



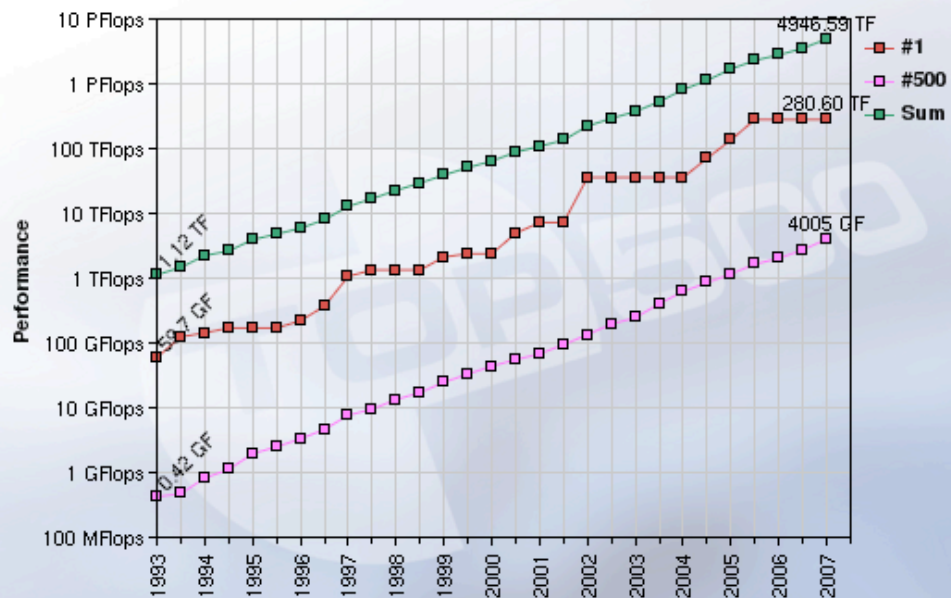
The Next Frontier

George Bosilca





Performance Development

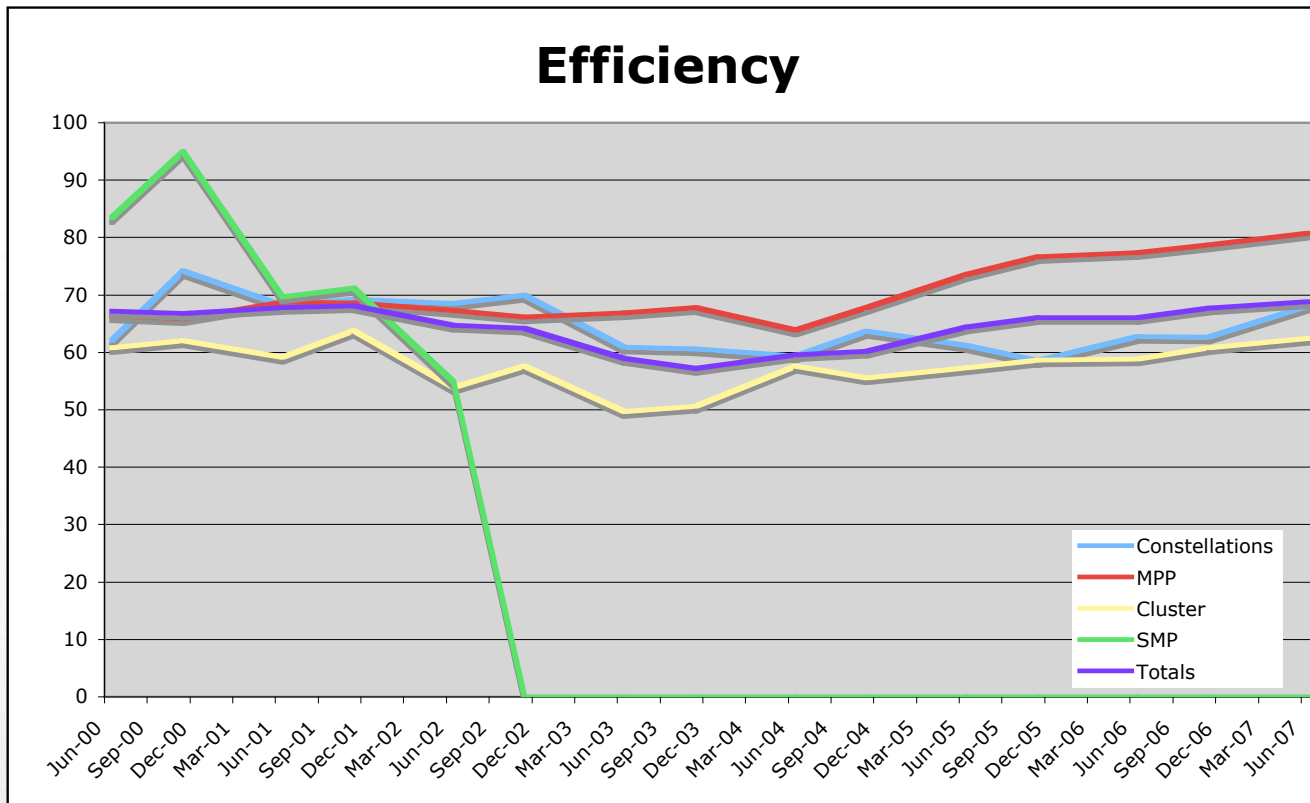


27/06/2007

<http://www.top500.org/>



Efficiency





Short version

- Network and processor
- Multi/Many cores
- Performance
- Programming model



Short version

- Network and proc **Fast and furious !**
- Multi/Many core **Hot, Hot and Hot !**
- Performance **Huge Potential**
- Program **MIA !** model



Programming Models

- Intel TBB, Codeplay Sieves, Cray's Chapel, Sun's Fortress, IBM's X10, HPF, UPC
- Open MP, Cilk, Nanos, CoArray Fortran
- Our requirements: Asynchronicity and Dynamic Scaling, Fine granularity and Unified approach for different memory architectures



Thread based MPI



MPI Processes

- MPI is process based, threads are external entities outside of MPI knowledge
- Point-to-point communications between threads are possible by crafting special tags
- Collectives are process based, one process participate in the collective once
- Threads fight for messages instead of collaborating



MPI Threads

- What if:
 - MPI became threads based, i.e. each thread get a rank
 - Each thread is allowed to behave as a MPI process today
- We can use a thread based programming approach, mixed with message synchronization and collective communication



MPI Threads

- What if:

- N

- Stay as close as possible to the current MPI standard (Nxl is a standard MPI application)

- E

- **MPI_COMM_WORLD** is still the same

- W

mix

communication

get a

rocess

roach,

ective



MPI_Init_thread

- `mpiexec -np NxM ...`
 - will start N processes and notify them that each will have at most M threads
- Extend the standard with `MPI_COMM_LOCAL` including all M local threads
- Each thread is required to call `MPI_Init_thread` to set its rank in the `MPI_COMM_LOCAL`



MPI ranks

- `MPI_COMM_LOCAL` is a fully featured intra-communicator
 - process based communicator vs. thread based communicator
- It can be used by any communicator creation function
- If any doubts about the rank of the thread in a communicator creation, the order will be based on the rank in the local communicator.

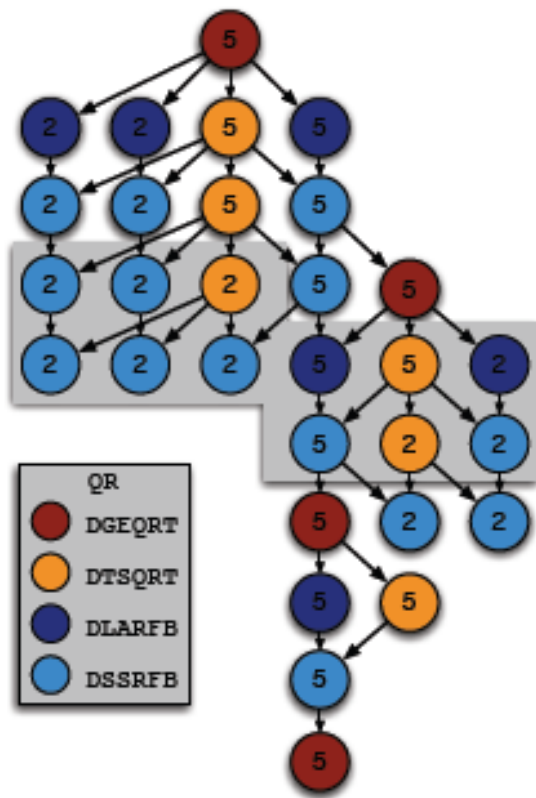


Receive Rules

- On the process based communicator such as `MPI_COMM_WORLD` all threads can match a receive
- On all mixed communicators the receives are named by rank (thread)
- Similar rules applies for collective communications, i.e. a process can participate multiple times in a collective.



PLASMA and DAGuE

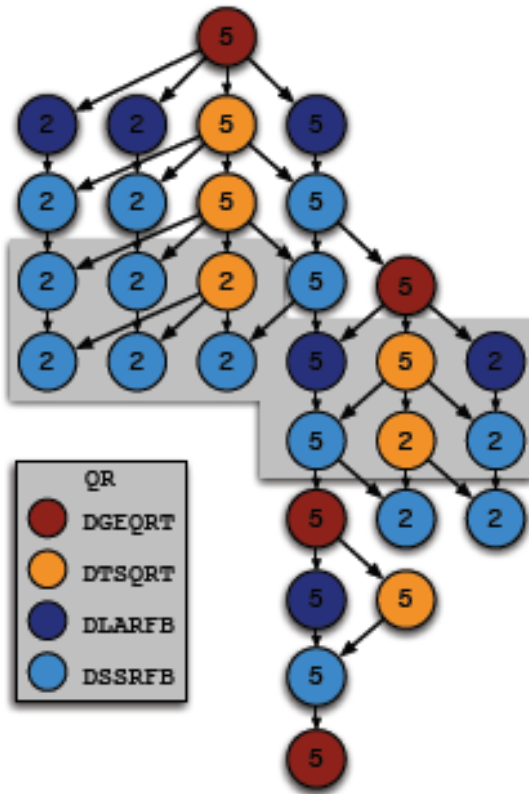


Tiles for QR Factorization

- Phred, Paralex, HeNCE and CODE, SCHEDULE (PVM based tools)
- acyclic representation of the algorithm as a directed graph with procedures attached to the nodes
- nodes are annotated with the list of input and output parameters
- special node for conditionals, loops and collective



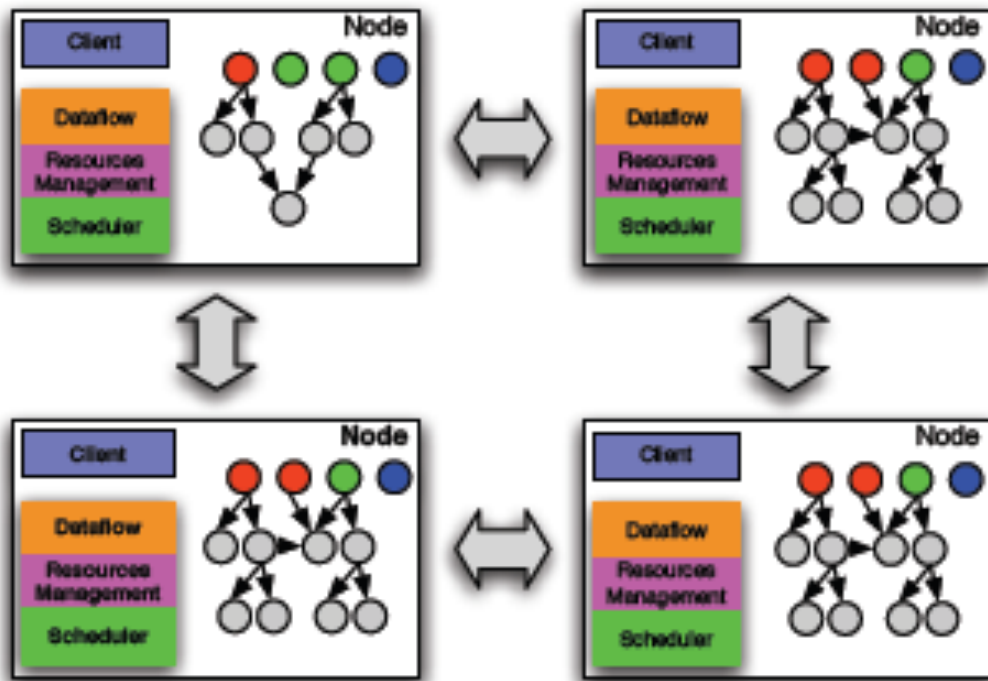
Challenges



- DAG construction and exploration
- initial approach: static partitioning and dynamic scheduling in each sub-domain
- “sliding window” approach
- Dynamic scheduling: trade between data reuse and aggressive pursuit of the critical path



SPMD/MPMD



- Some dependencies will point to local variables, while others point to external data
- Communications are implicit, and the scheduler can extract them from the DAG
- Potential for overlapping communications and computations



SPMD/MPMD

```
// Name
POTRF(k)

// Execution space
k = 0..SIZE

// Parallel partitioning
k % GRIDrows == rowRANK
k % GRIDcols == colRANK

// Parameters
INPUT T <- (k == 0) ? IN(k, k) : T SYRK(k, k)
      -> T TRSM(k, k+1..SIZE)
      -> OUT(k, k)

// Body
lapack_spotrf(lapack_lower, NB, T, NB, &info);
```

```
// Name
GEMM(k, m, n)

// Execution space
k = 0..SIZE
m = k+1..SIZE
n = 0..k-1

// Parallel partitioning
m % GRIDrows == rowRANK
k % GRIDcols == colRANK

// Parameters
IN  A <- C TRSM(n, k)
IN  B <- C TRSM(n, m)
INPUT C <- (n == 0) ? IN(m, n) : C GEMM(k, m, n-1)
      -> (n == k-1) ? C TRSM(k, m) : C GEMM(k, m, n+1)

// Body
cblas_sgemm(CblasColMajor,
           CblasNoTrans, CblasTrans,
           NB, NB, NB,
           1.0, B, NB, A, NB, 1.0, C, NB);
```

```
// Globals
GRIDrows, GRIDcols, NB
```

```
// Name
SYRK(k, n)

// Execution space
k = 0..SIZE
n = 0..k-1

// Parallel partitioning
k % GRIDrows == rowRANK
k % GRIDcols == colRANK

// Parameters
IN  A <- C TRSM(n, k)
INPUT T <- (n == 0) ? IN(k, k) : T SYRK(k, n-1)
      -> (n == k-1) ? T POTRF(k) : T SYRK(k, n+1)

// Body
cblas_ssyrk(CblasColMajor,
           CblasLower, CblasNoTrans,
           NB, NB, 1.0, A, NB, 1.0, T, NB);
```

```
// Name
TRSM(k, m)

// Execution space
k = 0..SIZE
m = k+1..SIZE

// Parallel partitioning
m % GRIDrows == rowRANK
k % GRIDcols == colRANK

// Parameters
IN  T <- T POTRF(k)
INPUT C <- (k == 0) ? IN(n, k) : C GEMM(k, m, k-1)
      -> A GEMM(m, m+1..SIZE, k)
      -> B GEMM(k+1..m-1, m, k)
      -> A SYRK(m, k)
      -> OUT(k, m)

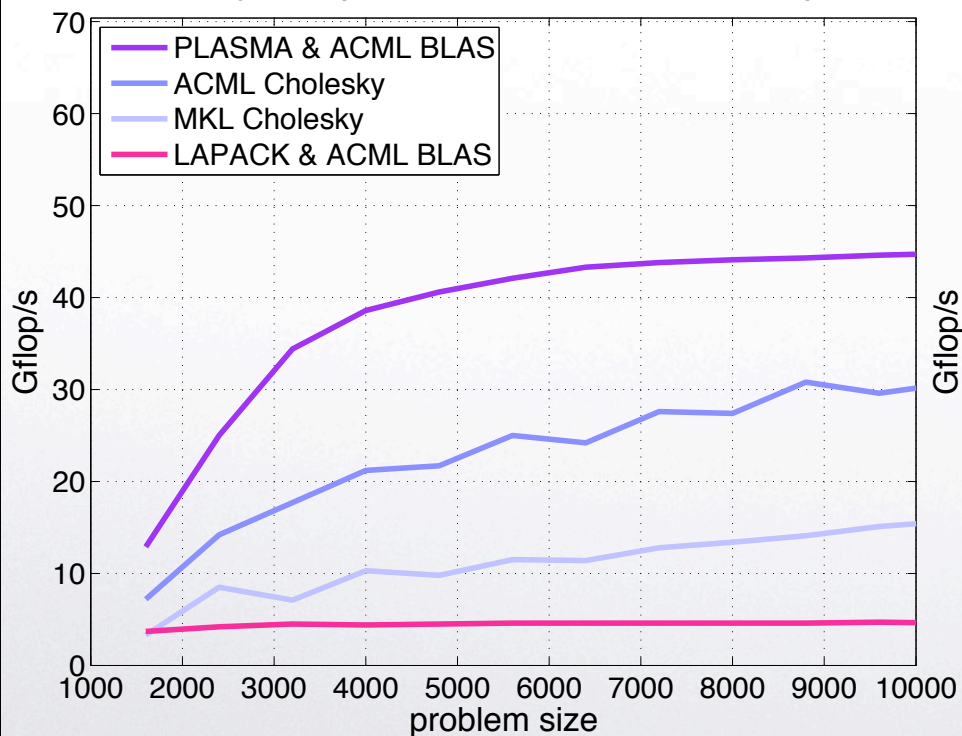
// Body
cblas_strsm(CblasColMajor,
           CblasRight, CblasLower,
           CblasTrans, CblasNonUnit,
           NB, NB, 1.0, T, NB, B, NB);
```

- text based description of the algorithm, input and output parameters
- language agnostic

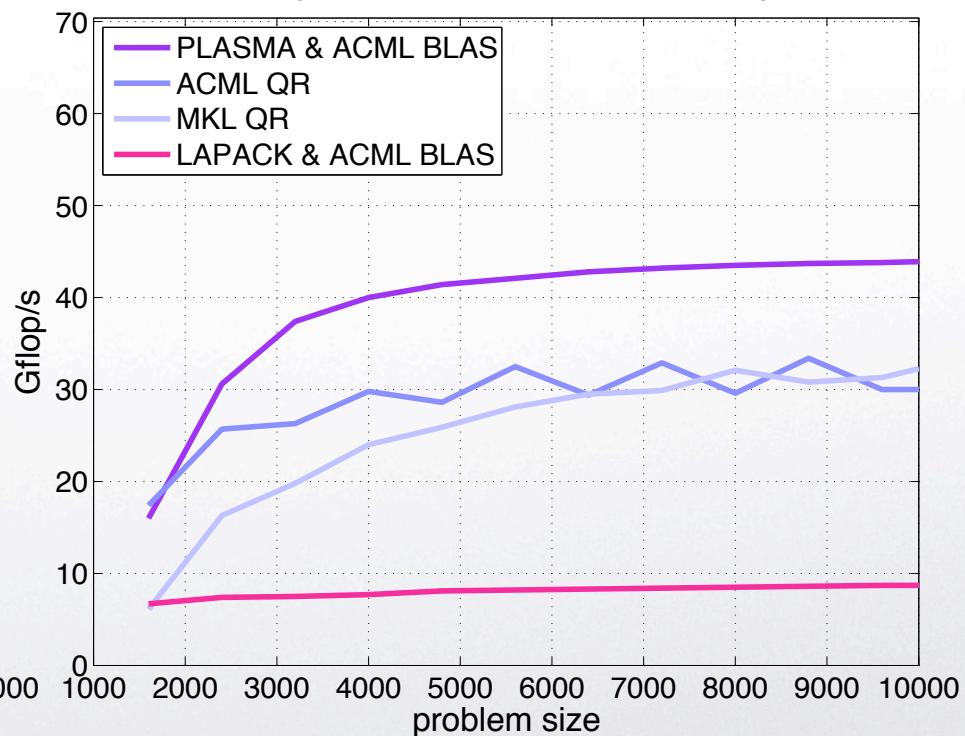


Early results

Cholesky -- quad-socket, dual-core Opteron

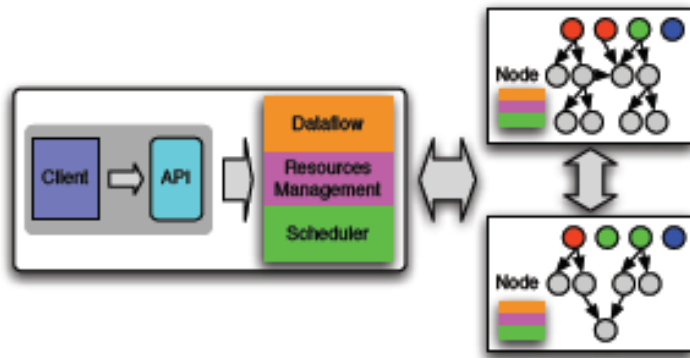


QR -- quad-socket, dual-core Opteron



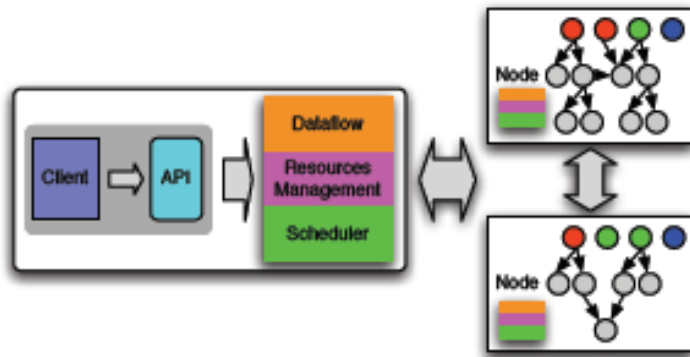


Master Worker





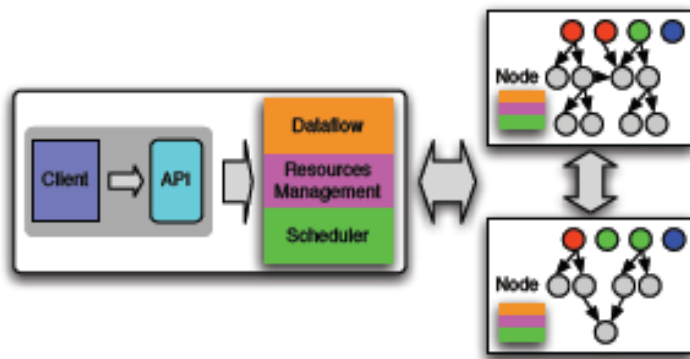
Master Worker



```
subroutine serv_conj_grad( param, colidx, rowstr, a, p, w)
...
do j = param(firstrow_index), param(lastrow_index)
  w(j) = 0.d0
  do k = rowstr(j), (rowstr(j+1) - 1)
    w(j) = w(j) + a(k)*p(colidx(k))
  enddo
enddo
```



Master Worker



```

subroutine serv_conj_grad( param, colidx, rowstr, a, p, q )
...
do j = param(firstrow_index), param(lastrow_index)
  w(j) = 0.d0
  do k = rowstr(j) , (rowstr(j+1) - 1)
    w(j) = w(j) + a(k)*p(colidx(k))
  enddo
enddo
enddo

```

```

CC dague( CREATE, colidx, rowstr, a, params, ... )
CC dague( BROADCAST, sizedata(colidx_id),colidx, (XX-1,NB_SERV;COLIDX_XX) )
CC dague( BROADCAST, sizedata(rowstr_id),rowstr, (XX-1,NB_SERV;ROWSTR_XX) )
CC dague( BROADCAST, sizedata(a_id), a_id, a, (XX-1,NB_SERV;A_XX) )
.....
do 40 it = 1, niter
  call conj_grad ( colidx, ..., rnorm )
  do 41 j=1, lastcol - firstcol+1
    norm_templ(1) = norm_templ(1) + x(j)*z(j)
  .....
c-----
subroutine conj_grad ( ... )
...
do 111 cgit = 1, cgitmax
CC   dague( BROADCAST, size(p_id), p, (XX-1,NB_SERV;P_XX) )
CC   dague( FOR, XX, 1, NB_SERV )
CC   dague( serv_conj_grad, PARAMS_XX, COLID_XX, ROWSTR_XX, A_XX, P_XX )
CC   dague( END_FOR )
CC   dague( GATHER, size(q_it), q, (XX-1,NB_SERV;P_XX) )
do 141 j=1, lastcol - firstcol+1
  sum = sum + p(j)*q(j)
.....

```



DAGuE - the runtime system

- Resource constraints
- Automatic Resource Management
- Asynchronous Task Executions
- Implicit communications
- Collective Communications
- Dynamic multi-level scheduling
- Fault Tolerance



FT-MPI



Why ?

- A lack of fault tolerant programming paradigms
- MPI is the de-facto programming model for parallel applications
- MPI Standard: “*Advice to implementors*: A good quality implementation will, to the greatest possible extent, circumvent the impact of an error, so that normal processing can continue after an error handler was invoked.”



How ?

- Define the behavior of MPI [state] in case an error occurs
- Give the application the possibility to recover from a node-failure
- A regular, non fault-tolerant MPI program will run using FT-MPI
- Follows the MPI-1 and MPI-2 specification as closely as possible (e.g. no additional function calls)
- On error user program must do something (!)

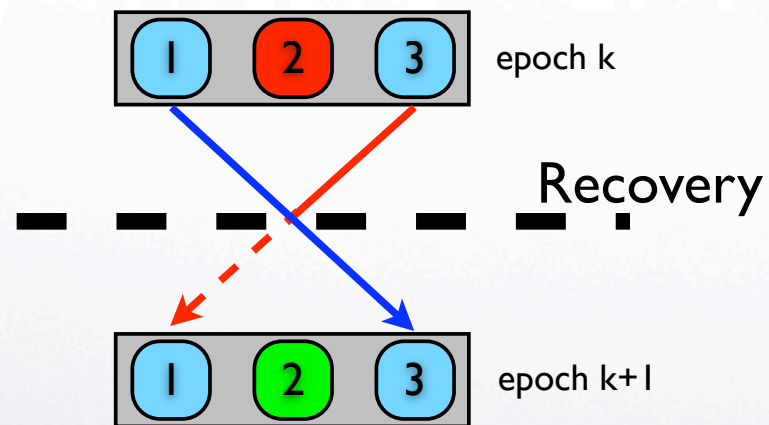


Recovery modes

- ABORT, **BLANK**, SHRINK and **REBUILD**
- **REBUILD**: a new process is created, and it will return `MPI_INIT_RESTARTED_PROC` from `MPI_Init`
- **BLANK**: dead processes replaced by `MPI_PROC_NULL`, all communications with such a process succeed, they do not participate in the collectives
- two sub-modes: local and global



Communications modes



- **RESET**: the epoch should match in addition to the MPI matching requirements
- **CONTINUE**: only MPI matching



FAQ

- Q: How do we know a process has failed ?
- A: The return code of an MPI operation returns **MPI_ERR_OTHER**. The failed process is not necessarily the process involved in the current operation!

- Q: What do I have to do if a process has failed ?
- A: You have to start a recovery operation before continuing the execution (MPI_Comm_dup on MPI_COMM_WORLD). All non-local MPI objects (e.g. communicators) have to be re-instantiated



FAQ

- Q: How many processes failed ?

- A: `MPI_Comm_get_attr (comm, FTMPI_NUM_FAILED_PROCS, &valp, &flag);`
`numfailedprocs = (int) *valp;`

QUIZ!

- Q: Who has failed ?

- A: `MPI_Comm_get_attr (comm, FTMPI_ERROR_FAILURE, &valp, &flag);`
`errorcode = (int) *valp;`

`MPI_Error_get_string (errcode, errstring, &flag);`
`parsestring (errstring);`

QUIZ



FAQ

- **Q: How many processes failed ?**

- **A:** `int alive = 1 (if MPI_Init returned MPI_SUCCESS, otherwise 0), howmany = 0;`
`MPI_Allreduce(&alive, &howmany, 1, MPI_INT, MPI_SUM,`
`MPI_COMM_WORLD);`

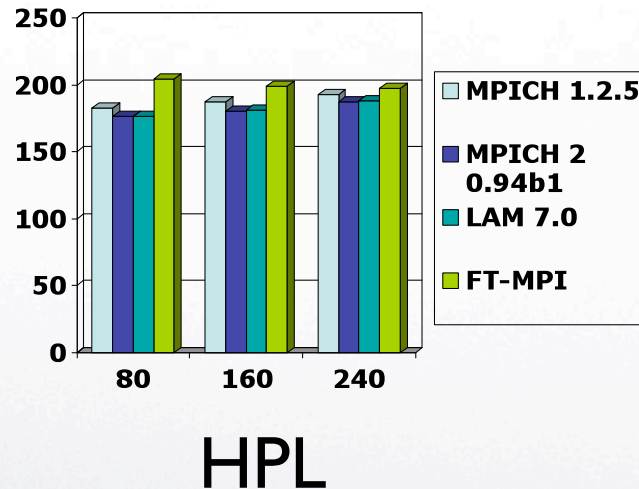
- **Q: Who has failed ?**

- **A:** `int whoswho[size] = 0;`
`whoswho[rank] = 1 (if MPI_Init returned MPI_SUCCESS, otherwise 0);`
`MPI_Allreduce(MPI_IN_PLACE, whoswho, size, MPI_INT, MPI_MAX,`
`MPI_COMM_WORLD);`

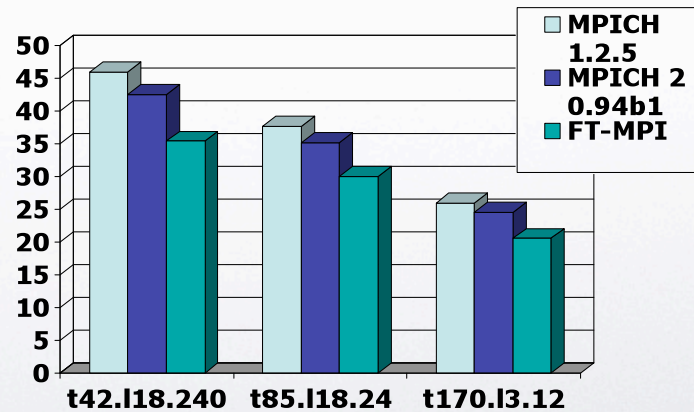


Shallow Water (PSTSWM) & HPL

32 nodes with Gigabit



PSTSWM





Diskless Checkpointing

P1 P2 P3 P4

4 available processors



Diskless Checkpointing

P1 P2 P3 P4

4 available processors

P1 + P2 + P3 + P4 = P5

Add a fifth and perform
a checkpoint(Allreduce)



Diskless Checkpointing

P1

P2

P3

P4

4 available processors

P1

+

P2

+

P3

+

P4

=

P5

Add a fifth and perform
a checkpoint(Allreduce)

P1

P2

P3

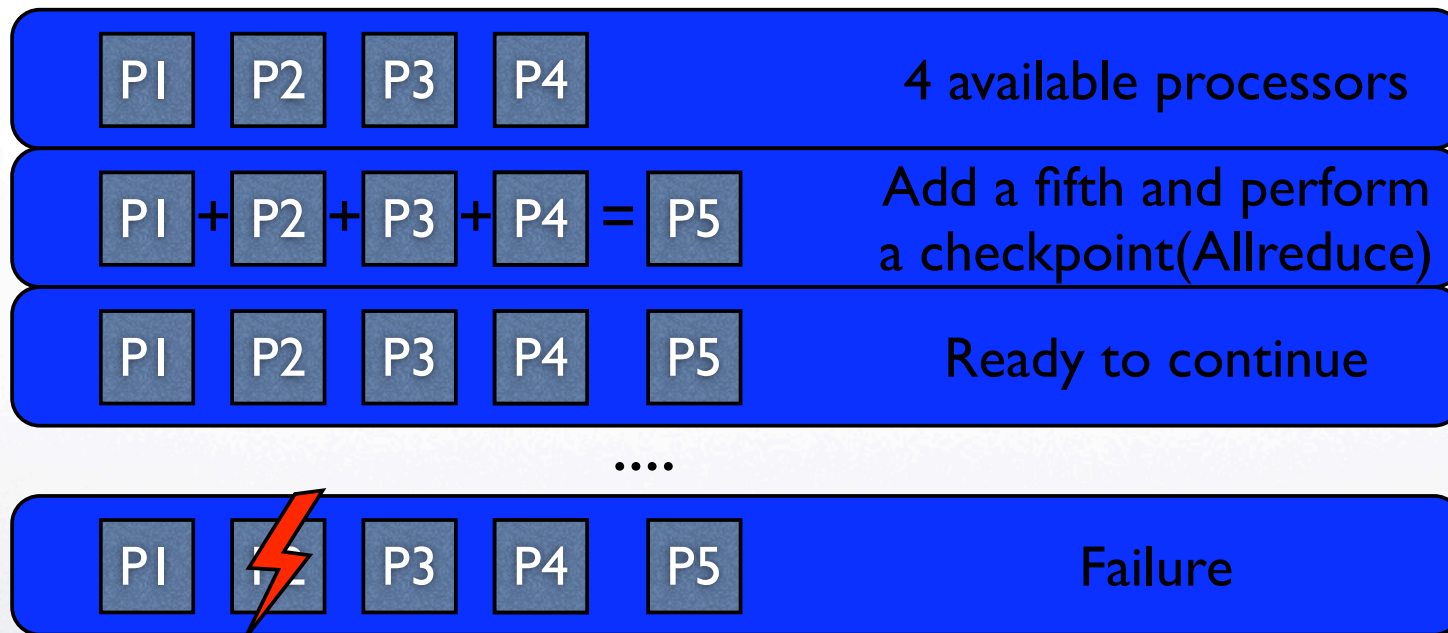
P4

P5

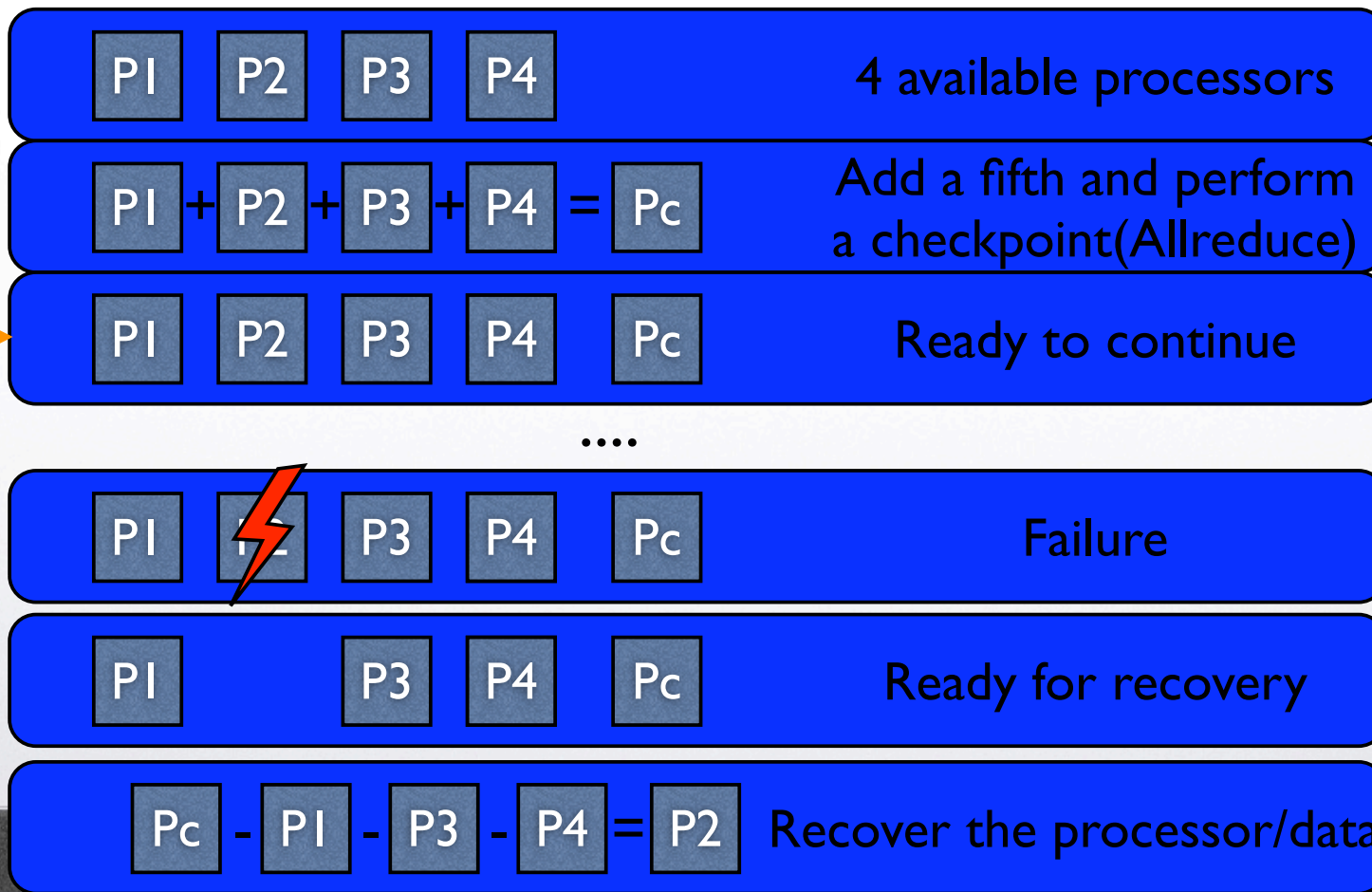
Ready to continue



Diskless Checkpointing



Diskless Checkpointing





Diskless Checkpointing

- How to checkpoint ?
 - either floating-point arithmetic or binary arithmetic will work
 - 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
- How to support multiple failures ?
 - Reed-Salomon algorithm
 - support p failures require p additional processors (resources)

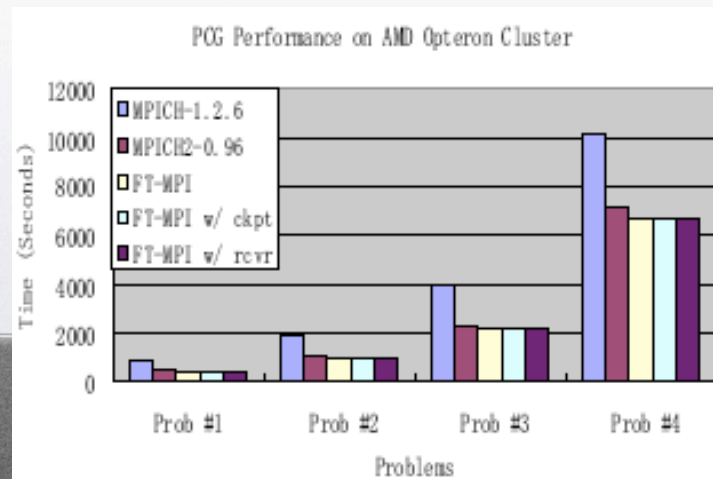


PCG

- Fault Tolerant CG
- 64x2 AMD 64 connected using GigE

	Size of the Problem	Num. of Comp. Procs
Prob #1	164,610	15
Prob #2	329,220	30
Prob #3	658,440	60
Prob #4	1,316,880	120

Performance of PCG with different MPI libraries



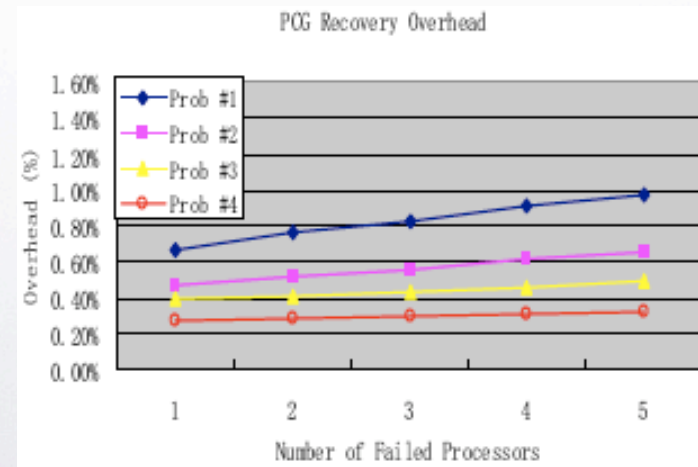
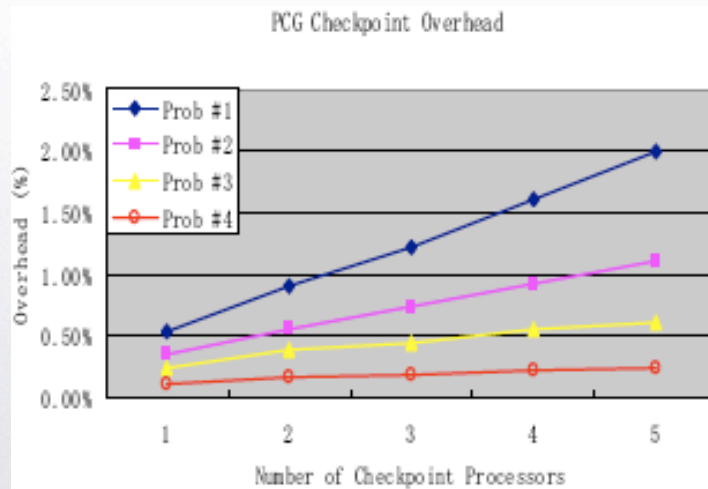
For ckpt we
generate one ckpt
every 2000
iterations



PCG

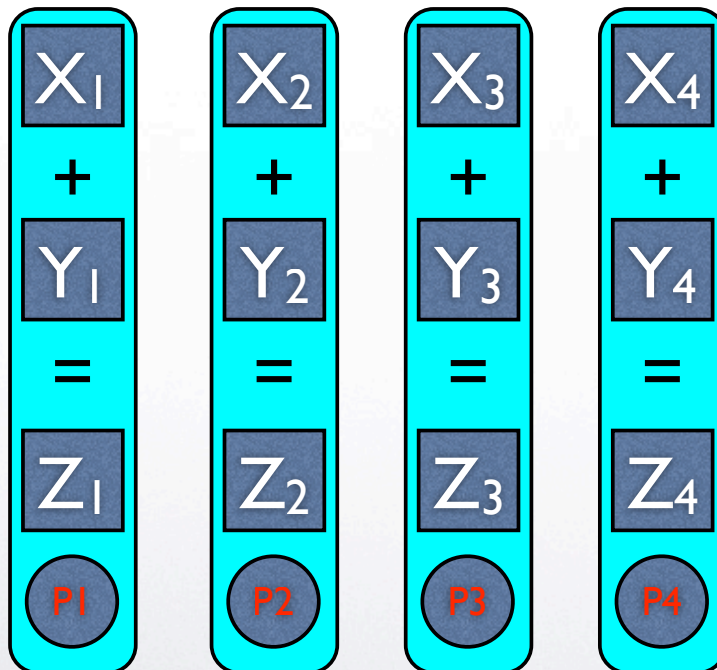
Time	Prob #1	Prob #2	Prob #3	Prob #4
1 ckpt	2.6	3.8	5.5	7.8
2 ckpt	4.4	5.8	8.5	10.6
3 ckpt	6.0	7.9	10.2	12.8
4 ckpt	7.9	9.9	12.6	15.0
5 ckpt	9.8	11.9	14.1	16.8

Checkpoint
overhead in
seconds





AFTB concept in example

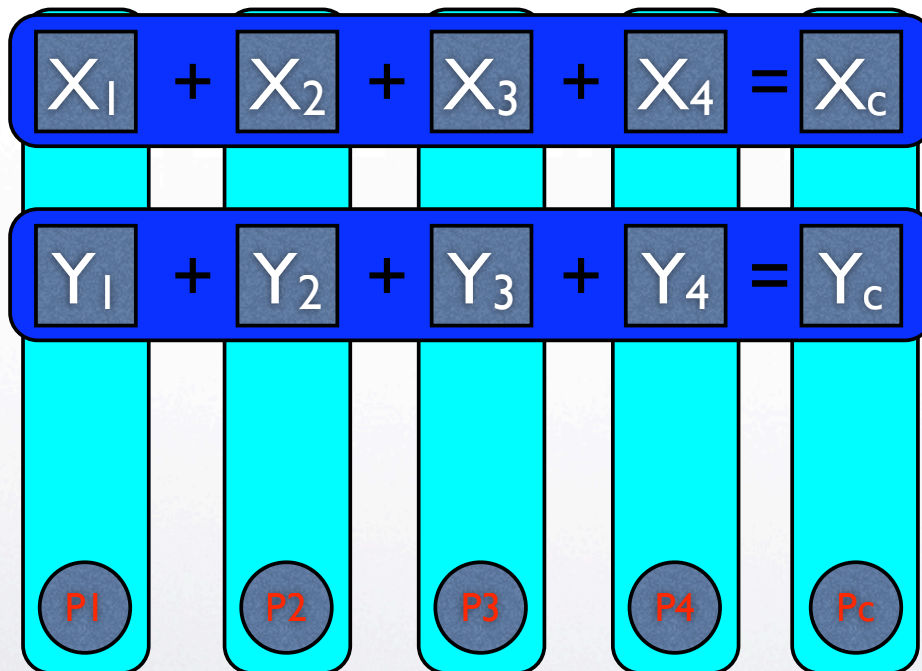


Perform in parallel
 $z = x + y$

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.



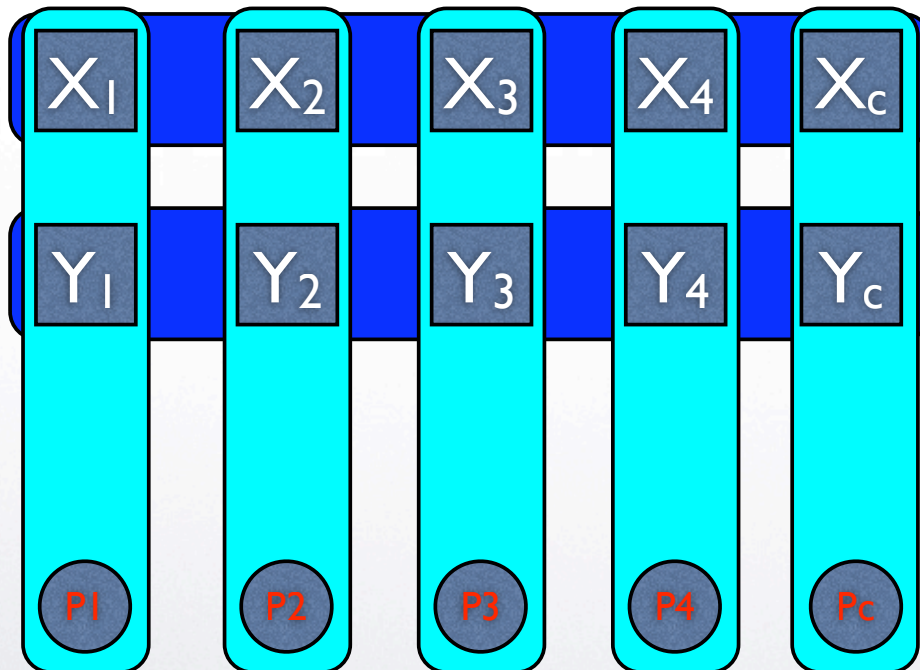
AFTB concept in example



Compute in parallel
the checksum of
x and y

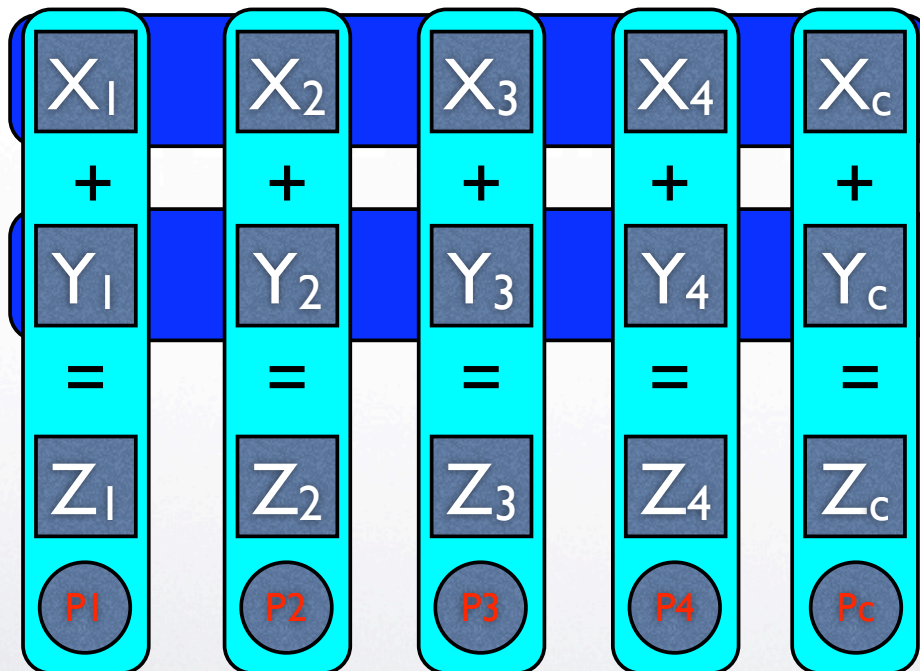


AFTB concept in example



We're ready to proceed with the sum

AFTB concept in example



Compute in parallel
the sum of
 x and y

Simultaneously we
can compute the sum
of the checkpoint

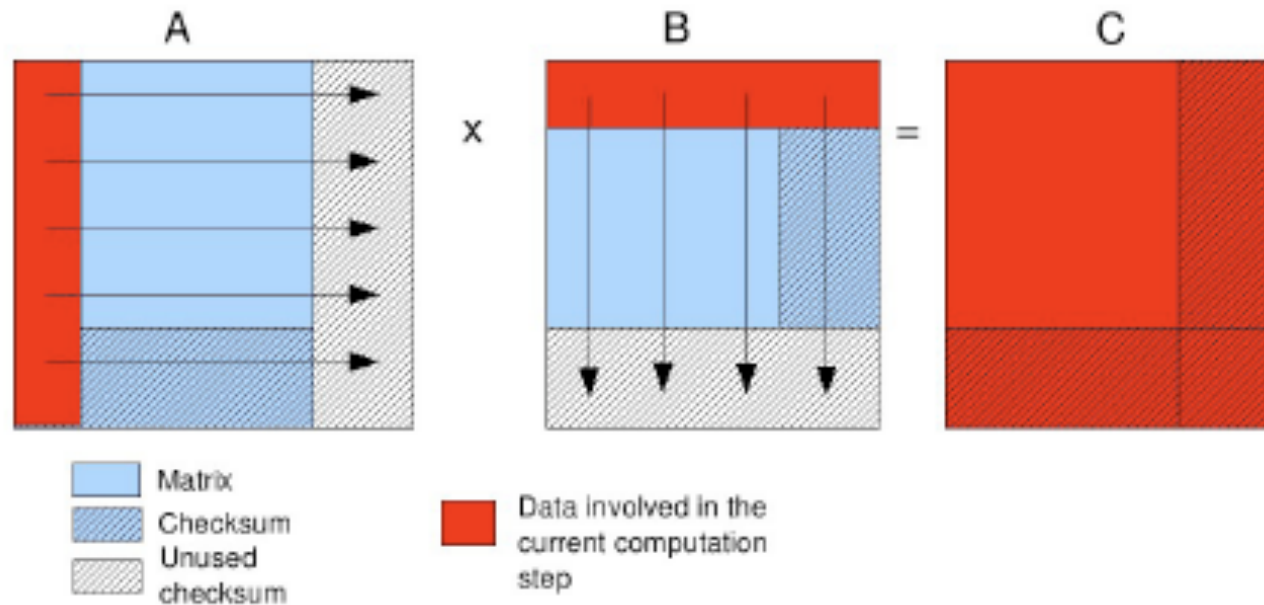


ABFT summary

- Relies on floating-point arithmetic
- Exploit the checksum processor
- Stable algorithms exist for any linear operation:
 - AXPY, SCAL (BLAS1)
 - GEMV (BLAS2)
 - GEMM (BLAS3)
 - LU, QR, Cholesky (LAPACK)
 - FFT



ABFT-PDGEMM

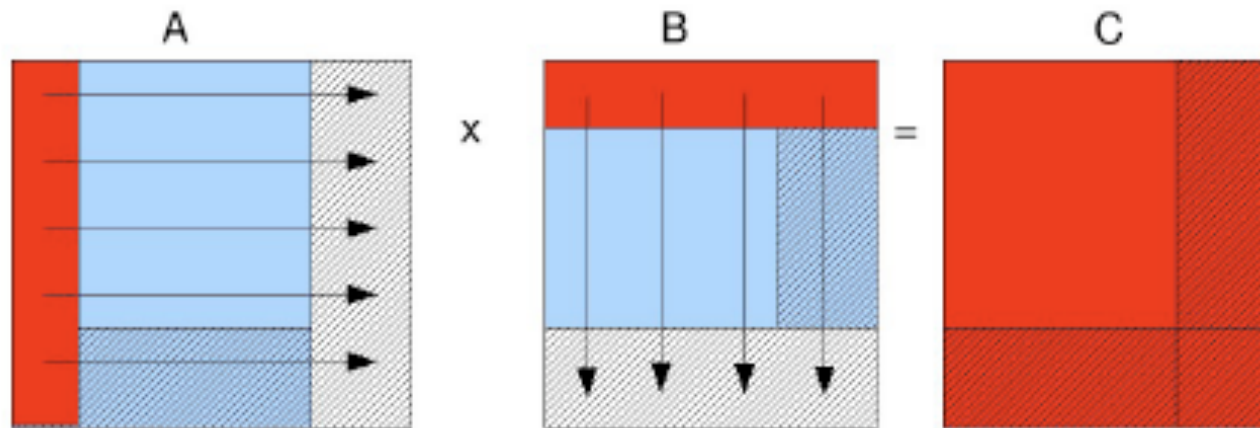


$$A_F = \begin{pmatrix} A & AC_R \\ C_C^T A & C_C^T AC_R \end{pmatrix} \quad \text{and} \quad B_F = \begin{pmatrix} B & BC_R \\ C_C^T B & C_C^T BC_R \end{pmatrix}$$

$$\begin{pmatrix} A \\ C_C^T A \end{pmatrix} \begin{pmatrix} B & BC_R \end{pmatrix} = \begin{pmatrix} AB & ABC_R \\ C_C^T AB & C_C^T ABC_R \end{pmatrix} = (AB)_F$$



ABFT-PDGEMM

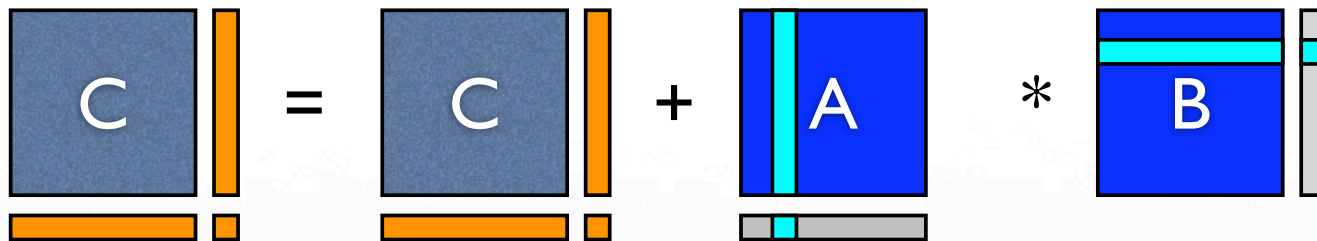


The overhead:

- $2p-1$ extra processes for p^2
- one extra process need to receive the data for the rows and columns

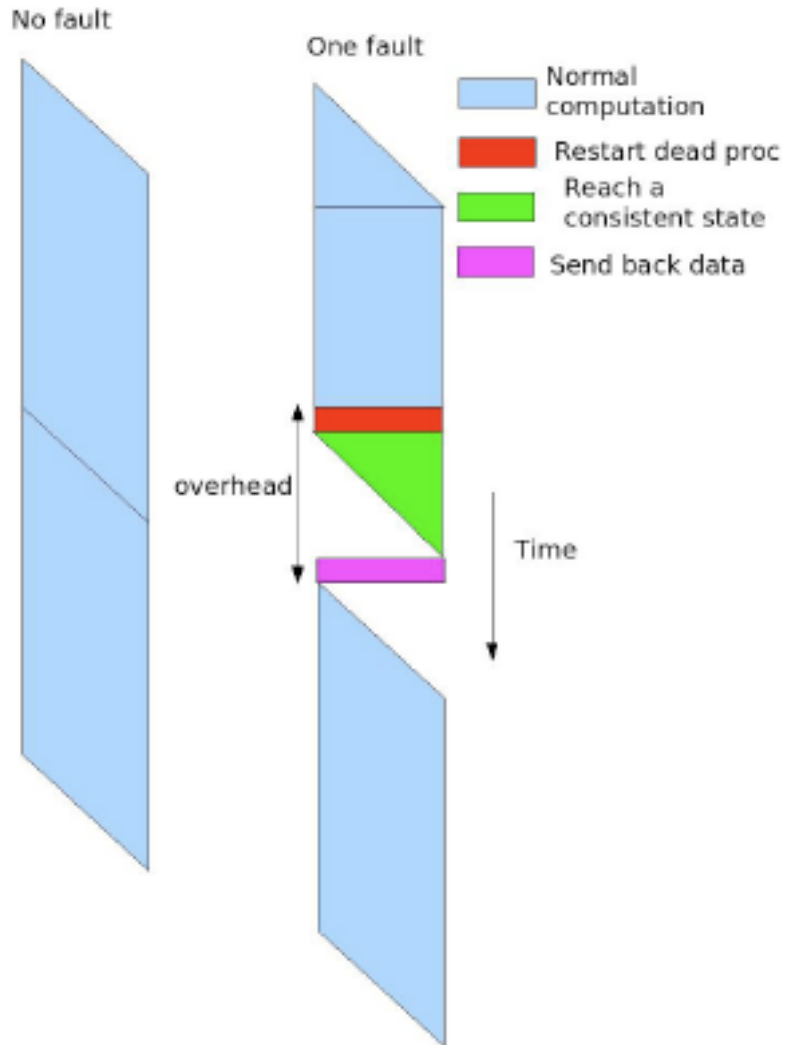
Conclusion: a **very scalable approach**, more processors means less overhead

ABFT-PDGEMM



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

- The algorithm maintain the consistency of the checkpoints of the matrix C naturally



Failure Overhead

- FT-MPI will take care of the fault management
- Once the new process joins the `MPI_COMM_WORLD` we have to rebuild the communicators
- Then we have to retrieve the data from the checkpoint processor

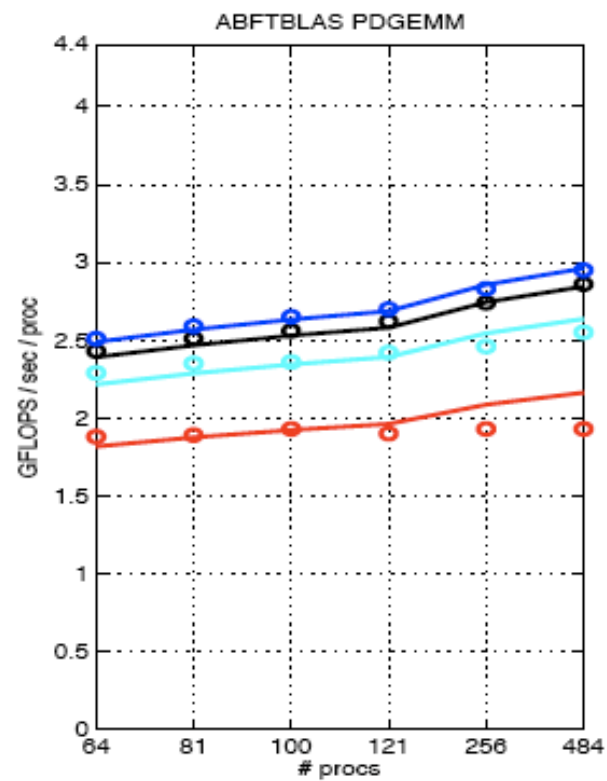
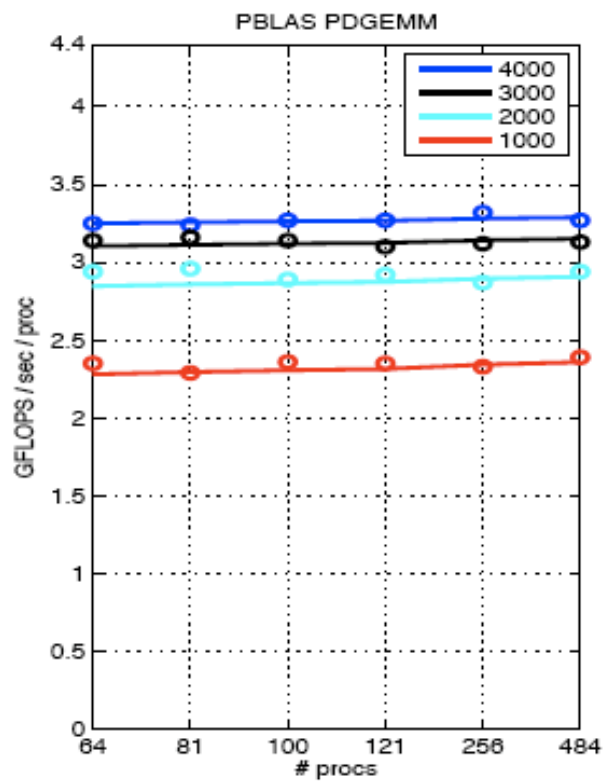


jacquard.nersc.gov

- **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 μ sec
- **Switch MPI Unidirectional Bandwidth (peak)** 620 MB/s
- **Global shared disk GPFS Usable disk space** 30 TBytes
- **Batch system** PBS Pro

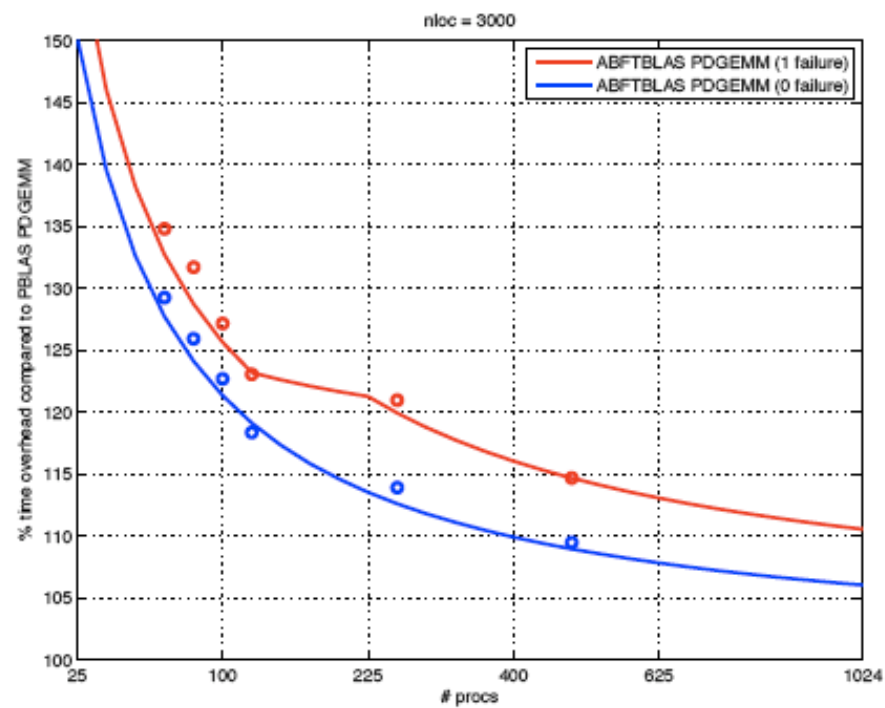
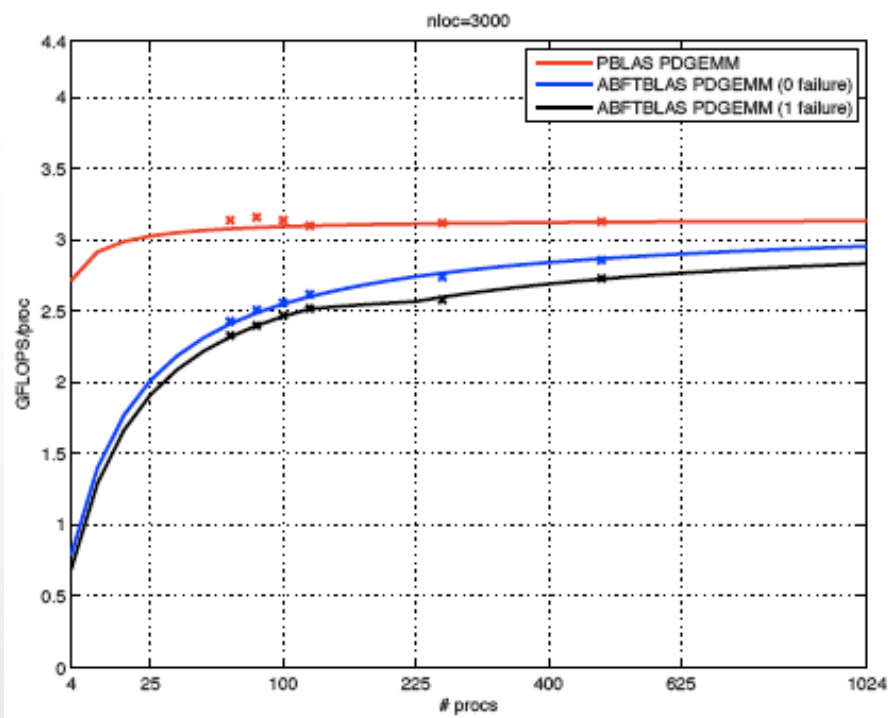


PBLAS vs. ABFT BLAS (0 failure)



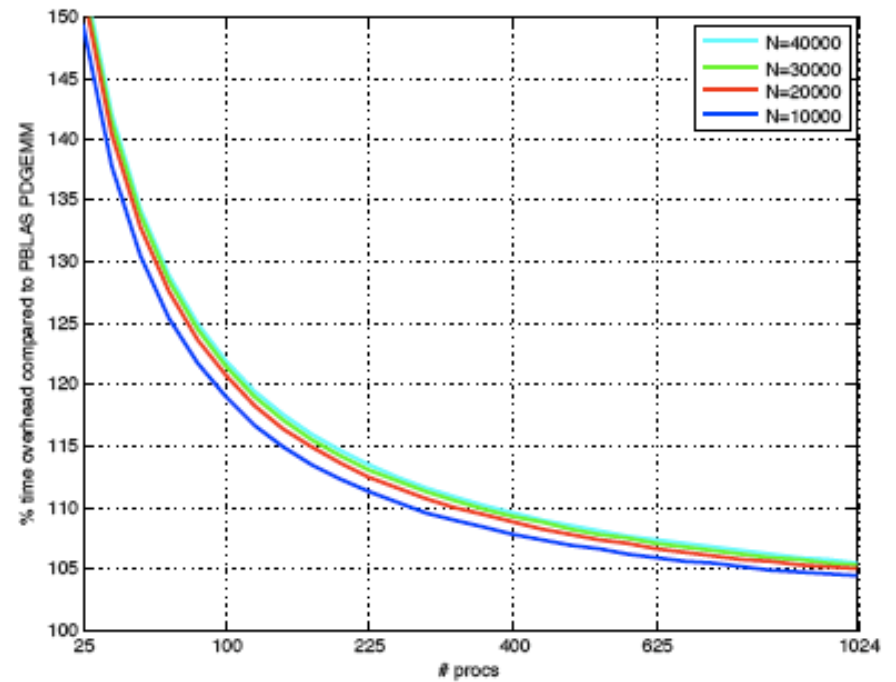
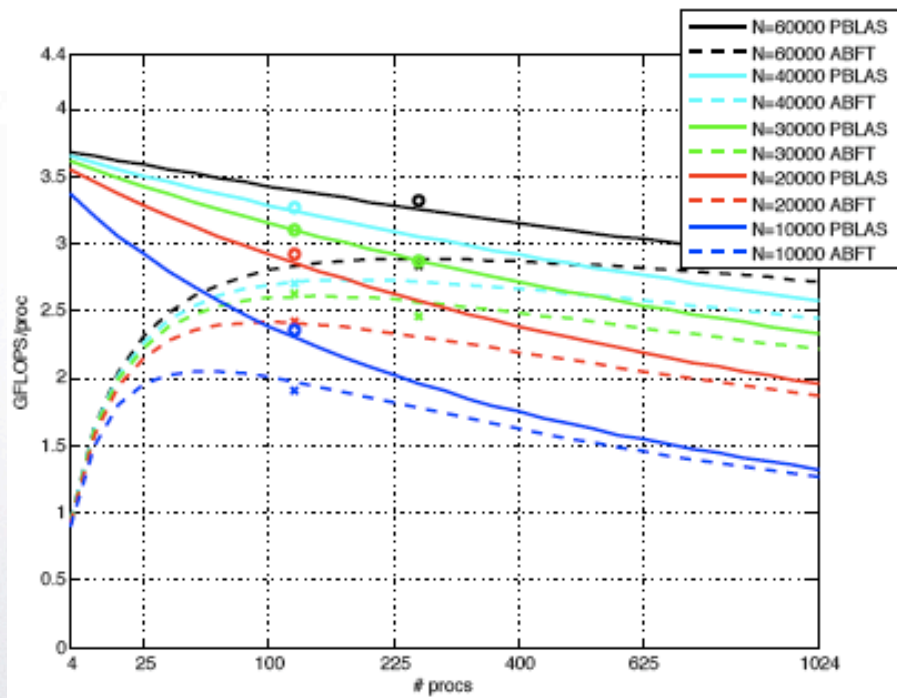


Weak scalability





Strong Scalability





Conclusion

- Data-flow programming models an interesting alternative
- Fault tolerance is a requirement
 - FT-MPI approach a viable possibility with algorithms already available
- The future of MPI is decided now !