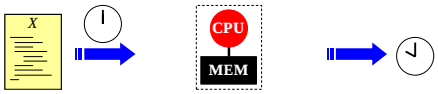
ZPL: A Region-Based Parallel Programming Language

Brad Chamberlain
CSE 505
December 5, 2001

The ZPL Project
University of Washington

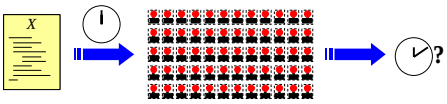
What is Parallel Computing?

Parallel Computing: Using multiple processors in cooperation to perform a computation.



What is Parallel Computing?

Parallel Computing: Using multiple processors in cooperation to perform a computation.



Ideally: Using *p* processors would allow…
…a program to run *p* times faster
…a problem that is *p* times as big to be solved

Reality: This is difficult to achieve

Why is Parallel Programming Hard?

- All the challenges of sequential programming
- Plus…
  - distribution of computation
  - distribution of data
  - data transfer between processes
  - synchronization between processes
  - potential for race conditions, deadlock, etc.
  - challenging to debug effectively

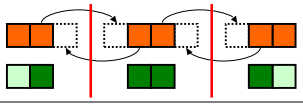
How is Parallel Programming Done?

- Parallel Languages
  - designed to ease parallel programming burdens
- Parallel Libraries
  - support for computing using multiple processes
  - *or* canned parallel implementations of routines
- Parallelizing Compilers?
  - less and less as time goes on
  - difficult to automatically turn a good sequential program into a good parallel one

Sample Computation

- Given two vectors `a` and `b`…
- Replace interior elements of `b` with the sum of their neighbors in `a`

- Parallel implementation requires communication to access neighboring values



What is MPI?

- "Message Passing Interface"
- Library for inter-process data transfer
  - sends/receives
  - broadcasts, reductions
  - scatters, gathers
- Portable!
- The *de facto* standard for parallel programming

MPI Example (C)

```
MPI_Comm_Size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_Rank(MPI_COMM_WORLD, &index);
vals_pp = (1000/nprocs)+2;

double a[vals_pp], b[vals_pp];

if (index < nprocs-1) {
  MPI_Send(&(a[vals_pp-2]), 1, MPI_DOUBLE, index+1, 1, MPI_COMM_WORLD);
}
if (index > 0) {
  MPI_Send(&(a[1]), 1, MPI_DOUBLE, index-1, 2, MPI_COMM_WORLD);
  MPI_Recv(&(a[0]), 1, MPI_DOUBLE, index-1, 1, MPI_COMM_WORLD);
}
if (index < nprocs-1) {
  MPI_Recv(&(a[vals_pp-1], 1, MPI_DOUBLE, index+1, 2, MPI_COMM_WORLD);
}
for (i = 1; i < vals_pp-1; i++) {
  b[i] = a[i-1] + a[i+1]
}
```

What is HPF?

- "High Performance Fortran"
- Extensions to Fortran 90 to support parallelism
- Directives used to indicate…
  - …array distribution
  - …array alignment
  - …non-obvious parallelism

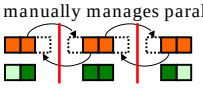
HPF Example

```
      REAL a(1000), b(1000)
!HPF$ DISTRIBUTE a(BLOCK)
!HPF$ ALIGN b(:) WITH a(:)

      …

      b(2:999) = a(1:998) + a(3:1000)
```

Local-view vs. Global-view

Local-view:

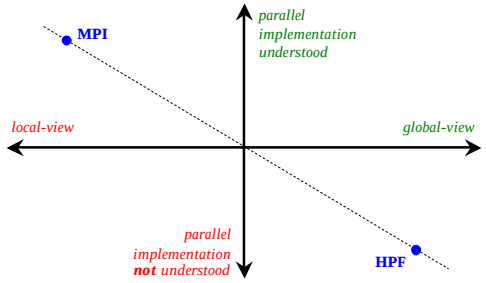
- code describes per-processor behavior
- user manually manages parallel details



Global-view:

- code describes problem as a whole
- compiler manages parallel details

Parallel Programming Stereotypes



Motivation for our work

Goal: To provide language support for parallel programming

Sub-goal: Preserve characteristics of successful sequential languages:
- performance
- portability
- clarity (global view)
- performance model (implementation understood)

The Challenge: Traditionally, there are tensions between these characteristics

Outline

Parallel Computing Overview

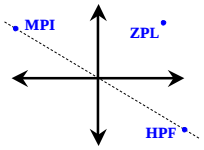
Introduction to Regions and ZPL
- Hierarchical Regions
- Sparse Regions
- Conclusion

What is ZPL?

- A parallel programming language
  - array-based
  - global-view
  - portable
- Developed at the University of Washington
- Publicly available on the web:
  www.cs.washington.edu/research/zpl

What makes ZPL unique?

Regions – a novel means of specifying parallel computation
- support a global view
- *Yet*, expose the parallel implementation



Regions

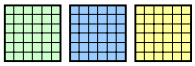
Regions: index sets that…

…can be named

```
region    R = [1..n  ,1..n  ];
       BigR = [0..n+1,0..n+1];
```

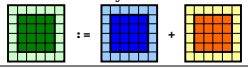
…are used to declare arrays

```
var A, B, C:[BigR] integer;
```

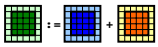


…specify indices for a statement's array references

```
[R] A := B + C;
```



Regions Eliminate Redundancy



C:
```
for (i=1; i<=n; i++) {
  for (j=1; j<=n; j++) {
    A[i][j] = B[i][j] + C[i][j];
  }
}
```

F90:
```
A(1:n,1:n) = B(1:n,1:n) + C(1:n,1:n)
```

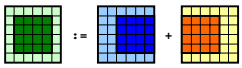
ZPL:
```
[1..n,1..n] A := B + C;
```
or:
```
[R] A := B + C;
```

Array Operators

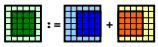
Array Operators: modify the current region's index set for an array reference

E.g., @-operator translates indices by a *direction*:

```
direction east = [0, 1];
          west = [0,-1];

[R] A := B@east + C@west;
```



Regions Emphasize Differences



C:
```
for (i=1; i<=n; i++) {
  for (j=1; j<=n; j++) {
    A[i][j] = B[i][j+1] + C[i][j-1];
  }
}
```

F90: `A(1:n,1:n) = B(1:n,2:n+1) + C(1:n,0:n-1)`

ZPL: `[1..n,1..n] A := B@[0,1] + C@[0,-1];`

or: `[R] A := B@east + C@west;`

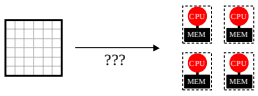
Array Operator Overview

operator	*effect*	*sample*
@	Translate references	`A@east`
flood	Replicate values across array dimensions	`>>[i,] A`
reduction	Collapse array dimensions	`+<<[R] A`
remap	Arbitrarily permute, gather, scatter, …	`A#[X,Y]`

Parallel Interpretation of Regions

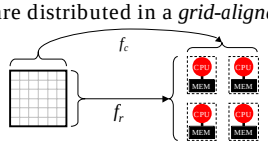
Each region's indices are distributed across the processor set

- defines data distribution
- defines work distribution



Region Distribution

1) Regions are distributed in a *grid-aligned* manner.



2) *Interacting regions* are distributed identically

`[R] A := B + C;`

ZPL's Performance Model

Distribution rules imply a performance model:

- Traditional operators are perfectly parallel

  `[R] A := B + C;`

- Array operators indicate communication of a particular style

  `[R] A := B@east + C@west;`

Array Operator Implications

operator	*communication*	*iconic view*
@	point-to-point	
flood	sub-grid broadcast	
reduction	sub-grid reduction	
remap	all-to-all	

ZPL / F77 Syntax Comparison

```
[R] A := B + C;                 do j = 1, n
                                  do i = 1, n
                                    A(i, j) = B(i, j) + C(i, j)
                                  enddo
                                enddo
[R] A := B@west + C@east;       do j = 1, n
                                  do i = 1, n
                                    A(i, j) = B(i, j-1) + C(i, j+1)
                                  enddo
                                enddo
[R] A := >>[Top] B;             do j = 1, n
                                  do i = 1, n
                                    A(i, j) = B(1, j)
                                  enddo
                                enddo
[Top] A := +<<[R] B;            do j = 1, n
                                  do i = 1, n
                                    A(1, j) = A(1, j) + B(i, j)
                                  enddo
                                enddo
[R] A := A#[B, C];              do j = 1, n
                                  do i = 1, n
                                    A(i, j) = A(B(i, j), C(i, j))
                                  enddo
                                enddo
```

ZPL Example

```
region R = [1..1000];
       Int = [2..999];
direction next = [1];
          prev = [-1];
var A, B: [R] double;
…
[Int] B := A@prev + A@next;
```

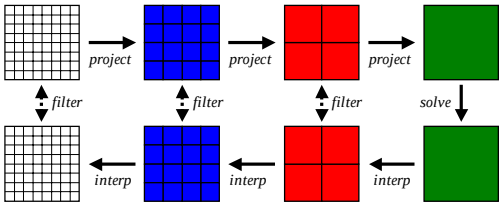
Outline

Parallel Computing Overview
Introduction to Regions and ZPL
Hierarchical Regions
- Sparse Regions
- Conclusion

Hierarchical Region Study

Goal: Evaluate parallel language support for *multigrid* applications in terms of:
- performance
- portability
- clarity

What is the Multigrid Method?



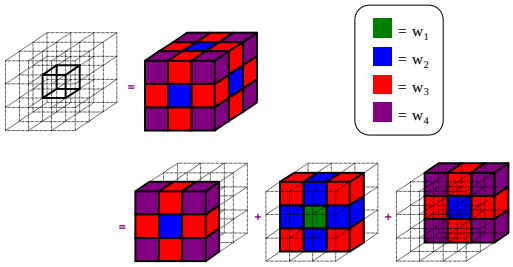
The NAS MG Benchmark

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

Algorithmically:

- requires hierarchical array support
- characterized by four 27-point stencils:
  - 2 inter-level (*rprj3, interp*)
  - 2 intra-level (*psinv, resid*)
- also requires periodic boundaries, reductions

The *rprj3* stencil



The Parallel Languages

Fortran 77 + MPI (F77+MPI)

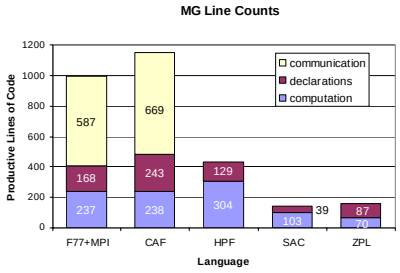
High Performance Fortran (HPF)

ZPL

Co-Array Fortran (CAF): *a Cray dialect of Fortran 90*

Single Assignment C (SAC): *a functional, array-based dialect of C*

Quantifying Clarity



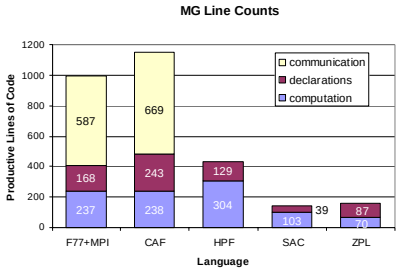
`rprj3` (ZPL)

```
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[N,0,0] + R@d[0,N,0] + R@d[0,0,N] +
      0.1250 * (R@d[1,1,0] + R@d[1,0,1] + R@d[0,1,1] +
                R@d[1,N,0] + R@d[1,0,N] + R@d[0,1,N] +
                R@d[N,1,0] + R@d[N,0,1] + R@d[0,N,1] +
                R@d[N,N,0] + R@d[N,0,N] + R@d[0,N,N])+
      0.0625 * (R@d[1,1,1] + R@d[1,1,N] +
                R@d[1,N,1] + R@d[1,N,N] +
                R@d[N,1,1] + R@d[N,1,N] +
                R@d[N,N,1] + R@d[N,N,N]);
end;
```

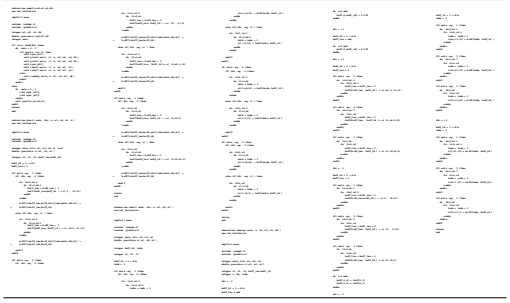
`rprj3` (F77+MPI)

```
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)
      + 0.12D0 * ( 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
```

Quantifying Clarity



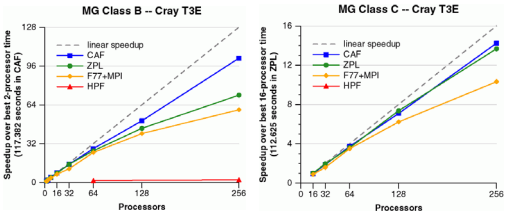
The Missing Communication



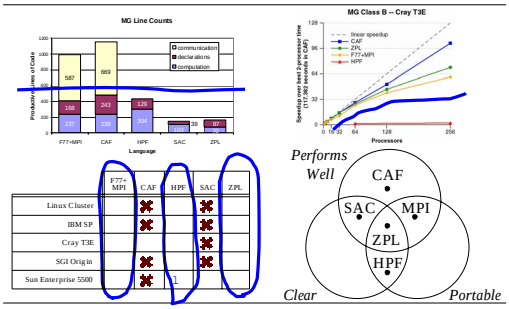
Experimental Platforms

	F77+ MPI	CAF	HPF	SAC	ZPL
Linux Cluster		✖		✖	
IBM SP		✖		✖	
Cray T3E				✖	
SGI Origin		✖		✖	
Sun Enterprise 5500		✖	1		

Performance Results (Cray T3E)



Summary of the Study



Outline

- Parallel Computing Overview
- Introduction to Regions and ZPL
- Hierarchical Regions
- Sparse Regions
- Conclusion

Sparse Arrays

A:

0	0	0	0	0	0
0	3	0	4	0	9
0	0	0	0	0	0
0	0	5	0	2	0
0	1	0	0	0	0
0	0	0	7	0	0

- Conceptually represent n^d values
- But only *nnz* are interesting, where $nnz = o(n^d)$
- Represent using $O(nnz)$ space and time
- Problem: ZPL's regions are regular, rectangular

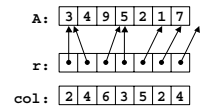
Compressed Sparse Row (CSR) Format

A:

0	0	0	0	0	0
0	3	0	4	0	9
0	0	0	0	0	0
0	0	5	0	2	0
0	1	0	0	0	0
0	0	0	7	0	0

= dense data vector

\+ directory information



Fortran Mat-Vect Multiplication

Dense Matrix-Vector Multiplication:

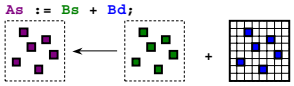
```
integer n
real*8 A(n,n)
real*8 t, V(n), S(n)


do i = 1,n
  t = 0.d0
  do j = 1,n
    t = t + A(i,j)*V(j)
  enddo
  S(i) = t
enddo
```

Sparse Matrix-Vector Multiplication:

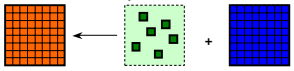
```
integer n, nnz
real*8 A(nnz)
real*8 t, V(n), S(n)
integer r(n+1),col(nnz)

do i = 1,n
  t = 0.d0
  do j = r(i),r(i+1)-1
    t = t + A(j)*V(col(j))
  enddo
  S(i) = t
enddo
```

Sparse Regions

```
region Rd = [1..n, 1..n];
       Rs = Rd where <boolean expression>;
var Ad,Bd:[Rd];
    As,Bs:[Rs];

[Rs] As := Bs + Bd;
```



```
[Rd] Ad := Bs + Bd;
```



(A-)ZPL Mat-Vect Multiplication

Dense Matrix-Vector Multiplication:

```
region R = [1..n,1..n];
       Row = [*,1..n];
       Col = [1..n,n];


var A:[R] double;
    V:[Row] double;
    S:[Col] double;

[Col] S := +<<[R] (A*V);
```

Sparse Matrix-Vector Multiplication:

```
region R = [1..n,1..n];
       Row = [*,1..n];
       Col = [1..n,n];
       Rs = R where …;

var A:[Rs] double;
    V:[Row] double;
    S:[Col] double;

[Col] S := +<<[Rs] (A*V);
```

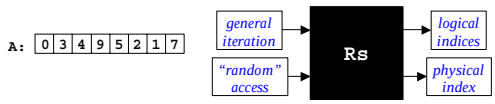
Sparse Arrays in A-ZPL

sparse array...

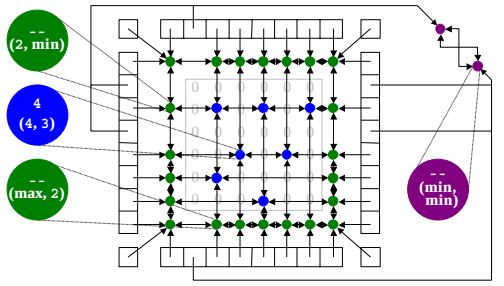
A:

0	0	0	0	0	0
0	3	0	4	0	9
0	0	0	0	0	0
0	0	5	0	2	0
0	1	0	0	0	0
0	0	0	7	0	0

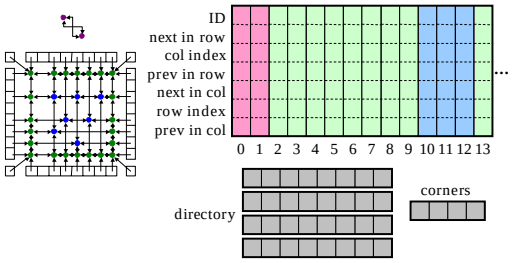
= 1D dense array + sparse region



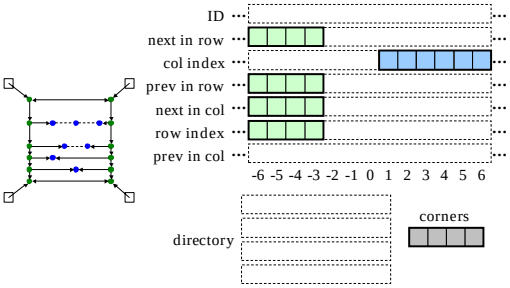
General Sparse Region Format



Sparse Format: Array of Records



Sparse Format: Fully Optimized



Sparse Benchmarks

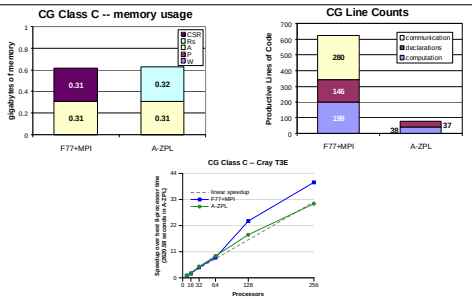
NAS CG:

- Conjugate Gradient computation
- Main computation: sparse matrix-vector multiply
- Class C = 150,000 × 150,000
- matrix has 36,121,066 nonzeroes (0.16%)

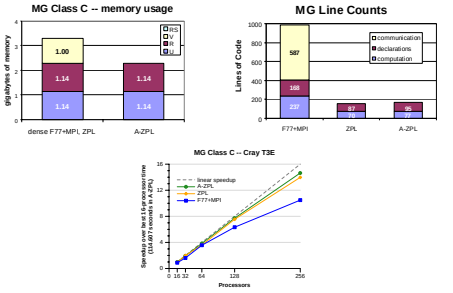
NAS MG:

- Class C = 512 × 512 × 512
- input has only 20 nonzeroes (14.9×10^{-6} %)

CG Results



MG Results



Outline

Parallel Computing Overview
Introduction to Regions and ZPL
Hierarchical Regions
Sparse Regions
Conclusion

Conclusions

High-level languages can be suitable for parallel computation

Regions are the reason for our success:
- support a clear, global view of computation
- expose the parallel implementation
- support portable parallelism
- allow performance competitive with hand-coded
- cleanly represent hierarchical/sparse algorithms

How To Find Out More

- Surf the website: www/research/zpl
- Take a seminar:
  - CSE590o: ZPL team seminar
    Winter 2001 -- introduction to language?
  - CSE590zpl: ZPL users seminar
    we teach language; users write & present apps
- Talk to a team member
  - me (brad@cs)
  - Steve Deitz (deitz@cs)
  - Larry Snyder (snyder@cs -- on sabbatical)
- Read papers
  - publications at website
  - my thesis (good for insomnia, strong gusts of wind)