Language Comparisons

We've seen several ways to program parallel computers ...
how do they compare?

Final Comments on Chapel

- Last time, Brad said regarding machine model

> 4) A Note on Machine Model
> - As with ZPL, the CTA is still present in our design to reason about locality
> - That said, it is probably more subconscious for us
> - And we vary in some minor ways:
>   - no controller node -- though we do utilize a front-end launcher node in practice
>   - nodes can execute multiple tasks/threads -- through software multiplexing if not hardware

- Is that really different from what we used?

© 2010 Larry Snyder, CSE

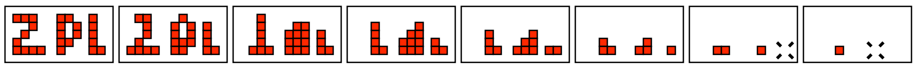
Chapel vs ZPL

- At one point Brad distinguished Chapel from ZPL by pointing out that Chapel doesn't have a WYSIWYG performance model
  - Does it matter?
  - Can you understand how C works even though it isn't WYSIWYG?
  - Is understanding the semantics sufficient?

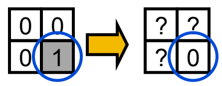
Wrap Up: Issues of Compilation

- If high level languages will save us in parallel computation, then the compiler is our primary tool for making the idea work ...
- How do compilers produce efficient code?

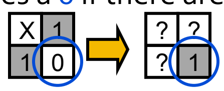
Connected Components Algorithm



- The Amazing Levialdi Shrinking Operator (1972)
  - Each pixel simultaneously changes state according to the following rules



(1) A 1 bit becomes a 0 if there are 0's to its West, NW, and North



(2) A 0 bit becomes a 1 if there are 1's to its West and North

(3) All other bits remain unchanged

8-way Connected Components

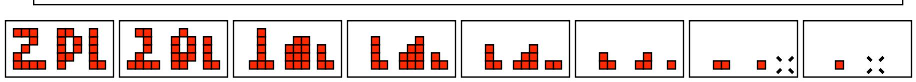
- ZPL Solution

```
. . .
Count := 0;
repeat
  Next  := Image & (Image@north | Image@nw | Image@west);
  Next  := Next  | (Image@west  & Image@north & !Image);
  Conn  := Next@east | Next@se | Next@south;
  Conn  := Image & !Next & !Conn;
  Count += Conn;
  Image := Next;
  smore := |<< Next;
until !smore;
. . .
```

Rule 1

Rule 2

Test for Poof



Loop Fusion

- Lines in an array language translate into loops

```
Next:=Image & (Image@north | Image@nw | Image@west);
  for (i=0; i<dim_1; i++){
    for (j=0; j<dim_2; j++){
      ... /* scalar code stmt 1 */
    }
  }
Next:=Next | (Image@west  & Image@north & !Image);
  for (i=0; i<dim_1; i++){
    for (j=0; j<dim_2; j++){
      ... /* scalar code stmt 2 */
    }
  }
```


© 2010 Larry Snyder, CSE

Loop Fusion

- When the ranges match, the bodies can be merged

```
for (i=0; i<dim_1; i++){
  for (j=0; j<dim_2; j++){
    ... /* scalar code stmt 1 */
    ... /* scalar code stmt 2 */
    ... /* scalar code stmt 3 */    basic block
    ... /* scalar code stmt 4 */
    ... /* scalar code stmt 5 */
   }
 }
```

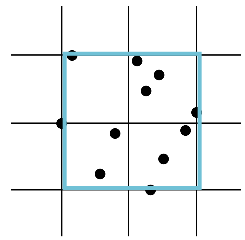
- Large basic block permit much optimizaiton

© 2010 Larry Snyder, CSE

Finding the Bounding Box

- Given
  - X and Y are 1D arrays of coordinates such that (X_i, Y_i) is a position in the coordinate plane
  - How do you compute the bounding box in ZPL?

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



Bounding Box Using Records

- Using a Point Type

```
type point = record
          x : integer;        -- x coordinate
          y : integer;        -- y coordinate
        end;
var Points : [1..n] point;          -- points in a
plane
. . .
[R] begin
        rightedge   := max<< Points.x;
        topedge     := max<< Points.y;
        leftedge    := min<< Points.x;
        bottomedge  := min<< Points.y;
    end;
```

Optimizing the Communication

- A key property is the regions are the same

```
[R] begin
   // rightedge  := max<< Points.x;
      val1=find_local_max(Points.x);
      reduce_upsweep_max(val1);
      rightedge=catch_broadcast();
   // topedge    := max<< Points.y;
      val2=find_local_max(Points.y);
      reduce_upsweep_max(val2);
      topedge=catch_broadcast();
      ...
    end;
```

5/26/10 © 2010 Larry Snyder, CSE 11

One Upsweep, One Downsweep

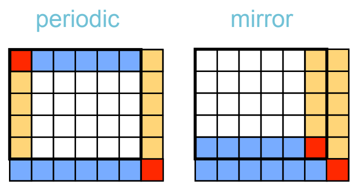
- Though no asymptotic benefit, performance win

```
[R] begin
      val1=find_local_max(Points.x);
      val2=find_local_max(Points.y);    loops fused
      val3=find_local_min(Points.x);
      val4=find_local_min(Points.y);
      reduce_upsweep_((max,val1),(max,va2),
            (min,val3),(min,val4));
      temp=catch_broadcast();
      rightedge=temp[0];
      topedge=temp[1];
      leftedge=temp[2];
      bottomedge=temp[3];
   end;
```

5/26/10 © 2010 Larry Snyder, CSE 12

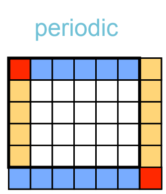
Boundary Conditions

- Data parallelism
  - Often quite regular except for the end-cases
- ZPL elevates the concept of a boundary condition



Boundaries

- The shallow benchmark



HPF

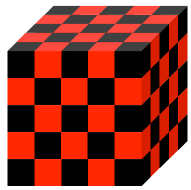
```
C Periodic boundary conditions
  uold(m+1,:n) = uold(1,:n)
  vold(m+1,:n) = vold(1,:n)
  pold(m+1,:n) = pold(1,:n)
  u(m+1,:n) = u(1,:n)
  v(m+1,:n) = v(1,:n)
  p(m+1,:n) = p(1,:n)
CAPR$ DO PAR on POLD<:,1>
  uold(:m,n+1) = uold(:m,1)
  vold(:m,n+1) = vold(:m,1)
  pold(:m,n+1) = pold(:m,1)
  u(:m,n+1) = u(:m,1)
  v(:m,n+1) = v(:m,1)
  p(:m,n+1) = p(:m,1)
  uold(m+1,n+1) = uold(1,1)
  vold(m+1,n+1) = vold(1,1)
  pold(m+1,n+1) = pold(1,1)
  u(m+1,n+1) = u(1,1)
  v(m+1,n+1) = v(1,1)
  p(m+1,n+1) = p(1,1)
```

ZPL

```
/* Periodic boundary conditions */
[e of I]  wrap U, Uold, V, Vold, P, Pold;
[s of I]  wrap U, Uold, V, Vold, P, Pold;
[se of I] wrap U, Uold, V, Vold, P, Pold;
```

Consider: Red/Black SOR

- Compute partial differential equations
  - Use successive over-relaxation
  - Arrange 3D values into red and black cells
  - Update in place by alternately computing values for red and black cells



Two Implementations

- Regions and region operators raise the level of abstraction

ZPL

```
for nrel := 1 to nITER do
  /* Red relaxation */
[I with Red]     U := factor*(hsq*F + U@top    + U@bot    + U@left+
                                      U@right + U@front + U@back);
  /* Black relaxation */
[I without Red] U := factor*(hsq*F + U@top    + U@top    + U@left+
                                      U@right + U@front + U@back);
end;
```

F90/HPF

```
      DO nrel = 1,iter
        where  (RED(2:NX-1,2:NY-1,2:NZ-1))
!     Relaxation of the Red points
        U(2:NX-1,2:NY-1,2:NZ-1)  =                                      &
     &              factor*(hsq*F(2:NX-1,2:NY-1,2:NZ-1)+                &
     &    U(1:NX-2,2:NY-1,2:NZ-1) + U(3:NX,2:NY-1,2:NZ-1)+              &
     &    U(2:NX-1,1:NY-2,2:NZ-1) + U(2:NX-1,1:NY-2,2:NZ-1)+            &
     &    U(2:NX-1,2:NY-1,1:NZ-2) + U(2:NX-1,2:NY-1,3:NZ))
        elsewhere
!     Relaxation of the Black points
        U(2:NX-1,2:NY-1,2:NZ-1)  =                                      &
     &              factor*(hsq*F(2:NX-1,2:NY-1,2:NZ-1)+                &
     &    U(1:NX-2,2:NY-1,2:NZ-1) + U(3:NX,2:NY-1,2:NZ-1)+
```

Two Implementations

Did you spot the bugs?

- Regions and region operators raise the level of abstraction

ZPL

```
for nrel := 1 to nITER do
  /* Red relaxation */
[I with Red]    U := factor*(hsq*F + U@top   + U@bot   + U@left+
                                     U@right + U@front + U@back);
  /* Black relaxation */
[I without Red] U := factor*(hsq*F + U@top   + U@top   + U@left+
                                     U@right + U@front + U@back);
end;
```

F90/HPF

```
      DO nrel = 1,iter
        where (RED(2:NX-1,2:NY-1,2:NZ-1))
!     Relaxation of the Red points
        U(2:NX-1,2:NY-1,2:NZ-1) =                                 &
     &              factor*(hsq*F(2:NX-1,2:NY-1,2:NZ-1)+          &
     &    U(1:NX-2,2:NY-1,2:NZ-1) + U(3:NX,2:NY-1,2:NZ-1)+        &
     &    U(2:NX-1,1:NY-2,2:NZ-1) + U(2:NX-1,1:NY-2,2:NZ-1)+      &
     &    U(2:NX-1,2:NY-1,1:NZ-2) + U(2:NX-1,2:NY-1,3:NZ))
        elsewhere
!     Relaxation of the Black points
        U(2:NX-1,2:NY-1,2:NZ-1) =                                 &
     &              factor*(hsq*F(2:NX-1,2:NY-1,2:NZ-1)+          &
     &    U(1:NX-2,2:NY-1,2:NZ-1) + U(3:NX,2:NY-1,2:NZ-1)+
```

3:NY

Comparing HPF and ZPL

- What's the difference in the two codes?
  - We cheated by not showing the definition of the Red mask in ZPL
- More fundamentally
  - Indexing is error prone
  - **Different things should look different**
    - With the explicit indices, everything looks similar
    - Why is this important?
  - **Abstraction principle**
    - If something is important, then it should be given a name and reused
    - Regions and directions support provide abstraction for data-parallel computation

Consider 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

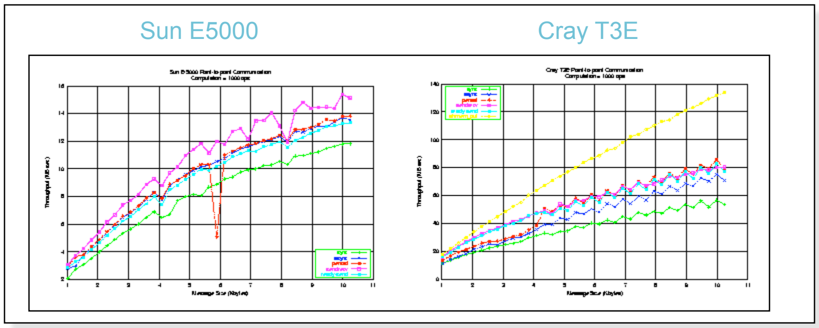
- Why so many choices?
- What problems does this create?

Problems with MPI's Wide Interface

- Short term problems
  - Complicates the interface
  - Some of the specialized routines are difficult to use
    - Eg. `MPI_Rsend()` assumes that the sender and receiver are already synchronized; if not, the message is dropped on the floor

Problems with MPI's Wide Interface

- Long term problems



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

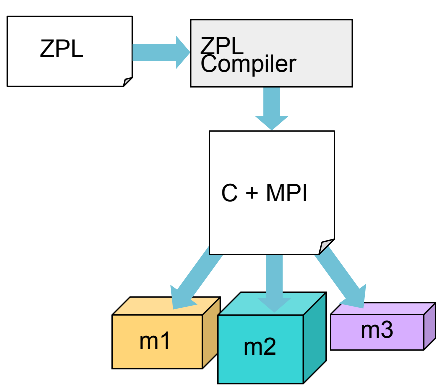
Premature Optimization

- 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

Why don't compilers have this same problem?

Compiling Higher Level Languages

- Option 1: Portable compiler
  - Compile to an intermediate language, such as C+MPI



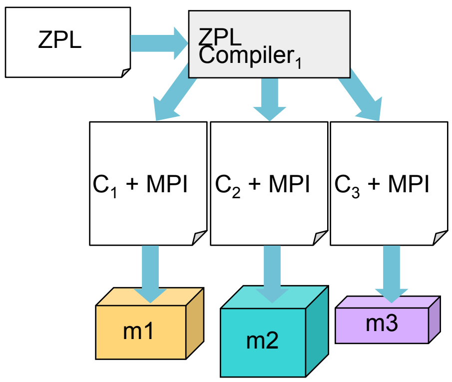
Advantages
- Intermediate code is portable
- Compiler has a single backend

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

 © 2010 Larry Snyder, CSE

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

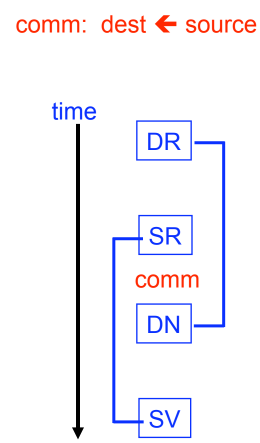
How can we resolve this conflict between portability and performance?

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 spec

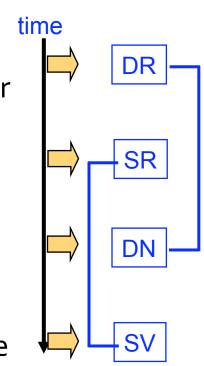
The Ironman Interface: Timing

- 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



The Ironman Interface: Actions

- DR– Destination Ready
  - Assuming the destination receives data into a buffer, the receive cannot occur until the buffer has been allocated, nor can it occur while the buffer's data is in use
- SR– Source Ready
  - Data cannot be sent until computed by sender
- DN– Destination Needed
  - The point at which the destination needs to use the data 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



Static Analysis– Identify Uses, 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 use of D after data transfer

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

© 2010 Larry Snyder, CSE

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

© 2010 Larry Snyder, CSE

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()	-

Q: Can we bind DR() to a receive?
A: No. It would be legal from P_2's point of view, but it would cause deadlock in an SPMD program in which processes both send and receive data

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

Performance Summary

- Extra procedure call overhead
  - Less than 1%
- On clusters and explicit MP machines
  - Can use MPI as envisioned by the designers
- On the Cray T3E and machines with 1-sided comm
  - One-sided communication is 60-66% faster than MPI
- On shared memory machines, use load/store
- Key benefit
  - Ironman produces code that is both portable and efficient though abstraction (dest ← source) and late binding

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

© 2010 Larry Snyder, CSE

Break

- In the second half we compare and contrast languages … be prepared to comment on how the language you reviewed compares

© 2010 Larry Snyder, CSE

Language Summary

- Key criteria to evaluate any parallel programming facility:
  - Correctness
  - Performance
  - Portability
  - Scalability

We discuss criteria for evaluating languages and identify good features that we expect future languages to have … think about how these compare with the language you reviewed

Correctness

- P-Independence
  - A parallel program is *P-independent* if and only if it always produces the same output on an input regardless of the number or arrangement of processes on which it is run; otherwise, it is called *P-dependent*
- Global view vs Local view
  - Classify ||-programming abstractions: locks, Send /Receive, forall loops, Barrier, Reduce/Scan
- How important is correctness in alg choice?

Performance

- Performance is difficult to achieve in many cases because …
  - <examples>
- What is the affect of ||-performance on sequential execution?
- What else is there in parallel computation besides performance???
  - Does performance affect the choice of algorithm?

Scalability

- Is scalability a concern in the the multicore world?
  - Does scalability affect the choice of algorithm?
- Good SW Engineering says that we should focus on getting the program working, and then optimize; if a program has been ||-ized by focusing on the 10% of the code where all of the time is spent, do we expect it to be scalable?

Portability

- It's a basic fact of CS that computers are universal, so programs "run" on an platform
- Performance portability is the term that stresses that parallel programs should "run well" everywhere
  - Is it worth it?
  - Does portability affect the choice of algorithm?

Comparing ZPL and Nesl

- Both high level
- Both rest on a small number of fundamental abstractions
- Both get their parallelism by data parallel evaluation of array expressions

- Key difference – ZPL's performance model gives direct info on how program will run
- Nesl's complexity model uses idealized PRAM

Key Lessons For Future

- We have seen several concepts that we want in future languages
  - Hidden parallelism
  - Transparent performance
  - Knowledge of Affects on Locality
  - Constrained Parallelism
  - Implicit vs Explicit Parallelism

Consider Each

© 2010 Larry Snyder, CSE

Hidden Parallelism

- If we didn't have to give it another thought, we'd all be happy!
  - If we can benefit from parallelism without explicitly thinking about, we win
  - Find abstractions that are hand for programming but which also allow the compiler to generate parallelism

© 2010 Larry Snyder, CSE

Transparent Performance

- We need to know when we're winning and when we are losing in order to make effective algorithm choices
  - Somehow we must "see" the effects of our decisions
  - WYSIWYG may be overkill, but vague, nonexistant or inaccurate information is a barrier to effective engineering

 © 2010 Larry Snyder, CSE

Locality

- As with merchandizing, in parallel computing (actually, computing generally) its locality, locality, locality
- The main component of the CTA (after P) is λ and that value must be in our mind always
- Languages must guide us to exploit locality
  - locales in Chapel
  - places in X-10

 © 2010 Larry Snyder, CSE

Constrained Parallelism

- Finding the right set of facilities for parallel programming is a balancing act – enough flexibility to get the job done, but not enough to be a barrier to productivity
  - Correctness impacts
  - Performance impacts
  - Unlimited parallelism

 © 2010 Larry Snyder, CSE

Implicit vs Explicit Parallelism

- Allowing the compiler to find the parallelism is ideal, assuming it does a perfect job
- Being able to say where the parallelism is can guarantee that we achieve it our goals of performance, scalability and portability
- But neither extreme is perfect
  - Multiple levels (possibly like Chapel) might be best
  - Application specific with experts doing the heavy lifting might also work

 © 2010 Larry Snyder, CSE

Considering Languages You Reviewed

- Are there further comments regarding the languages you reviewed and the goals for the future?

© 2010 Larry Snyder, CSE