Parallel Programming on the Origin

Outline

Overview of shared-memory parallel directives

Performance tuning: exploiting cache and memory locality

- cache false sharing
- data placement

Implementation and optimizations

Performance

Overview of Parallel Directives

```
c$doacross local(i), shared(a)
do i = 1,n
   a(i) = •••
enddo
```

Parallel Region Directive

```
c$par parallel local(a,i,j), shared(b)
  do i = 1,n
     a(i) = •••
  enddo

c$pdo
  do j = 1,n
     do i = 1,n
        b(j) = b(j) + a(i)
     enddo
  enddo
c$par end parallel
```

Shared memory parallelism is NOT just fine-grain and loop-level

Synchronization Directives

```
c$par critical
   s = s + 1
c$par end critical

c$par barrier

call synchronize
```

Atomic Operations

```
y = __fetch_and_add (&x, 1);

while (__lock_test_and_set(&x,1) != 0);
```

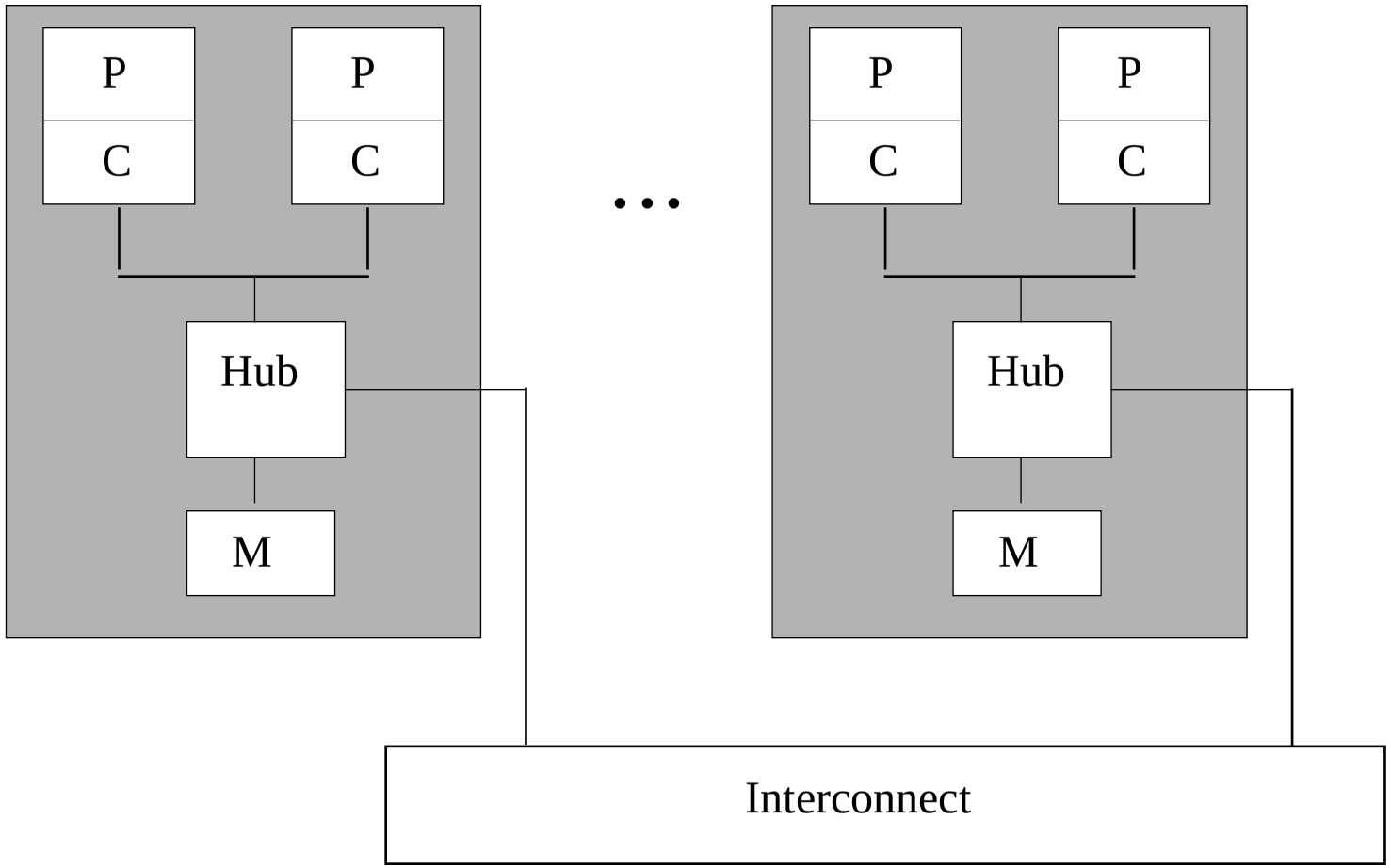
OpenMP

Effort to standardize shared-memory directives across all platforms

- Goal: **portability** across machines from different vendors
- includes directives for Fortran 77, Fortran 90, C, and C++
- includes vendors such as DEC, IBM, KAI

Caution: OpenMP is NOT public yet

Origin Architecture Overview



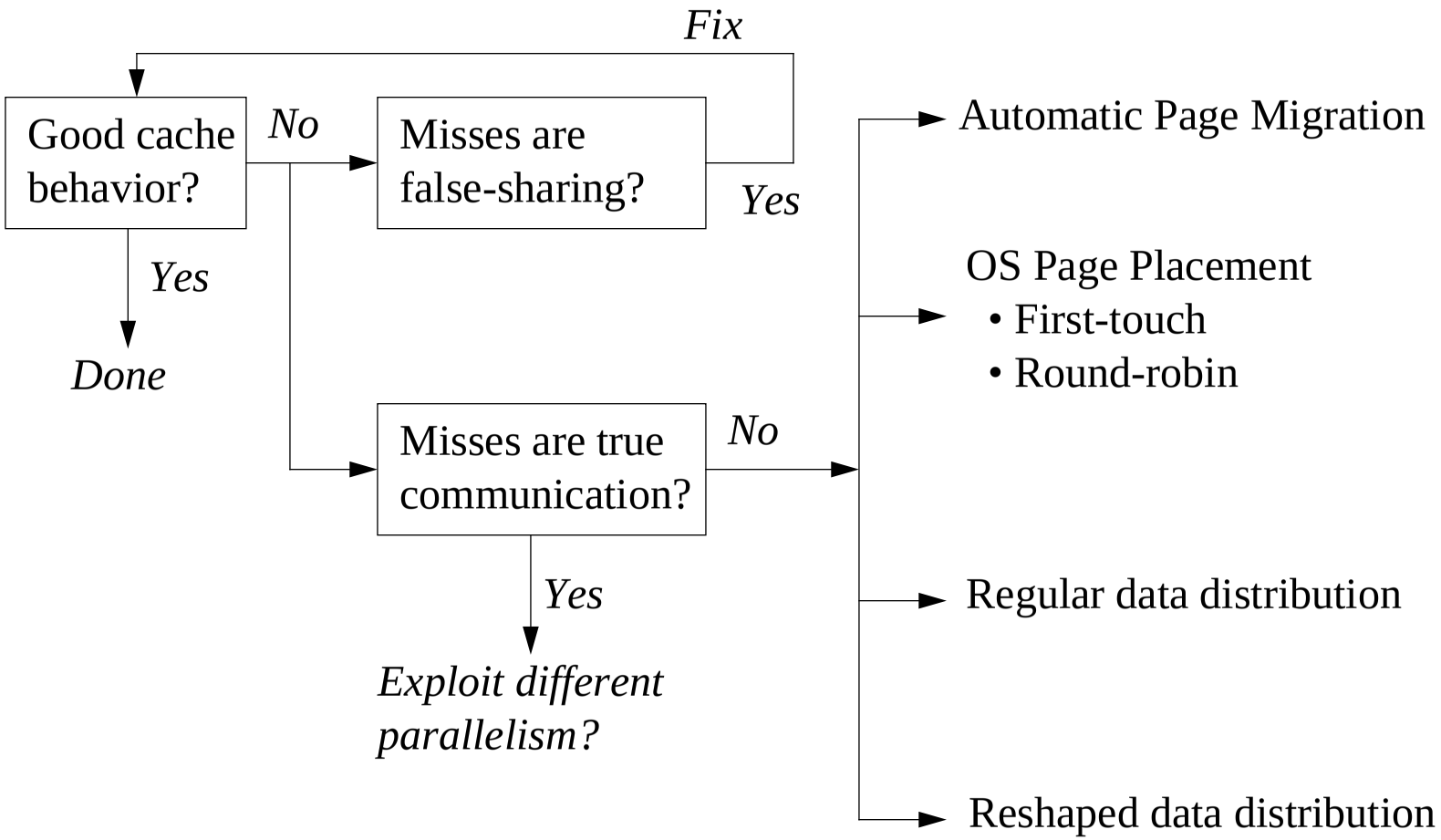
Tuning Parallel Programs

Has the program been properly parallelized?

- enough of the program parallelized (Amdahl's law)
- is the load well-balanced?

Tuning for cache and memory locality

Tuning for Cache and Memory Locality



False Sharing

- Use counters to identify false sharing (perfex, speedshop) (store exclusive to shared block)

```
      integer m, n, i, j
      real a(n,m), s(m)

c$doacross local(i,j) shared(s,a)
      do i = 1,m
         s(i) = 0.0
         do j = 1,n
            s(i) = s(i) + a(j,i)
         enddo
      enddo
```

False Sharing: Fixed

```
      integer m, n, i, j
      real a(n,m), s(32,m)

c$doacross local(i,j) shared(s,a)
      do i = 1,m
         s(1,i) = 0.0
         do j = 1,n
            s(1,i) = s(1,i) + a(j,i)
         enddo
      enddo
```

False Sharing: Another Example

```
      integer lock
      integer somevar

C acquire the lock
      do while (lock_test_and_set(lock, 1) .ne. 0)
      enddo
```

Fill and Align Directives

```
c*$* fill_symbol (lock,{L1cacheline,L2cacheline,page,<n>})
```

- Ensure that no other symbol shares line/page with “s”

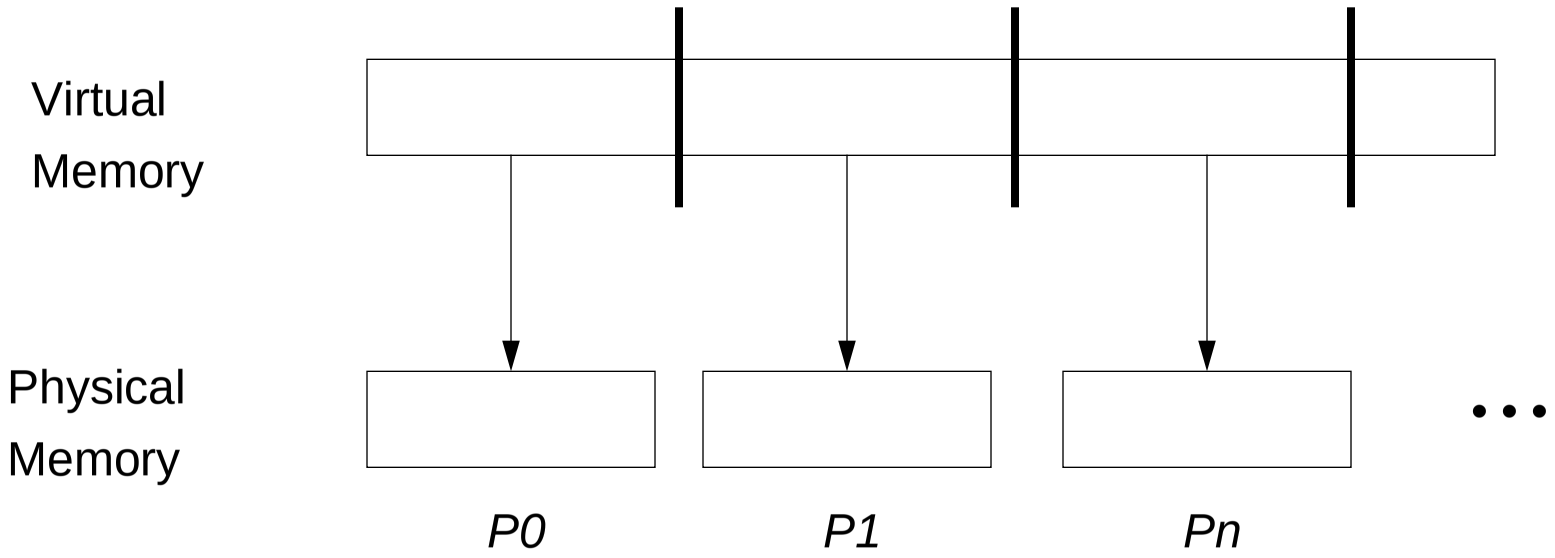
```
c*$* align_symbol (lock, •••)
```

- Align “s” on specified boundary

- Work on globals, COMMONs, VLAs (fill also works on locals)
- Must be supplied consistently at all declarations of a COMMON
- No structure elements, malloc’d variables, allocatable arrays

Data Placement: Underlying Mechanism

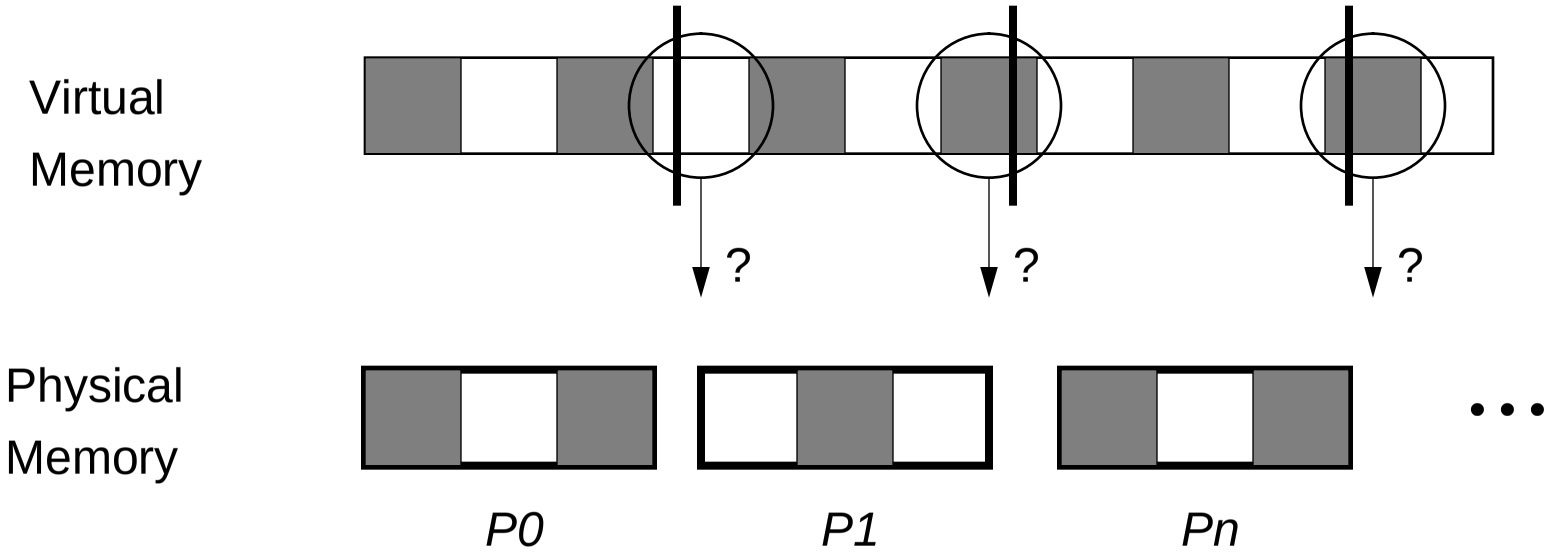
```
    real*8 A(n)
c$doacross
    do i = 1,n
        A(i) = •••
    enddo
```



Data distribution through virtual ---> physical page mapping

Page-Level False Sharing

real*8 A(n)

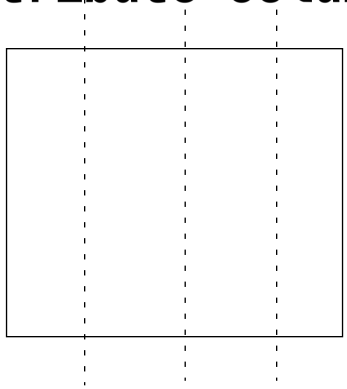


Possible page-level false sharing

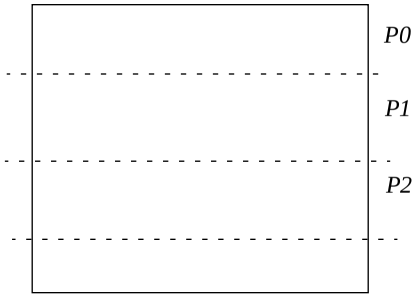
May need to reorganize array layout

Distributing a Two-Dimensional Array

```
real*8 A (1000, 1000)
distribute columns
```



```
real*8 A (1000, 1000)
distribute rows
```



Tradeoffs between page mapping and reorganizing array layout

Choice depends on size and layout of individual processor portions

Allow user-level control over the two choices

Data Placement

Techniques not requiring any code modifications

- round-robin data placement
- automatic page migration

Techniques requiring code modification

- parallel data initialization
- data distribution directives

Round-Robin Placement

Default placement policy is first-touch

- great for most programs, but
- sequential initialization can create bottlenecks

Round-robin placement enabled with an environment variable

```
setenv _DSM_PLACEMENT ROUND_ROBIN
```

Page Migration

Page migration is off by default

- expensive
- impacts all CPUs
- useful for correcting initial poor placement

Page migration enabled with an environment variable

```
setenv _DSM_MIGRATION {ON, OFF, ALL_ON}
setenv _DSM_MIGRATION_LEVEL [0-100]
```

Techniques Requiring Code Modification

First-touch: initialize data in parallel

Add page_place directives (**`c$page_place`**)

Add regular data distribution directives (**`c$distribute`**)

Add reshaped data distribution directives (**`c$distribute_reshape`**)

First Touch Placement Policy

```
      integer i,j,n,niters
      parameter (n = 8*1024*1024, niters = 1000)
      real a(n), b(n), q
C     initialization
      do i = 1,n
         a(i) = 1.0 - 0.5*i
         b(i) = -10.0 + 0.01*(i*i)
      enddo
C     real work
      do it = 1,niters
         q = 0.01*it
         do i = 1,n
            a(i) = a(i) + q*b(i)
         enddo
      enddo
```

First Touch Placement Policy

```
C       initialization
C$doacross local(i), shared(a,b)
        do i = 1,n
           a(i) = 1.0 - 0.5*i
           b(i) = -10.0 + 0.01*(i*i)
        enddo
C       real work
        do it = 1,niters
           q = 0.01*it
C$doacross local(i), shared(a,b,q)
           do i = 1,n
              a(i) = a(i) + q*b(i)
           enddo
        enddo
```

Explicit Page Placement

c$page_place (<*address*>, <*size*>, <*thread-number*>)

- Place virtual address range within memory of a particular thread

Page Placement: Example

```
      do i = 1,n
         ••• initialization •••
      enddo
C Explicitly place the data
      sz = n/p
      do i = 1,p
C$page_place (a(1+(i-1)*sz), sz*4, i-1)
C$page_place (b(1+(i-1)*sz), sz*4, i-1)
      enddo

      do it = 1,niters
C$doacross local(i), shared(a,b,q), affinity(i)=thread(i/sz)
         do i = 1,n
            •••
         enddo
      enddo
```

Data Distribution

HPF-like mechanisms for distribution of array data structures

Provide only core HPF functionality, avoid complex features

Very different implementation strategy

- exploit CC-NUMA hierarchy (flavors of data distribution)
- NOT “owner-computes”
- NOT message-passing

Data Distribution Constructs

```
real*8 A(m, n, . . .)
c$distribute A (<dist>, <dist>, . . .)
```

<dist> may be one of

block
cyclic
cyclic(<expr>)
*

Two flavors of distribution

```
c$distribute          A (<dist>, <dist>, ...)
c$distribute_reshape  B (<dist>, <dist>, ...)
```

Restrictions on Data Distribution

- Distribute local, global, common, formal arrays
- Arrays may be adjustable-size (i.e. with dynamic bounds)
- Number of threads determined at runtime

Restrictions

- no alignment directive
- no offset
- only for true arrays
  (provide adjustable-size arrays in C, C++ ala Cray, GNU-CC)

Distribution Examples

<pictures: jeff.sc, slide #86)

Regular Data Distribution: Example

```
C Explicitly place the data
C$ distribute a(block), b(block)
      do i = 1,n
         ••• initialization •••
      enddo

      do it = 1,niters
C$doacross local(i), shared(a,b,q), affinity(i)= data(a(i))
         do i = 1,n
            •••
         enddo
      enddo
```

- can also do **`affinity(i) = data(A(c1*i+c2))`**

Distribution in Multiple Dimensions

```
      real*8 A(m,n)
c$distribute A(block, cyclic)

c$doacross nest(i,j), affinity(i,j) = data(A(j,i))
      do i = 1,n
         do j = 1,m
            A(j,i) = 1.0
         enddo
      enddo
```

Dynamic Data Redistribution

```
      real*8 A(m,n)
c$distribute A(*,block)

      ...

c$redistribute A(block,block)

      ...
```

- dynamically reallocate the pages
- affect subsequent affinity scheduling
- global in scope

Topology

Optional ONTO clause to control aspect ratio across distributed dimensions

```
c$distribute A (block, cyclic(k)) onto (1, 2)
c$distribute A (block, cyclic(k)) onto (4, *)

c$redistribute A (block,cyclic(k)) onto (2,3)

c$doacross nest(i,j) onto (1,2)
do i = 1,m
   do j = 1,n
      ...
   enddo
enddo
```

Reshaped Data Distribution

```
real A(m,n)
c$distribute_reshaped A(block, cyclic)
```

Program makes no assumptions about storage layout of a reshaped array

Compiler **guarantees** desired distribution

Restrictions on Reshaped Arrays

Reshape specifications must be consistent across COMMON blocks

Reshaped arrays cannot be equivalenced with other storage

(no dynamic reshaping)

Enforced by the compiler

Parameter Passing with Reshaped Arrays

User specifies distribution at array definition

Compiler clones called subroutine for incoming distribution

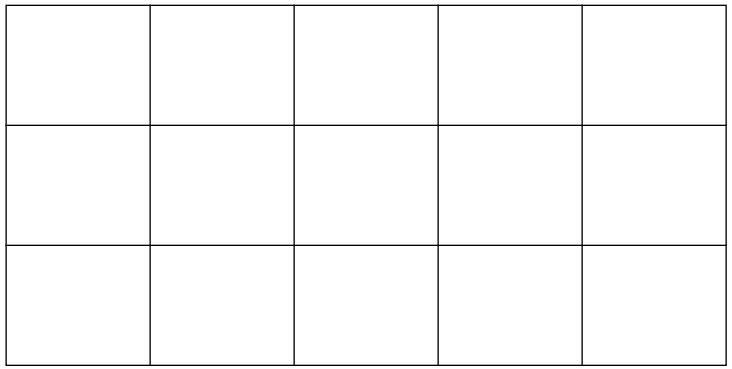
- simple user model
- efficient --- all reshaped distributions always known statically
- potential for code bloat
  (in practice few distinct distributions)

Passing Portions of Reshaped Arrays

```
c$distribute_reshape A(block, cyclic(k))
call matmul (A(i,j))
```

Called procedure can use the actual argument as either

- a scalar
- a contiguous portion of the reshaped array (based on the distribution)



- provide a rich set of intrinsics to query and traverse distributed portions

Catching User Errors

Compile-time checks

- inconsistencies in reshaped arrays across COMMON blocks
- inconsistencies in reshaped arrays across EQUIVALENCE statements
- inconsistencies in distribution on actual and formal parameters

Optional runtime checks (flag: -MP:check_reshape)

- inconsistencies in array bound declarations on formal parameters
- inconsistencies in declared bounds of portions of reshaped arrays

Together with standard array bounds-checking,

catch all errors with reshaped arrays

Automatic Data Distribution

Compiler option to *guess* the desired data distribution

- examine arrays accessed within parallel loops
- generate data distribution directive

Explicit Communication

Task-common data: each thread gets its own copy

Explicit “one-sided” communication (ala SHMEM)

- get/put operations on task-common data

- shmem_get from remote processor

```
mp_shmem_get (&dest, &src, length, threadnum)
```

- shmem_put to copy on remote processor

```
mp_shmem_put (&dest, &src, length, threadnum)
```

Summary of Constructs

Pragma	*Purpose*
c$page_place (<*addr*>, <*sz*>, <*thread*>)	Explicit placement of data
c$doacross affinity (i) = thread (<*expr*>)	Thread-affinity scheduling
c$fill_symbol (s, L2cacheline)	Pad variable
c$align_symbol (s, L2cacheline)	Align variable
c$distribute A (<*dist*>, <*dist*>, • • •)	Data distribution
c$redistribute A(<*dist*>, <*dist*>)	Dynamic data redistribution
c$dynamic A	Redistributable annotation
c$distribute_reshape B(<*dist*>)	Data distribution with reshaping
c$doacross affinity (i) = data (A(i))	Data-affinity scheduling
mp_shmem_get	Explicit communication
mp_shmem_put	Explicit communication

Programming Model Summary

Shared-memory programming model

Focus on problem decomposition

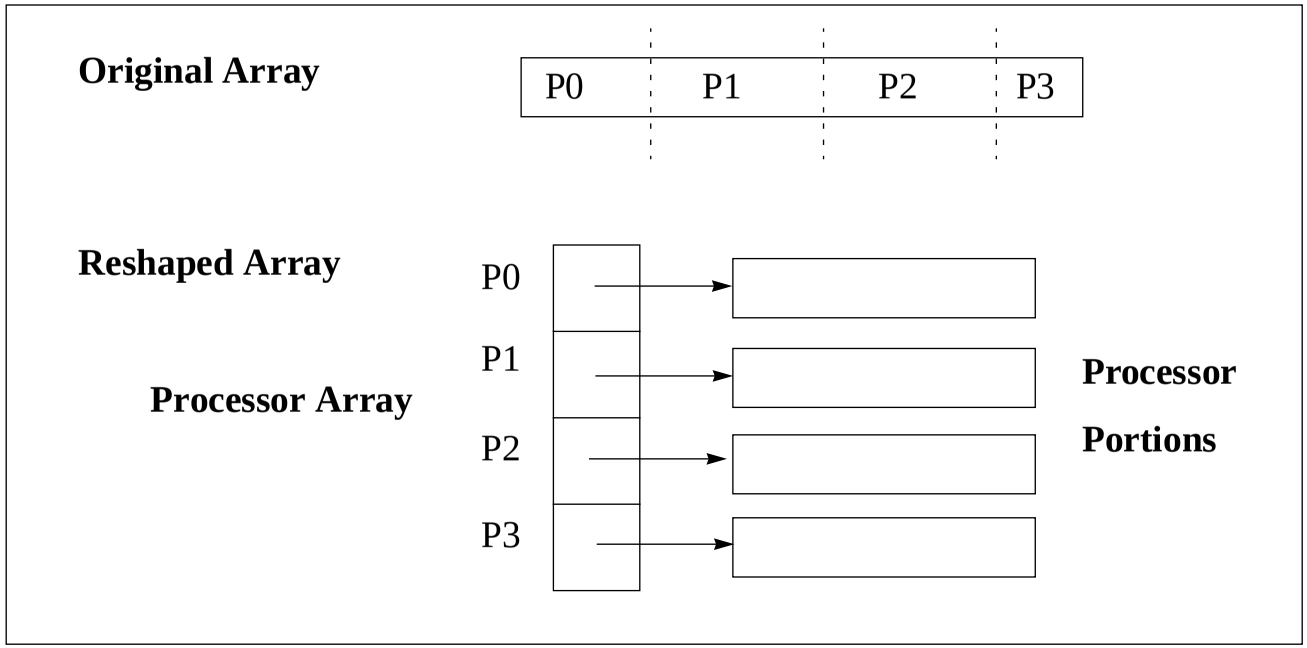
Extensions for enhancing locality

- incremental attention to selected data structures

Hierarchy of optimizations, exploit CC-NUMA hierarchy

- page migration and round-robin distribution
- thread data allocated locally by first-touch
- explicit page placement directives
- distribute shared data based on decomposition
- use regular/reshaped distribution as appropriate
- use shmem-like calls for explicit communication

Implementation Overview: Reshaped Arrays



Distribution	*Transformed Reference*
Block	A[i/b] [i%b]
Cyclic	A[i%P] [i/P]
Cyclic (k)	A[(i/k)%P] [i/(Pk)] [i%k]

Optimization of Reshaped References

```
real*8 A(n)
c$distribute_reshape
      A(block)

do i = 2, n
  A(i)=A(i)+A(i-1)
enddo

do i = 2, n
 A[i/b][i%b] = •••
enddo
```

```
do p = 0, P-1
 lb = MAX(p*b, 2)
 ub = MIN(((p+1)*b, n)
 if (lb .LE. ub) then
  •••
 endif
 idx = (lb+1) % b
 do i= lb+1, ub
  A[p][idx]=A[p][idx]+A[p][idx-1]
  idx = idx+1
 enddo
enddo
```

Loop tiling and peeling

General Tiling w/Affinity

affinity (i) = data (A(s*i+c)). do i = LB, UB, step

Block

```
do p = 0, P-1, 1
  do i = max(LB, ⌈(pb − c)/s⌉), ⌊                    ⌋, step
```

Cyclic (1) s == 1

```
do p = 0, P-1, 1
  do i =                  , UB, P
```

Cyclic(k)

```
do p = 0, P-1, 1
  do j = ⌈                ⌉ , ⌈                ⌉ , 1
    do i =      ⌈          ⌉,      ⌊                ⌋, step
```

Performance: Examples

128-processor SGI Origin-2000

- 4MB cache per processor
- 16 GB of memory
- Local latency: ~ 70 processor cycles
- Remote latency: ~110-180 cycles
- Default first-touch page placement, optional round-robin placement

Results on

- LU (NAS 2.1 parallel benchmarks)
- Matrix transpose
- 2-D convolution

LU: Optimization of Reshaped Array References

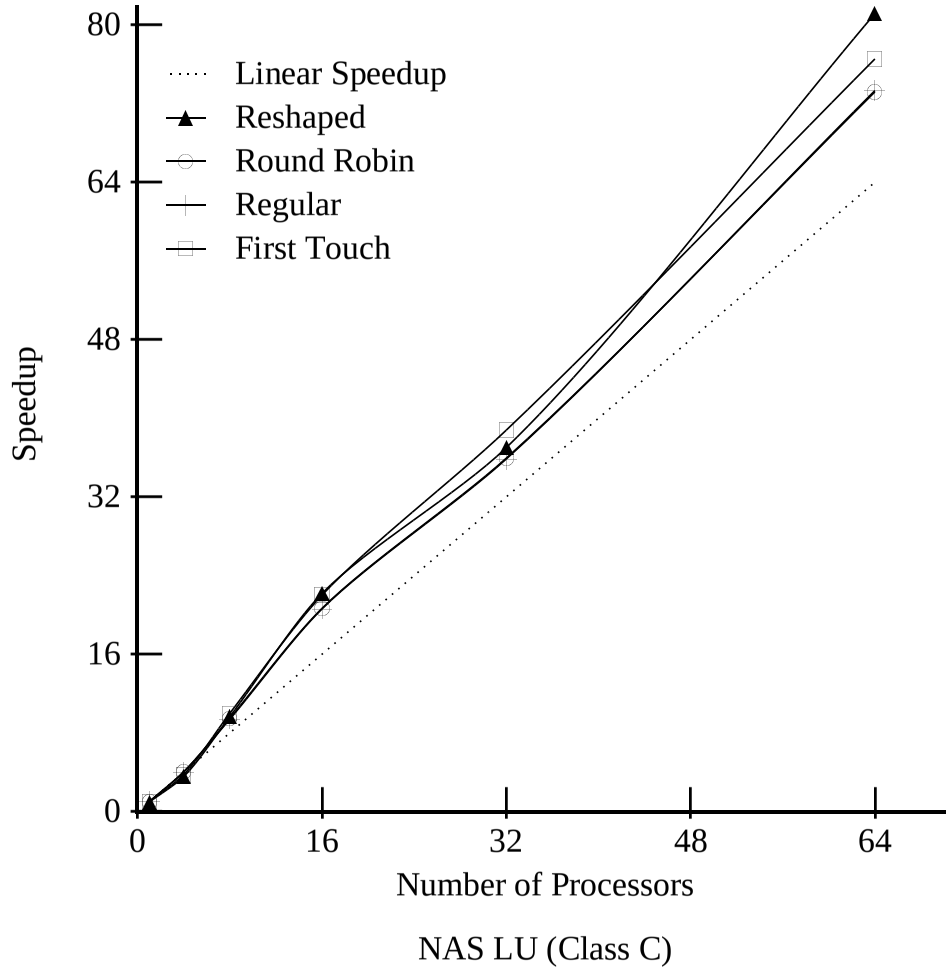
```
real*8  U(5,166,166,166), V(5,166,166,166)
c$distribute_reshape U(*,block,block,*)
c$distribute_reshape V(*,block,block,*)
```

Optimization	*Time (secs)*
Reshape, no optimizations	*83.91*
Reshape, tile and peel	53.26
Reshape, tile and peel, hoist	46.23
Original code without reshaping	45.71

- single processor numbers
- effectiveness of optimizations of reshaped array references

LU: Performance

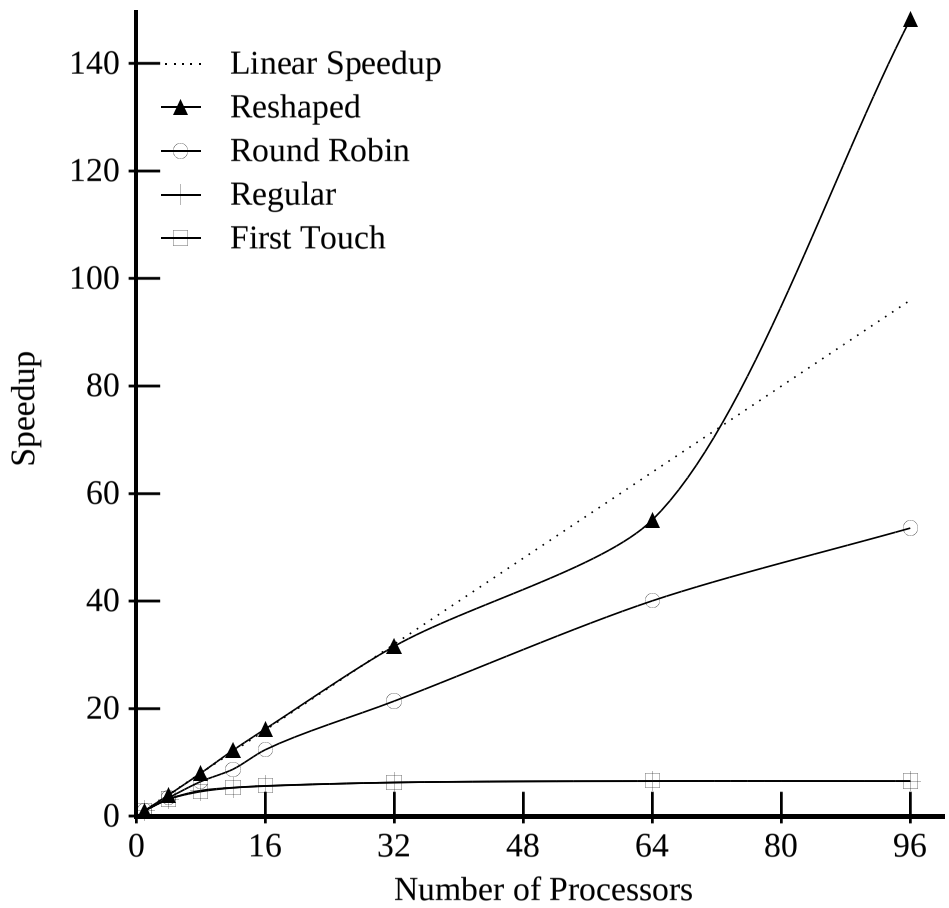
- data is initialized in parallel
- important to spread data (bandwidth)
- superlinear speedup (1-P memory and cache)



Matrix Transpose

```
c$distribute A(*,block)
c$distribute B(block,*)
c$doacross local(i,j)
do i = 1,m
  do j = 1,m
    A(j,i) = B(i,j)
  enddo
enddo
```

- first-touch, regular - poor distr.
- round-robin - good bandwidth
- reshape - cache, memory, TLB

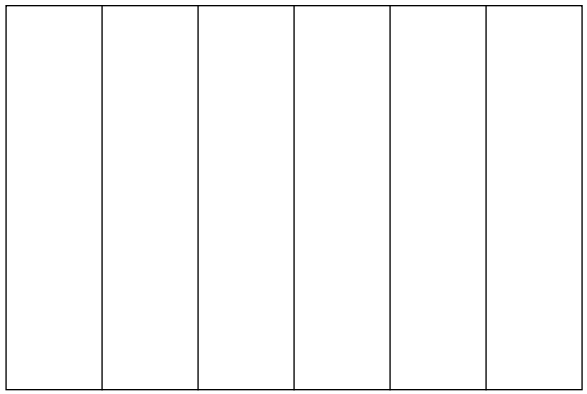


Matrix Transpose 5000x5000

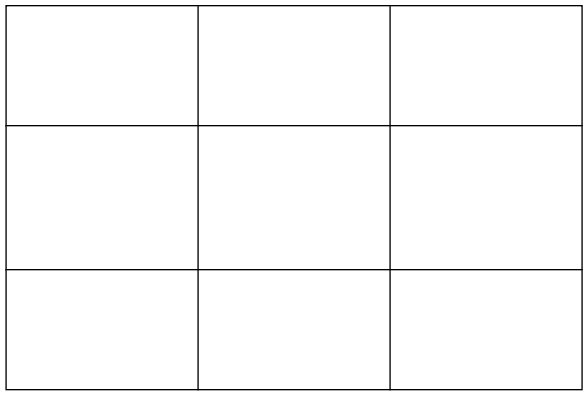
2-D Convolution

```
do j=2,n-1
  do i=2,n-1
    A(i,j)=(B(i,j)+B(i,j-1)+B(i,j+1)+B(i-1,j)+B(i+1,j))/5
  end do
end do
```

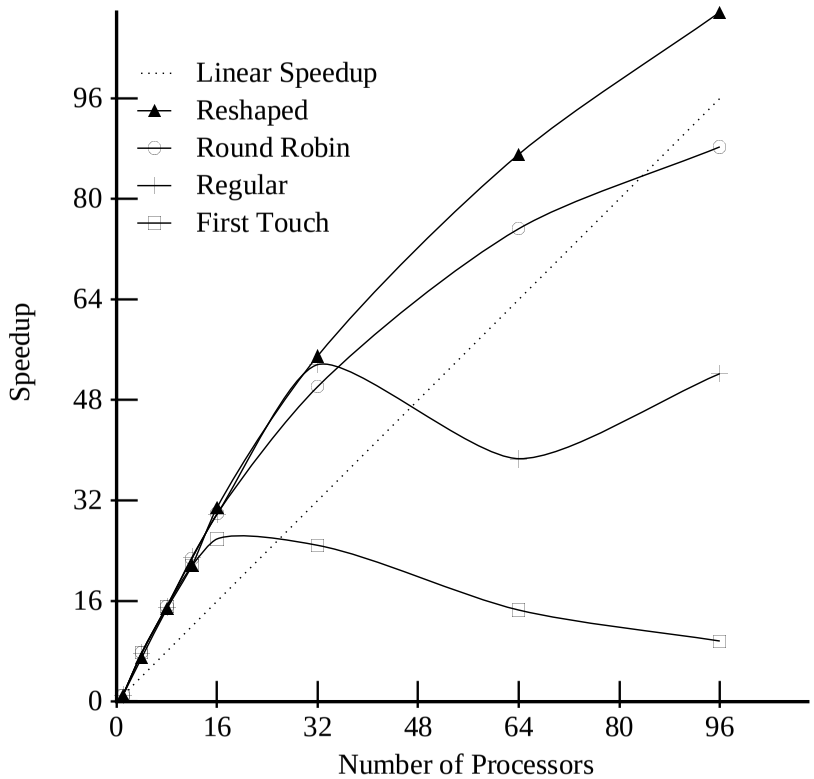
(* block)



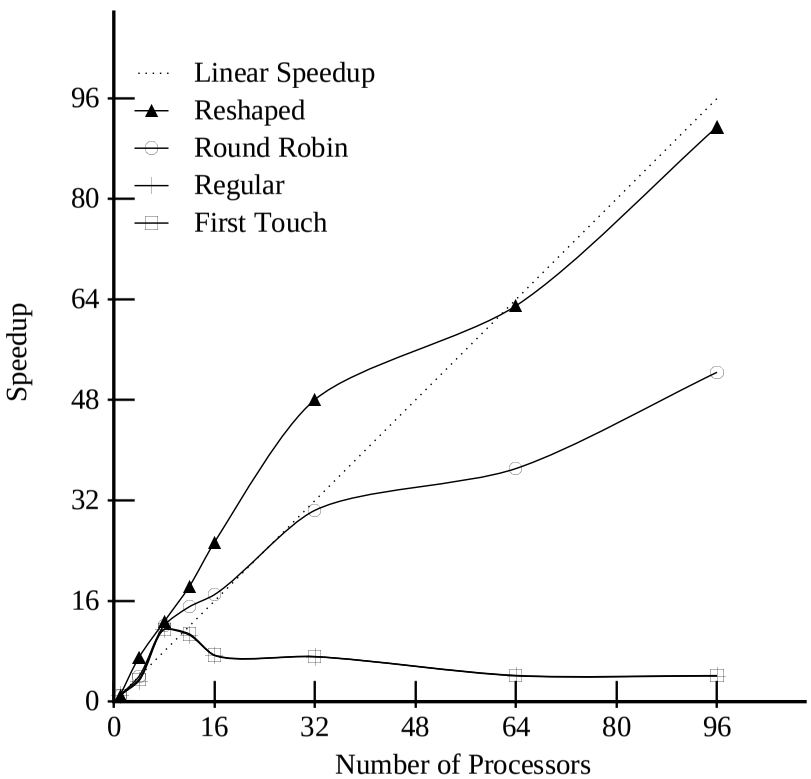
(block, block)



2-D Convolution: Performance



2D Convolution 1000x1000, (*,block)



2D Convolution 1000x1000, (block,block)

- regular is good if processor portions are large and contiguous
- reshaped best for (block, block)

Summary

Tuning for caches is most important

Use machine bandwidth (round-robin)

Large individual portions? page-level distribution is sufficient

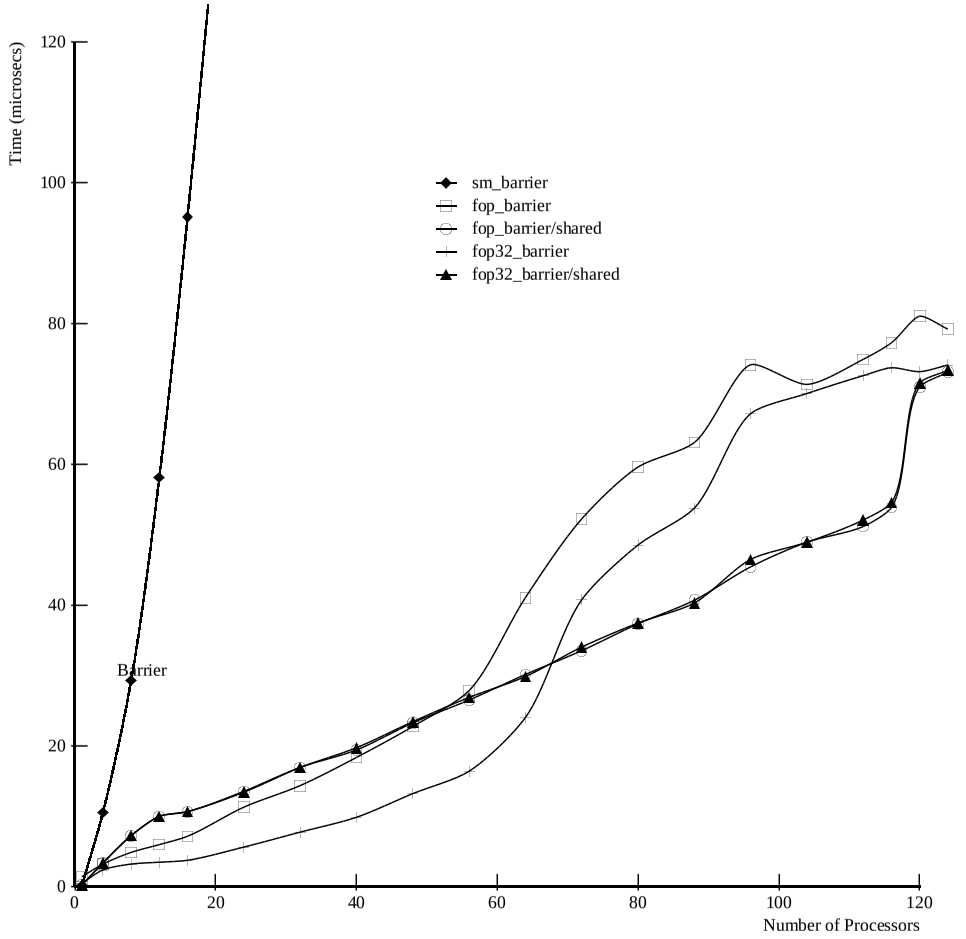
Reshaping can improve TLB, cache, and memory performance

Comparison with HPF

HPF

- designed for both message-passing and shared-memory architectures

Our Abstractions

- regular and reshaped distributions
- no dynamic redistribution of reshaped data
- simple parameter passing model
- provide only core distribution constructs
    - no templates, alignment, ...

Barrier Using Uncached Operations