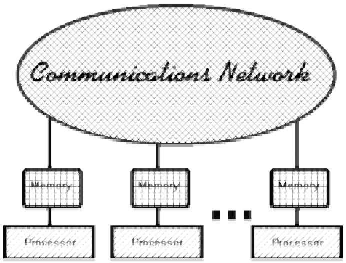
BSP
Bulk Synchronous Parallelism

BSP

- Bulk Synchronous Parallelism
- Model invented by Leslie Valiant at Harvard
- Some similarity to LogP, but model invented earlier
- Both a model and a library
- SPMD-style, but has remote DMA as well as message-passing
- BSPlib originally implemented by Bill McColl at Oxford

Typical BSP Computer



BSP Parameters: lsgp

- l = latency, cost in steps of achieving barrier synchronization
- s = processor speed (steps per second)
- g = cost, in steps per word, of delivering message data
- p = number of processors
- (1 step is a single step on local data)

Typical g (comm. per word) and l (barrier latency)

- For a network of workstations, with say processors capable of 20 Megaflops (sustained), might have g values in the range of a few 100 flops per word transferred, and l values of the order of 10,000 to 100,000 flops.
- Machines such as the Cray T3E, with sustainable node performance of the order of 45 Megaflops, could have g values as low as 1.5 to 2.5 flops per word and l values of a few hundred flops.

Typical g (comm. per word) and l (barrier latency)

- Silicon Graphics Power Challenge, with a sustainable node performance of 75 Megaflops, could have g values of the order of 10 flops per word and l values of the order of 1000 flops.

Limits on g and l

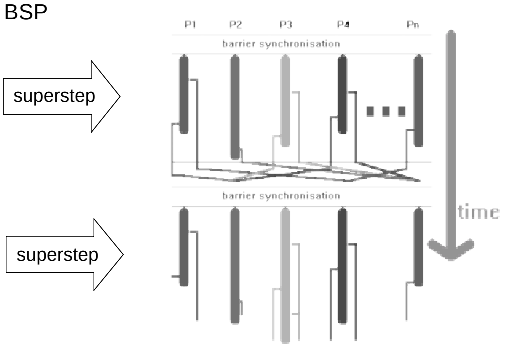
- With g → l and l →1, the performance becomes more scalable.

- 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.

BSP Supersteps

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

BSP



superstep

superstep

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:

  $w + g*\max(h_s,h_r) + l$

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.

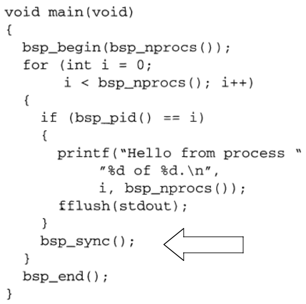
BSP C calls: Pure SPMD

```
bsp_begin(maxprocs);

… SPMD part of code …

bsp_end();
```

BSP C calls:
Sequential followed by pure SPMD

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

void spmd_part()
  {
  bsp_begin(nprocs);
  … SPMD part of code ...
  bsp_end(void);
  }
```

Simple Example of Initialization and Barriers

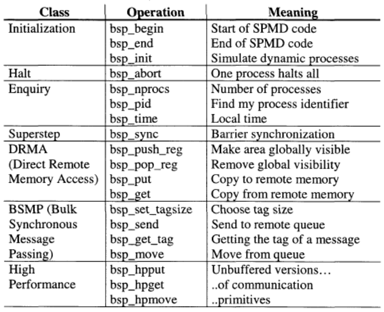


```
void main(void)
{
  bsp_begin(bsp_nprocs());
  for (int i = 0;
       i < bsp_nprocs(); i++)
  {
    if (bsp_pid() == i)
    {
      printf("Hello from process "
           "%d of %d.\n",
           i, bsp_nprocs());
      fflush(stdout);
    }
    bsp_sync();
  }
  bsp_end();
}
```

Synchronization of a **Subset** of the Processors

- There isn’t any.

BSP Library Functions



Class	Operation	Meaning
Initialization	bsp_begin	Start of SPMD code
	bsp_end	End of SPMD code
	bsp_init	Simulate dynamic processes
Halt	bsp_abort	One process halts all
Enquiry	bsp_nprocs	Number of processes
	bsp_pid	Find my process identifier
	bsp_time	Local time
Superstep	bsp_sync	Barrier synchronization
DRMA (Direct Remote Memory Access)	bsp_push_reg	Make area globally visible
	bsp_pop_reg	Remove global visibility
	bsp_put	Copy to remote memory
	bsp_get	Copy from remote memory
BSMP (Bulk Synchronous Message Passing)	bsp_set_tagsize	Choose tag size
	bsp_send	Send to remote queue
	bsp_get_tag	Getting the tag of a message
	bsp_move	Move from queue
High Performance	bsp_hpput	Unbuffered versions…
	bsp_hpget	..of communication
	bsp_hpmove	..primitives

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

Direct Remote Memory Access

Registration

```
void bsp_push_reg(const void *ident,
                  int size);
void bsp_pop_reg(const void *ident);
```

Copy to remote memory

```
void bsp_put(int pid, const void *src,
       void *dst, int offset, int nbytes);
void bsp_hpput(int pid, const void *src,
       void *dst, int offset, int nbytes);
```

Copy from remote memory

```
void bsp_get(int pid, const void *src,
       int offset, void *dst, int nbytes);
void bsp_hpget(int pid, const void *src,
       int offset, void *dst, int nbytes);
```

Example: Reverse values over array of processors

```
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

```
int bsp_sum(int *xs, int nelem) {
  int *local_sums,i,j,result=0;
  for(j=0;j<nelem;j++) result += xs[j];
  bsp_push_reg(&result,sizeof(int));
  bsp_sync();

  local_sums = calloc(bsp_nprocs(),sizeof(int));
  if (local_sums==NULL)
    bsp_abort("{bsp_sum} no memory for %d int",bsp_nprocs());
  for(i=0;i<bsp_nprocs();i++)
    bsp_hpget(i,&result,0,&local_sums[i],sizeof(int));
  bsp_sync();

  result=0;
  for(i=0;i<bsp_nprocs();i++) result += local_sums[i];
  bsp_pop_reg(&result);
  free(local_sums);
  return result;
}
```

local sums

get remotes

sum remotes

Bulk Synchronous Message Passing

Choose tag size:

```
void bsp_set_tagsize (int *tag_nbytes);
```

Send to remote queue:

```
void bsp_send(int pid, const void *tag,
       const void *payload,
       int payload_nbytes);
```

Number of messages in queue:

```
void bsp_qsize(int *nmessages,
       int *accum_nbytes);
```

Getting the tag of a message:

```
void bsp_get_tag(int *status, void *tag);
```

Move from queue:

```
void bsp_move(void *payload,
       int reception_bytes);
```

A non-copying method for receiving a message:

```
int bsp_hpmove(void **tag_ptr,
       void **payload_ptr);
```

Example: All-gather of a sparse vector

```
int all_gather_sparse_vec(float *dense,int n_over_p,
                          float **sparse_out,
                          int **sparse_ivec_out){
  int global_idx,i,j,tag_size,
      nonzeros,nonzeros_size,status, *sparse_ivec;
  float *sparse;

  tag_size = sizeof(int);
  bsp_set_tagsize(&tag_size);
```

Parallel Languages

Parallel Language Approaches

- Existing language + parallel system calls
- Existing language augmented with parallel language constructs
- Sequential language + very smart compiler
- Totally new language and paradigm, e.g. vectors, dataflow, etc.
- “Glue” language for coordination in the large

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 $\oplus$
- identity I
- elements $a_0, a_1, \ldots, a_{n-1}$
- returns
  $[I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})]$
- We can get, in one additional parallel $\oplus$:
  $[a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-1})]$

Scan Examples

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

More Scan Examples

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

More Scan Examples

- arg vector [F, T, T, F, T, F, F]
- "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],
        [2, 4, 1, 0, 3, 5])

=>      [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, F])
=>  [4, 2, 2, 5, 7, 3, 1, 7]
```

Exercise

- How would you implement split using scan 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]

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

Using split to Implement Radix Sort

- Assume d-bit numbers
- V = original vector of numbers;

```
for I = 0 to d-1
    {
    Flags = $i^{th}$ 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]]

  [T, F, F, T, F, T, F, F]
- This method cannot represent empty segments however

Representation of Nested Lists

- Example: The **lengths method**

  [[3, 2, 1], [5, 7], [6, 4, 0]]

  [3, 0, 2, 4]
- Example: The **head-pointers**

  [[3, 2, 1], [5, 7], [6, 4, 0]]

  [0, 3, 3, 5]

Segmented scan operations

- These are scan operations done separately within each segment
- Examplewith **head-flags method**

  [3, 2, 1, 5, 7, 6, 4, 0]

  [T, F, F,T, F, T, F, F]
- seg-+-scan =>

  [0, 3, 5, 0, 5, 0, 6, 10]

Segmented scan operations

- These are scan operations done separately within each segment
- Examplewith **head-flags method**

  [3, 2, 1, | 5, 7, | 6, 4, 0]

  [T, F, F, | T, F, | T, F, F]
- 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+(Vector1, Vector2) (pair-wise sum)
    - +-scan(Vector)
    - max-scan(Vector)
    - etc.

NESL Set-Patterns

(after Miranda)

- {*pattern* : *var* in *Vector*}
- {*pattern* : *var1* in *Vector1; var2* in *Vector2*}

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

---

matrix-multiply

- matrix-multiply(A, B) =

```
{
  { sum( {x*y: x in rowA; y in colB} )
        : colB in transpose(B)
   }
  : rowA in A
}
```

Quicksort in NESL

(similar to Quicksort in SISAL)

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

Quicksort Implementation using Segmented Scan

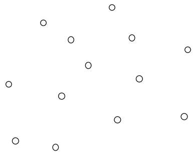
- [3, 1, 2, 7, 6, 11, 5, 4, 9, 10, 12, 8]
  pivot = 3
  [=, <, <, >, >, >, >, >, >, >, >, >]
  3-way split & segment
  [1, 2,| 3,| 7, 6, 11, 5, 4, 9, 10, 12, 8]
- segmented splits based on pivots
  [1,| 2,| 3,| 6, 5, 4,| 7,| 11, 9, 10, 12, 8]
  [1,| 2,| 3,| 4, 5,| 6,| 7,| 9, 10, 8,| 11| 12]
- etc.

Quicksort analysis

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

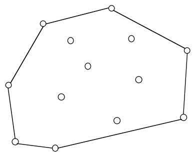
Convex Hull Algorithm (“Quickhull”)

- 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.



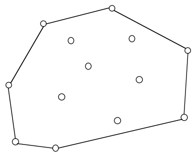
Convex Hull Algorithm (“Quickhull”)

- 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.



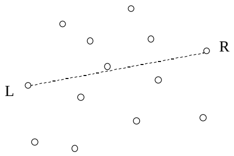
Convex Hull Algorithm (“Quickhull”)

- Begin by finding the two extrema on the x axis.



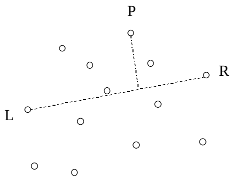
Convex Hull Algorithm (“Quickhull”)

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



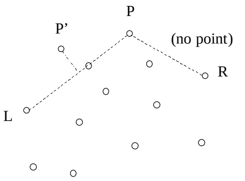
Convex Hull Algorithm (“Quickhull”)

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



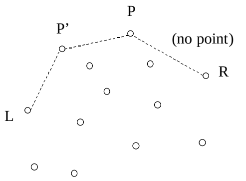
Convex Hull Algorithm (“Quickhull”)

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



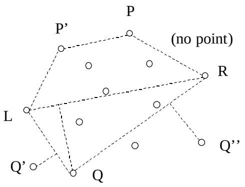
Convex Hull Algorithm (“Quickhull”)

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

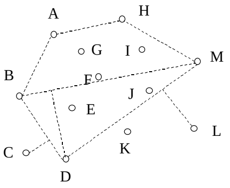


Convex Hull Algorithm (“Quickhull”)

- Meanwhile, also be doing this with points on the other side of LR (call those points Q, Q’, …)



Representation as NESL Lists



[A B C D E F G H I J K L M]

⇩

[B [A F G H I] [C D E J K L] M]

⇩

[B [A] H [C] D [E J K] L M]

⇩

[B A H C D [K] L M]

⇩

[B A H C D L M]

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 | plusp(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 minx to
  maxx (upper) and clockwise from maxx to minx (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);
```

NESL Program for Quickhull

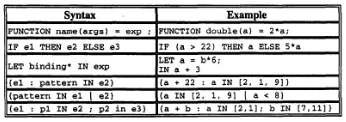
```
%
This example finds the convex hull in 2 dimensions for a set of points
using the QuickHull algorithm.  The algorithm is described in the NESL
language definition.
%

% 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
```

Analysis

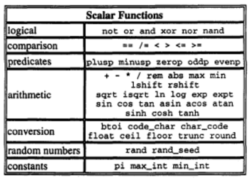
- Similar to quicksort
- For “well-distributed” set of points, requires O(log n) VRAM steps overall.
- In worst case, can require O(n) VRAM steps.

Execution as a NESL Program

NESL Reference Card (1)



Syntax	Example
`FUNCTION name(args) = exp ;`	`FUNCTION double(a) = 2*a;`
`IF e1 THEN e2 ELSE e3`	`IF (a > 22) THEN a ELSE 5*a`
`LET binding* IN exp`	`LET a = b*6;` `IN a + 3`
`{e1 : pattern IN e2}`	`{a + 22 : a IN [2, 1, 9]}`
`{pattern IN e1 | e2}`	`{a IN [2, 1, 9] | a < 8}`
`{e1 : p1 IN e2 ; p2 in e3}`	`{a + b : a IN [2,1]; b IN [7,11]}`



Scalar Functions	
logical	`not or and xor nor nand`
comparison	`== /= < > <= >=`
predicates	`plusp minusp zerop oddp evenp`

arithmetic	`+ - * / rem abs max min`
	`lshift rshift`
	`sqrt isqrt ln log exp expt`
	`sin cos tan asin acos atan`
	`sinh cosh tanh`

conversion	`btoi code_char char_code` `float ceil floor trunc round`
random numbers	`rand rand_seed`
constants	`pi max_int min_int`

NESL Reference Card (2)

[and there’s more]