Today's Plan

Higher level languages

- Using ZPL
- Reasoning about performance with ZPL programs
  - A case study that pulls together many ideas that we've discussed this semester

Distributed Work Queue Discussion (II)

PSP Discussion

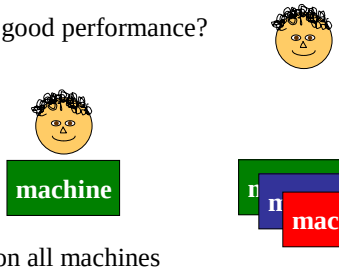
Quiz

1. Chapter 8 discusses ZPL's performance model, claiming that programmers can get a rough estimate of performance by examining the syntax of their ZPL programs. What is the largest weakness of this model?

Performance Portability

Tension

- How do we get both portability and good performance?
- For performance reasons:
  - Exploit machine details
  - Defeats portability
- For portability reasons:
  - Avoid all machine details
  - May lead to poor performance on all machines

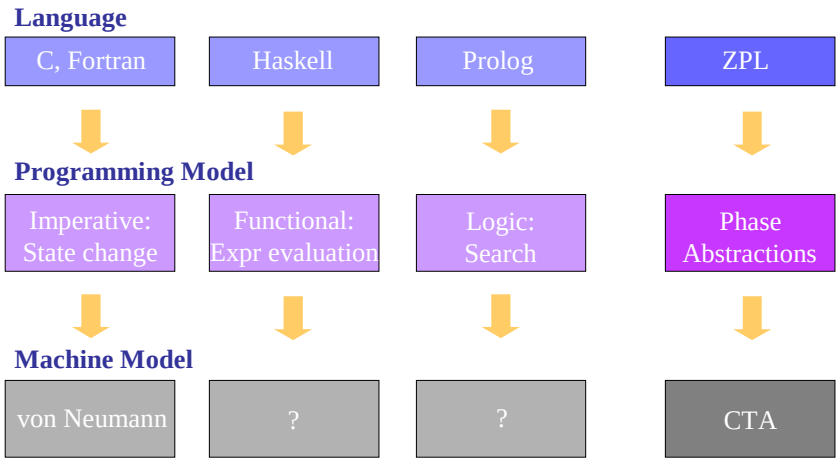


Reason by analogy

- In sequential computing, languages such as C and Fortran achieve both portability and good performance
- Why is this?
- Do all sequential languages achieve performance portability?

CS380P Lecture 17 ZPL's Performance Model 3

Consider Some Examples



CS380P Lecture 17 ZPL's Performance Model 4

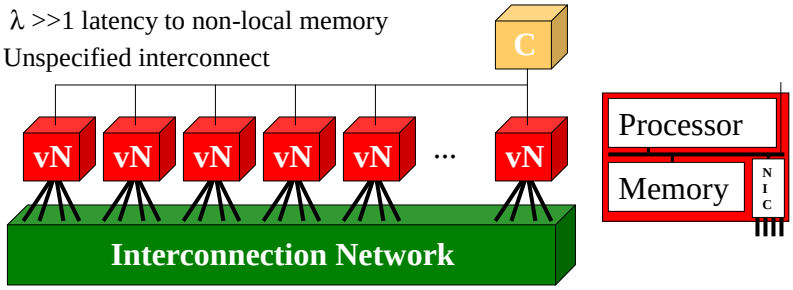
The CTA—The Candidate Type Architecture

CTA

- An abstract machine model
- Describes essential features of a wide class of MIMD machines
- Ignores machine-specific idiosyncrasies

The model

- P processors
- λ >>1 latency to non-local memory
- Unspecified interconnect



CS380P Lecture 17 ZPL's Performance Model 5

The Phase Abstractions [Alverson, et al, 1992]

Focus on scalable parallelism

- Code
- Data
- Communication

} **Can we scale these as a unit?**

CS380P Lecture 17 ZPL's Performance Model 6

Reasoning About Performance

Specify the following

- How processors are allocated to computation
- How regions (and arrays) are allocated in memory
- Rules of operation for primitive ZPL facilities including costs for computation and communication

Ensure that all source language features are explained

Explain the interactions with optimizations

Assigning Work to Processors

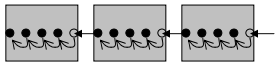
Two approaches for data-parallel computation

- Consider as an example: `A := B+C@east`
- **Virtual Processors:** (Unlimited Parallelism Approach)
  Each processor is allocated one scalar data value, that is, think of there being as many processors as array elements
- **Multiple Points Per Processor:** (Scalable Parallelism Approach)
  Each processor is allocated n values
  - n=1 is the virtual processor case
  - n=all-elements approximates the sequential case
  - Algorithms with this feature are scalable

Assigning Work (cont)

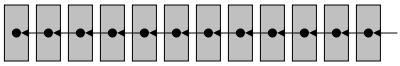
Are the two views equivalent?

- Virtualization ignores grain size
- Emulation misses the advantages of long instruction sequences
  - Pipelining
  - Caching
  - Prefetching



- Virtualization misses significant costs such as local shifting:

```
V:=V@right
```



ZPL Assumes Multiple Points Per Processor

Region allocation rule

- ZPL allocates regions to processors so that many contiguous elements are assigned to each

Array allocation rules

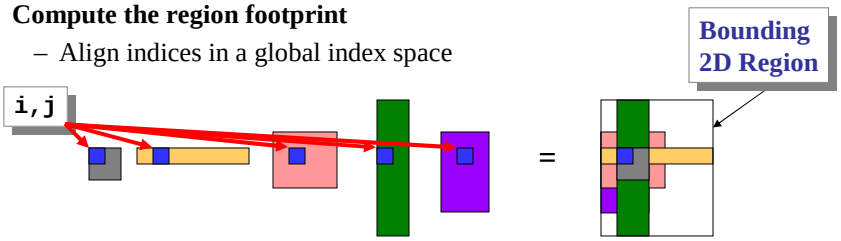
- Union the regions together to compute the **bounding region**
- Get processor number and arrangement from the command line
- Allocate the bounding region to the processors

Let's walk-through the process

Union the Regions Together

Compute the region footprint

- Align indices in a global index space



- Only interacting regions are unioned
- If region R is used to declare an array which is manipulated in the scope of region S, then R and S are said to **interact**

The bounding region is allocated to processors

CS380P Lecture 17 ZPL's Performance Model 11

Get Processor Number + Arrangement

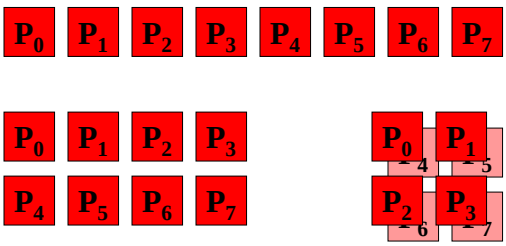
Number of processors

- Specified on the command line
- May also be specified in the program

Arrangement

- To understand allocation, assume that the processors are arranged in a grid

The CTA does not favor any arrangement, so use a generic one

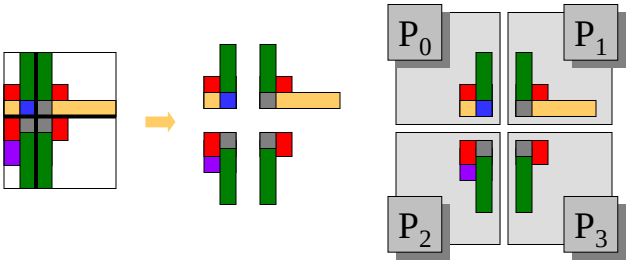


CS380P Lecture 17 ZPL's Performance Model 12

Allocate the Bounding Region to the Grid

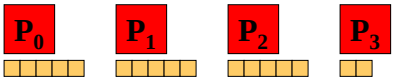
Attempt to balance the distribution

- Regions inherit their position from the bounding region
- Array elements inherit their positions—and hence their allocation—from their index's position in the region

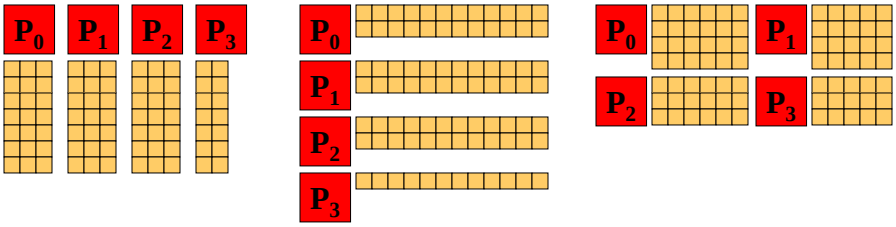


More Typical Allocations

1D is segmented



2D is panels, strips or blocks

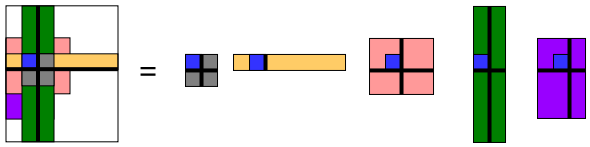


3D?

Fundamental Property of ZPL

Simple alignment

– For any arrays `A`, `B` of the same rank and having an element `[i,...,k]`, that element will be stored on the same processor for each array



Corollary

– Element-wise operations do not require any communication:

```
[R] ... A+B ...
```


Rules of Operation

WYSIWYG performance

– Performance is given in terms of the CTA
– Coarse performance features are visible
– Performance is relative,
e.g. operation *x* is more expensive in communication than operation *y*

Rules

`A + B -- Element-wise array operations`

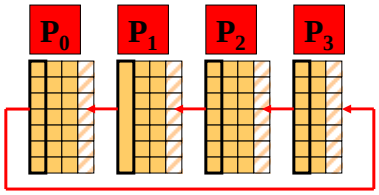
– No communication
– Per processor work is some constant C
– Work fully parallelizable, i.e. time = work/P

Rules of Operation (cont)

Rules

```
B + A@east   -- @ references
```

– Arrays allocated with **overlap regions** for every direction used



Recall the CTA

Charge λ time for communicating through the network

– Nearest neighbor point-to-point communication of edge elements, i.e. small communication, little congestion

– Edge communication benefits from surface-to-volume advantage: an n increase in elements, adds √n communication load

– Possible local data motion

Unlimited Parallelism does not support this view

Is This Simplistic Analysis Accurate?

It's not even close, but it's often good enough

– Contention in the network yields variance

– Processors may not be adjacent

– Processors are not synchronized

– Transmission time depends on message length

  – A better model: $\alpha + \beta n$

    where n is the message length,

    α is the startup cost and β is the per-byte cost

– Software costs can dominate network time

A Contrarian View

Model as accurately as possible

"Communication is the most expensive aspect of parallel computing, so structure the computation so that it optimizes the use of communication"

Is this a good idea?

- Structuring a program to optimize communication will embed properties and assumptions of a specific computer into the source code
- Parallel machines vary widely in their characteristics => source code must be changed for every machine

Wisdom

- Do not try to be too accurate
- Think of @-communication as a small but non-negligible cost, and leave the optimization to the compiler
- We will see a great example of this next lecture

CS380P Lecture 17 ZPL's Performance Model 19

Rules of Operation (cont)

Rules

`+<<A -- Reduce`

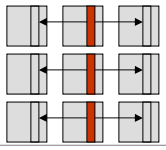
- Accumulate local elements
- Ladner/Fischer O(log P) tree accumulation, or better
- Broadcast, which is worst case O(log P), but usually less

`+||A -- Scan`

- Accumulate local elements
- Ladner/Fischer O(log P) tree parallel prefix logic
- Update of local elements

`>>[1..n,k]A -- Flood`

- Multicast array segments, O(log P) worst case
- Represent data non-redundantly



No other parallel language has a performance model

Applying The WYSIWYG In Real Life

```
program Life;
config var n : integer = 512;
region     R = [1..n, 1..n];
         BigR = [0..n+1,0..n+1];
direction  N = [-1, 0]; NE = [-1, 1
           E = [ 0, 1]; SE = [ 1, 1];
           S = [ 1, 0]; SW = [ 1,-1];
           W = [ 0,-1]; NW = [-1,-1];
var NN : [R] ubyte; TW : [BigR] boolean;
procedure Life();
      [R] begin
           TW := (Index1 * Index2) % 2; -- Make some data
           repeat
             NN := (TW@N + TW@NE + TW@E + TW@SE
                  + TW@S + TW@SW + TW@W + TW@NW);
             TW := (NN=2 & TW) | NN=3;
           until !|<<TW;
          end;
```

Declarations
Element-wise
Element-wise with @'s
Parallel prefix

Performance costs implied by WYSIWYG model

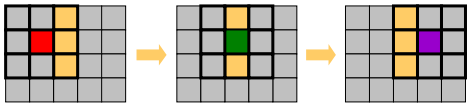
Optimizations Can Help

WYSIWYG is the worst case

– Optimizations are possible

Sequential Optimizations

– Stencil optimizations



8 additions specified per element, but fewer additions are possible—How?

– Compute sum of yellow items once per processor rather than once per element

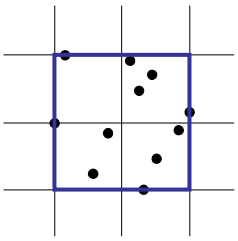
Parallel Optimizations

– Asynchronous communication overlaps communication with computation and hides communication latency

More Optimizations

Recall the bounding box code

```
[R] begin
      rightedge  := max<< X;
      topedge    := max<< Y;
      leftedge   := min<< X;
      bottomedge := min<< Y;
    end;
```



Each reduction has the same structure

- Iterate to find local max/min
- Aggregate using the Ladner/Fischer algorithm
- Broadcast the result to all processors

More Optimizations (cont)

```
loop1 max (X)
aggregate (maxX)
broadcast (maxX)
loop2 max (Y)
aggregate (maxY)
broadcast (maxY)
loop3 min (X)
aggregate (minX)
broadcast (minX)
loop4 min (Y)
aggregate (minY)
broadcast (minY)
```

Combine operations

- Compiler reorders code
  - Fuses loops
  - Combines aggregates
  - Combines broadcasts

```
loop max(X), max(Y), min(X), min(Y)
aggregate (maxX, maxY, minX, minY)
broadcast (maxX, maxY, minX, minY)
```

- Code runs about 4 times faster

Summarizing ZPL's Performance Model

Data distribution

– The mapping of data to processors is known

Execution costs

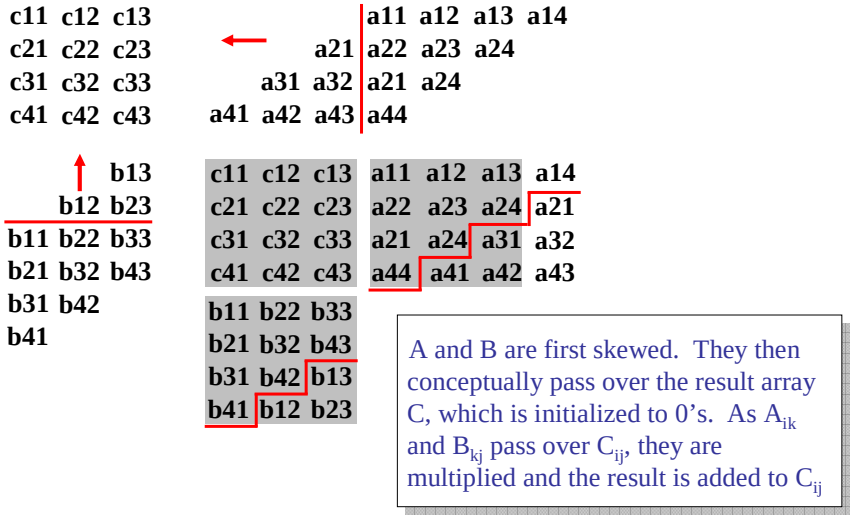
– The performance of all language constructs is explained in terms of these allocations and the CTA

The result

– Can compare worst-case analysis of program alternatives

The WYSIWYG model allows us to compare algorithms.
How do our two matrix multiplication algorithms compare?

Recall Cannon's Algorithm [1969]



A and B are first skewed. They then conceptually pass over the result array C, which is initialized to 0's. As A_{ik} and B_{kj} pass over C_{ij}, they are multiplied and the result is added to C_{ij}

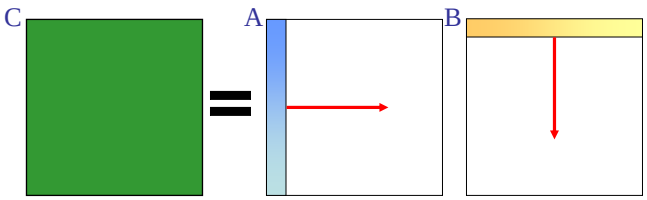
Cannon's Algorithm

Skew A, Skew B, Multiply, Accumulate, Rotate

```
            for i := 2 to m do   -- Skew A
[right of Lop]  wrap A;          -- Move col 1 to right
  [i..m, 1..n]  A := A@right;    -- Shift last i rows left
            end;
            for i := 2 to m do   -- Skew B
[right of Rop]  wrap B;          -- Move row 1 to below last
  [i..n, 1..p]  B := B@below;    -- Shift last i columns up
            end;
          [Res]  C := 0.0        -- Initialize C
            for i := 1 to n do   -- For A & B's common dim
          [Res]  C := C + A * B; -- Accumulate product
[right of Lop]  wrap A;          -- Send first col right
          [Lop]  A := A@right;   -- Shift array left
[below of Rop]  wrap B;          -- Send top row down
          [Rop]  B := B@below;   -- Shift array up
            end;
```

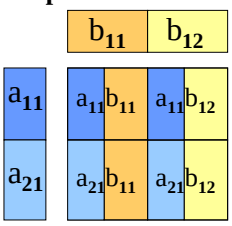
Recall the SUMMA Algorithm [van de Geijn and Watts 1995]

The SUMMA Algorithm



Broadcast a column of A, broadcast a row of B, and compute the k^{th} term of the dot product, repeat.

Example

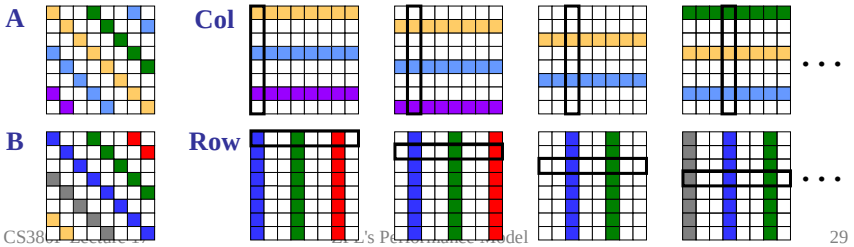


SUMMA in ZPL

SUMMA algorithm

- Iteratively flood a column of A and a row of B into temporary matrices
- Multiply and accumulate these results into C

```
[1..n,1..n]  C := 0.0;              -- Initialize C
[1..n,1..n]  for k := 1 to n do
       [,*]    Col := >>[,k] A;     -- Flood kth col of A
       [*,]    Row := >>[k,] B;     -- Flood kth row of B
               C   := C+Col*Row;    -- Accumulate product
             end;
```



Comparing the Two Algorithms

Which matrix multiplication algorithm is better?

Cannon

~~Declarations~~
Skew A
Skew B
~~Initialize C~~
~~loop through n~~
~~C += A*B~~
rotate A, B

SUMMA

~~Declarations~~
~~Initialize C~~
~~loop through n~~
flood A[,k]
flood B[k,]
~~C += A*B~~

- Rotate vs. flood

Cannon's Algorithm

Skew A, Skew B, {Multiply, Accumulate, Rotate}

```
          for i := 2 to m do -- Skew A
  [i..m, 1..n] A := A@^right;
          end;
          for i := 2 to p do -- Skew B
  [1..n, i..p] B := B@^below;
          end;

        [Res] C := 0.0;      -- Initialize C
          for i := 1 to n do -- For common dim
        [Res] C := C + A*B;  -- For product
        [Lop] A := A@^right; -- Rotate A
        [Rop] B := B@^below; -- Rotate B
          end;
```

Rotations have latency λ, but much local data motion

SUMMA Algorithm Analysis

Floods

– The flood is typically more expensive than λ time, but less than λ(log P)
. . . probably much less, and there are fewer of them

```
[1..m,1..p]    C := 0.0;       -- Initialize C
           for k := 1 to n do
   [1..m,*]  Col := >>[ ,k] A; -- Flood kth col of A
   [*,1..p]  Row := >>[k, ] B; -- Flood kth row of B
[1..m,1..p]    C += Col*Row;   -- Combine elements
           end;
```

SUMMA does not require as much communication or data motion as Cannon's, nor does it touch the array as much

The Bottom Line

SUMMA is the better algorithm

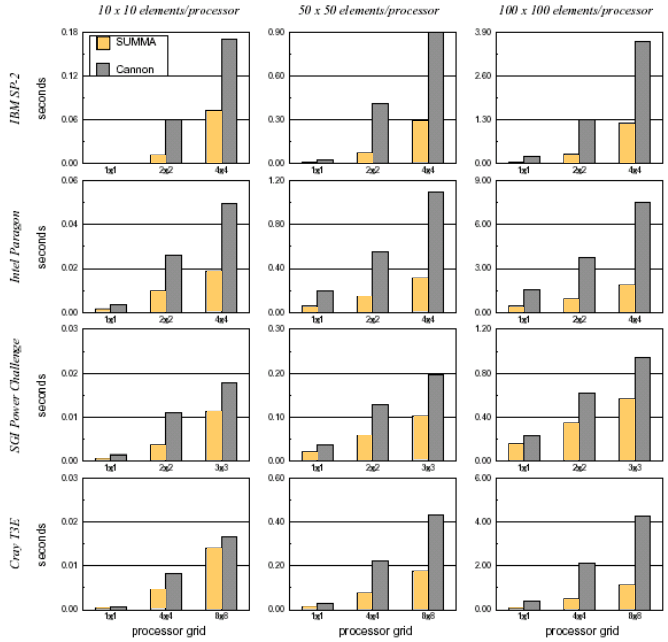
- Does "potentially more expensive communication", but less of it
- Its non-redundant flood arrays cache well
- There is less local data motion

Analytically . . .

Algorithm	# of Comm. Operations	Comm. Complexity	Communication Volume	FLOPS	Elements Referenced
Cannon	***4n***	***1***	***n***	***$2n^3$-n^2***	***n ($2n^2/2 + 3n^2$)***
SUMMA	***2n***	***log p***	***n***	***$2n^3$***	***n (n^2+2n)***

Empirically . . .

Performance Results



Summary

Performance portability

- ZPL builds on the CTA, an abstract parallel machine
- The CTA provides capabilities that can be efficiently implemented on a wide class of MIMD machines
- The mapping of ZPL programs to the CTA is well-specified
  - This mapping is less flexible than HPF's
- ZPL's performance model allows programmers to make good algorithmic choices

Discussion Topics

Distributed Work Queue (II)

Problem Space Promotion

Next Time

Reading

- None

Compilation support for performance portability

- Revisit the tension between specificity and generality