The BSP model, tools, algorithms and implementation

Jonathan Hill
Oxford University Computing Laboratory

Overview

1. The BSP model
2. *BSPlib*
   - Put vs. get one-sided communication
   - buffered vs. unbuffered communication
   - registration
3. BSP algorithm design
   - BSP cost analysis
   - parallel algorithm design
   - profiling tools

The Bulk Synchronous Parallel model

BSP is a *model* of parallel computation. It describes, in general terms, how a parallel computation should proceed.

BSP does not prescribe the way in which:

- local computations are carried out
- communication actions are expressed.

The BSP model (cont.)

The essence of the BSP model is the notion of a *superstep*. A superstep is decomposed into three phases:

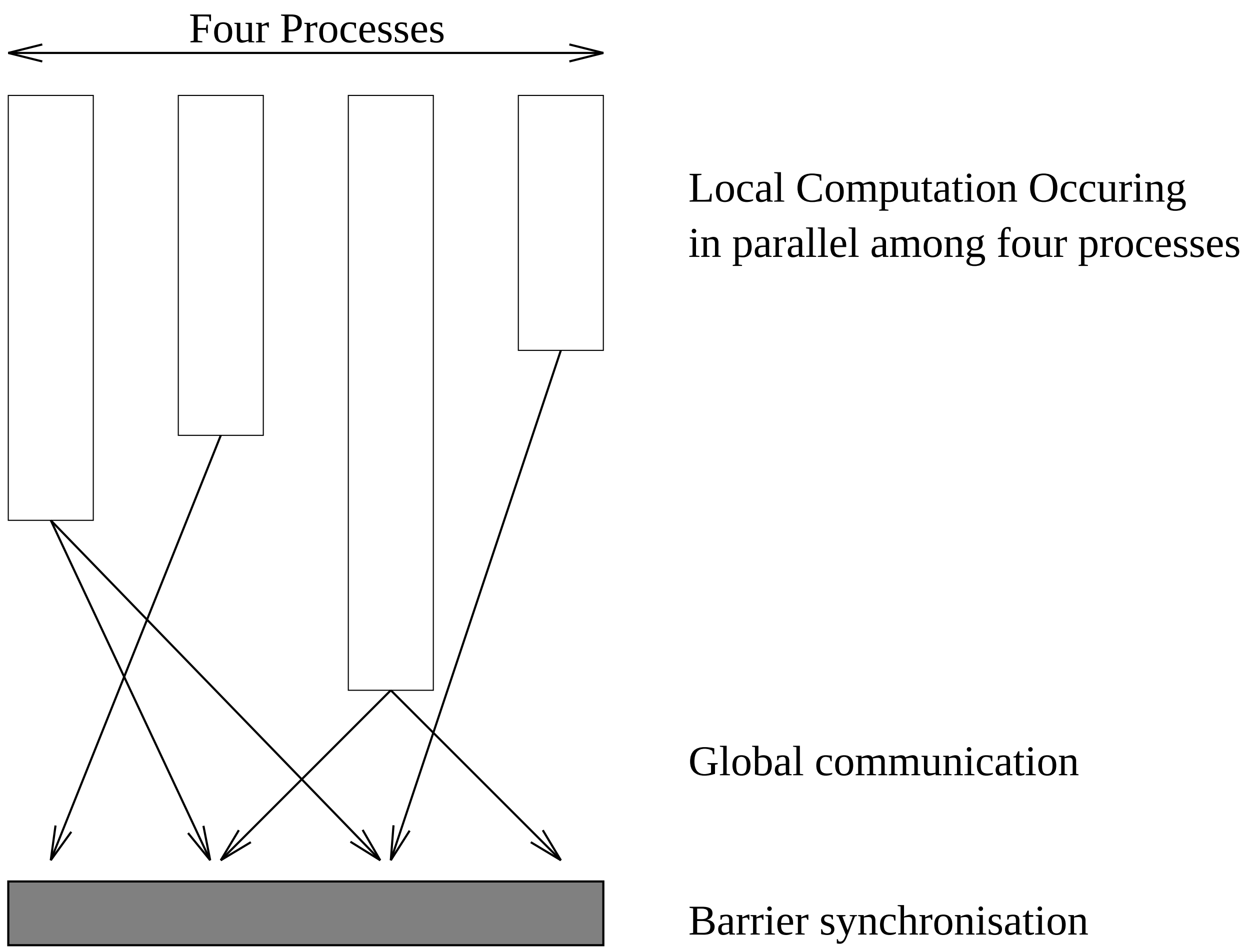
Local computation within each process on local data-values.

Non-blocking communication can be initiated with other processors.

A barrier synchronisation of *all processes* is used to delimit the end of the superstep where all communications are guaranteed to have occurred by.

Communication and synchronisation are completely decoupled

The BSP model (cont.)



***BSPlib*: what is it?**

A standard library for BSP programming developed by:

Jonathan M. D. Hill[1]
Bill McColl[1]
Dan C. Stefanescu[2,3]
Mark W. Goudreau[4]
Kevin Lang[5]
Satish B. Rao[5]
Torsten Suel[6]
Thanasis Tsantilas[7]
Rob Bisseling[8]

Authors' affiliations:
(1) Oxford University
(2) Harvard University
(3) Suffolk University
(4) University of Central Florida
(5) NEC Research Institute, Princeton
(6) Bell Laboratories, Lucent Technologies
(7) Columbia University
(8) Utrecht University

see ***`http://www.bsp-worldwide.org`*** *for more details*

BSPlib: communication styles

Two styles of communication available in the library:

- **Direct Remote Memory Access (DRMA)** in which processes can copy data from or into a remote processes memory, without the active participation of the remote process.
- **Bulk Synchronous Message Passing (BSMP)** in which processes send a packet to a queue in a remote process, which can be accessed by the remote process in the following superstep.

A small library of 20 operations (MPI-1: 129 operations; MPI-2: 202 operations, PVM: 90 operations):

Class	**Operation**	**Meaning**
Initialisation	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_pid	Find my process id
	bsp_nprocs	Number of processes
	bsp_time	Local time
Superstep	bsp_sync	Barrier synchronisation
DRMA	bsp_push_reg	Make region globally visible
	bsp_pop_reg	Remove global visibility
	bsp_put	Copy to remote memory
	bsp_get	Copy from remote memory
BSMP	bsp_set_tag_size	Choose tag size
	bsp_send	Send to remote queue
	bsp_qsize	Number of messages in queue
	bsp_get_tag	Match tag with message
	bsp_move	Move from queue
High Performance	bsp_hpput	Unbuffered versions
	bsp_hpget	of communication
	bsp_hpmove	primitives

BSPlib: Direct Remote Memory Access (DRMA)

- One-sided communication—a process can copy a block of data from its own memory, into the memory of another process, *without the active participation of the remote process*.
- Two different styles:

get that *reads* or fetches a copy of a data-structure from a remote process into the process initiating the get.

put offloads or *writes* data from the initiating process into a remote process.

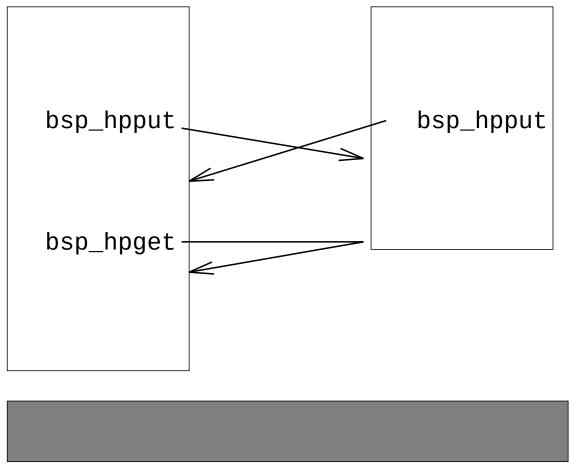
***BSPlib*: Direct Remote Memory Access (DRMA) (cont..)**

The semantics (i.e., meaning) of the DRMA operations depend upon the interpretation of a superstep. i.e., do the communications:

1. **buffered**: take effect *at* the end of the superstep.
2. **high performance unbuffered**: take effect *any time up to* the end of the superstep.

These two different styles of DRMA are provided in *BSPlib*.

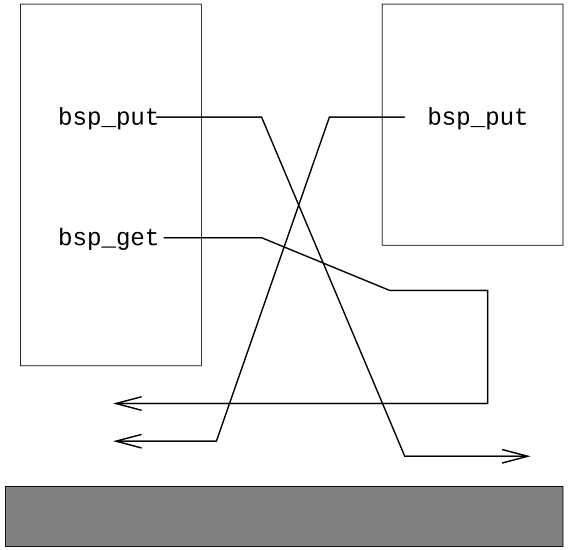
***BSPlib*: high-performance DRMA**



Communication and computation can be overlapped. However,

- data-structures communicated can not be changed locally until the end of the superstep.
- hard to reason about programs due to the non-determinism of:
  - DRMA writes changing local variables
  - interaction of puts and gets

BSPlib: standard (aka buffered) DRMA



The phases of local computation, communication, and synchronisation do not overlap. Therefore,

- communicated data-structures may be changed locally after this kind of DRMA operation
- the only non-determinism is due to multiple processes writing into the same location(s).

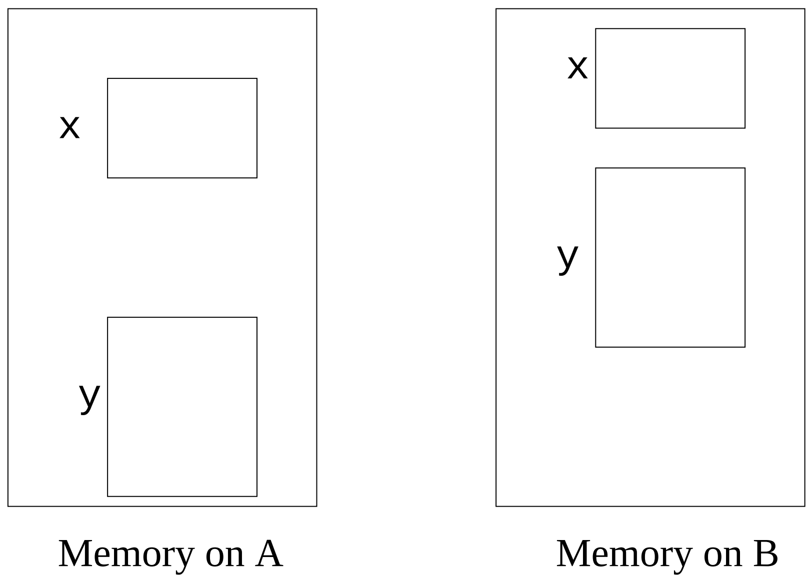
***BSPlib*: DRMA registration, the problem**

When using DRMA, if process A executes:

$$\texttt{bsp\_put}(B, x, y, 0, \texttt{sizeof}(x));$$

then the data-structure x on process A is moved into the data-structure y on process B, where x and y are the *names* of arrays in the user program.

- As *BSPlib* is a SPMD library, then x and y will probably exist on all the processes.
- At runtime, on a particular process A, x and y reference the data-structures denoted by x and y on process A.
- All the different values of x, in all the processes will probably be different.

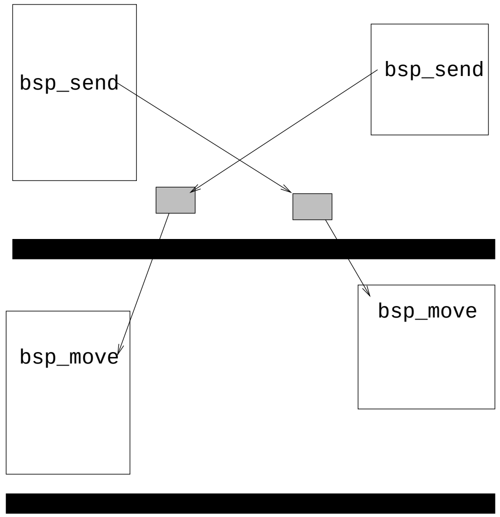


BSPlib: registration

Registration is the process by which the user informs *BSPlib* the names of the *same* data-structure on each process. It enables:

- Communication to and from heap allocated data
- Communication to and from stack allocated data
- *BSPlib* to work in heterogeneous environments
- bounds checking of communications

BSPlib: Bulk Synchronous Message Passing

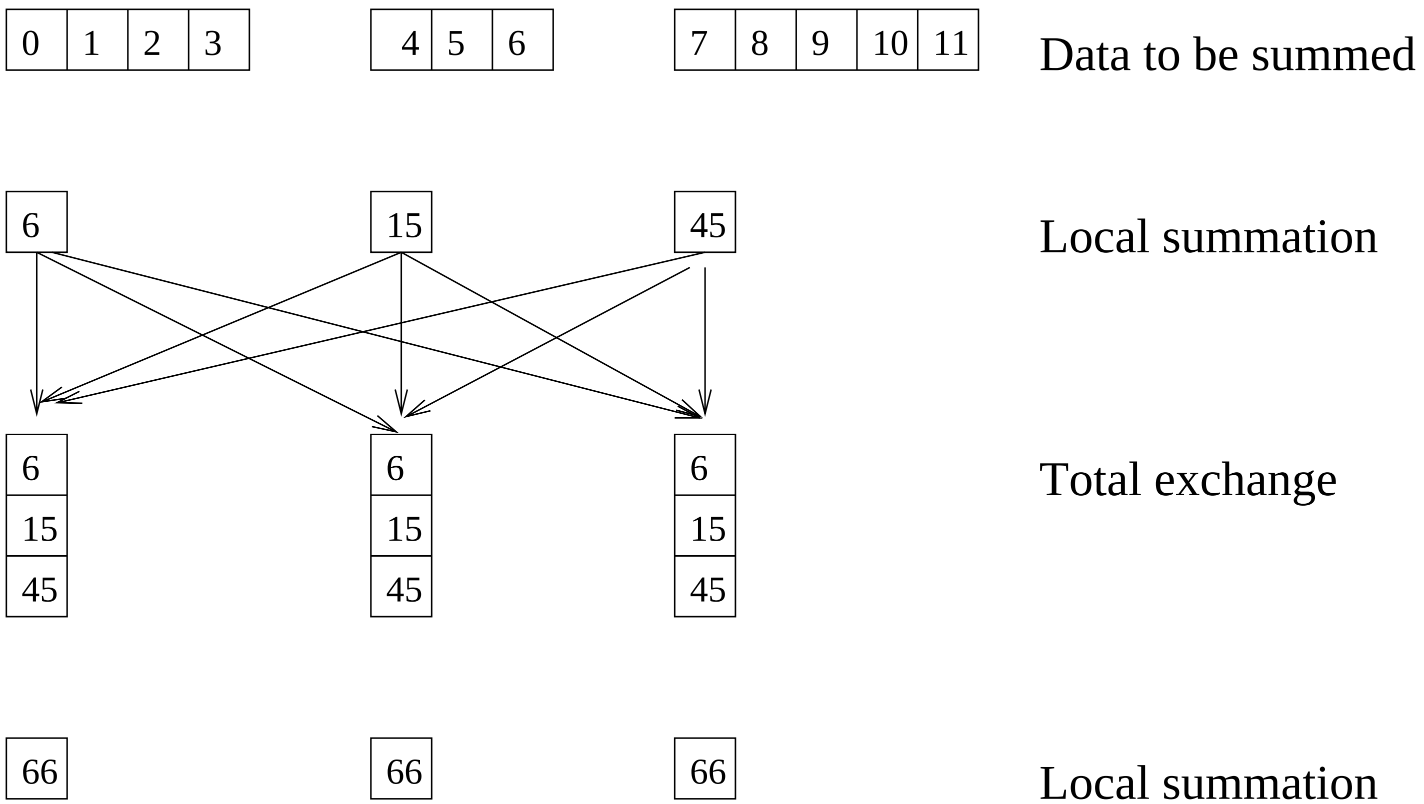


- Standard non-blocking send and receive operations.
- all messages have a tag and payload field. The tag is of fixed size for all messages in a superstep. The payload can be arbitrarily sized.
- Synchronisation is guaranteed by the *barrier at the end of the superstep*.

BSPlib: example

Summing n values distributed among p processors (where $n > p$).

e.g.,



```
int bsp_sum(int *xs, int nelem) {
  int *local_sums,i,j,result=0;
  for(j=0;j<nelem;j++) result = 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 = result+ local_sums[i];
  bsp_pop_reg(&result); free(local_sums);
  return result;
}
```

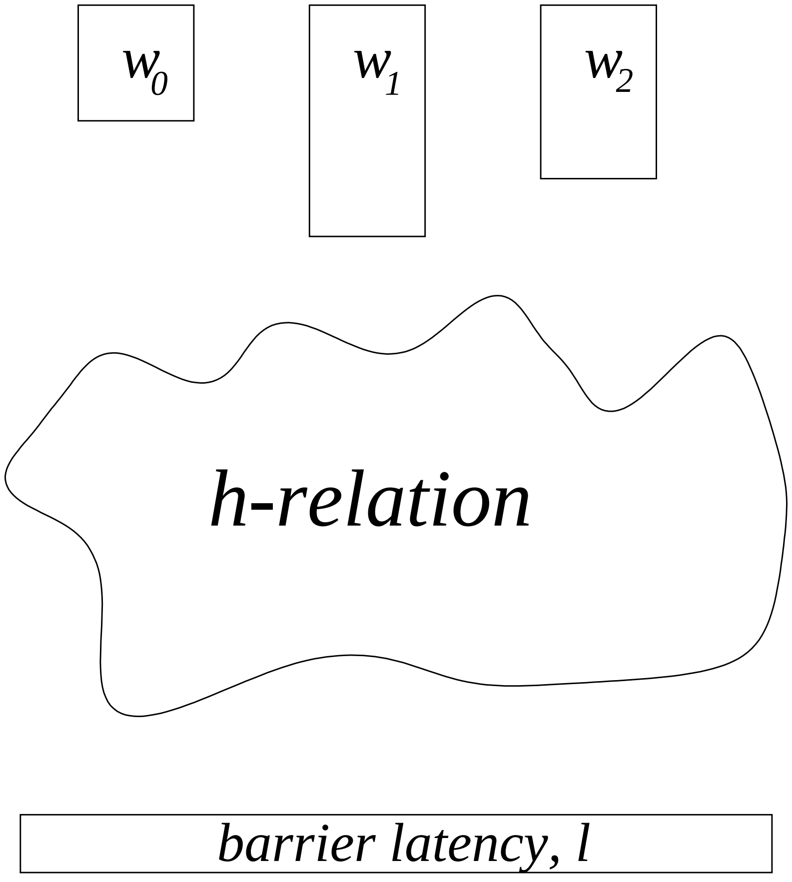
BSP design

BSP is not just another communications library. The simple structure of BSP programs facilitates:

- performance prediction
- parallel algorithm design

The BSP cost model is based around the observation that the dominating cost in communicating a message is the cost of crossing the boundary of the network.

The BSP cost model: costing a single superstep



The BSP cost model: costing a single superstep

$$\textit{superstep cost} = \underbrace{\mathsf{MAX}\ w_i}_{\text{Computation}} + \underbrace{h \times g + l}_{\text{Communication}}$$

$$\begin{aligned} \text{where } h &= \mathsf{MAX}\ h_i \\ h_i &= \text{largest amount of data} \\ &\quad\ \text{entering or leaving process } i \end{aligned}$$

The BSP cost model: costing multiple supersteps

The cost of a series of supersteps is just the sum of the costs of each separate superstep.

The performance of BSP algorithms is captured by formula with the structure:

$$
\begin{aligned}
\textit{execution cost (in flops)} &= \textit{comp.} + \textit{comm.} + \textit{synch.} \\
&= \alpha + \beta g + \gamma l
\end{aligned}
$$

The BSP cost model: machine parameters

α, β and γ are *application-dependent costs*.

s, l, and g are *architecture-dependent constants*:

- l is the synchronisation latency cost in units of s.
- g is the number of flops/word required for all processors to simultaneously communicate a message.
- s is the speed of computation of a process in flops/s.

Machine	s	p	l flops	l/s μs	g flops/word	g/s μs/word
Origin 2000	101	2	804	7.9	8.26	0.08
		4	1789	17.9	10.24	0.10
		8	3914	39.8	15.10	0.15
		16	15961	158.5	44.9	0.45
PowerChallenge	74	2	1132	15.3	10.2	0.14
		4	1902	25.7	9.3	0.13
Pentium II NOW	61	2	52745	870.4	484.5	8.00
10Mbit Enet		4	139981	2309.9	1128.5	18.62
		8	826054	13631.3	2436.3	32.91
Cray T3E	46.7	2	269	5.7	2.14	0.05
		4	357	7.6	1.77	0.04
		8	506	10.8	1.64	0.03
		16	751	16.0	1.66	0.04
Sp2 switch	26	1	244	9.4	1.3	0.05
		8	5412	208.2	11.4	0.43
Cray T3D	12	2		5.6	0.3	0.02
		8	175	14.4	0.8	0.07
		64	148	12.3	1.7	0.14
		256	387	32.1	2.4	0.20

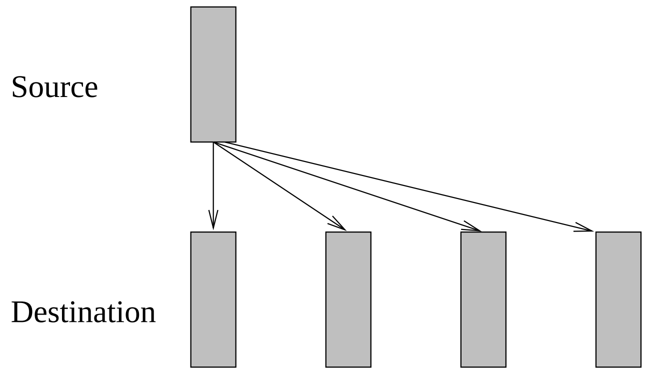
A BSP design strategy

The BSP cost model makes it clear that the following strategies should be used to write efficient BSP programs:

- balance the computation in each superstep between processes, since the computation time for a superstep is a *maximum* over the computation times in each process, and the barrier synchronisation must wait for the slowest process;
- balance the communication between processes, since h is a *maximum* over fan-in and fan-out of data; and
- minimise the number of supersteps, since this determines the number of times l appears in the final cost.

Example algorithm: one-phase broadcast

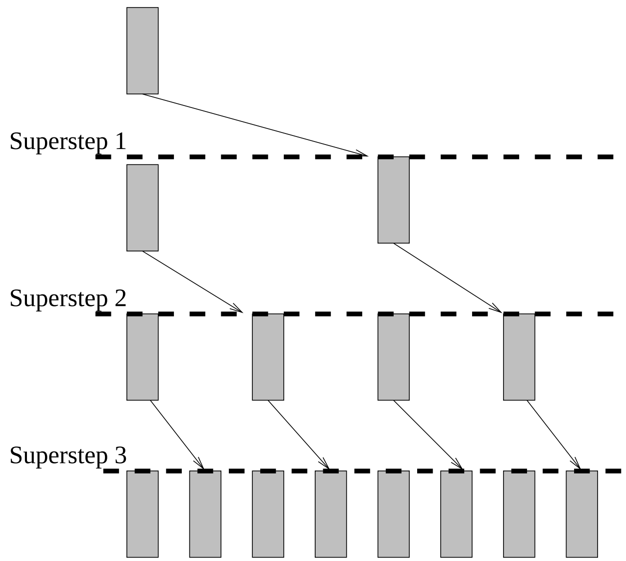
The broadcasting process transmits p messages of size N to all other processes.



$$Naive\ broadcast\ cost = l + pNg \tag{1}$$

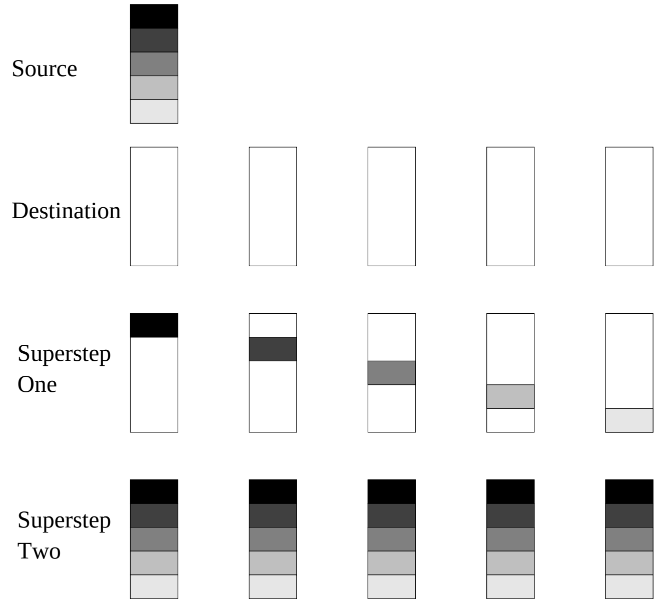
Example algorithm: tree broadcast

A logarithmic number of rounds, in which each process transmits 1 message of size N to another process.



$$Tree\ \ broadcast\ \ cost = (\log p)l + (\log p)Ng \tag{2}$$

Example algorithm: two-phase broadcast



$$\begin{aligned} Two\ stage\ broadcast &= \left(l + \frac{N}{p}pg\right) + \left(l + \frac{N}{p}pg\right) && (3)\\ &= 2l + 2Ng && (4) \end{aligned}$$

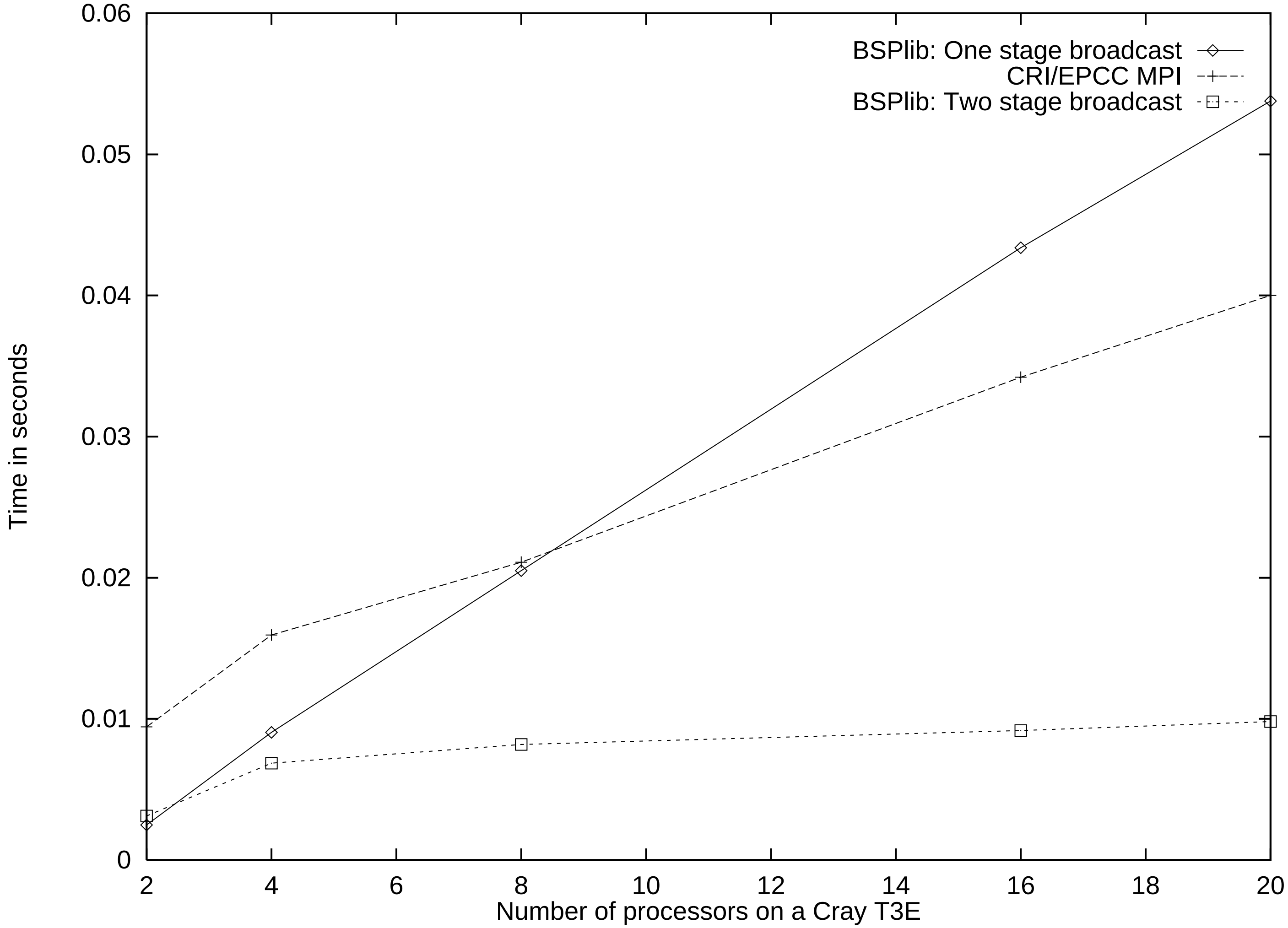
Example algorithm: summary on broadcasting

Algorithm	Communication	Sync
One-stage	pNg	l
Tree	$(\log p)Ng$	$(\log p)l$
Two-stage	$2Ng$	$2l$

- When N is small (i.e., $N < \frac{l}{pg-2g}$) use the one-stage broadcast.
- The two-stage broadcast is *always* better than the tree broadcast when $P > 2$ (it is the same when $P = 2$).
- Most communication libraries still implement broadcast by the tree based technique.

$$\boxed{\text{Good BSP design} \equiv \text{Balanced design}}$$

Broadcasting 100,000 doubles on a 20 processor Cray T3E



Using a profiling tool to aid good parallel design

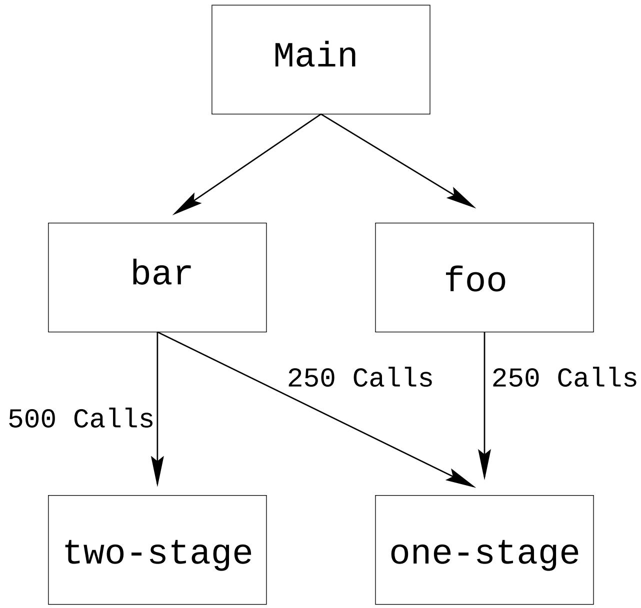
Aim: develop a profiling tool that identifies those imbalances in programs *that are amenable to improvement.*

i.e., can a profiling tool aid the optimisation of the $\mathcal{O}(p)$ one-stage broadcast into the $\mathcal{O}(1)$ two-stage broadcast?

In general, this enables the asymptotic complexity of a program to be improved, which therefore provides the basis for architecture independent optimisation.

A call-graph profiling tool: the broadcast algorithms

Call-graph profiling of the two broadcast algorithms:



- The leaves in the graph are textual positions of supersteps.
- Interior nodes are procedures.

Call-graph profiling: a leaf node

A leaf node has the form:

bcast.c line 34
[500 syncs]

	Max	Avg	Min
Comp	3.38s	99%	99%
Comm	1.34s	11%	5%
Wait	1.29s	94%	0%
Hrel	9.4e+07	12%	7%

It shows that a `bsp_sync()` at line 34 of `bcast.c` was entered 500 times during program execution.

bcast_onestage
[500 syncs]

	Max	Avg	Min
Comp	19.48s	100%	100%
Comm	19.55s	7%	0%
Wait	19.49s	94%	0%
Hrel	1.5e+09	12%	7%

The sequence of costs $\langle C^1, \ldots, C^{500} \rangle$ were recorded for the procedure `bcast_onestage`. This data is summarised as:

$$maximum\ \ cost = \sum_k \max \{\ w_i^k \mid 0 \leq i < p\ \}$$

$$average\ \ cost = \sum_k \frac{1}{p} \left(\sum_{0 \leq i < p} w_i^k \right)$$

$$minimum\ \ cost = \sum_k \min \{\ w_i^k \mid 0 \leq i < p\ \}$$

Call-graph profiling: architecture independent optimisation

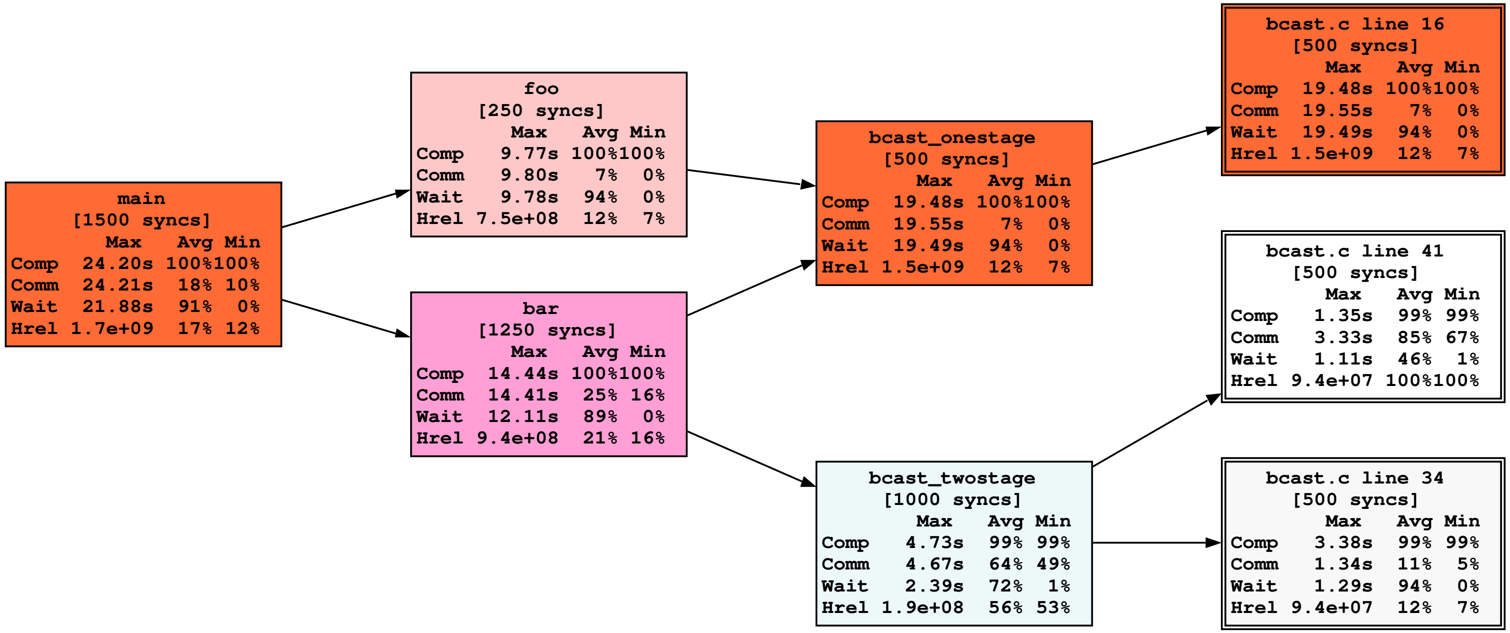
The difference between the maximum and average cost gives a measure of imbalance.

Imbalance in communication size (i.e., imbalance in h-relation size) is an architecture independent metric that can be used to optimise programs.

$$\boxed{\text{Good parallel design} \equiv \text{balanced design.}}$$

Call-graph profiling: broadcast

Results of broadcasting $100,000$ doubles (i.e., 316×316 matrix) on a 16 processor Cray T3E:



Conclusions

- The simplicity of the BSP cost model provides a parallel design methodology.
- *BSPlib* provides a *small*, simple to use communications library.
- Programming with barriers is simple and deadlock free.
- When the communication pattern is balanced amongst the processors, there are potential benefits in the asymptotic cost of an algorithm. i.e., a $\mathcal{O}(p)$ algorithm is transformed into a $\mathcal{O}(1)$ algorithm.

Availability

From `http://www.BSP-Worldwide.org/implmnts/oxtool/`

1. Silicon Graphics Power Challenge (Irix 6.x), Origin 2000.
2. IBM SP2
3. Cray T3E, T3D, (*C90 available soon*)
4. Parsytec Explorer
5. Convex SPP
6. Digital 8400 and Digital Alpha farm
7. Hitachi SR2001
8. Intel SMP running Windows NT

Plus generic versions on top of either TCP/IP, system V shared memory, or MPI.

Coming soon

- Within the next month:
  1. UDP/IP implementation
  2. Fault tolerance and process migration for NOWs
- Within the next four months:
  1. Intel NOW running Windows NT
  2. Process migration between SMPs and NOWs