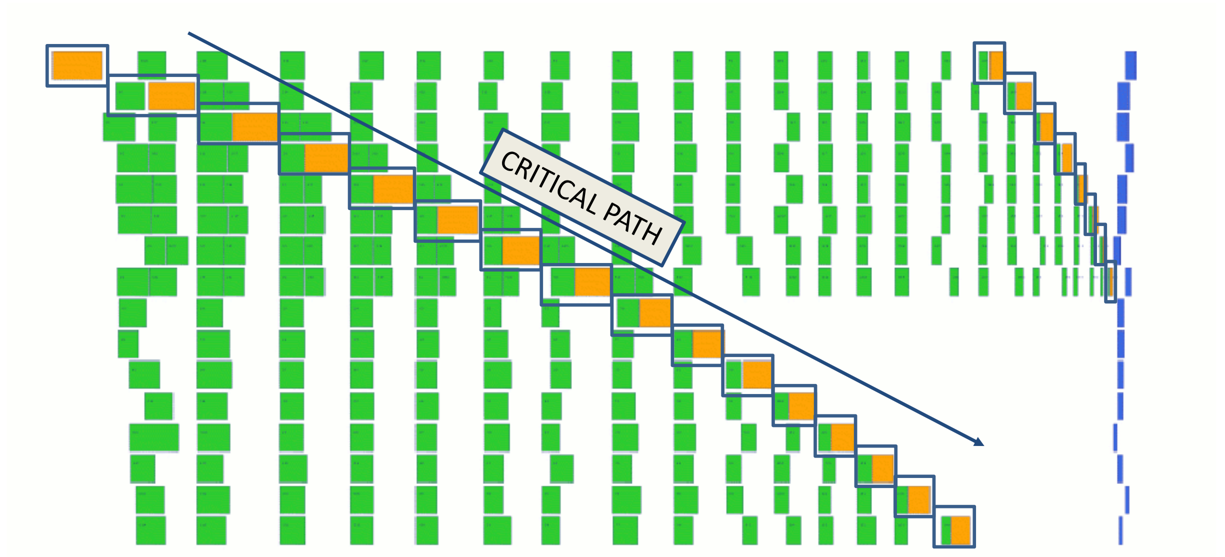
What about strong scalability?

N = 1536

NB = 64

procs = 16

We can not hide the panel factorization in the MM, actually it is the MMs that are hidden by the panel factorizations!



Courtesy from Jakub Kurzak, UTennessee

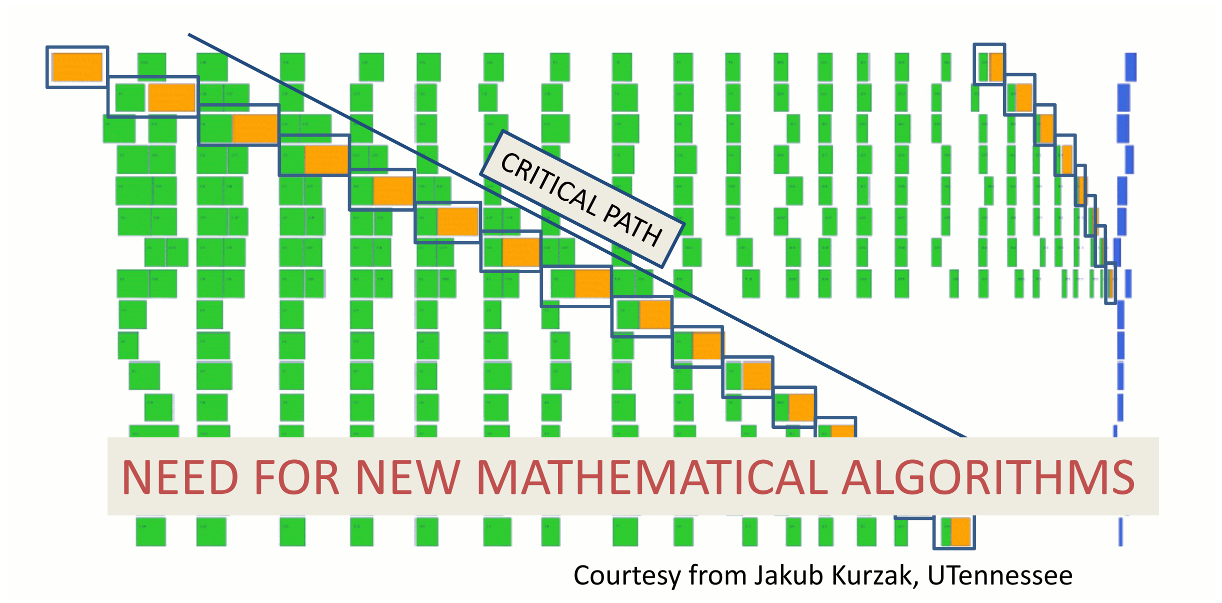
What about strong scalability?

N = 1536

NB = 64

procs = 16

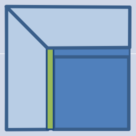
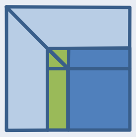
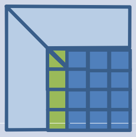
We can not hide the panel factorization (n^2) with the MM(n^3) , actually it is the MMs that are hidden by the panel factorizations!



NEED FOR NEW MATHEMATICAL ALGORITHMS

Courtesy from Jakub Kurzak, UTennessee

A new generation of algorithms?

Algorithms follow hardware evolution along time.		
LINPACK (80's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (90's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
New Algorithms (00's) (multicore friendly)		Rely on - a DAG/scheduler - block data layout - some extra kernels

Those new algorithms

- have a very **low granularity**, they scale very well (multicore, petascale computing, …)
- **removes a lots of dependencies** among the tasks, (multicore, distributed computing)
- **avoid latency** (distributed computing, out-of-core)
- **rely on fast kernels**

Those new algorithms need new kernels and rely on efficient scheduling algorithms.

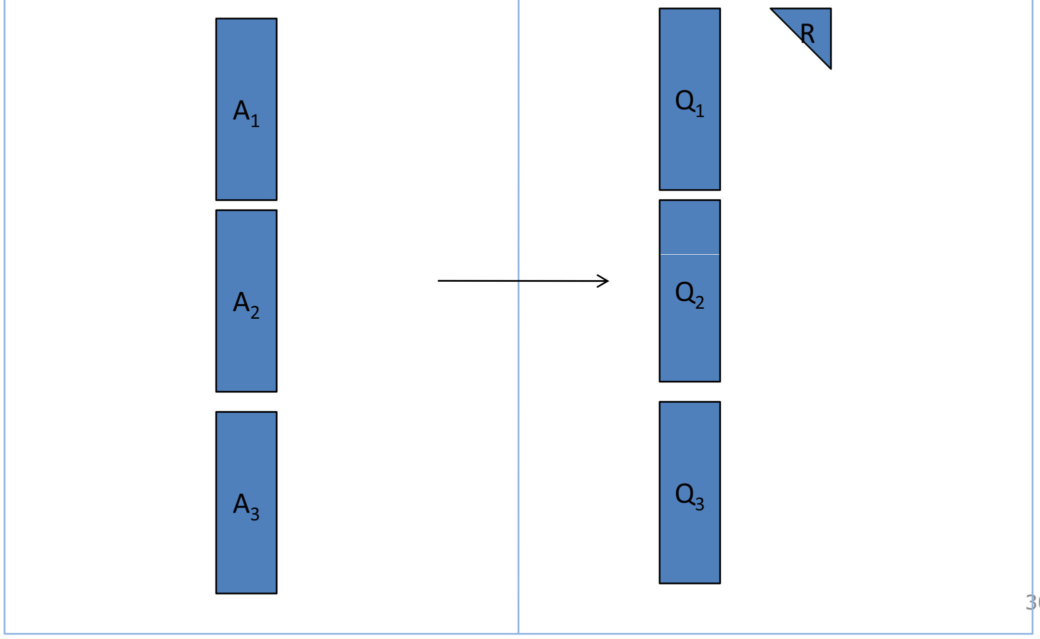
On multicore machines.

- See Alfredo's previous talk.

Reduce Algorithms: Introduction

The QR factorization of a long and skinny matrix with its data partitioned vertically across several processors arises in a wide range of applications.

Input: A is block distributed by rows	**Output:** Q is block distributed by rows R is global



Example of applications: in block iterative methods.

a) in iterative methods with multiple right-hand sides (block iterative methods:)

1) Trilinos (Sandia National Lab.) through Belos (R. Lehoucq, H. Thornquist, U. Hetmaniuk).
2) BlockGMRES, BlockGCR, BlockCG, BlockQMR, …

b) in iterative methods with a single right-hand side

1) s-step methods for linear systems of equations (e.g. A. Chronopoulos),
2) LGMRES (Jessup, Baker, Dennis, U. Colorado at Boulder) implemented in PETSc,
3) Recent work from M. Hoemmen and J. Demmel (U. California at Berkeley).

c) in iterative eigenvalue solvers,

1) PETSc (Argonne National Lab.) through BLOPEX (A. Knyazev, UCDHSC),
2) HYPRE (Lawrence Livermore National Lab.) through BLOPEX,
3) Trilinos (Sandia National Lab.) through Anasazi (R. Lehoucq, H. Thornquist, U. Hetmaniuk),
4) PRIMME (A. Stathopoulos, Coll. William & Mary),
5) And also TRLAN, BLZPACK, IRBLEIGS.

Reduce Algorithms: Introduction

Example of applications:

a) in block iterative methods (iterative methods with multiple right-hand sides or iterative eigenvalue solvers),

b) in dense large and more square QR factorization where they are used as the panel factorization step, or more simply

c) in linear least squares problems which the number of equations is extremely larger than the number of unknowns.

Reduce Algorithms: Introduction

Example of applications:

a) in block iterative methods (iterative methods with multiple right-hand sides or iterative eigenvalue solvers),

b) in dense large and more square QR factorization where they are used as the panel factorization step, or more simply

c) in linear least squares problems which the number of equations is extremely larger than the number of unknowns.

The main characteristics of those three examples are that

a) there is **only one column of processors involved** but several processor rows,

b) **all the data is known from the beginning**,

c) and **the matrix is dense**.

Reduce Algorithms: Introduction

Example of applications:

a) in block iterative methods (iterative methods with multiple right-hand sides or iterative eigenvalue solvers),

b) in dense large and more square QR factorization where they are used as the panel factorization step, or more simply

c) in linear least squares problems which the number of equations is extremely larger than the number of unknowns.

The main characteristics of those three examples are that

a) there is **only one column of processors involved** but several processor rows,

b) **all the data is known from the beginning**,

c) and **the matrix is dense**.

Various methods already exist to perform the QR factorization of such matrices:

a) Gram-Schmidt (**mgs(row),cgs**),

b) Householder (**qr2**, **qrf**),

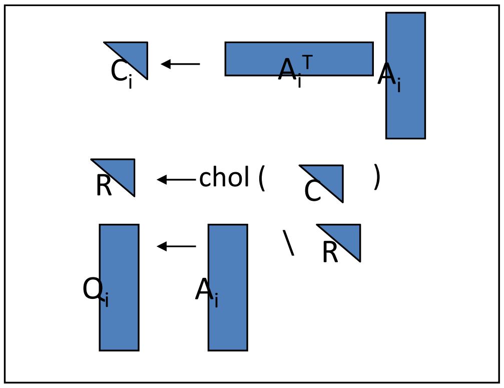
c) or **CholeskyQR**.

We present a new method:

Allreduce Householder (**rhh_qr3**, **rhh_qrf**).

The CholeskyQR Algorithm

SYRK:	C:= A^TA	(mn^2)
CHOL:	R := chol(C)	($n^3/3$)
TRSM:	Q := A\R	(mn^2)



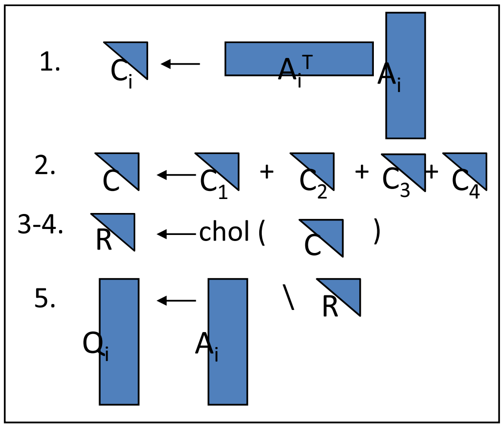
Bibligraphy

- A. Stathopoulos and K. Wu, A block orthogonalization procedure with constant synchronization requirements, *SIAM Journal on Scientific Computing*, 23(6):2165-2182, 2002.
- Popularized by iterative eigensolver libraries:
  1) PETSc (Argonne National Lab.) through BLOPEX (A. Knyazev, UCDHSC),
  2) HYPRE (Lawrence Livermore National Lab.) through BLOPEX,
  3) Trilinos (Sandia National Lab.) through Anasazi (R. Lehoucq, H. Thornquist, U. Hetmaniuk),
  4) PRIMME (A. Stathopoulos, Coll. William & Mary).

Parallel distributed CholeskyQR

The CholeskyQR method in the parallel distributed context can be described as follows:

1: SYRK:	$C := A^TA$	(mn^2)
2: MPI_Reduce:	$C := \mathrm{sum}_{procs}\ C$	(on proc 0)
3: CHOL:	$R :=$ chol(C)	($n^3/3$)
4: MPI_Bdcast	Broadcast the R factor on proc 0 to all the other processors	
5: TRSM:	$Q := A\backslash R$	(mn^2)



This method is extremely fast. For two reasons:

1. first, there is **only one or two communications phase**,
2. second, **the local computations are performed with fast operations**.

Another advantage of this method is that the resulting code is exactly four lines,

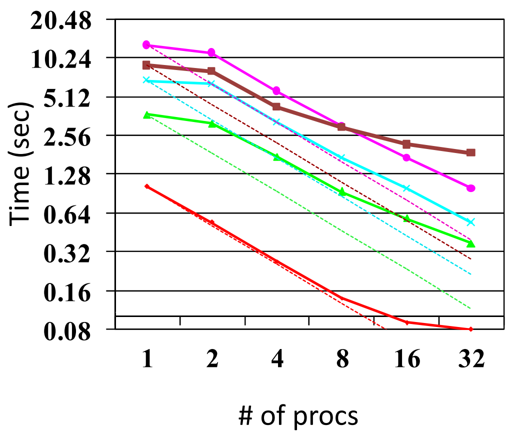
3. so the method is **simple** and relies heavily on other libraries.

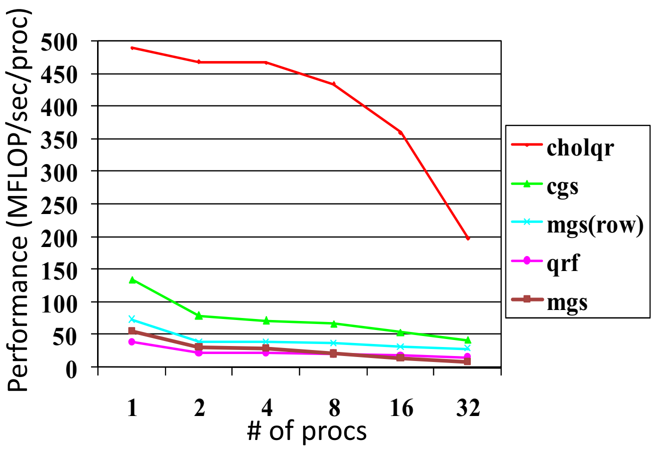
Despite all those advantages,

4. this method is **highly unstable**.

In this experiment, we fix the problem: **m=100,000** and **n=50**.

Efficient enough?



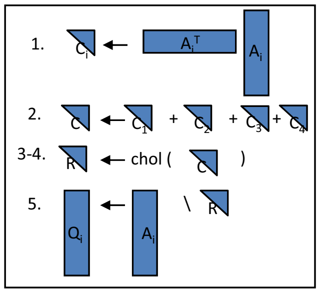


# of procs	cholqr		cgs		mgs(row)		qrf		mgs	
1	489.2	(1.02)	134.1	(3.73)	73.5	(6.81)	39.1	(12.78)	56.18	(8.90)
2	467.3	(0.54)	78.9	(3.17)	39.0	(6.41)	22.3	(11.21)	31.21	(8.01)
4	466.4	(0.27)	71.3	(1.75)	38.7	(3.23)	22.2	(5.63)	29.58	(4.23)
8	434.0	(0.14)	67.4	(0.93)	36.7	(1.70)	20.8	(3.01)	21.15	(2.96)
16	359.2	(0.09)	54.2	(0.58)	31.6	(0.99)	18.3	(1.71)	14.44	(2.16)
32	197.8	(0.08)	41.9	(0.37)	29.0	(0.54)	15.8	(0.99)	8.38	(1.87)

MFLOP/sec/proc

Time in sec

Simple enough?

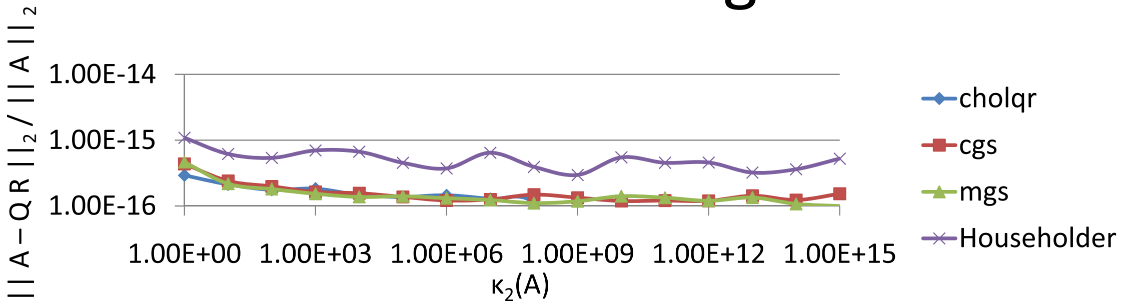


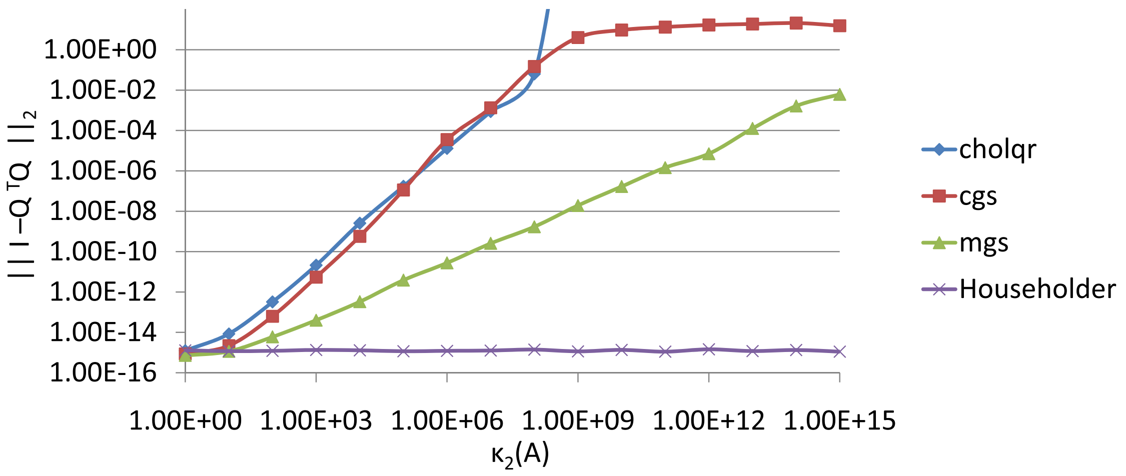
```
int choleskyqr_A_v0(int mloc, int n, double *A, int lda, double *R, int ldr,
            MPI_Comm mpi_comm){

            int info;
            cblas_dsyrk( CblasColMajor, CblasUpper, CblasTrans, n, mloc,
                        1.0e+00, A, lda, 0e+00, R, ldr );
            MPI_Allreduce( MPI_IN_PLACE, R, n*n, MPI_DOUBLE, MPI_SUM, mpi_comm );
            lapack_dpotrf( lapack_upper, n, R, ldr, &info );
            cblas_dtrsm( CblasColMajor, CblasRight, CblasUpper, CblasNoTrans, CblasNonUnit,
                        mloc, n, 1.0e+00, R, ldr, A, lda );
            return 0;

}
```

(And OK, you might want to add an MPI user defined datatype to send only the upper part of R)

m=100, n=50

Stable enough?

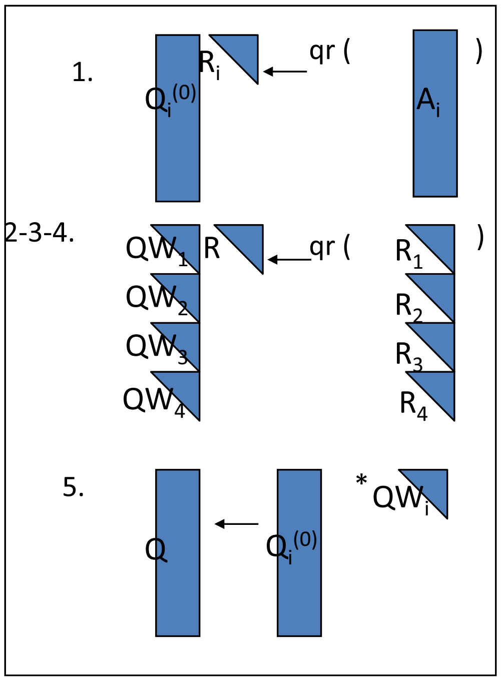




Reduce Algorithms

The gather-scatter variant of our algorithm can be summarized as follows:

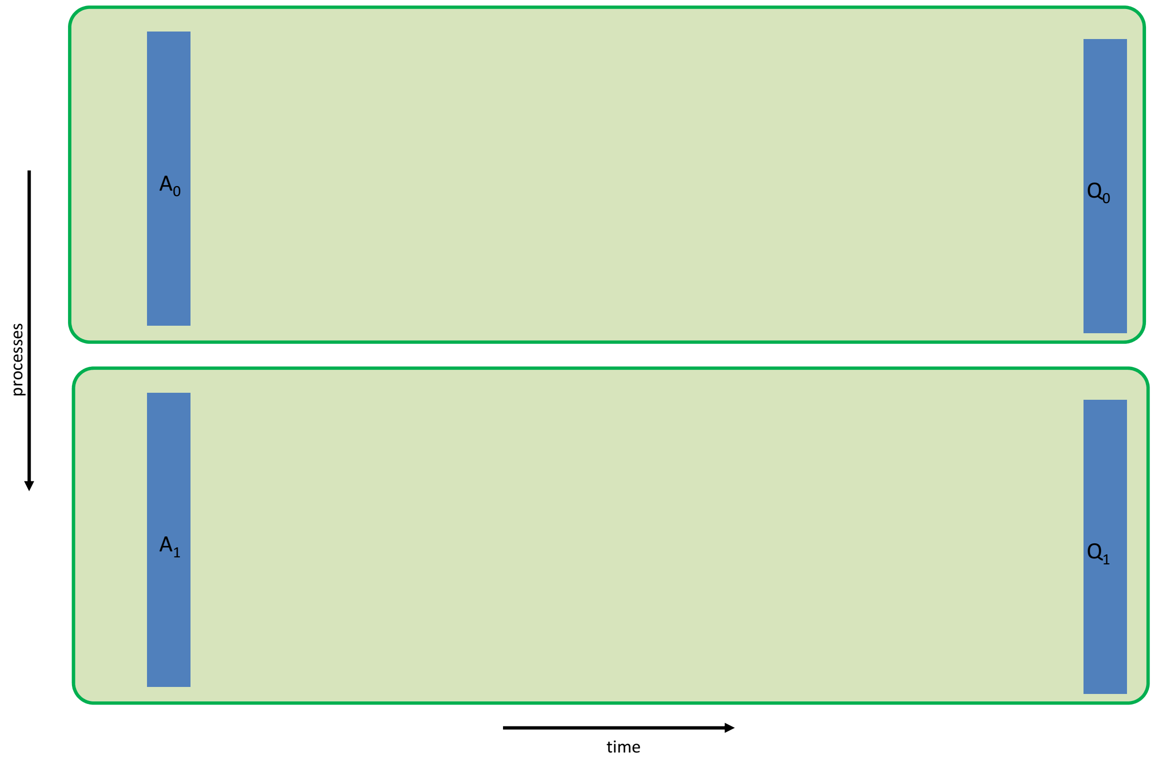
1. perform local QR factorization of the matrix A
2. gather the p R factors on processor 0
3. perform a QR factorization of all the R put the ones on top of the others, the R factor obtained is the R factor
4. scatter the the Q factors from processor 0 to all the processors
5. multiply locally the two Q factors together, done.



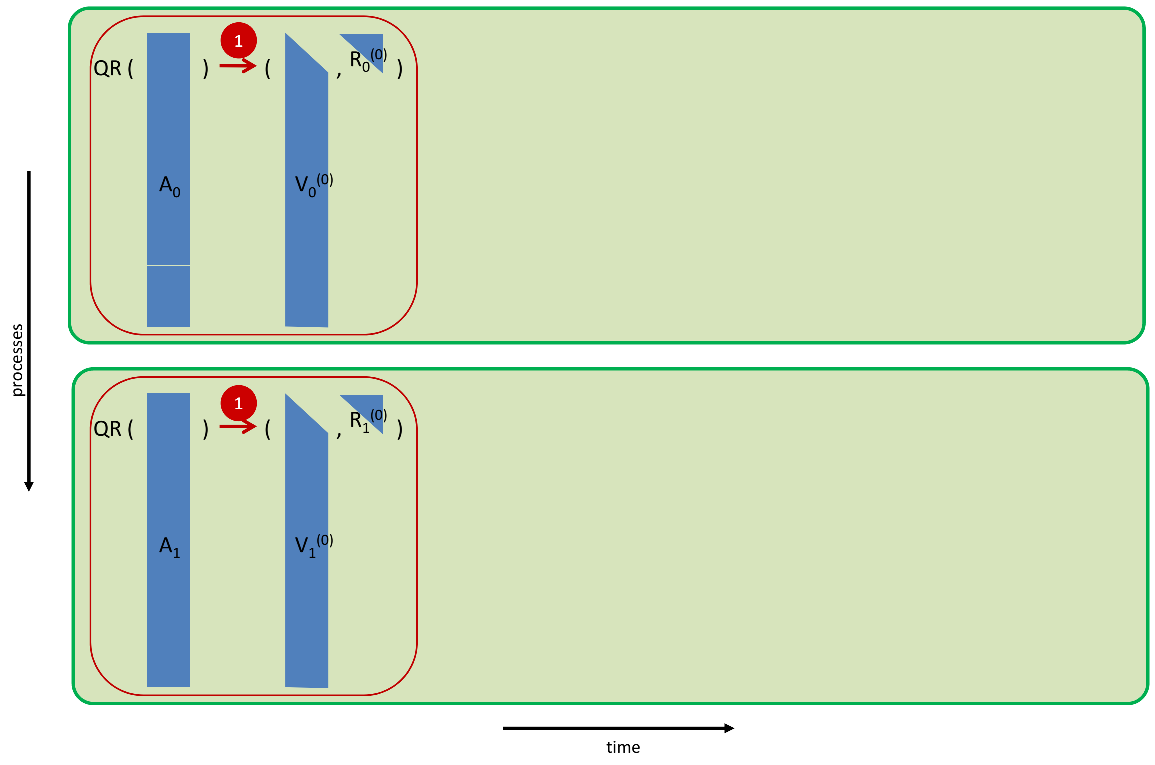
Reduce Algorithms

- This is the scatter-gather version of our algorithm.
- This variant is not very efficient for two reasons:
  - first the communication phases 2 and 4 are highly involving processor 0 ;
  - second the cost of step 3 is $p/3*n^3$, so can get prohibitive for large p.
- Note that the CholeskyQR algorithm can also be implemented in a **scatter-gather** way but **reduce-broadcast**. This leads naturally to the algorithm presented below where a reduce-broadcast version of the previous algorithm is described. This will be our final algorithm.

On two processes

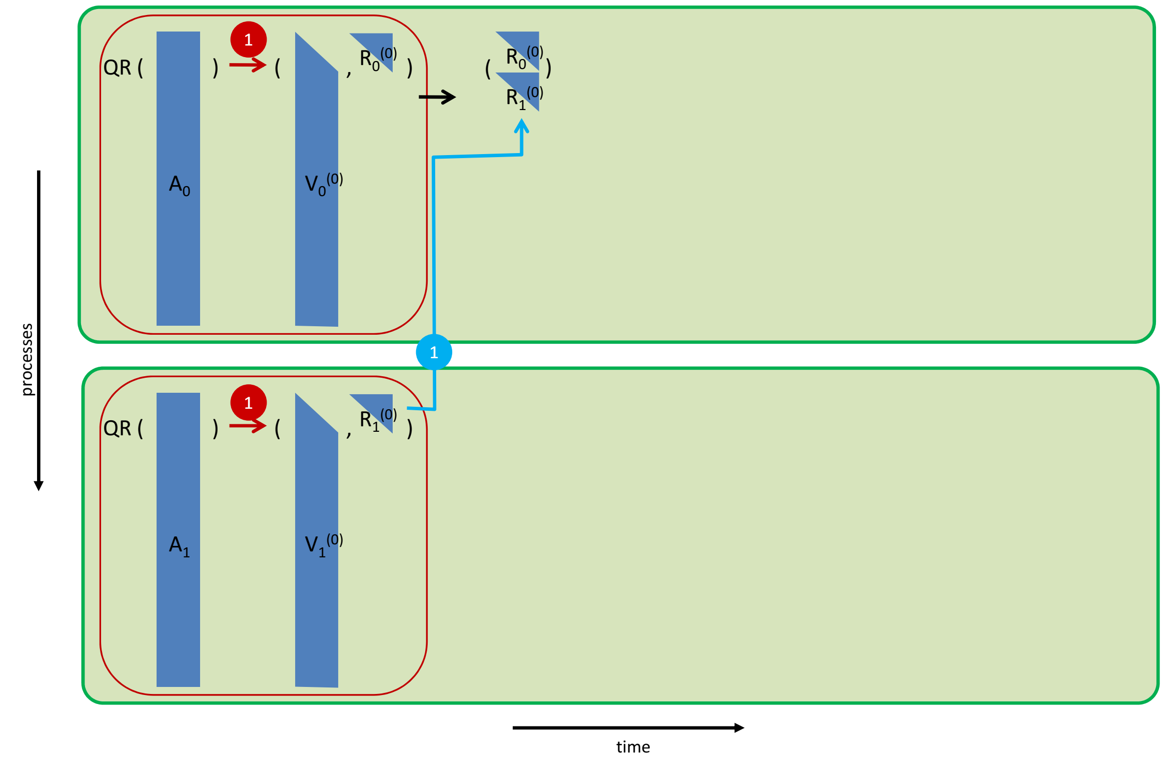


On two processes

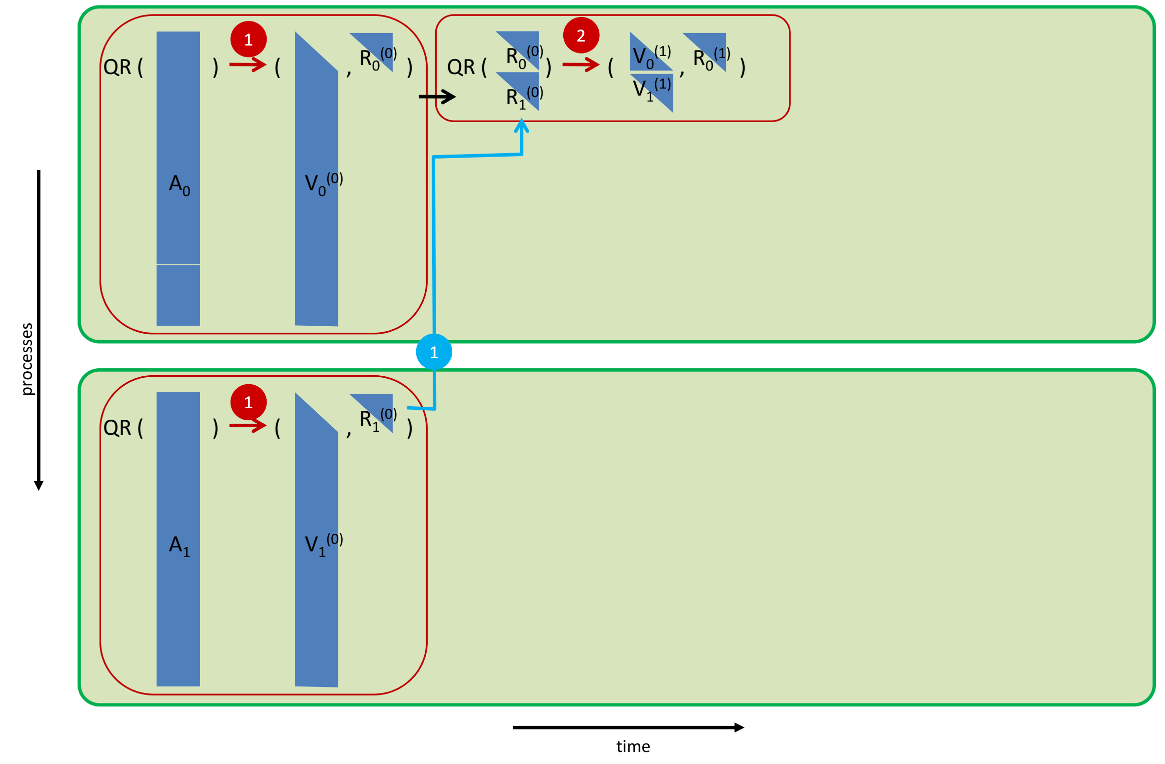


QR (A_0) $\xrightarrow{1}$ ($V_0^{(0)}$, $R_0^{(0)}$)
QR (A_1) $\xrightarrow{1}$ ($V_1^{(0)}$, $R_1^{(0)}$)
processes
time

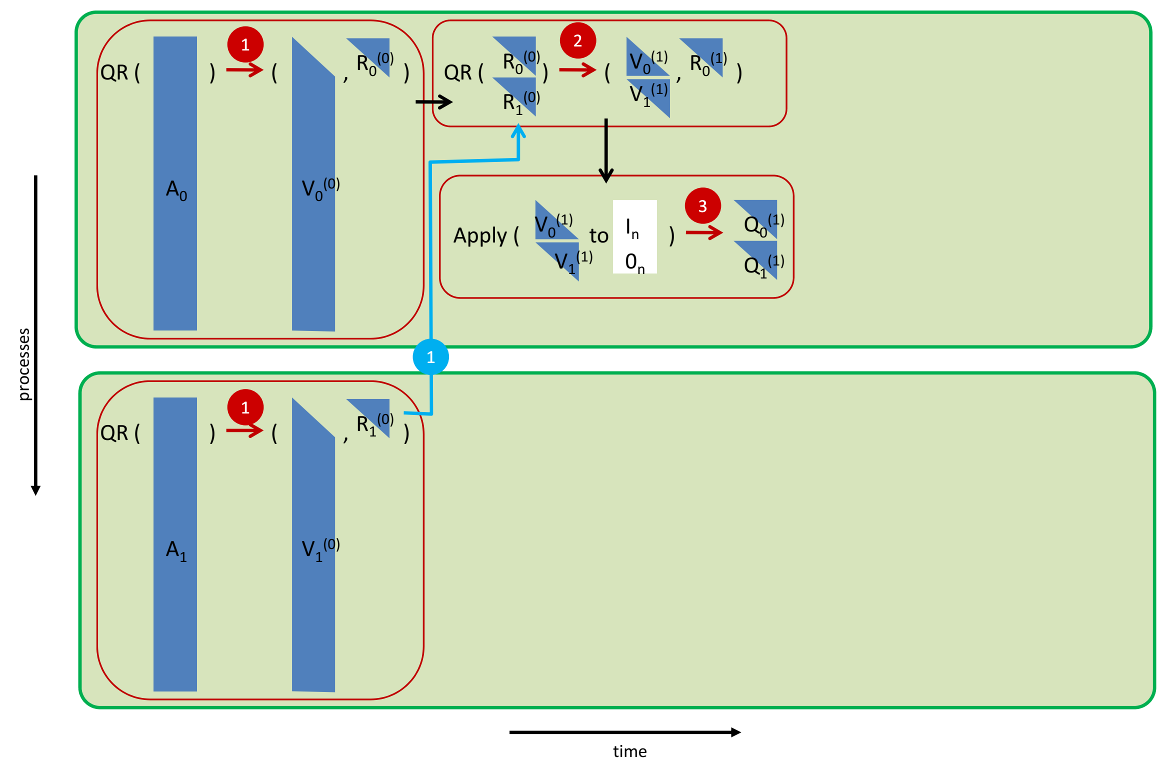
On two processes


QR (A_0) $\xrightarrow{1}$ ($V_0^{(0)}$, $R_0^{(0)}$)
($R_0^{(0)}$ / $R_1^{(0)}$)
QR (A_1) $\xrightarrow{1}$ ($V_1^{(0)}$, $R_1^{(0)}$)
1
processes
time

On two processes

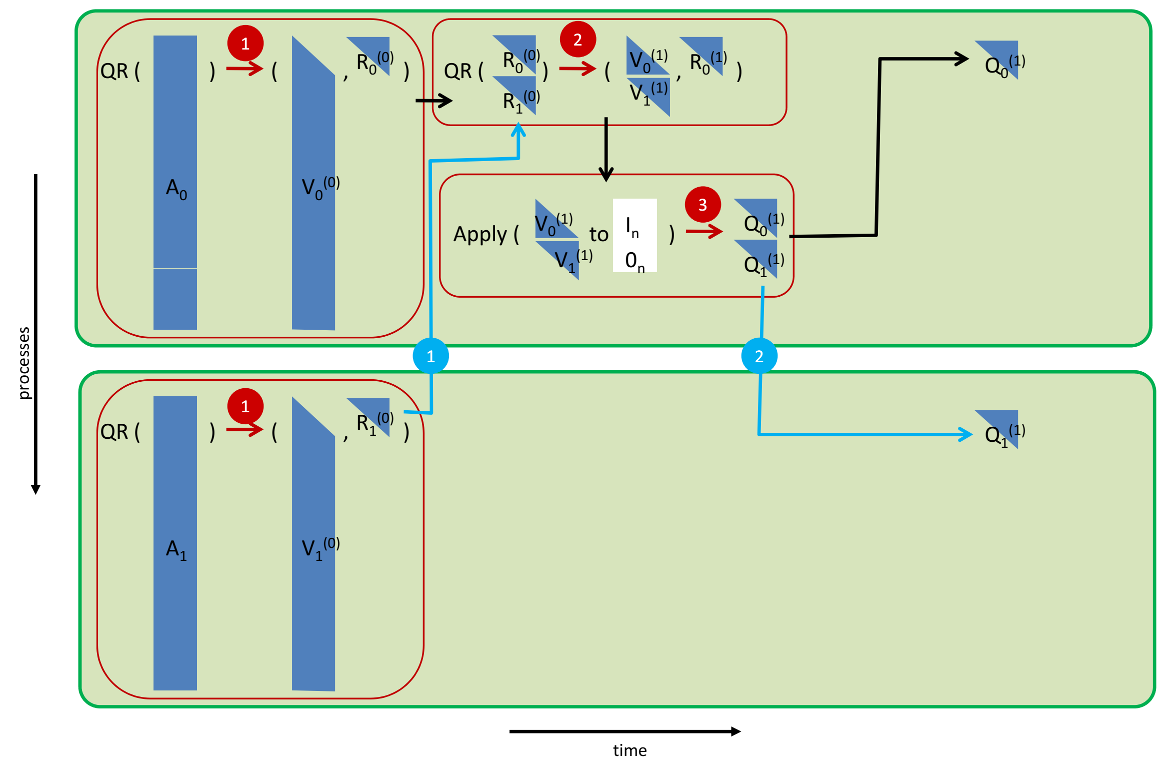

QR (A_0) $\xrightarrow{1}$ ($V_0^{(0)}$, $R_0^{(0)}$)
QR ($R_0^{(0)}$ / $R_1^{(0)}$) $\xrightarrow{2}$ ($V_0^{(1)}$ / $V_1^{(1)}$, $R_0^{(1)}$)
QR (A_1) $\xrightarrow{1}$ ($V_1^{(0)}$, $R_1^{(0)}$)
1
processes
time

On two processes



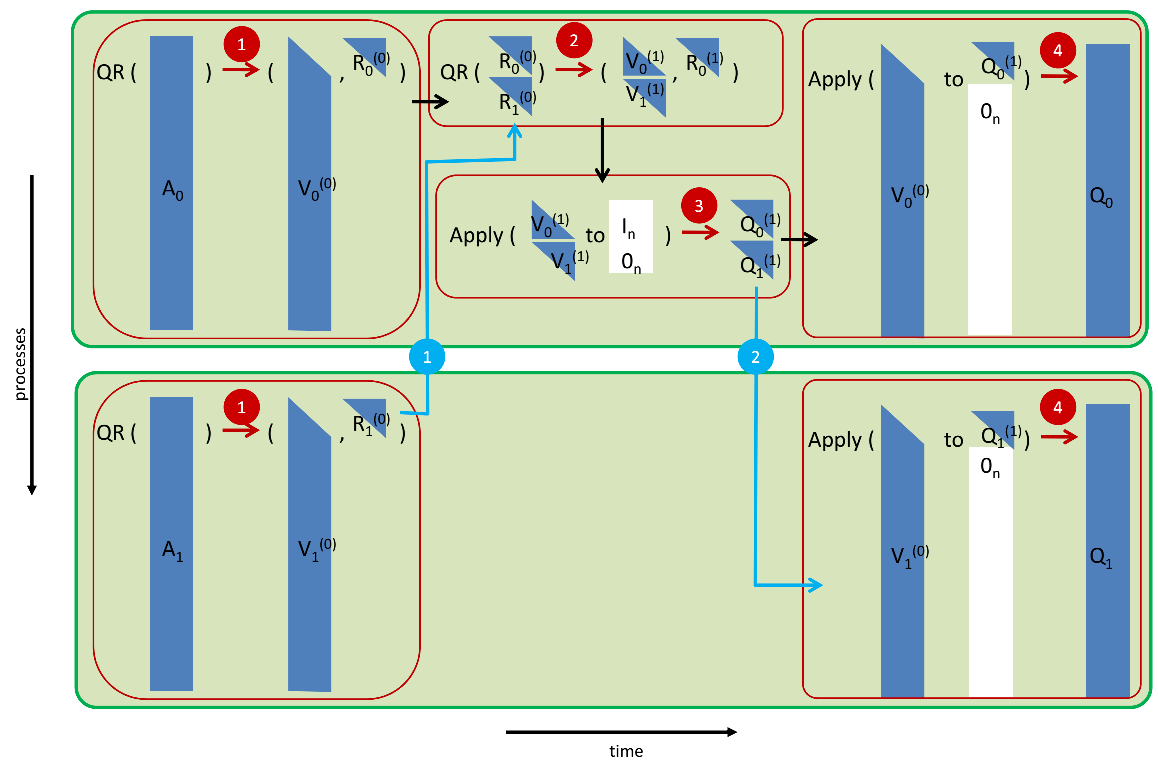
QR (A_0) → ($V_0^{(0)}$, $R_0^{(0)}$)
QR ($R_0^{(0)}$ $R_1^{(0)}$) → ($V_0^{(1)}$ $V_1^{(1)}$, $R_0^{(1)}$)
Apply ($V_0^{(1)}$ $V_1^{(1)}$ to I_n 0_n) → $Q_0^{(1)}$ $Q_1^{(1)}$
QR (A_1) → ($V_1^{(0)}$, $R_1^{(0)}$)
1
2
3
processes
time

On two processes



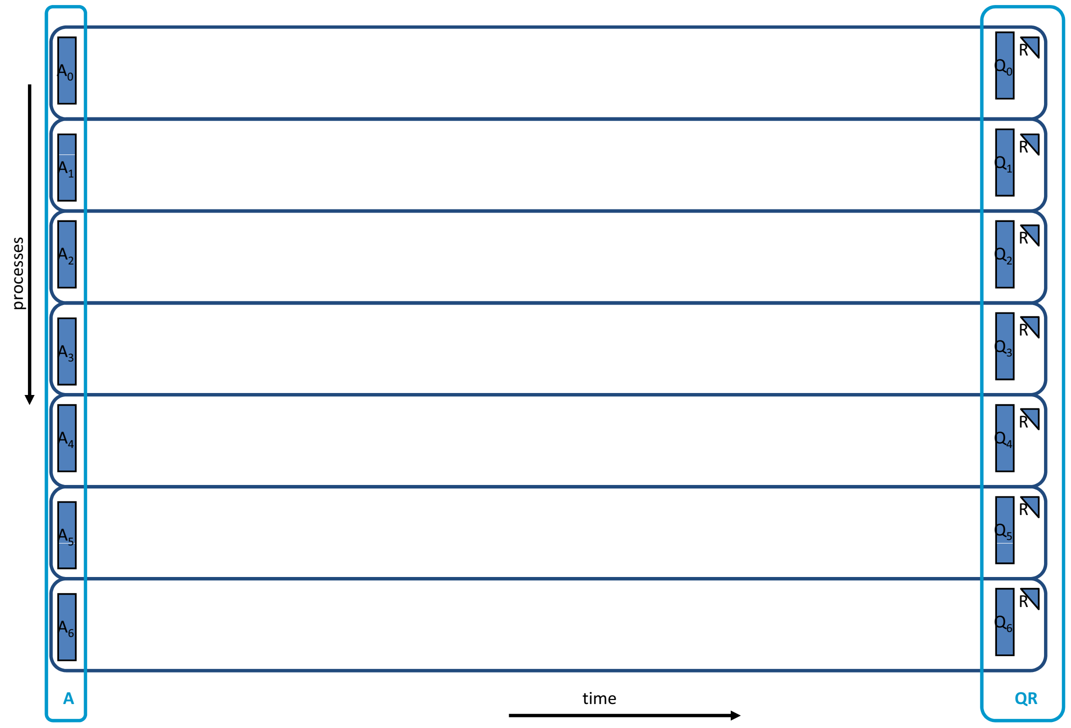
QR (A_0) → ($V_0^{(0)}$, $R_0^{(0)}$)
QR ($R_0^{(0)}$ $R_1^{(0)}$) → ($V_0^{(1)}$ $V_1^{(1)}$, $R_0^{(1)}$)
Apply ($V_0^{(1)}$ $V_1^{(1)}$ to I_n 0_n) → $Q_0^{(1)}$ $Q_1^{(1)}$
$Q_0^{(1)}$
QR (A_1) → ($V_1^{(0)}$, $R_1^{(0)}$)
$Q_1^{(1)}$
1
2
3
processes
time

On two processes

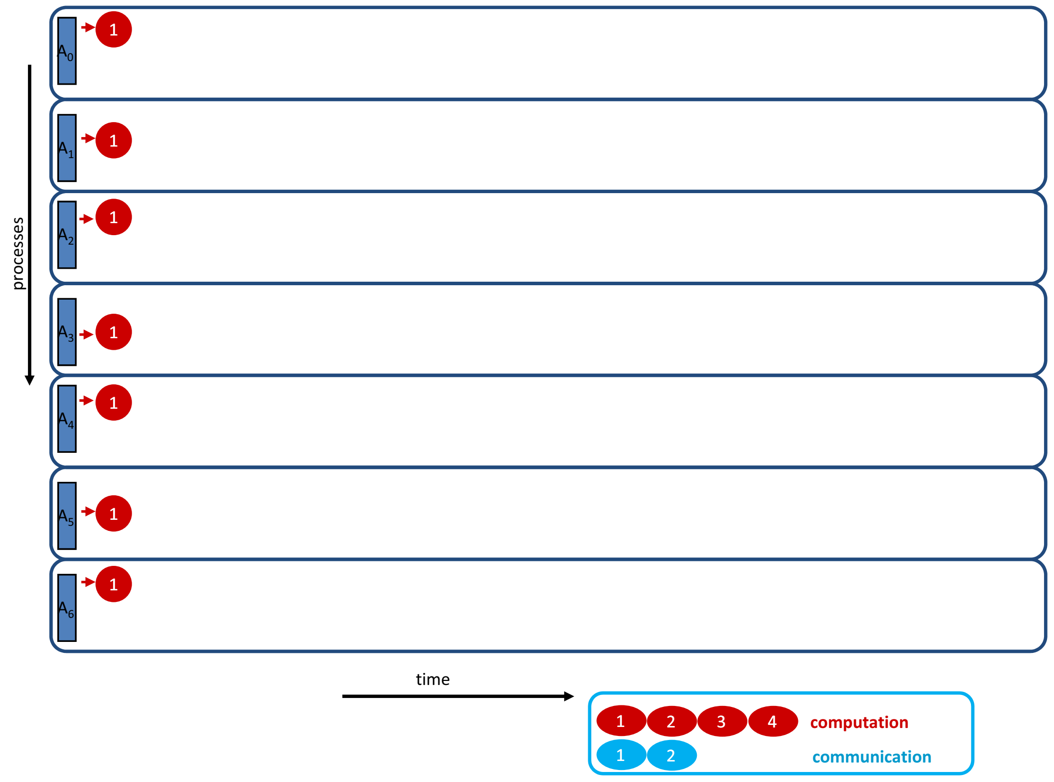


QR (A_0) → (V_0^(0) , R_0^(0))
QR (R_0^(0) R_1^(0)) → (V_0^(1) V_1^(1) , R_0^(1))
Apply (V_0^(1) V_1^(1) to I_n 0_n) → Q_0^(1) Q_1^(1)
Apply (V_0^(0) to Q_0^(1) 0_n) → Q_0
QR (A_1) → (V_1^(0) , R_1^(0))
Apply (V_1^(0) to Q_1^(1) 0_n) → Q_1
1
2
3
4
processes
time

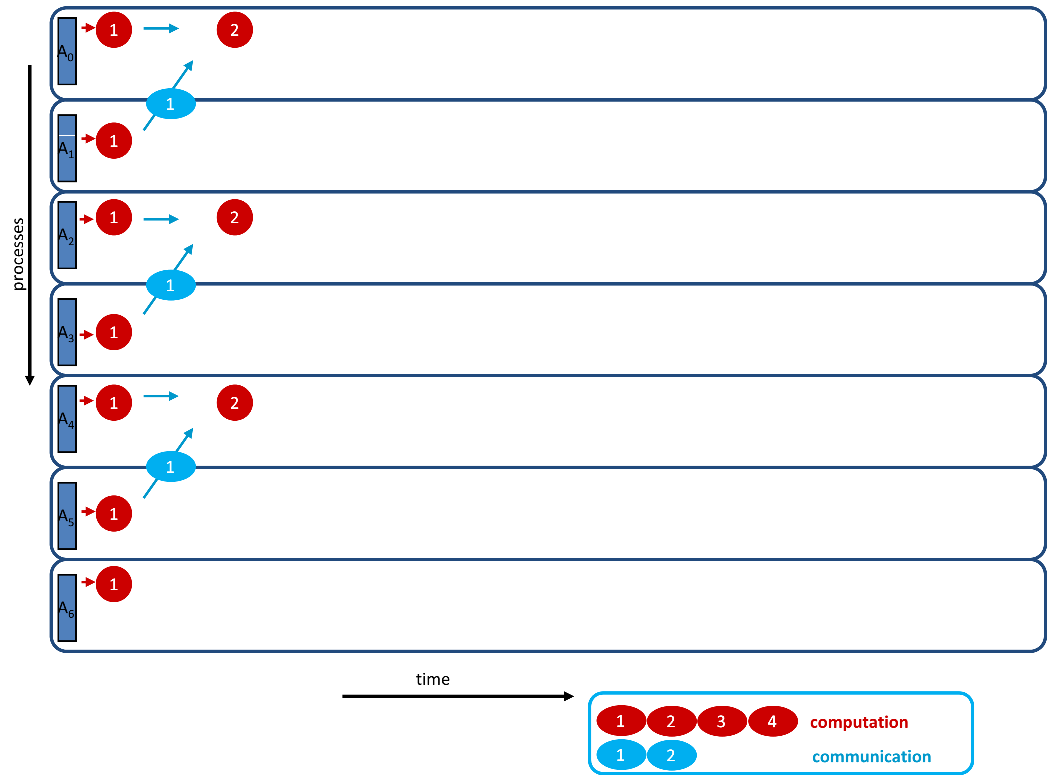
The big picture ….



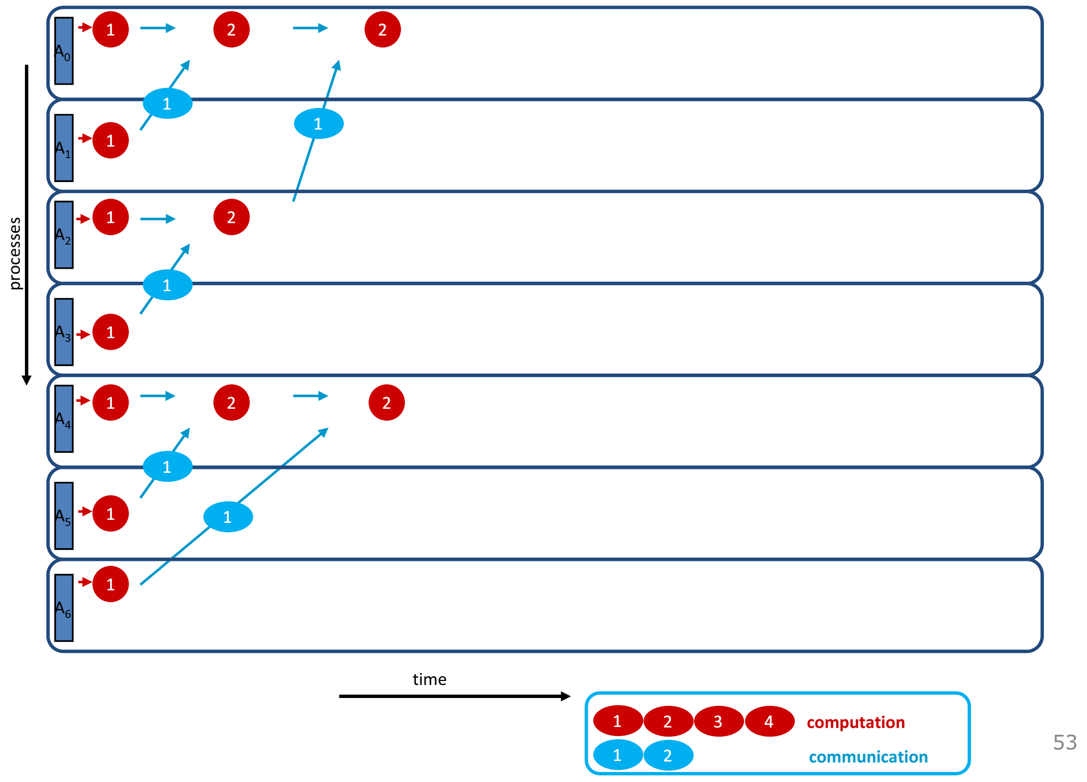
The big picture ….



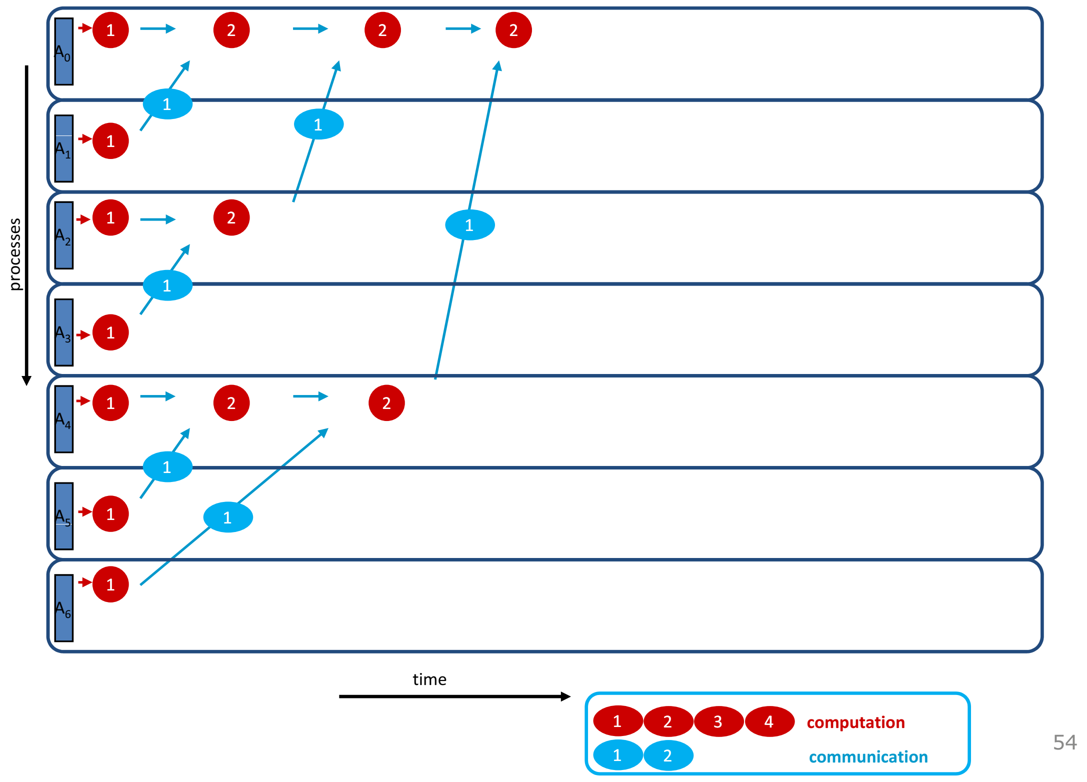
The big picture ….



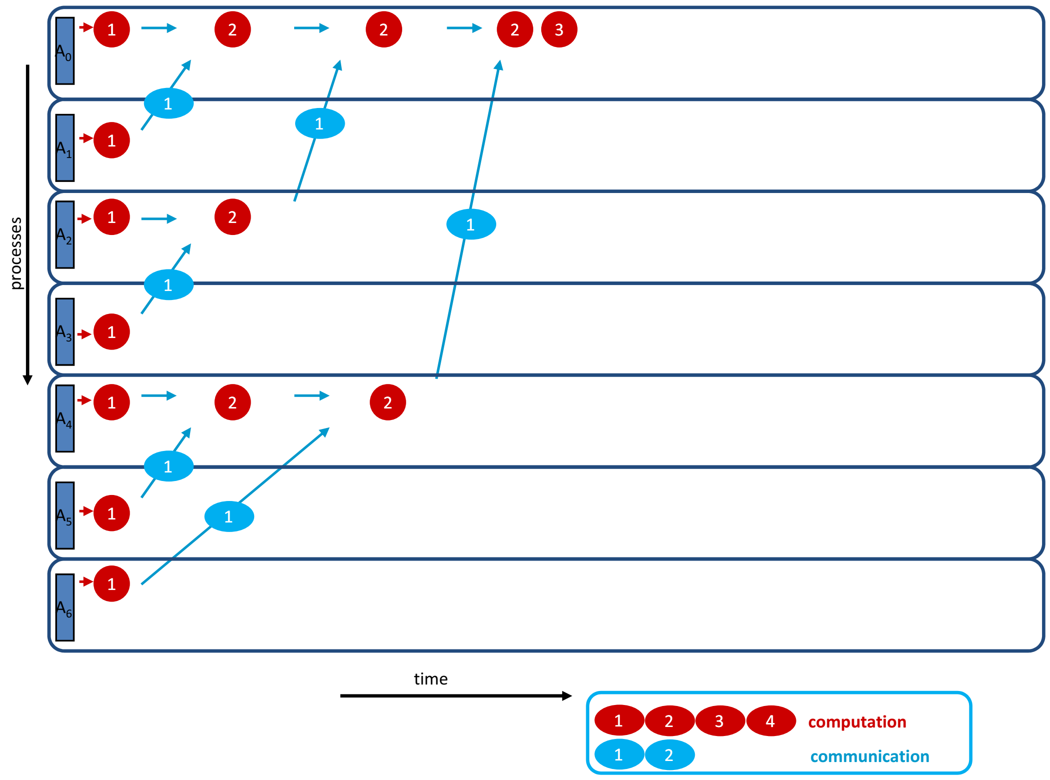
The big picture ....



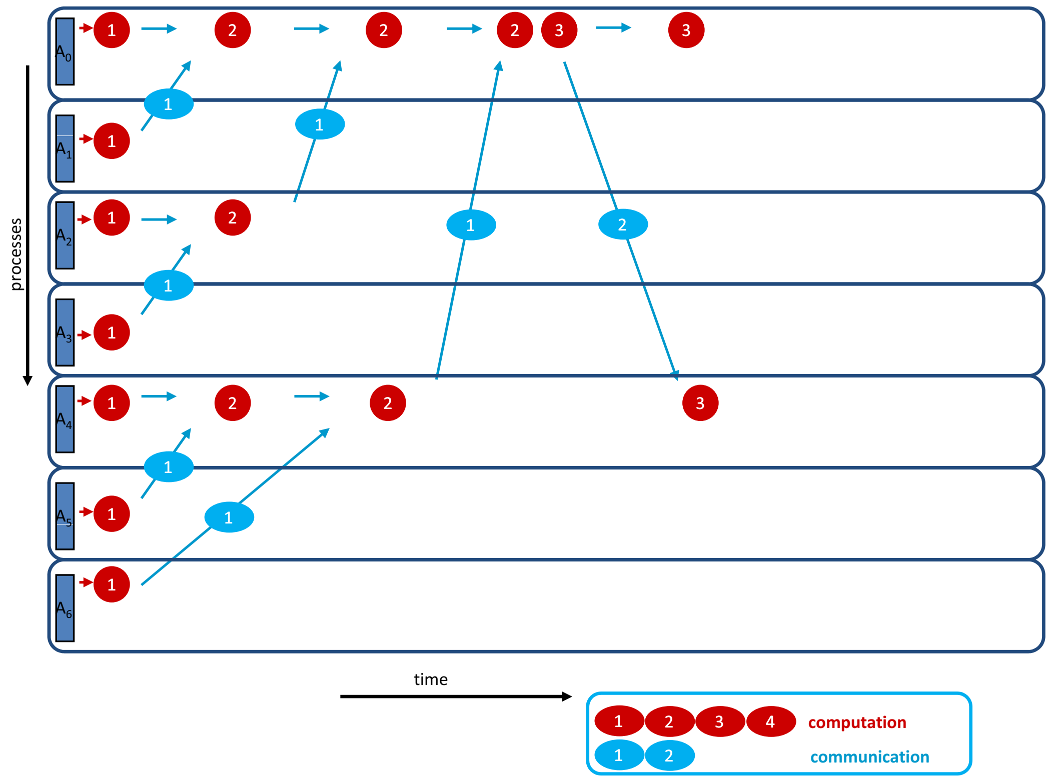
The big picture ....



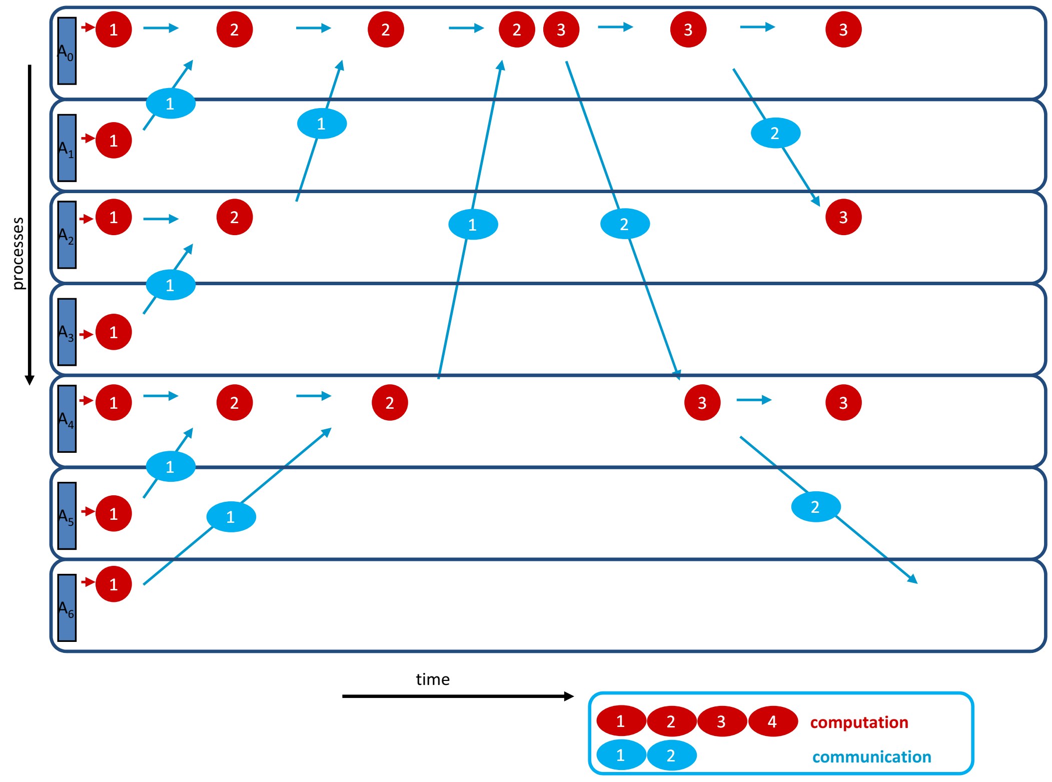
The big picture ….



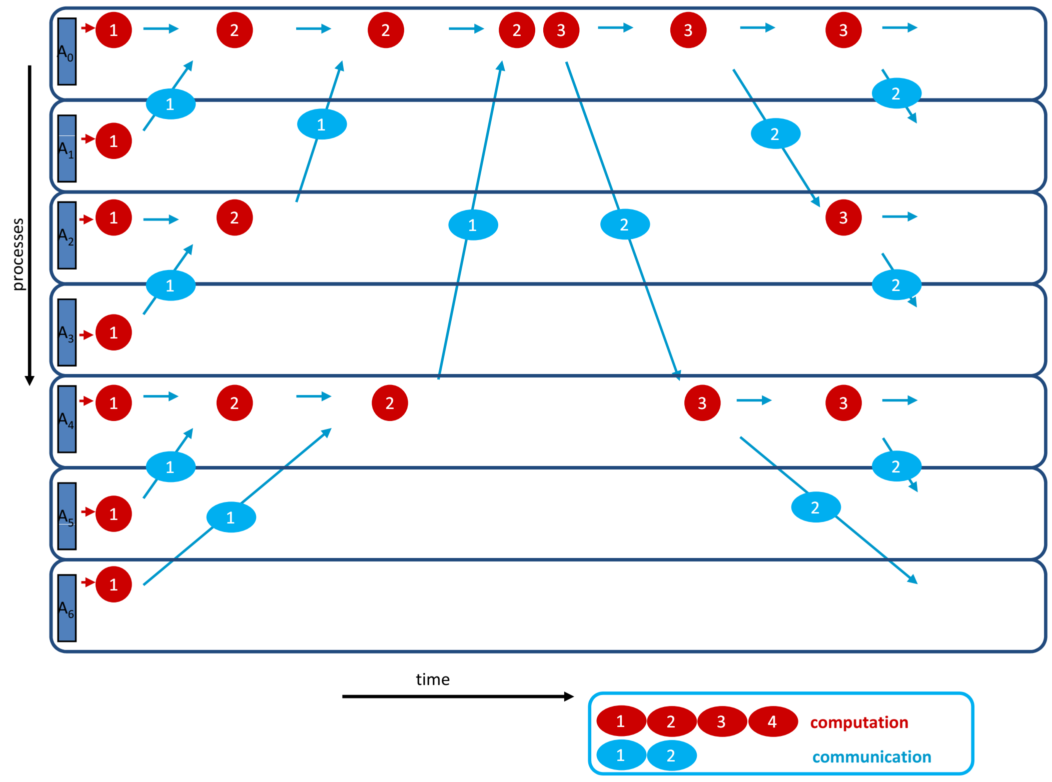
The big picture ….



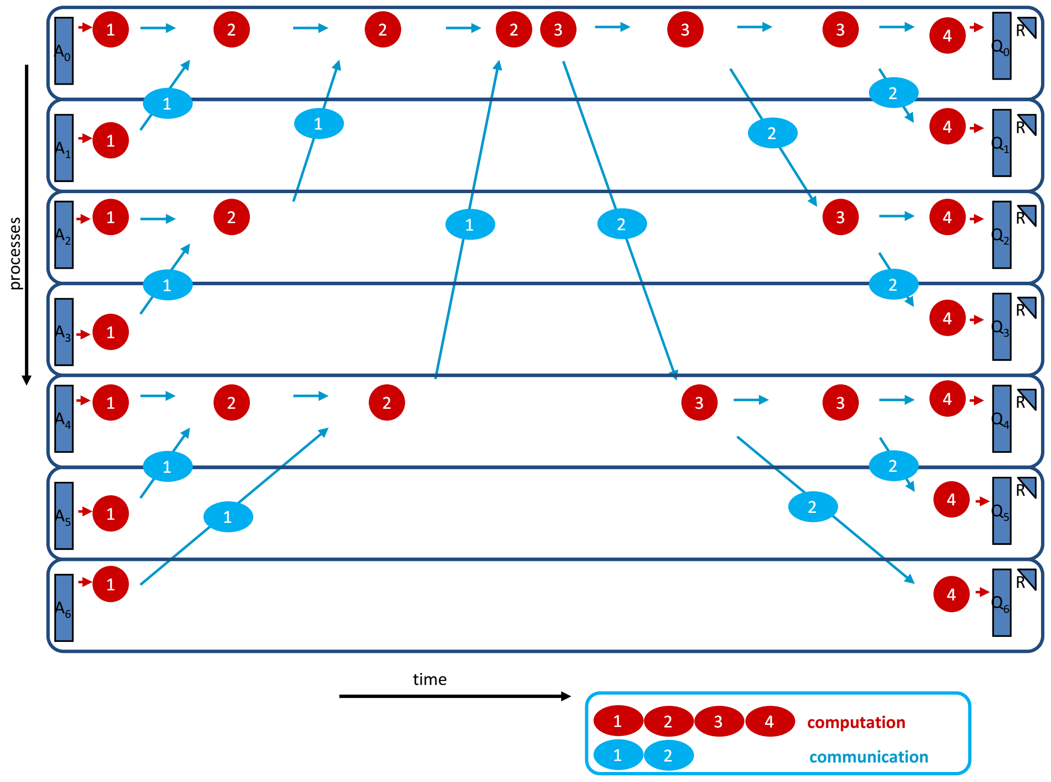
The big picture ….



The big picture ….



The big picture ....

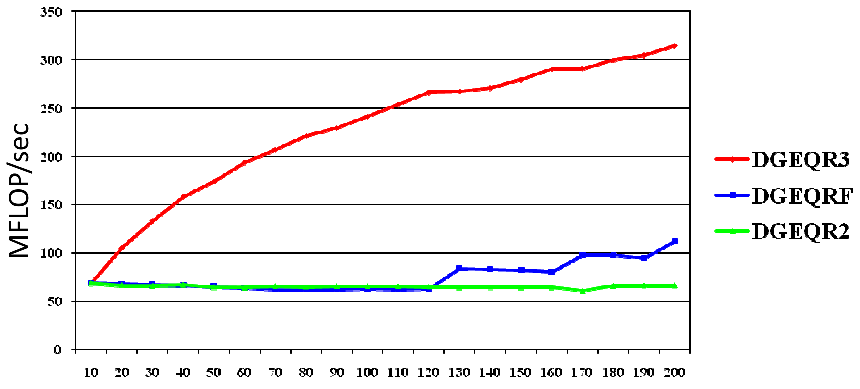


Latency but also possibility of fast panel factorization.

- **DGEQR3** is the recursive algorithm (see Elmroth and Gustavson, 2000), **DGEQRF** and **DGEQR2** are the LAPACK routines.
- Times include QR and DLARFT.
- Run on Pentium III.

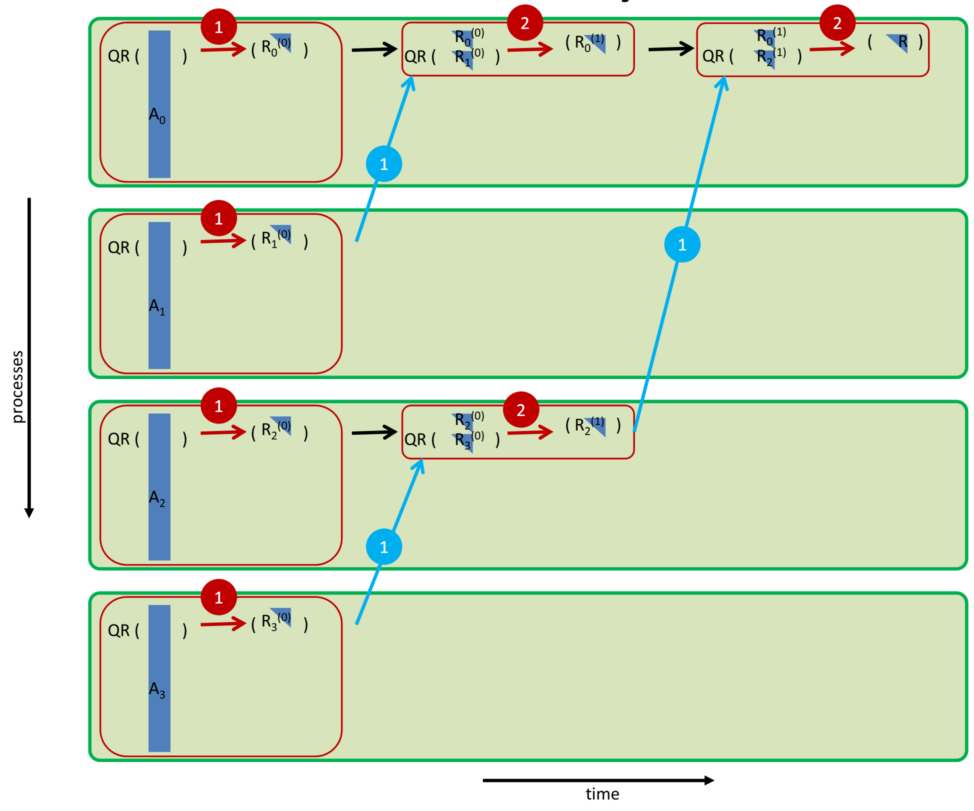
QR factorization and construction of T
m = 10,000
Perf in MFLOP/sec (Times in sec)

n	DGEQR3		DGEQRF		DGEQR2	
50	173.6	(0.29)	65.0	(0.77)	64.6	(0.77)
100	240.5	(0.83)	62.6	(3.17)	65.3	(3.04)
150	277.9	(1.60)	81.6	(5.46)	64.2	(6.94)
200	312.5	(2.53)	111.3	(7.09)	65.9	(11.98)



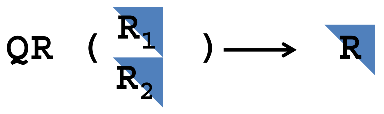
m=1000,000, the x axis is n

When only R is wanted



When only R is wanted: The MPI_Allreduce

In the case where only R is wanted, instead of constructing our own tree, one can simply use MPI_Allreduce with a user defined operation. The operation we give to MPI is basically the Algorithm 2. It performs the operation:



This **binary** operation is **associative** and this is all MPI needs to use a user-defined operation on a user-defined datatype. Moreover, if we change the signs of the elements of R so that the diagonal of R holds positive elements then the binary operation `Rfactor` becomes **commutative**.

The code becomes two lines:

```
lapack_dgeqrf( mloc, n, A, lda, tau, &dlwork, lwork, &info );
MPI_Allreduce( MPI_IN_PLACE, A, 1, MPI_UPPER,
        LILA_MPIOP_QR_UPPER, mpi_comm);
```

Does it work?

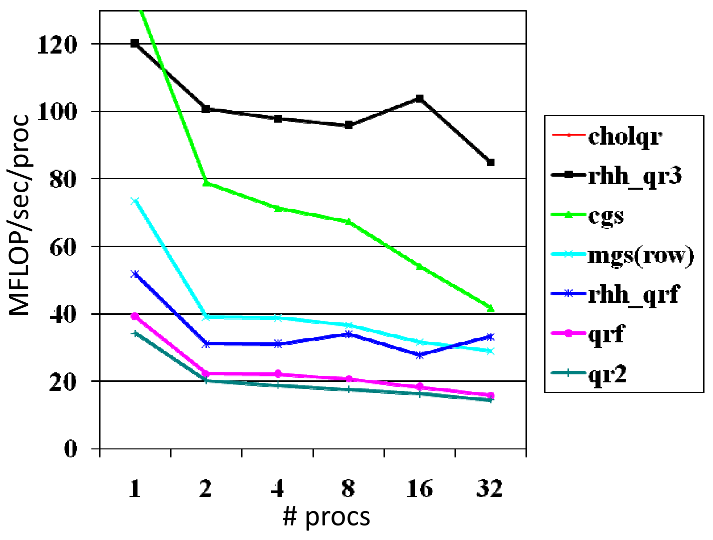
- The experiments are performed on the beowulf cluster at the University of Colorado at Denver. The cluster is made of 35 bi-pro Pentium III (900MHz) connected with Dolphin interconnect.
- Number of operations is taken as $2mn^2$ for all the methods
- The block size used in ScaLAPACK is 32.
- The code is written in C, use MPI (mpich-2.1), LAPACK (3.1.1), BLAS (goto-1.10), the LAPACK Cwrappers (http://icl.cs.utk.edu/~delmas/lapwrapmw.htm) and the BLAS C wrappers (http://www.netlib.org/blas/blast-forum/cblas.tgz)
- The codes has been tested in various configuration and have never failed to produce a correct answer, releasing those codes is in the agenda

Number of operations is taken as $2mn^2$ for all the methods

	FLOPs (total) for R only	FLOPs (total) for Q and R
CholeskyQR	$mn^2 + n^3/3$	$2mn^2 + n^3/3$
Gram-Schmidt	$2mn^2$	$2mn^2$
Householder	$2mn^2 - 2/3n^3$	$4mn^2 - 4/3n^3$
Allreduce HH	$(2mn^2 - 2/3n^3) + 2/3\, n^3\, p$	$(4mn^2 - 4/3n^3) + 4/3\, n^3\, p$

Q and R: Strong scalability

- In this experiment, we fix the problem: **m=100,000** and **n=50**. Then we increase the number of processors.
- Once more the algorithm **rhh_qr3** is the second behind **CholeskyQR**. Note that **rhh_qr3** is incondionnally stable while the stability of **CholeskyQR** depends on the square of the condition number of the initial matrix.

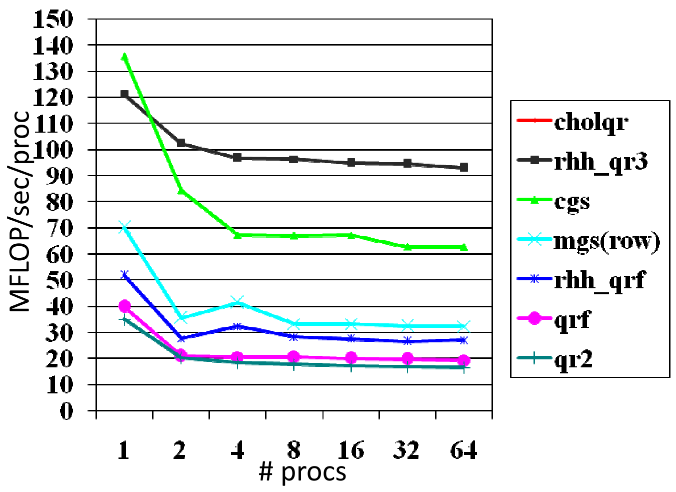


MFLOP/sec/proc Time in sec

# of procs	cholqr		rhh_qr3		cgs		mgs(row)		rhh_qrf		qrf		qr2	
1	489.2	(1.02)	120.0	(4.17)	134.1	(3.73)	73.5	(6.81)	51.9	(9.64)	39.1	(12.78)	34.3	(14.60)
2	467.3	(0.54)	100.8	(2.48)	78.9	(3.17)	39.0	(6.41)	31.2	(8.02)	22.3	(11.21)	20.2	(12.53)
4	466.4	(0.27)	97.9	(1.28)	71.3	(1.75)	38.7	(3.23)	31.0	(4.03)	22.2	(5.63)	18.8	(6.66)
8	434.0	(0.14)	95.9	(0.65)	67.4	(0.93)	36.7	(1.70)	34.0	(1.84)	20.8	(3.01)	17.7	(3.54)
16	359.2	(0.09)	103.8	(0.30)	54.2	(0.58)	31.6	(0.99)	27.8	(1.12)	18.3	(1.71)	16.3	(1.91)
32	197.8	(0.08)	84.9	(0.18)	41.9	(0.37)	29.0	(0.54)	33.3	(0.47)	15.8	(0.99)	14.5	(1.08)

Q and R: Weak scalability with respect to m

- We fix the local size to be **mloc=100,000** and **n=50**. When we increase the number of processors, the global m grows proportionally.
- **rhh_qr3** is the Allreduce algorithm with recursive panel factorization, **rhh_qrf** is the same with LAPACK Householder QR. We see the obvious benefit of using recursion. See as well (6). **qr2** and **qrf** correspond to the ScaLAPACK Householder QR factorization routines.

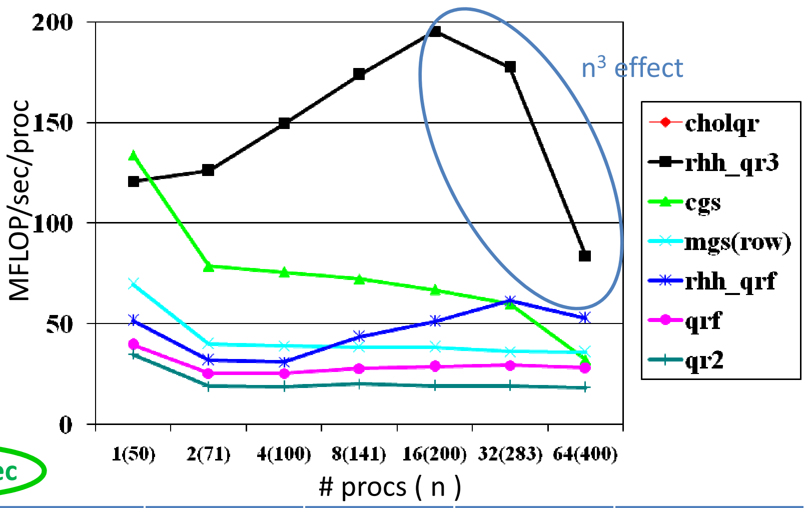


MFLOP/sec/proc Time in sec

# of procs	cholqr		rhh_qr3		Cgs		mgs(row)		rhh_qrf		qrf		qr2	
1	489.2	(1.02)	121.2	(4.13)	135.7	(3.69)	70.2	(7.13)	51.9	(9.64)	39.8	(12.56)	35.1	(14.23)
2	466.9	(1.07)	102.3	(4.89)	84.4	(5.93)	35.6	(14.04)	27.7	(18.06)	20.9	(23.87)	20.2	(24.80)
4	454.1	(1.10)	96.7	(5.17)	67.2	(7.44)	41.4	(12.09)	32.3	(15.48)	20.6	(24.28)	18.3	(27.29)
8	458.7	(1.09)	96.2	(5.20)	67.1	(7.46)	33.2	(15.06)	28.3	(17.67)	20.5	(24.43)	17.8	(28.07)
16	451.3	(1.11)	94.8	(5.27)	67.2	(7.45)	33.3	(15.04)	27.4	(18.22)	20.0	(24.95)	17.2	(29.10)
32	442.1	(1.13)	94.6	(5.29)	62.8	(7.97)	32.5	(15.38)	26.5	(18.84)	19.8	(25.27)	16.9	(29.61)
64	414.9	(1.21)	93.0	(5.38)	62.8	(7.96)	32.3	(15.46)	27.0	(18.53)	19.4	(25.79)	16.6	(30.13)

Q and R: Weak scalability with respect to n

- We fix the global size **m=100,000** and then we increase n as sqrt(p) so that the workload mn^2 per processor remains constant.
- Due to better performance in the local factorization or SYRK, **CholeskyQR**, **rhh_q3** and **rhh_qrf** exhibit increasing performance at the beginning until the n^3 comes into play

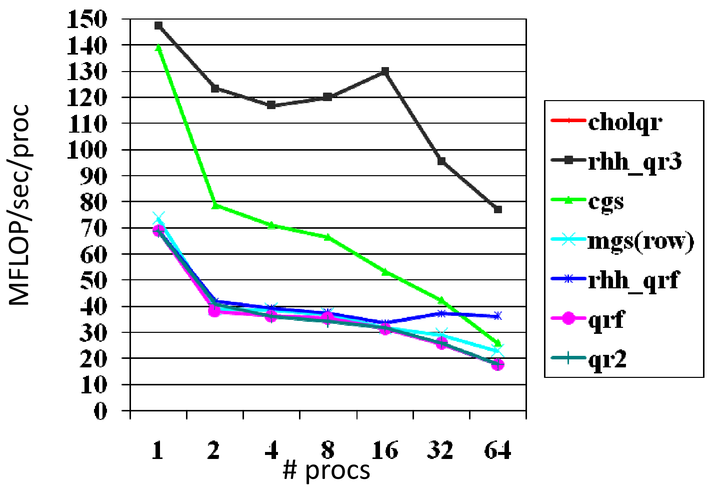


MFLOP/sec/proc Time in sec

# of procs	cholqr		rhh_qr3		cgs		mgs(row)		rhh_qrf		qrf		qr2	
1	490.7	(1.02)	120.8	(4.14)	134.0	(3.73)	69.7	(7.17)	51.7	(9.68)	39.6	(12.63)	39.9	(14.31)
2	510.2	(0.99)	126.0	(4.00)	78.6	(6.41)	40.1	(12.56)	32.1	(15.71)	25.4	(19.88)	19.0	(26.56)
4	541.1	(0.92)	149.4	(3.35)	75.6	(6.62)	39.1	(12.78)	31.1	(16.07)	25.5	(19.59)	18.9	(26.48)
8	540.2	(0.92)	173.8	(2.86)	72.3	(6.87)	38.5	(12.89)	43.6	(11.41)	27.8	(17.85)	20.2	(24.58)
16	501.5	(1.00)	195.2	(2.56)	66.8	(7.48)	38.4	(13.02)	51.3	(9.75)	28.9	(17.29)	19.3	(25.87)
32	379.2	(1.32)	177.4	(2.82)	59.8	(8.37)	36.2	(13.84)	61.4	(8.15)	29.5	(16.95)	19.3	(25.92)
64	266.4	(1.88)	83.9	(5.96)	32.3	(15.46)	36.1	(13.84)	52.9	(9.46)	28.2	(17.74)	18.4	(27.13)

R only: Strong scalability

- In this experiment, we fix the problem: **m=100,000** and **n=50**. Then we increase the number of processors.

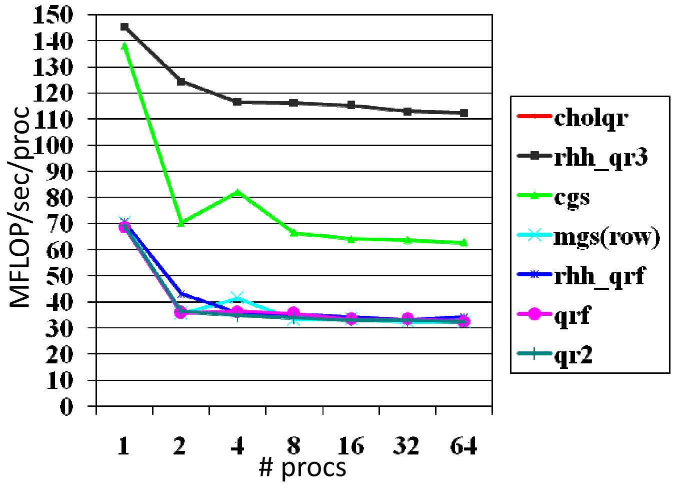


MFLOP/sec/proc Time in sec

# of procs	cholqr		rhh_qr3		cgs		mgs(row)		rhh_qrf		qrf		qr2	
1	1099.046	(0.45)	147.6	(3.38)	139.309	(3.58)	73.5	(6.81)	69.049	(7.24)	69.108	(7.23)	68.782	(7.27)
2	1067.856	(0.23)	123.424	(2.02)	78.649	(3.17)	39.0	(6.41)	41.837	(5.97)	38.008	(6.57)	40.782	(6.13)
4	1034.203	(0.12)	116.774	(1.07)	71.101	(1.76)	38.7	(3.23)	39.295	(3.18)	36.263	(3.44)	36.046	(3.47)
8	876.724	(0.07)	119.856	(0.52)	66.513	(0.94)	36.7	(1.70)	37.397	(1.67)	35.313	(1.77)	34.081	(1.83)
16	619.02	(0.05)	129.808	(0.24)	53.352	(0.59)	31.6	(0.99)	33.581	(0.93)	31.339	(0.99)	31.697	(0.98)
32	468.332	(0.03)	95.607	(0.16)	42.276	(0.37)	29.0	(0.54)	37.226	(0.42)	25.695	(0.60)	25.971	(0.60)
64	195.885	(0.04)	77.084	(0.10)	25.89	(0.30)	22.8	(0.34)	36.126	(0.22)	17.746	(0.44)	17.725	(0.44)

R only: Weak scalability with respect to m

- We fix the local size to be **mloc=100,000** and **n=50**. When we increase the number of processors, the global m grows proportionally.



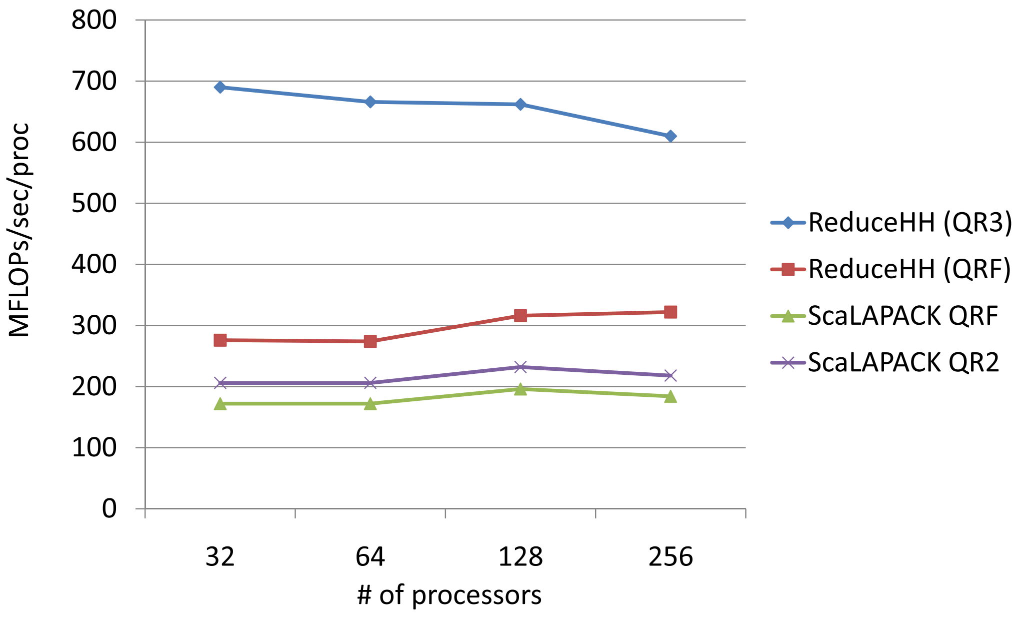
MFLOP/sec/proc Time in sec

# of procs	cholqr		rhh_qr3		cgs		mgs(row)		rhh_qrf		qrf		qr2	
1	1098.7	(0.45)	145.4	(3.43)	138.2	(3.61)	70.2	(7.13)	70.6	(7.07)	68.7	(7.26)	69.1	(7.22)
2	1048.3	(0.47)	124.3	(4.02)	70.3	(7.11)	35.6	(14.04)	43.1	(11.59)	35.8	(13.95)	36.3	(13.76)
4	1044.0	(0.47)	116.5	(4.29)	82.0	(6.09)	41.4	(12.09)	35.8	(13.94)	36.3	(13.74)	34.7	(14.40)
8	993.9	(0.50)	116.2	(4.30)	66.3	(7.53)	33.2	(15.06)	35.1	(14.21)	35.5	(14.05)	33.8	(14.75)
16	918.7	(0.54)	115.2	(4.33)	64.1	(7.79)	33.3	(15.04)	34.0	(14.66)	33.4	(14.94)	33.0	(15.11)
32	950.7	(0.52)	112.9	(4.42)	63.6	(7.85)	32.5	(15.38)	33.4	(14.95)	33.3	(15.01)	32.9	(15.19)
64	764.6	(0.65)	112.3	(4.45)	62.7	(7.96)	32.3	(15.46)	34.0	(14.66)	32.6	(15.33)	32.3	(15.46)

Blue Gene L
frost.ncar.edu

Q and R: Strong scalability

In this experiment, we fix the problem: **m=1,000,000** and **n=50**.
Then we increase the number of processors.



Conclusions

We have described a new method for the Householder QR factorization of skinny matrices. The method is named **Allreduce Householder** and has four advantages:

1. there is **only one synchronization point** in the algorithm,
2. the method **harvests most of efficiency of the computing unit** by large local operations,
3. the method is **stable**,
4. and finally the method is **elegant** in particular in the case where only R is needed.

Allreduce algorithms have been depicted here with Householder QR factorization. However it can be applied to *anything* for example Gram-Schmidt or LU.

Current development is in writing a 2D block cyclic QR factorization and LU factorization based on those ideas.

What will a petascale machine looks like.

(aka what I learned during this conference.)

Possible petascale machine	
1. Number of cores per nodes	10 – 100 cores
2. Performance per nodes	100 – 1,000 GFLOPs/sec
3. Number of nodes	1000-10,000 nodes
4. Latency inter-nodes	1 µsec
5. Bandwidth inter-nodes	10 Gb/sec
6. Memory per nodes	10 GB

- Part I: First rule in linear algebra: Have an efficient DGEMM
  - Motivation in
    2. performance per node 5. bandwidth inter-nodes 6. memory per nodes
- **Part II: Algorithm for multicore / latency avoiding algorithm for LU, QR …**
  - **Motivation in:**
    1. Number of cores per node 2. performance per node 4. Latency inter-nodes
- Part III: Algorithm for fault tolerance
  - Motivation in:
    1. Number of cores per node 3. number of nodes

What will a petascale machine looks like.

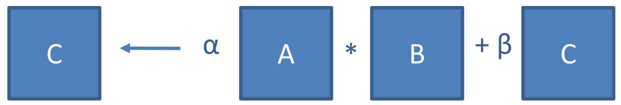
(aka what I learned during this conference.)

Possible petascale machine	
1. Number of cores per nodes	10 – 100 cores
2. Performance per nodes	100 – 1,000 GFLOPs/sec
3. Number of nodes	1000-10,000 nodes
4. Latency inter-nodes	1 µsec
5. Bandwidth inter-nodes	10 Gb/sec
6. Memory per nodes	10 GB

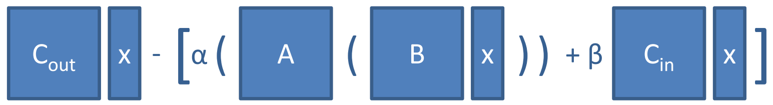
- Part I: First rule in linear algebra: Have an efficient DGEMM
  - Motivation in
    2. performance per node 5. bandwidth inter-nodes 6. memory per nodes
- Part II: Algorithm for multicore / latency avoiding algorithm for LU, QR …
  - Motivation in:
    1. Number of cores per node 2. performance per node 4. Latency inter-nodes
- **Part III: Algorithm for fault tolerance**
  - **Motivation in:**
    1. Number of cores per node 3. number of nodes

Experiments on MM

- Goal:
  – Write a FT-PDGEMM (Fault Tolerant matrix Matrix Multiply)



- Testing:
  – Perform FT-PDGEMM in a loop and check results with residual checking



on top of this add on automatic process killer.

ABFT-BLAS: a Parallel Fault Tolerant library BLAS based on ABFT techniques

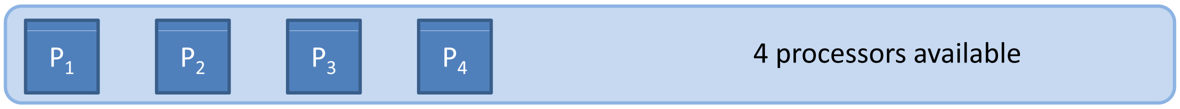
- Constructed on top of FT-MPI.
- Provides users a fault-tolerant environment:
  – Detect failures
  – Recover data automatically
  – Enables the user to stack computational routines the one on top of the others
  – Goal: research library for conducting experiments on fault tolerance
- Provides us with an automatic process killer

EXAMPLE CODE

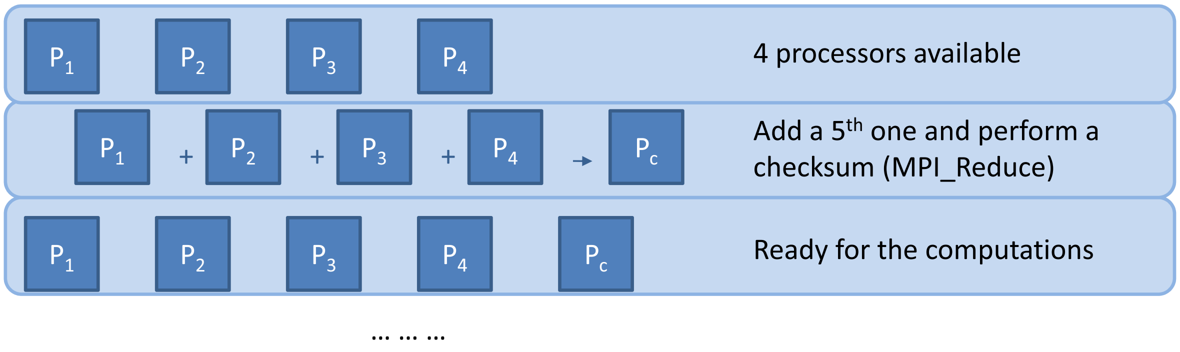
```
int rc;
struct Vector v;
struct Matrix a;
struct Dataworld worldmpi;
struct Global_ddata normv;
struct Global_idata nbr_iter;
...
rc = MPI_Init(&argc, &argv);
rc = init_world(&worldmpi, p, q, rc);
rc = get_info_on_grid(&worldmpi, &me,
                      &myrow, &mycol, &nprow, &npcol);
...
rc = allocate_vector(&v, POS_ROW, 0, nb_n, &worldmpi, "v");
rc = allocate_matrix(&a, m, n, nb_m, nb_n, &worldmpi, "a");
rc = allocate_dglobal(&normv, 1, &worldmpi);
rc = allocate_iglobal(&nbr_iter, 1, &worldmpi);
...
if (!worldmpi.recovering)
{
... here goes the user code to initialize objects ...
rc = make_checksum_matrix(&a, &worldmpi);
rc = make_checksum_vector(&v, &worldmpi);
}
```

```
if (worldmpi.user_state == 0)
{
rc = ftdnrm2(&worldmpi, &v, normv.data);
worldmpi.user_state = 1;
}
if (worldmpi.user_state == 1)
{
... here goes any call to the ABFT-BLAS numerical routines ...
worldmpi.user_state = 2;
}

free_vector(&v);
free_matrix(&a);
free_dglobal(&normv);
free_iglobal(&nbr_iter);
exit(0);
```

Diskless checkpointing



Diskless checkpointing

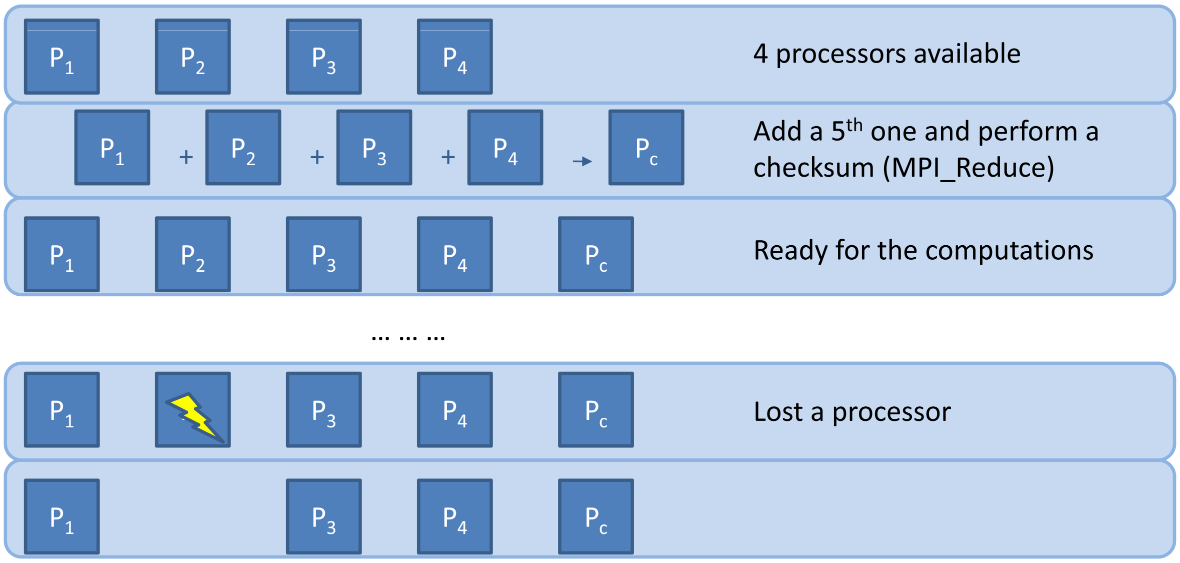

P_1 P_2 P_3 P_4
$P_1 + P_2 + P_3 + P_4 \rightarrow P_c$
P_1 P_2 P_3 P_4 P_c
… … …

4 processors available

Add a 5th one and perform a checksum (MPI_Reduce)

Ready for the computations

Diskless checkpointing



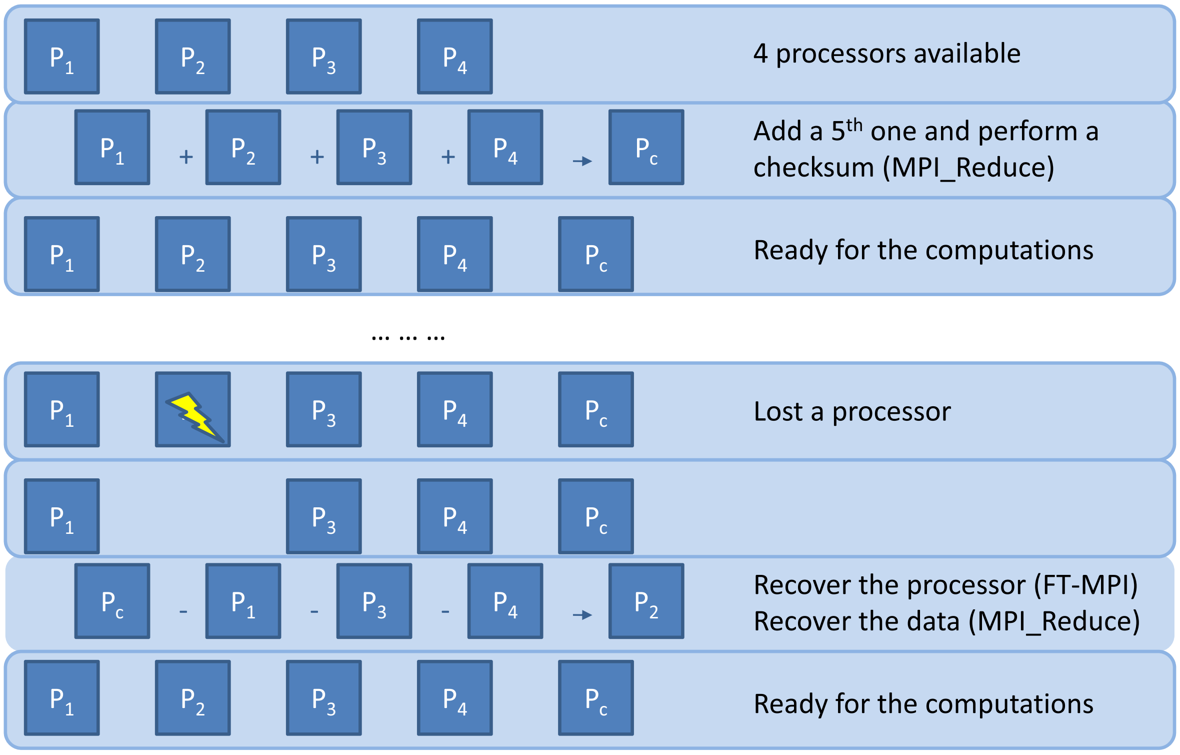
4 processors available

Add a 5th one and perform a checksum (MPI_Reduce)

Ready for the computations

Lost a processor

Diskless checkpointing



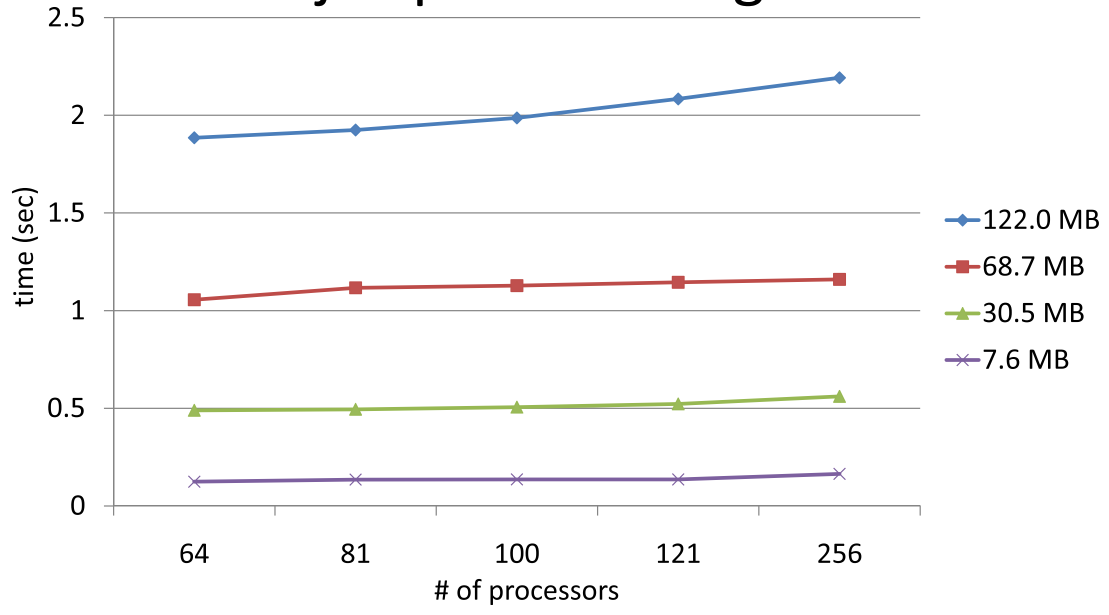
P_1 P_2 P_3 P_4
4 processors available
$P_1 + P_2 + P_3 + P_4 \rightarrow P_c$
Add a 5th one and perform a checksum (MPI_Reduce)
P_1 P_2 P_3 P_4 P_c
Ready for the computations
… … …
P_1 P_3 P_4 P_c
Lost a processor
P_1 P_3 P_4 P_c
$P_c - P_1 - P_3 - P_4 \rightarrow P_2$
Recover the processor (FT-MPI)
Recover the data (MPI_Reduce)
P_1 P_2 P_3 P_4 P_c
Ready for the computations

Diskless checkpointing (remarks)

- You can use either floating-point arithmetic or binary arithmetic for the checksum

- Multiple failures supported through Reed-Solomon algorithm, optimal algorithm in the sense that, to support p simultaneous failures, only need to add p processes.

Time for a MPI_Reduce (using MVAPICH) on Infiniband on jacquard.nersc.gov

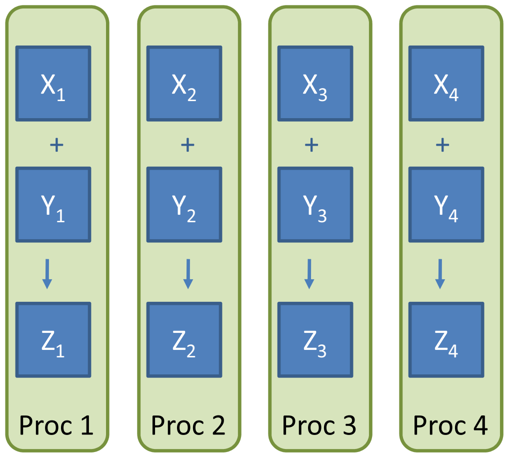


ABFT = Algorithm Based Fault Tolerance.

- K. Huang, J. Abraham, "*Algorithm-Based Fault Tolerance for Matrix Operations*," IEEE Trans. on Comp. (Spec. Issue Reliable & Fault-Tolerant Comp.), C-33, 1984, pp. 518-528.
- If checkpoints are performed in floating-point arithmetic then we can exploit the linearity of the mathematical relations on the object to maintain the checksums

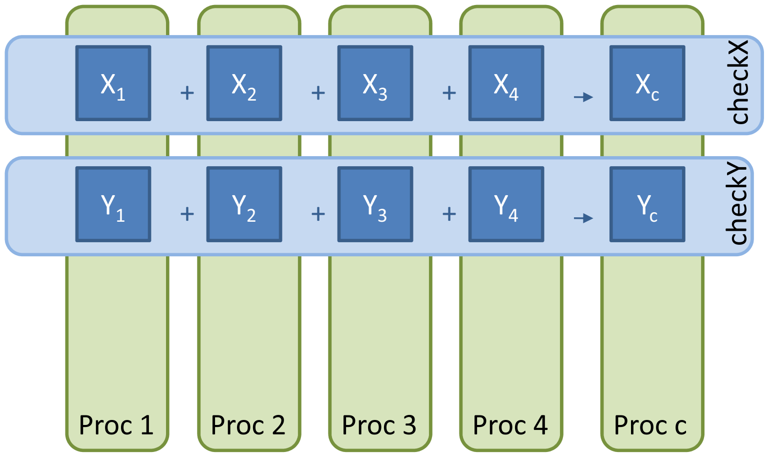
ABFT concept in an example

We want to perform z = x+y.



ABFT concept in an example

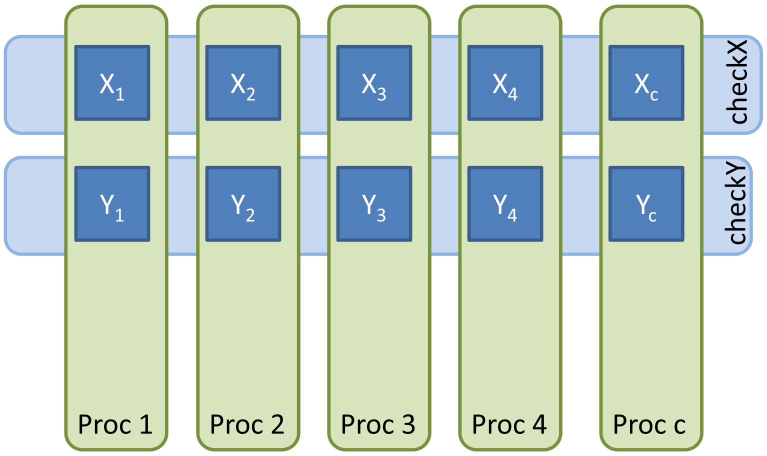
We want to perform z = x+y.



X_1 + X_2 + X_3 + X_4 → X_c
checkX
Y_1 + Y_2 + Y_3 + Y_4 → Y_c
checkY
Proc 1
Proc 2
Proc 3
Proc 4
Proc c

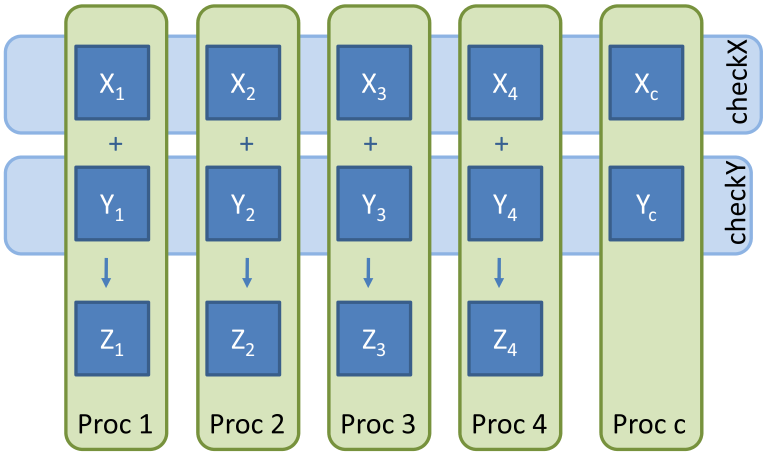
ABFT concept in an example

We want to perform z = x+y.



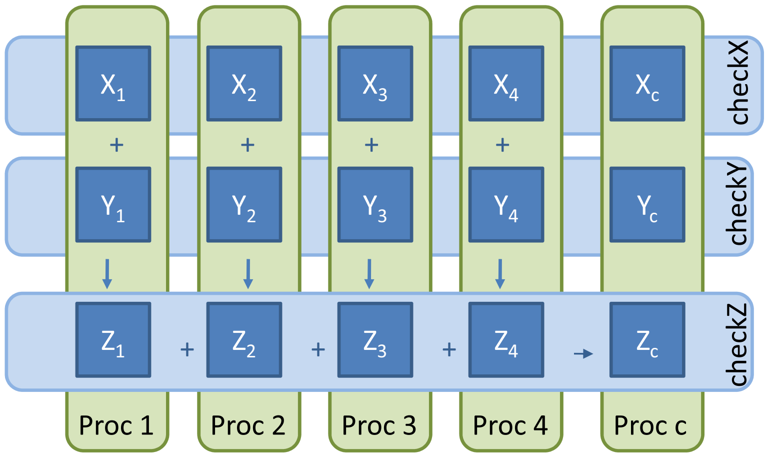
ABFT concept in an example

We want to perform z = x+y.



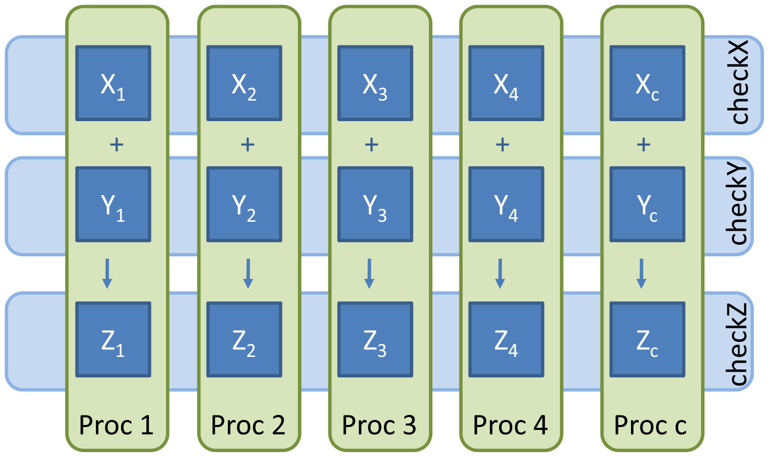
ABFT concept in an example

We want to perform z = x+y.



ABFT concept in an example

We want to perform z = x+y.



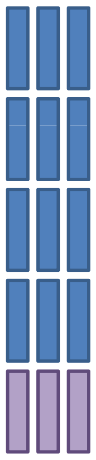
No overhead to compute the checksum of Z.

Same idea apply to the operation λ x + μ y

ABFT summary.

- Relies on floating-point arithmetic checksums
- Exploit the checksum processors
- Algorithms exist for any linear operations:
  - AXPY, SCAL, (BLAS1)
  - GEMV (BLAS2)
  - GEMM (BLAS3)
  - LU, QR, Cholesky (LAPACK)
  - FFT

An example with modified Gram-Schmidt.



A nonsingular m x 3

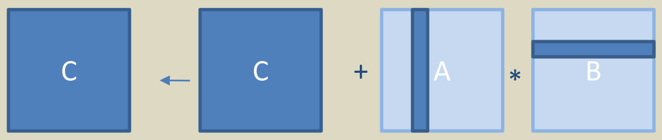
```
$Q = A$

$r_{11} = || Q_1 ||_2$
$Q_1 = Q_1 / r_{11}$

$r_{12} = Q_1^T Q_2$
$Q_2 = Q_2 - Q_1 r_{12}$
$r_{22} = || Q_2 ||_2$
$Q_2 = Q_2 / r_{22}$

$r_{13} = Q_1^T Q_3$
$Q_3 = Q_3 - Q_1 r_{13}$
$r_{23} = Q_2^T Q_3$
$Q_3 = Q_3 - Q_2 r_{23}$
$r_{33} = || Q_3 ||_2$
$Q_3 = Q_3 / r_{33}$
```

$A = QR \quad Q^TQ = I_3$

PDGEMM.

PDGEMM :

```
For k = 1:nb:n,
```



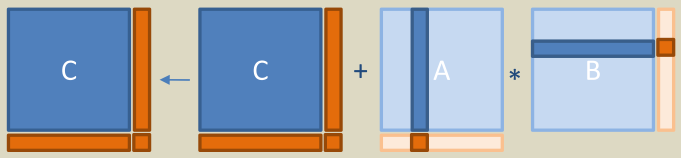
```
End For
```

PDGEMM-SUMMA
$\frac{2n^3}{p}\gamma + 2(n + 2\sqrt{p} - 3)(\frac{n}{\sqrt{p}}\beta)$

ABFT-PDGEMM.

ABFT-PDGEMM :

```
For k = 1:nb:n,
```



```
End For
```

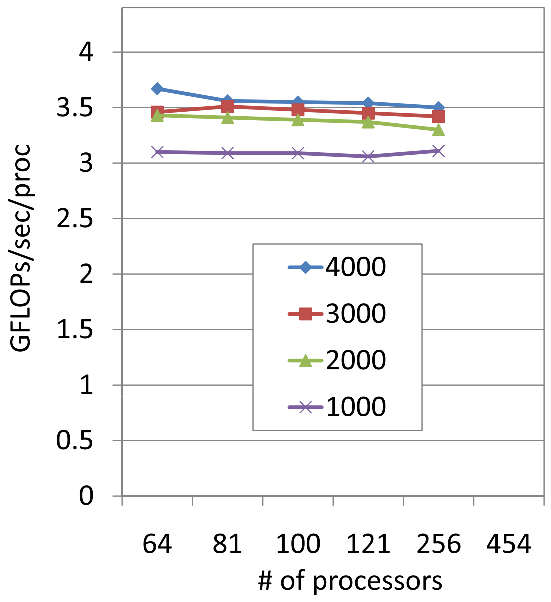
- The algorithm maintains the consistency of the checkpoints of the matrix C naturally.

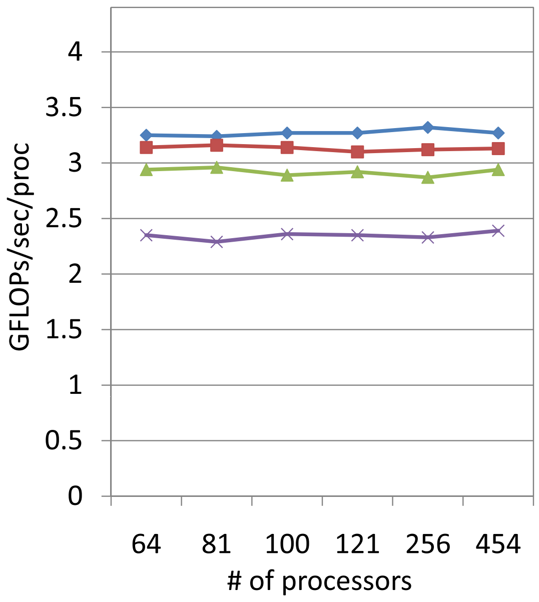
PDGEMM-SUMMA	ABFT-PDGEMM-SUMMA
$\frac{2n^3}{p}\gamma + 2(n + 2\sqrt{p} - 3)(\frac{n}{\sqrt{p}}\beta)$	$\frac{2n(n+nloc)^2}{p}\gamma + 2(n + 2\sqrt{p} - 3)(\frac{(n+nloc)}{\sqrt{p}}\beta)$

jacquard.nersc.gov

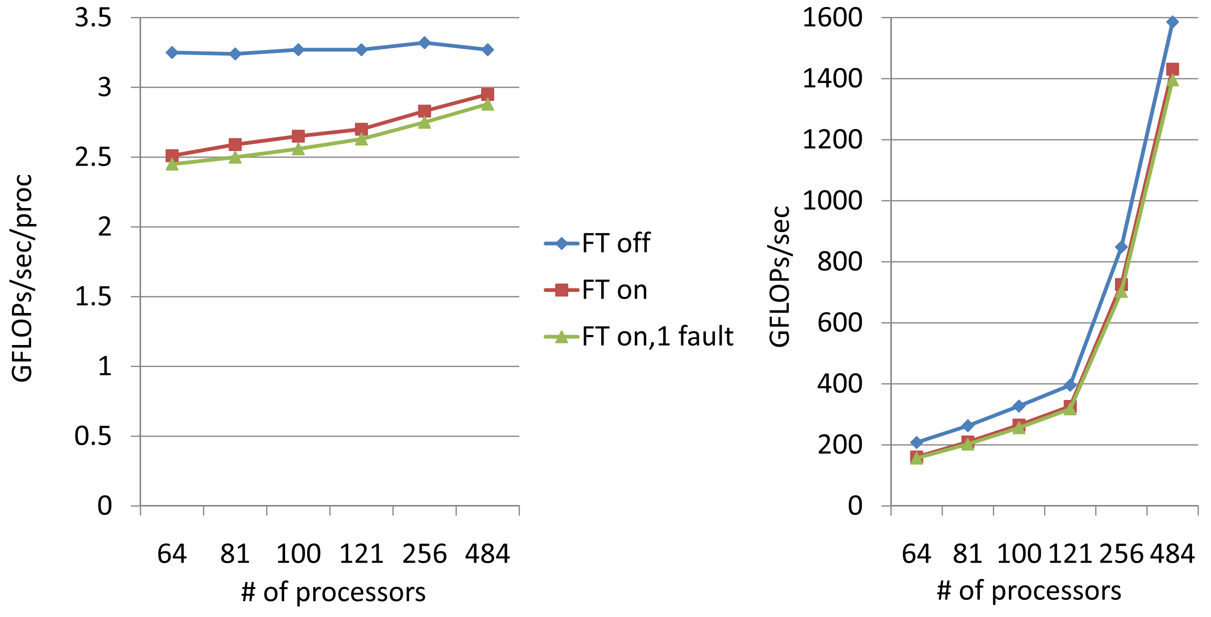
- **Processor type** Opteron 2.2 GHz
- **Processor theoretical peak** 4.4 GFlops/sec
- **Number of application processors** 712
- **System theoretical peak (computational nodes)** 3.13 TFlops/sec
- **Number of shared-memory application nodes** 356
- **Processors per node** 2
- **Physical memory per node** 6 GBytes
- **Usable memory per node** 3-5 GBytes
- **Switch Interconnect** InfiniBand
- **Switch MPI Unidirectional Latency** 4.5 µsec
- **Switch MPI Unidirectional Bandwidth (peak)** 620 MB/s
- **Global shared disk GPFS Usable disk space** 30 TBytes
- **Batch system** PBS Pro

Mvapich vs FTMPI

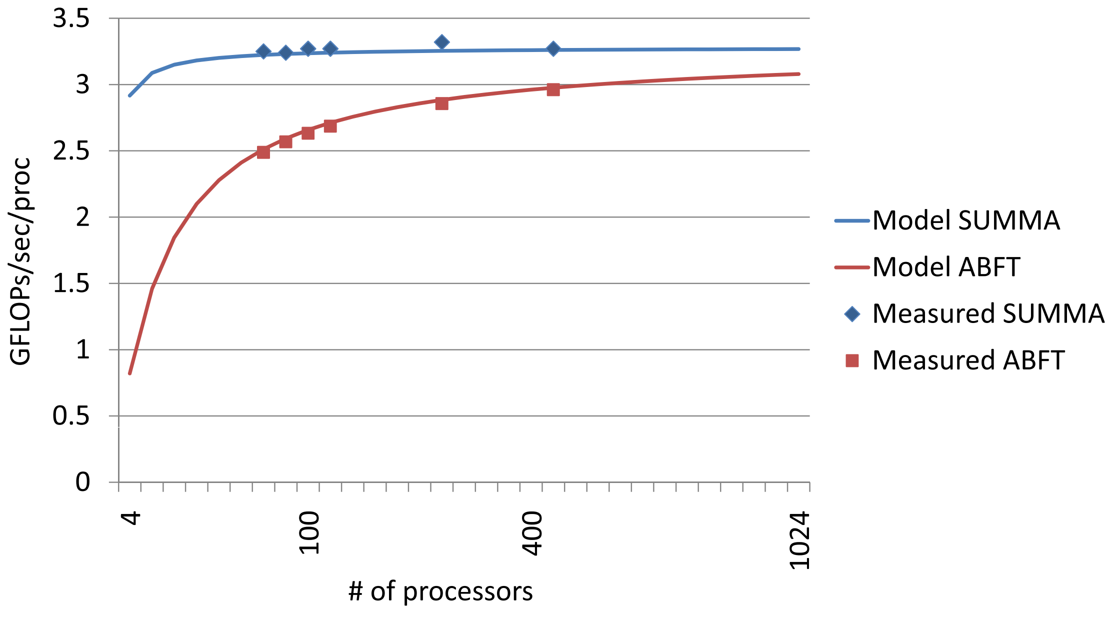




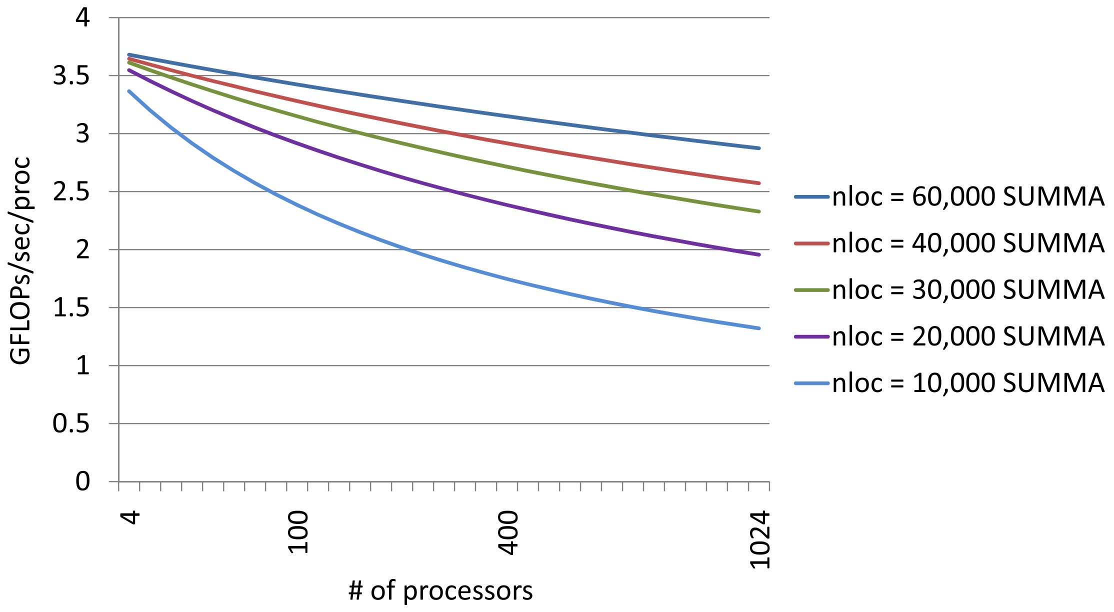
FT-PDGEMM -- nloc=4,000



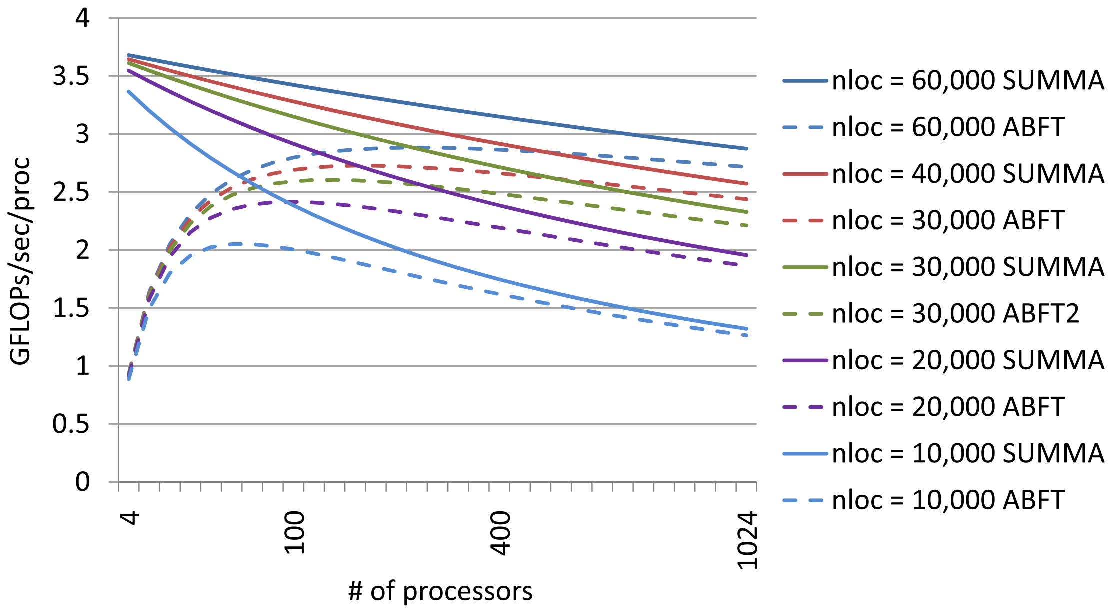
Performance modeling



Strong scalability

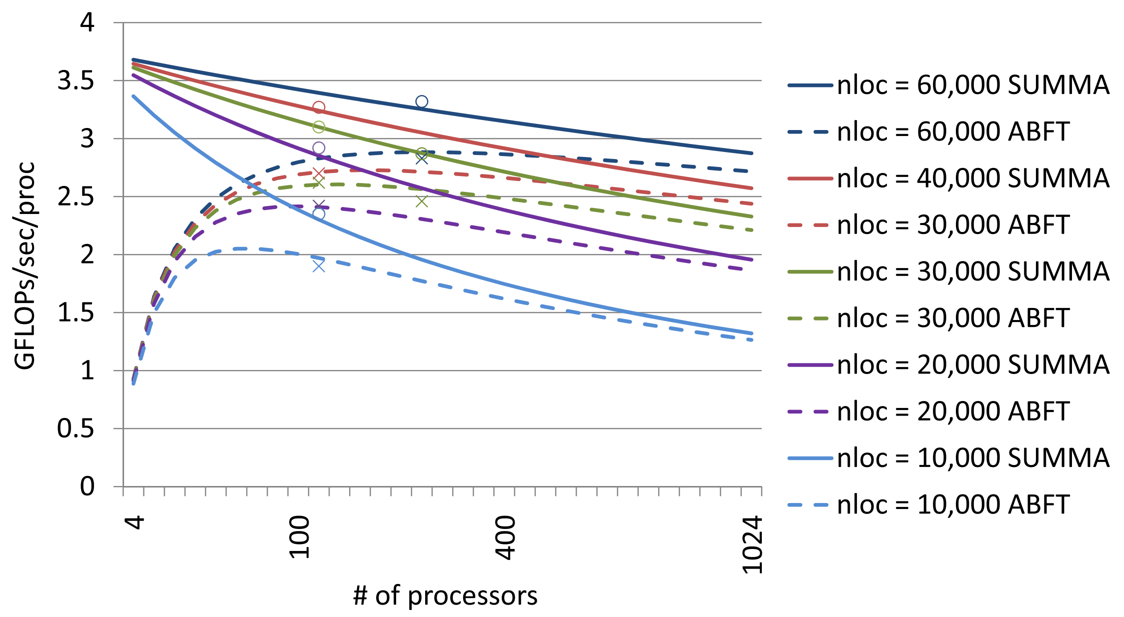


Strong scalability



ABFT represents the only known alternative to address fault tolerance in strong scalability

Strong scalability



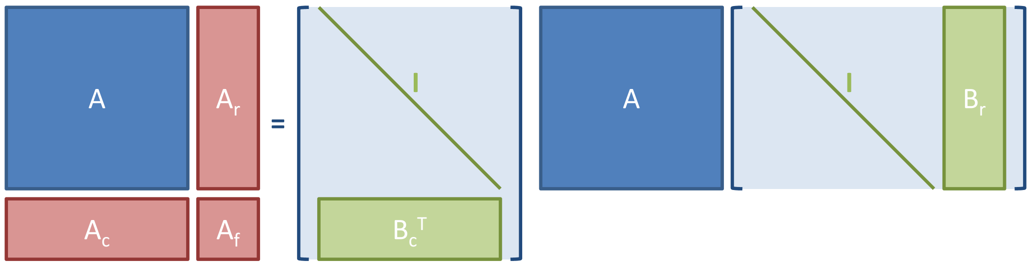
ABFT represents the only known alternative to address fault tolerance in strong scalability

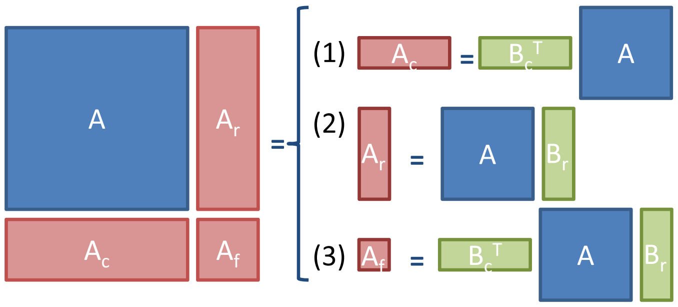
ABFT advantages:

- Independent of the Surface (n^2) / volume (n^3).
  - Important for n/n operations (e.g. FFT)
  - Important for MM with small n
- Independent of failure rate
  - No need to guess parameters
- Fits nicely in the algorithm
  - No need for explicit synchronization for example

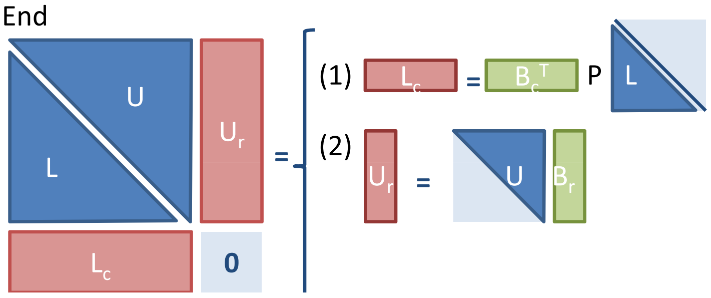
LU factorization:

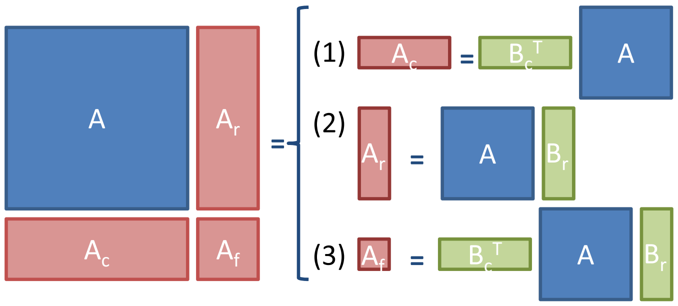
Starting from the encoding:



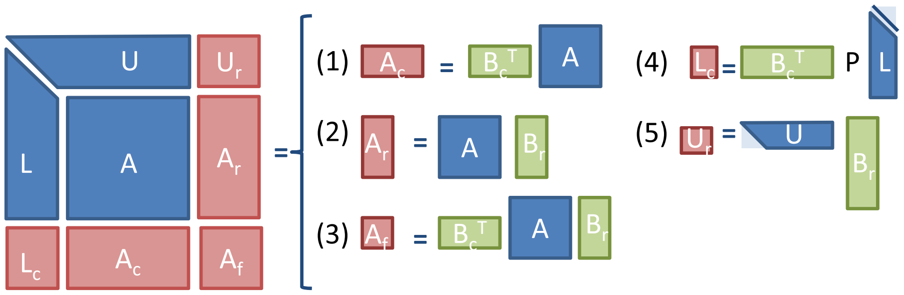


End

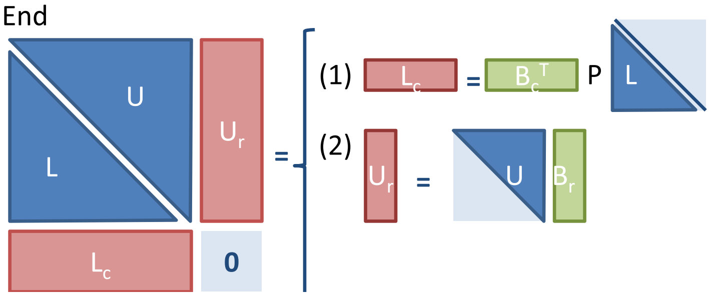




For j=1:nb:n,



End



Conclusions

If a petascale machine looks like.

Possible petascale machine	
1. Number of cores per nodes	10 – 100 cores
2. Performance per nodes	100 – 1,000 GFLOPs/sec
3. Number of nodes	1000-10,000 nodes
4. Latency inter-nodes	1 µsec
5. Bandwidth inter-nodes	10 Gb/sec
6. Memory per nodes	10 GB

- We will be able to have a fairly efficient DGEMM (which a sine-qua-non condition for having any DLA operation efficient.)
- Third generation of dense linear algebra algorithm coming to handle multicore, out-of-core and parallel distributed. Algorithms done in a year or so. A major software to appear in two-to-three years.
- Require support from compiler / language / scheduling to help in the constructing the framework. In particular for parallel distributed machines. (Some collaborations already on going.)
- Some work on fault-tolerance mature if fault tolerance needed.