BSP
Bulk Synchronous Parallelism

What is it?

- A computational model
- A library (for C, Fortran77, Python, …)
- A way to do remote shared memory (+ message passing + SPMD)

BSP

- **B**ulk **S**ynchronous **P**arallelism
- Based on “Bridging Model” proposed by Prof. Leslie Valiant at Harvard (CACM, Aug. 1990)
- “bridging” parallel and sequential processing
- Generalizes PRAM.
- An **objective**: predictable performance, independent of architecture.
- Some similarity to Berkeley “LogP” model, but invented earlier. (BSP = LogP - overhead + barriers)

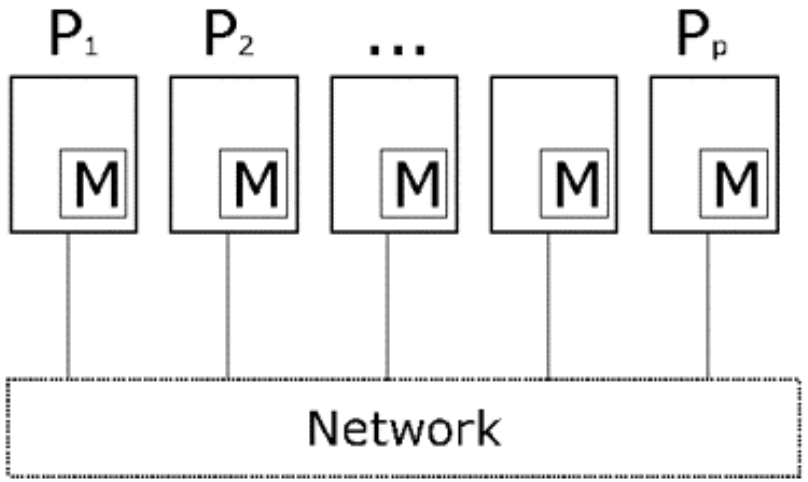
BSP

- Programming styles:
  - SPMD
  - Remote DMA
  - Message-passing
- BSP Library implemented by Bill McColl, et al. at Oxford University
- A *language*, BSP-L, was proposed by Cheatham at Harvard.
  - “forall” type of language, with arrays, etc.
  - Not clear this is still being pursued.

Historic Reference: BSP-L Matrix Multiply (1995)

```
Let A,B: array(< 1..N, 1..N >, int)
Let tsize be N/exp(p,1/3)
Let TA: A tiled tsize by tsize
Let TB: B tiled tsize by tsize
Let d be dim(TA,1)                              /* d= p^(1/3) */
Let C: array(< 1..N, 1..N >, int) tiled tsize by tsize/d

DEF_THREAD mat(index:tuple ;
           d:int, tsize:int, A:array(<1..tsize, 1..tsize>, int),
           B:array(<1..tsize, 1..tsize>,int);
           D:array(<1..tsize, 1..tsize/d>, int) initially 0)

            Let C0: array(<1..tsize, 1..tsize>, int)
            Let tsize1 be tsize/d            /* tsize1 = N/p^(2/3) */
            Let TC0: C0 tiled tsize by tsize1
            Let C: array(<1..tsize, 1..tsize1, 1..d>, int)

/*superstep 2: multiplication of tiles and redistribution of smaller tiles */

            For s in 1 to tsize do
              For q in 1 to tsize do
                For r in 1 to tsize do
                   C0[s, r] <- C0[s, r] + A[s, q] * B[q, r]
            For r in 1 to d
                 put(<index[1], r, index[3]>, TC0, <1..1, r..r>, _)
            For q in 1 to d
               get(_, C, <1..tsize, 1..tsize1, q>, _)

            synch

/* superstep 3: final summations */

            For q in 1 to tsize
              For r in 1 to tsize1
                For s in 1 to d do
                   D[q,r] <- D[q,r] + C[q,r,s]
END_THREAD

/* Main program: start p threads and name them according to a cube */

Forall i in 1 to d
       j in 1 to d
         k in 1 to d do

/* superstep 1: distribution of tiles of size N/p^(1/3) by thread call*/

           mat(<i,j,k>;d,tsize,TA[i,j],TB[j,k];C[i, k + (j-1)*d] )

/* superstep 4: write back tiles via thread return */
```

BSP Computer Model



BSP Supersteps

- A BSP computation consists of a **sequence of supersteps**.
- A superstep is a parallel set of
  - a series of **local** operations on virtual processors, followed by
  - message-based **communication**, followed by
  - a **barrier** synchronization.
- It is quite possible that # of virtual processors exceeds # of physical.

Superstep

Time

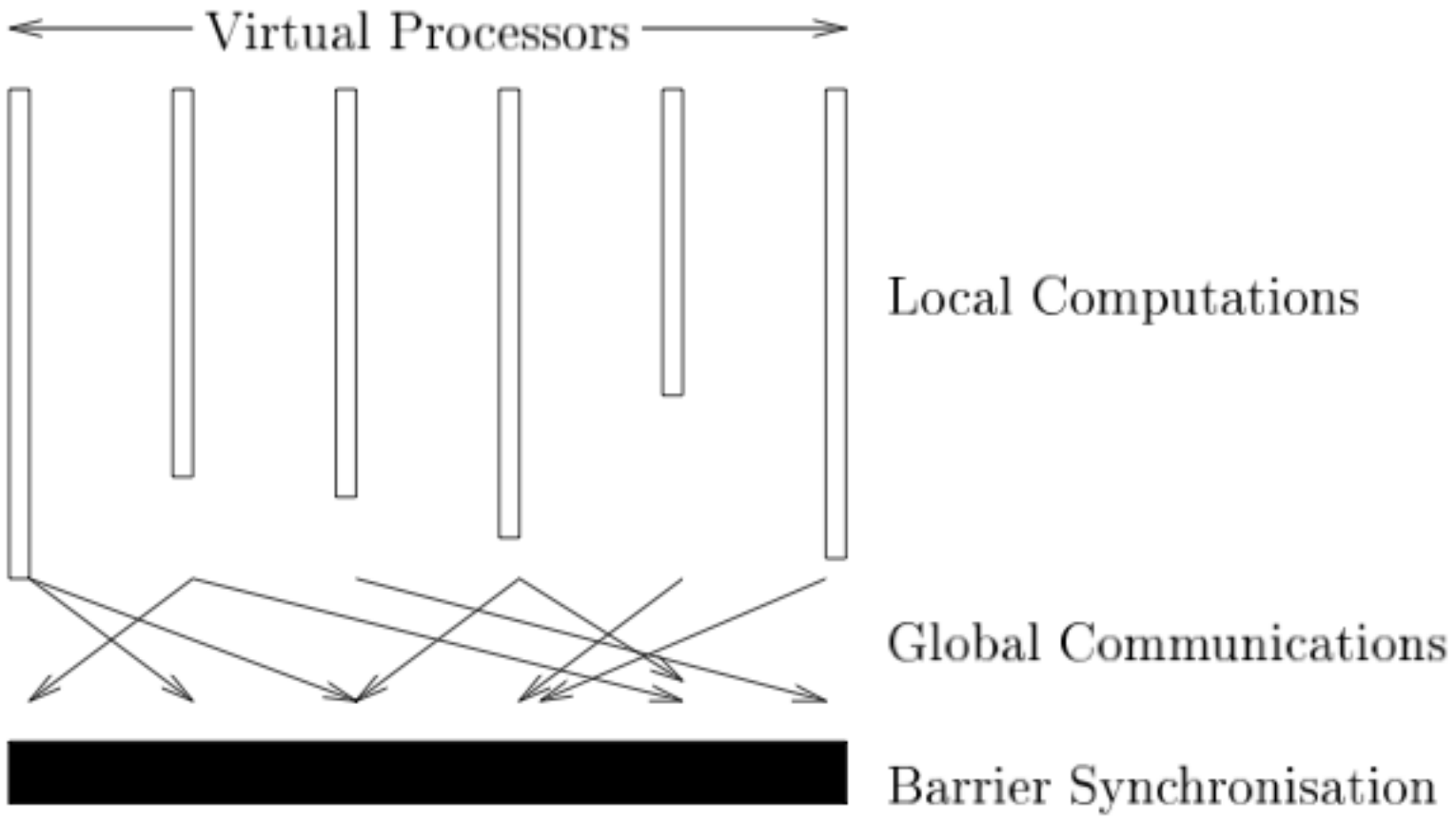


Figure 1: A Superstep

Processor View of Superstep

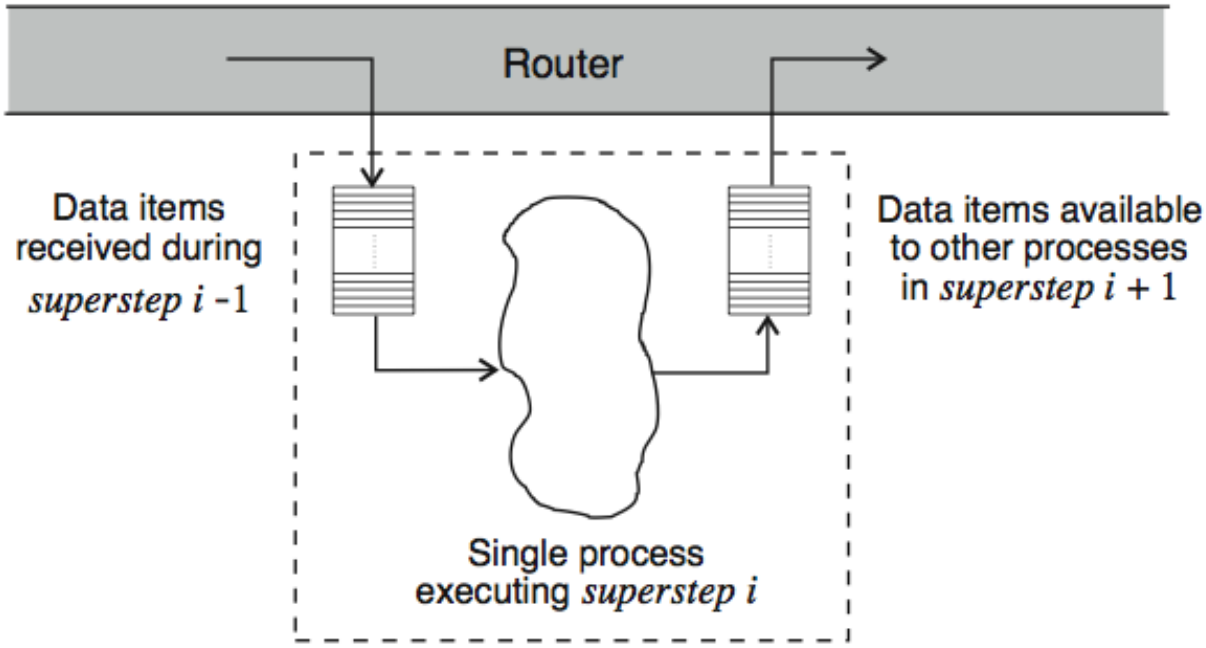


Figure 3.1: Conceptual view of a superstep execution

BSP Parameters: L s g p

(Usually L is lower-case, but that looks like 1 in some fonts.)

- s = processor **speed** (in Mflops per second) [flop = “floating-point operation”]
- L = **latency**, cost, of achieving barrier synchronization (in flops-equivalent time)
- g = cost of delivering message data (i.e. bandwidth) (in flops-equivalent **per word**)
- p = number of processors

L g s p

- The idea is that, once these parameters are available for a specific architecture,

  the performance of a BSP program can be **predicted**.

- Obviously some approximations are going to be involved.

Time cost of 1 superstep

- Time = w + g h + L
- w = time of the sequence of operations (in Mflops)
- g = transmission bandwidth (flop equivalent / word)
- h = upper bound on number of message **words** sent or received in a superstep
- L = barrier latency (flop equivalents)

BSP Parameter s: instruction rate (in Mflops per second)

g and l parameters are normalized by the **instruction rate, s**, of each processor.

Lower-bound for s: measures the **cost of an inner product**, where O(n) operations are performed on a data structure of size n. The value of n is chosen to be **far greater than the cache size** on each processor. This benchmark therefore gives a lower-bound megaflop rate for the processor as **each arithmetic operation induces a cache miss**.

Upper-bound for s: measures the **cost of a dense matrix multiplication**, where $O(n^3)$ operations are performed on a data structures of size n^2. Because a large percentage of the computation can be kept in cache, this benchmark gives an upper-bound megaflop rate for the processor.

The values of s given in the **table is an average** of the upper and lower bound values for s.

BSP Parameter l: barrier cost (in flops)

The cost of a barrier synchronization comes in two parts:

The **cost caused by the variation in the completion times** of the computation steps that participate. There is not much that an implementation can do about this, but it does suggest that balance in the computation parts of a superstep is a good thing.

The **cost of reaching a globally-consistent state in all of the processors**. This depends on the communication network, whether or not special-purpose hardware is available for synchronizing, and the way in which interrupts are handled by processors.

Parameter l captures the *latter* of these costs. The diameter of the communication network, or at least the **length of the longest path** that allows state to be moved from one processor to another clearly **imposes a lower bound on l**. However, it is also affected by many other factors, so that, in practice, an accurate value of l for each parallel architecture is obtained empirically.

BSP Parameter g: communication bandwidth (in flops per word)

The parameter g is **related to the bisection bandwidth** of the communication network but they are not equivalent --- g also depends on factors such as:

the protocols used to interface with and within the communication network,
buffer management by both the processors and the communication network,
the routing strategy used in the communication network, and
the BSP runtime system.

So g is bounded below by the r**atio of p to the bisection bandwidth**, suitably normalized, but may be much larger because of these other factors. Only a very unusual network would have a bisection bandwidth that grew faster than p, so g is a monotonically increasing function of p. The precise value of g is, in practice, determined empirically for each parallel computer, by running suitable benchmarks.

Note that g is not the single-word delivery time, but the **single-word delivery time under continuous traffic conditions**. This difference is subtle but crucial.

Example BSP Parameters (1997)

http://www.bsp-worldwide.org/implmnts/oxtool/params.html

MACHINE	s *(Mflop/s)*	p *(no. procs)*	l *(flops)*	g *(flops/word)*
BSP cluster	88	1	128	1.3
		2	5654	33.5
		4	11759	31.5
		8	18347	30.9
SGI PowerChallenge	74	1	226	0.50
		2	1132	10.2
		3	1496	9.5
		4	1902	9.3
Origin 2000	100.7	1	286	1.36
		2	804	8.26
		3	1313	8.36
		4	1789	10.24
		5	2474	11.06
		6	2963	12.25
		7	3867	14.28

Limits on g (bandwidth) and L (barrier cost)

- With g → 1 flops/word and L → 1 flops, performance becomes more scalable.
- With both equal to 1, the cost of accessing remote data is approximately the same as accessing local data.
- The model can emulate a PRAM in this limiting case.

BSP Library Primitives

Table 2. A quick reference to the 20 primitives of BSPlib. An empty field for return type or parameters denotes `void`.

Class	Primitive	Meaning	Return type	Parameters
SPMD	`bsp_begin`	Start of SPMD part		`int maxprocs`
	`bsp_end`	End of SPMD part		
	`bsp_init`	Simulate dynamic processes		`void(*spmd_part)(void), int argc, char *argv[]`
	`bsp_abort`	One process halts all		`char *format,...`
	`bsp_nprocs`	Number of processes	`int`	
	`bsp_pid`	My process identifier	`int`	
	`bsp_time`	Elapsed local time	`double`	
	`bsp_sync`	Barrier synchronisation		
DRMA	`bsp_push_reg`	Make area globally visible		`const void *ident, int size`
	`bsp_pop_reg`	Remove global visibility		`const void *ident`
	`bsp_put`	Copy to remote memory		`int pid, const void *src, void *dst, int offset, int nbytes`
	`bsp_hpput`	Unbuffered put		`int pid, const void *src, void *dst, int offset, int nbytes`
	`bsp_get`	Copy from remote memory		`int pid, const void *src, int offset, void *dst, int nbytes`
	`bsp_hpget`	Unbuffered get		`int pid, const void *src, int offset, void *dst, int nbytes`
BSMP	`bsp_set_tagsize`	Set tag size		`int *tag_nbytes`
	`bsp_send`	Send to remote queue		`int pid, const void *tag, const void *payld, int payld_nbytes`
	`bsp_qsize`	Number of messages in queue		`int *nmessages, int *accum_nbytes`
	`bsp_get_tag`	Get message tag		`int *status, void *tag`
	`bsp_move`	Move message from queue		`void *payload, int reception_nbytes`
	`bsp_hpmove`	High performance move	`int`	`void **tag_ptr, void **payload_ptr`

Methods for Communicating

- Each process **registers** private memory in its physical address space.

- Communication is via either
  - Message passing
  - Direct Remote Memory Access (DRMA)

Registration vs. Allocation

- A given data structure is not necessarily at the same address in every processor.
- Registration establishes a correspondence between the variable name and the actual location of the data.

First BSP Programming Abstraction: DRMA-Direct Remote Memory Access

- Data requestor issues a ***get***.
- 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.

Direct Remote Memory Access (DRMA)

- **Remotely accessed areas** must be **registered** through bsp commands.
- **bsp_push_reg** *registers* the start of a local area into which a remote processor may store.
- **bsp_put** deposits local data into registered remote memory on a target processor.
- **bsp_get** copies data from registered local memory into ordinary local memory.
- **bsp_pop_reg** *unregisters* the area.
- Note: push/pop do ***not*** imply a stack-like discipline is required.

bsp_push_reg

(http://wwwcs.uni-paderborn.de/~bsp/docu/pub8/pub.html)

void bsp_push_reg(t_bsp* bsp, void* ident, int size)

parameter	type	description
bsp	t_bsp*	pointer to the BSP object
ident	void*	local address of the variable, also identifier for bsp_get and bsp_put
size	int	size of variable, can be different on different nodes

This function registers a variable for global access. All nodes must call bsp_push_reg with `ident` pointing to a local variable. The size of the variabale (`size` bytes) can be different on each node.

Then every node can access the variables on other nodes with bsp_get (bsp_hpget) and bsp_put (bsp_hpput) using the local address as an identifier.

Attention:

If different variables have to be registered all processors have to call the bsp_push_reg functions in the same order.

bsp_put, bsp_hpput

void bsp_put (t_bsp* bsp, int destPID, void* src, void* dest, int offset, int nbytes)
void bsp_hpput (t_bsp* bsp, int destPID, void* src, void* dest, int offset, int nbytes)

parameter	type	description
bsp	t_bsp*	pointer to the BSP object
destPID	int	id of destination node
src	void*	pointer to source memory
dest	void*	pointer to destination variable
offset	int	offset in destination variable
nbytes	int	number of bytes to copy

Does not appear in earlier code.

Must have been registered using push.

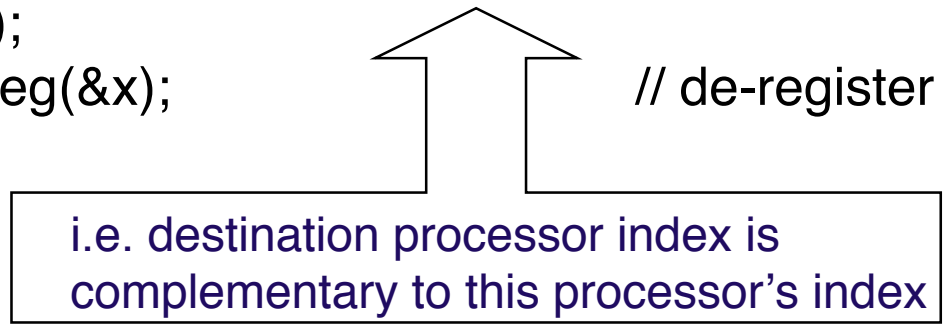
These functions copy `nbytes` bytes from local memory `src` to variable `dest` (with offset `offset`) on node `destPID`.

bsp_put: The destination is written on in the synchronization after all bsp_get operations are finished.

bsp_hpput: This is the high performance version of bsp_put. The copy is done asynchronously and unbuffered, so it is finished after the next call to bsp_sync bsp_oblsync), the buffer src must not be changed before. Do not use this function if destination-buffer is used in this super-step because the time the destination is written is undefined.

Example: Reverse int values over array of processes

```
int reverse(int x)
{
bsp_push_reg(&x, sizeof(int));                    // registers a 1-int block
bsp_sync();
bsp_put(bsp_nprocs() - bsp_pid() - 1, &x, &x, 0, sizeof(int));
bsp_sync();
bsp_pop_reg(&x);                                  // de-register the block
return x;
}
```



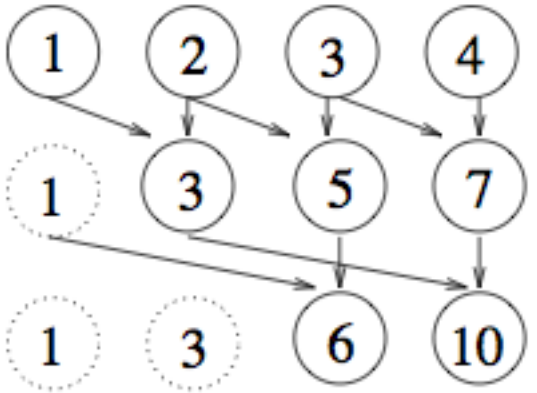
Prefix Sums Example
(log n supersteps)

x_i is assumed to be stored in processor i

```
int bsp_allsums1(int x) {
  int i, left, right;
  bsp_pushregister(&left,sizeof(int));
  bsp_sync();

  right = x;
  for(i=1;i<bsp_nprocs();i*=2) {
    if (bsp_pid()+i < bsp_nprocs())
      bsp_put(bsp_pid()+i,&right,&left,0,sizeof(int));
    bsp_sync();
    if (bsp_pid()>=i) right = left + right;
  }
  bsp_popregister(&left);
  return right;
}
```

register **left**'s address so it can be used as a destination (by another processor) below

transmit my **right** to his **left**.



Buffering Options

- **Buffered on source**: Read data from remote process at the end of a superstep, *before any remote writes*.
- **Buffered on destination**: Write at *end* of superstep, after all remote reads.
- **Unbuffered on destination**: Write at *any time* during superstep. [prefixed with **hp** for “high performance”]
- **Unbuffered on source**: Read at any time during superstep. [prefixed with **hp** for “high performance”]

2-Stage Broadcast Example (using DRMA)

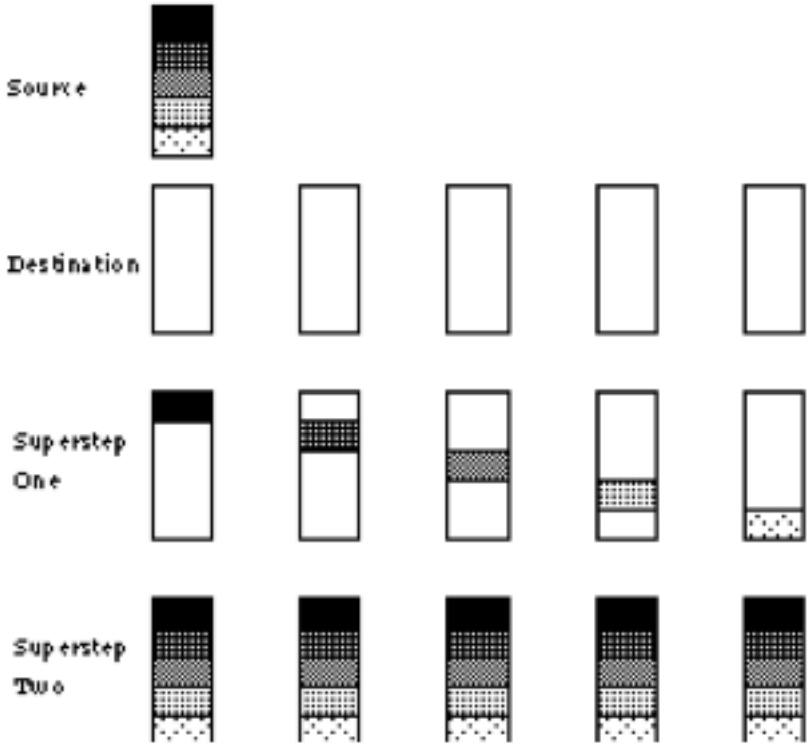


Figure 2: Two stage broadcast using total exchange.

2-Stage Broadcast Code

```
void bcast_twostage(int frompid, void *src, void *dst, int nbytes) {
  int i,n_over_p, last_n_over_p;

  n_over_p      = nbytes / nprocs;                    // nominal chunk size
  last_n_over_p = nbytes - (n_over_p * (nprocs-1)); // last chunk size

  if (frompid == pid) {
    for(i=0;i < nprocs; i++) {
      bsp_hpput(i,
                ((char*) src)+(n_over_p * i),
                dst,
                (n_over_p * i),
                (i==(nprocs-1))? last_n_over_p : n_over_p);
    }
  }
  bsp_sync();

  for(i=0; i < nprocs; i++)
    bsp_hpput(i,
              ((char*) dst)+(n_over_p*pid),
              dst,
              (n_over_p*pid),
              (pid==(nprocs-1)) ? last_n_over_p : n_over_p);
  bsp_sync();       }
```

source processor distributes its data in chunks

each processor distributes its chunk to others

Contrast Naïve (1-Stage) Broadcast

```
void bcast_onestage(int frompid, void *src, void *dst, int nbytes) {
  int i;
  if (pid==frompid)
    for(i=0;i&ltnprocs;i++)
      bsp_hpput(i,src,dst,0,nbytes);
  bsp_sync();
}
```

The cost increases linearly with p, in contrast with the 2-stage algorithm. (n is the size of the data structure)

cost of 1-stage broadcast = npg + L

proportional to p

cost of 2-stage broadcast = ((n/p)pg + L) + ((n/p)pg + L) = 2ng + 2L

independent of p

DRMA 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,p=bsp_nprocs();
  for(j=0;j<nelem;j++) result += xs[j];
  bsp_push_reg(&result,sizeof(int));
  bsp_sync();

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

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

called by each processor

compute local sums

register local results

get remote sums

add remote sums locally

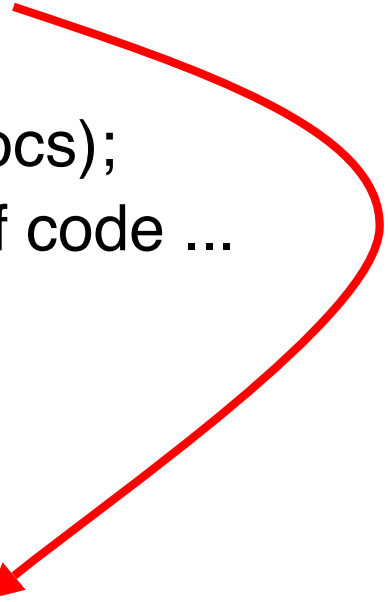
de-register local results

Second BSP Programming Abstraction: Pure SPMD

```
bsp_begin(nprocs);

… SPMD part of code …

bsp_end();
```

BSP C calls: Sequential followed by pure SPMD

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

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



Third Programming Abstraction: BSMP Bulk-Synchronous Message Passing

Direct Remote Memory Access is **less convenient** for computations where the **volumes of data being communicated in supersteps are irregular and data dependent**, and where the computation to be performed in a superstep depends on the quantity and form of data received at the start of that superstep. A more appropriate style of programming in such cases is bulk synchronous message passing (**BSMP**).

In BSMP, a **non-blocking send operation is provided that delivers messages to a system buffer associated with the *destination* process**. The message is guaranteed to be in the destination buffer at the **beginning of the subsequent superstep**, and can be accessed by the destination process only during that superstep. **If the message is not accessed during that superstep, it is removed from the buffer**.

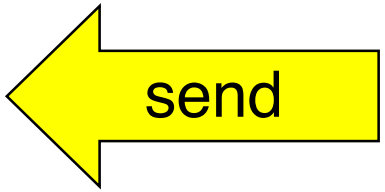
In keeping with BSP superstep semantics, the messages sent to a process during a superstep have no implied ordering at the receiving end; a **destination buffer may therefore be viewed as a queue, where the incoming messages are enqueued in arbitrary order and are dequeued (accessed) in that same order**. Note that although messages are typically identified with tags, BSPlib provides **no tag-matching facility for the out-of-order access** of specific incoming messages.

Message-Passing Primitives

bsp_send	send a memory block
bsp_nmsgs	get number of received messages
bsp_getmsg	get pointer to received message
bsp_findmsg	find received message according to sender
bspmsg_data	get pointer to data of a message
bspmsg_size	get size of a message
bspmsg_src	get id of the sender-processor

BSMP 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,p=bsp_nprocs(),
      nonzeros,nonzeros_size,status, *sparse_ivec;
  float *sparse;

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

  for(i=0;i<n_over_p;i++)
    if (dense[i]!=0.0) {
      global_idx = (n_over_p * bsp_pid())+i;
      for(j=0;j<p;j++)
        bsp_send(j,&global_idx,&dense[i],sizeof(float));
    }
  bsp_sync();

  bsp_qsize(&nonzeros,&nonzeros_size);
  if (nonzeros>0) {
    sparse      = calloc(nonzeros,sizeof(float));
    sparse_ivec = calloc(nonzeros,sizeof(int));
    if (sparse==NULL || sparse_ivec==NULL)
      bsp_abort("Unable to allocate memory");
    for(i=0;i<nonzeros;i++) {
      bsp_get_tag(&status,&sparse_ivec[i]);
      if (status!=sizeof(float))
         bsp_abort("Should never get here");
      bsp_move(&sparse[i],sizeof(float));
    }
  }
  bsp_set_tagsize(&tag_size);
  *sparse_out      = sparse;
  *sparse_ivec_out = sparse_ivec;
  return nonzeros;
}
```



Sample Results: FFT Speedup

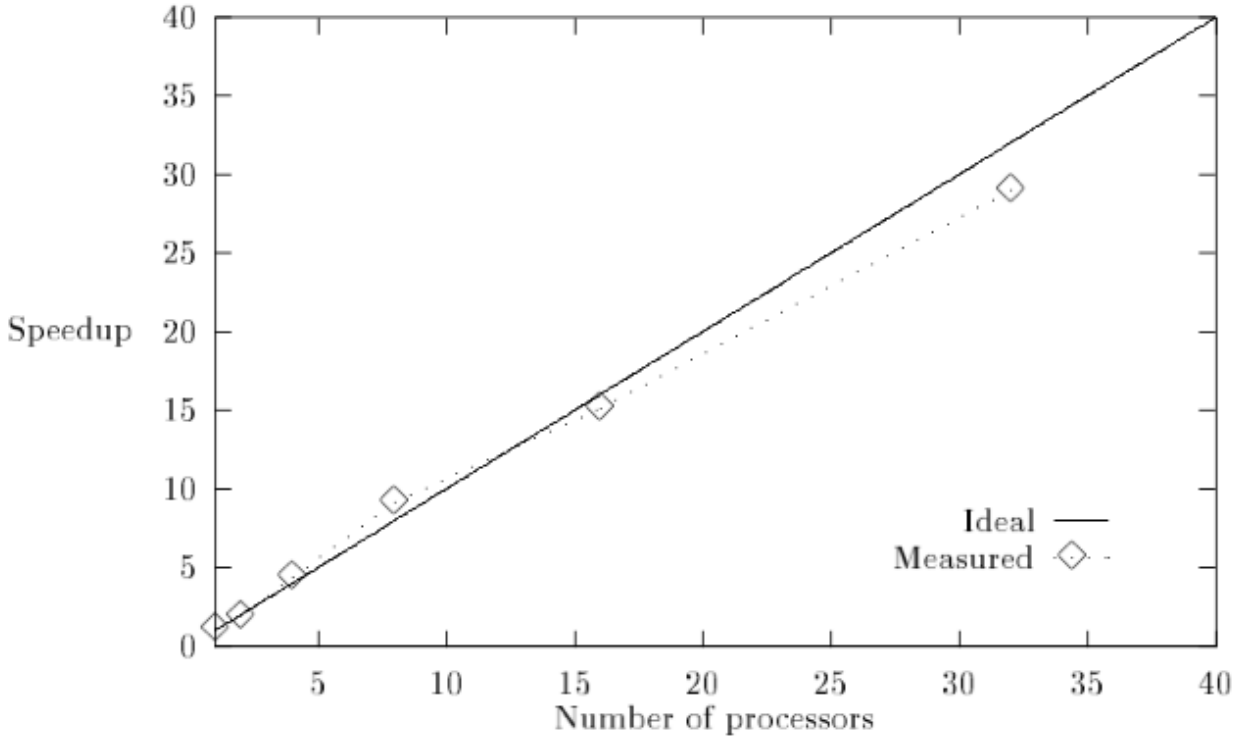


Fig. 1. Absolute speedup of parallel FFT on a Cray T3E with 32 processors. The problem size is $n = 16234$. Speedups are relative to the sequential execution time of 0.108 s of a radix-2 FFT program.

Results: Barnes-Hut Speedup

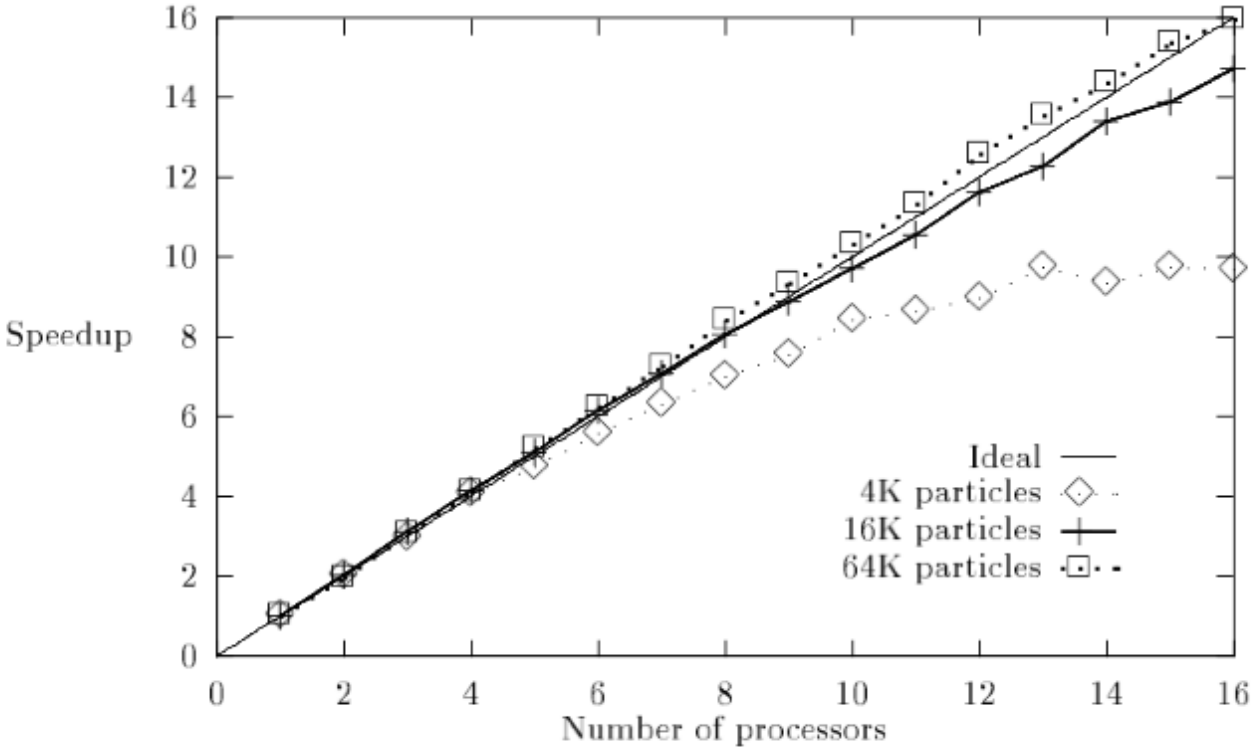


Fig. 3. Relative speedup for a molecular dynamics computation on an SGI Challenge with 16 processors. Speedups are relative to the parallel program with $p = 1$, which takes 13.64 s per iteration for 4000 particles, 102.32 s for 16000, and 474.7 s for 64000.

Aggregate Operators

name	description
bsp_reduce	determine the sum over an interval of processors
bsp_scan	determine the prefix sum over an interval of processors
bsp_allreduce	determine the sum over an interval of processors and broadcast this value
bsp_allscan	determine prefix sum and broadcast the sum to the interval processors
bsp_lreduce	determine the sum over arbitrary processors
bsp_lscan	determine the prefix sum over arbitrary processors
bsp_lallreduce	determine the sum over arbitrary processors and broadcast this value
bsp_lallscan	determine prefix sum and broadcast the sum to arbitrary processors

Collection Operations used for database apps

- map
- filter
- fold (reduce)
- set difference
- set union
- Cartesian product

Cartesian Product for Example

Given two collections S of type $collection\langle\tau\rangle$ and T of type $collection\langle\sigma\rangle$, then the cartesian product $S \times T$ calculates the collection of all pairs from S and T of type $collection\langle\tau \times \sigma\rangle$. The BSP implementation works by transposing the data blocks of the smaller collection, say T, in successive supersteps, and then computing the local cartesian product. Clearly, this will bring every element s_i of S with every element t_i of T into the same processor, to form the pair (s_i, t_i) within $p-1$ supersteps, as shown in figure 4.3.

similar to all-to-all distribution example

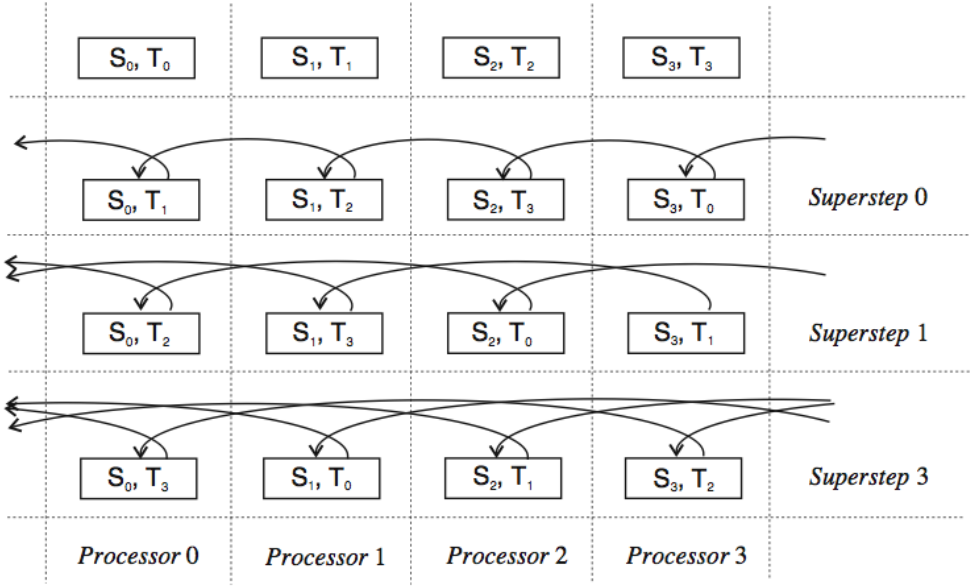


Figure 4.3: Cartesian product example with 4-processors

Example: LLCS Problem (Length of Longest Common Subsequence)

Peter Krusche and Alexander Tiskin, Univ. of Warwick, 2006

Let $X = x_1x_2 \ldots x_m$ and $Y = y_1y_2 \ldots y_n$ be two strings on a finite alphabet

- **Subsequence** U of string X: U can be obtained by deleting zero or more elements from X

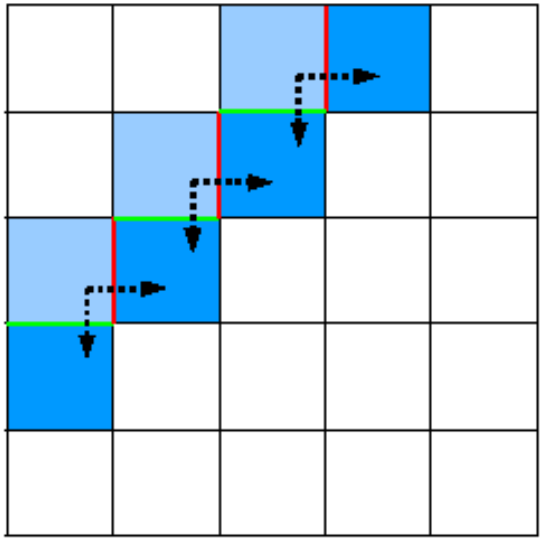
  i.e. $U = x_{i1}x_{i2} \ldots x_{ik}$ and $i_q < i_{q+1}$ for all q with $1 \leq q < k$.

- Strings X and Y : $LCS\ (X, Y)$ is any string which is subsequence of **both** X and Y and has **maximum** possible length.

- Length of these sequences: LLCS (X, Y).

Parallel LLCS Algorithm

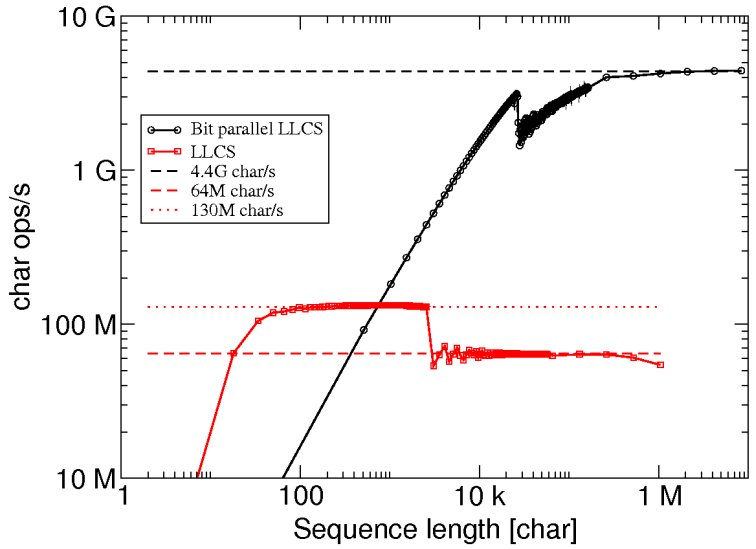
- Based on a simple parallel algorithm for grid DAG computation
- **Dynamic programming matrix** L is partitioned into a grid of rectangular blocks of size *(m/G)×(n/G)* (*G* : grid size)
- Blocks in a **wavefront** can be processed in parallel
- Assumptions:
  - Strings of equal length *m = n*
  - Ratio α = G/p is an integer



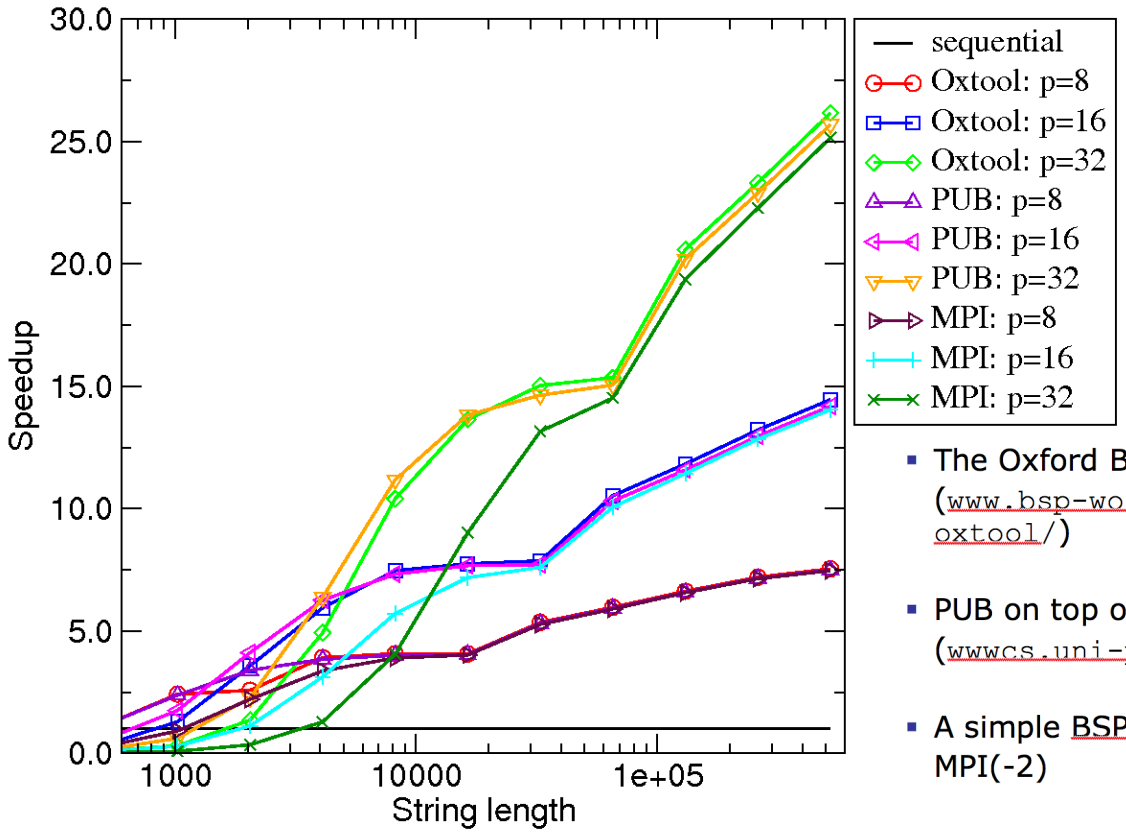
A Bit-Parallel Algorithm for LLCS

- Bit-parallel computation processes ω entries of L in parallel (ω : machine word size)

- This leads to substantial speedup for the sequential computation phase and slightly lower communication cost per superstep.



Speedup Results for LLCS



- The Oxford BSP Toolset on top of MPI (www.bsp-worldwide.org/implmnts/oxtool/)
- PUB on top of MPI (except on the SGI) (wwwcs.uni-paderborn.de/~bsp/)
- A simple BSPlib implementation based on MPI(-2)

Many Scattered References

- Worldwide: http://www.bsp-worldwide.org/
- http://www.bsp-worldwide.org/bspww3000.html
- Oxford: http://www.bsp-worldwide.org/implmnts/oxtool/
- Paderborn: http://wwwcs.uni-paderborn.de/~bsp/
- Carleton: http://proton.scs.carleton.ca/~bsp/
- BSP on MPI: http://bsponmpi.sourceforge.net/
- Harvard: http://people.seas.harvard.edu/~valiant/