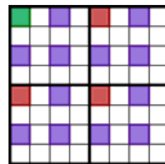
A Brief Introduction to the NAS MG Benchmark

LaR 2005

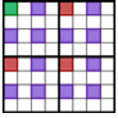
November 12, 2005



Brad Chamberlain

Cray Inc.

CRAY



NAS Parallel Benchmarks (NPB)

- A set of benchmarks developed…

  …in the early `90's by NASA's Advanced Supercomputing division

  …to model computations and data access patterns from CFD codes

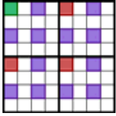
  …originally released as paper & pencil benchmarks (v1.x)

  …then as MPI reference implementations (v2.x)

  …now versions available in a variety of languages

    - Java, OpenMP, HPF (v3.x)
    - UPC, Co-Array Fortran, Titanium, ZPL, … (by respective groups)
- Among the most useful benchmark suites in HPC
  - well-designed and -maintained
  - good variety of data access patterns, communication requirements
  - open-source
  - well-understood, -used

CRAY



NAS Parallel Benchmarks (NPB)

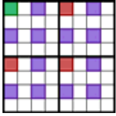
- 8 Benchmarks:
  - 5 kernels:
    - **EP**: embarrassingly parallel
    - **MG**: multigrid
    - **CG**: conjugate gradiant
    - **FT**: Fourier transform
    - **IS**: integer sort
  - 3 pseudo-applications
    - **BT**: block transpose
    - **LU**: LU factorization
    - **SP**: pentadiagonal

– Though useful, also domain-specific
  - focus on CFD algorithms is good, but restrictive
  - other HPC application areas would do well to create similar suites

– Often difficult to understand from the code
  - terse variable names
  - SPMD-style programming details

http://www.nas.nasa.gov/Software/NPB/

CRAY



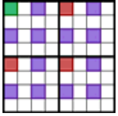
NPB Basics

- Each benchmark supports a number of problem sizes:
  - **S**: Small
  - **W**: Workstation
  - **A**, **B**, **C**, **D**: "Production" sizes (added as machines grow)
- NPB codes use common…

  …timing routines

  …pseudo-random number generators

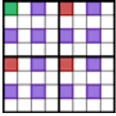
  …output routines
- NAS implementations are simply "one way to write the algorithm"
  - typically written to work well with both caches and vectors
  - how closely they should be followed can be a tricky question



The NAS MG Benchmark

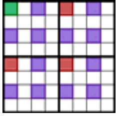
Mathematically: use a 3D multigrid method to find an approximate solution to a discrete Poisson problem ($\nabla^2 u = v$)

CRAY



Disclaimers

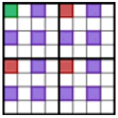
- I tend to take a very operational approach to describing algorithms
  - what are data structures?
  - what operations are performed on them?
  - what data distribution is used for parallel version?
  - what communication is required?
- I'm not a multigrid expert
  - More of a parallel algorithm/language person
  - A good deal of experience with NAS MG



NAS MG: Operational View

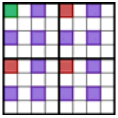
- Data structures:
  - 3D arrays & 3D hierarchical arrays (2D in my pictures)
  - 3D sparse arrays can also be useful
- 4 primary kernels:
  - each computed using 27-point stencils
    - **resid**: compute residual
    - **psinv**: compute approximate inverse
    - **rprj3**: projection from fine grid to coarse
    - **interp**: interpolation from coarse grid to fine
  - periodic boundary conditions
- computation of approximate norms
  - **norm2u3**: approximate L2 & uniform norms
- initialization, output

CRAY



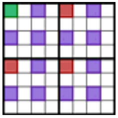
NAS MG: Parallel Implementation

- Arrays typically use 3D block distributions
  - optimal load balance
  - good surface-to-volume ratio
  - ghost cells allocated for caching neighbors' values
- Communication Idioms:
  - 4 kernels require point-to-point communication
  - toroidal communication required for boundaries
  - global reductions required to compute norms
  - reductions useful during initialization as well

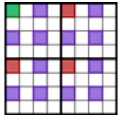


Hierarchical Arrays

	level 0	level 1	level 2	level 3
conceptually:				
dense indexing:	(1:8,1:8)	(1:4,1:4)	(1:2,1:2)	(1:1,1:1)
strided indexing:	(1:8:1,1:8:1)	(1:8:2,1:8:2)	(1:8:4,1:8:4)	(1:8:8,1:8:8)



Hierarchical Arrays



MG's arrays

V:

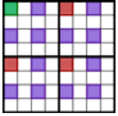
input array

U:

R:

hierarchical work arrays

CRAY



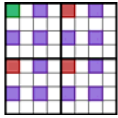
Overview of MG

run timed portion

initialize V

output norms & timings

CRAY



MG's Timed Portion

repeat *nit* times

resid

resid

norm2u3

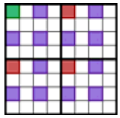
- *rnm2*
- *rnmu*

mg3p

Configurations

S:	32^3	(4 iterations)
W:	64^3	(40 iterations)
A:	256^3	(4 iterations)
B:	256^3	(20 iterations)
C:	512^3	(20 iterations)
D:	1024^3	(50 iterations)

CRAY



MG's Guts (*mg3P*)

rprj3

rprj3

rprj3

psinv

interp

resid

psinv

interp

resid

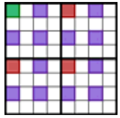
psinv

interp

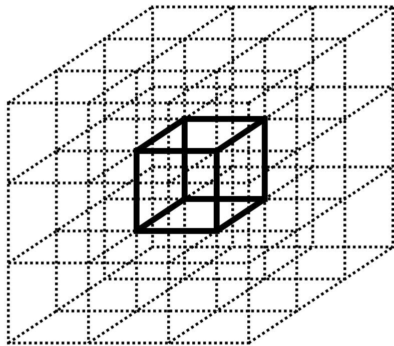
resid

psinv

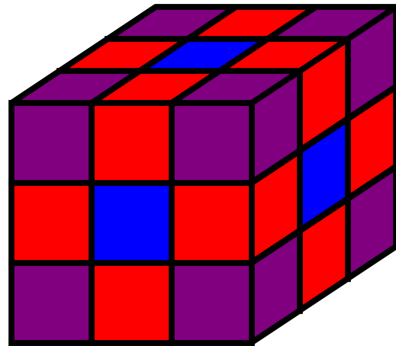
CRAY

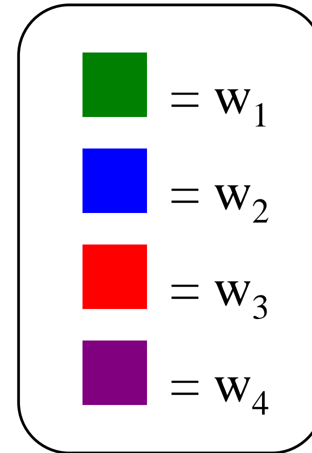


27-point stencils

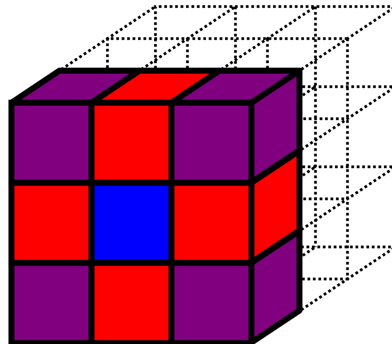


=

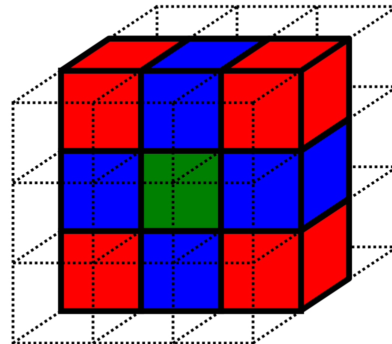




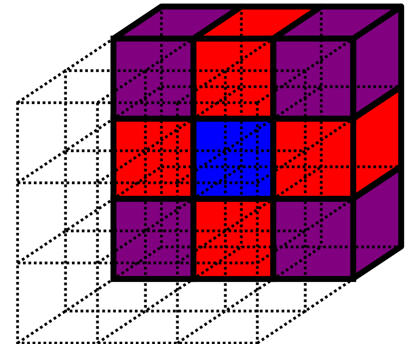
=

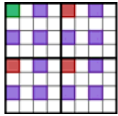


+



+



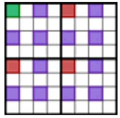


Our First Stencil: *resid*(R, V, U)

R = V - convolve(, U)

R = V - $\sum$ U

![Cray logo]

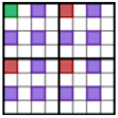


Periodic Boundary Conditions

$$R = V - \sum U$$

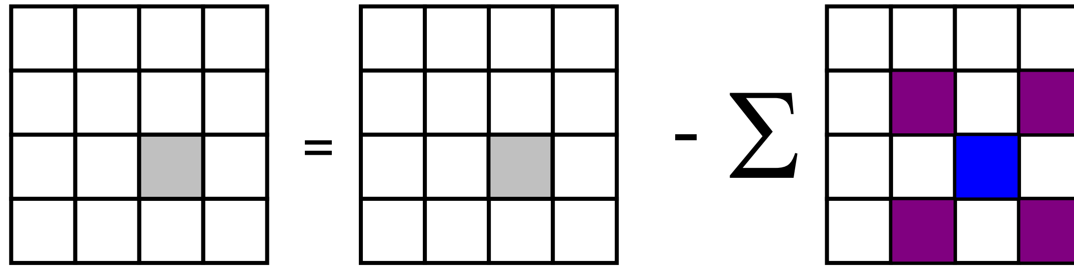
$$R = V - \sum U$$

CRAY

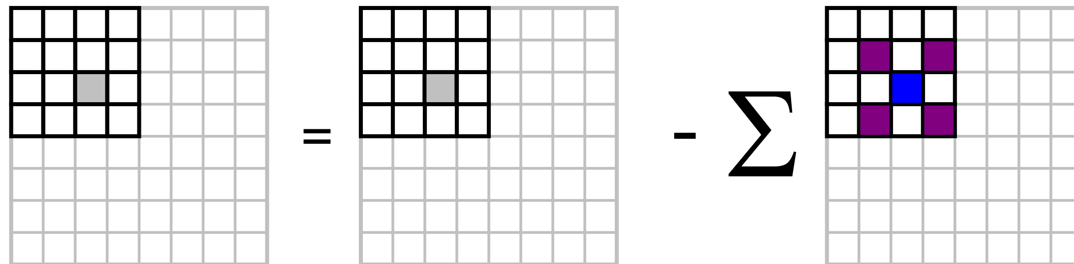


At Other Levels of the Hierarchy

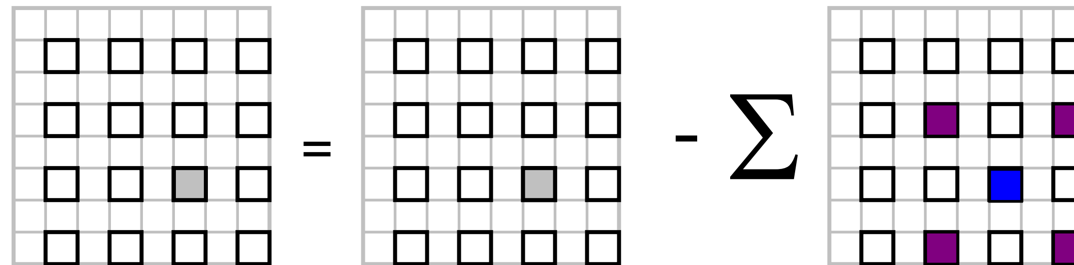
conceptually:



dense indexing:

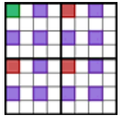


strided indexing:

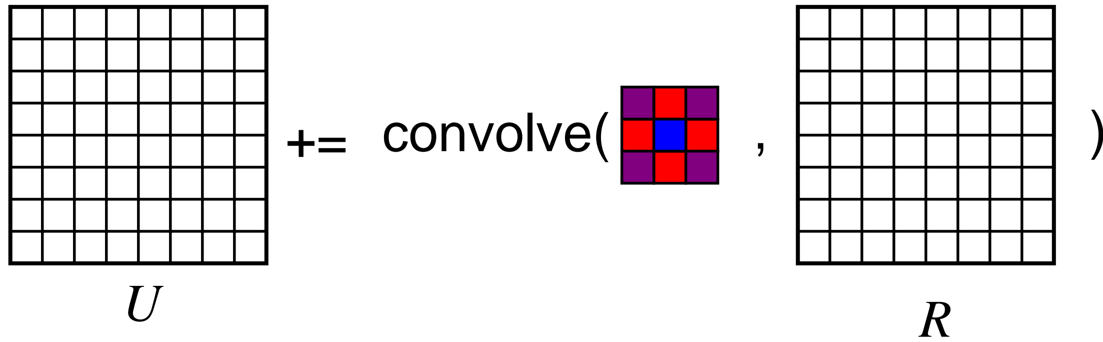


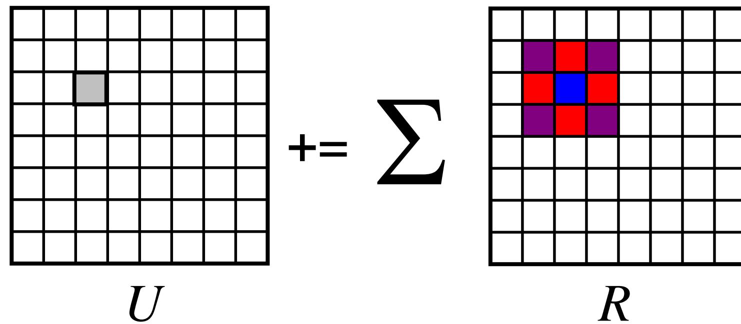
R *V* *U*

CRAY

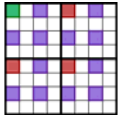


psinv(U, R)

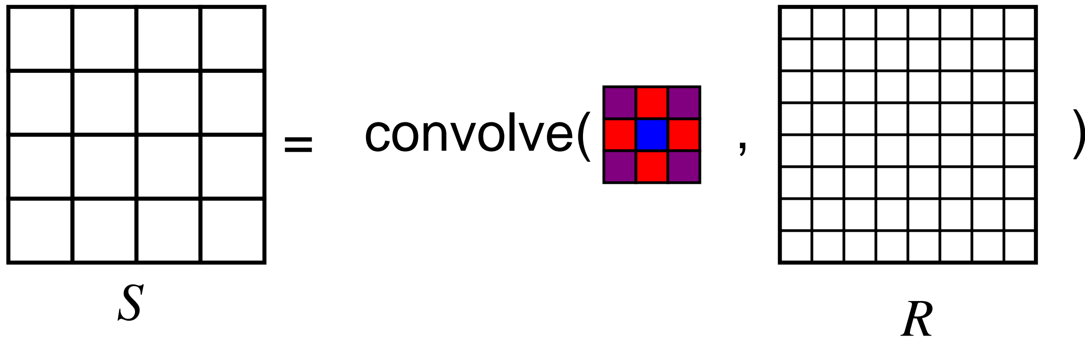


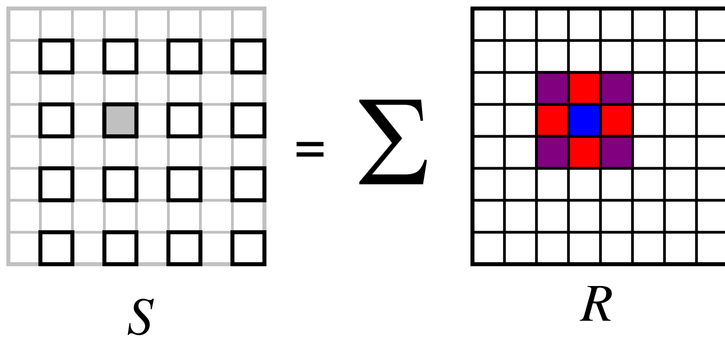


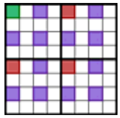
CRAY



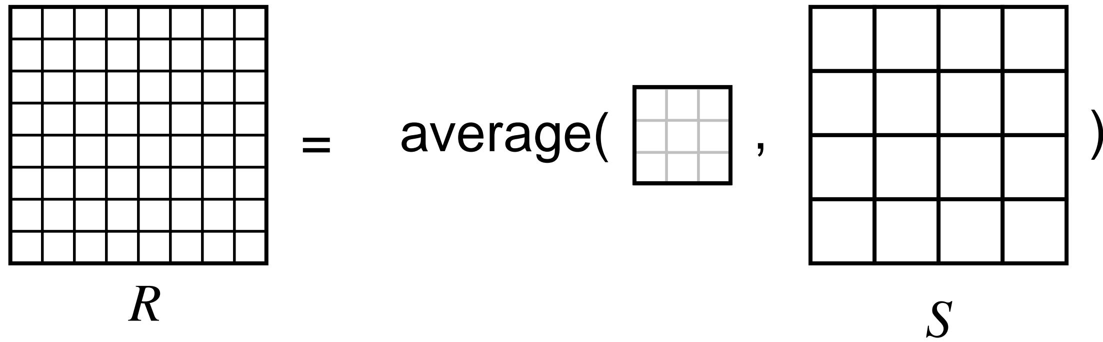
rprj3(S, R)

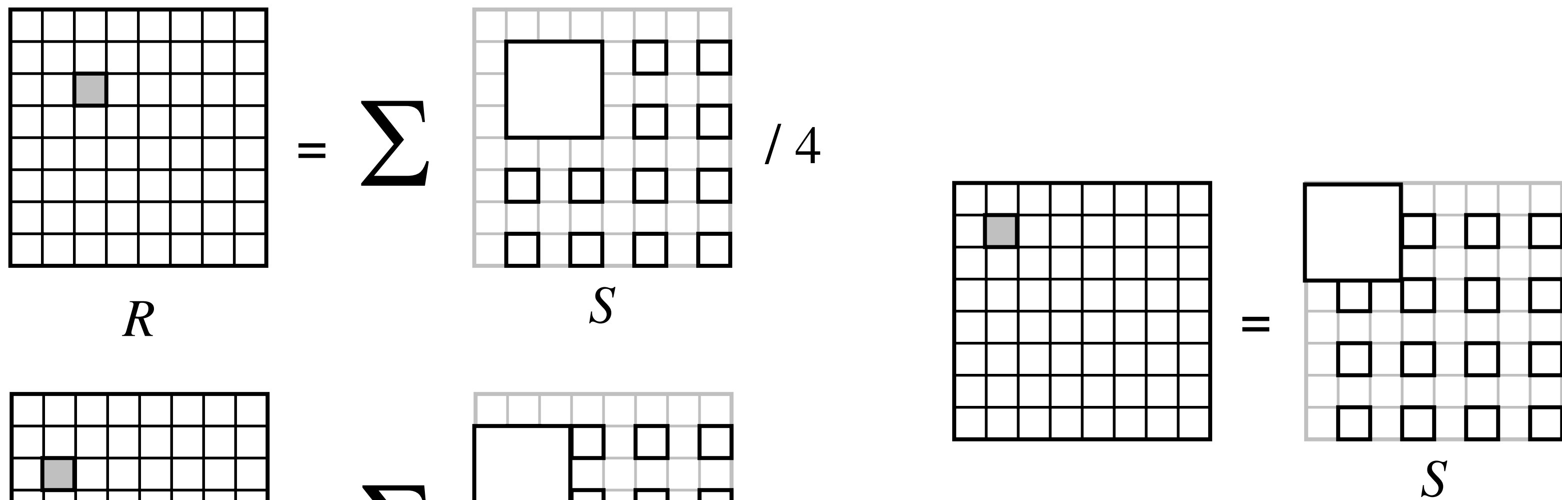


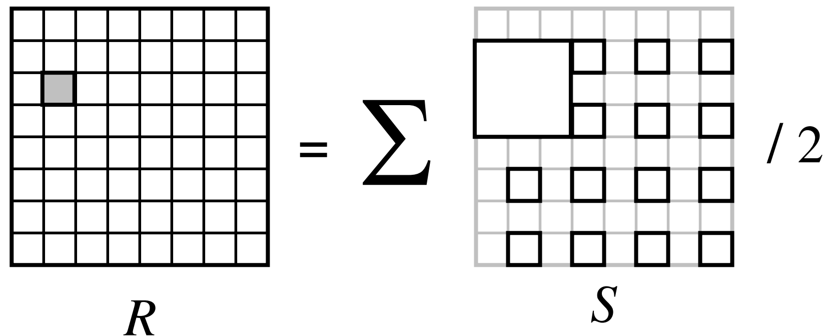


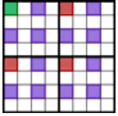


interp(R, S)





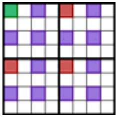




rprj3 in Fortran

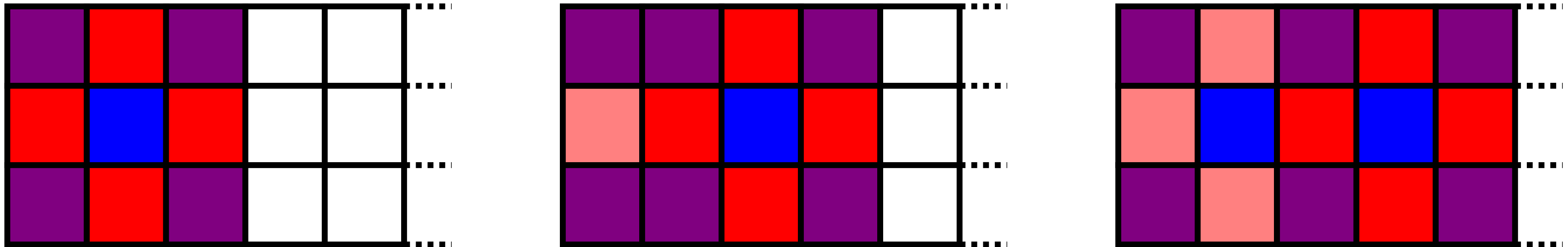
```
   do   j3=2,m3j-1
    i3 = 2*j3-d3
    do  j2=2,m2j-1
      i2 = 2*j2-d2
      do  j1=2,m1j-1
       i1 = 2*j1-d1
       s(j1,j2,j3) =
>           0.5D0 * r(i1,i2,i3)
>         + 0.25D0 * (r(i1-1,i2,i3) + r(i1+1,i2,i3)
>                    + r(i1,  i2-1,i3  ) + r(i1, i2+1,i3  )
>                    + r(i1,  i2,  i3-1) + r(i1, i2,  i3+1))
>         + 0.125D0 * (r(i1-1,i2-1,i3  ) + r(i1-1,i2+1,i3  )
>                    + r(i1-1,i2,  i3-1) + r(i1-1,i2  ,i3+1)
>                    + r(i1+1,i2-1,i3  ) + r(i1+1,i2+1,i3  )
>                    + r(i1+1,i2,  i3-1) + r(i1+1,i2  ,i3+1))
>                    + r(i1,  i2-1,i3-1) + r(i1,  i2-1,i3+1)
>                    + r(i1,  i2+1,i3-1) + r(i1,  i2+1,i3+1))
>         + 0.0625D0 * (r(i1-1,i2-1,i3-1) + r(i1—1,i2-1,i3+1)
>                     + r(i1-1,i2+1,i3-1) + r(i1-1,i2+1,i3+1)
>                     + r(i1+1,i2-1,i3-1) + r(i1+1,i2-1,i3+1)
>                     + r(i1+1,i2+1,i3-1) + r(i1+1,i2+1,i3+1))
       enddo
     enddo
   enddo
```

CRAY

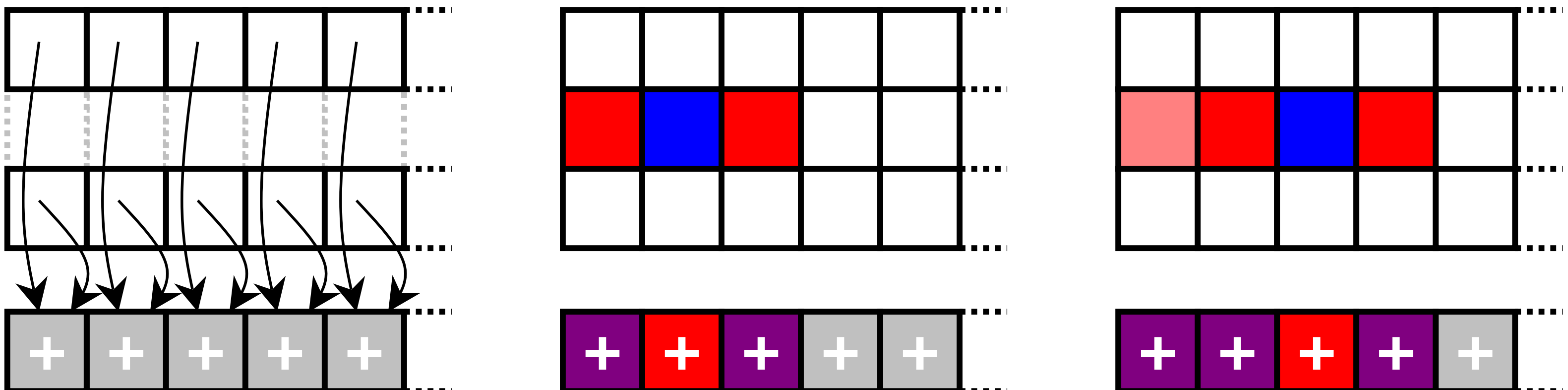


Stencil Optimization (2D)

- Adjacent stencils use common subexpressions:

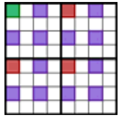


- Observation: Cache partial sums for reuse…



- Benefits are greater for 3D stencils…

CRAY



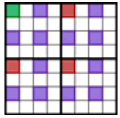
MG Stencil Optimization

+ + =

+ +

+ + +

CRAY



MG Stencil Optimization

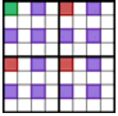
+

+

+

+

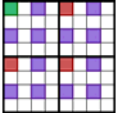
CRAY



rprj3 in Fortran with stencil opt.

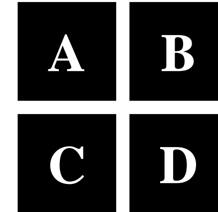
```
  do  j3=2,m3j-1
   i3 = 2*j3-d3
   do  j2=2,m2j-1
     i2 = 2*j2-d2
     do j1=2,m1j
      i1 = 2*j1-d1
      x1(i1-1) = r(i1-1,i2-1,i3  ) + r(i1-1,i2+1,i3  )
>              + r(i1-1,i2,  i3-1) + r(i1-1,i2,  i3+1)
      y1(i1-1) = r(i1-1,i2-1,i3-1) + r(i1-1,i2-1,i3+1)
>              + r(i1-1,i2+1,i3-1) + r(i1-1,i2+1,i3+1)
     enddo
     do  j1=2,m1j-1
      i1 = 2*j1-d1
      y2 = r(i1,  i2-1,i3-1) + r(i1,  i2-1,i3+1)
>        + r(i1,  i2+1,i3-1) + r(i1,  i2+1,i3+1)
      x2 = r(i1,  i2-1,i3  ) + r(i1,  i2+1,i3  )
>        + r(i1,  i2,  i3-1) + r(i1,  i2,  i3+1)
      s(j1,j2,j3) =
>          0.5D0 * r(i1,i2,i3)
>        + 0.25D0 * (r(i1-1,i2,i3) + r(i1+1,i2,i3) + x2)
>        + 0.125D0 * ( x1(i1-1) + x1(i1+1) + y2)
>        + 0.0625D0 * ( y1(i1-1) + y1(i1+1) )
      enddo
    enddo
  enddo
```

CRAY

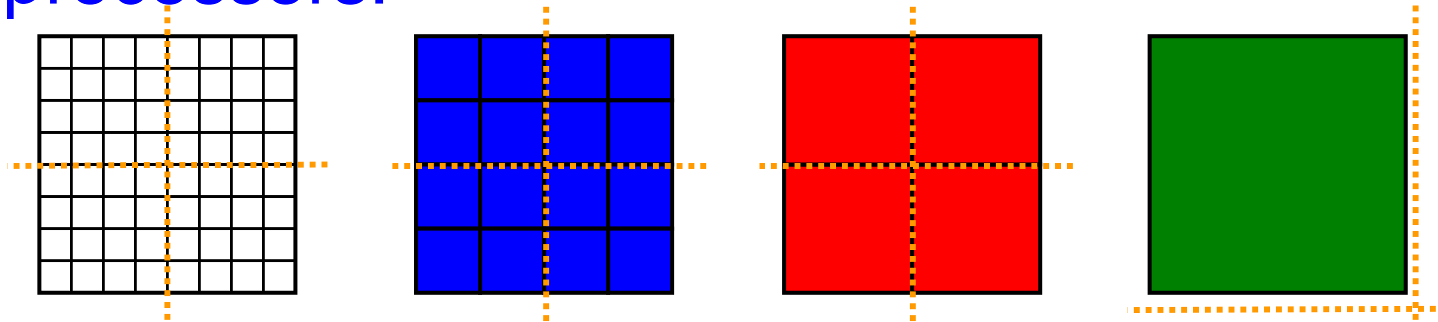


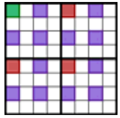
Parallel Data Distribution

- Given a virtual processor grid…



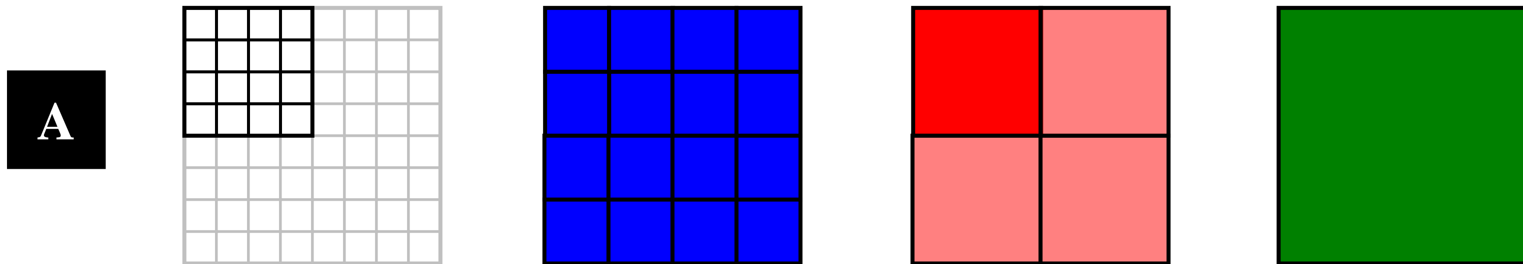
…arrays are block-distributed between processors:



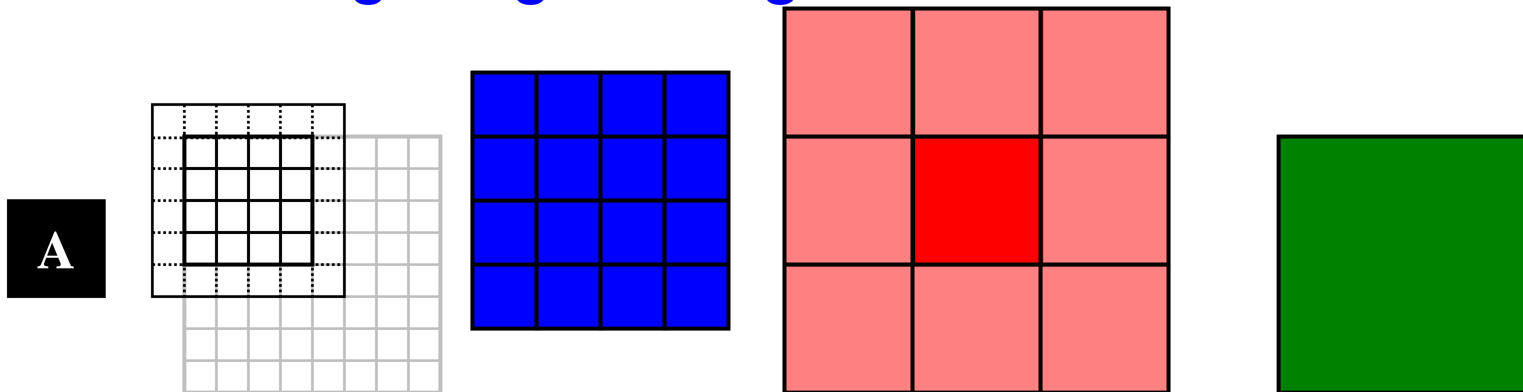


Per-processor Data Allocation

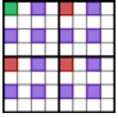
- In addition to its local block of values…



…each processor allocates ghost cells for caching neighboring values

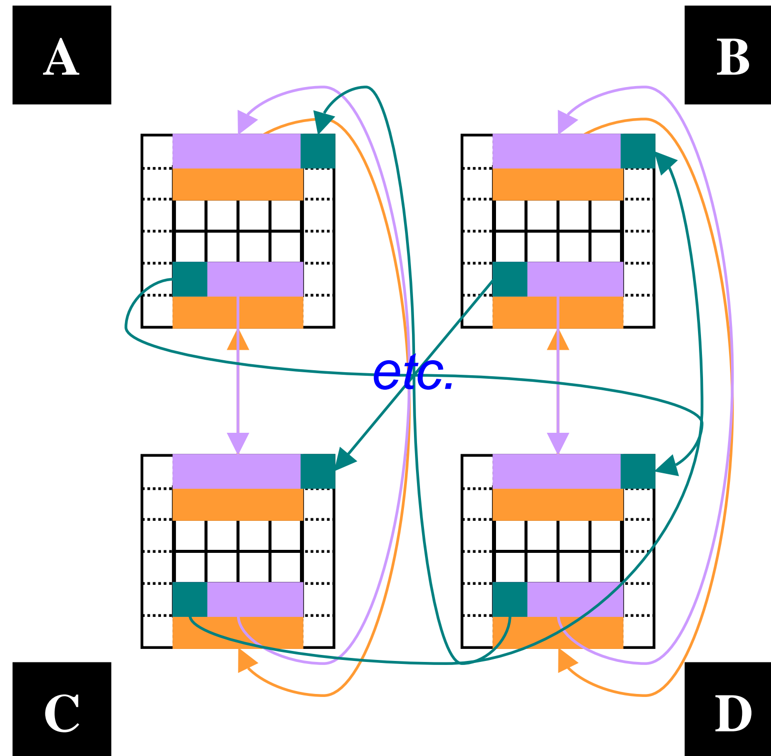


CRAY



Stencil Communication

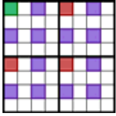
Prior to computing a stencil, communication is typically required to refresh the ghost cells



Notes:

- Lots of optimization opportunities
- Have to eventually start skipping processors for coarser levels

CRAY



Distributed *rprj3* in Fortran

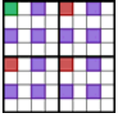
```fortran
subroutine rprj3(r,m1k,m2k,m3k,s,m1j,m2j,m3j,k)
implicit none
include 'cafnpb.h'
include 'globals.h'

integer m1k, m2k, m3k, m1j, m2j, m3j,k

double precision r(m1k,m2k,m3k), s(m1j,m2j,m3j)
integer j3, j2, j1, i3, i2, i1, d1, d2, d3, j
double precision x1(m), y1(m), x2,y2

if(m1k.eq.3)then
  d1 = 2
else
  d1 = 1
endif

if(m2k.eq.3)then
  d2 = 2
else
  d2 = 1
endif

if(m3k.eq.3)then
  d3 = 2
else
  d3 = 1
endif
```

```fortran
      do   j3=2,m3j-1
       i3 = 2*j3-d3
       do  j2=2,m2j-1
         i2 = 2*j2-d2
         do j1=2,m1j
          i1 = 2*j1-d1
          x1(i1-1) = r(i1-1,i2-1,i3  ) + r(i1-1,i2+1,i3  )
>                  + r(i1-1,i2,  i3-1) + r(i1-1,i2,  i3+1)
          y1(i1-1) = r(i1-1,i2-1,i3-1) + r(i1-1,i2-1,i3+1)
>                  + r(i1-1,i2+1,i3-1) + r(i1-1,i2+1,i3+1)
         enddo
         do  j1=2,m1j-1
          i1 = 2*j1-d1
          y2 = r(i1,  i2-1,i3-1) + r(i1,  i2-1,i3+1)
>            + r(i1,  i2+1,i3-1) + r(i1,  i2+1,i3+1)
          x2 = r(i1,  i2-1,i3  ) + r(i1,  i2+1,i3  )
>            + r(i1,  i2,  i3-1) + r(i1,  i2,  i3+1)
          s(j1,j2,j3) =
>               0.5D0 * r(i1,i2,i3)
>             + 0.25D0 * (r(i1-1,i2,i3) + r(i1+1,i2,i3) + x2)
>             + 0.125D0 * ( x1(i1-1) + x1(i1+1) + y2)
>             + 0.0625D0 * ( y1(i1-1) + y1(i1+1) )
          enddo
        enddo
      enddo
      j = k-1
      call comm3(s,m1j,m2j,m3j,j)
      return
      end
```

Fortran MPI Communication

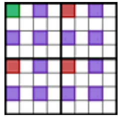
```fortran
subroutine comm3(u,n1,n2,n3,kk)
use caf_intrinsics

implicit none

include 'cafnpb.h'
include 'globals.h'

integer n1, n2, n3, kk
double precision u(n1,n2,n3)
integer axis

if( .not. dead(kk) )then
   do  axis = 1, 3
      if( nprocs .ne. 1) then
         call sync_all()
         call give3( axis, +1, u, n1, n2, n3, kk )
         call give3( axis, -1, u, n1, n2, n3, kk )
         call sync_all()
         call take3( axis, -1, u, n1, n2, n3 )
         call take3( axis, +1, u, n1, n2, n3 )
      else
         call comm1p( axis, u, n1, n2, n3, kk )
      endif
   enddo
else
   do  axis = 1, 3
      call sync_all()
      call sync_all()
   enddo
   call zero3(u,n1,n2,n3)
endif
return
end


subroutine give3( axis, dir, u, n1, n2, n3, k )
use caf_intrinsics


implicit none

include 'cafnpb.h'
include 'globals.h'

integer axis, dir, n1, n2, n3, k, ierr
double precision u( n1, n2, n3 )

integer i3, i2, i1, buff_len,buff_id

buff_id = 2 + dir
buff_len = 0

if( axis .eq.  1 )then
   if( dir .eq. -1 )then

      do  i3=2,n3-1
         do  i2=2,n2-1
            buff_len = buff_len + 1
            buff(buff_len,buff_id ) = u( 2,  i2,i3)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   else if( dir .eq. +1 ) then

      do  i3=2,n3-1
         do  i2=2,n2-1
            buff_len = buff_len + 1
            buff(buff_len, buff_id ) = u( n1-1, i2,i3)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   endif
endif

if( axis .eq.  2 )then
   if( dir .eq. -1 )then

      do  i3=2,n3-1
         do  i1=1,n1
            buff_len = buff_len + 1
            buff(buff_len, buff_id ) = u( i1,  2,i3)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   else if( dir .eq. +1 ) then

      do  i3=2,n3-1
         do  i1=1,n1
            buff_len = buff_len + 1
            buff(buff_len,  buff_id )= u( i1,n2-1,i3)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   endif
endif

if( axis .eq.  3 )then
   if( dir .eq. -1 )then

      do  i2=1,n2
         do  i1=1,n1
            buff_len = buff_len + 1
            buff(buff_len, buff_id ) = u( i1,i2,2)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   else if( dir .eq. +1 ) then

      do  i2=1,n2
         do  i1=1,n1
            buff_len = buff_len + 1
            buff(buff_len, buff_id ) = u( i1,i2,n3-1)
         enddo
      enddo

      buff(1:buff_len,buff_id+1)[nbr(axis,dir,k)] =
>     buff(1:buff_len,buff_id)

   endif
endif

return
end


subroutine take3( axis, dir, u, n1, n2, n3 )
use caf_intrinsics


implicit none

include 'cafnpb.h'
include 'globals.h'

integer axis, dir, n1, n2, n3
double precision u( n1, n2, n3 )

integer buff_id, indx

integer i3, i2, i1

buff_id = 3 + dir
indx = 0

if( axis .eq.  1 )then
   if( dir .eq. -1 )then

      do  i3=2,n3-1
         do  i2=2,n2-1
            indx = indx + 1
            u(n1,i2,i3) = buff(indx, buff_id )
         enddo
      enddo

   else if( dir .eq. +1 ) then

      do  i3=2,n3-1
         do  i2=2,n2-1
            indx = indx + 1
            u(1,i2,i3) = buff(indx, buff_id )
         enddo
      enddo

   endif
endif

if( axis .eq.  2 )then
   if( dir .eq. -1 )then

      do  i3=2,n3-1
         do  i1=1,n1
            indx = indx + 1
            u(i1,n2,i3) = buff(indx, buff_id )
         enddo
      enddo

   else if( dir .eq. +1 ) then

      do  i3=2,n3-1
         do  i1=1,n1
            indx = indx + 1
            u(i1,1,i3) = buff(indx, buff_id )
         enddo
      enddo

   endif
endif

if( axis .eq.  3 )then
   if( dir .eq. -1 )then

      do  i2=1,n2
         do  i1=1,n1
            indx = indx + 1
            u(i1,i2,n3) = buff(indx, buff_id )
         enddo
      enddo

   else if( dir .eq. +1 ) then

      do  i2=1,n2
         do  i1=1,n1
            indx = indx + 1
            u(i1,i2,1) = buff(indx, buff_id )
         enddo
      enddo

   endif
endif

return
end


subroutine comm1p( axis, u, n1, n2, n3, kk )
use caf_intrinsics


implicit none

include 'cafnpb.h'
include 'globals.h'

integer axis, dir, n1, n2, n3
double precision u( n1, n2, n3 )

integer i3, i2, i1, buff_len,buff_id
integer i, kk, indx

dir = -1

buff_id = 3 + dir
buff_len = nm2

do  i=1,nm2
   buff(i,buff_id) = 0.0D0
enddo


dir = +1

buff_id = 3 + dir
buff_len = nm2

do  i=1,nm2
   buff(i,buff_id) = 0.0D0
enddo

dir = +1

buff_id = 2 + dir
buff_len = 0

if( axis .eq.  1 )then
   do  i3=2,n3-1
      do  i2=2,n2-1
         buff_len = buff_len + 1
         buff(buff_len, buff_id ) = u( n1-1, i2,i3)
      enddo
   enddo
endif

if( axis .eq.  2 )then
   do  i3=2,n3-1
      do  i1=1,n1
         buff_len = buff_len + 1
         buff(buff_len,  buff_id )= u( i1,n2-1,i3)
      enddo
   enddo
endif

if( axis .eq.  3 )then
   do  i2=1,n2
      do  i1=1,n1
         buff_len = buff_len + 1
         buff(buff_len, buff_id ) = u( i1,i2,n3-1)
      enddo
   enddo
endif

dir = -1

buff_id = 2 + dir
buff_len = 0

if( axis .eq.  1 )then
   do  i3=2,n3-1
      do  i2=2,n2-1
         buff_len = buff_len + 1
         buff(buff_len,buff_id ) = u( 2,  i2,i3)
      enddo
   enddo
endif

if( axis .eq.  2 )then
   do  i3=2,n3-1
      do  i1=1,n1
         buff_len = buff_len + 1
         buff(buff_len, buff_id ) = u( i1,  2,i3)
      enddo
   enddo
endif

if( axis .eq.  3 )then
   do  i2=1,n2
      do  i1=1,n1
         buff_len = buff_len + 1
         buff(buff_len, buff_id ) = u( i1,i2,2)
      enddo
   enddo
endif

do  i=1,nm2
   buff(i,4) = buff(i,3)
   buff(i,2) = buff(i,1)
enddo

dir = -1

buff_id = 3 + dir
indx = 0

if( axis .eq.  1 )then
   do  i3=2,n3-1
      do  i2=2,n2-1
         indx = indx + 1
         u(n1,i2,i3) = buff(indx, buff_id )
      enddo
   enddo
endif

if( axis .eq.  2 )then
   do  i3=2,n3-1
      do  i1=1,n1
         indx = indx + 1
         u(i1,n2,i3) = buff(indx, buff_id )
      enddo
   enddo
endif

if( axis .eq.  3 )then
   do  i2=1,n2
      do  i1=1,n1
         indx = indx + 1
         u(i1,i2,n3) = buff(indx, buff_id )
      enddo
   enddo
endif


dir = +1

buff_id = 3 + dir
indx = 0

if( axis .eq.  1 )then
   do  i3=2,n3-1
      do  i2=2,n2-1
         indx = indx + 1
         u(1,i2,i3) = buff(indx, buff_id )
      enddo
   enddo
endif

if( axis .eq.  2 )then
   do  i3=2,n3-1
      do  i1=1,n1
         indx = indx + 1
         u(i1,1,i3) = buff(indx, buff_id )
      enddo
   enddo
endif

if( axis .eq.  3 )then
   do  i2=1,n2
      do  i1=1,n1
         indx = indx + 1
         u(i1,i2,1) = buff(indx, buff_id )
      enddo
   enddo
endif

return
end
```

CRAY



rprj3 in A-ZPL

```
procedure rprj3(var S,R: [,,] double;
                d: array [] of direction);
begin
  S := 0.5000 * R +
       0.2500 * (R@^d[ 1, 0, 0] + R@^d[ 0, 1, 0] + R@^d[ 0, 0, 1] +
                 R@^d[-1, 0, 0] + R@^d[ 0,-1, 0] + R@^d[ 0, 0,-1] +
       0.1250 * (R@^d[ 1, 1, 0] + R@^d[ 1, 0, 1] + R@^d[ 0, 1, 1] +
                 R@^d[ 1,-1, 0] + R@^d[ 1, 0,-1] + R@^d[ 0, 1,-1] +
                 R@^d[-1, 1, 0] + R@^d[-1, 0, 1] + R@^d[ 0,-1, 1] +
                 R@^d[-1,-1, 0] + R@^d[-1, 0,-1] + R@^d[ 0,-1,-1])+
       0.0625 * (R@^d[ 1, 1, 1] + R@^d[ 1, 1,-1] +
                 R@^d[ 1,-1, 1] + R@^d[ 1,-1,-1] +
                 R@^d[-1, 1, 1] + R@^d[-1, 1,-1] +
                 R@^d[-1,-1, 1] + R@^d[-1,-1,-1]);
end;
```

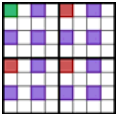


NAS MG Linecounts

1200
1000
800
600
400
200
0

comm/sync/dir
declarations
computation

MPI
Java
HPF
OpenMP
Serial
A-ZPL



MG Performance

MG Class C -- Cray T3E

Speedup over best 16-processor time
(114.607 seconds in A-ZPL)

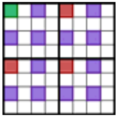
16
12
8
4
0

linear speedup
A-ZPL
ZPL
F77+MPI

0 16 32 64 128 256

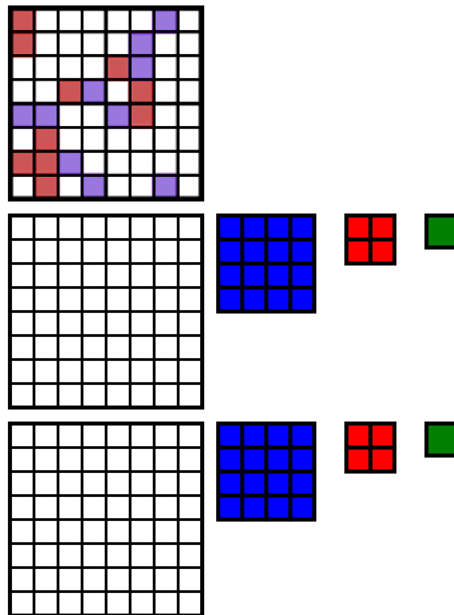
Processors

CRAY

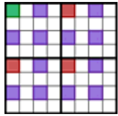


Sparse Arrays in MG

- MG's input array, V, has 20 non-zeroes regardless of problem size



- Computes resid against *V* twice per iteration
- Wasted time and space

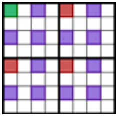


NAS MG Memory Usage

MG Class C -- memory usage

	U	R	V	RS
dense F77+MPI, ZPL	1.14	1.14	1.00	
A-ZPL	1.14	1.14		

CRAY



MG Performance

MG Class C -- Cray T3E

Speedup over best 16-processor time (114.607 seconds in A-ZPL)

16
12
8
4
0

linear speedup
A-ZPL
ZPL
F77+MPI

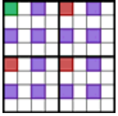
0 16 32 64 128 256

Processors

This uses the sparse input array

This one is dense

CRAY



Summary

- MG is a useful application
  - models real-world computations, parallel idioms
  - small enough to be easy to study
- Not so hard to describe in pictures and words
- Coding up a parallel version of it tends to be a different matter
- Surely we can do better than the current crop of parallel languages

CRAY