Supercomputing with ZPL and Other Approaches

ZPL Classic is fine for base computations, but serious scientific computations need more control. Also, we summarize other popular parallel programming languages

Sorting Solution from Homework

- Code for ranking sort of [1,1..n] S : integer;

[*,1..n] RowFlood := >>[1,1..n]S; --*Replicate S as row*

[1..n,*] ColFlood := >>[1..n,1]S#[Index2,Index1]; --*Find S^T, dup as col*

[1,1..n] Rank := +<<[1..n,1..n](ColFlood <= RowFlood); --*Compare,Add*

3	1	4	5	9	2
3	1	4	5	9	2
3	1	4	5	9	2
3	1	4	5	9	2
3	1	4	5	9	2
3	1	4	5	9	2

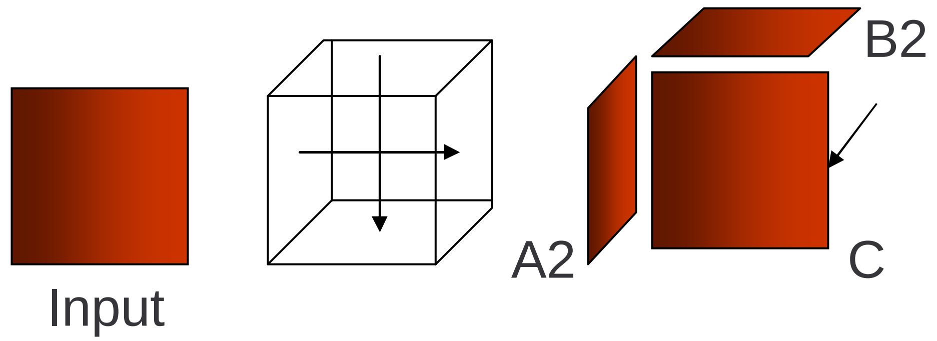
3	3	3	3	3	3
1	1	1	1	1	1
4	4	4	4	4	4
5	5	5	5	5	5
9	9	9	9	9	9
2	2	2	2	2	2

$$\begin{array}{c|cccccc|}
 & 3 & 1 & 4 & 5 & 6 & 2 \\
 & 3 & 1 & 4 & 5 & 9 & 2 \\
\hline
3 & 1 & 0 & 1 & 1 & 1 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 1 \\
4 & 0 & 0 & 1 & 1 & 1 & 0 \\
5 & 0 & 0 & 0 & 1 & 1 & 0 \\
9 & 0 & 0 & 0 & 0 & 1 & 0 \\
2 & 1 & 0 & 1 & 1 & 1 & 1 \\
\hline
\end{array}$$

Problem Space Promotion

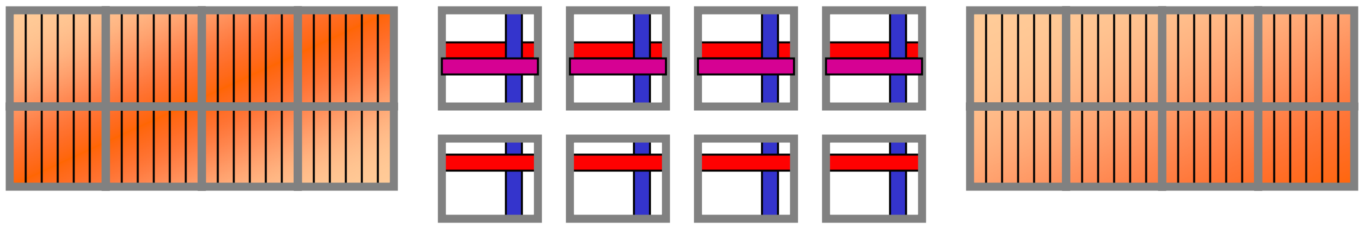
- The ranking sort is an instance of a new programming paradigm in ZPL
    - Problem Space Promotion is to solve a D dimensional problem in a dimension D'>D using floods to avoid explicit creation of data structures
- Other PSPs mentioned so far: 3D MM

```
[IstarK]   A2  := A#[Index1,Index2];
[starJK]   B2  := B#[Index2,Index1];
[IJstar]    C  := +<<[IJK](A2*B2);
```



PSP

- Explicit N-Body and other ‘all pairs’ computations work well
- PSP works well because ZPL floods are space efficient
    - The ops are the same, but the data motion is less and it benefits from caching



Where We Are, and the Plan

- ZPL Classic + WYSIWYG is plenty powerful for producing *quality* parallel solutions of serious scientific computations
- Large applications--protein folding, galaxy simulation, etc.--require control over data placement and processor work assignment
- Complete ZPL has facilities for managing those tasks … we look at
    - Grids, Distributions
    - Grid variables
    - FFT example

Recall Free Variables

- Free variables contrast with scalar variables:
  - Declarations

    ```
    var x, y : integer;          -- scalar declarations
    free var fx, fy : integer; -- free variable declarations
    ```

  - Semantics
    - Scalar Vars--one copy on each processor, but they act like one global variable (coherent)
    - Free Vars--one copy on each processor, but they behave independently (not coherent)
  - Uses

    ```
    Add globally:  [R] x := +<<A;  -- Global Reducation
    Add locally:   [R] fx += A;     -- Accumulate local values
    Add globally:      x := +<<fx;  -- Reduction extended to free
    ```

Local Computation with Free Variables

- What's happening with the free variable?

```
free var freeSum : integer = 0;
[R]       freeSum += A;
```

All elements of A covered by the region are processed over that portion of the region local to each processor … adding all the while

```
free var localMax : integer = MININTEGER;
[R] if A > localMax then
        localMax := A;
      end;
```

The shattered case; generally assignment to scalars is illegal

Sorting Columns: Return to Homework

In the observations of the homework, how could we sort the columns? Start with vector of arrays

```
var Obs : [1..n] array [1..m] of float; -- Vector of arrays
free var ftemp : integer;
    [1..n] for i := 1 to m-1 do              -- Simple exchange sort
              for j := 2 to m do
                   if Obs[i]>Obs[j] then
                     ftemp := Obs[i]; -- Free variable is needed
                     Obs[i] := Obs[j]; -- Indexed op always OK
                     Obs[j] := ftemp; -- Free variable is needed
                   end;
               end;
          end;
```

Each processor does the vectors it stores

A vector of arrays may be a good DS for this problem, but not always

Grid Dimensions to the Rescue

- Grid dimensions (::) are “between” flood (*) and range (..) and can the seen as extending the free concept to flood
- Grid dimensions associate 1 value *per processor* unlike flood with 1 value for all processors
- For example

```
var A : [::, 1..n] integer;  -- Place n elements on each proc
var A : [0..numLocales()-1,1..n] integer; -- Like above
```

One great use of grid dimensions is to control computation over regular arrays declared with ranges

Computing Over Grid Dimensions

Computing over grid dimensions lets a dimension act as an array of arrays. Back to HW again

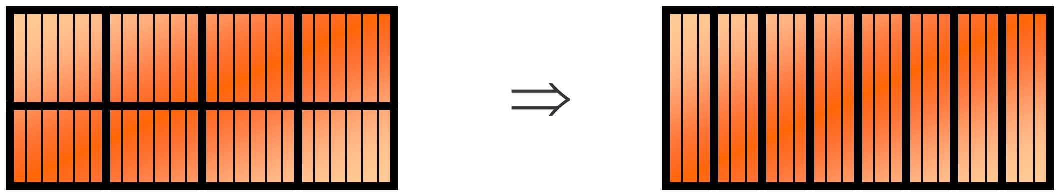
```
var Ob : [1..m,1..n] float;
free var ftemp, i, j : integer;                    -- Simple exchange sort
           [::,1..n] for i := blockLocalLo(Ob,1) to blockLocalHi(Ob,1)-1 do
                     for j := blockLocalLo(Ob,1)+1 to blockLocalHi(Ob,1) do
                       if Ob[i]>Ob[j] then
                          ftemp := Ob[i]; -- Free variable is needed
                          Ob[i] := Ob[j];
                          Ob[j] := ftemp; -- Free variable is needed
                        end;
                      end;
                     end;
```

Range over the column as with vector of arrays; no change of structure

NB blockLocalLo is presently reglo(R, dim)
blockLocalHi is presently reghi(R, dim)

But our Original Formulation was 2x4

- *The fine print*: This use of grid dimensions is local computation, implying that all the values have to be on the same processor, but the original grid configuration had multiple processors in a column



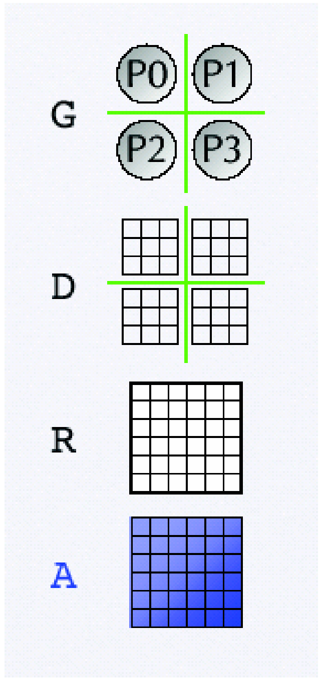
- What to do when the different parts of the computation want different proc arrangements?

ZPL allows processor allocations to be changed … though this problem might not be worth it

ZPL's Meta Concepts

Processor allocations (grids) and distributions can be changed by programmer on-the-fly

- New concepts: `grid` and `distribution`
- There is a hierarchy of concepts



`grid`

`distribution`

`region`

`array`

The issue is managing the data and work allocations dynamically

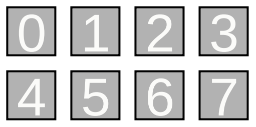
Grids

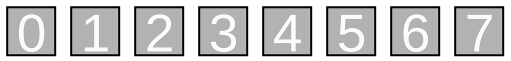
A grid is an logical arrangement of processors used as an abstraction for allocations

Declare

grid G1 = [1..2,1..p/2]; -- original proc grid

grid G2 = [1,1..p]; -- desired proc grid





which are arrangements we have and the one we want

- The plan is to reallocate the array so that the columns are on a single processor

We have to say how we want regions assigned

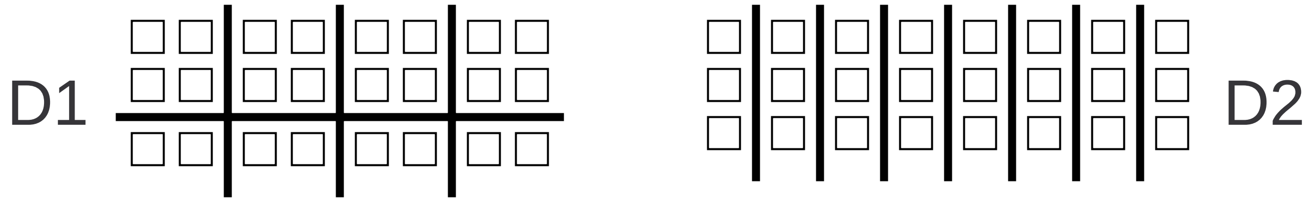
Distributions

Distributions say how a regions are distributed across a grid

Declare

```
distribution D1 : G1 = [blk(1,m),blk(1..n)];
distribution D2 : G2 = [blk(1,m),blk(1..n)];
```

which allocates all array elements by blocks in each dimension of the grids specified



Now we must assign the regions

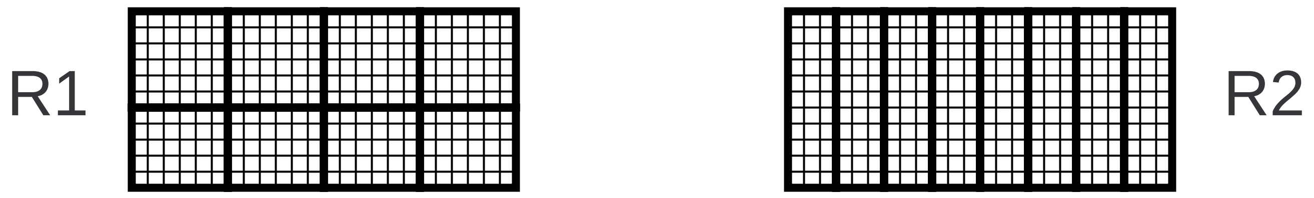
Regions ...

Regions as defined so far take the default distribution, but distributions can be specified

Declare

```
region R1 : D1 = [1..m, 1..n];
       R2 : D2 = [1..m, 1..n];
```

which distributes all of the indices as we need



Arrays

Arrays are specified in the usual way ...

Declare

```
var A : [R1] double;
    B : [R2] double;
```

- All of this preparation is set up for assigning A to B to reallocate the data:

  ```
  [1..m,1..n] B := A#[Index1,Index2];
  ```

  which requires the remap because there may be communication and WYSIWYG needs to expose that

The set up is all declarations taking no time, only thinking … restructuring is conceptually trivial; it should also be easy

Grid, Distribution, Region, Array Hierarchy

All of the meta concepts are “first class,” meaning they can be variables and manipulated by the programmer … strong control

We set up syntax to declare variables

```
var G : grid <..,..> = [1,1..p];  -- 2D arrangement of procs
    D : [G] distribution<block,block> = [blk(1,m),blk(1,n)];
    R : [D] region = [1..m, 1..n];
    A : [R] float;
```

Interesting technical problem: What happens to the data when you reallocate the indices?

```
<==  Destructive assignment, data lost
<=#  Preserving assignment, data save by index
```

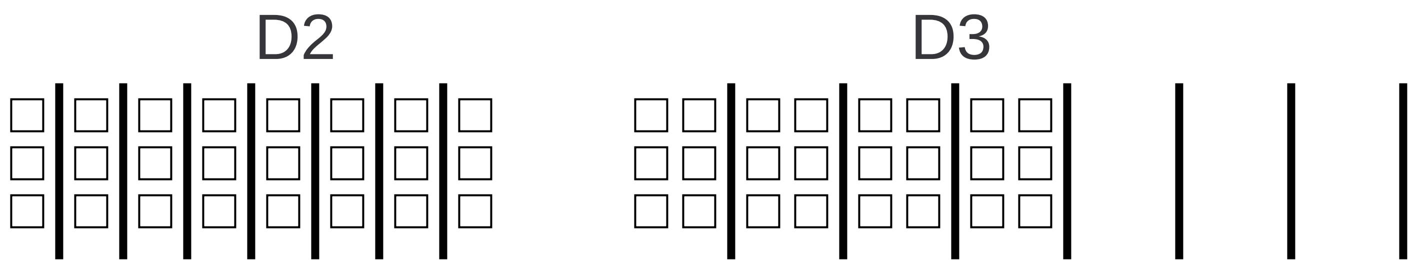
Restructuring Distribution

- For example ...

  Recall

  distribution D2 : G2 = [blk(1,m),blk(1,n)]; -- B’s distribution

  D3 : G2 = [blk(1,m),blk(1,2*n]; -- New dist



The first n elements are allocated to the left half of the processors in D3

Change A Region’s Distribution

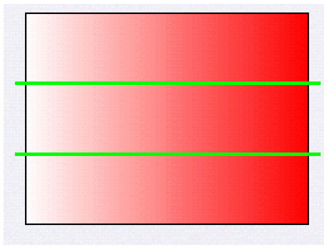
```
var G : grid <..,..> = [1,1..p];
       D : [G] distribution = [block,block]; --generic allocation
      D1 : [G] distribution = [blk(1,m),blk(1,n)];-- std allocation
      D2 : [G] distribution = [blk(1,m),blk(1,2*n)]; -- left alloc
       R : [D] region = [1..m,1..n];            -- R covers procs
    A,B  : [R] integer;                          -- Actual arrays
      D <== D1;                 -- Bind an initial allocation
      …
      D <=# D2;                 -- Shift array elements to left
      R <== [1..m,1..2*n];    -- Change region, flush data
```

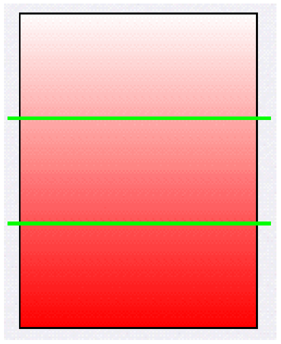
Change distribution, flush data, fast

Change distribution, save data, need comm

Abstractions Give New Algorithms

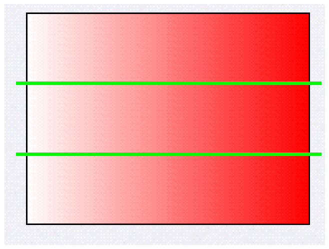
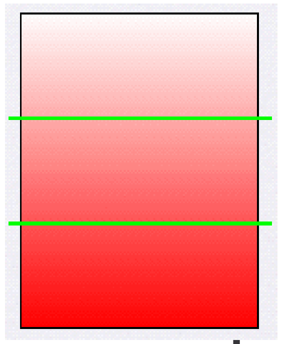
- 2D FFT is a standard scientific building block
- Solution: 1D FFT on rows, transpose, 1D FFT on columns (now rows); allocate so "butterfly" is local



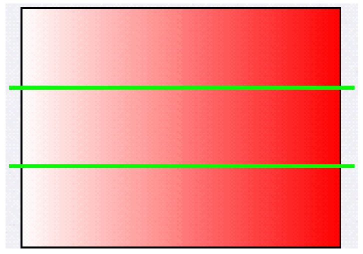
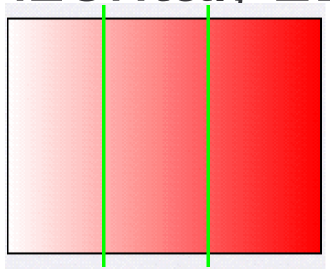


ZPL Abstractions Give New Algorithms

- 2D FFT is a standard scientific building block
- Solution: 1D FFT on rows, transpose, 1D FFT on columns (now rows); allocate so “butterfly” is local

- Alternative: 1D FFT on rows, change the grid from vertical to horizontal, 1D FFT on columns

Faster!

ZPL Summary

- We’ve taught perhaps 85% of the language
- Good News
    - Global view allows high level solution; clean programs
    - CTA + WYSIWYG let you know what’s going on
    - Fast programs can be written quickly; portable everywhere
- Bad News
    - The language may be intuitive (or not), but it *is* different
    - Think of solutions by manipulating arrays, not step-at-a-time implementations … different algorithms are relevant
    - ZPL is not yet vendor supported … open source means you fix your own bugs
- ZPL is a creative response to parallel prog’g

Parallel Language

There have been easily 100 parallel languages proposed, but what's the point? Will anyone adopt a new language even if it's wondrous?

Issues:

- Learning Curve … if it *really* helps it won't look like C++
- Software investment … there are millions of lines of code
- Existing codes are trusted … validation is a serious concern
- User community … discipline scientists have little deep knowledge about computing; crude use of MATLAB is limit so who does the programming?

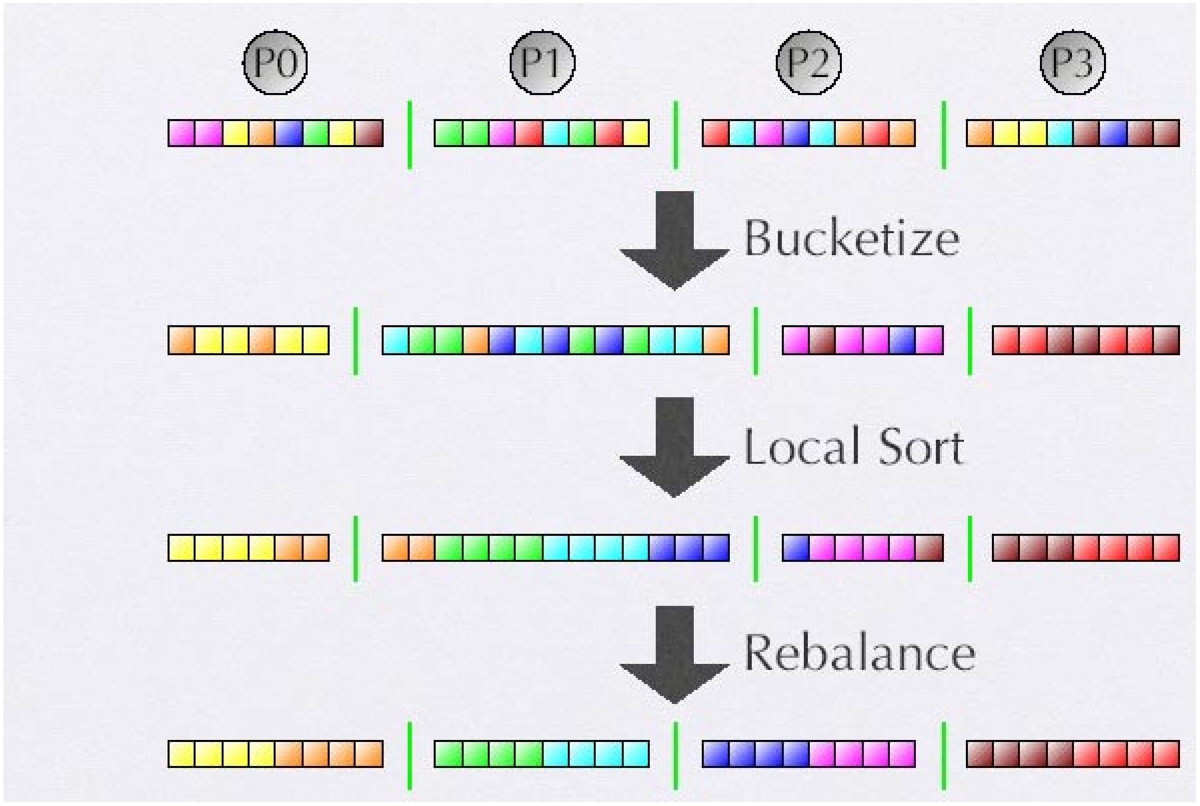
<Discuss>

Break

- 10 minutes

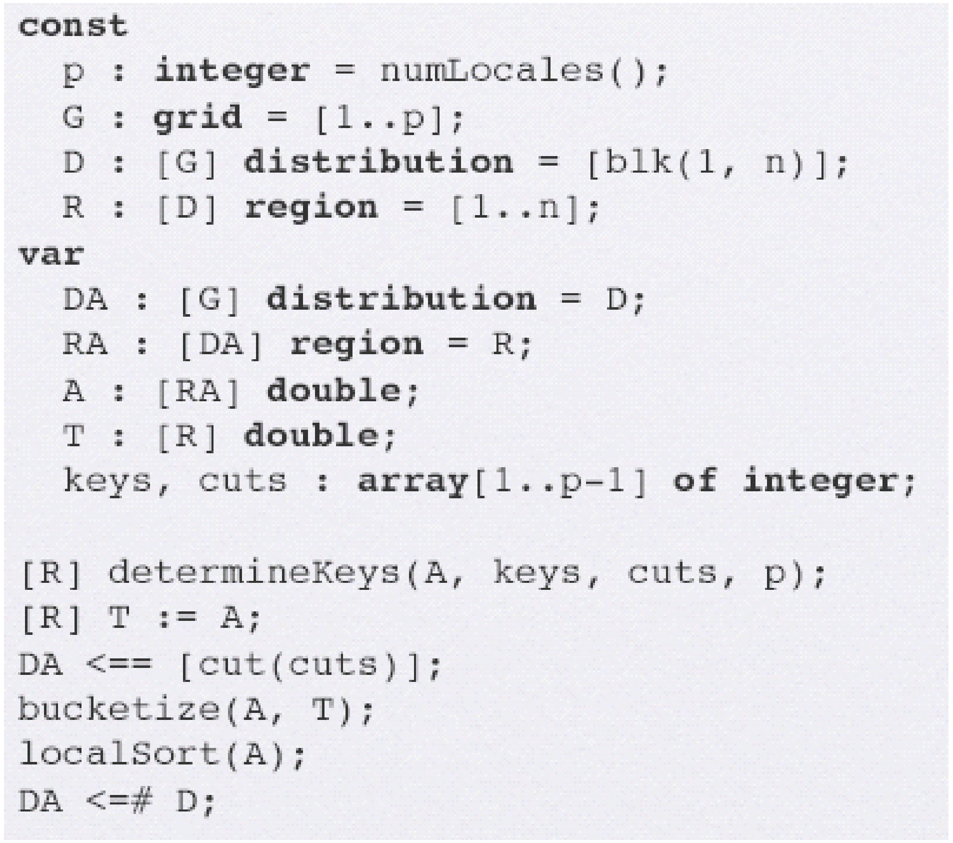
Sample Sort Logic

Bucketize" means send data to processor where it will probably end up



Sample Sort in ZPL

“cut” is an alternative block distribution given by vector of integers, the highest item alloc’t’ed

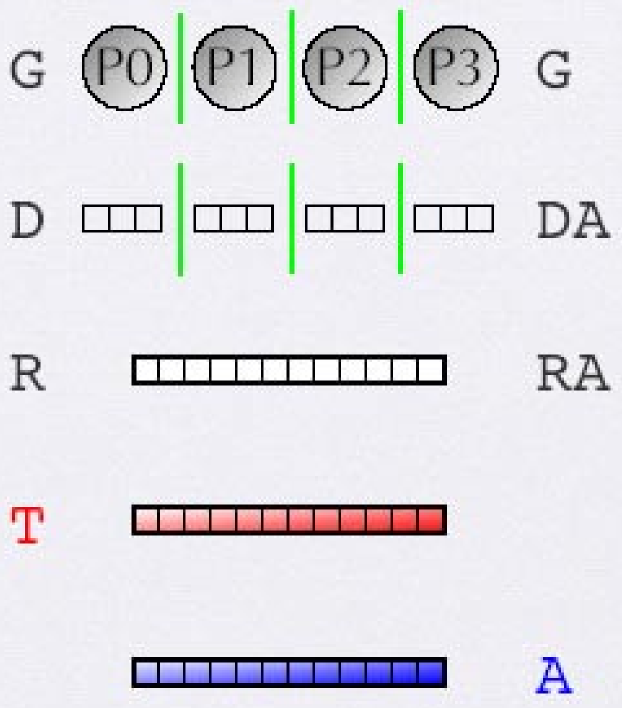


```
const
   p : integer = numLocales();
   G : grid = [1..p];
   D : [G] distribution = [blk(1, n)];
   R : [D] region = [1..n];
var
   DA : [G] distribution = D;
   RA : [DA] region = R;
   A : [RA] double;
   T : [R] double;
   keys, cuts : array[1..p-1] of integer;

[R] determineKeys(A, keys, cuts, p);
[R] T := A;
DA <== [cut(cuts)];
bucketize(A, T);
localSort(A);
DA <=# D;
```

Schematic of Constants and Variables

Set up structures to prepare for redistribution

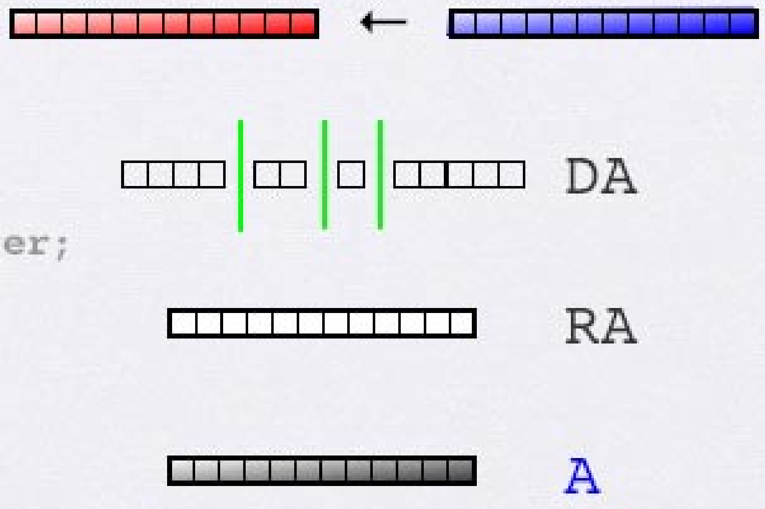
```
const
  p : integer = numLocales();
  G : grid = [1..p];
  D : [G] distribution = [blk(1, n)];
  R : [D] region = [1..n];
var
  DA : [G] distribution = D;
  RA : [DA] region = R;
  A : [RA] double;
  T : [R] double;
  keys, cuts : array[1..p-1] of integer;

[R] determineKeys(A, keys, cuts, p);
[R] T := A;
DA <== [cut(cuts)];
bucketize(A, T);
localSort(A);
DA <=# D;
```



Compute How to Redistribute

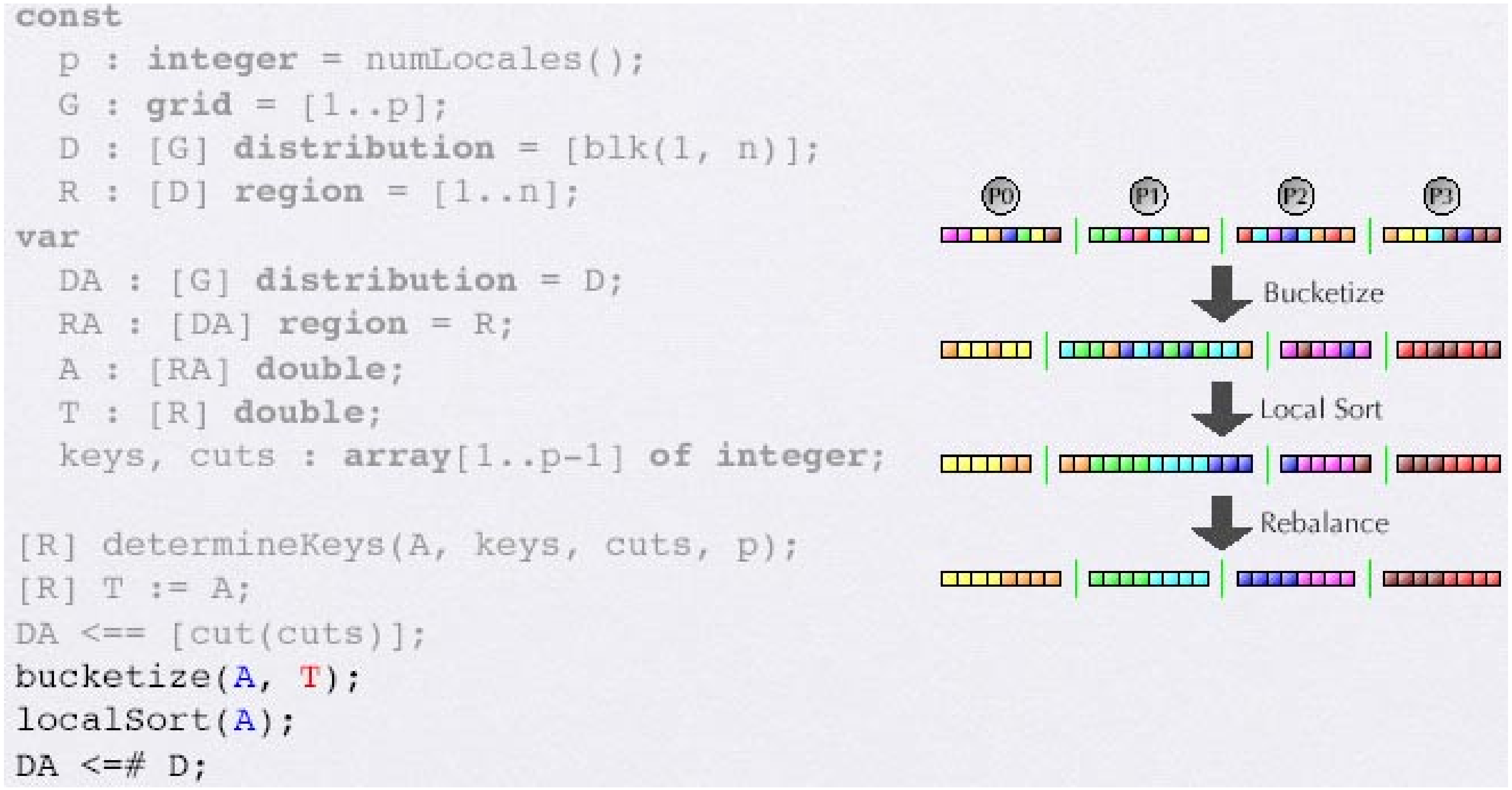
Distributions change at “bucketize” time by cuts

```
const
   p : integer = numLocales();
   G : grid = [1..p];
   D : [G] distribution = [blk(1, n)];
   R : [D] region = [1..n];
var
   DA : [G] distribution = D;
   RA : [DA] region = R;
   A : [RA] double;
   T : [R] double;
   keys, cuts : array[1..p-1] of integer;

[R] determineKeys(A, keys, cuts, p);
[R] T := A;
DA <== [cut(cuts)];
bucketize(A, T);
localSort(A);
DA <=# D;
```



Finish Up

The final “scooch” is simply a distribution change



```
const
  p : integer = numLocales();
  G : grid = [1..p];
  D : [G] distribution = [blk(1, n)];
  R : [D] region = [1..n];
var
  DA : [G] distribution = D;
  RA : [DA] region = R;
  A : [RA] double;
  T : [R] double;
  keys, cuts : array[1..p-1] of integer;

[R] determineKeys(A, keys, cuts, p);
[R] T := A;
DA <== [cut(cuts)];
bucketize(A, T);
localSort(A);
DA <=# D;
```

If You’re Not Using ZPL, Then What?

- Practical parallel programming is done in
  - Message passing libraries (MPI, PVM) for cluster machines and large parallel processors (CTA cases)
  - OpenMP library for shared memory SMP type multiprocessors
  - Combination, because large machines are becoming collections of SMPs
  - Very rarely, a proper parallel language
- Libraries augment a scalar language or possibly Fortran 90/95/…
- Libraries are parallel assembly languages … programmers create their own abstractions

Message Passing

- Two libraries dominate...
  - PVM (Parallel Virtual Machine) Oak Ridge National Lab
  - MPI (Message Passing Interface) Consortium
- Libraries provide mainly communication routines but there’s other stuff
  - Initialization and process spawning
  - Synchronization, timers, etc.
  - Collective Communication, i.e. reduction, broadcast
- Libraries are widely available, vendor provided, so they are “portable”

MM in MPI -- 1

A “master--slave” solution

```
MPI_Status status;
main(int argc, char **argv) {
int numtasks,                   /* number of tasks in partition */
   taskid,                      /* a task identifier */
   numworkers,                  /* number of worker tasks */
   source,                      /* task id of message source */
   dest,                        /* task id of message destination */
   nbytes,                      /* number of bytes in message */
   mtype,                       /* message type */
   intsize,                     /* size of an integer in bytes */
   dbsize,                      /* size of a double float in bytes */
   rows,                        /* rows of matrix A sent to each worker */
   averow, extra, offset,        /* used to determine rows sent to each
     worker */
   i, j, k,                     /* misc */
   count;
double a[NRA][NCA],             /* matrix A to be multiplied */
     b[NCA][NCB],               /* matrix B to be multiplied */
     c[NRA][NCB];               /* result matrix C */
```

MM in MPI -- 2

```
intsize = sizeof(int);
dbsize = sizeof(double);

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
numworkers = numtasks-1;

/****************************** master task **************************************/
if (taskid == MASTER) {
for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      a[i][j]= i+j;
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j]= i*j;
```

NRB? Wouldn’t ‘Index1’ be better?

MM in MPI -- 3

```
/* send matrix data to the worker tasks */
  averow = NRA/numworkers;
  extra = NRA%numworkers;
  offset = 0;
  mtype = FROM_MASTER;
  for (dest=1; dest<=numworkers; dest++) {
    rows = (dest <= extra) ? averow+1 : averow;
    MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
    count = rows*NCA;
    MPI_Send(&a[offset][0], count, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
    count = NCA*NCB;
    MPI_Send(&b, count, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);

    offset = offset + rows;
    }
```

MM in MPI -- 4

```
/* wait for results from all worker tasks */
  mtype = FROM_WORKER;
  for (i=1; i<=numworkers; i++)    {
    source = i;
    MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
    count = rows*NCB;
    MPI_Recv(&c[offset][0], count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD,&status);
}
/***************************** worker task **************************************/
if (taskid > MASTER) {
  mtype = FROM_MASTER;
  source = MASTER;
  MPI_Recv(&offset, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
  MPI_Recv(&rows, 1, MPI_INT, source, mtype, MPI_COMM_WORLD, &status);
  count = rows*NCA;
  MPI_Recv(&a, count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);
```

MM in MPI -- 5

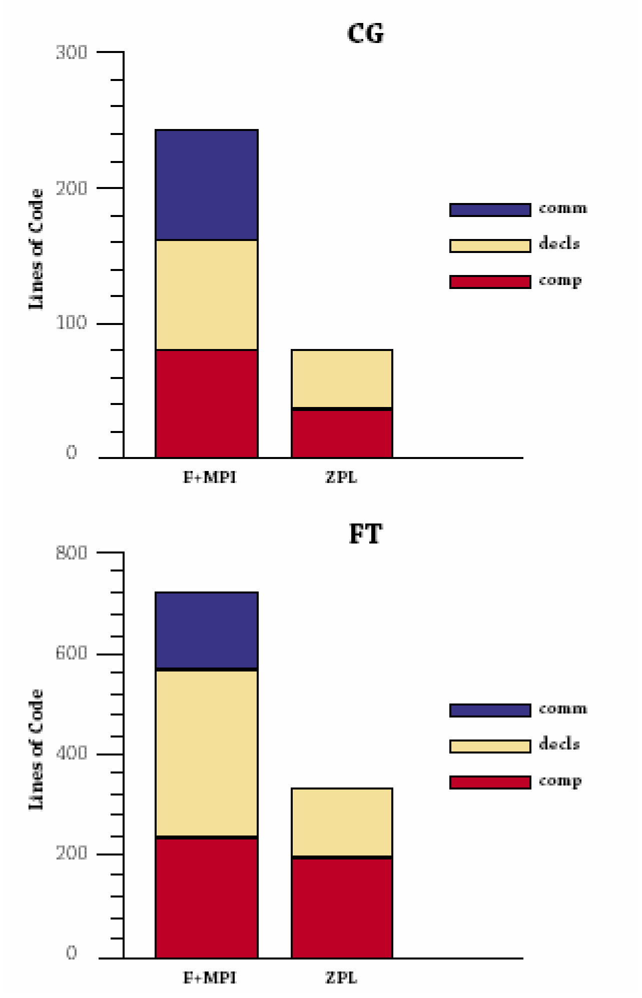
```
count = NCA*NCB;
 MPI_Recv(&b, count, MPI_DOUBLE, source, mtype, MPI_COMM_WORLD, &status);

 for (k=0; k<NCB; k++)
  for (i=0; i<rows; i++) {
   c[i][k] = 0.0;
   for (j=0; j<NCA; j++)
    c[i][k] = c[i][k] + a[i][j] * b[j][k];
   }

 mtype = FROM_WORKER;
 MPI_Send(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
 MPI_Send(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD);
 MPI_Send(&c, rows*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD);

 }  /* end of worker */
```

Actual Multiply

91 “Net” Lines

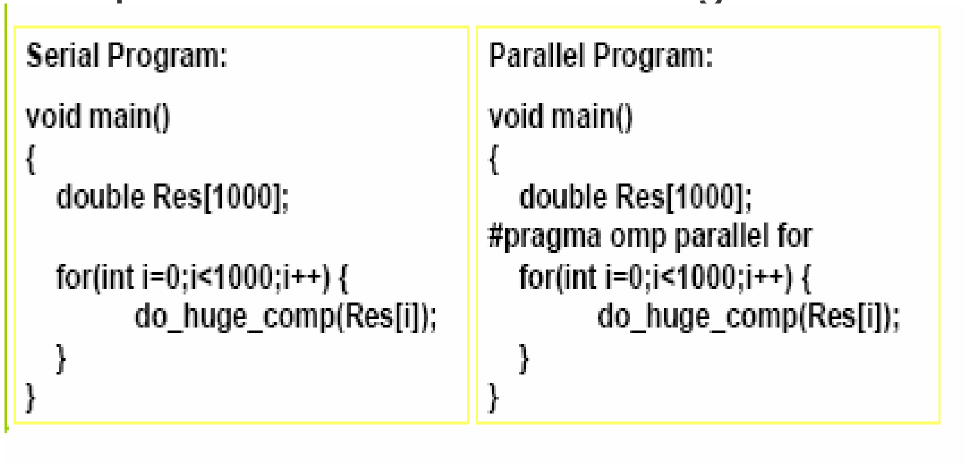
Level of Work

Lines of code is a questionable metric for productivity, but ...



OpenMP

- OpenMP is a standard API for threading on shared memory multiprocessors … suitable for SMPs
- Strong vendor support with applications seeming to be commercial rather scientific
    - Standard scientific libraries are available
    - OpenMP used for threading with MPI on hybrid machines



```
Serial Program:

void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
          do_huge_comp(Res[i]);
   }
}
```

```
Parallel Program:

void main()
{
   double Res[1000];
#pragma omp parallel for
   for(int i=0;i<1000;i++) {
          do_huge_comp(Res[i]);
   }
}
```

Languages

- Parallel language design has been a popular indoor sport for decades … most academic, few seriously implemented
- HPF (High Performance Fortran)
    - Most visible effort of the last decade
    - Well funded, strongly backed by vendors, community developed with substantial consensus
    - Extended Fortran 90 (companion Rice version extended F77) by adding compiler directives to orchestrate ||ism
    - Several compilers implemented (most of) initial design, both academic and commercial
    - Many applications efforts at U's, labs (Japanese successful)
    - Global language like ZPL

MM in HPF

```
 PROGRAM ABmult
     IMPLICIT NONE
     INTEGER, PARAMETER :: N = 100
     INTEGER, DIMENSION (N,N) :: A, B, C
     INTEGER :: i, j

!HPF$ PROCESSORS square(2,2)
!HPF$ DISTRIBUTE (BLOCK,BLOCK) ONTO square :: C
!HPF$ ALIGN A(i,*) WITH C(i,j)
!    replicate copies of row A(i,*)
!    onto processors which compute C(i,j)

!HPF$ ALIGN B(*,j) WITH C(i,j)
!    replicate copies of column B(*,j))
!    onto processors which compute C(i,j)

     DO i = 1, N
       DO j = 1, N
!        All the work is local due to ALIGNs
         C(i,j) = DOT_PRODUCT(A(i,:), B(:,j))
       END DO
     END DO

     END
```

And The Verdict Is ...

HPF efforts ended in US; some still overseas

- Why did a concerted effort not succeed?
  - Funding, community & vendor interest not issues
  - Answers are necessarily opinion, mine are …
    - Can a language be designed by a committee?
    - HPF chose not to adopt an abstract machine model
    - Directives were taken as “suggestions” to the compiler

    ∴Programmers were unable to know what was happening
  - Debate continues …
- Undaunted, feds are funding 3 new efforts

Another Recent Effort ...

Co-Array Fortran

- Developed within Cray (originally F--) by Numrich&Reed
- Motivated to use T3D/T3E's shmem facilities
- Add's a processor "co-dimension" to the arrays of F95

```
REAL, DIMENSION (N) [*] :: X,Y   !Declare 2 size n vectors
X(:) = Y(:) [PE]      !If PE is same on all images, copy Y to X
```

- Also has a few collective operations, synch. primitives
- CAF provides a clean way to manage (shmem) communication in a "local view" language … machine model is CTA
- Cray supports CAF

MM in CoArray Fortran

```
real,dimension(n,n)[p,*] :: a,b,c

do k=1,n
 do q=1,p
    c(i,j)[myP,myQ] = c(i,j)[myP,myQ]
                          + a(i,k)[myP, q]*b(k,j)[q,myQ]
 enddo
enddo
```

Global Address Space (GAS) Languages

- Global shared memory's difficulties motivated global address space language … coherence controlled by programmer through local view
  - UPC (Universal Parallel C) Center for CS, MD
  - Titanium (a Java Dialect) Berkeley
  - Co-Array Fortran
- Titanium's "single" is opposite of ZPL's "free" and defaults are opposite
- Whereas ZPL prohibits comm in shattered control (recall shattered @) GAS languages encourage it as the main mechanism

Summary on Languages

- There's a bunch of other languages that have been implemented, but they are mostly of academic interest (like ZPL)
- Programmers with large problems to solve are reduced to writing message passing code
- Libraries exist that package communication for moving arrays around as a unit--saves work but all the rest of the programming (and the optimizations) require low level scalar programming

Built-in Constants

```
extern constant PROCESSORS : integer;                               -- number of processors
extern prototype numLocales() : integer;
extern free prototype localeID() : integer;
extern prototype GRIDPROCS(grd : grid; dim : integer) : integer;   -- grid query functions
extern prototype GRIDPROC(grd : grid; dim : integer) : integer;
extern prototype blk(lo, hi : integer) : integer;                   -- built-in distributions
extern prototype cut(a : generic) : integer;
extern free prototype reglo(reg : region; dim : integer) : integer;  -- region query functions
extern free prototype reghi(reg : region; dim : integer) : integer;
extern prototype _ARR_REG(inout a : genericensemble) : region;-- Regions, Grids
extern prototype _ARR_DIST(inout a : genericensemble) : distribution;       -- Distributions
extern prototype _ARR_GRD(inout a : genericensemble) : grid;
extern prototype _REG_DIST(inout r : region) : distribution;
extern prototype _REG_GRD(inout r : region) : grid;
extern prototype _DIST_GRD(d : distribution) : grid;
extern prototype open(s1, s2 : string) : file;                      -- file i/o
extern prototype eof(f: file) : integer;
extern prototype close(inout f : file) : integer;
extern prototype bind_write_func(inout e : genericensemble; f : generic) : integer;
extern prototype bind_read_func(inout e : genericensemble; f : generic) : integer;
extern prototype unbind_write_func(inout e : genericensemble) : integer;
extern prototype unbind_read_func(inout e : genericensemble) : integer;
extern type timer = opaque;                                         -- built-in timers
extern prototype ClearTimer(inout t : timer);
extern prototype StartTimer(free inout t : timer; sync : boolean);
extern prototype StopTimer(free inout t : timer);
extern free prototype ReadTimer(free inout t : timer) : double;
```

Citations

B. Chamberlain, E Lewis, L. Snyder, “Problem Space Promotion,” Proc. of International Conference on Supercomputing

S. Deitz, B. Chamberlain, L. Snyder, “Abstractions for Dynamic Data Distributions,” IEEE Workshop on High-level Parallel Programming Models and Supportive Environments, 2004

Parallel Computing Languages for example ...

http://www.cs.rit.edu/~ncs/parallel.html#languages

Project Parameters

[The project’s not yet completely written out, but will be posted on the Web this week.]

- Due date: 14 March 2005
- Required: Write and experiment with a substantial ZPL program; write a short (1-3 pages) report on results such as WYSIWYG analysis, speedup, etc.
- A few (2-4) problem domains will be described with a basic computation, and suggested extensions [you can pick your own, but check 1st]
- Get code running on cluster, measure base computation and enhancements; cycle to improve