**ZPL: A Region-Based Parallel Programming Language**

Brad Chamberlain  
CSE 505  
December 5, 2001

---

**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 \( a \)

---

prepared by Brad Chamberlain
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_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

```
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

- local-view
  - parallel implementation understood
- global-view
  - parallel implementation not understood
- MPI
  - parallel implementation understood
- HPF
  - parallel implementation not understood
**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](http://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**

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

F90:
```f90
A(1:n,1:n) = B(1:n,1:n) + C(1:n,1:n)
```

ZPL:
```zpl
[A..n,A..n] A := B + C;
```

---

prepared by Brad Chamberlain
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

<table>
<thead>
<tr>
<th>operator</th>
<th>effect</th>
<th>sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>Translate references</td>
<td>A@east</td>
</tr>
<tr>
<td>flood</td>
<td>Replicate values across array dimensions</td>
<td>&gt;&gt;[i,] A</td>
</tr>
<tr>
<td>reduction</td>
<td>Collapse array dimensions</td>
<td>&lt;&lt;&lt;&lt;[R] A</td>
</tr>
<tr>
<td>remap</td>
<td>Arbitrarily permute, gather, scatter,…</td>
<td>A#[X,Y]</td>
</tr>
</tbody>
</table>

Parallel Interpretation of Regions

Each region’s indices are distributed across the processor set

- defines data distribution
- defines work distribution

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

<table>
<thead>
<tr>
<th>operator</th>
<th>communication</th>
<th>iconic view</th>
</tr>
</thead>
<tbody>
<tr>
<td>@</td>
<td>point-to-point</td>
<td>![image]</td>
</tr>
<tr>
<td>flood</td>
<td>sub-grid broadcast</td>
<td>![image]</td>
</tr>
<tr>
<td>reduction</td>
<td>sub-grid reduction</td>
<td>![image]</td>
</tr>
<tr>
<td>remap</td>
<td>all-to-all</td>
<td>![image]</td>
</tr>
</tbody>
</table>

ZPL / F77 Syntax Comparison

<table>
<thead>
<tr>
<th>ZPL</th>
<th>F77</th>
</tr>
</thead>
<tbody>
<tr>
<td>A := B + C;</td>
<td>A(i, j) = B(i, j) + C(i, j);</td>
</tr>
<tr>
<td>A := Bwest + Ceast;</td>
<td>A(i, j) = B(i, j-1) + C(i, j+1);</td>
</tr>
<tr>
<td>A := &gt;&gt;[Top] B;</td>
<td>A(i, j) = B(1, j);</td>
</tr>
<tr>
<td>A := &lt;&lt;[Top] B;</td>
<td>A(i, j) = A(i, j) + B(i, j);</td>
</tr>
<tr>
<td>A := A</td>
<td>[B, C];</td>
</tr>
</tbody>
</table>

ZPL Example

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

Hierarchical Region Study

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

Outline

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

What is the Multigrid Method?

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

```plaintext
procedure rprj3(var S, R: [ , , ] double;
  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[N,1,1] +
                             R@d[N,1,N] + R@d[1,N,N] +
                             R@d[N,N,1] + R@d[N,N,N]);
  end;
```

rprj3 (F77+MPI)

```plaintext
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
        x2 = r(i1, i2-1,i3 ) + r(i1, i2+1,i3 ) +
             r(i1, i2, i3-1) + r(i1, i2, i3+1)
        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)
        s(j1,j2,j3) =
                      0.5D0 * r(i1,i2,i3) + 0.25D0 * (r(i1-1,i2,i3) + r(i1+1,i2,i3) + x2) +
                      0.125D0 * ( x1(i1-1) + x1(i1+1) + y2) +
                      0.0625D0 * ( y1(i1-1) + y1(i1+1))
      enddo
    enddo
  enddo
  j = k-1
  call comm3(s,m1j,m2j,m3j,j)
  return
end
```
**Summary of the Study**

**Experimental Platforms**

<table>
<thead>
<tr>
<th>Language</th>
<th>F77+ MPI</th>
<th>CAF</th>
<th>HPF</th>
<th>SAC</th>
<th>ZPL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux Cluster</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
</tr>
<tr>
<td>IBM SP</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cray T3E</td>
<td>✔️</td>
<td></td>
<td></td>
<td></td>
<td>✔️</td>
</tr>
<tr>
<td>SGI Origin</td>
<td>✔️</td>
<td>✔️</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sun Enterprise 5500</td>
<td>✔️</td>
<td></td>
<td>✔️</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Performance Results (Cray T3E)**

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

- Conceptually represent \( m \times n \) values
- But only \( m \times n_{nnz} \) are interesting, where \( n_{nnz} = o(n^2) \)
- Represent using \( O(mn_{nnz}) \) space and time
- Problem: ZPL’s regions are regular, rectangular

Compressed Sparse Row (CSR) Format

- dense data vector \( A \)
- directory information \( r \) and \( col \)

Fortran Mat-Vect Multiplication

Dense Matrix-Vector Multiplication:

```fortran
integer n
real*8 A(n,n), V(n), S(n)
do i = 1, n
  t = 0.0
  do j = 1, n
    t = t + A(i,j) * V(j)
  enddo
  S(i) = t
enddo
```

Sparse Matrix-Vector Multiplication:

```fortran
integer n, nnz
real*8 A(nnz), V(n), S(n)
integer r(n+1), col(nnz)
do i = 1, n
  t = 0.0
  do j = r(i), r(i+1)-1
    t = t + A(j) * V(col(j))
  enddo
  S(i) = t
enddo
```

Sparse Regions

- region \( R = \{1..n, 1..n\} \)
- \( R_{A} = R_{B} \) where boolean expression
- \( var \ A_{B}, B_{D}: \{R_{A}\} \)

(A-)ZPL Mat-Vect Multiplication

Dense Matrix-Vector Multiplication:

```zpl
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:

```zpl
region R = \{1..n, 1..n\};
Row = \{*,1..n\};
Col = \{1..n,n\};
var A:R double;
V:Row double;
[Col] S := +<<[R] (A*V);
```

Sparse Arrays in A-ZPL

- sparse array...
- \( A \)
- \( R_{S} \)
- \( A_{B} = R_{D} \)
- \( var \ A_{B}, B_{D}: \{R_{A}\} \)

ZPL Array Operator Wrap-up
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 nonzeros (0.16%)

NAS MG:
- Class C = 512 × 512 × 512
- input has only 20 nonzeros (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
  - brad (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)