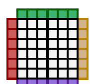
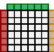
The ZPL Compiler : Portable Parallel Programming

Brad Chamberlain
The ZPL Project



CSE 401
May 28, 1999



What is Parallel Programming?

Parallel Programming: Computing using several processors cooperatively (“*in parallel*”)

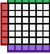
The promise: “If I run my program on 1,000 processors, it will run 1,000 times faster!” (*perfect speedup*)

The reality: This is very difficult to achieve

Why is Perfect Speedup Difficult to Achieve?

Imagine trying to get 1,000 people to solve a problem cooperatively:

- for trivial problems (stuffing a million envelopes), it probably *would* go 1,000× faster
- BUT, for more interesting problems, people would require meetings or communications…
  - …to compare notes
  - …to exchange data
  - …to pass a task off from one person to another
- this is *overhead* (wasn't required by a lone worker)
- the same issues apply to parallel programming

 Example of Parallel Overhead

- Matrix Addition

1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

+

1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

=

2	3	4	5
3	4	5	6
4	5	6	7
5	6	7	8

- Matrix Multiplication

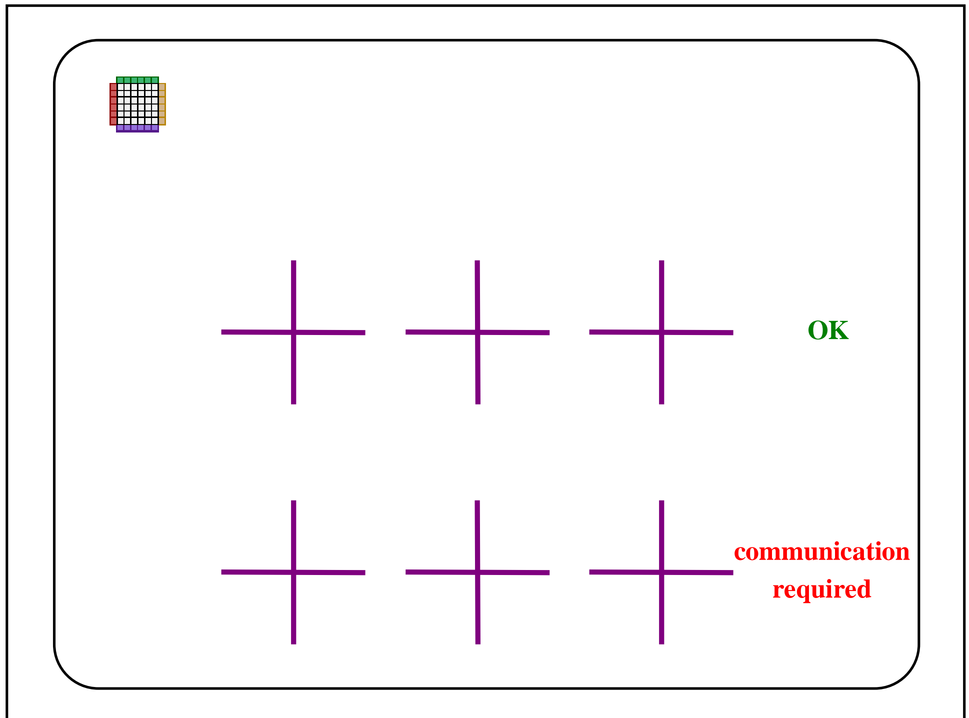
1	2	3	4
1	2	3	4
1	2	3	4
1	2	3	4

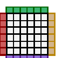
×

1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4

=

30	30	30	30
30	30	30	30
30	30	30	30
30	30	30	30



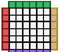


Writing Parallel Programs

Q: How can I write a parallel program?

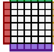
A: Lots of ways:

- use a traditional language (C/Fortran) plus a library that supports communication between processors (PVM/MPI) *– tedious, painstaking*
- use a traditional language and a *parallelizing compiler* that will "automatically" make your program run in parallel *– hard problem, cross fingers*
- use a new language designed to make parallel programming easier (HPF, NESL, ZPL)



What is ZPL?

- An array programming language
  - atomic operations are supported on arrays
- A *data-parallel* programming language
  - array operations are executed in parallel
- Targets large-scale scientific applications
- Developed at UW, 1991 – present

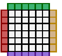


ZPL's Goals

Expressiveness: Allow programmers to express parallel computations elegantly

Portability: The program should run on all modern parallel platforms

Performance: The program should run quickly (ideally, as fast as a hand-coded parallel program)



Outline

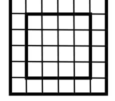
- ✔ Introduction to Parallel Programming
- Introduction to ZPL
- The ZPL Compiler
  - Achieving Portability
  - Achieving Performance

Region Overview

Regions: index sets that…

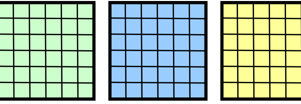
…can be named

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



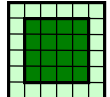
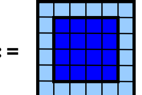
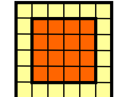
…are used to declare arrays

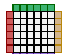
```
var A, B, C: [BigR] integer;
```



…specify indices for a statement's array references

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


:=

+


 Array Operators

Array Operators: applied to array expressions to modify the statement's region indices

- *e.g.*, the *@-operator* translates indices by an offset

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

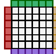
```
[R] A := (A@east + A@west)/2;
```

Regions vs. Indexing

Compare:

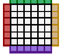
C:

```
for (i=1; i<=n; i++) {
  for (j=1; j<=n; j++) {
    A[i,j] = B[i,j] + C[i,j];
    A[i,j] = B[i,j+1] + C[i,j-1];
    T[i,j] = (A[i,j+1] + A[i,j-1])/2;
  }
}
for (i=1; i<=n; i++) {
  for (j=1; j<=n; j++) {
    A[i,j] = T[i,j];
  }
}
```



Challenges to ZPL's Portability

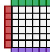
- Unlike sequential machines, parallel architectures aren't well-understood
- As a result, they change rapidly
  - since entering grad school, I've seen about a dozen different parallel architectures
  - about half of these are no longer in use
  - each architecture has its own unique characteristics, quirks
- Thus, writing a compiler to work on all of them could present quite a challenge



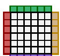
Achieving Portability

ZPL achieves platform-independence in two ways:

- posits an abstract parallel architecture
- uses ANSI C as its back-end "assembly language"

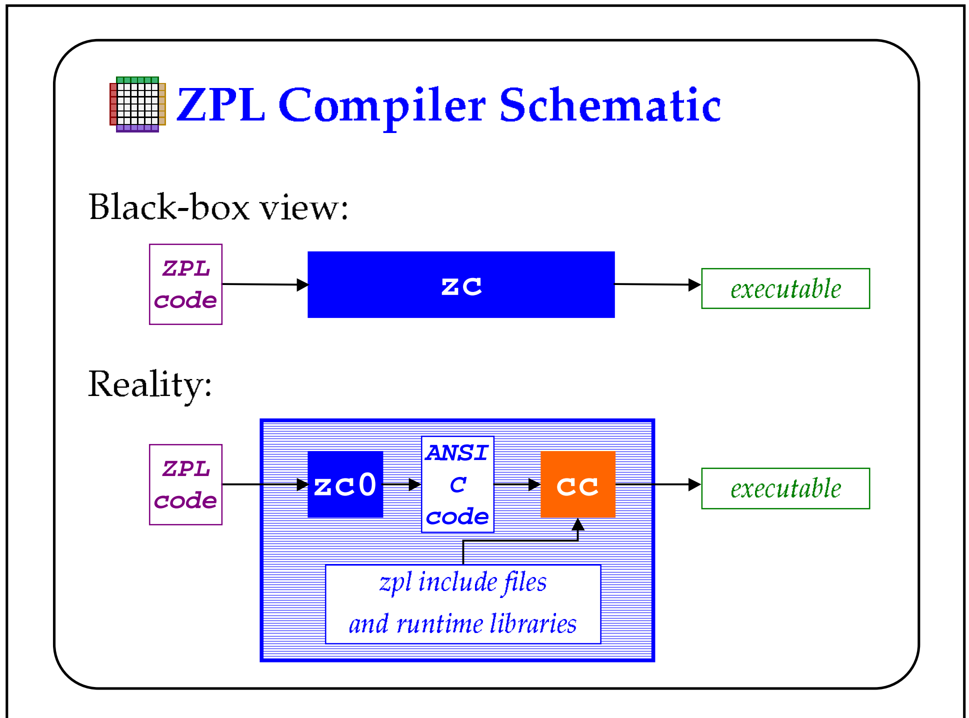
 ZPL's Machine Model

- ZPL relies on a *machine model* called the CTA
- The CTA provides an abstract view of parallel computers:
  - referring to data on another processor takes longer than referring to your own local data
  - the way that processors are interconnected is not terribly important
- When compiling ZPL, we focus on the CTA rather than each machine's particular characteristics

 Compiling to ANSI C

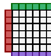
The ZPL compiler doesn't translate ZPL into assembly code, but rather into ANSI C

- don't need to learn each machine's instruction set
- don't have to write a new back-end for every new machine that emerges
- knowing how C translates to assembly allows us to generate efficient code
- don't have to worry about traditional scalar optimizations, register allocation, etc.



Runtime Library Overview

- Runtime libraries implement all machine-specific behavior:
  - interprocessor communication
  - memory allocation
  - allocating processors
  - coordinated I/O
- Porting the compiler to a new machine is as simple as porting the libraries
  - often no work at all, thanks to communication libraries like PVM and MPI

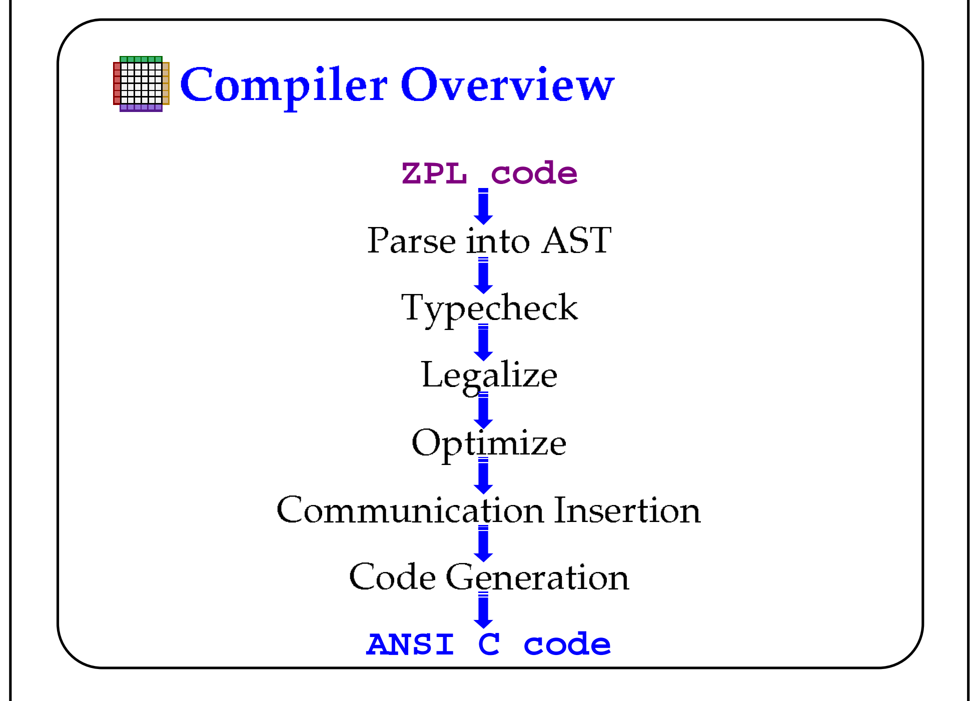


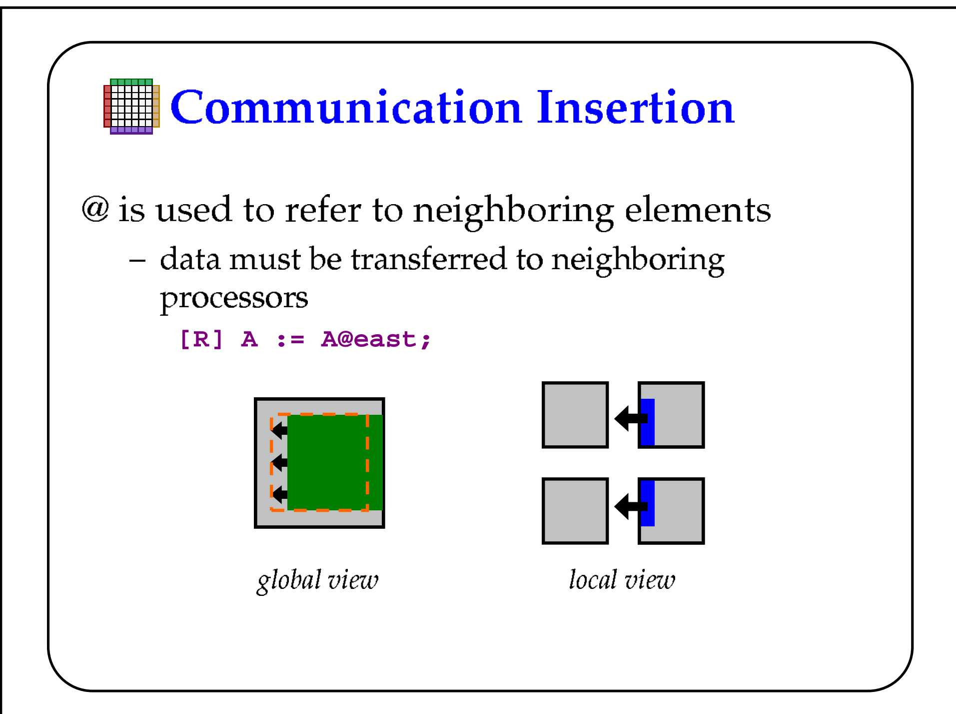
Compiler Output

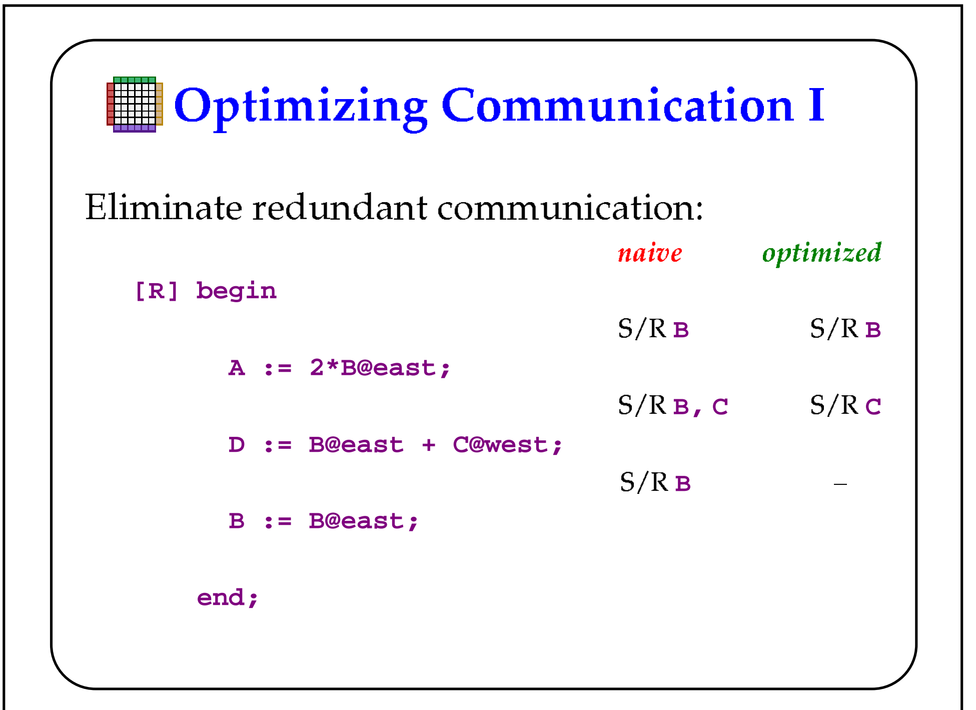
Compiler produces a *Single Program, Multiple Data* (SPMD) C program

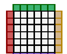
- run one copy of the program on each processor
- each processor given unique index
- based on its index, each processor performs a fraction of the total work:

```
for (i=_MYLO(R,0); i<=_MYHI(R,0); i++) {
  for (j=_MYLO(R,1); j<=_MYHI(R,1); j++) {
    _ACCESS(A,i,j) = _ACCESS(B,i,j) +
                     _ACCESS(C,i,j);
  }
}
```



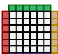

Communication Insertion
@ is used to refer to neighboring elements
– data must be transferred to neighboring processors
[R] A := A@east;
global view
local view


Optimizing Communication I
Eliminate redundant communication:
naive
optimized
[R] begin
S/R B
S/R B
A := 2*B@east;
S/R B, C
S/R C
D := B@east + C@west;
S/R B
–
B := B@east;
end;

 Optimizing Communication II

Overlap Communication and Computation

Code	*naive*	*optimized*
`[R] begin`		
	S/R `B`	S/R `B`; S `C`
`A := 2*B@east;`		
	S/R `C`	R `C`
`D := B@east + C@west;`		
	–	–
`B := B@east;`		
`end;`		

 Achieving Good Performance

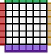
- High-level optimizations
  - communication optimizations
  - array elimination (*contraction*)
- Low-level optimizations
  - efficient array access
  - dealing with C's quirks

Implementing ZPL Arrays

- C doesn't support 2D dynamic arrays
- We need them because the number of processors is unknown at compile time
- Thus, we implement them by hand:

```
A = malloc((_MYHI(R,0)-_MYLO(R,0)+1) *
           (_MYHI(R,1)-_MYLO(R,1)+1) *
           sizeof(element));

ACCESS(A,i,j) = A + ((i-_MYLO(R,0))*numcols) +
                    (j-_MYLO(R,1));
```

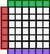
 Optimizing ZPL Arrays

- This method requires 2 subtractions, 2 additions, and 1 multiplication per array, per iteration
- But all we're doing is iterating over a solid block of memory!!
- Instead we can walk a pointer over memory:

```
Walker = A;
for (i=…) {
  for (j=…) {
    *Walker = *Walker + 1;
    *Walker += Col_Offset;
  }
  *Walker += Row_Offset;
}
```

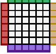
col_offset

row_offset



Conclusions

- For portably compiling a high-level language like ZPL, using C as a back-end is great
  - + compiles on all machines
  - + saves us from implementing traditional optimizations
  - + instead, can focus on issues unique to ZPL and parallel programming
    - communication insertion
    - communication optimizations
    - array semantics
  - – though similar to assembly, C is still not perfect



ZPL Summary

- Performance competitive with hand-coded C + MPI/PVM
- Released to the public July 1997 via web
- User community spans many disciplines
- Continual improvements to language and compiler

http://www.cs.washington.edu/research/zpl
mailto:zpl-info@cs.washington.edu