Language Approaches

Parallel Language Approaches

- Existing language + parallel system calls: MPI, PVM, pthreads, Linda, ...
- Existing language augmented with parallel language constructs: parbegin, forall, foreach, doall, ...
- Sequential language + very smart compiler
- Totally new language and paradigm, e.g. vectors, dataflow, etc.: NESL, ZPL, ...
- “Glue” language for coordination in the large

BSP

Bulk Synchronous Parallelism

- Bulk Synchronous Parallelism
- Model invented by Leslie Valiant at Harvard (Communications ACM, 33,8, Aug 1990)
- Some similarity to LogP (Berkeley), but model invented earlier
- Both a model and a library
- SPMD-style, with two types of communication
  - message-passing
  - remote DMA
- BSPlib originally implemented by Bill McColl at Oxford

Typical BSP Computer
BSP Parameters: “PLUGS”

- \( P \) = number of processors
- \( L \) = latency, cost in steps of achieving barrier synchronization
- \( U \) = unused (for acronym pronounceability)
- \( G \) = cost, in steps per word, of delivering message data (software dependent)
- \( S \) = processor speed (steps per second)
- (1 step is a single step on local data)

Improvement of \( G \) and \( L \)

- As \( G \) and \( L \) decrease, the performance becomes more scalable.
- With both equal to 1, the cost of accessing remote data is approximately the same as accessing local data and the calculation scales to the limit.

Nominal Data Points

<table>
<thead>
<tr>
<th>Architecture</th>
<th>processors</th>
<th>latency</th>
<th>message cost</th>
<th>proc. Speed</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOW (Pentium II)</td>
<td>10</td>
<td>10</td>
<td>30</td>
<td>90</td>
</tr>
<tr>
<td>SGI Power Challenge</td>
<td>16-512</td>
<td>1</td>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>Cray T-3E</td>
<td>32 - 2048</td>
<td>1</td>
<td>1</td>
<td>50</td>
</tr>
</tbody>
</table>

Floors = “floating point operation”

klop = 1000 flop

mflops = flop/second

Detailed measurements may be found on the BSP web site: [ftp://ftp.coml.ox.ac.uk/pub/Packages/BSP/papers/BSPparameters.ps.gz](ftp://ftp.coml.ox.ac.uk/pub/Packages/BSP/papers/BSPparameters.ps.gz)

BSP Supersteps

- A superstep is a parallel set of a series of local operations, followed by a barrier synth.
- A BSP computation consists of a sequence of supersteps.

Superstep Time Analysis

- Let \( S \) be a superstep, and let
  - \( w \) = maximum number of steps by any one processor during \( S \)
  - \( h_s \) = max number of messages sent by any one processor during \( S \)
  - \( h_r \) = max number of messages received by any one processor during \( S \)
- Then time for \( S \) is:
  \[ w + G \times \max(h_s, h_r) + L \]
BSP Programming Abstraction

- Data requestor need only issue a `get`.
- The user does not need to buffer messages because the BSP Library will supply buffering, if and when it is necessary.
- Optimization of communications is handled by the BSP library, not the user code.

BSP C calls:
Sequential followed by pure SPMD

```c
int nprocs;
basp_init(spmd_part, argc, argv);
nprocs=ReadInteger();
spmd_part();

void spmd_part()
{
basp_begin(nprocs);
... SPMD part of code ...
basp_end(void);
}
```

Simple Example of Initialization and Barriers

```c
void main(void)
{
basp_begin(basp_nprocs());
for (int i = 0;
i < basp_nprocs(); i++)
{
    if (basp_pid() == i)
    {
        printf("Hello from process ",
                "#d of #d\n",
                i, basp_nprocs());
        (flush(stdout);
    }
basp_sync();
}
basp_end();
}
```

Synchronization of a Subset of the Processors

- There isn’t any.

PVM vs. BSP

- PVM relies on
  - having matching pairs of SENDs and RECEIVES (send & recv)
  - the user providing buffering for messages
  - the user being aware of the type of communication in the system
    (ethernet, token ring, shared memory etc) and taking appropriate
    action to secure the best performance.

- BSP:
  - data requestor need only issue a FETCH
  - the user does not need to buffer messages because the BSP
    Library will supply buffering if and when it is necessary
  - optimization of communications is handled by the BSP library, not
    the user code.

BSP Library Functions

<table>
<thead>
<tr>
<th>Class</th>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initiation</td>
<td>basp_begin</td>
<td>Start of SPMD code</td>
</tr>
<tr>
<td></td>
<td>basp_end</td>
<td>End of SPMD code</td>
</tr>
<tr>
<td></td>
<td>basp_init</td>
<td>Initiate dynamic processes</td>
</tr>
<tr>
<td>Enquiry</td>
<td>basp_nprocs</td>
<td>Number of processes</td>
</tr>
<tr>
<td></td>
<td>basp_pid</td>
<td>Find my process ID</td>
</tr>
<tr>
<td></td>
<td>basp_time</td>
<td>Local time</td>
</tr>
<tr>
<td>Superopt</td>
<td>basp_sync</td>
<td>Barrier synchronization</td>
</tr>
<tr>
<td>DRMA</td>
<td>baspouncements</td>
<td>Make all globally visible</td>
</tr>
<tr>
<td></td>
<td>basp_remote</td>
<td>Remote global visibility</td>
</tr>
<tr>
<td>Memory Access</td>
<td>basp_data</td>
<td>Copy across memory</td>
</tr>
<tr>
<td>BSPM (Bulk)</td>
<td>basp_send</td>
<td>Send to remote space</td>
</tr>
<tr>
<td></td>
<td>basp_get</td>
<td>Get from remote space</td>
</tr>
<tr>
<td></td>
<td>basp_move</td>
<td>Move from space</td>
</tr>
<tr>
<td>High Performance</td>
<td>basp_type</td>
<td>Unidentified version of communication</td>
</tr>
<tr>
<td></td>
<td>basp_length</td>
<td>Length of communication</td>
</tr>
</tbody>
</table>
Methods for Communicating

- Message passing
- Direct Remote Memory Access

Direct Remote Memory Access (DRMA)

- Remotely accessed areas must be registered through bsp commands.
- `bsp_push_reg` registers the start of a local area to be available for global remote use.
- `bsp_put` deposits local data into registered remote memory on a target processor.
- `bsp_get` copies data from registered local memory into local memory
- `bsp_pop_reg` unregisters the area

Example: Reverse values over array of processors

```c
int reverse(int x)
{
    bsp_push_reg(&x, sizeof(int));
    bsp_sync();
    bsp_put(bsp_nprocs() - bsp_pid() - 1, &x, &x, 0, sizeof(int));
    bsp_sync();
    bsp_pop_reg(&x);
    return x;
}
```

Buffering Options

- **Buffered on destination**: Write at end of superstep, after all remote reads.
- **Unbuffered on destination**: Write at any time during superstep.
- **Buffered on source**: Read data from remote process at the end of a superstep, before any remote writes.
- **Unbuffered on source**: Read at any time during superstep.

Example: Sum values in a distributed array and redistribute to all

```c
int bsp_sum(int *xs, int nlocal) {
    int *local_sums = malloc(sizeof(int)*nlocal);
    if (local_sums == NULL)
        return -1;
    local_sums[0] = 0;
    for (int i = 1; i <= nlocal; i++)
        local_sums[i] = local_sums[i-1] + xs[i-1];
    bsp_push_reg(local_sums, 0);
    bsp_sync();
    local_sums[0] = 0;
    get_remotes:(
        local_sums = calloc(bsp_nprocs(), sizeof(int));
        if (local_sums == NULL)
            return -1;
        for (int i = 0; i < bsp_nprocs(); i++)
            local_sums[i] = local_sums[i] + xs[i];
        bsp_push_reg(local_sums, 0);
        bsp_sync();
        for (int i = 0; i < bsp_nprocs(); i++)
            result += local_sums[i];
    }
    free(local_sums);
    return result;
}
```
**Bulk Synchronous Message Passing**

- Choose tag size:
  ```c
  void bsp_set_tagsize(int *tag_size);
  ```

- Send to remote queue:
  ```c
  void bsp_send(uint pid, const void *tag, const void *payload, int payload_size);
  ```

- Number of messages in queue:
  ```c
  void bsp_queue_count(int *messages, int *queue_size);
  ```

- Getting the tag of a message:
  ```c
  void bsp_get_tag(int *status, void *tag);
  ```

- Move from queue:
  ```c
  void bsp_move(void *payload, int reception_bytes);
  ```

- A non-copying method for receiving a message:
  ```c
  int bsp_recv(void **tag_ptr, void **payload_ptr);
  ```

**Example: All-gather of a sparse vector**

```c
// Example code...
send
move
```

**NESL: “Nested Parallelism Language”**

- Guy Blelloch @ CMU
- A language coupled with a parallel complexity theory
- Functional, data-parallel, borrowing from APL, SETL, ML, Miranda, ...
- Implemented on a variety of parallel machines
- Concise specification of parallel algorithms

**Basis for Complexity**

- Organizing by vectors makes counting easier.
- VRAM: Vector Random-Access Machine
- Similar to PRAM, but ...
- Assumes scan (= parallel prefix) operations can be done in O(1) time.
- On a PRAM, we know this takes O(log n) time, so could just apply a log n factor to any result we obtain.
- On p << n processors, 1 VRAM is O(n/p)

**Blelloch’s scan primitive**

- associative binary operator ⊕
- identity I
- elements a₀, a₁, ..., aₙ₋₁
- returns
  ```c
  [I, a₀ ⊕ a₁, ..., (a₀ ⊕ a₁ ⊕ ... ⊕ aₙ₋₁)]
  ```
- We can get, in one additional parallel ⊕:
  ```c
  [a₀, (a₀ ⊕ a₁), ..., (a₀ ⊕ a₁ ⊕ ... ⊕ aₙ₋₁)]
  ```
Scan Examples

- arg vector: [3, 5, 2, 7, 6, 1, 4]
- results:
  - +-scan: [0, 3, 8, 10, 17, 23, 24]
  - max-scan: [\infty, 3, 5, 7, 7, 7]
  - min-scan: [\infty, 3, 3, 2, 2, 2, 1]
  - copy: [3, 3, 3, 3, 3, 3, 3]
  (what is operator and Identity?)

More Scan Examples

- results:
  - or-scan: [F, F, T, T, T, T, T]
  - "enumerate" operation:
    add up the number of T's to the left:
    enumerate => [0, 0, 1, 2, 2, 3, 3]

More Scan Examples

- "enumerate-x" operation:
  add up the number of x's to the left:
  enumerate-T => [0, 0, 1, 2, 2, 3, 3]
- "back-enumerate-x" operation:
  add up the number of x's to the right:
  back-enumerate-T => [2, 2, 1, 1, 0, 0, 0]

Permutation

- permute(Vector, PermutationVector)

  permute([3, 1, 5, 1, 2, 4],
          [3, 1, 5, 1, 2, 4])

  => [1, 5, 3, 2, 1, 4]

Splitting

- Packs Vector elements corresponding to F flag in lower part, T flag in upper part:

  split([5, 7, 3, 1, 4, 2, 7, 2],
        [T, T, T, T, F, F, T, T])

  => [4, 2, 2, 5, 7, 3, 1, 7]

Exercise

- How would you implement split using scan operations?
  - Determine new index for each element:
    - Enumerate F determines indices for lower part
    - Back-enumerate T using complement vector determines indices for upper part
  - Compute vector of length- elements above
  - Select one index or the other, based upon original T-F vector
  - Permute
  - About 5 VRAM operations
Example

- split([5, 7, 3, 1, 4, 2, 6, 0], [T, T, T, T, F, F, T, F]):
- enumerate-F => [0, 0, 0, 0, 0, 1, 2, 2]
- back-enum-T => [4, 3, 2, 1, 1, 1, 0, 0]
- subtract back-enum from length-1 (7) => [3, 4, 5, 6, 6, 6, 7, 7]
- Select from one of the two vectors based on T-F => [3, 4, 5, 6, 0, 1, 7, 2]
- Permute => [4, 2, 0, 5, 6, 3, 1, 7]

Using split to Implement Radix Sort

- Assume d-bit numbers
- V = original vector of numbers;
  for l = 0 to d-1
    Flags = lth bit of numbers;
    V = split(V, Flags);
- Time O(d) on VRAM

Representation of Nested Lists

- Customarily we use pointer structures
- Instead, NESL / VRAM uses a bit vector to represent segment boundaries:
- Example: The head-flags method
  [[3, 2, 1], [5, 7], [6, 4, 0]]
- This method cannot represent empty segments however

Example: The lengths method
[[3, 2, 1], [], [5, 7], [6, 4, 0]]
[3, 0, 2, 4]

Example: The head-pointers method
[[3, 2, 1], [], [5, 7], [6, 4, 0]]
[0, 3, 3, 5]

Segmented scan operations

- These are scan operations done separately within each segment
- Example with head-flags method
  [3, 2, 1, 5, 7, 6, 4, 0]
- seg+-scan =>
  [0, 3, 5, 0, 5, 0, 6, 10]

Segmented scan operations

- These are scan operations done separately within each segment
- Example with head-flags method
  [3, 2, 1, 5, 7, 6, 4, 0]
- seg+-scan =>
  [0, 3, 5, 0, 5, 0, 6, 10]
Enumerate

- Add up the number of T's to the left

Basic NESL Philosophy

- Try to convert algorithms to exploit scan primitives as much as possible:
  - $O(1)$ VRAM computations
    - length of a Vector
    - sum of a Vector
    - permute(Vector, Index Vector)
    - $p+x$ (Vector1, Vector2) (pair-wise sum)
    - $+$-scan(Vector)
    - max-scan(Vector)
    - etc.

NESL Set-Patterns
(after Miranda)

- $\{\text{pattern} : \text{var} \in \text{Vector}\}$
- $\{\text{pattern} : \text{var1} \in \text{Vector1}; \text{var2} \in \text{Vector2}\}$

Example:
- $\{f(x) : x \in V\}$ is essentially a map operation
- $\{a + b : a \in [1, 3]; b \in [5, 9]\} \Rightarrow [6, 12]$

matrix-multiply

- $\text{matrix-multiply}(A, B) =$
  - $\{ \sum (x \cdot y : x \in \text{rowA}; y \in \text{colB})$
  - $: \text{colB} \in \text{transpose}(B)$
  - $: \text{rowA} \in A$

Quicksort in NESL
(similar to Quicksort in SISAL)

- $\text{function qsort(a) =}$
  - if( $\#a < 2$ ) then a else
  - let pivot = a[$\#a / 2$];
  - lesser = $\{e \in a : e < \text{pivot}\}$;
  - equal = $\{e \in a : e == \text{pivot}\}$;
  - greater = $\{e \in a : e > \text{pivot}\}$;
  - result = $\{\text{qsort(v) : v in [lesser, greater]}$ in result[0] ++ equal ++ result[1]

Quick sort Implementation
using Segmented Scan

- $[3, 1, 2, 7, 6, 11, 5, 4, 9, 10, 12, 8]$
  - pivot = 3
  - $[=, <, <, >, >, >, >, >, >, >, >]$ 3-way split & segment
  - $[1, 2, 3, 7, 6, 11, 5, 4, 9, 10, 12, 8]$
  - segmented splits based on pivots
  - $[1, 2, 3, 4, 5, 6, 7, 11, 9, 10, 8, 11, 12]$
  - etc.
### Quicksort analysis

- Worst case $O(n)$ VRAM steps
- Average case $O(\log n)$ VRAM steps

### Convex Hull Algorithm ("Quickhull")

- **Problem:** Given $n$ points in the plane, determine the subset that lie on the perimeter of the smallest convex region containing all of the points.

### Convex Hull Algorithm ("Quickhull")

- Begin by finding the two extrema, $L$ and $R$, in the $x$ dimension.
- $L$ and $R$ will be in the convex hull.
- Imagine a line between these extrema.

### Convex Hull Algorithm ("Quickhull")

- Find the point $P$ above and farthest from line $LR$, if any.
- $P$ will also be in the convex hull.

### Convex Hull Algorithm ("Quickhull")

- Repeat the process with lines $LP$ and $PR$, until there is no point outside.
- The new points, $P'$, etc. are in the convex hull.
Convex Hull Algorithm ("Quickhull")

- Repeat the process with lines LP and PR, until there is no point outside.
- The new points, P', etc. are in the convex hull.

Meanwhile, also be doing this with points on the other side of LR (call those points Q, Q', ...)

Representation as NESL Lists

NESL Program for Quickhull (1)

% Used to find the distance of a point (o) from a line (line).
% function cross_product(o,line) =
%   let (xo,yo) = o;
%   ((x1,y1),(x2,y2)) = line;
%   in (x1-xo)*(y2-yo) - (y1-yo)*(x2-xo);

% Given two points on the convex hull (p1 and p2), hsplit finds all the points on the hull between p1 and p2 (clockwise), inclusive of p1 but not of p2.
% function hsplit(points,p1,p2) =
%    let cross = {cross_product(p,(p1,p2)) : p in points};
%    packed = {p in points; c in cross | plus p(c)};
%    in if ( #packed < 2) then [p1] + packed
%      else let pm = points[max_index(cross)];
%             in flatten({hsplit(packed, p1,p2): p1 in [p1,pm]; p2 in [pm,p2]});

NESL Program for Quickhull (2)

% Finds the points with minimum and maximum x coordinates, and then finds the upper and lower convex hull: the part clockwise from min to max (upper) and clockwise from max to min (lower).
% function convex_hull(points) =
%    let x = {x : (x,y) in points};
%    minx = points[min_index(x)];
%    maxx = points[max_index(x)];
%    in hsplit(points,minx,maxx) ++ hsplit(points,maxx,minx);

Analysis

- Similar to quicksort
- For "well-distributed" set of points, requires O(log n) VRAM steps overall.
- In worst case, can require O(n) VRAM steps.
### NESL Reference Card (1)

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>define</code></td>
<td><code>define a = b + c</code></td>
</tr>
<tr>
<td><code>if</code></td>
<td><code>if x &gt; 0 then</code></td>
</tr>
<tr>
<td><code>while</code></td>
<td><code>while i &lt; n do</code></td>
</tr>
<tr>
<td><code>for</code></td>
<td><code>for i = 0 to n do</code></td>
</tr>
<tr>
<td><code>proc</code></td>
<td><code>proc myproc(x, y)</code></td>
</tr>
<tr>
<td><code>return</code></td>
<td><code>return x</code></td>
</tr>
<tr>
<td><code>call</code></td>
<td><code>call myproc(x)</code></td>
</tr>
<tr>
<td><code>break</code></td>
<td><code>break</code></td>
</tr>
<tr>
<td><code>continue</code></td>
<td><code>continue</code></td>
</tr>
<tr>
<td><code>assert</code></td>
<td><code>assert x &gt; 0</code></td>
</tr>
<tr>
<td><code>assert false</code></td>
<td><code>assert false</code></td>
</tr>
<tr>
<td><code>assert true</code></td>
<td><code>assert true</code></td>
</tr>
<tr>
<td><code>assert</code></td>
<td><code>assert x &gt; 0</code></td>
</tr>
<tr>
<td><code>assert false</code></td>
<td><code>assert false</code></td>
</tr>
<tr>
<td><code>assert true</code></td>
<td><code>assert true</code></td>
</tr>
<tr>
<td><code>assert</code></td>
<td><code>assert x &gt; 0</code></td>
</tr>
<tr>
<td><code>assert false</code></td>
<td><code>assert false</code></td>
</tr>
<tr>
<td><code>assert true</code></td>
<td><code>assert true</code></td>
</tr>
<tr>
<td><code>assert</code></td>
<td><code>assert x &gt; 0</code></td>
</tr>
<tr>
<td><code>assert false</code></td>
<td><code>assert false</code></td>
</tr>
<tr>
<td><code>assert true</code></td>
<td><code>assert true</code></td>
</tr>
<tr>
<td><code>assert</code></td>
<td><code>assert x &gt; 0</code></td>
</tr>
<tr>
<td><code>assert false</code></td>
<td><code>assert false</code></td>
</tr>
<tr>
<td><code>assert true</code></td>
<td><code>assert true</code></td>
</tr>
<tr>
<td><code>assert</code></td>
<td><code>assert x &gt; 0</code></td>
</tr>
<tr>
<td><code>assert false</code></td>
<td><code>assert false</code></td>
</tr>
<tr>
<td><code>assert true</code></td>
<td><code>assert true</code></td>
</tr>
<tr>
<td><code>assert</code></td>
<td><code>assert x &gt; 0</code></td>
</tr>
<tr>
<td><code>assert false</code></td>
<td><code>assert false</code></td>
</tr>
<tr>
<td><code>assert true</code></td>
<td><code>assert true</code></td>
</tr>
<tr>
<td><code>assert</code></td>
<td><code>assert x &gt; 0</code></td>
</tr>
<tr>
<td><code>assert false</code></td>
<td><code>assert false</code></td>
</tr>
<tr>
<td><code>assert true</code></td>
<td><code>assert true</code></td>
</tr>
<tr>
<td><code>assert</code></td>
<td><code>assert x &gt; 0</code></td>
</tr>
<tr>
<td><code>assert false</code></td>
<td><code>assert false</code></td>
</tr>
<tr>
<td><code>assert true</code></td>
<td><code>assert true</code></td>
</tr>
<tr>
<td><code>assert</code></td>
<td><code>assert x &gt; 0</code></td>
</tr>
<tr>
<td><code>assert false</code></td>
<td><code>assert false</code></td>
</tr>
<tr>
<td><code>assert true</code></td>
<td><code>assert true</code></td>
</tr>
<tr>
<td><code>assert</code></td>
<td><code>assert x &gt; 0</code></td>
</tr>
<tr>
<td><code>assert false</code></td>
<td><code>assert false</code></td>
</tr>
<tr>
<td><code>assert true</code></td>
<td><code>assert true</code></td>
</tr>
</tbody>
</table>

### NESL Reference Card (2)

[and there's more]