CSEP 524: Parallel Computation
(week 7)

Brad Chamberlain

Tuesdays 6:30 – 9:20

MGH 231

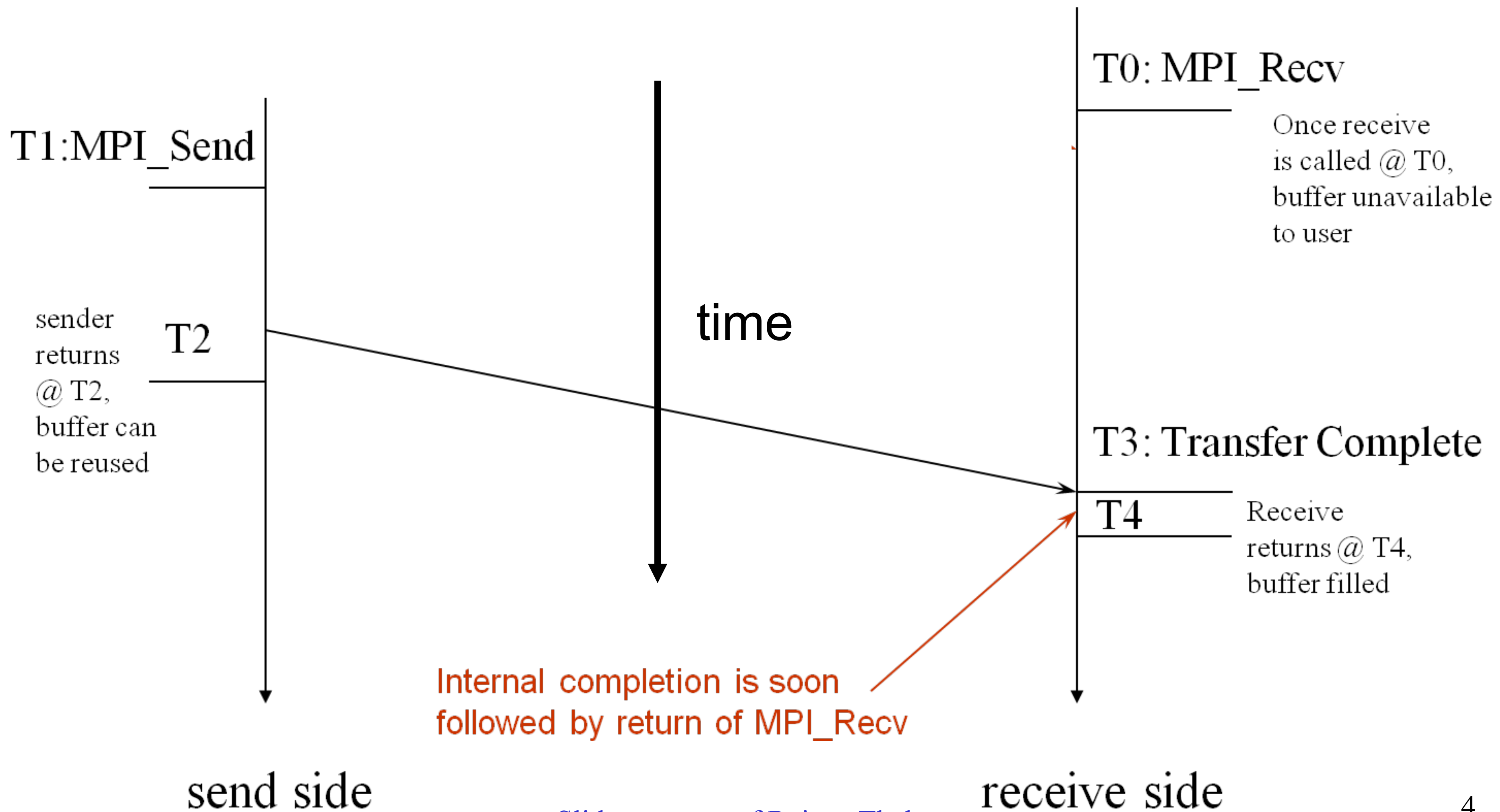
MPI Wrap-up

Primary MPI Concepts

1) Point-to-Point Communications (Sends/Receives)***:***

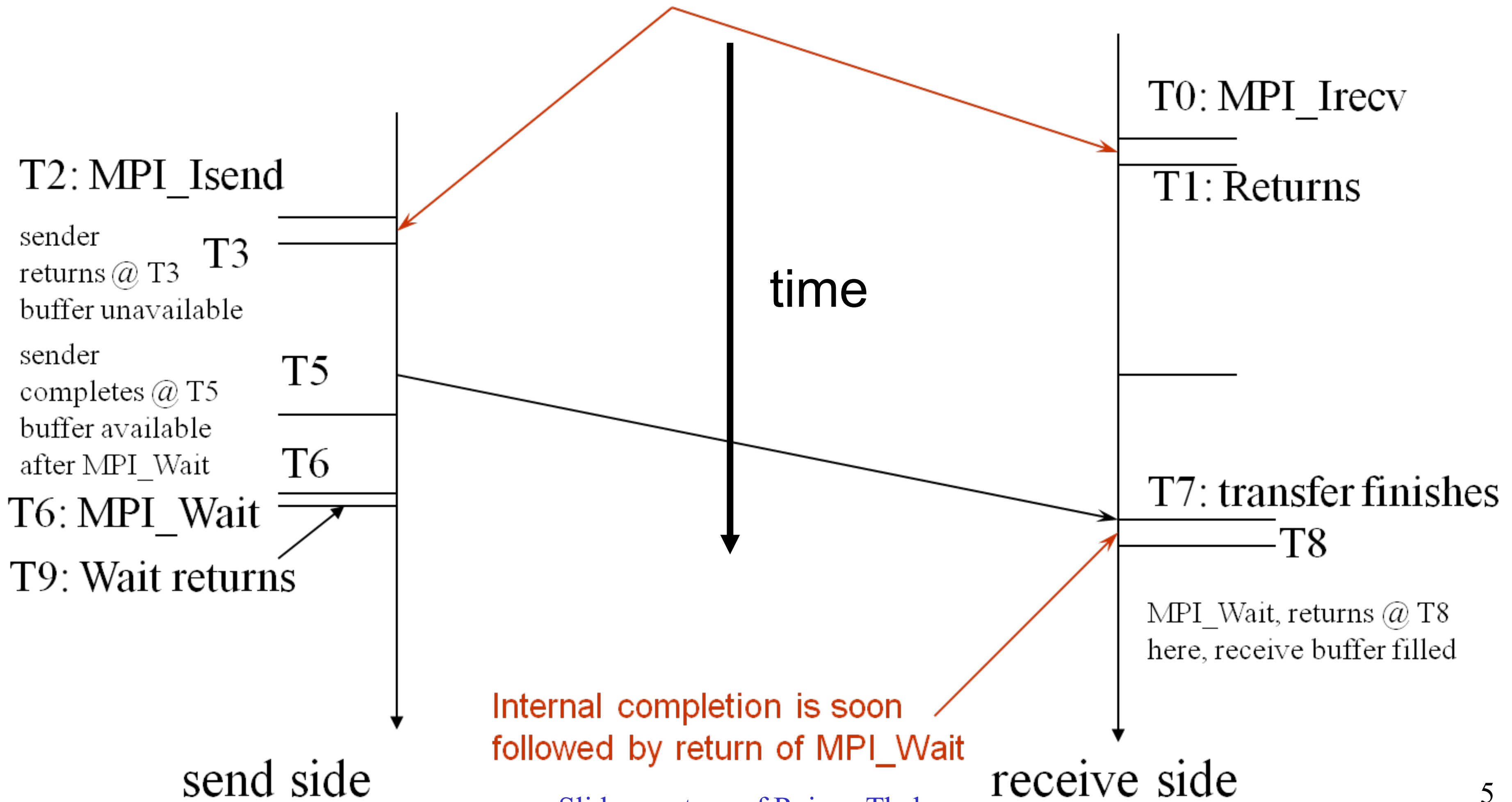
- primary building block for communication
- many different flavors
  - Send/Recv: vanilla
  - Isend/Irecv: non-blocking ("Immediate")
  - …

Blocking Send-Receive Diagram



Slide courtesy of Rajeev Thakur

Non-Blocking Send-Receive Diagram



Slide courtesy of Rajeev Thakur

Primary MPI Concepts

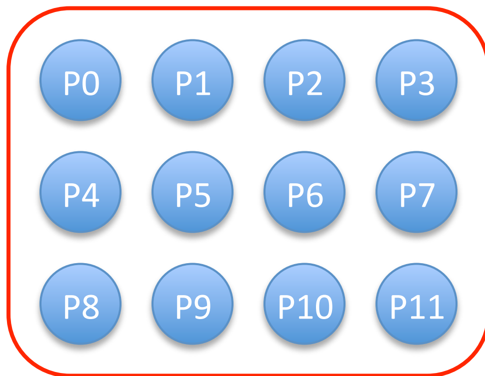
1) Point-to-Point Communications (Sends/Receives)***:***

- primary building block for communication
- many different flavors
  - Send/Recv: vanilla
  - Isend/Irecv ("Immediate"): non-blocking
  - Ssend ("Synchronous"): communication waits until recipient in recv call
  - Rsend ("Ready"): requires that the receive is guaranteed to be posted
  - Bsend ("Buffered"): send that provides its own buffer
  - Ibsend, Irsend, Issend: Non-blocking versions of the previous
  - SendRecv: does a send and a receive in one fell swoop
- various send types can be received by any recv type
  - e.g., Isend can match against recv; or send against Irecv

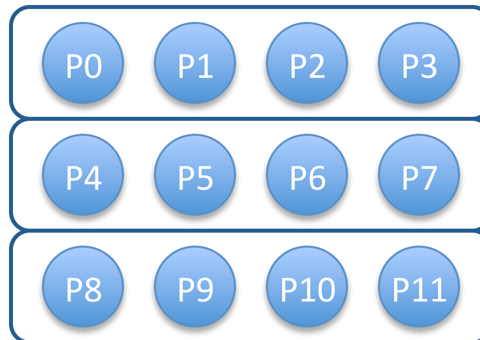
Primary MPI Concepts

2) Communicators (Process Groups)*:*

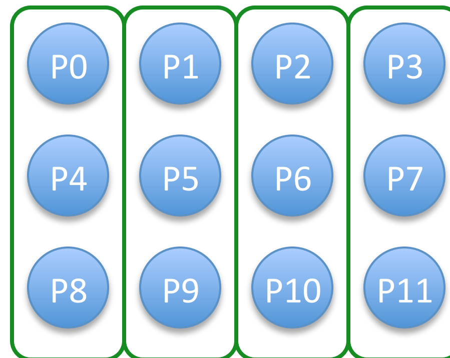
– Motivating example: 2D virtual process grid



MPI_COMM_WORLD



row communicators



column communicators

Useful for expressing…
…partial reductions
…partial scans
…partial broadcasts

Linear algebra algorithms often based on such operations

Primary MPI Concepts

2) Communicators (Process Groups)*:*

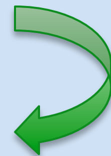
– Motivating example #2: Writing libraries using MPI

mylib.c:

```
void foo(…) {
  MPI_Send(…, dest=0, tag=1000, MPI_COMM_WORLD);
}

void bar(…) {
  MPI_Recv(…, src=1, tag=1000, MPI_COMM_WORLD);
}
```

myprog.c:

```
#include "mylib.h"

foo(…);
if (…) {
  MPI_Recv(…, src=myID+1, tag=1000, MPI_COMM_WORLD);
else {
  MPI_Send(…, dest=myID-1, tag=1000, MPI_COMM_WORLD);
}
bar(…);
```



Primary MPI Concepts

2) Communicators (Process Groups)*:*

– Motivating example #2: Writing libraries using MPI

mylib.c:

```
void foo(…) {
  MPI_Send(…, dest=0, tag=1000, MPI_COMM_WORLD);
}

void bar(…) {
  MPI_Recv(…, src=1, tag=1000, MPI_COMM_WORLD);
}
```

myprog.c:

```
#include "mylib.h"

foo(…);
if (…) {
  MPI_Recv(…, src=myID+1, tag=1000, MPI_COMM_WORLD);
else {
  MPI_Send(…, dest=myID-1, tag=1000, MPI_COMM_WORLD);
}
bar(…);
```

Primary MPI Concepts

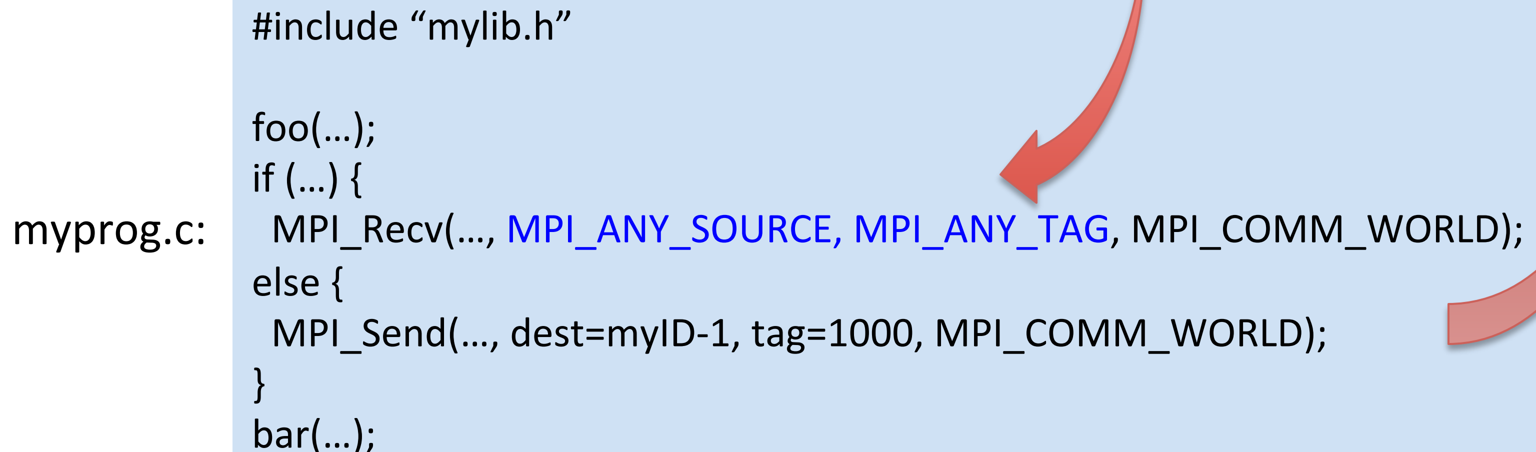
2) Communicators (Process Groups):

– Motivating example #2: Writing libraries using MPI

mylib.c:

```
void foo(…) {
  MPI_Send(…, dest=0, tag=1000, MPI_COMM_WORLD);
}

void bar(…) {
  MPI_Recv(…, src=1, tag=1000, MPI_COMM_WORLD);
}
```

myprog.c:



Primary MPI Concepts

2) Communicators (Process Groups):

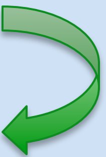
– Motivating example #2: Writing libraries using MPI

mylib.c:

```
void foo(…) {
  MPI_Send(…, dest=0, tag=1000, MPI_COMM_MYLIB);
}

void bar(…) {
  MPI_Recv(…, src=1, tag=1000, MPI_COMM_MYLIB);
}
```

myprog.c:

```
#include "mylib.h"

foo(…);
if (…) {
  MPI_Recv(…, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD);
else {
  MPI_Send(…, dest=myID-1, tag=1000, MPI_COMM_WORLD);
}
bar(…);
```



Primary MPI Concepts

3) Collectives (Communicator-based Operations):

- many different styles of communications/operations:
  - barrier
  - broadcast
  - scatter/gather
  - all-to-all
  - reduce
  - scan
- variations based on whether results go to one/all images
- variations in which messages can have uniform/variable sizes

Message Passing Hazards

- Main issues you're likely to run into:
  - mismatch between sends/receives
    - e.g., send doesn't have a matching receive or vice-versa
    - e.g., send and receive don't name right tag, source/destination
  - collectives in which participants are missing
    - e.g., a process never calls into a barrier or reduction
  - issues related to resource constraints/timing
    - e.g., insufficient memory to buffer things
    - (not likely to hit this in this class)

- These tend to manifest themselves like deadlocks
  - or as "out-of-resource" errors or degraded performance

Beyond MPI-1

MPI-2 (1990's):

– support for coordinated parallel I/O
– (poor support for) single-sided communication*
– dynamic process creation ("add a new MPI rank now")

MPI-3 (circa 2012):

– better support for single-sided communication
– better support for multithreading within MPI
– active messages*, **
– better support for GPUs/accelerators**
– better compiler support for MPI**

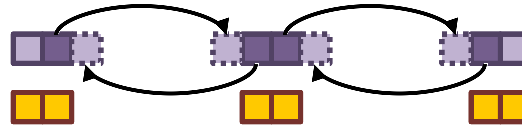
* = we'll be defining this term next week; ** = a work-in-progress?

The Stencil Ramp:
(a series of increasingly complex Stencil-based algorithms)

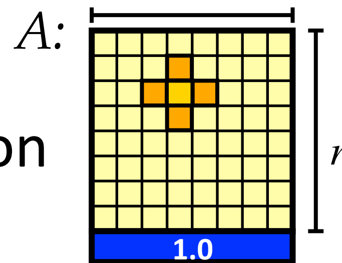
Stencils we have known

Stencils we have known (and loved!):

– The 3-point stencil



– The Jacobi iteration



– The 9-point stencil from homework

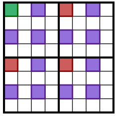
Some more advanced uses of stencils

– The Multigrid method (MG)

– The Fast Multipole Method (FMM)

A Distributed Memory Algorithm: The Multigrid Method

(as told through the NAS MG benchmark)



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

  *CFD = Computational Fluid Dynamics

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

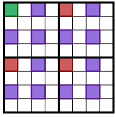
  …then as MPI reference implementations (v2.x)

  …then 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)

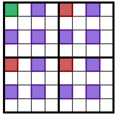
- 8 Benchmarks:
  - 5 kernels:
    - **EP**: embarrassingly parallel
    - **MG**: multigrid
    - **CG**: conjugate gradiant
    - **FT**: Fourier transform
    - **IS**: integer sort
  - 3 pseudo-applications
    - **BT:** block transpose
    - **LU:** LU factorization
    - **SP:** pentadiagonal

– 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/

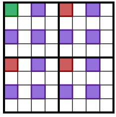
CRAY



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



MG’s arrays

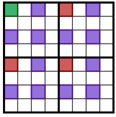
V:

input array

U:

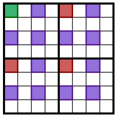
R:

hierarchical work arrays



Overview of MG

CRAY



MG's Timed Portion

repeat *nit* times

resid

resid

norm2u3

- *rnm2*
- *rnmu*

mg3p

Configurations

S: 32^3 (4 iterations)

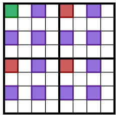
W: 64^3 (40 iterations)

A: 256^3 (4 iterations)

B: 256^3 (20 iterations)

C: 512^3 (20 iterations)

D: 1024^3 (50 iterations)



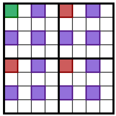
Hierarchical Arrays

	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)

Requires a distinct conceptual distribution per level

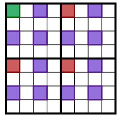
Admits a single conceptual distribution across levels

CRAY



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)	(2:8:2,2:8:2)	(4:8:4,4:8:4)	(8:8:8,8:8:8)



MG's Guts (*mg3P*)

rprj3

rprj3

rprj3

psinv

interp

resid

psinv

interp

resid

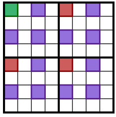
psinv

interp

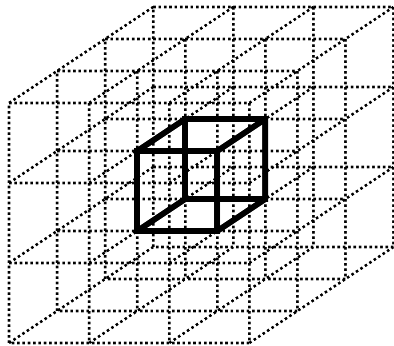
resid

psinv

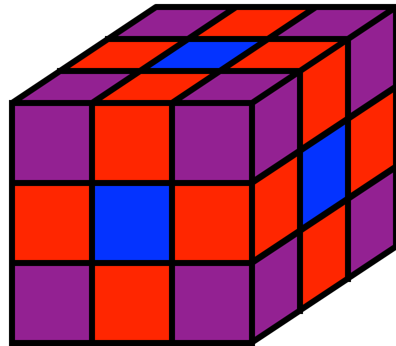
CRAY

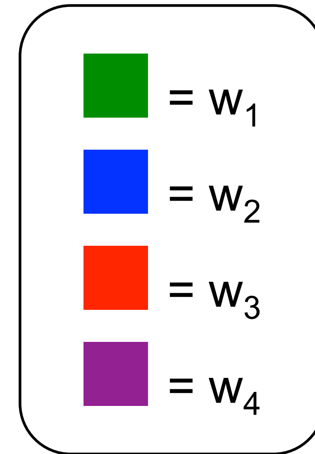


27-point stencils

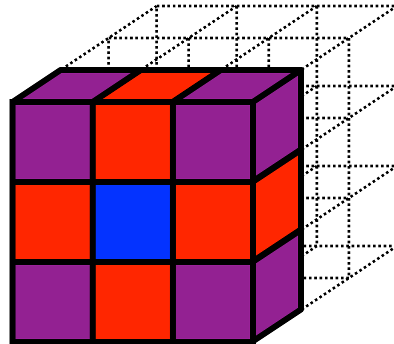


=

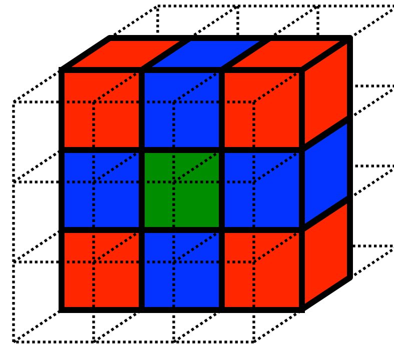




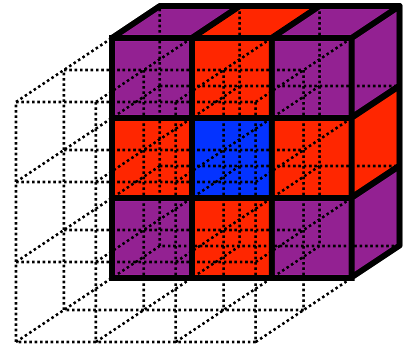
=



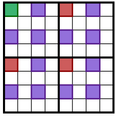
+



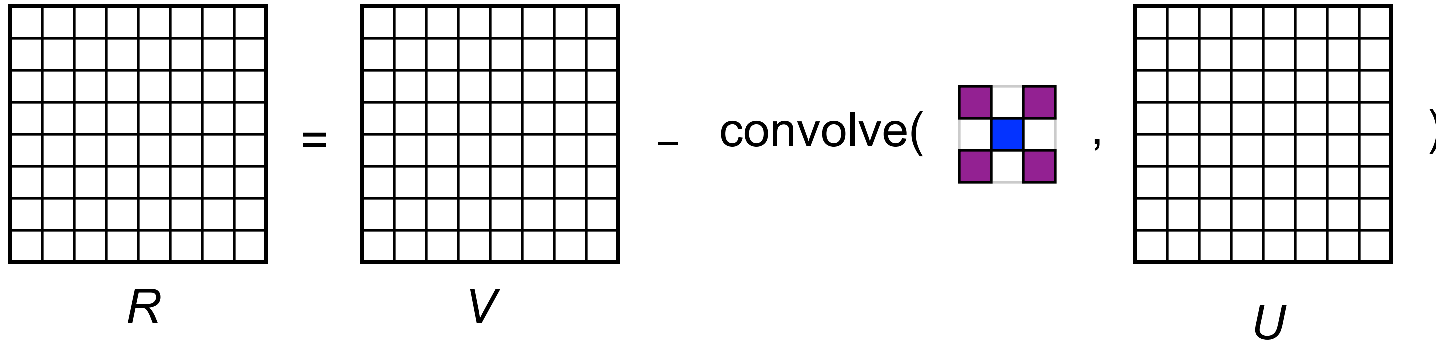
+

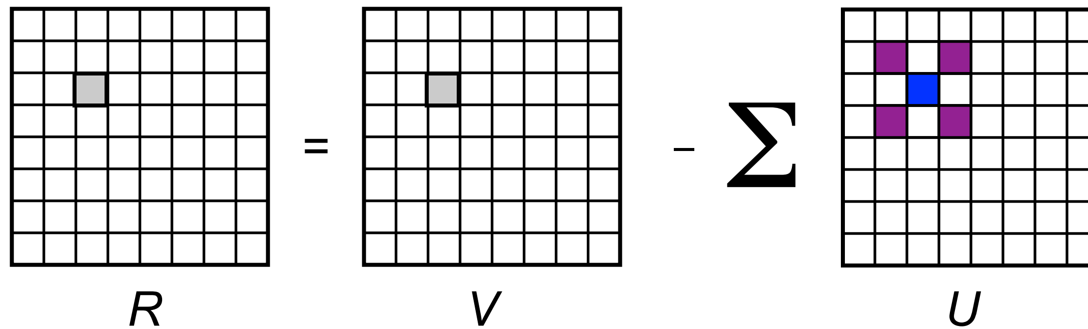


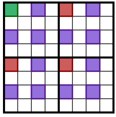
CRAY



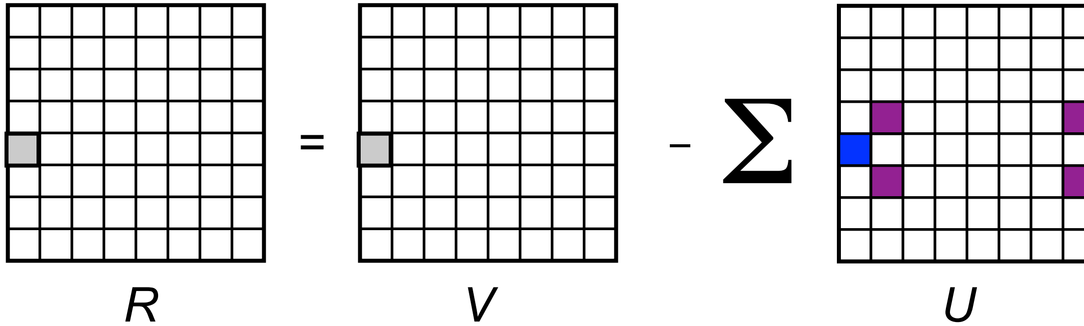
Our First Stencil: *resid*(R, V, U)

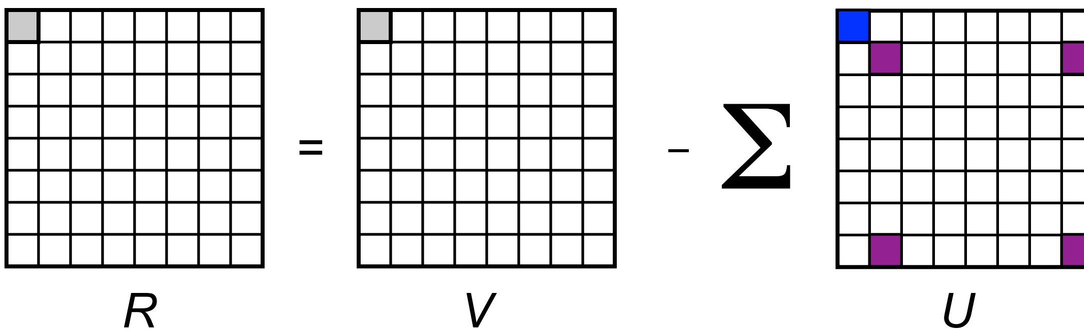


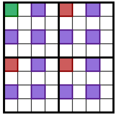




Periodic Boundary Conditions







At Other Levels of the Hierarchy

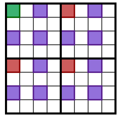
conceptually: =   – Σ

dense indexing: =   – Σ

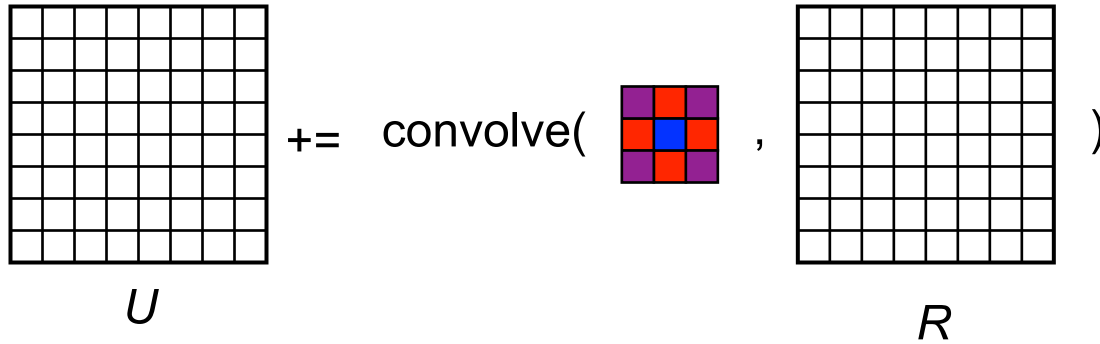
strided indexing: =   – Σ

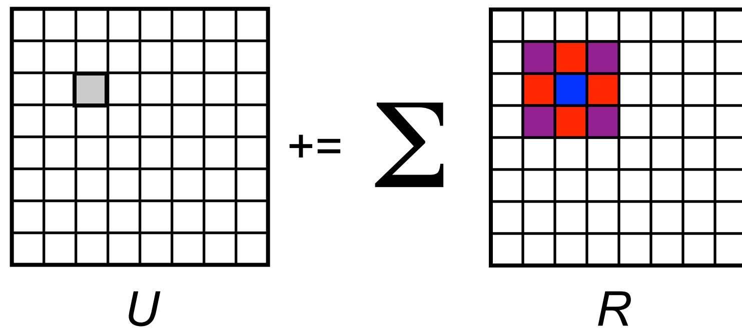
R   *V*   *U*

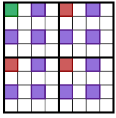
CRAY



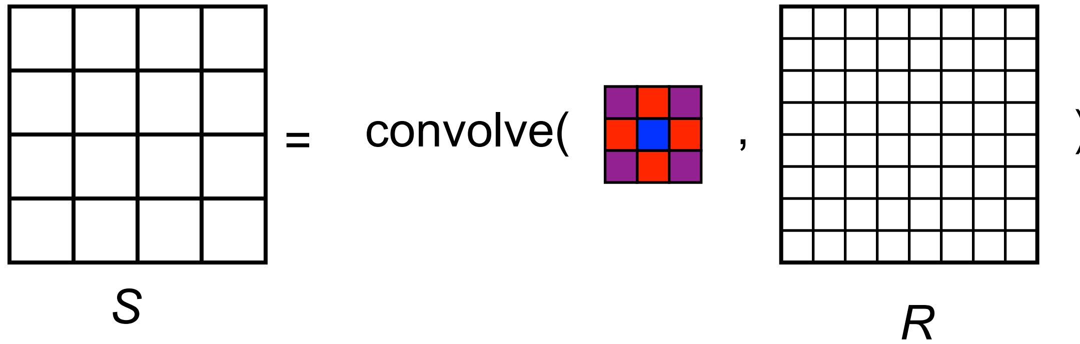
psinv(U, R)

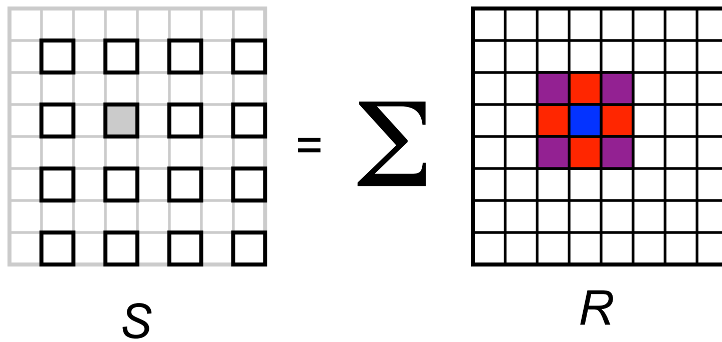




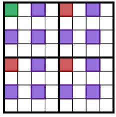


rprj3(S, R)

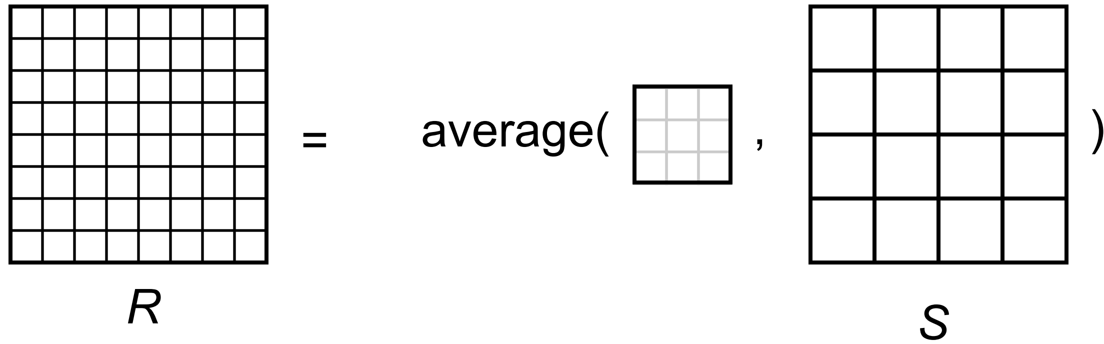


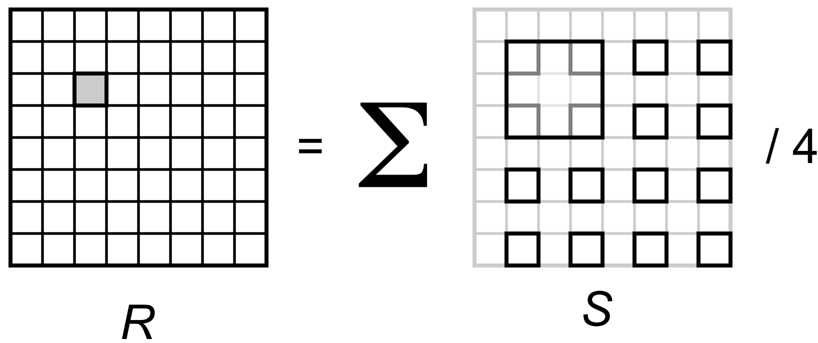


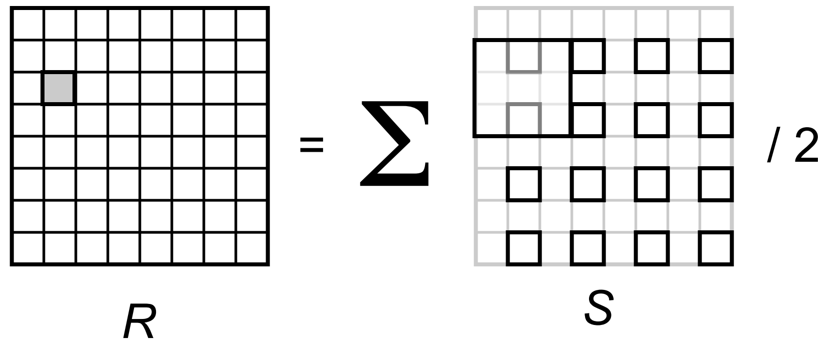
CRAY

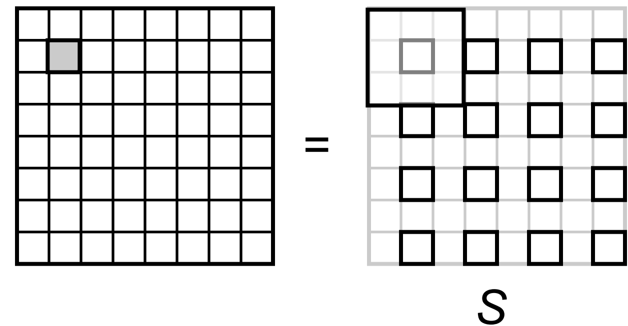


interp(R, S)

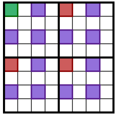






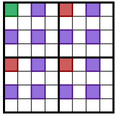


CRAY



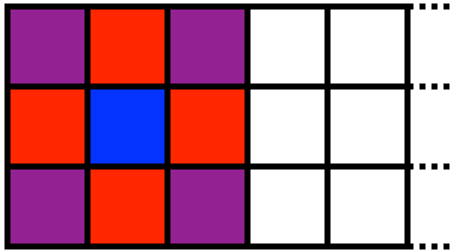
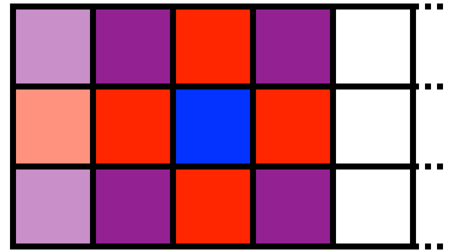
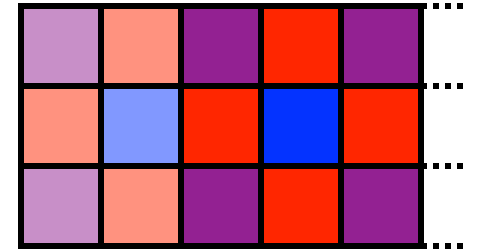
rprj3 in Fortran

```
  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
```

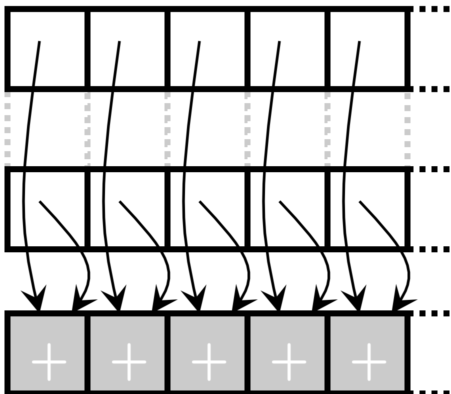
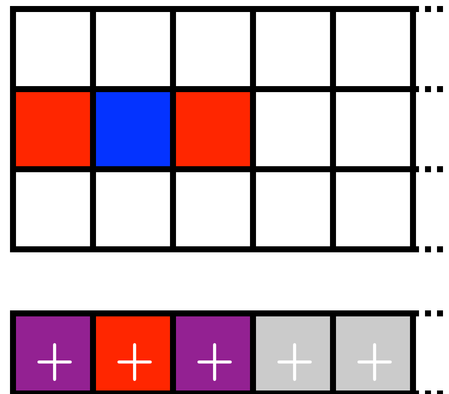
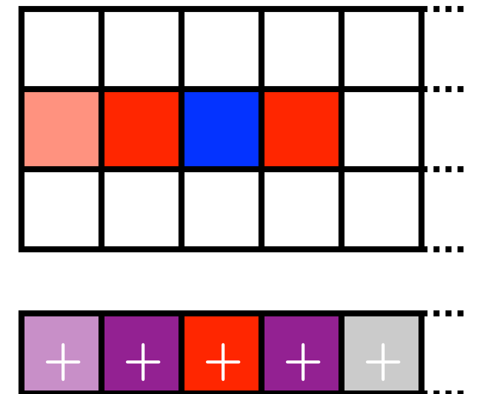


Stencil Optimization (2D)

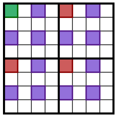
- Adjacent stencils use common subexpressions:

- Observation: Cache partial sums for reuse…

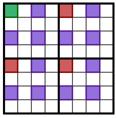




- Benefits are greater for 3D stencils…

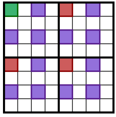


MG Stencil Optimization

CRAY



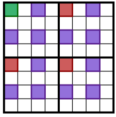
MG Stencil Optimization



rprj3 in Fortran with stencil opt.

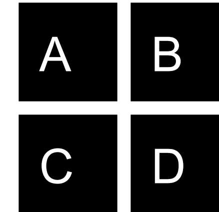
```
  do  j3=2,m3j-1
   i3 = 2*j3-d3
   do  j2=2,m2j-1
     i2 = 2*j2-d2
     do j1=2,m1j
      i1 = 2*j1-d1
      x1(i1-1) = r(i1-1,i2-1,i3  ) + r(i1-1,i2+1,i3  )
>                + r(i1-1,i2,  i3-1) + r(i1-1,i2,  i3+1)
      y1(i1-1) = r(i1-1,i2-1,i3-1) + r(i1-1,i2-1,i3+1)
>                + r(i1-1,i2+1,i3-1) + 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

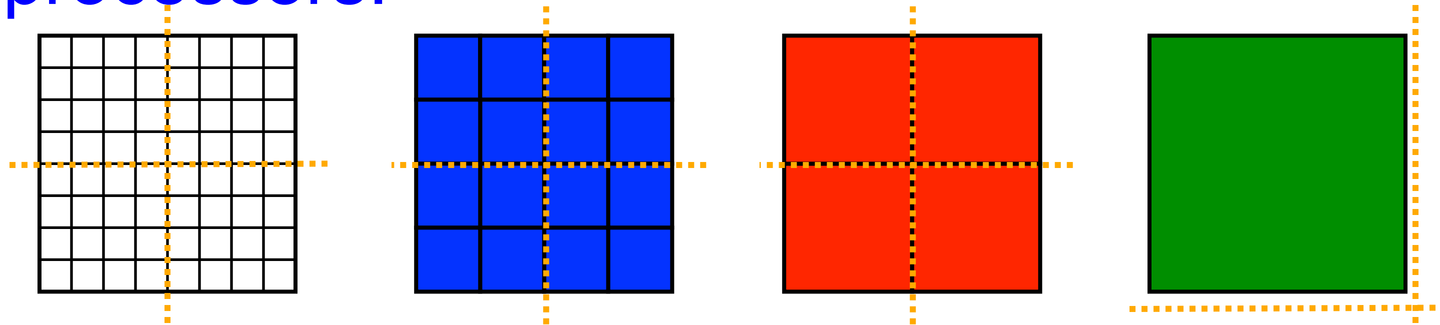


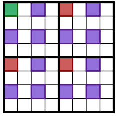
Parallel Data Distribution

- Given a virtual processor grid…



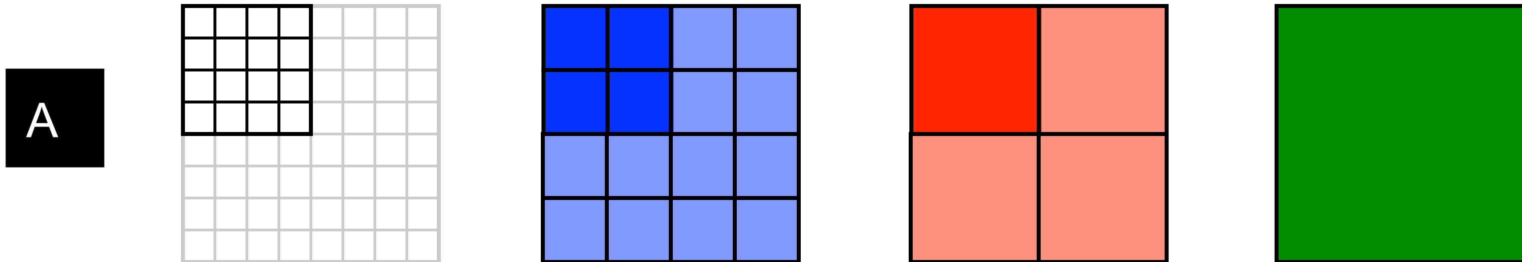
…arrays are block-distributed between processors:



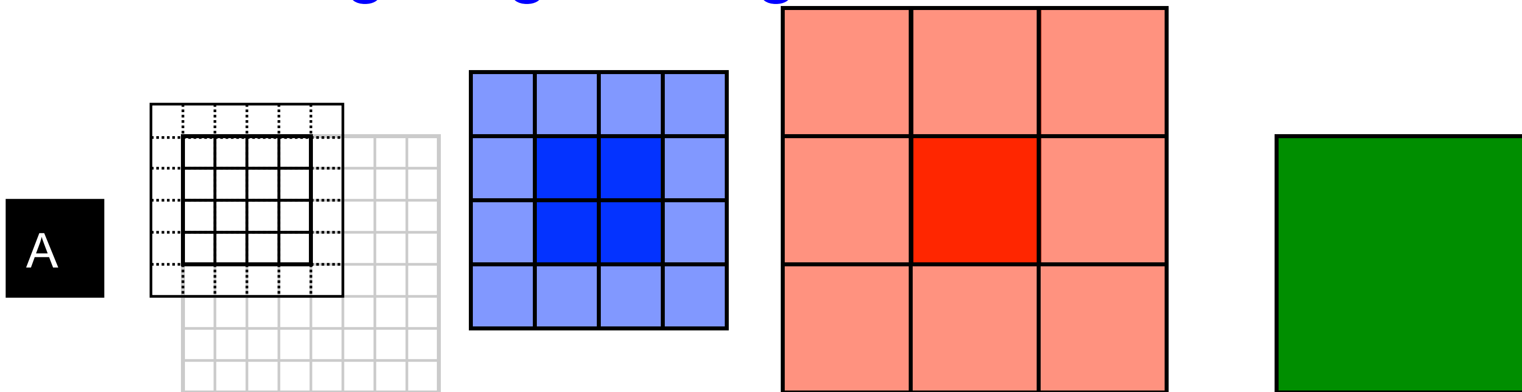


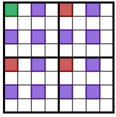
Per-processor Data Allocation

- In addition to its local block of values…



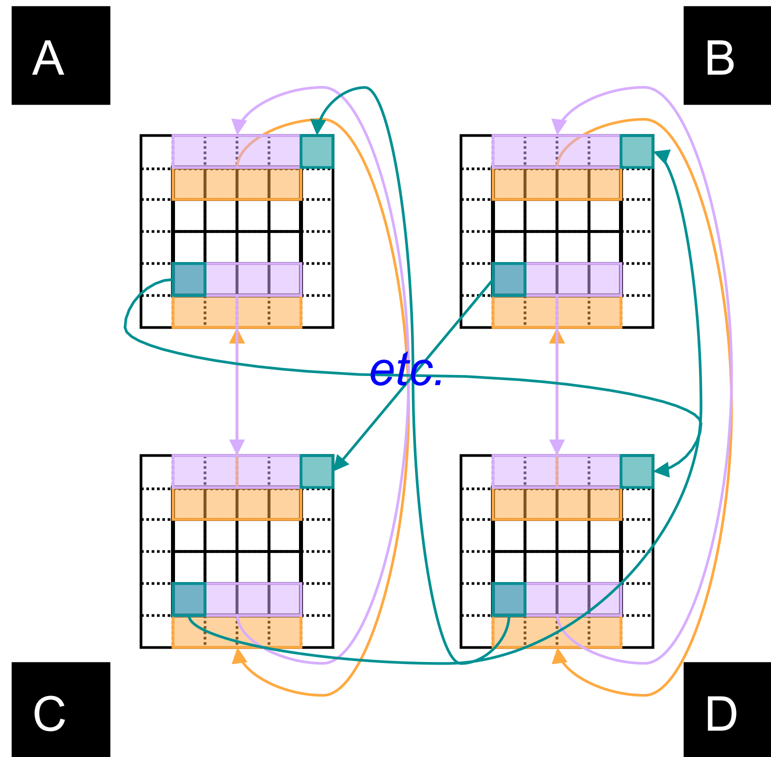
…each processor allocates ghost cells for caching neighboring values





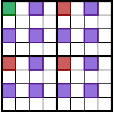
Stencil Communication

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



Notes:

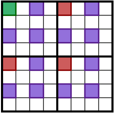
- Lots of optimization opportunities
- Have to eventually start skipping processors for coarser levels



Distributed *rprj3* in Fortran

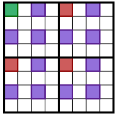
```
subroutine rprj3(r,m1k,m2k,m3k,s,m1j,m2j,m3j,k)
implicit none
include 'cafnpb.h'
include 'globals.h'

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
```

```
      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
```



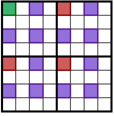
Fortran+MPI NAS MG *rprj3* stencil

```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

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

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
```



rprj3 in 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;
```



Code Size

1200
1000
800
600
400
200
0

Lines of Code

communication
declarations
computation

566
202
242

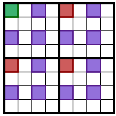
87
70

95
77

F+MPI
ZPL
A-ZPL

Language

CRAY



Code Size Notes

- the ZPL codes are 5.5–6.5x shorter because it supports a global view of parallelism rather than an SPMD programming model
  - ⇒ little/no code for communication
  - ⇒ little/no code for array bookkeeping

More important than the size difference is that it is easier to write, read, modify, and maintain

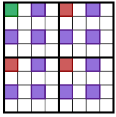
communication
declarations
computation

800
600
200
0
...es of Code

566
202
242
87
70
95
77

F+MPI
ZPL
A-ZPL

Language



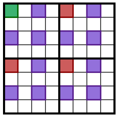
NAS MG Linecounts

1200
1000
800
600
400
200
0

comm/sync/dir
declarations
computation

MPI
Java
HPF
OpenMP
Serial
A-ZPL

CRAY



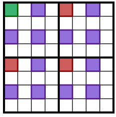
NAS MG Speedup: ZPL vs. Fortran + MPI

ZPL scales better than MPI since its communication is expressed in an implementation-neutral way; this permits the compiler to use SHMEM on this Cray T3E but MPI on a commodity cluster

ZPL also performs better at smaller scales where communication is not the bottleneck ⇒ new languages need not imply performance sacrifices

Similar observations—and more dramatic ones—have been made using more recent architectures, languages, and benchmarks

CRAY



Generality Notes

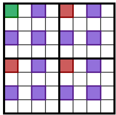
MG

Each ZPL binary supports:
- an arbitrary load-time problem size
- an arbitrary load-time # of processors
- 1D/2D/3D data decompositions

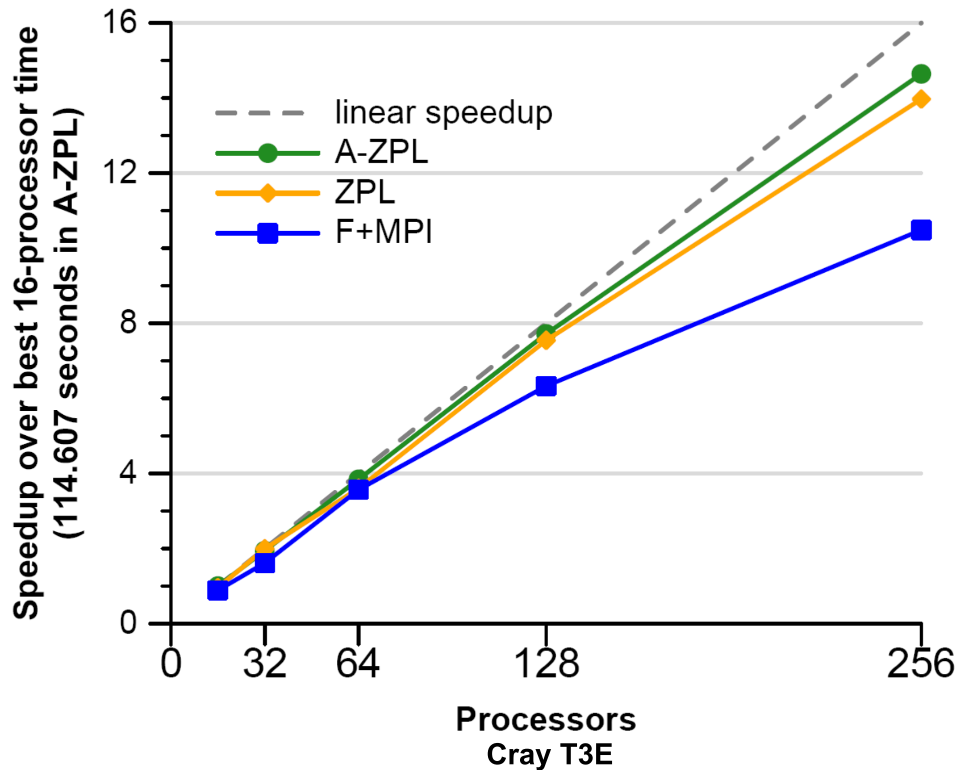
This MPI binary only supports:
- a static 2***k* problem size
- a static 2***j* # of processors
- a 3D data decomposition

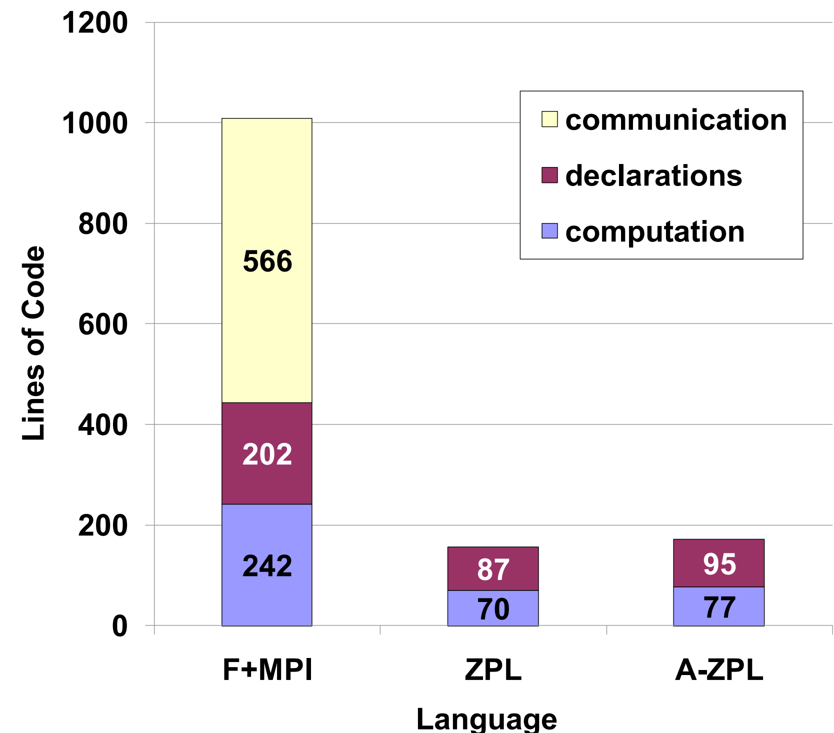
The code could be rewritten to relax these assumptions, but at what cost?
- in performance?
- in development effort?

CRAY



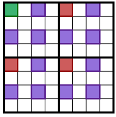
Global-view models can benefit Productivity





- more programmable, flexible
- able to achieve competitive performance
- more portable; leave low-level details to the compiler

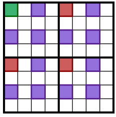
CRAY



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

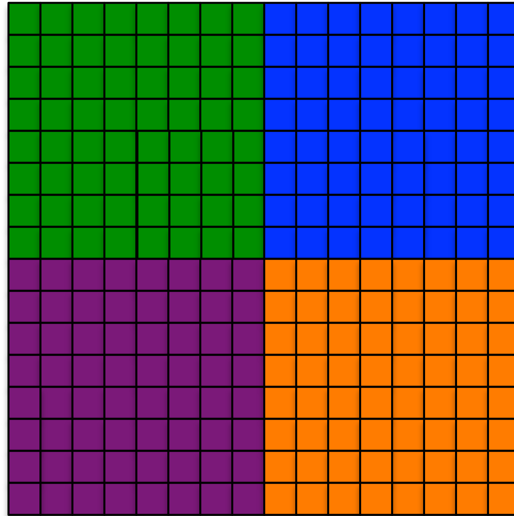
CRAY

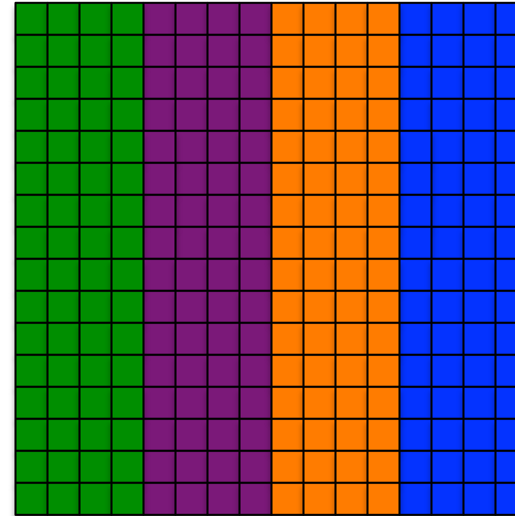


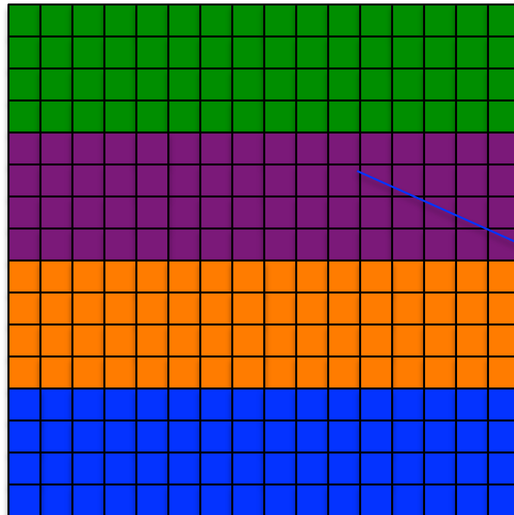
NAS MG: Parallel Implementation

- Arrays typically use block distributions
  - good load balance (computation is homogenous)
  - 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

Q: In a Shared-Memory setting, which would you use from the perspective of memory?







Reduces opportunity for false sharing

Q: In the setting of MG, which would you use?

Best surface to volume ratio
(good for stencil computations)

Abstract Machine Models

Abstract Machine Models

Abstract Machine Model: A simplified representation of the target architecture that is useful for programmers to think about

In sequential programming: the von Neumann architecture

- sequential processor
- flat memory
- …

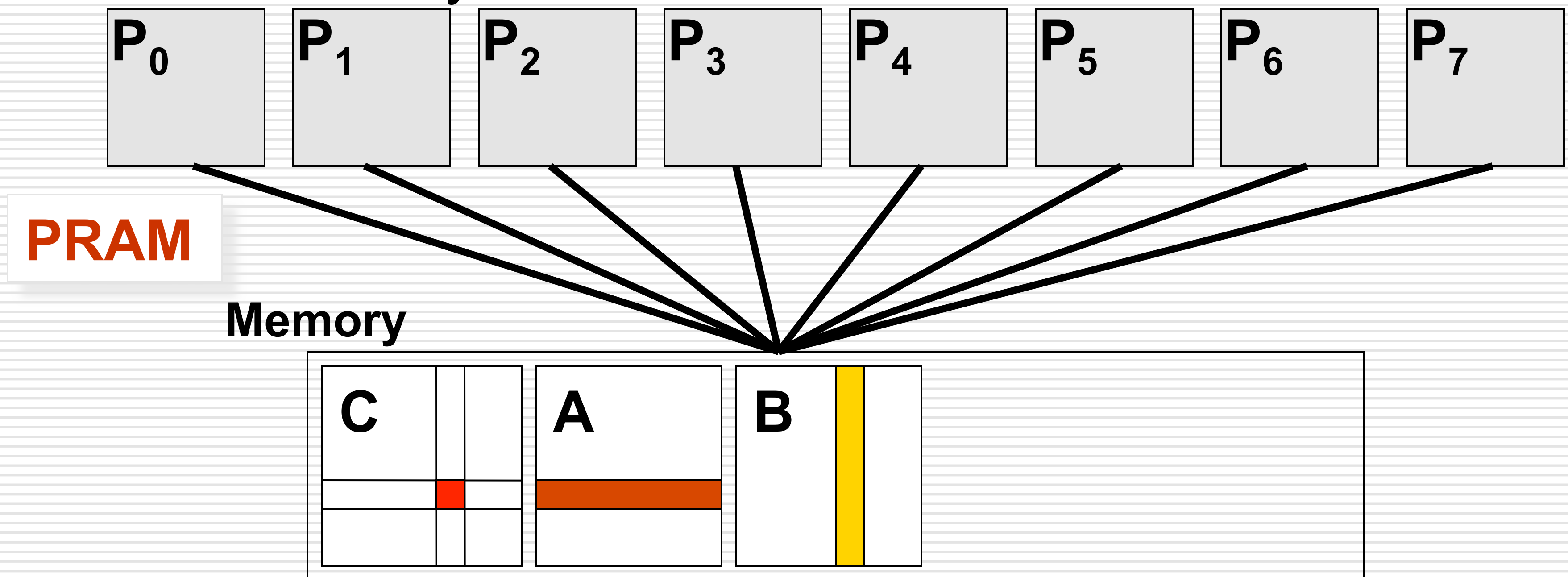
C serves as a good programming model for von Neumann

- arguably the reason that it serves as a portable assembly language of sorts

Recall Parallel Random-Access Machine

PRAM has any number of processors

- Every proc references any memory in "time 1"
- Memory read/write collisions must be resolved



SMPs implement PRAMs for small P … not scalable

Source: Larry Snyder, CSEP524, UW

PRAM Often Proposed As A Candidate

- PRAM (Parallel RAM) ignores memory organization, collisions, latency, conflicts, etc.
- Ignoring these are *claimed* to have benefits ...
  - Portable everywhere since it is very general
  - It is a simple programming model ignoring only insignificant details -- off by "only log P"
  - Ignoring memory difficulties is OK because hardware can "fake" a shared memory
  - Good for getting started: Begin with PRAM then refine the program to a practical solution if needed

Source: Larry Snyder, CSEP524, UW

Variations on PRAM

Resolving the memory conflicts considers read and write conflicts separately

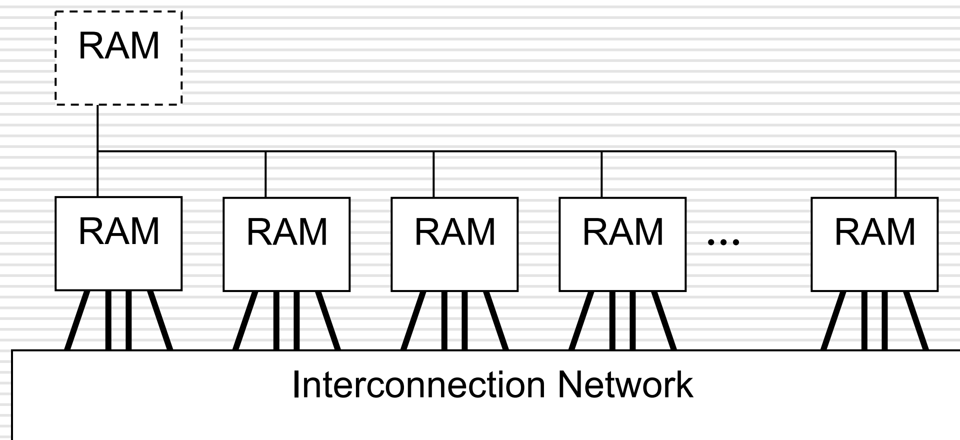
- Exclusive read/exclusive write (EREW)
  - The most limited model
- Concurrent read/exclusive write (CREW)
  - Multiple readers are OK
- Concurrent read/concurrent write (CRCW)
  - Various write-conflict resolutions used
- There are at least a dozen other variations

All theoretical -- not used in practice

Source: Larry Snyder, CSEP524, UW

CTA Model

- Candidate Type Architecture: A model with *P* standard processors, *d* degree, λ latency



- Node == processor + memory + NIC

Key Property: Local memory ref is 1, global memory is λ

Source: Larry Snyder, CSEP524, UW

What CTA Doesn't Describe

- CTA has no global memory … but memory could be globally *addressed*
- Mechanism for referencing memory not specified: shared, message passing, 1-side
- Interconnection network not specified
- λ is not specified beyond λ>>1 -- cannot be because every machine is different
- Controller, combining network "optional"

Source: Larry Snyder, CSEP524, UW

Discuss logP paper here

Closing note on logP

- Refinements have been proposed over time
  - "Oh, if we also measured *xyz*, the model would be better!"
  - e.g., logGP: takes "long messages" into account
  - Challenge: when to stop?

Abstract Machine Model Summary: My Take

PRAM: Too abstract/unrealistic!

logP: Too parameterized (?)

– or, perhaps most appropriate for low-level library writer

CTA: The goldilocks solution?

By analogy: Consider sequential programming in a cache-aware manner without worrying about an architecture's precise…

…latencies to access different levels of cache/memory

…cache lines sizes

…etc.

It can yield a significant fraction of peak performance, while remaining quite portable

Partitioned Global Address Space (PGAS) Programming Models

Partitioned Global Address Space Languages

(Or perhaps: partitioned global namespace languages)

- abstract concept:
  - support a shared namespace on distributed memory
    - permit any parallel task to access any lexically visible variable
    - doesn't matter if it's local or remote
  - establish a strong sense of ownership
    - every variable has a well-defined location
    - local variables are cheaper to access than remote ones

The commercial: *"Your shared memory convenience is in my distributed memory locality model!"*

"Mmmmmmmm"

Traditional PGAS Languages

- Founding fathers: UPC, Co-Array Fortran, Titanium
  - extensions to C, Fortran, and Java, respectively
  - details vary, but potential for:
    - arrays that are decomposed across nodes
    - pointers that refer to remote objects
  - note that earlier languages could also be considered PGAS, but that the term didn't exist yet

- If I had a week to spare, we would spend some time on these first before getting to Chapel
  - instead, we'll do Chapel this week and come back to CAF and UPC next week

Distributed Memory Chapel (ahem… “multi-locale Chapel”)

(switch to other slide decks)

Using Chapel for distributed memory

- Primary change:
  - **CHPL_COMM=none (or unset):** use shared memory
  - **CHPL_COMM=gasnet:** use distributed memory
- Depending on your platform, you may also need to tell GASNet how to launch an SPMD program
  - Brandon has prepared READMEs for the VM and UW cluster
- One snafu: The pre-built Chapel I gave you embedded my paths into your build – so you'll need to rebuild it
  - Sorry… though GASNet gets the blame

Smith-Waterman Algorithm for Sequence Alignment

Smith-Waterman

Goal: Determine the similarities/differences between two protein sequences/nucleotides.

– e.g., ACACACTA and AGCACACA*

Basis of Computation: Defined via a recursive formula:

$$H(i,0) = 0$$

$$H(0,j) = 0$$

$$H(i,j) = f(H(i\text{-}1, j\text{-}1), H(i\text{-}1, j), H(i, j\text{-}1))$$

Caveat: This is a classic, rather than cutting-edge sequence alignment algorithm, but it illustrates an important parallel paradiagm: wavefront computation

*Source of running example: Wikipedia

Smith-Waterman

Naïve Task-Parallel Approach:

```
proc computeH(i,j) {
  if (i==0 || j == 0) then
    return 0;
  else
    var h1, h2, h3: int;

    begin h1 = computeH(i-1, j-1);
    begin h2 = computeH(i-1, j);
    begin h3 = computeH(i, j-1);

    return f(h1,h2,h3);
}
```

Note: Recomputes most subexpressions redundantly

This is a case for dynamic programming!

Smith-Waterman

Dynamic Programming Approach:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0								
2	0								
3	0								
4	0								
5	0								
6	0								
7	0								
8	0								

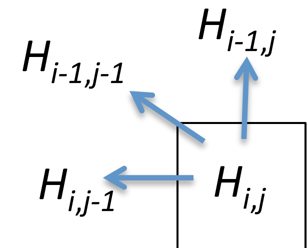
Step 1: Initialize boundaries to 0

Smith-Waterman

Dynamic Programming Approach:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0								
2	0								
3	0								
4	0				etc.				
5	0								
6	0								
7	0								
8	0								

Step 2: Compute cells as we're able to



Smith-Waterman

Dynamic Programming Approach:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	2	1	2	1	2	1	0	2
2	0	1	1	1	1	1	1	0	1
3	0	0	3	2	3	2	3	2	1
4	0	2	2	5	4	5	4	3	4
5	0	1	4	4	7	6	7	6	5
6	0	2	3	6	6	9	8	7	8
7	0	1	4	5	8	8	11	10	9
8	0	2	3	6	7	10	10	10	12

Step 3: Follow trail of breadcrumbs back

Smith-Waterman

Dynamic Programming Approach:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	2	1	2	1	2	1	0	2
2	0	1	1	1	1	1	1	0	1
3	0	0	3	2	3	2	3	2	1
4	0	2	2	5	4	5	4	3	4
5	0	1	4	4	7	6	7	6	5
6	0	2	3	6	6	9	8	7	8
7	0	1	4	5	8	8	11	10	9
8	0	2	3	6	7	10	10	10	12

Step 3: Follow trail of breadcrumbs back

Smith-Waterman

Dynamic Programming Approach:

		A	C	A	C	A	C	T	A
	0	0	0	0	0	0	0	0	0
A	0	2	1	2	1	2	1	0	2
G	0	1	1	1	1	1	1	0	1
C	0	0	3	2	3	2	3	2	1
A	0	2	2	5	4	5	4	3	4
C	0	1	4	4	7	6	7	6	5
A	0	2	3	6	6	9	8	7	8
C	0	1	4	5	8	8	11	10	9
A	0	2	3	6	7	10	10	10	12

Step 4: Interpret the path against the original sequences

```
AGCACAC-A
A-CACACTA
```

How could we do this in parallel?

Smith-Waterman

Data-Parallel Approach:

```
proc computeH(H: [0..n, 0..n] int) {
  for upperDiag in 1..n do
    forall diagPos in 0..#upperDiag {
      const (i,j) = [diagPos+1, upperDiag-diagPos];
      H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    }
  for lowerDiag in 1..n-1 do
    forall diagPos in lowerDiag..n-1 by -1 {
      const (i,j) = [diagPos+1, lowerDiag+diagPos];
      H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    }
}
```

Loop over diagonals serially

Traverse each diagonal in parallel

Advantages:

- Reasonably clean (if I got my indexing correct)

Disadvantages:

- Not so great in terms of cache use
- A bit fine-grained
- Not ideal for distributed memory

Smith-Waterman

Naïve Data-Driven Task-Parallel Approach:

```
proc computeH(H: [0..n, 0..n] int) {
  const ProbSpace = H.domain.translate(1,1);
  var NeighborsDone: [ProbSpace] atomic int = 0;
  var Ready$: [ProbSpace] sync int;
  NeighborsDone[1, ..].add(1);
  NeighborsDone[.., 1].add(1);
  NeighborsDone[1, 1].add(1);
  Ready$[1,1] = 1;
  coforall (i,j) in ProbSpace {
    const goNow = Ready$[i,j];
    H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const eastReady = NeighborsDone[i,j+1].fetchAdd(1);
    const seReady = NeighborsDone[i+1,j+1].fetchAdd(1);
    const southReady = NeighborsDone[i+1,j].fetchAdd(1);
    if (eastReady == 2) then Ready$[i,j+1] = 1;
    if (seReady == 2) then Ready$[i+1,j+1] = 1;
    if (southReady == 2) then Ready$[i+1,j] = 1;
  }
}
```

Create domain describing shifted version off H's domain

Arrays to count how many of our 3 neighbors are done; and to signal when we can compute

Set up boundaries: north and west elements have a neighbor done; top-left is ready

Create a task per matrix element and have it block until ready

Compute our element

Increment our neighbors' counts

Signal our neighbors as ready if we're the third

Smith-Waterman

Naïve Data-Driven Task-Parallel Approach:

```
proc computeH(H: [0..n, 0..n] int) {
  const ProbSpace = H.domain.translate(1,1);
  var NeighborsDone: [ProbSpace] atomic int = 0;
  var Ready$: [ProbSpace] sync int;
  NeighborsDone[1, ..].add(1);
  NeighborsDone[.., 1].add(1);
  NeighborsDone[1, 1].add(1);
  Ready$[1,1] = 1;
  coforall (i,j) in ProbSpace {
    const goNow = Ready$[i,j];
    H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const eastReady = NeighborsDone[i,j+1].fetchAdd(1);
    const seReady = NeighborsDone[i+1,j+1].fetchAdd(1);
    const southReady = NeighborsDone[i+1,j].fetchAdd(1);
    if (eastReady == 2) then Ready$[i,j+1] = 1;
    if (seReady == 2) then Ready$[i+1,j+1] = 1;
    if (southReady == 2) then Ready$[i+1,j] = 1;
  }
}
```

Disadvantages:

- Still not great in cache use
- Uses n^2 tasks
- Most spend most of their time blocking

Smith-Waterman

Slightly Less Naïve Data-Driven Task-Parallel Approach:

```
proc computeH(H: [0..n, 0..n] int) {
  const ProbSpace = H.domain.translate(1,1);
  var NeighborsDone: [ProbSpace] atomic int = 0;
  NeighborsDone[1, ..].add(1);
  NeighborsDone[.., 1].add(1);
  NeighborsDone[1, 1].add(1);
  sync { computeHHelp(1,1); }

  proc computeHHelp(i,j) {
    H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const eastReady = NeighborsDone[i,j+1].fetchAdd(1);
    const seReady = NeighborsDone[i+1,j+1].fetchAdd(1);
    const southReady = NeighborsDone[i+1,j].fetchAdd(1);
    if (eastReady == 2) then begin computeHHelp(i,j+1);
    if (seReady == 2) then begin computeHHelp(i+1,j+1);
    if (southReady == 2) then begin computeHHelp(i+1,j);
  }
}
```

Rather than create the tasks *a priori*, fire them off once we know they're legal

sync to ensure they're all done before we go on

Smith-Waterman

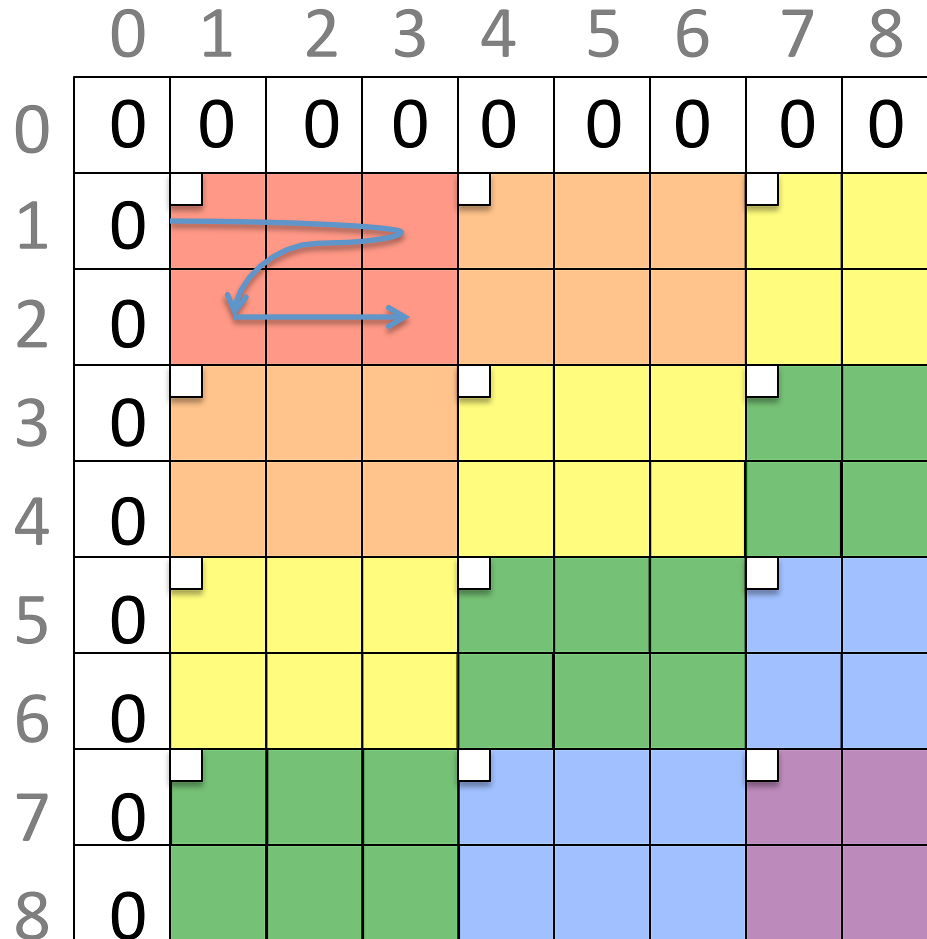
Slightly Less Naïve Data-Driven Task-Parallel Approach:

```
proc computeH(H: [0..n, 0..n] int) {
  const ProbSpace = H.domain.translate(1,1);
  var NeighborsDone: [ProbSpace] atomic int = 0;
  NeighborsDone[1, ..].add(1);
  NeighborsDone[.., 1].add(1);
  NeighborsDone[1, 1].add(1);
  sync { computeHHelp(1,1); }

  proc computeHHelp(i,j) {
    H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const eastReady = NeighborsDone[i,j+1].fetchAdd(1);
    const seReady = NeighborsDone[i+1,j+1].fetchAdd(1);
    const southReady = NeighborsDone[i+1,j].fetchAdd(1);
    if (eastReady == 2) then begin computeHHelp(i,j+1);
    if (seReady == 2) then begin computeHHelp(i+1,j+1);
    if (southReady == 2) then begin computeHHelp(i+1,j);
  }
}
```

Disadvantages:
- Still uses a lot of tasks
- Each task is very fine-grained

Smith-Waterman

Coarsening the Parallelism:



Smith-Waterman

Stride indices to get to next chunk

Blocked Data-Driven Task-Parallel Approach:

```
proc computeH(H: [0..n, 0..n] int) {
  const ProbSpace = H.domain.translate(1,1) by (rowsPerChunk, colsPerChunk);
  var NeighborsDone: [ProbSpace] atomic int = 0;
  NeighborsDone[1, ..].add(1);
  NeighborsDone[.., 1].add(1);
  NeighborsDone[1, 1].add(1);
  sync { computeHHelp({1..rowsPerChunk,1..colsPerChunk}); }

  proc computeHHelp(inds) {
    for (i,j) in H.domain[inds] do
      H[i,j] = f(H[i-1,j-1], H[i-1,j], H[i,j-1]);
    const (i,j) = inds.low;
    const eastReady = NeighborsDone[i,j+colsPerChunk].fetchAdd(1);
    const seReady = NeighborsDone[i+rowsPerChunk,j+colsPerChunk].fetchAdd(1);
    const southReady = NeighborsDone[i+rowsPerChunk,j].fetchAdd(1);
    if (eastReady == 2) then begin computeHHelp(i,j+colsPerChunk);
    if (seReady == 2) then begin computeHHelp(i+rowsPerChunk,j+colsPerChunk);
    if (southReady == 2) then begin computeHHelp(i+rowsPerChunk,j);
  }
}
```

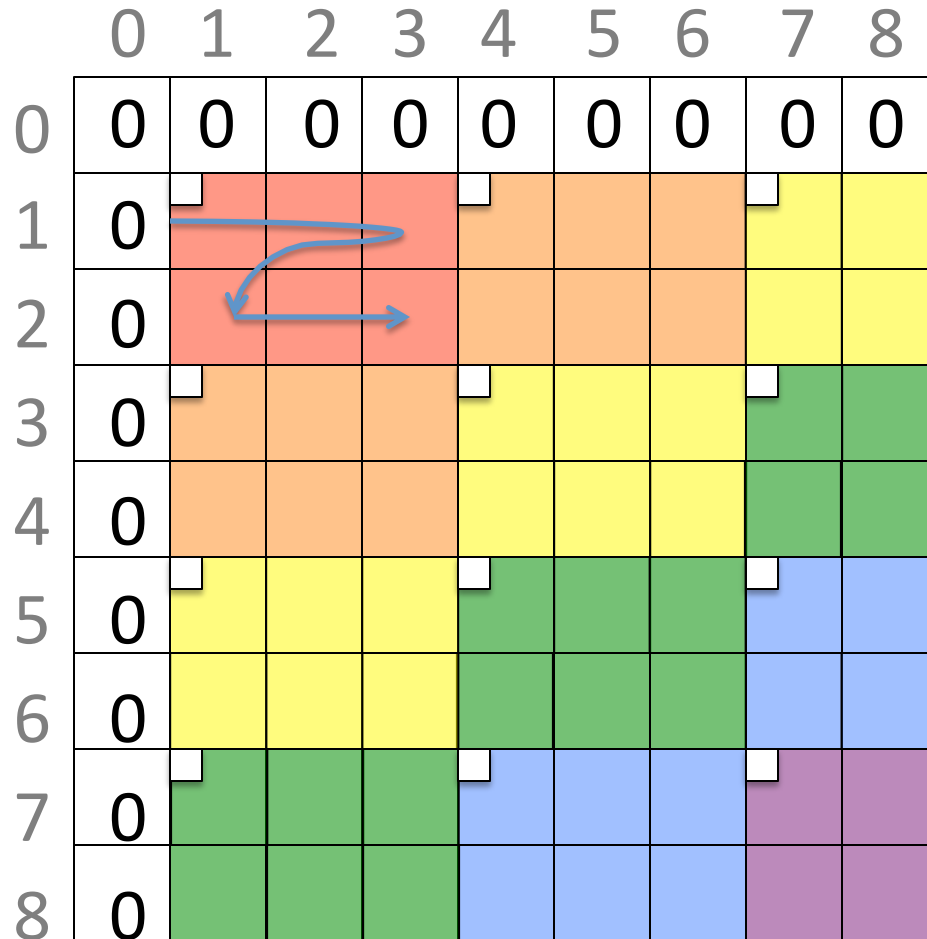
Can now use strided array for atomics

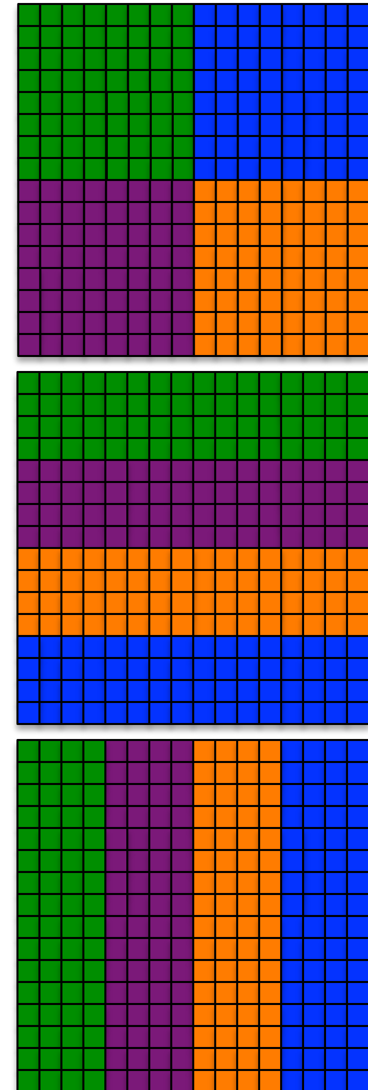
Change helper to take a domain describing the chunk to compute

Compute over chunk serially

Smith-Waterman

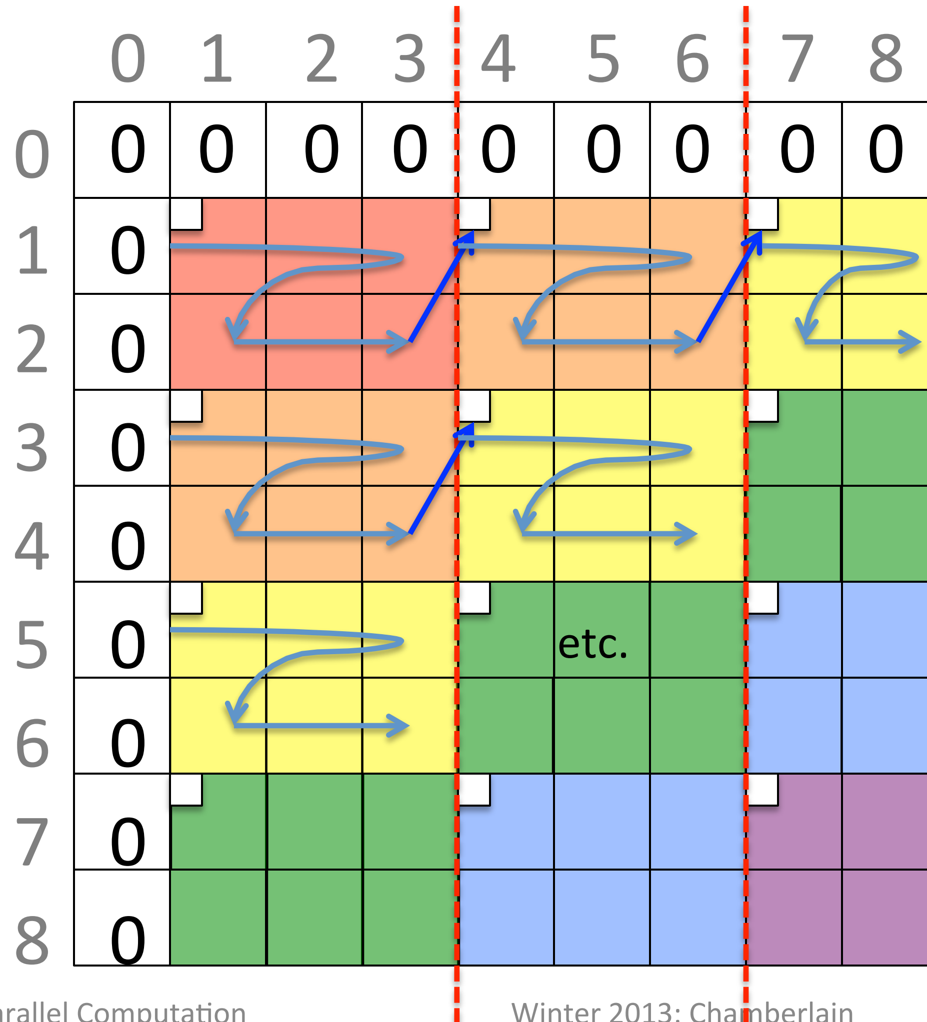
Now, what about distributed memory?





Smith-Waterman

Now, what about distributed memory?



Advantages:

- Good cache behavior: Nice fat blocks of data touchable in memory order
- Pipeline parallelism: Good utilization once pipeline is filled