High-Performance Linear Algebra

Adrian Tate (CSAR)

Patrick Briddon (University of Newcastle)

Content

AIMPRO

AIMPRO - Code for first principles electronic structure

Standard problem in computational physics:

- matrix eigenvalue problem
- A x = _ x

Arises when solving a PDE eigenvalue problem via basis set expansion.

Occurs in applications in many areas of science or engineering

Eigenvalue Problem

Standard approach when matrix dense and a substantial proportion (e.g. 10%) of eigenvectors needed

- Reduce matrix to tridiagonal form. The eigenvalues/eigenvectors of this are then more easily found. This is done in (e.g.) LAPACK. We will look at efficiency of tri-diagonalisation using householder method.

Eigenvalue Problem

Old implementations (e.g. eispack) use level one BLAS (ddot/daxpy).

Newer implementations (LAPACK) try to use level 3 as much as possible

Structure of lapack code is:

LAPACK code

```
do  i=n-1,1,-1
   ... setup elementary reflector x(1:i), O(i)
   work
   y(1:i) = dsymv ( a(1:i,1:i), x(1:i) ) // symmetric
   matrix vector multiply
   ... O(i) work generating vectors u(:) and v(:) ..
   call dsyr2( u(1:i), v(1:i), a(1:i, 1:i) ) // a_ij =
   a_ij +u_i*v_j

enddo
```

LAPACK

Both dsymv and dsyr2 do equal work, $(2N^3)/3$ giving $(4N^3)/3$ total op count

But ... both dsymv and dsyr2 mem bandwidth limited. LAPACK chooses strategy of blocking. Result is that dsyr2 replaced with level 3 routine dsyr2k. However, the dsymv's remain.

Resulting performance = $2N^3/3$ FLOP at level 2 BLAS speed + $2N^3/3$ FLOP at level 3 BLAS speed.

Parallel

How well does this go in parallel - Scalapack routine is pdsytrd. This uses exactly same strategy - build on parallel versions of dsymv and dsyr2k (these are called pdsymv and pdsyr2k)

But two Problems:

- pdsymv performance poor (poor serial + poor parallel scaling)

Parallel

2. Work done in preparing matrix as input for pdsyr2k is not shared by all processors, leading to load imbalance. Some pdsytrd scaling is poor - notice big drop off on 1000 matrix even on 16 nodes.

Often we may want to repeatedly diagonalise matrices of 2000 or so on larger numbers of processors. ScaLAPACK is not very good at this.

Optimisation

Alternative approach followed here

- ➢ write code based on level 2 routine, but remove dsyr2 from end of previous iteration and fuse with dsymv.
- ➢ Exposes more flops per element of matrix a(i,j) loaded.

BLAS fusing

```
do  i=n-1,1,-1

    if(i==n-1)then
    y(1:i) = dsymv ( a(1:i,1:i), x(1:i) )
   else
    call dsyr2( u(1:i), v(1:i), a(1:i, 1:i) )
   y(1:i) = dsymv ( a(1:i,1:i), x(1:i) )
   endif
    do dsyr2 BUT ONLY FOR COLUMN "i" of a (remainder done
   in next iteration).

  enddo
  call dsyr2 (just for the last iteration)
```

Parallel Code

New thing : we need a handwritten blas routine that fuses dsymv and dsyr2. Not hard to get it to go well - just unrolling gets reasonable speed.

Result .. serial performance roughly same as previous LAPACK routine without need for level 3 routines ... therefore parallelism begins from same starting performance.

Parallel Code

The parallel version of our handwritten BLAS is even easier than serial - the ScaLAPACK block scattered distribution means that apart from a very small number of diagonal blocks we deal with rectangular matrices.

Parallel Code

This is already 50% better than ScaLAPACK and scales better

From this point we can profile the code to optimise for parallel

This exposes the main parallel inefficiency - the transposing of row distributed u/v/x/y into column equivalents or vice versa.

Exposing Parallel Inefficiency

1	0.000129	10	0.000359
2	0.004264	11	0.104754
3	0.019923	12	0.001907
4	0.407934	13	0.208461
5	0.001052	14	0.605466
6	0.001217	15	0.001203
7	0.708108	16	0.240084
8	0.002171	17	0.736583
9	0.008711	18	0.000085

Parallel Code

Earlier performance derived from improved cache utilisation, but further optimisation will involve outright beating of library routine PBDTRNV, can this be done?

This is a communication intensive routine

There are a number of ways forward:

Optimising PBDTRNV

Switch to 1-sided communications

Tune to requirements of program

Remove barriers

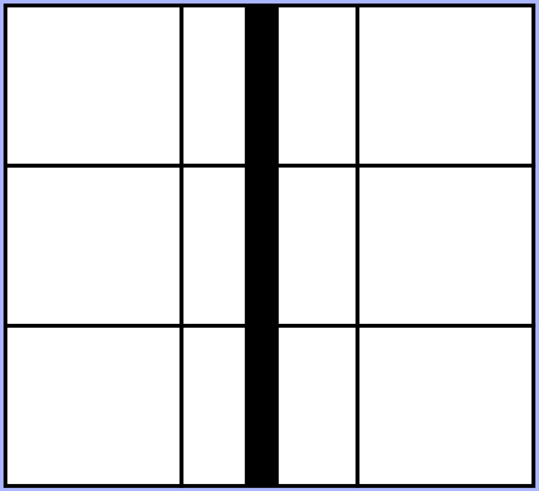
Exploit grid shape and requirements of program

(Always) look out for cache reuse possibilities

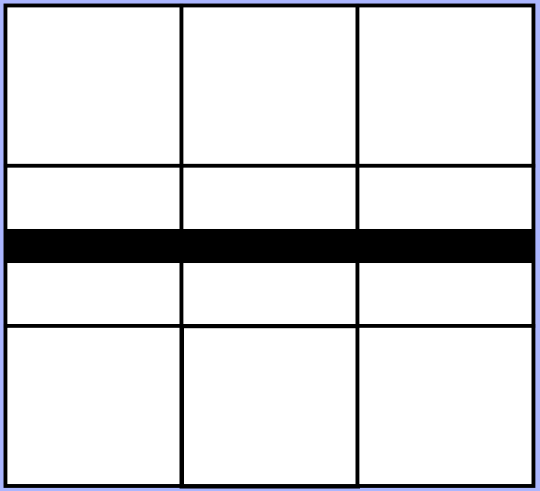
PBDTRNV

ScaLAPACK routine PBDTRNV takes a global block-cyclically distributed vector and creates its transpose, i.e.

PBDTRNV



PBDTRNV

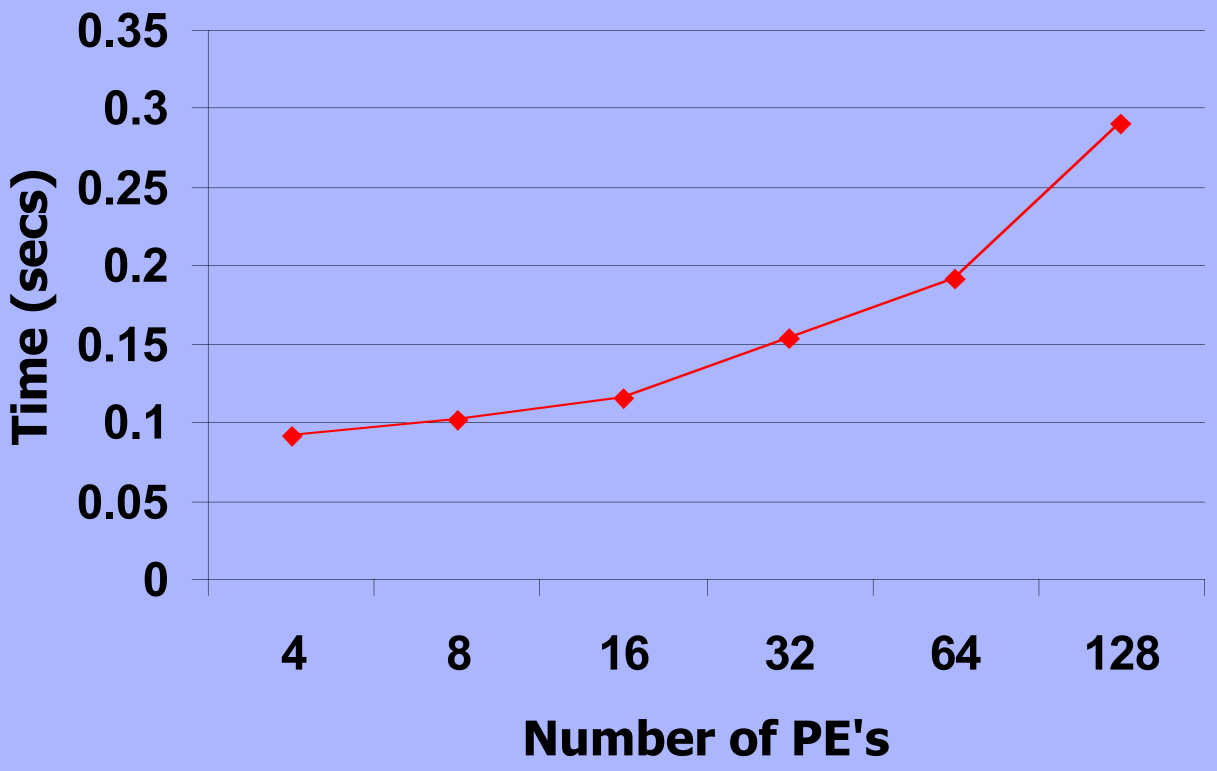


PBDTRNV

Seemingly simple operation

- Extremely complex program
- Affects Scaling badly

PBDTRNV performance



SHTRAN

Data transfer via shmem_ptr

```
DOUBLE PRECISION :: X_local,X_remote
POINTER(ptr,X_local)

Ptr = shmem_ptr(X_remote,target)
```

X_local then represents remote object X_remote

Performance better than shmem_get/put

Synchronization

To maximise the performance gains of 1-sided, synchronisation must be kept to a minimum

In particular, barriers are out of the question

Hence, synchronization must always be carefully orchestrated

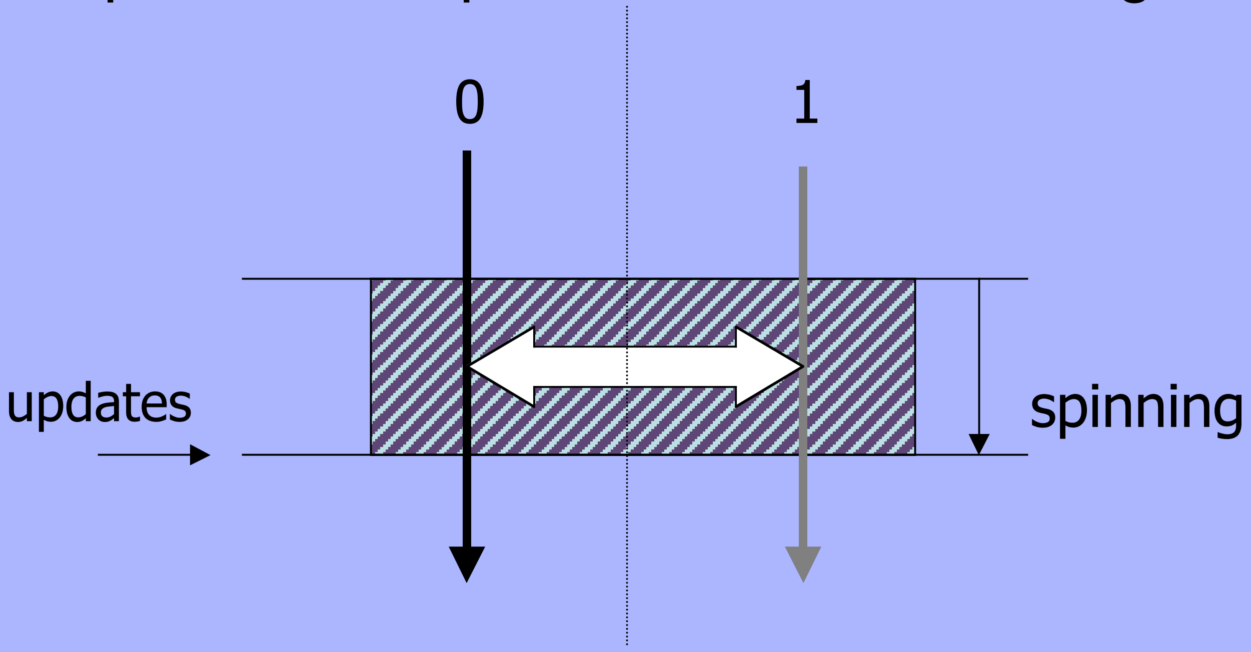
- Point of transfer
- ‘Trailing’ processors

Synchronization

Shmem_ptr is used to access a remote ‘safety’ integer

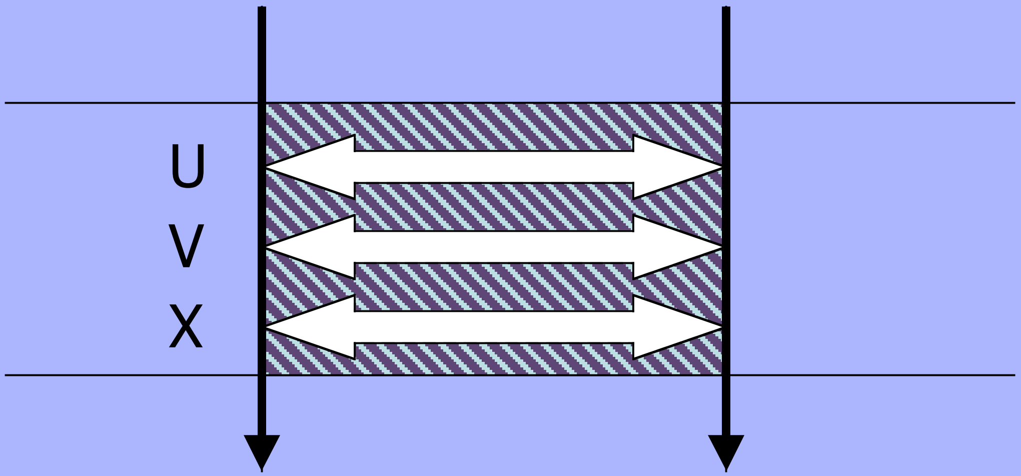
After data transfer the local host updates integer

Remote processes ‘spin’ on the value of integer



Synchronization sharing

There are three calls to PBDTRNV for vectors U,V and X. This results in a three-fold synchronization overhead. Instead, simply perform second and third transposes within the first routine, since the synchronization would be the same.



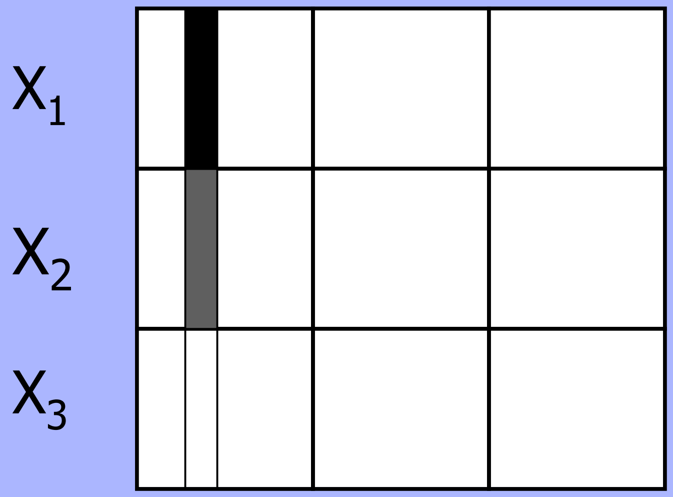
Exploit Grid shape

AIMPRO uses as close to a square BLACS grid as possible. This can be exploited.

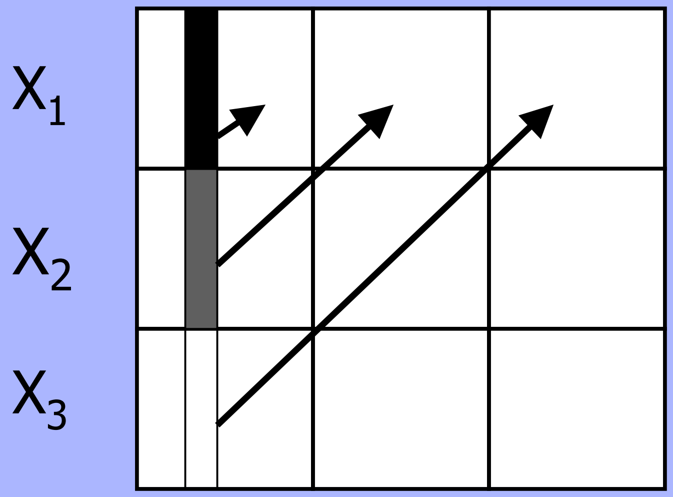
Block Cyclic distribution of vectors has a useful side-effect for square grids.

- If a global vector X is block –cyclically distributed onto a column of a 3x3 grid, then each process in column would have a section X_1, X_2 , $X_{3.}$ For square grids, redistribution of the vector after transposing results in identical distribution along the row, i.e. simulates a vector linear inverse

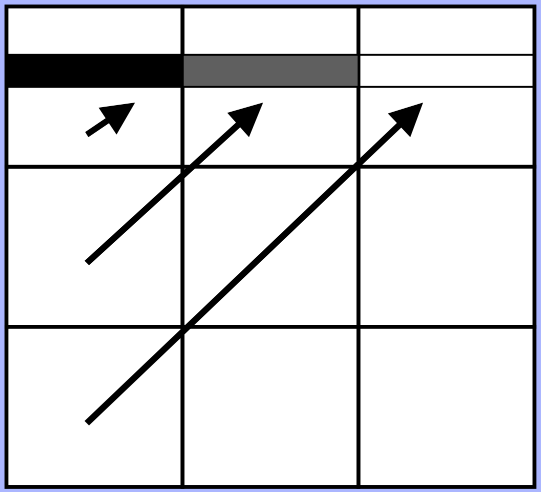
Square Grids



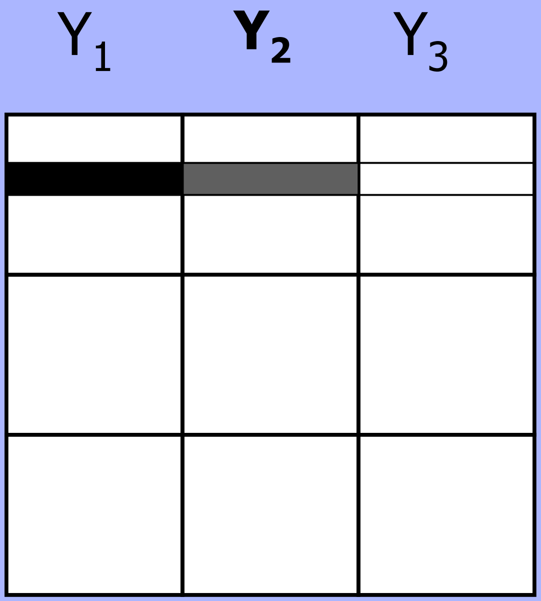
Square Grids



Square Grids



Square Grids



Exploit grid size.

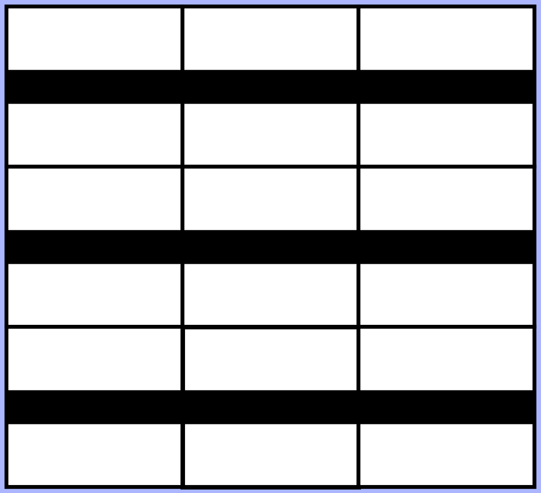
- For n x 2n grids can be optimised similarly. Here the local vector must be distributed only amongst 2 process elements
- In fact, whenever the grid has dimensions that are divisible, this can be exploited
- SHTRAN transfers the whole vector to target processor who then extracts relevant sections according to the blocking factor

Utilise the Functionality

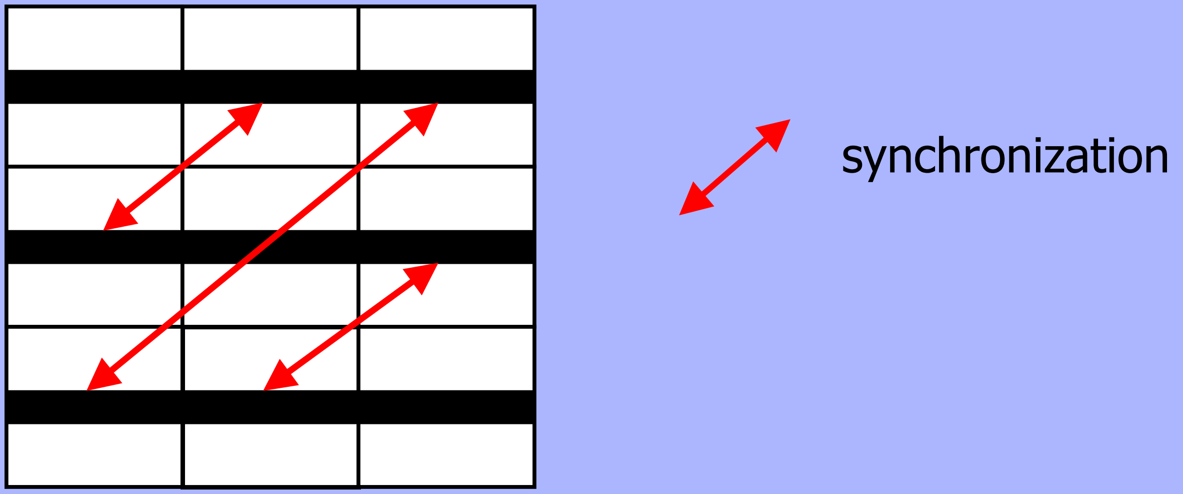
The transpositions in AIMPRO involve a row process transpose on every row simultaneously, or vise-versa

Resultant transposition is a series of np point-to-point data transfers, without the need for global communications or barriers

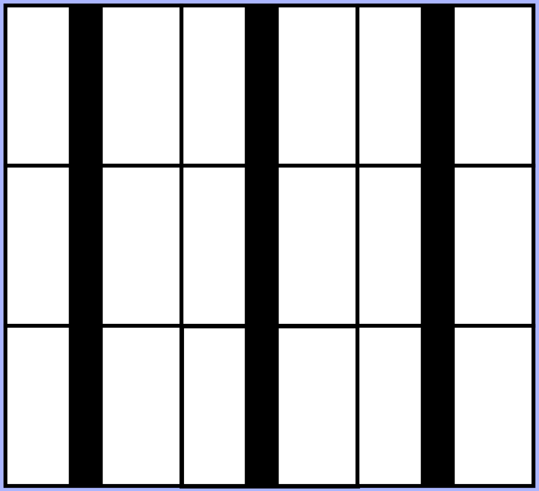
HPTRAN



HPTRAN



HPTRAN



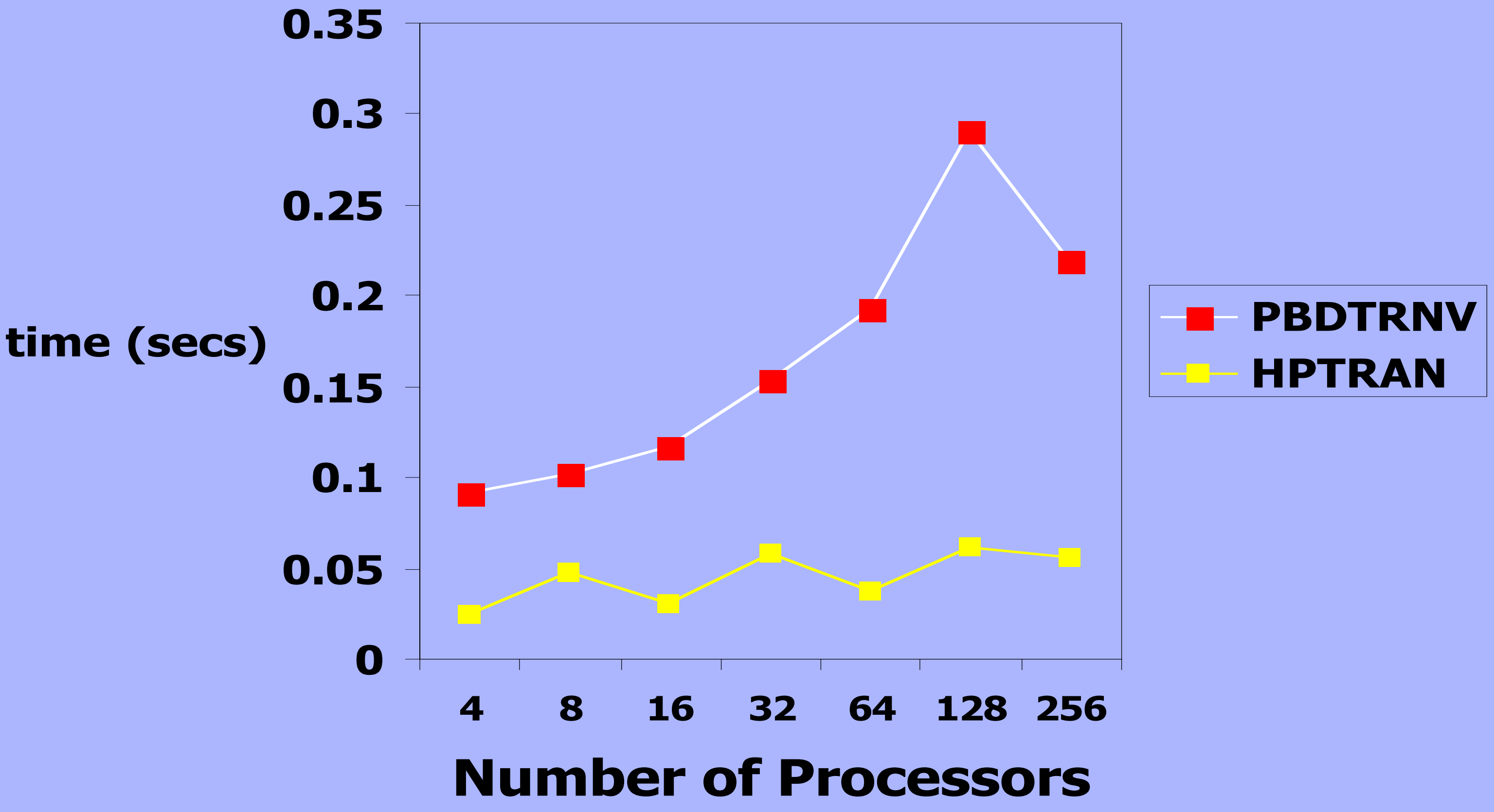
Performance

Performance is best for square grids

n x 2n and n x 3n grids still outperform PBDTRNV

Important feature is that execution time stays reasonably constant with number of processors

Performance



Outcome

By removing this major impediment the code scales much better, and performs well

But, similarly poor performance was obtained in the following areas of the code

- A column/ row summation of vectors
- A later solitary transposition
- A broadcast of a vector along rows

The same methods could be employed here. (?)

Problem

Code requires two summations

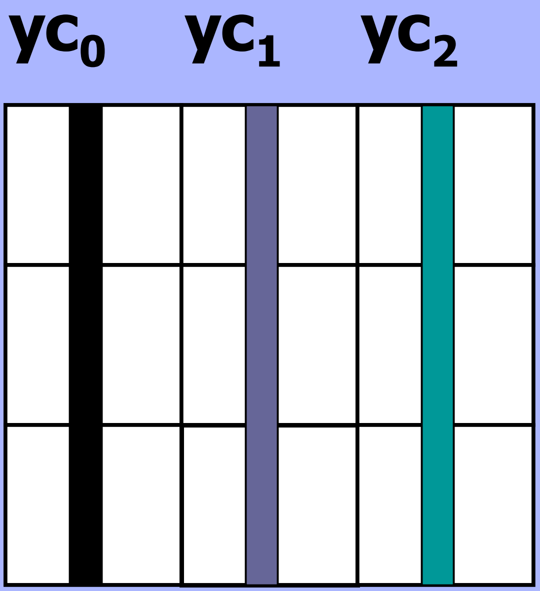
- 1 over a process row
- 1 over a process column

Followed by

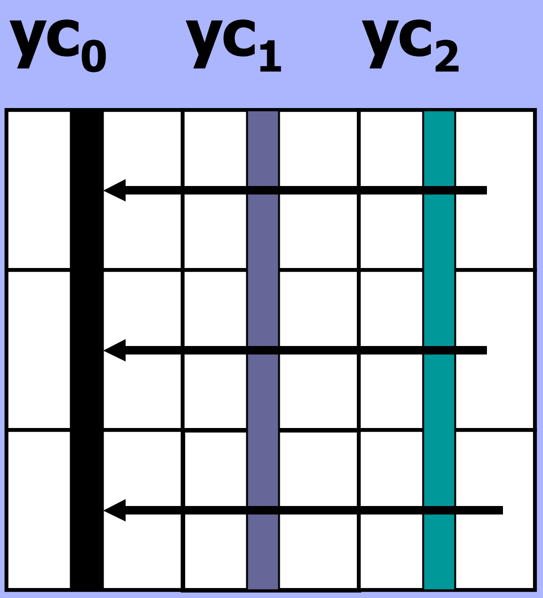
- Transposition of row vector
- Broadcast along process row

i.e…

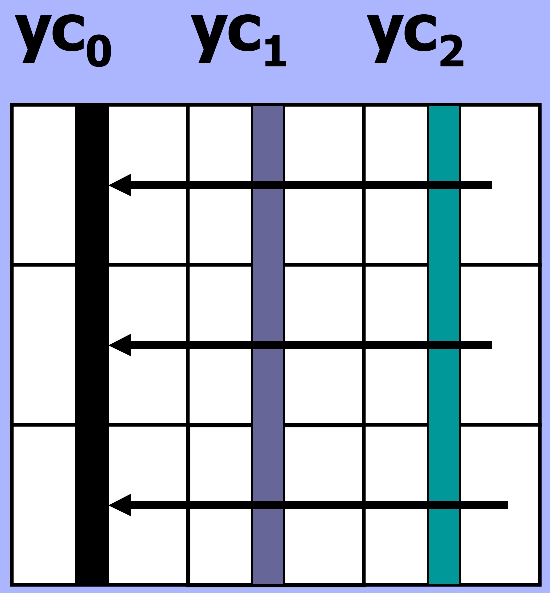
Vector yc summed over process row



Vector yc summed over process row

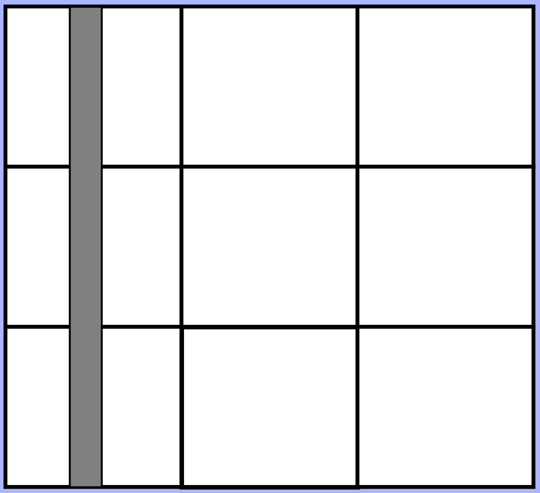


giving..

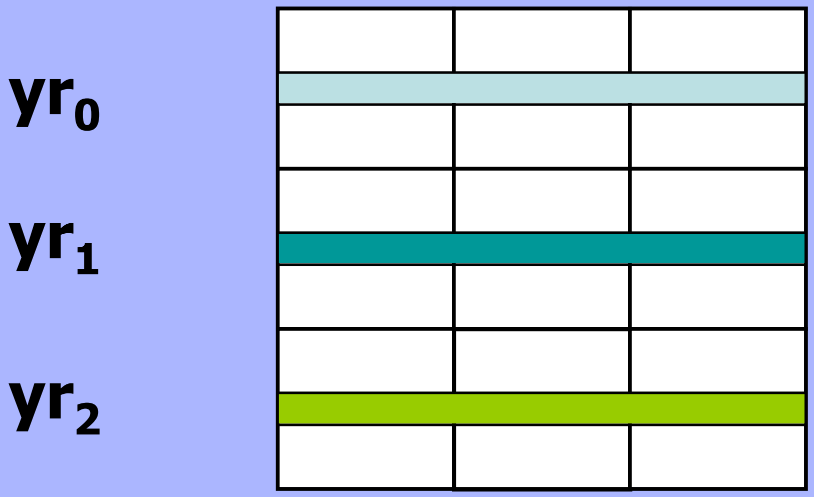


giving..

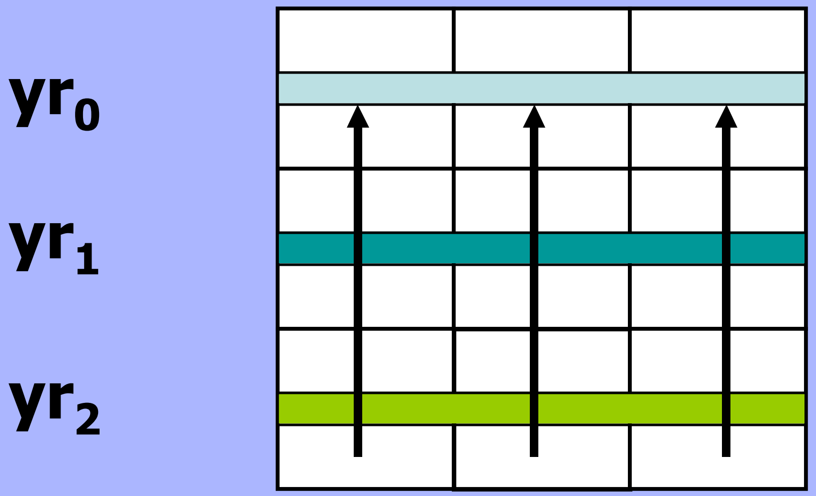
Yc



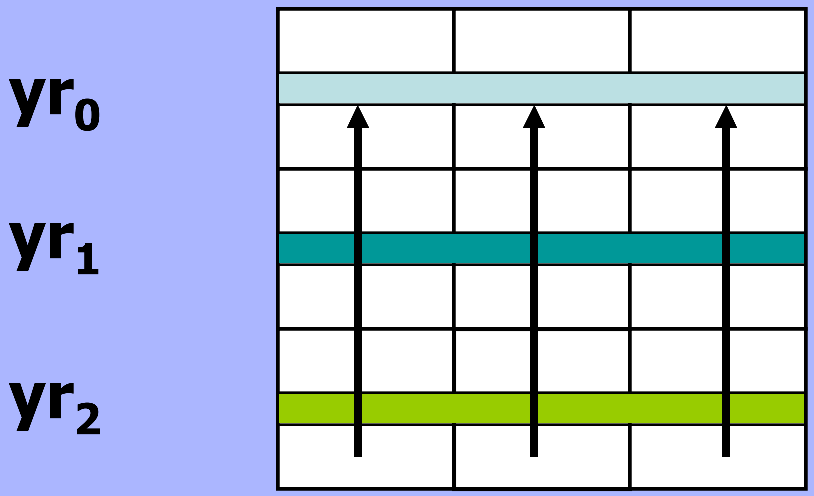
Similarly, yr is summed over process column



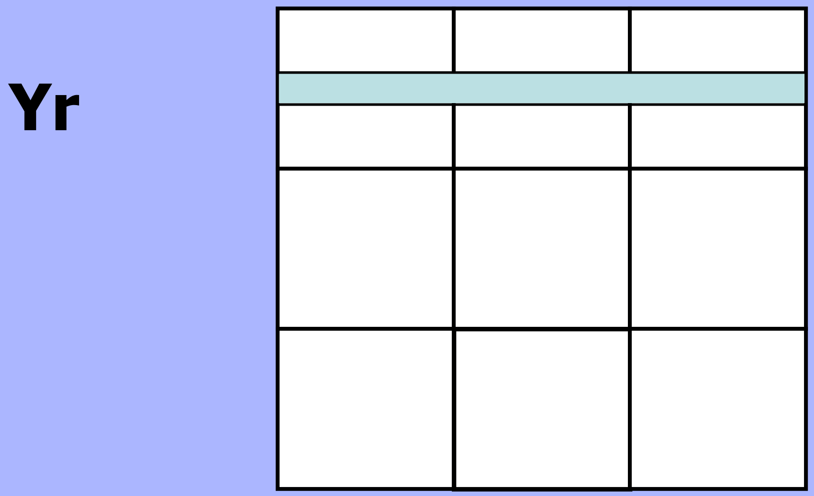
Similarly, yr is summed over process column



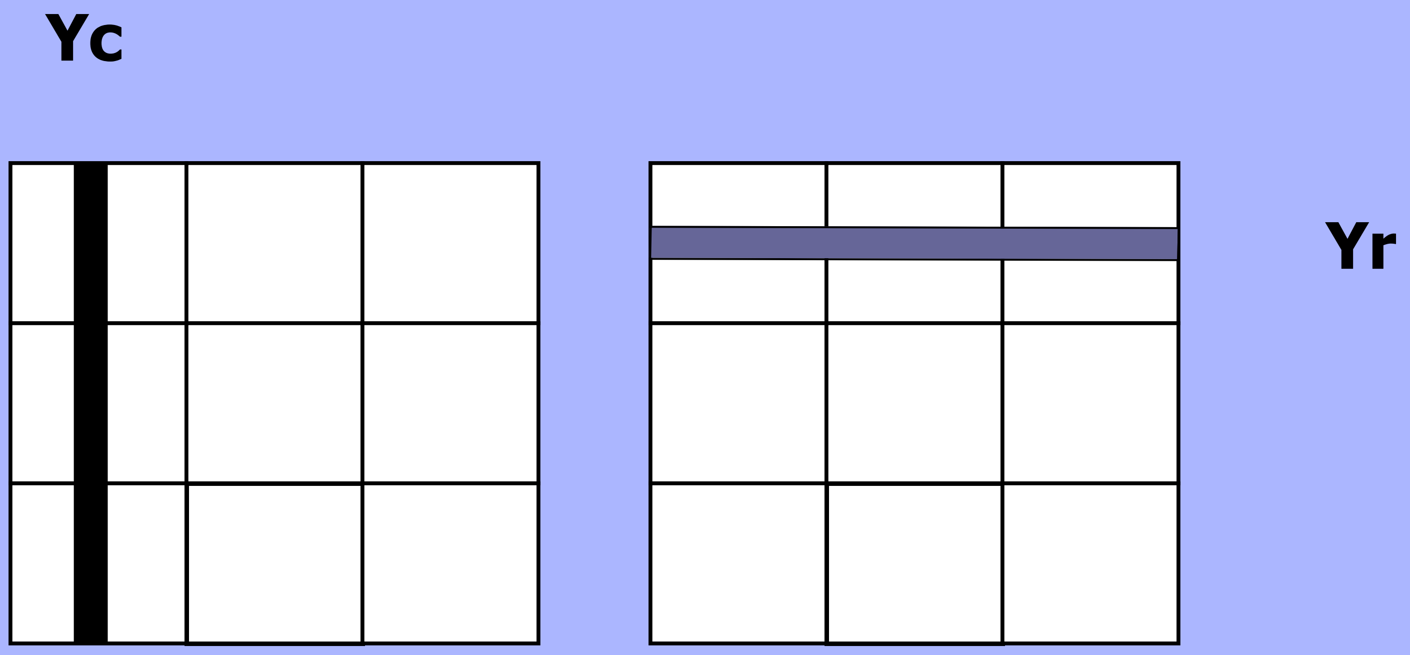
giving..



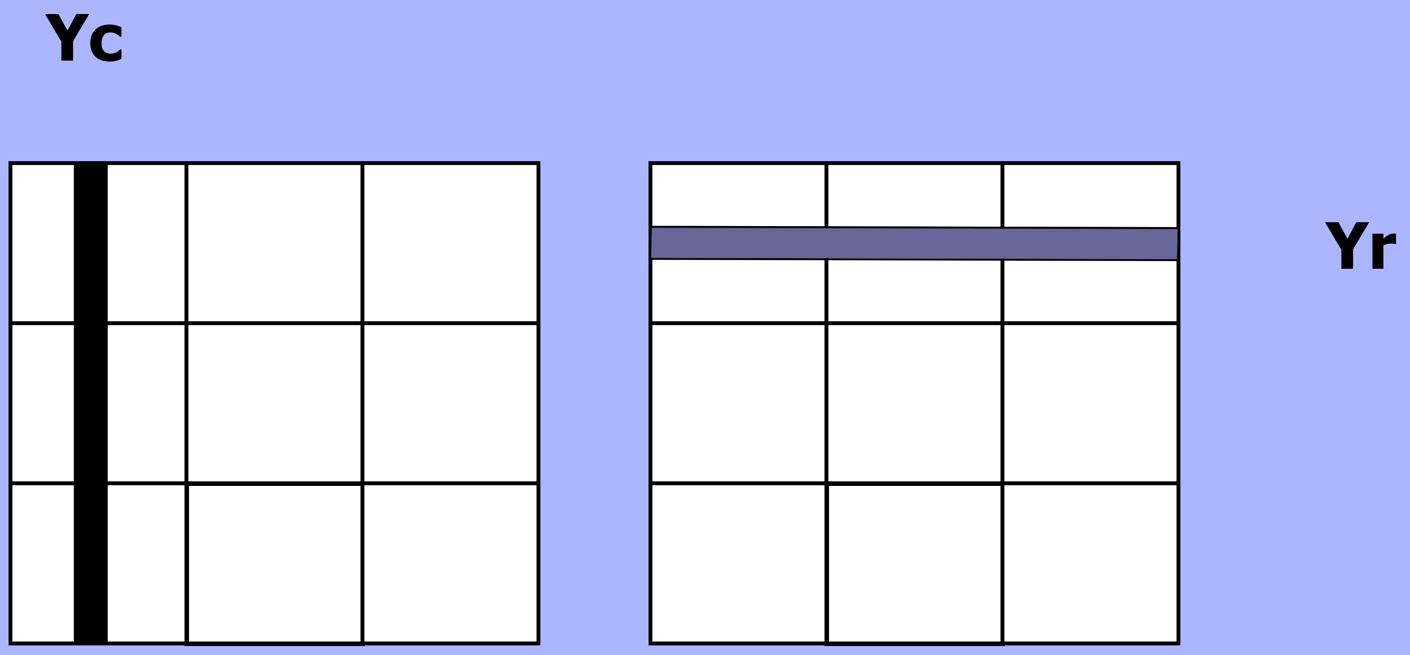
giving..



Now have Yc and Yr



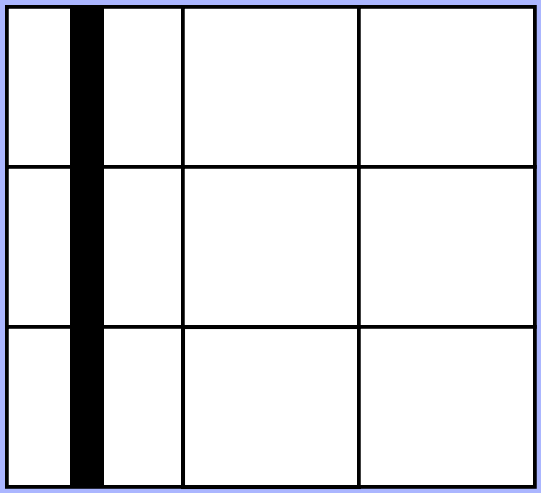
..transpose Yr

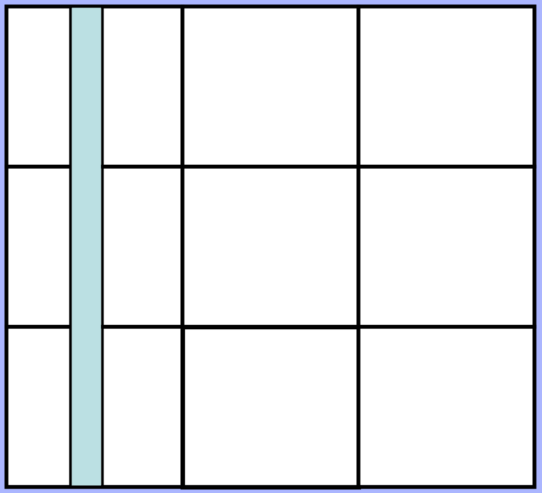


..transpose Yr

Yc

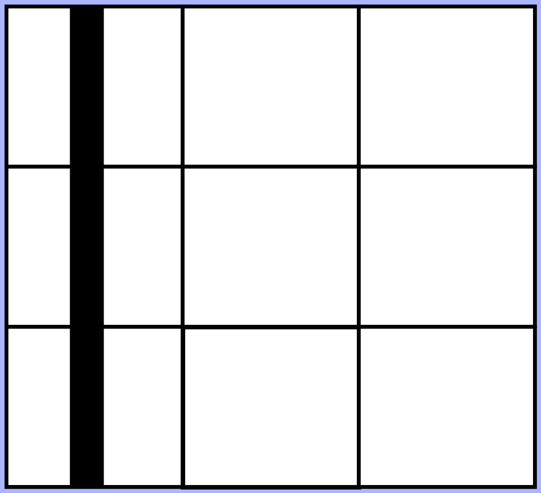
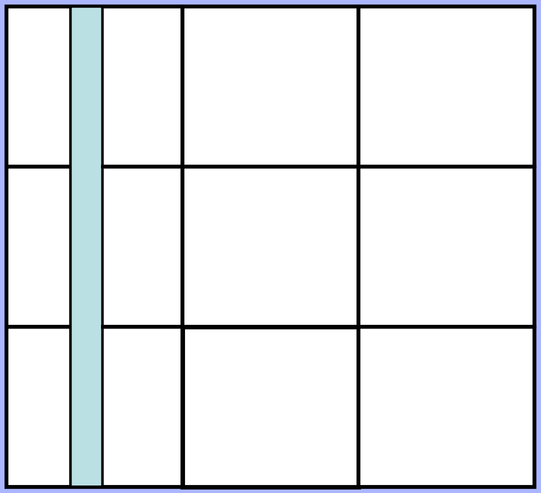
Yr^T





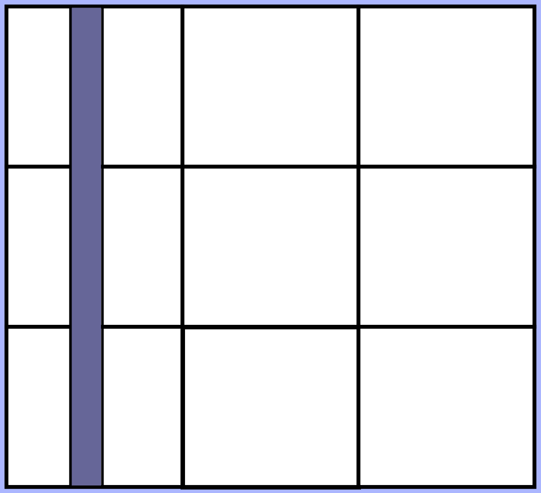
..add together

Yc **Yr^T**

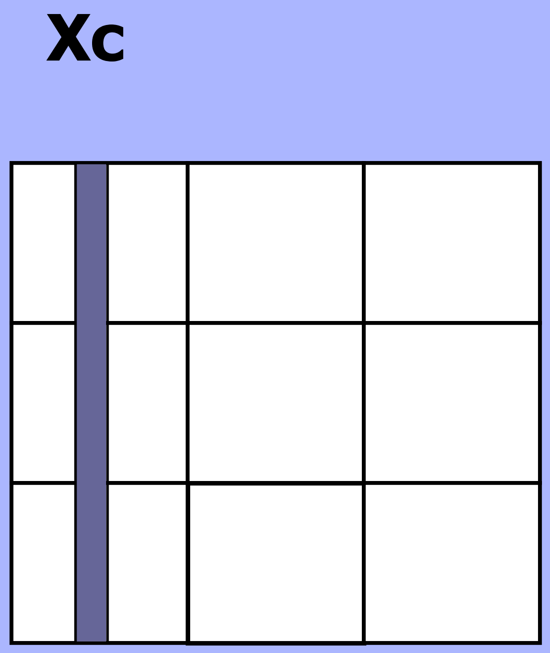



giving…

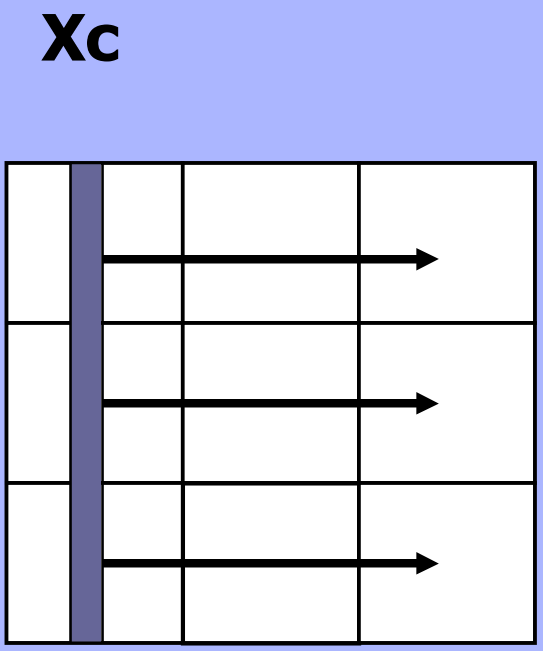
Xc



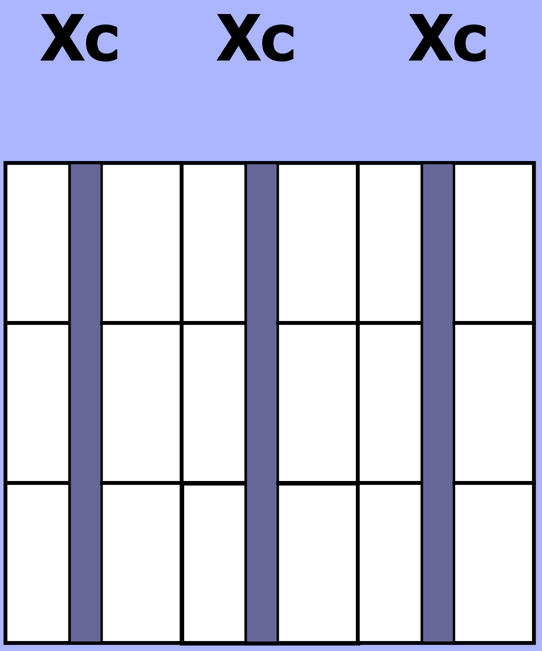
Then, broadcast over entire row



Broadcast over entire row



Broadcast over entire row



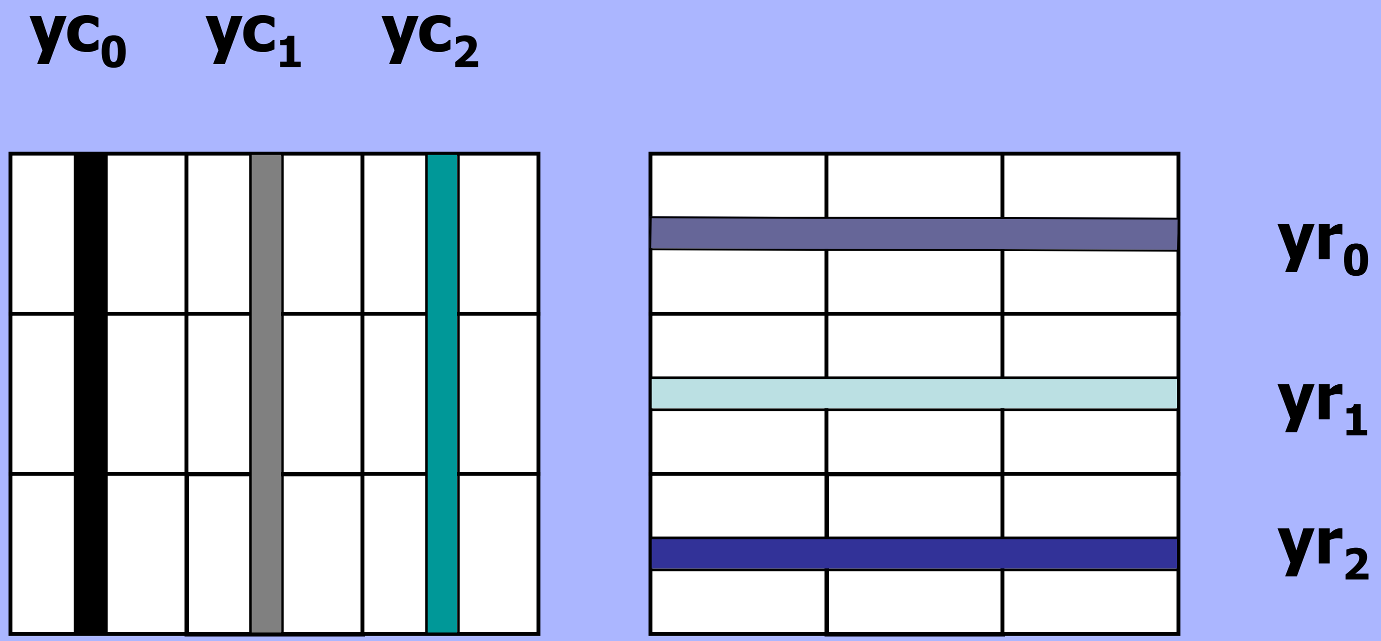
Optimisation

Broadcast and summation code developed using shmem etc

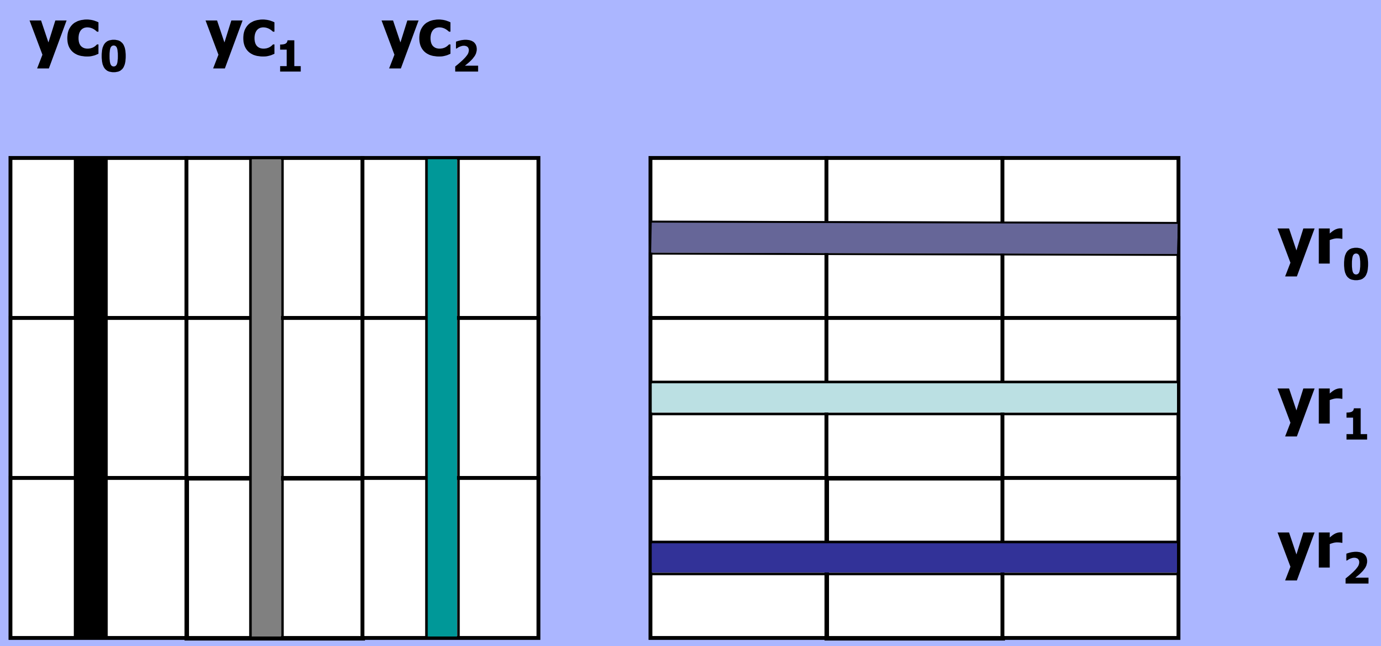
But, the operations here cannot be fused together, since the syncronization would be far too aggressive – code would undoubtedly be slower

..Unless the mathematics is re-ordered

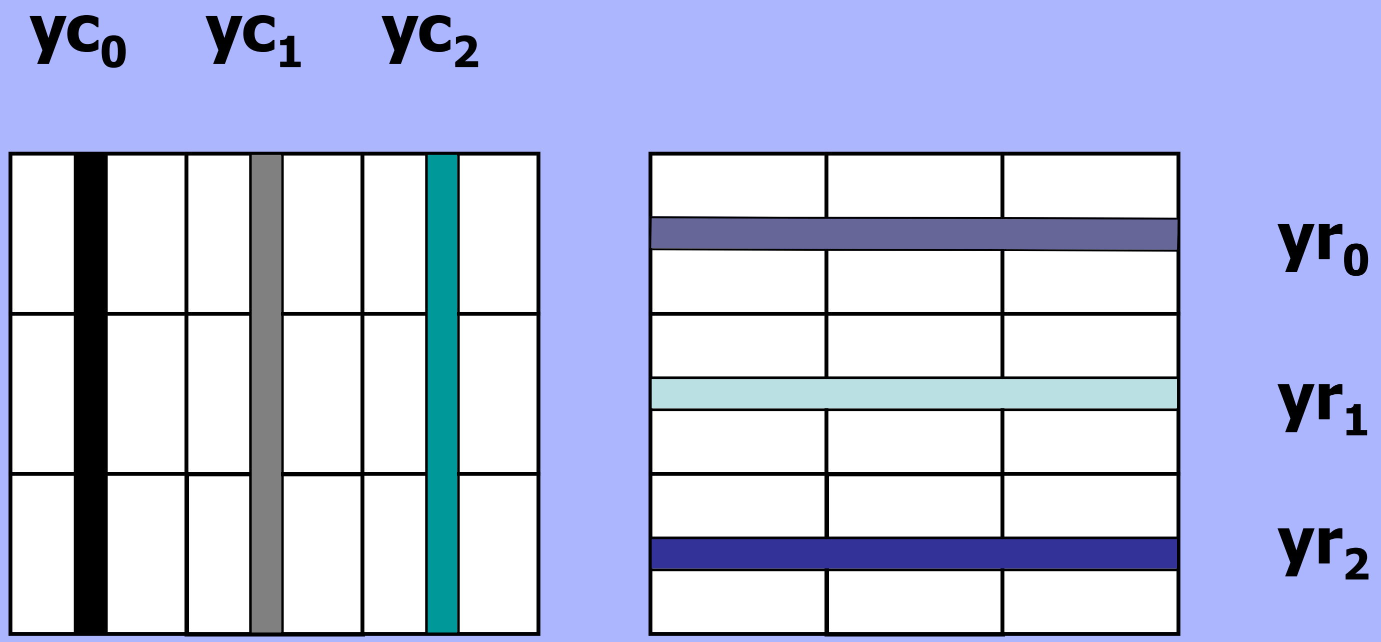
yr and yc



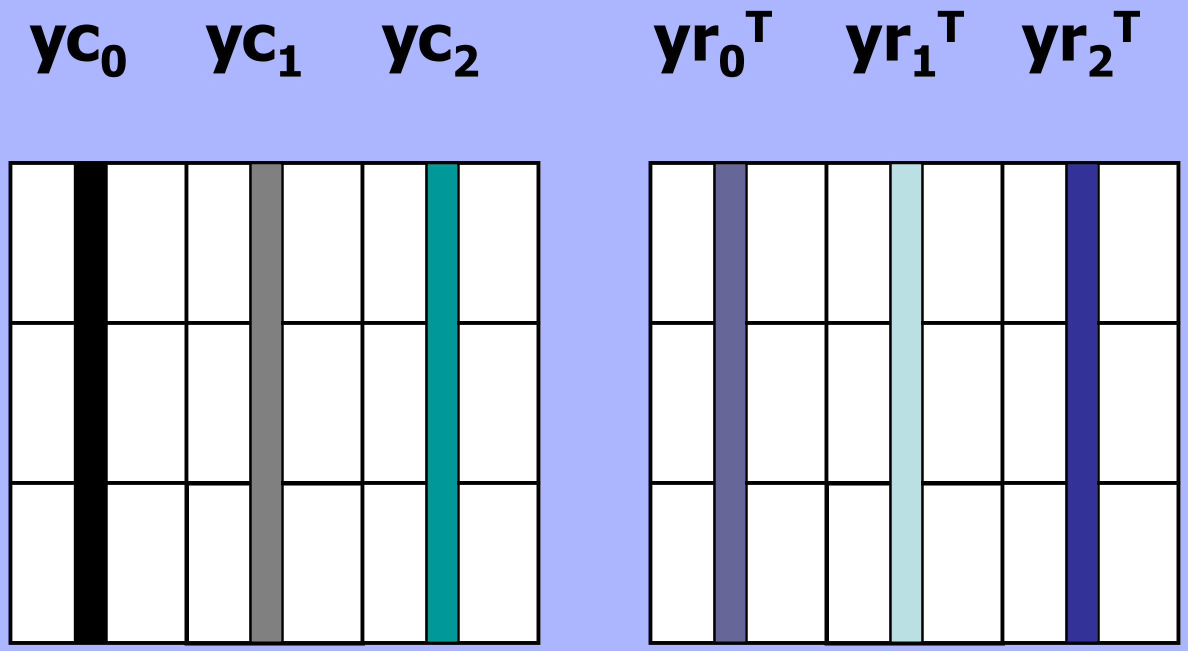
First transpose yr using HPTRAN



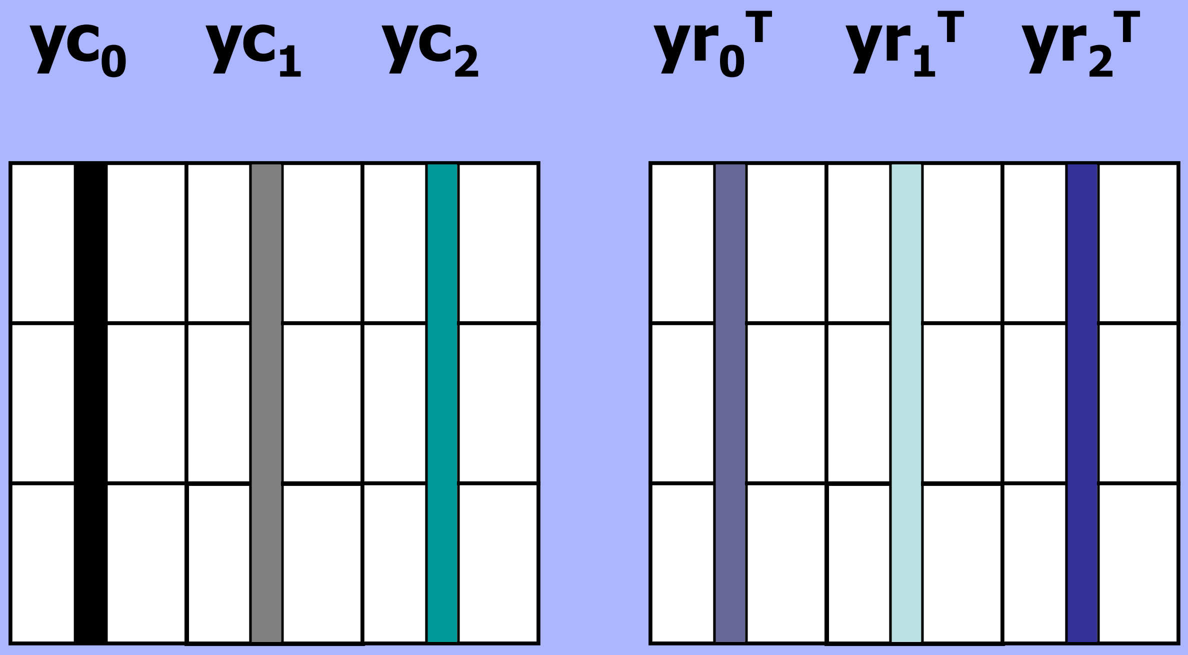
giving..



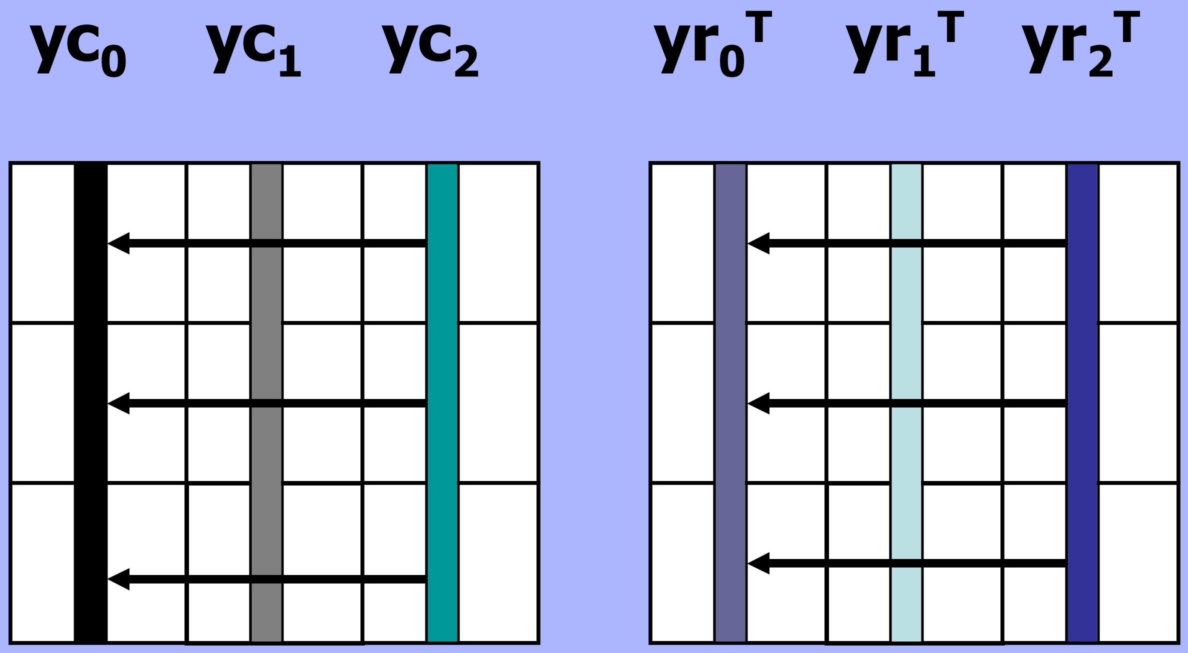
giving..



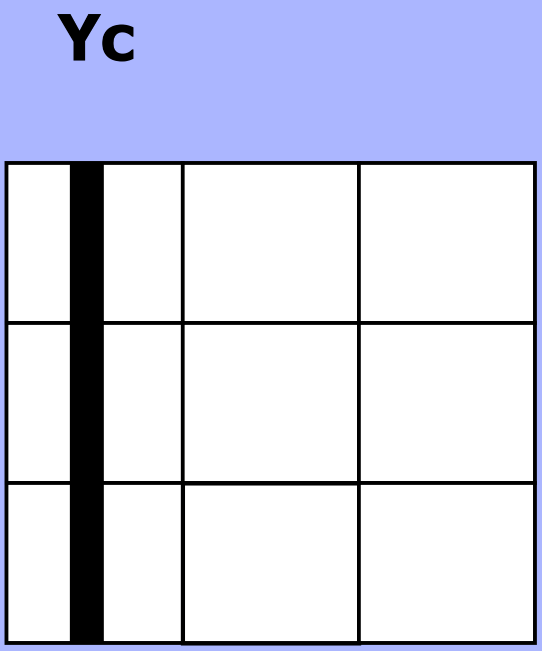
Then sum in unison using HPSUM

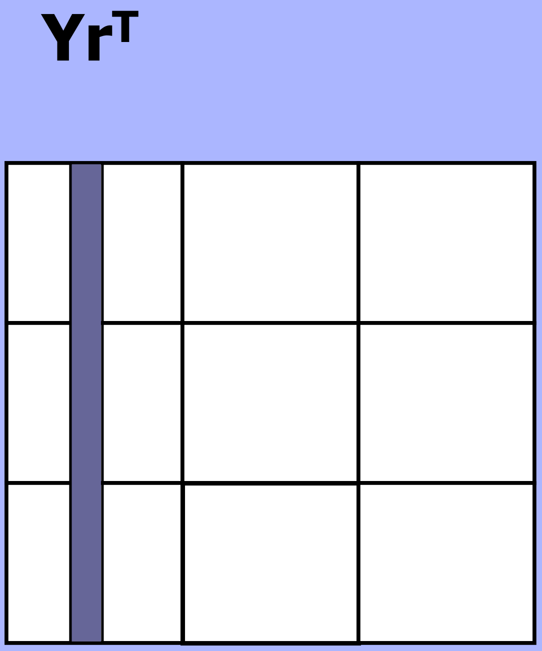


Then sum in unison using HPSUM

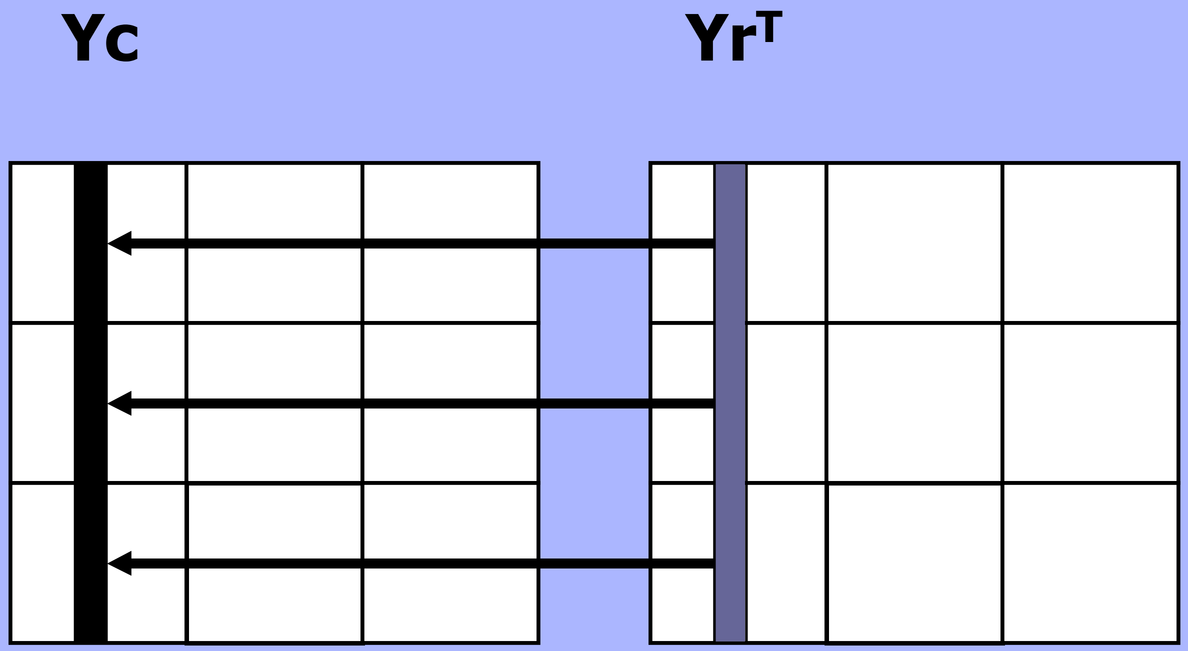


Also within HPSUM, simply add together

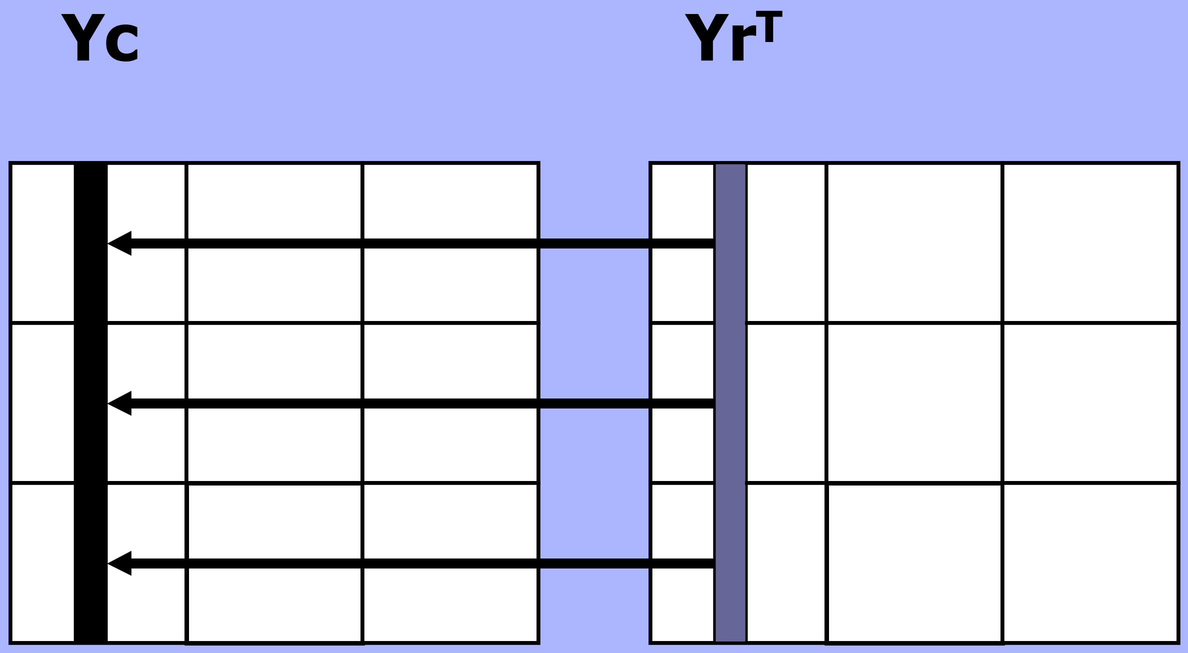




Also within HPSUM, simply add together

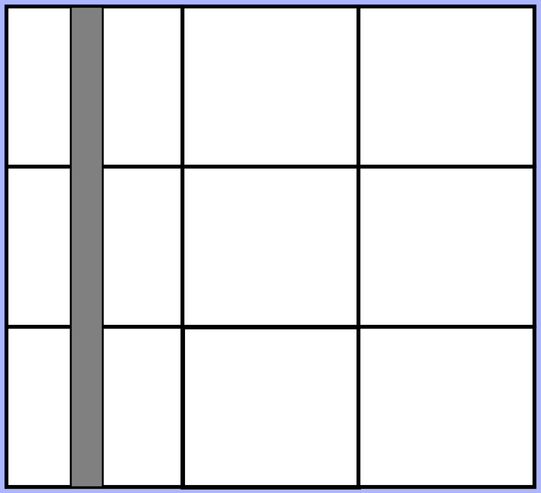


giving..

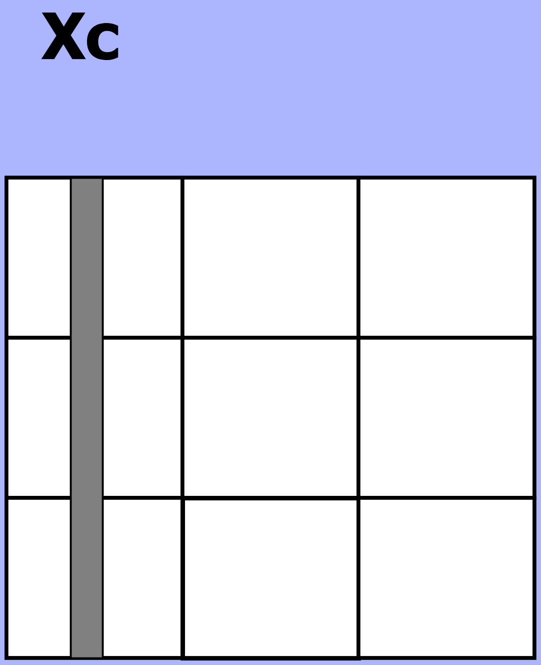


giving..

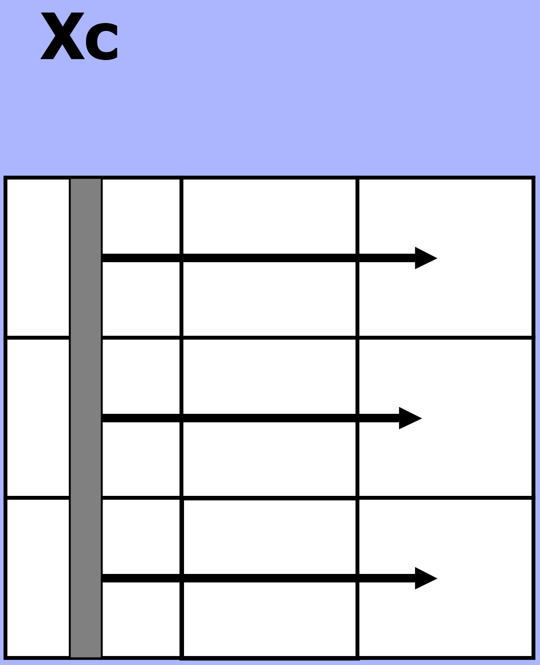
Xc



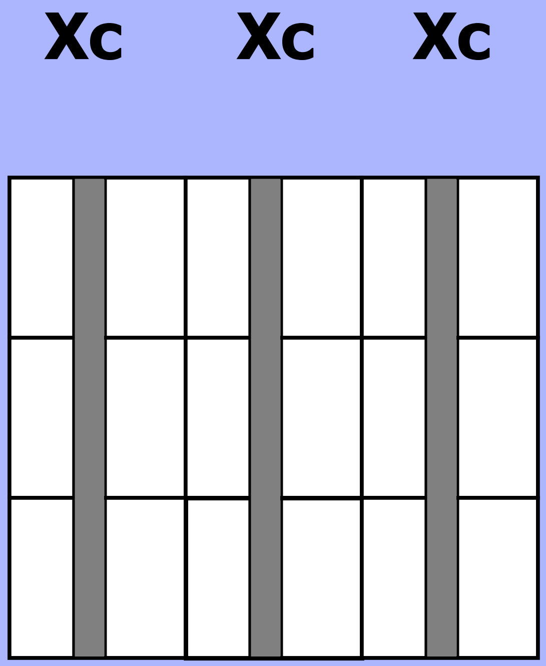
Which can then be broadcast over row process



Which can then be broadcast over row process



Giving same resultant replicated vector

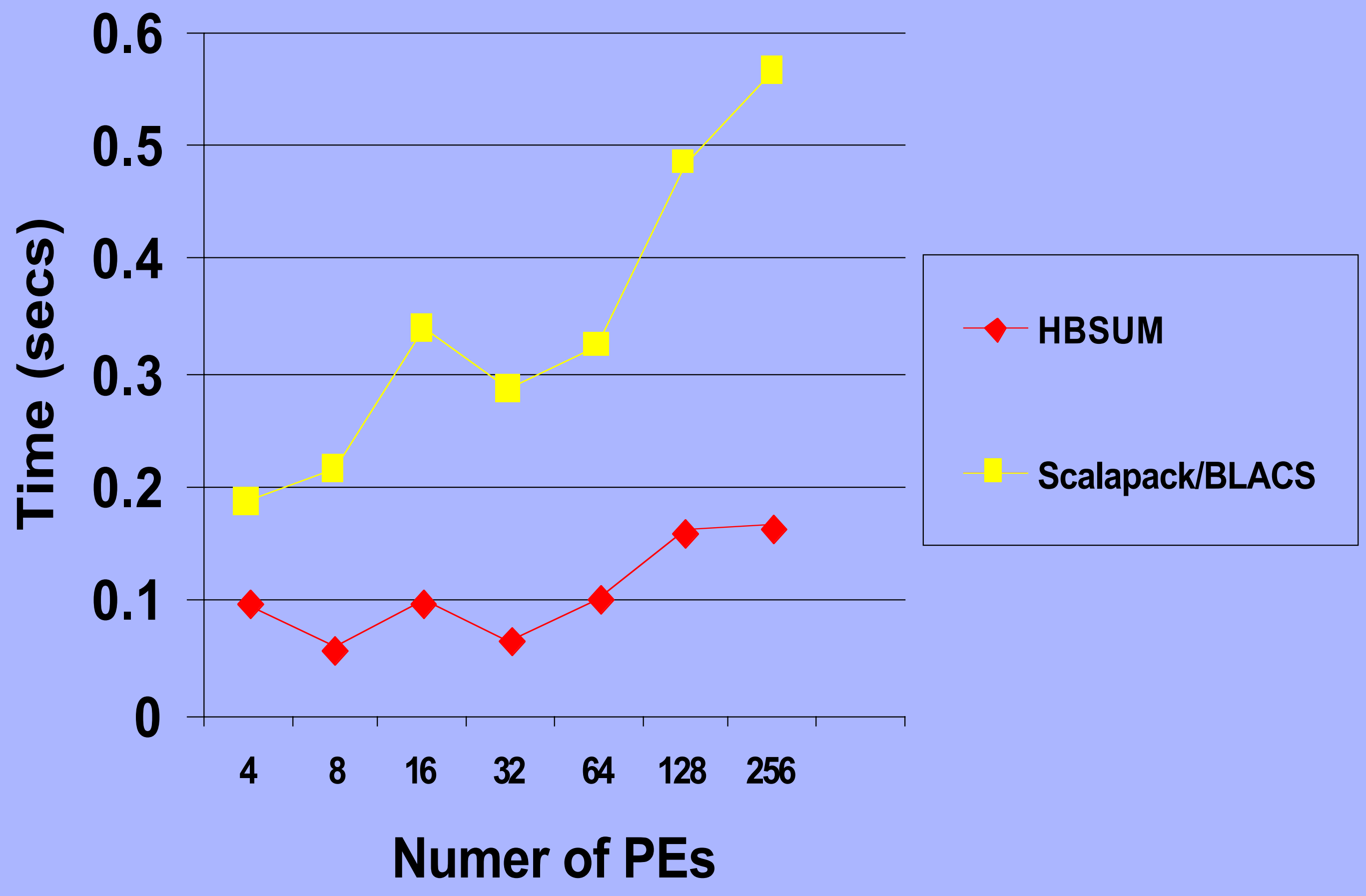


HPBSUM

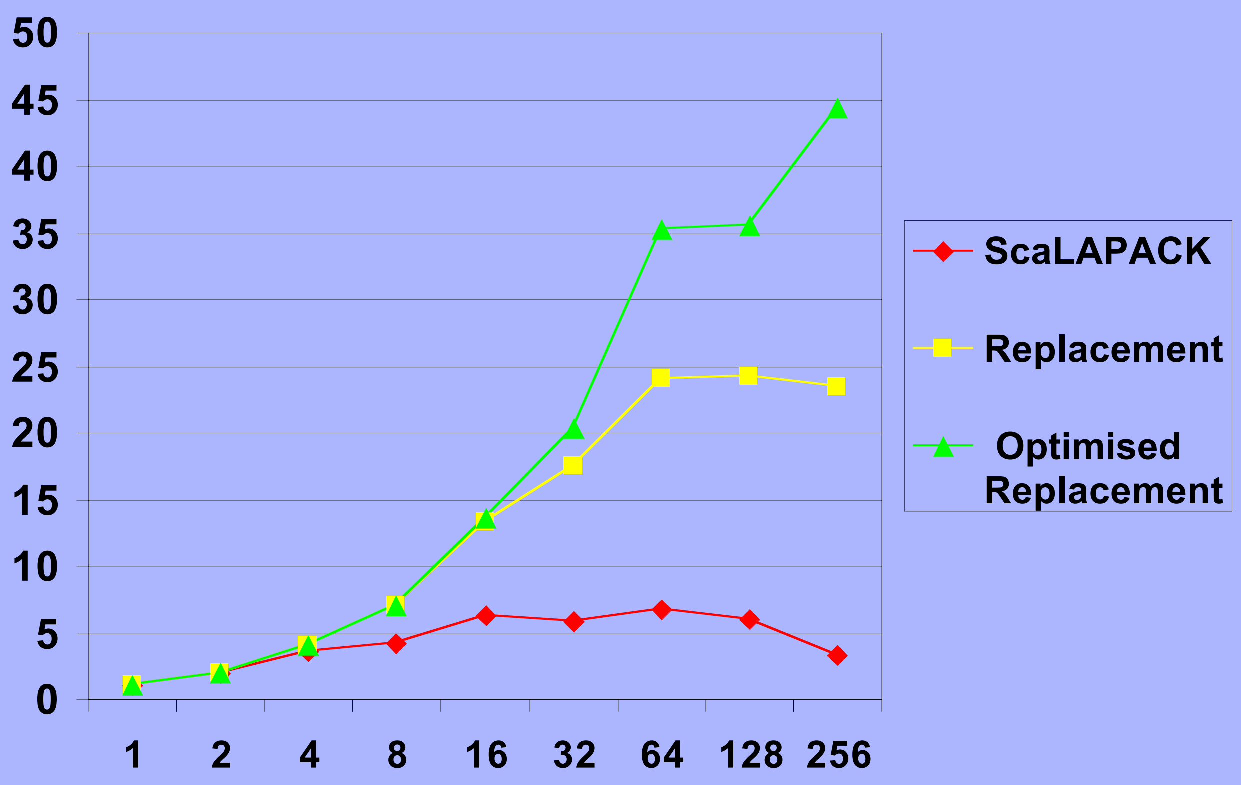
In fact, synchronicity can be shared for summation, local sum and broadcast (partly) to give the same functionality with only 1 synchronization overhead.

Resultant HPBSUM gives vast performance improvement over previous BLACS routines

Performance



Scaling of the whole code



Conclusion

Code fusion can improve performance due to

- Better cache reuse
- Reduction in synchronisation

Sca/LAPACK design philosophy - always best for HPC?

At least in this case, ScaLAPACK **can** be beaten

Further Work

Many ScaLAPACK routines perform badly

CSAR users report ScaLAPACK dependence inhibits production of capability work

Two avenues for future work

- Specific replacement of routines for CSAR users
- Create library of HPC linear algebra routines for Origins