Distributed Shared Memory

Ruud van der Pas

SiliconGraphics

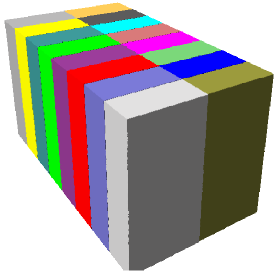
European High Performance Computing Team

Topics

- *Why and When Data Placement ?*
- *Operating System choices*
- *Page migration and replication*
- *Data placement tools*
- *Compiler support for data placement*
- *Environment variables*

Image courtesy of Kaiser Electro-Optics

Why and When Data Placement ?



Scalability and Data Placement

Consider data placement tuning if

- Properly parallelized
- No cache coherency contention
- Scaling is less than expected

IRIX tries to optimize

- The program's topology
- The page placement

IRIX provides

- MLDsets in cluster and cube configurations
- First–touch, round–robin, and fixed page placements
- Dynamic page migration

Data placement tuning is specifying the appropriate policies and topologies and programming with them in mind.

Prescription for Performance – 1

First tune single–processor performance

Tuning parallel programs

Has the program been properly parallelized?
- Is enough of the program parallelized (Amdahl's law)?
- Is the load well–balanced?

MP library programs
- Cache friendly programs: no data placement tuning
- Noncache friendly programs
  - Is false sharing a problem?
    - Determine using *perfex*(1) and *speedshop*(1)
    - Fix by modifying data structures

Prescription for Performance – 2

Tuning data placement

First try techniques involving no code modification
- Try round–robin memory allocation
- Try page migration
- Try page migration and round–robin memory allocation

If necessary, use techniques involving code modification
- Initialize data in parallel and rely on first–touch
- Add regular data distribution directives
- Add reshaped directives

Tuning non–MP library programs (MPI, PVM)

Use *dplace*(1) to ensure optimal data placement
As of MPI 3.0, the implementation is NUMA aware

MP Library Programs

Platform–independent performance criteria

Amdahl's law
Load balance

Platform–dependent performance criteria

Eliminate cache coherency contention (shared memory)
Data placement (distributed shared memory)

Data placement is *not* a problem for:

Single–processor programs
Cache friendly programs

```
% perfex -a -x -y a.out
```


Non Cache Friendly Programs

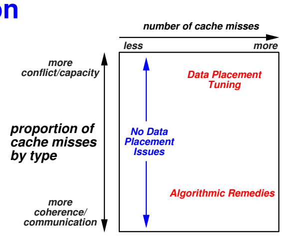
If scaling is poor, the first thing to check for is cache coherency contention

Memory contention

One CPU repeatedly updates a cache line that other CPUs use for input

False sharing

Two or more CPUs repeatedly update the *same* cache line



Identified by high number of cache invalidations

Use *perfex*(1) and *speedshop*(1) with counter 31

Tuning Data Placement for MP Library Programs

Turn to data placement tuning after eliminating other possibilities

Two classes of data placement tuning

Techniques involving no code modification

Use MP library environment variables to change default data placement policies

Techniques requiring code modification

Parallelize data initializations
Use data distribution directives
Modify algorithms

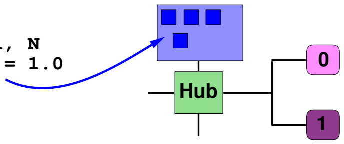
Operating System Choices

Operating System choices

- *The OS decides upon data placement*
- *MLD sets and PMs are used to do this*
- *Default OS behaviour:*

  The CPU that first touches the data will get the corresponding page(s) in it's node memory

```
DO i = 1, N
   A(i) = 1.0
END DO
```



Page Migration

Page Migration

Automatic page migration tries to move pages close to where they are being requested most often

- **Works best when application shows *phase persistence:***

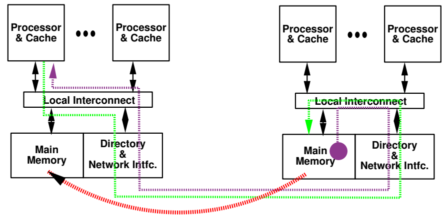
  The working set and pattern of references into it tends to change only at infrequent phase boundaries marking changes in program behavior

- **Migration threshold *n%* means:**

  Consider migrating page when any remote reference count exceeds the local reference count by more than n% of a fixed system constant value (2048) over some period.

- **Read–only pages are replicated; read–write pages are migrated**
- **The Block Transfer Engine (BTE) in the Hub is used for this**

Page Migration – Mechanism



Hardware accelerated BCOPY and TLB update:

HUBs contain page reference counters (64/page)

*If the OS detects that remote > threshold * local: migrate page*

+ *No remote BW or latency impact and simpler HW*

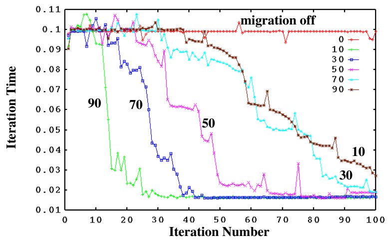
– *Less quick to react to changing access patterns*

Experimenting with Migration Levels

Migration aggressiveness controlled by level *L*

```
% setenv _DSM_MIGRATION_LEVEL L
                              (higher is more aggressive)
```

Iteration time as data migrates from initial placement in a single node



Data Placement – The Easy Way

- **Although**

  First–touch data placement
  Round–robin data placement
  Page migration

  are conceptually very simple, significant performance gains be obtained in those cases where the defaults are not adequate

- **IRIX provides for a very easy way to experiment with these parameters**

First, Try Round–Robin Placement

Default placement policy is first–touch

Great for most programs, but ...
Sequential initializations can create bottlenecks

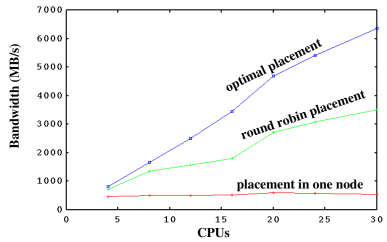
Test and/or fix these problems using round–robin placement

```
% setenv _DSM_ROUND_ROBIN
```


Round–Robin Placement

Performance effect for simple vector code

```
a(i) = b(i) + q*c(i)
```



Try Enabling Page Migration

If round–robin solves the performance problem, you're done!

If not, try page migration

Page migration is off by default

Expensive
Impacts all CPUs
Useful for correcting poor initial placements

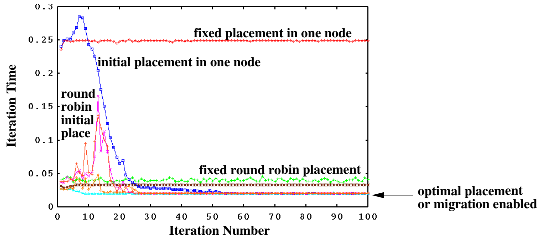
Enabled with *MP library* environment variables

```
% setenv _DSM_MIGRATION ON
% setenv _DSM_MIGRATION ALL_ON
```


Combining Migration and Round–Robin Placement

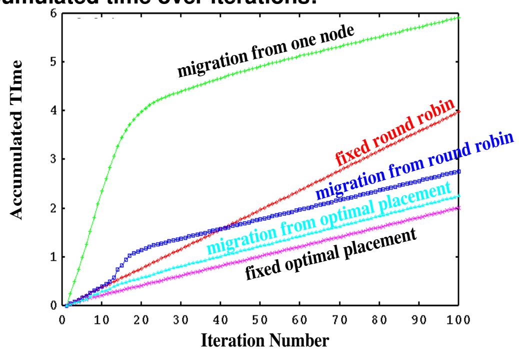
Performance over time with all combinations of:

Serial/parallel initialization
First–touch/round–robin
Migration on/off



Combining Migration and Round–Robin Placement (cont.)

Accumulated time over iterations:



Tuning Data Placement without Code Modification

Recommended steps

First, try round–robin. If the performance is good enough, you're done.
Next try migration. If no better than round–robin, round–robin is preferred.
If migration is better than round–robin, try them both together.

These steps apply only to MP library programs

Sequential programs aren't affected by data placement
First–touch is the right choice for message–passing programs
The environment variables covered are only understood by the MP library

Data placement tuning only helps memory–intensive, cache unfriendly programs

Data Placement Tools

dplace

dplace – A Data Placement Tool

The Big Picture:

- The system will *usually* place data and execution threads together properly, *and* the penalty for putting data on another node is usually fairly small.
- The system will *usually* migrate the data to follow the executing thread if that thread has to move and then remain on another cpu.
- But.... *sometimes* more specific control over data and thread placement is desired.

This is when dplace might do it for you!

dplace – A Data Placement Tool

% *dplace* [–place *placement_file*] [–verbose] [options] program [args]

Uses a high level placement specification in *placement_file* to:

- set up *policy modules*
- create and place *mldsets* of specific topology and *resource affinity*
- assign *threads* and *mld's* to nodes
- place *address ranges* on nodes

All without recompilation.

dplace also provides a *subroutine library interface* to allow modification of the choices and explicitly controlled data migration during the program's execution.

dplace – Placement Capabilities

Static Specifications

Declare number of threads to use

Declare number of memories to use

Specify distribution of threads onto memories

Specify distribution of virtual addresses onto memories

Specify text/data/stack page sizes

Specify migration policy and threshold

Dynamic Specifications

Migrate virtual address ranges between memories

Move threads from one node to another

dplace – Command Line Options

```
dplace
     [-place placement_file]
     [-data_pagesize n-bytes]
     [-stack_pagesize n-bytes]
     [-text_pagesize n-bytes]
     [-migration threshold]
     [-propagate]
     [-mustrun]
     [-v[erbose]]
     program [program-arguments]
```


dplace – Example: Page Migration

dplace controls migration aggresiveness via the migration threshold:

`% dplace -migration` *`threshold`* `a.out`

The threshold is the percentage difference between the number of remote memory accesses and local memory accesses, relative to a maximum counter value:

threshold = 100 – level

A high threshold is less aggressive
A low threshold is more aggressive
A threshold of 0 turns off migration

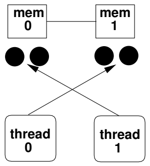
If you enable migration, use a conservative threshold (~90)

dplace – Example Placement File

Default is topology–free, or *cluster* topology

Topology specified in placement file:

```
# placement file
memories 2 in topology cube
threads 2
run thread 0 on memory 1
run thread 1 on memory 0
```



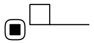
Distance Metric

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

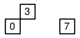
dplace – More Examples

memories 6 in topology cube near /dev/graphics

memories 5 in topology cluster near /dev/graphics



**memories 3 in topology physical near /hw/.../n0 **
/hw/.../n3 /hw/.../n7



dplace – Placement File Syntax

This file must include

Number of memories being used, and, optionally, their topology:

```
memories m [[in] topology none|cluster|cube]
```

Number of threads:

```
threads n
```

Assignment of threads to memories:

```
run thread n on memory m [using cpu k]

distribute threads [t0:t1[:dt]] across memories \
[m0:m1[:dm]][block [m]] | [cyclic [n]]
```

Optional specifications for command line options:

```
policy stack|data|text pagesize n [k|K]
policy migration n [%]
mode verbose [on|off|toggle]
```


dplace – Indicating Resource Affinity

Origin system components are listed in the *hardware graph* (see *hwgraph*(4))

```
 % find /hw -name node -print
      /hw/module/1/slot/n1/node
      /hw/module/1/slot/n2/node
      /hw/module/1/slot/n3/node
      /hw/module/1/slot/n4/node
      /hw/module/2/slot/n1/node
                  .
% find /hw -name kona -print
      /hw/module/1/slot/io5/xwidget/kona
```

One can use these names to specify resource affinity:

```
memories m [[in] topology none|cluster|cube|physical] [near [/hw/*]+]

memories 3 in topology physical near /hw/module/2/slot/n1/node \
/hw/module/1/slot/n2/node \
/hw/module/3/slot/n3/node

memories 3 in topology cluster near /hw/module/1/slot/io5/xwidget/kona
```

dplace – Symbolic Placement Files

dplace allows use of environment variables and simple arithmetic in placement files.

One type of common placement file will be built on:

```
memories $MP_SET_NUMTHREADS in topology cube
threads $MP_SET_NUMTHREADS
distribute threads across memories block 1
```

This will create $MP_SET_NUMTHREADS *mld's* and will distribute the threads across the memories in blocks of 1, so one thread will be allocated per memory.

As each of the parallel threads touches new memory locations, the pages will be faulted in from the *mld* on the node where the thread is running. If later accesses are dominantly remote, the page *may* migrate (if enabled).

dplace – Another Example

A second common placement file will be built on:

```
memories ($MP_SET_NUMTHREADS+1)/2 in \
          topology cube
threads $MP_SET_NUMTHREADS
distribute threads across memories block 2
```

This will create one *mld* for every two threads and will distribute the threads across the memories in blocks of 2, so one thread will be allocated per cpu.

You would probably choose 1 thread per node for codes that are *memory–bandwidth–bound*, while you would probably choose 2 threads per node for *cpu–bound* codes or codes with large amounts of data sharing between threads.

dplace and MPI 2.0

MPI placement file

```
# scalable placement_file for MPI
# two threads per memory
memories ($MPI_NP + 1)/2 in topology cluster # set up memories
threads $MPI_NP + 1                          # number of threads
distribute threads 1:$MPI_NP across memories # ignore the lazy thread
```

Dplace command follows mpirun command

```
% mpirun -np <nprocs> /usr/sbin/dplace <dplace_args> a.out <a.out args>
% mpirun -p <machine> <nprocs> /usr/sbin/dplace <dplace_args> \
           ./a.out <a.out args>
```

Use ssrun command before dplace command

```
% mpirun -np <nprocs> /bin/ssrun <ssrun_args> /usr/sbin/dplace \
           <dplace_args> a.out <a.out args>
% mpirun -p <machine> <nprocs> /bin/ssrun <ssrun_args> \
           /usr/sbin/dplace <dplace_args> ./a.out <a.out args>
```


dplace – Example using addresses

Placing data on specific nodes requires knowing the virtual addresses involved. These can be obtained from using the *dprof* profiler, explicitly printing the values, or using dbx.

Placement file syntax:

```
place range k to l on memory m [[with] pagesize n [k|K]]
```

An example placement file might contain:

```
threads 2
memories 2
distribute threads across memories block 1
place range 0xfffbffc000-0xfffc000000 on memory 0
place range 0xfffd2e4000-0xfffda68000 on memory 1
```

This is intended to fix placement problems for existing binaries
It is designed to be used in conjunction with *dprof*(1)

```
% dprof -pout dplace_file a.out
```

dplace – API

Dynamic specifications can be implemented:

C

```
void dplace_file(char *filename);
void dplace_line(char *line);
```

Fortran

```
CHARACTER*n string
CALL dplace_file (string)
CALL dplace_string (string)
```

These are useful for the dynamic specifications:

```
"migrate range va1 to va2 to memory m"
"move thread n to memory m"
"mode verbose [on|off|toggle]"
```

Link with –ldplace

dplace – API/Example

```
      CHARACTER*128 s
      np = mp_numthreads()
      WRITE(s,*) 'memories ',np,' in topology cluster'
      CALL dplace_line(s)

      WRITE(s,*) 'threads ',np
      CALL dplace_line(s)

      DO i=0, np-1
         WRITE(s,*) 'run thread',i,' on memory',i
         CALL dplace_line(s)
         head = %loc( a( 1+i*(n/np) ) )
         tail = %loc( a( (i+1)*(n/np) ) )
         WRITE(s,*) 'place range',head,' to',tail,
     &              ' on memory',i
         CALL dplace_line(s)
      END DO
      DO i=0, np-1
         WRITE(s,*) 'move thread',i,' to memory',np-1-i
         CALL dplace_line(s)
      END DO
      DO i=0, np-1
         head = %loc( a( 1+i*(n/np) ) )
         tail = %loc( a( (i+1)*(n/np) ) )
         WRITE(s,*) 'migrate range',head,' to',tail
     &              ' to memory',np-1-i
         CALL dplace_line(s)
       END DO
```


Data Placement Tools

dprof

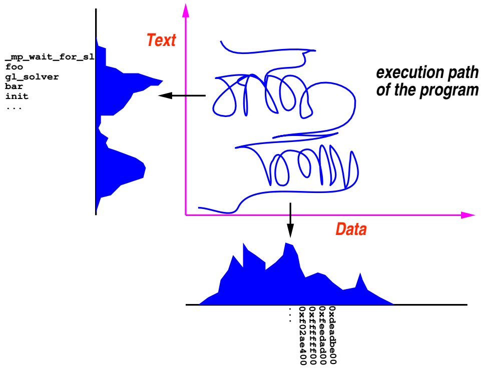
dprof – A data movement profiler

dprof **gives users the ability to analyze which regions of virtual memory each thread of a program accesses.**

The output is a histogram showing reads and writes for each thread for each memory region.

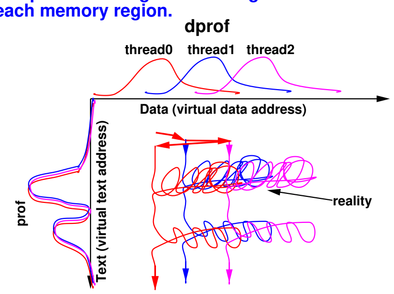
Under Engineering Development

Data Profiling



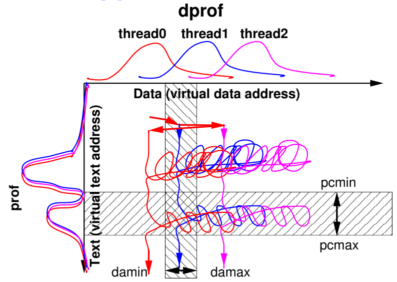
Dprof: Memory Access Analysis Tool

- dprof gives users the ability to analyze which regions of virtual memory each thread of a program accesses.
- The output is a histogram showing reads and writesfor each thread for each memory region.



Using dprof

Program Counter and Data Address Range Filters are supported



dprof – data movement profiler

dprof **[options] program [args]**

Brief Overview:

Choose the "collection method" and sampling rate

Choose Program Counter and/or Data Address range filters

Choose histogram bin size (page, cache line, word)

Choose location for histogram output file

Choose whether or not to build a *dplace* placement file.

Using dprof

% dprof [options] program [args]

Collection method and sampling rate:
[–itimer [–ms n]]
([–hwpc–sc | –hwpc–pc | ...])[–ovfl m]

Program counter and/or data address range filters:
[–pcmin min] [–pcmax max]
[–damin min] [–damax max]

Histogram bin size:
[–page | –line | –double | –int]

Location for histogram output file:
[–out file]

Whether or not to build a dplace placement file
[–pout file]

dprof – Example

The histogram output file contains (base) addresses, thread number, and numbers of observed reads and writes:

address	thread	reads	writes
0x0010048000	0	300	10
0x001004c000	0	2	0
0x0010078000	0	50	10
0x0010048000	1	2	10
0x001004c000	1	300	5
0x0010078000	1	10	50

(etc, ad nauseum)

In the above example, the page at x48000 is used almost exclusively by thread 0, the page at x4c000 is used almost exclusively by thread 1, and the page at x78000 is used almost evenly by both threads.

Determine the relevant addresses

```
#ifdef DPROF
# define MEM_ADDRS(N, P, A)                                  \
                                                             \
        printf("ADDR - %15s, processor %3d: 0x%llx\n",       \
                N, P, (unsigned long long) (A));

#else
# define MEM_ADDRS(N, P, A)
#endif
```

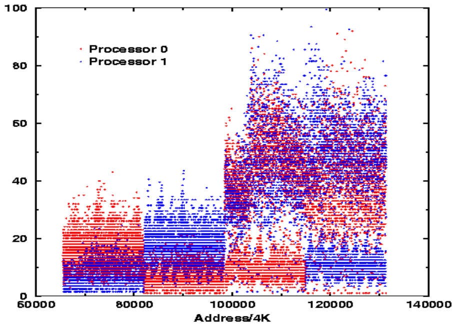
Use something like this in your code to print the addresses

```
-------------------------------------------------------------------------------
3-D FFT PDE TEST
DIMENSIONS =  256  256  128
ADDR -     x0, processor   0: 0x1001c000
ADDR -     x0, processor   1: 0x1405c000
ADDR -    x0p, processor  -1: 0x1809c008
ADDR -     x1, processor   0: 0x180a2180
ADDR -     x1, processor   1: 0x1c0e6180
ADDR -    x1p, processor  -1: 0x2012a188
ADDR -    scr, processor   0: 0x20168000
ADDR -    scr, processor   1: 0x2016c000
ADDR -   scrp, processor  -1: 0x2012a610
T =    1     CHECKSUM =    5.046735008193e+02    5.114047905510e+02
T =    2     CHECKSUM =    5.059412319734e+02    5.098809666433e+02
T =    3     CHECKSUM =    5.069376896287e+02    5.098144042213e+02
T =    4     CHECKSUM =    5.077892868474e+02    5.101336130759e+02
T =    5     CHECKSUM =    5.085233095391e+02    5.104914655194e+02
T =    6     CHECKSUM =    5.091487099959e+02    5.107917842803e+02
TEST COMPLETED
CPU TIME =   31.294603
WALLCLOCK TIME =   50.338659
```

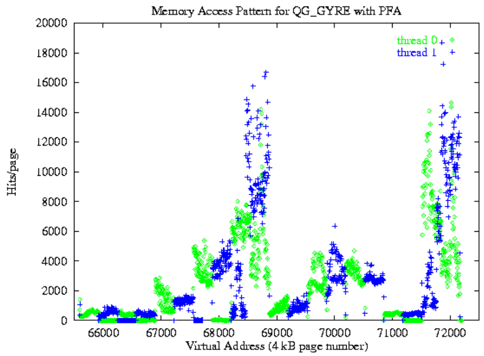
Dprof's sampling causes the run time to increase

Plotting dprof's output – DIY

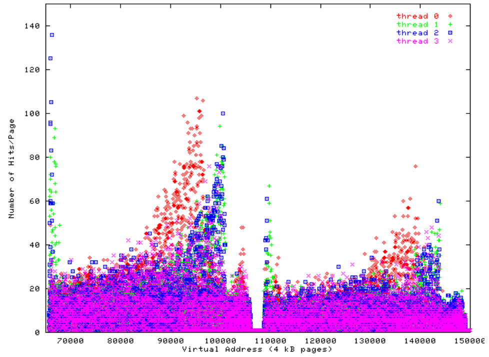
This program has reasonably good data locality



Example of Good Data Distribution



Example of Bad Data Distribution



Compiler Support for Data Placement

MIPSpro 7.2 Placement Directives

Implemented as extensions to MIPSPro compilers:

- **Release 7.1: First partial implementation**
- **Release 7.2: Full implementation for F77, F90, C and C++**

New directives and pragmas

Directive	Purpose
`c$doacross affinity(i) = data(A(a*i+b))`	Clause that puts loop iterations on the node that owns the data used in that iteration
`c$doacross affinity(i) = thread(<expr>)`	Clause that assigns loop iterations to threads based on the induction variable
`c$doacross nest(i,j,...) [onto (<np>,<np>,...)]`	Clause that allows perfectly nested parallel loops
`c$page_place A(<addr>,<sz>,<thread>)`	Declares specific placement of range of addresses to threads
`c$distribute A(<dist>,<dist>,...) [onto (<np>,<np>,...)]`	Declares static distribution of arrays by pages
`c$distribute_reshape A(<dist>,<dist>,...) [onto (<np>,<np>,...)]`	Declares static distribution of array regardless of page boundaries
`c$redistribute A(<dist>,<dist>,...) [onto (<np>,<np>,...)]`	Executable statement that redistributes array at runtime
`c$dynamic A`	Declares that array can be dynamically redistributed at runtime

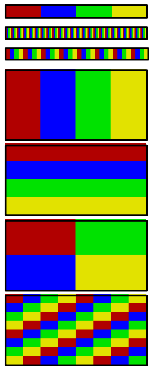
Data Distributions

The distribution can be:

- BLOCK
  divide index range (N) into M blocks,
  each with ~N/M elements
  assign to M memories linearly
- CYCLIC(k)
  divide index range into pieces of size k elements each
  assign round–robin to M memories
- "*"
  not distributed
  moving along this axis stays within one memory

This is analogous to SCHEDTYPE in iteration space (BLOCK = simple, CYCLIC = interleaved)

Data Distributions – Examples

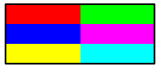


a) X(BLOCK)
b) X(CYCLIC) *or* X(CYCLIC(1))
c) X(CYCLIC(3))
d) Y(*, BLOCK)
e) Y(BLOCK, *)
f) Y(BLOCK, BLOCK)
g) Y(CYCLIC(8), CYCLIC(16))

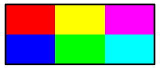
Data Distributions – ONTO clause

6 Processors

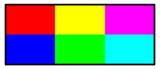
c$distribute A(block,block)



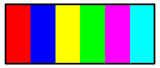
c$distribute A(block,block) onto (2,3)



c$distribute A(block,block) onto (1,2)



c$distribute A(block,block) onto (1,*)



New pragmas in C/C++ / Examples

```
double a[1000][1000];

#pragma distribute_reshape a[block][cyclic]
#pragma pfor affinity(i) = data (b[i])
#pragma pfor nest(i,j) affinity(i,j)=data(a[i][j])
```


Initialization: serial
Real work: parallel

```
      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
c$doacross local(i), shared(a,b,q)
         do i = 1, n
            a(i) = a(i) + q*b(i)
         enddo
      enddo
```


Distribution of Arrays a and b

```
      integer i, j, n, niters
      parameter (n = 8*1024*1024, niters = 1000)
c
c-----Note that the distribute directive is used after the arrays
c-----are declared.
c
      real a(n), b(n), q
c$distribute a(block), b(block)

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
c$doacross local(i), shared(a,b,q), affinity (i) = data(a(i))
         do i = 1, n
            a(i) = a(i) + q*b(i)
         enddo
      enddo
```


The Affinity Clause

Data form:

```
      c$distribute a(block,*)

      c$doacross local(i,j), shared (a)
      c$&,       affinity(i) = data(a(2*i+3,j))
            do i = 1, n
               do j = 1, n
                  a(2*i+3,j) = a(2*i+3,j-1)
               enddo
            enddo
```

Thread form:

```
            integer n, p, i
            parameter (n = 8*1024*1024)
            real a(n)

            p = 1
      c$    p = mp_numthreads()

      c$doacross local (i), shared (a,p)
      c&,        affinity(i) = thread(i/((n+p-1)/p))
            do i = 1, n
               a(i) = 0.0
            enddo
```

The Nest Clause

Parallelizes perfectly nested loops:

```
      real a(m,n)
c$distribute a(block,block)

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

May combine with affinity and onto clauses:

```
      real a(m,n)
c$distribute a(block,block)

c$doacross nest(i,j), local(i,j), shared (a,m,n)
c$&,       affinity(i,j) = data(a(i,j)) onto (2,1)
      do j = 1, n
         do i = 1, m
            a(i,j) = 0.0
         enddo
      enddo
```


Directive page_place

Useful for irregular and dynamically allocated data structures

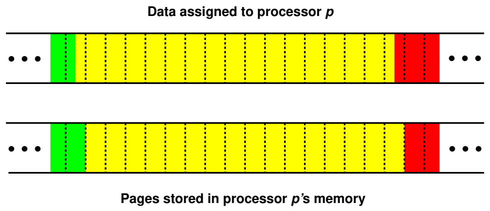
```
            integer n, p, npp, i
            parameter (n = 8*1024*1024)
            real a(n)

            p = 1
c$          p = mp_numthreads()

c-----distribute a using a block mapping

            npp = (n + p-1)/p        ! number of elements per processor

c$doacross local(i), shared(a,npp)
            do i = 0, p-1
c$page_place (a(1 + i*npp), npp*4, i)
            enddo
```

Use for initial placement rather than redistribution

C example:

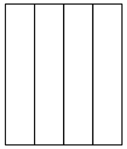
```
float a[N];
#pragma page_place (a, N*sizeof(float), iproc)
```


Page Granularity



Page Granularity

1000x1000 64 bit–array



c$distribute A(*,block)

8*10^6/P bytes per processor

1 page = 16*1024 bytes

~500/P *pages* per processor

1000x1000 64 bit–array

c$distribute A(block,*)

8*10^6/P bytes per processor

1 page = 16*1024 bytes

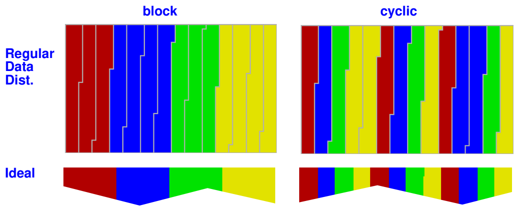
8*1000/P *contiguous bytes* per processors

This is always less than a page!

c$distribute_reshape solves this problem

Bad data–layout example

Page Boundaries:



+ no changes to virtual addresses
+ simple to add to existing code (incrementally)
– imperfect distribution at page boundaries
– page granularity limits choices of distribution

Data Distribution

- Applicable to all local, global and common block arrays
- An array falls into one of these categories:
  - *distributed*
  - *distributed_reshape*
  - *none of the above (default)*
- This holds for the duration of the program
- There are several restrictions on distributed reshaped arrays

Data Reshaping – *Rules of the Game/1*

Restrictions on any array subject to reshape:

1. The array can not be equivalenced to another array, neither explicit or implicit

If the array occurs in a common block, then each declaration of that block must:

- *Contain an array at the same offset within the common block*
- *Declare the array with the same number of dimensions*
- *Specify the same reshaped distribution for the array*

Data Reshaping – *Rules of the Game/2*

Restrictions on any array subject to reshape:

2. If the entire array is passed as an argument, then the number of dimensions and the size of each dimension must match between formal and actual parameters

Passing an <u>element</u> of a reshaped array is treated as passing a portion of a distributed array, where the size and the portion depend on the actual distribution

Data Reshaping – *Rules of the Game/3*

Restrictions on any array subject to reshape:

Besides noncontiguous storage layout,

- *BLOCK DATA initialized data cannot be reshaped*
- *alloca(3)/malloc(3) arrays cannot be reshaped*
- *F90: A dynamically sized automatic array is "distribute"able, but an F90 allocatable array is not*
- *Cannot mix I/O for a reshaped array with namelist I/O or a function call in the same I/O statement*
- *A common block containing a reshaped array cannot be linked –Xlocal*

Data Reshaping – *Example*

This is a valid use of a reshaped array:

```
program slices

REAL*8 A(1000)
c$distribute_reshape A(cyclic(5))

DO i = 1, 1000, 5
   CALL mysub(A(i))
END DO

STOP
END

SUBROUTINE mysub(X)
real*8 X(5)
....
END
```


Data Reshaping – *Compiler Support/1*

- **Reshaped distributions do *not* have to be specified on formal parameters of subroutines**

distribute_reshape directives need to be supplied only at array definition points

These are then propagated down the call chain by the compiler

Different invocations of the same routine, using different data distributions, are cloned automatically by the compiler

Data Reshaping – *Compiler Support/2*

- **Compile–time, link–time and runtime error–detection support is provided to enforce consistency**

Amongst other things, checks are performed for:

equivalence of reshaped arrays

consistency of common blocks

Runtime support is useful when a (portion of a) reshaped array is passed as an argument to a subroutine

Runtime is controlled through a compiler option

Data Reshaping – *Compiler Support/3*

- Intrinsics are provided to traverse the individual portions of a distributed array

These can be used by the programmer to verify the data distribution characteristics

Doing Your Own Data Reshaping

You can accomplish the same no–page–granularity distribution via dynamic memory allocation and pointers

- Allocate enough space for each processor so that pages don't have to be shared
- Pack each processor's pieces into its pages
- Use first–touch or page_place to make sure each processor's pages are stored locally
- Set up an array of pointers for convenient access to data

Redistributed Arrays

- The distribution is changed at *runtime*
- Implemented with a directive, but it resolves to an *executable statement* (can appear anywhere):

  c$redistribute B(BLOCK,*)

- It requires

  c$dynamic B

  in the declaration section of any function that contains a c$doacros loop with data affinity for B
  Data affinity is then determined at runtime

- The runtime lookup incurs extra overhead (can override with explicit c$distribute instead of c$dynamic if the distribution is known within the routine; more efficient)

To Redistribute or Not ?

- Redistributing an array is possible, but it is costly
- In certain cases it can be more efficient to copy the data into a local array
- It depends on the specific details which one is best:

  c$redistribute
  c$page_place
  explicit copying

  Also combinations of these can be used of course

Data Distributions – Summary

The following distributions are available:

C$DISTRIBUTE ARRAY(dist1, dist2, ...)

This is useful for (large) arrays where page granularity effects can be ignored

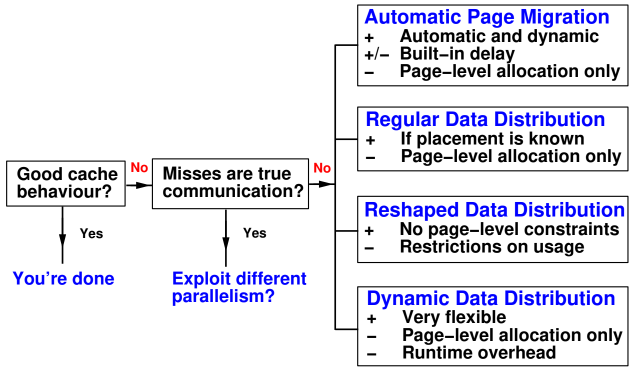
C$DISTRIBUTE_RESHAPE ARRAY(dist1, dist2, ...)

This is useful for (large) arrays where page granularity effects can NOT be ignored, but there are restrictions on usage

C$REDISTRIBUTE ARRAY(dist1, dist2, ...)

This is an executable statement with a global effect . It can be useful if a different lay–out is required during program execution

Tuning for Data Locality



The DSM Environment

The DSM Environment

- The Origin system supports additional control mechanisms:
  - DSM Environment Variables
  - Query library DSM routines
  - New DSM compiler options
- These controls are part of the new MP library (libmp)
- Note that explicit placement control overrules dplace

MPlib – Environment Variables/1

Variable	Use	Default
_DSM_VERBOSE	Get runtime report of memory and thread placement	not set
_DSM_ROUND_ROBIN	Causes pages to be allocated in a cyclic way	not set
_DSM_MIGRATION	Set ON to enable migration of data, except explicitly placed using dd directives; ALL_ON to enable migration of any data	OFF
_DSM_MUSTRUN	Set (any value) to lock threads to processors. Not recommended in a timeshared environment	not set

MPlib – Environment Variables/2

Variable	Use	Default
_DSM_BARRIER	Enable synchronization type Can be set to: SHM – Use memory LLSC – Use LL/SC instructions FOP – Use fetch&op on HUB	SHM
_DSM_WAIT	What do at synchronization points SPIN – Pure busy–wait YIELD – Spin, but with "naps"	YIELD
_DSM_PPM	Process per node memory; set to 1 to use only one CPU/memory	2
_DSM_OFF	Set (any value) to disable data distribution	not set
PAGESIZE_STACK PAGESIZE_DATA PAGESIZE_TEXT	Set virtual page sizes in KByte. May reduce number of TLB faults	16

Using the environment variables

Use _DSM_VERBOSE to verify data distribution:

```
% setenv _DSM_VERBOSE
% a.out
[ 0]: 16 CPUs, using 4 threads.
        1 processors per memory
        Migration: OFF
        MACHINE: IP27 --- is NUMA ---
        MACHINE: IP27 --- is NUMA ---
Created 4 MLDs
Created and placed MLD-set. Topology-free, Advisory.
MLD attachments are:
        MLD  0. Node /hw/module/3/slot/n4/node
        MLD  1. Node /hw/module/3/slot/n3/node
        MLD  2. Node /hw/module/3/slot/n1/node
        MLD  3. Node /hw/module/3/slot/n2/node
[ 0]: process_mldlink: thread  0 (pid  3832) to memory  0. -advisory-.
Pagesize: stack 16384 data 16384 text 16384 (bytes)
--- finished MLD initialization ---
                                  .
                                  .
                                  .
[ 0]: process_mldlink: thread  1 (pid  3797) to memory  1. -advisory-.
[ 0]: process_mldlink: thread  2 (pid  3828) to memory  2. -advisory-.
[ 0]: process_mldlink: thread  3 (pid  3843) to memory  3. -advisory-.
[ 0]: process_mldlink: thread  0 (pid  3832) to memory  0. -advisory-.
```

```
% dplace a.out
[ 0]: 16 CPUs, using 4 threads.
      DSM disabled.
      Migration: OFF
      MACHINE: IP27 --- is NUMA ---

                 etc
```


MP compiler options

-MP:dsm={on, off} (default on)

All the data–distribution and scheduling features are enabled by default under -mp compilation. To disable all the DSM–specific directives (for example, distribution and affinity scheduling), compile with -MP:dsm=off.

-MP:clone={on, off} (default on)

The compiler automatically clones procedures that are called with reshaped arrays as parameters for the incoming distribution. However, if you have explicitly specified the distribution on all relevant formal parameters, then you can disable auto–cloning with -MP:clone=off. The consistency checking of the distribution between actual and formal parameters is not affected by this flag, and is always enabled.

-MP:check_reshape={on, off} (default off)

Enables generation of the run–time consistency checks across procedure boundaries when passing reshaped arrays (or portions thereof) as parameters.

Data Distribution: *Query Routines/1*

extern INT64 dsm_numthreads (void* array, INT64 dim)

Called with a distributed array and a dimension number. Returns the number of threads in that dimension.

extern INT64 dsm_chunksize (void* array, INT64 dim)

Returns the chunk size (ignoring partial chunks) in the given dimension for each of block, cyclic(..), and star distributions.

extern INT64 dsm_this_chunksize (void* array,INT64 dim,INT64 index)

Returns the chunk size for the chunk containing the given index value for each of block, cyclic(..), and star. This value may be different from dsm_chunksize due to edge effects that may lead to a partial chunk.

extern INT64 dsm_rem_chunksize (void* array,INT64 dim,INT64 index)

Returns the remaining chunk size from index to the end of the current chunk, inclusive of each end point. Essentially it is the distance from index to the end of that contiguous block, inclusive.

extern INT64 dsm_this_startingindex (void* array,INT64 dim,INT64 index)

Returns the starting index value of the chunk containing the supplied index.

Data Distribution: *Query Routines/2*

extern INT64 dsm_numchunks (void* array, INT64 dim)

Returns the number of chunks (including partial chunks) in given dim for each of block, cyclic(..), and star distributions.

extern INT64 dsm_this_threadnum(void* array,INT64 dim,INT64 index)

Returns the thread number for the chunk containing the given index value for each of block, cyclic(..), and star distributions.

extern INT64 dsm_distribution_block (void* array, INT64 dim)
extern INT64 dsm_distribution_cyclic (void* array, INT64 dim)
extern INT64 dsm_distribution_star (void* array, INT64 dim)

Boolean routines to query the distribution of a given dimension.

extern INT64 dsm_isreshaped (void* array)

Boolean routine to query whether reshaped or not.

extern INT64 dsm_isdistributed (void* array)

Boolean routine to query whether distributed (regular or reshaped) or not.