Today's Plan

The role of abstractions

- How compilers can break the tradeoff between performance and portability
- Compare MPI and ZPL
  - Performance
  - Portability
  - Programmability
- Weaknesses of ZPL

Recall MPI

MPI provides a wide interface

- 12 ways to perform point-to-point communication
- MPI 2.0 offers one-sided communication

	Normal	Sync	Ready	Buffered
Normal	MPI_Send	MPI_Ssend	MPI_Rsend	MPI_Bsend
Nonblock	MPI_Isend	MPI_Issend	MPI_Irsend	MPI_Ibsend
Persistent	MPI_Send_init	MPI_Ssend_init	MPI_Rsend_init	MPI_Bsend_init

Why so many choices?

What problems does this create?

Premature Optimization

The root of all evil

- Requires manual changes to the application source code
- Embeds optimizations into the source code

Long term implications

- Complicates maintenance
- Defeats portability

What's the fundamental problem?

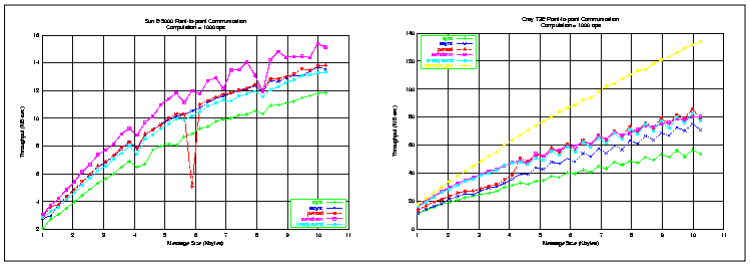
- MPI is too low level
- MPI over-specifies the communication
  - It specifies what to send, when to send it, and how to send it by specifying details of the implementation, such as the marshalling of data, synchronization, and buffering

Problems with MPI's Wide Interface (cont)

Long term problems

- No performance portability
- A form of premature optimization

Sun E5000 **Cray T3E**



Portable High Level Languages

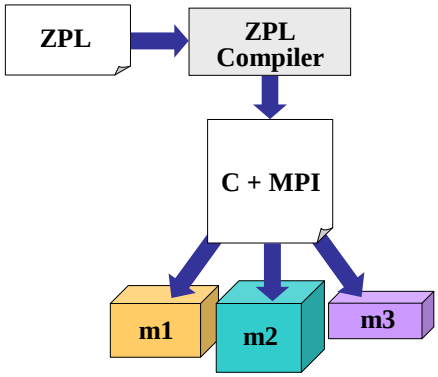
How do we produce portable code?

– What kind of communication code should we produce?

Compiling Higher Level Languages

Option 1: Portable compiler

– Compile to an intermediate language, such as C+MPI



Disdvantages

– What if m3 is a multi-core?

Advantages

– Intermediate code is portable

– Compiler has a single backend

Disadvantages

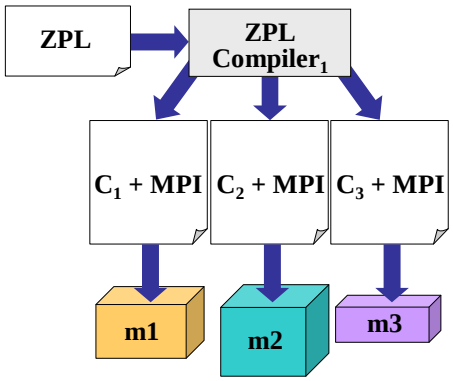
– Favors portability over performance

– We're still using the MPI interface, so we have the same performance portability problems that an MPI programmer faces

Compiling Higher Level Languages

Option 2: Machine-specific compiler

– Create multiple backends for multiple target platforms



Advantages

– Can exploit machine assumptions

Disadvantages

– Intermediate code is not portable

– Lots of work in building backends

Can we resolve this conflict between portability and performance?

Ironman Interface

A communications interface

– A set of four calls which define constraints about possible communication

– Individually, each call has little meaning

– Collectively, they can be bound to different mechanisms for different machines

The name is not based on the comic book

– It's a reference to Strawman, Woodman, Tinman and Ironman, . . .

which were different versions of the Ada language specification

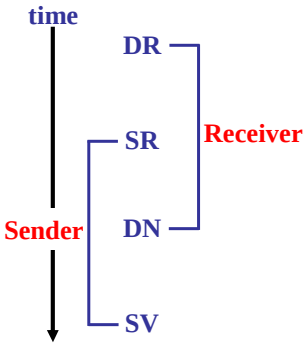
The Ironman Interface

DR– Destination Ready
- Earliest point at which the destination can receive data

SR– Source Ready
- Earliest point at which the sender can transmit data

DN– Destination Needed
- Latest point at which destination can receive data

SV– Sender Volatile
- Latest point by which data must be transmitted from the sender



CS380P Lecture 18 Performance Portability 9

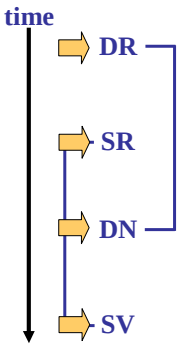
The Ironman Interface (cont)

DR– Destination Ready
- Assuming that the destination receives data into a buffer, this receive cannot occur until the buffer has been allocated, and it cannot occur while the buffer's data is in use

SR– Source Ready
- The data cannot be sent until its been computed by the sender

DN– Destination Needed
- The point at which the destination needs to use the data that it's receiving

SV– Source Volatile
- If the sender is re-using the buffer, then this is the point at which the source's data is no longer valid



CS380P Lecture 18 Performance Portability 10

Example Bindings

Synchronous Sends

Effect at P_1	SPMD code	Effect at P_2
-	DR()	-
Send data from P_1	SR()	-
-	DN()	Receive data in P_2
-	SV()	-

Example Bindings II

Non-blocking Sends and non-blocking Receives

Effect at P_1	SPMD code	Effect at P_2
-	DR()	Non-blocking receive in P_2
Non-blocking send from P_1	SR()	-
-	DN()	Wait for receive at P_2
Wait for send to complete	SV()	-

Example Bindings III

User-Defined Callback Routines

Effect at P_1	SPMD code	Effect at P_2
Synchronize	DR()	Post receive callback
Send data	SR()	-
-	DN()	Wait for receive to complete
-	SV()	-

Usage

- This binding is similar to the use of non-blocking receives, but when the message is complete, a user-defined callback routine is called to un-marshall the data as it arrives

Example Bindings IV

One-sided Communication

Effect at P_1	SPMD code	Effect at P_2
Synchronize	DR()	Synchronize
Put data into destination	SR()	-
Synchronize	DN()	Synchronize
-	SV()	-

Usage

- Some hardware allows one processor to Put data onto another processor's memory
- This mechanism is one-sided because the destination process is not involved

Static Analysis– Identify Uses and Defs

Example ZPL code

```
X := D;
DR();
. . .
S := . . .;
SR();
. . .
D := S@east
DN();
Y := D;
. . .
SV();
S := . . .;
```

Last use of D before data transfer
Cannot receive into D before this point

Last modification of S before data transfer
Cannot send D before this point

Need to receive D by this point
Next of use of D after data transfer

Need to send S by this point
Next of modification of S after data transfer

Static Analysis (cont)

Example ZPL code

```
X := D;
DR();
. . .
S := . . .;
SR();
. . .
D := S@east;
DN();
Y := D;
. . .
SV();
S := . . .;
```

Overall compilation scheme

- Identify the need for communication
- Use dependence analysis to identify Defs and Uses, which define the four points of interest
- Perform code motion to push the four locations apart
- Assign static Communication Tags to each set of Ironman calls
  - These tags are used to maintain state across calls at runtime
- Insert parameters to each call

Array language semantics help by reducing control flow

Performance Summary

Extra procedure call overhead

- Less than 1%

On an Intel Paragon

- Can use MPI, which maps well to Intel's NX message passing library

On the Cray T3E

- One-sided communication is 60-66% faster than MPI

Key benefit

- Ironman produces code that is both portable and efficient

The Larger Lessons?

Higher level languages

- Can use richer and more complicated interfaces
- No human would want to use the Ironman interface

Abstract interfaces

- Abstract interfaces can convey more information than lower-level interfaces
- Abstract interfaces can be both portable and efficient—but they need to convey the right information
- In the case of communication, they should specify what and when to transfer data and nothing more

MPI Summary

MPI strengths

- Has proven to be practically useful
- Runs on almost all parallel platforms
- Relatively easy to implement
- Can often serve as a building block for higher level languages

MPI weaknesses

- Too low-level of an interface
- Limited process model
- Forces programmer to maintain a mental map between a global view of data and multiple local views of data
  - Can we articulate the problems that this causes?

Programmer Productivity

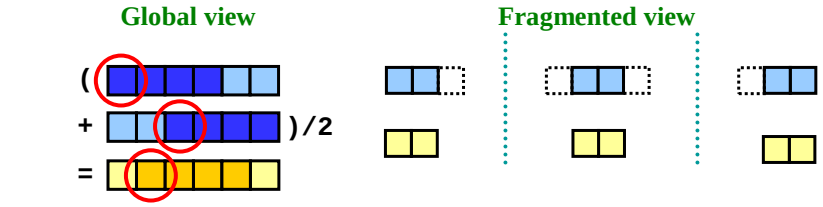
Global View abstractions

- Language constructs that produce the same result regardless of the number of processors that is used
- Allows programmers to debug sequentially
- Leads to more clear and concise code

Global View vs. Fragmented View

Example

– 3 point stencil of a vector

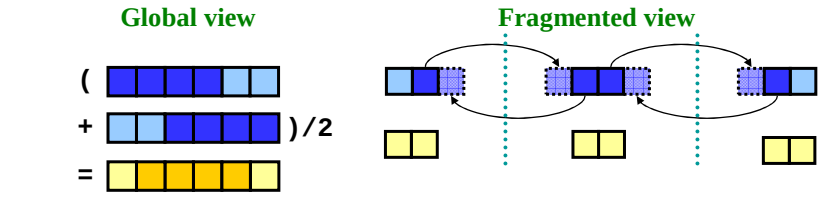



```
B = (A@east + A@west)/2;
```

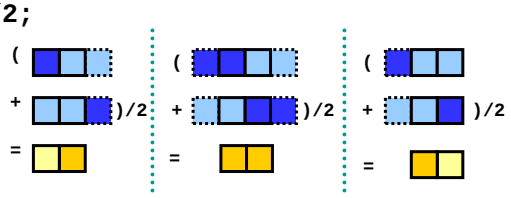

Global View vs. Fragmented View

Example

– 3 point stencil of a vector




```
B = (A@east + A@west)/2;
```



Global View vs. Fragmented View

Example

– 3 point stencil of a vector

Global View

```
begin
region R = [1..n];
var    n : int;
    A, B : [R] real;
procedure main()
[R] begin
    B := (A@west+A@east)/2;
    end;
end;
```

Fragmented View

```
def main() {
  var n: int = 1000;
  var locN: int = n/numProcs;
  var a, b: [0..locN+1] real;

  if (iHaveRightNeighbor) {
    send(right, a(locN));
    recv(right, a(locN+1));
  }
  if (iHaveLeftNeighbor) {
    send(left, a(1));
    recv(left, a(0));
  }
  forall i in 1..locN {
    b(i) = (a(i-1) + a(i+1))/2;
  }
}
```


Global View vs. Fragmented Vie

Assumes *numProcs* divides *n*; a more general version would require additional effort

Example

– 3 point stencil of a vector

Global View

```
begin
region R = [1..n];
var    n : int;
    A, B : [R] real;
procedure main()
[R] begin
    B := (A@west+A@east)/2;
    end;
end;
```

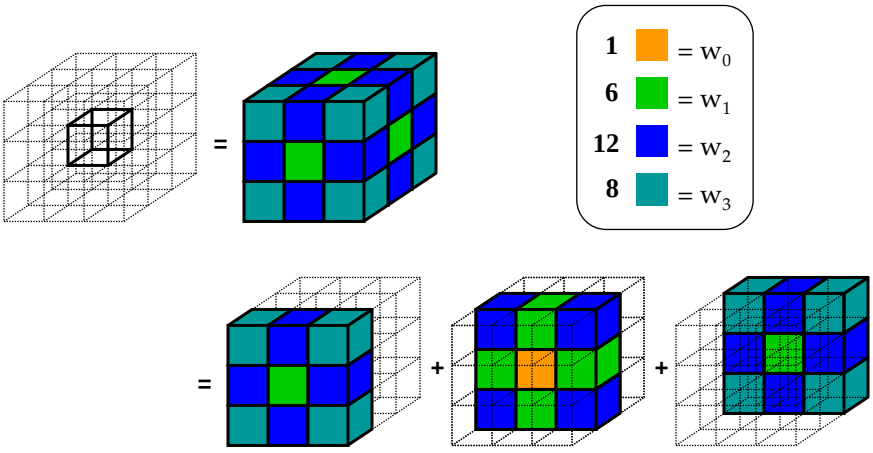
Fragmented View

```
def main() {
  var n: int = 1000;
  var locN: int = n/numProcs;
  var a, b: [0..locN+1] real;
  var innerLo: int = 1;
  var innerHi: int = locN;

  if (iHaveRightNeighbor) {
    send(right, a(locN));
    recv(right, a(locN+1));
  } else {
    innerHi = locN-1;
  }
  if (iHaveLeftNeighbor) {
    send(left, a(1));
    recv(left, a(0));
  } else {
    innerLo = 2;
  }
  forall i in innerLo..innerHi {
    b(i) = (a(i-1) + a(i+1))/2;
  }
}
```

Communication becomes geometrically more complex for higher-dimensional arrays

Consider the *rprj3* stencil from NAS MG



NAS MG *rprj3* stencil in ZPL

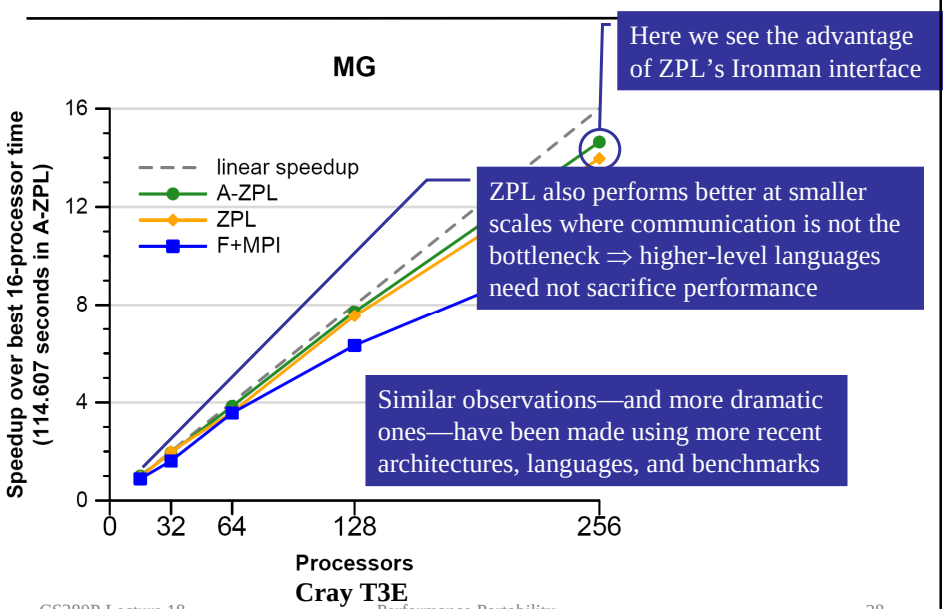
```
procedure rprj3(var S,R: [,,] double;
                d: array [] of direction);
begin
  S := 0.5    *  R
     + 0.25   * (R@^d[ 1, 0, 0] + R@^d[ 0, 1, 0] + R@^d[ 0, 0, 1] +
                 R@^d[-1, 0, 0] + R@^d[ 0,-1, 0] + R@^d[ 0, 0,-1])
     + 0.125  * (R@^d[ 1, 1, 0] + R@^d[ 1, 0, 1] + R@^d[ 0, 1, 1] +
                 R@^d[ 1,-1, 0] + R@^d[ 1, 0,-1] + R@^d[ 0, 1,-1] +
                 R@^d[-1, 1, 0] + R@^d[-1, 0, 1] + R@^d[ 0,-1, 1] +
                 R@^d[-1,-1, 0] + R@^d[-1, 0,-1] + R@^d[ 0,-1,-1])
     + 0.0625 * (R@^d[ 1, 1, 1] + R@^d[ 1, 1,-1] +
                 R@^d[ 1,-1, 1] + R@^d[ 1,-1,-1] +
                 R@^d[-1, 1, 1] + R@^d[-1, 1,-1] +
                 R@^d[-1,-1, 1] + R@^d[-1,-1,-1]);
end;
```

Yikes

- Looks quite messy because it uses a 27-point stencil
- With 27 directions, even naming them is inconvenient
- What does this code look in Fortran + MPI?

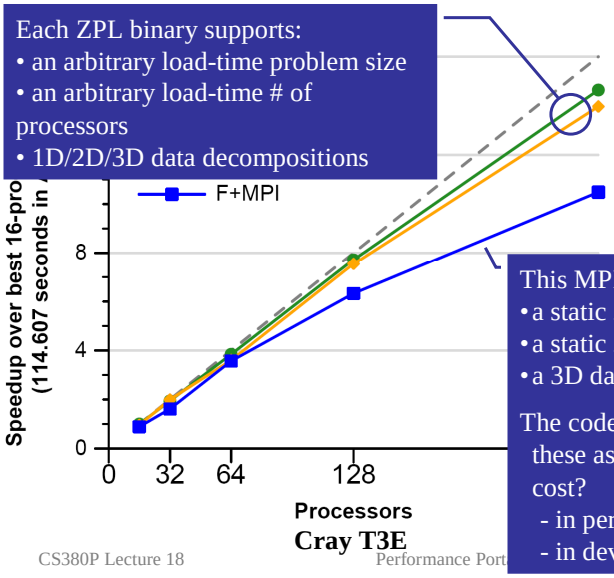
NAS MG *rprj3* stencil in Fortran+MPI

Performance Notes



Generality Notes

Each ZPL binary supports:
- an arbitrary load-time problem size
- an arbitrary load-time # of processors
- 1D/2D/3D data decompositions

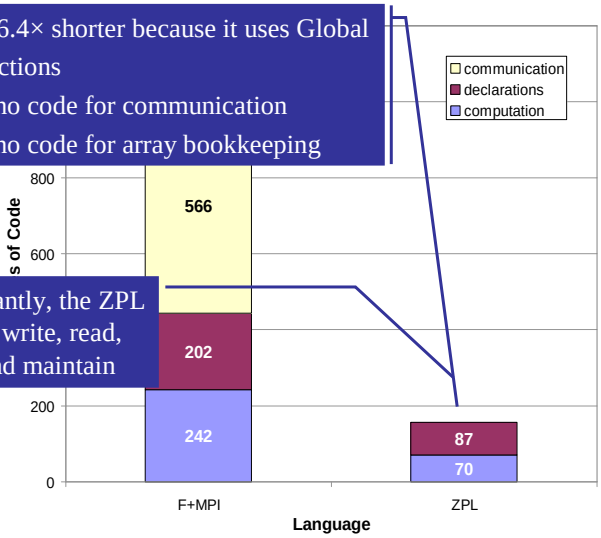


This MPI binary only supports:
- a static 2^k problem size
- a static 2^j # of processors
- a 3D data decomposition

The code could be rewritten to relax these assumptions, but at what cost?
- in performance?
- in development effort?

Code Size Notes

The ZPL is 6.4× shorter because it uses Global View abstractions
⇒ Little/no code for communication
⇒ Little/no code for array bookkeeping



More importantly, the ZPL is easier to write, read, modify, and maintain

Critiquing ZPL

Strengths

- Concise
- Global View abstractions
- Clear performance model

Weaknesses

- Focuses on a data parallelism
- Focuses on arrays
- Focuses on regular parallelism
- Unfamiliar to many programmers
- Is it too restrictive?

Next Time

Chapel

- Cascade High Productivity Programming Language

Reading

- Chapter 9
- Chapel paper (on website)