Introduction to the Bulk Synchronous Parallel model

Albert-Jan N. Yzelman

3rd of October, 2014

Derived from the book slides by Prof. dr. Rob H. Bisseling, found at

`www.math.uu.nl/people/bisseling/Education/PA/pa.html`

Why?

A: some problems are too **large** to be solved on one machine

- Google pageranking,
- climate modelling,
- structural stability (skyscrapers, air planes, ...),
- financial market pricing,
- movie rendering,
- ...

But what is 'large'?

1. time to solution takes too long;
2. the problem does not fit on one machine.

Why?

B: sequential performance has **stalled**

Moore's Law: the number of transistors that one can cost-effectively place on a unit surface, doubles every 1.5 years.

Corollary: processor speeds double every 1.5 years

Why?

B: sequential performance has **stalled**

Moore's Law: the number of transistors that one can cost-effectively place on a unit surface, doubles every 1.5 years.

Corollary: processor speeds double every 1.5 years

This corollary **broke down** around 2007 due to power unscalability.

Solutions:

- Multi-core processors, optionally using
- slower clock speeds.

Software has to become parallel to keep up with Moore's Law!

Why not?

There are also reasons for not going parallel:

- writing parallel code is more **difficult**:
  - work distribution,
  - communication minimisation,
  - conflicts due to concurrency.
- Architecture **variability**: parallel code may run fast on one architecture, yet surprisingly slow on others.

To mitigate these issues, we should program

- **portably**, and
- **transparently**.

Efficient parallel programming

There are many different

- architectures (RISC, GPUs, x86, ...),
- network interconnects (Hypercube, ring, fat tree, ...),
- vendors (Intel, Cray, IBM, Oracle, Cisco, ...),
- programming interfaces.

We would like **one standard model** to design for, both for

- algorithm design, and
- hardware design.

How?

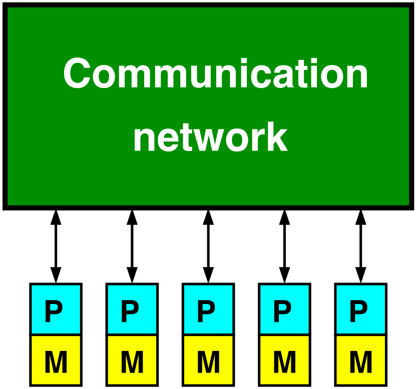
Solution: **bridging models**

- Message Passing Interface (MPI, interface only)
- **Bulk Synchronous Parallel** (BSP, model and interface)

Ref.: Leslie G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, Volume 33 (1990), pp. 103–111.

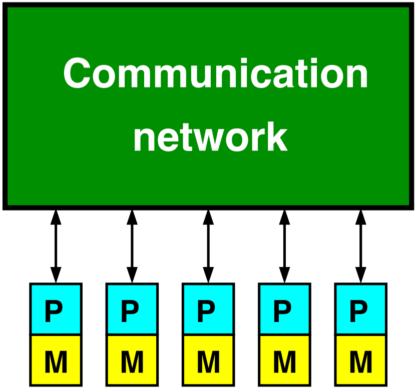
The Bulk Synchronous Parallel model

BSP computer: abstract model



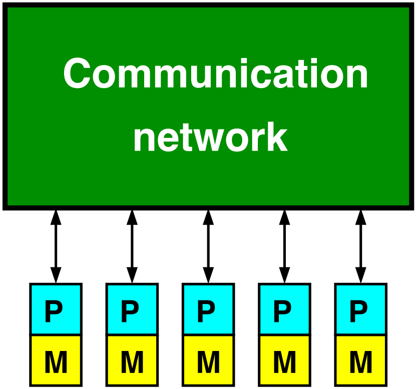
A Bulk Synchronous Parallel (BSP) computer.

BSP computer: abstract model



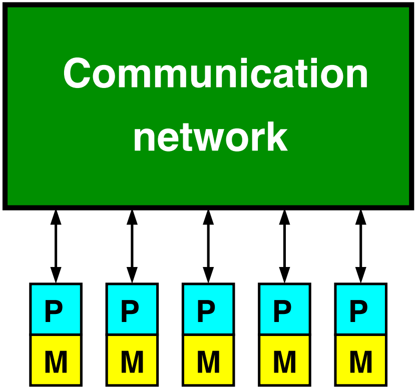
- A BSP computer consists of a collection of processors, each with its own memory. It is a distributed-memory computer.

BSP computer: abstract model



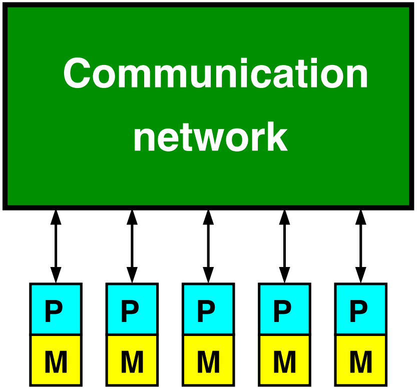
- Access to own memory is fast, access to remote memory is slow.
- Uniform access time to all remote memories.

BSP computer: abstract model



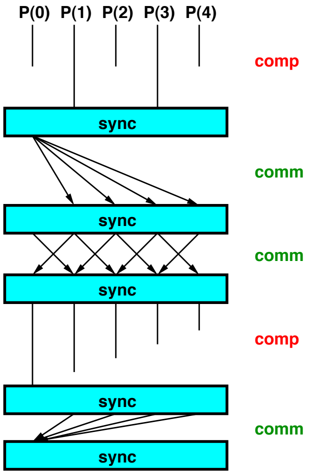
- A black-box communication network; algorithm designers should not worry about network details, only about global performance.

BSP computer: abstract model



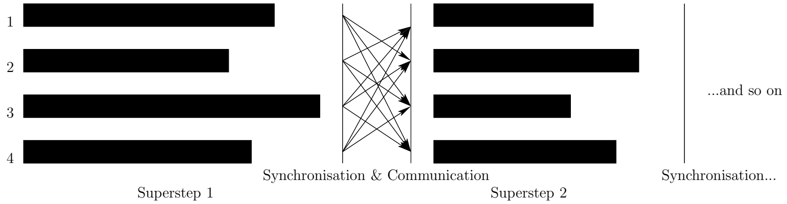
- Algorithms designed for a BSP computer are portable: they can be run efficiently on many different parallel computers.

BSP algorithm



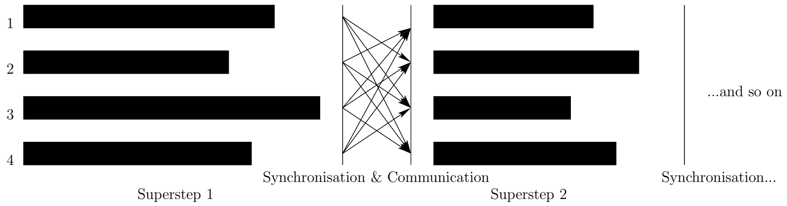
A BSP algorithm.

BSP algorithm



- A BSP algorithm consists of computation and communication **supersteps**.
- A superstep is always followed by a synchronisation **barrier**.
- A computation superstep is a **sequential** program.

BSP algorithm



- A communication superstep consists of communication operations that **transfer data words** from one processor to another.
- In theory, we distinguish between the two types of supersteps. This helps in the **design and analysis** of parallel algorithms.
- In practice, the distinction may be dropped, either automatically or manually; there may lead to **tradeoffs**.

The cost of a communication superstep

Suppose:

- h_{s} is the maximum number of data words sent by a processor.
- h_{r} is the maximum number of data words received by a processor.

The cost of a communication superstep

Suppose:

- h_s is the maximum number of data words sent by a processor.
- h_r is the maximum number of data words received by a processor.

An **h-relation** is a communication superstep in which every processor sends and receives at most h data words:

$$h = \max\{h_\mathrm{s}, h_\mathrm{r}\}.$$

The cost of a communication superstep

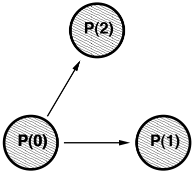
Suppose:

- h_s is the maximum number of data words sent by a processor.
- h_r is the maximum number of data words received by a processor.

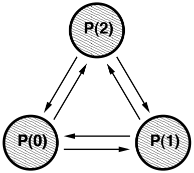
An **h-relation** is a communication superstep in which every processor sends and receives at most h data words:

$$h = \max\{h_\mathrm{s}, h_\mathrm{r}\}.$$

2-relations:



(a)



(b)

Cost of a communication superstep

The entry points and exit points of a communication superstep is the bottleneck of the time spent communicating.

The *h*-relation determines the cost of a communication superstep:

$$T_{\text{comm}} = hg,$$

where

- g is the time per data word, and

Cost of a communication superstep

The entry points and exit points of a communication superstep is the bottleneck of the time spent communicating.

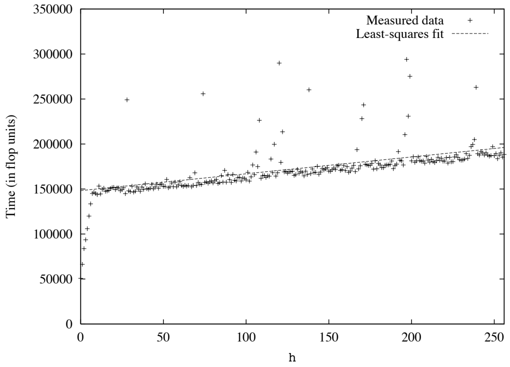
The *h*-relation determines the cost of a communication superstep:

$$T_{\text{comm}} = hg + l,$$

where

- g is the time per data word, and
- l is the start-up latency.

Time taken for *h*-relations on an 8-processor IBM SP2



$r = 212$ Mflop/s, $p = 8$, $g = 187$ flop $(0.88\mu s)$,
$l = 148212$ flop $(698\ \mu s)$

Cost of computation superstep

The cost of a computation superstep:

- for scientific computation, the cost is measured in floating point operations (**flops**).
- Let w be the **maximum number of flops** of a processor in the computation superstep, and
- let r be the speed of a single CPU (in flop per second, flop/s).

Then, the computation cost is given by

$$T_{\text{comp}} = w/r + l.$$

Note that

- processors with less than w work will be **idle**.
- This is the same l as in communication superstep, for simplicity.

BSP cost

The cost of computation followed by communication is given by the sum of the cost of its supersteps:

$$T_{\text{comm}} + T_{\text{comp}} = w + gh + 2l,$$

- to measure T, a **wall clock** that measures elapsed time is needed.
- Note that $g = g(p)$ and $l = l(p)$ are in general a function of the number of processors p.

BSP cost

The cost of computation followed by communication is given by the sum of the cost of its supersteps:

$$T_{\text{comm}} + T_{\text{comp}} = w + gh + 2l,$$

- to measure T, a **wall clock** that measures elapsed time is needed.
- Note that $g = g(p)$ and $l = l(p)$ are in general a function of the number of processors p.

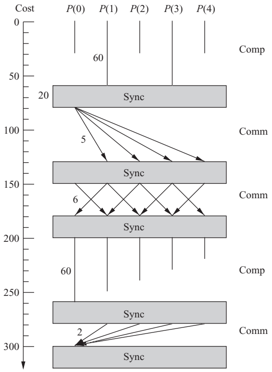
In general, the total BSP cost equals:

$$\sum_{i=0}^{N-1} \max w_i^{(s)}/r + h_i g + l,$$

with N the total number of supersteps of the BSP algorithm.

- The parameters $w_i^{(s)}$, h_i depend on p and on a problem size n.

BSP cost



For $p = 5$, $g = 2.5$, and $l = 20$:
First computation superstep costs $60 + 20 = 80$ flops.
First communication superstep costs $4 \cdot 5 \cdot 2.5 + 20 = 70$ flops.
$\Rightarrow$ the total cost of the BSP algorithm is 320 flops.

Summary

An abstract BSP computer is given by (p, r, g, l), with

p	number of processors
r	computing rate (in flop/s)
g	communication cost per data word (in flops)
l	global BSP latency cost (in flops)

The BSP model consists of

1. a distributed-memory architecture with a black box network that provides uniform access time to remote memories;
2. an algorithmic framework formed by a sequence of supersteps;
3. a cost model giving cost expressions of the form $a + bg + cl$.

Parallel inner-product computation

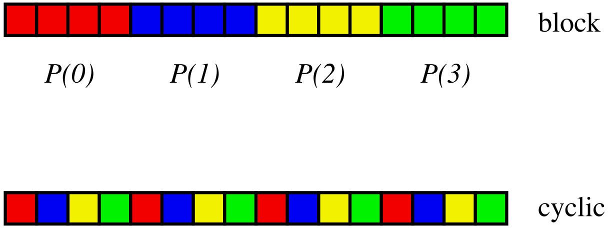
Inner product of two vectors

The inner product of two vectors $\mathbf{x} = (x_0, \ldots, x_{n-1})^{\mathrm{T}}$ and $\mathbf{y} = (y_0, \ldots, y_{n-1})^{\mathrm{T}}$ is defined by

$$\alpha = \mathbf{x}^{\mathrm{T}}\mathbf{y} = \sum_{i=0}^{n-1} x_i y_i .$$

Here, 'T' denotes transposition. All vectors are column vectors.

Vector data distributions



p = #processors = 4

n = vector length = 16

Block distribution

- The block distribution is defined by

$$x_i \longmapsto P(i \operatorname{div} b), \text{ for } 0 \leq i < n.$$

  Here, the div operator stands for dividing and rounding down:

$$i \operatorname{div} b = \lfloor i/b \rfloor.$$

- The block size is $b = \lceil \frac{n}{p} \rceil = \frac{n}{p}$, rounded up.
- For $n = 9$ and $p = 4$, this assigns 3, 3, 3, 0 vector components to the processors, respectively.
  You may blink at an empty processor, but this distribution is just as good as 3, 2, 2, 2. Really!

Cyclic distribution

The cyclic distribution is defined by

$$x_i \longmapsto P(i \bmod p), \text{ for } 0 \leq i < n.$$

This distribution is easiest to compute. Note the advantage of starting to count at zero: the formula becomes very simple.

Parallel inner product computation

Design pattern:

- Assign data so that the bulk of the computations are **local**.

Assign x_i and y_i to the same processor, for all i. This makes computing $x_i \cdot y_i$ a local operation. Thus $\mathrm{distr}(\mathbf{x}) = \mathrm{distr}(\mathbf{y})$.

Parallel inner product computation

Design pattern:

- Assign data so that the bulk of the computations are **local**.

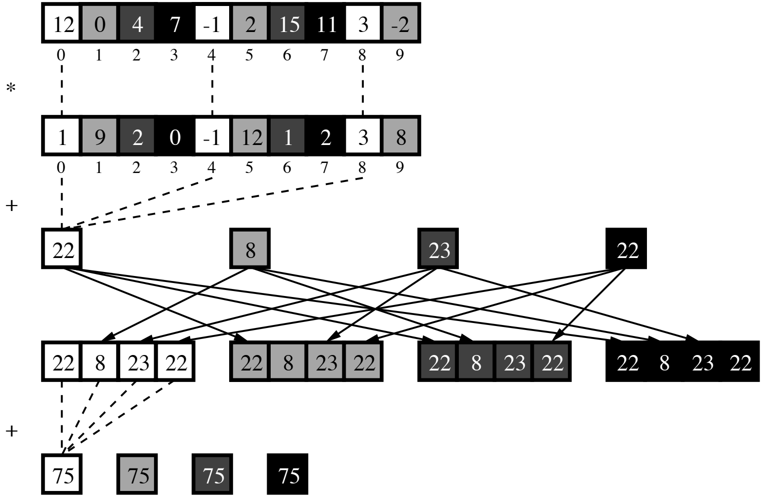
Assign x_i and y_i to the same processor, for all i. This makes computing $x_i \cdot y_i$ a local operation. Thus $\mathrm{distr}(\mathbf{x}) = \mathrm{distr}(\mathbf{y})$.

- Give each process the same amount of work (**load balance**)

Choose a distribution with an even spread of vector components. Both block and cyclic distributions are fine.
Here we chose cyclic, following the way card players deal their cards.

These constraints naturally lead to a parallel algorithm.

Example for $n = 10$ and $p = 4$



Parallel inner product algorithm for $P(s)$

input: $\mathbf{x}, \mathbf{y}$: vector of length n,
$\mathrm{distr}(\mathbf{x}) = \mathrm{distr}(\mathbf{y}) = \phi$,
with $\phi(i) = i \bmod p$, for $0 \leq i < n$.

output: $\alpha = \mathbf{x}^T\mathbf{y}$.

```
(0)   $\alpha_s := 0$;
      for $i := s$ to $n-1$ step $p$ do
          $\alpha_s := \alpha_s + x_i y_i$;
```

Parallel inner product algorithm for $P(s)$

input:	$\mathbf{x}, \mathbf{y}$: vector of length n,
	$\mathrm{distr}(\mathbf{x}) = \mathrm{distr}(\mathbf{y}) = \phi$,
	with $\phi(i) = i \bmod p$, for $0 \leq i < n$.
output:	$\alpha = \mathbf{x}^T \mathbf{y}$.

```
(0)   $\alpha_s := 0$;
      **for** $i := s$ **to** $n - 1$ **step** $p$ **do**
          $\alpha_s := \alpha_s + x_i y_i$;

(1)   **for** $t := 0$ **to** $p - 1$ **do**
          put $\alpha_s$ in $P(t)$;
```

Parallel inner product algorithm for $P(s)$

input: $\mathbf{x}, \mathbf{y}$: vector of length n,
$\mathrm{distr}(\mathbf{x}) = \mathrm{distr}(\mathbf{y}) = \phi$,
with $\phi(i) = i \bmod p$, for $0 \le i < n$.

output: $\alpha = \mathbf{x}^T \mathbf{y}$.

```
(0)  $\alpha_s := 0$;
     for $i := s$ to $n-1$ step $p$ do
         $\alpha_s := \alpha_s + x_i y_i$;

(1)  for $t := 0$ to $p-1$ do
         put $\alpha_s$ in $P(t)$;

(2)  $\alpha := 0$;
     for $t := 0$ to $p-1$ do
         $\alpha := \alpha + \alpha_t$;
```

Single Program, Multiple Data (SPMD)

- Only one program text needs to be written. All processors run the same program, but on their own data.
- The program text is parametrised in the processor number s, $0 \leq s < p$, also called processor identity.
- The execution of the program depends on s.

Single Program, Multiple Data (SPMD)

- Only one program text needs to be written. All processors run the same program, but on their own data.
- The program text is parametrised in the processor number s, $0 \leq s < p$, also called processor identity.
- The execution of the program depends on s.
- Processor $P(s)$ computes a **local** partial inner product

$$\alpha_s = \sum_{0 \leq i < n,\ i \bmod p = s} x_i y_i.$$

- The corresponding computation superstep (0) costs

$$2 \left\lceil \frac{n}{p} \right\rceil + l.$$

(1 addition and 1 multiplication per local vector component.)

Result needed on all processors

- The partial inner products must be added. This could have been done by $P(0)$, i.e. processor 0.

Result needed on all processors

- The partial inner products must be added. This could have been done by $P(0)$, i.e. processor 0.
- Sending the α_s to $P(0)$ is a $(p-1)$-relation. Sending them to $P(*)$, i.e., to all the processors, costs the same. The cost is $(p-1)g+l$.

Result needed on all processors

- The partial inner products must be added. This could have been done by $P(0)$, i.e. processor 0.
- Sending the α_s to $P(0)$ is a $(p-1)$-relation. Sending them to $P(*)$, i.e., to all the processors, costs the same. The cost is $(p-1)g+l$.
- Computing α on $P(0)$ costs the same as computing it on all the processors redundantly, i.e. in a replicated fashion. The cost is $p+l$.

Result needed on all processors

- The partial inner products must be added. This could have been done by $P(0)$, i.e. processor 0.
- Sending the α_s to $P(0)$ is a $(p-1)$-relation. Sending them to $P(*)$, i.e., to all the processors, costs the same. The cost is $(p-1)g+l$.
- Computing α on $P(0)$ costs the same as computing it on all the processors redundantly, i.e. in a replicated fashion. The cost is $p+l$.
- Often, the result is needed on all processors, e.g. for iterative linear system solvers. This algorithm does just this.
- Sending the local result to all processors is best if each processor contributes one value. If there are more values per processor, a different broadcasting approach might be better.

Total BSP cost of inner product

$$T_{\mathrm{inprod}} = 2\left\lceil \frac{n}{p} \right\rceil + p + (p-1)g + 3l.$$

One-sided communication

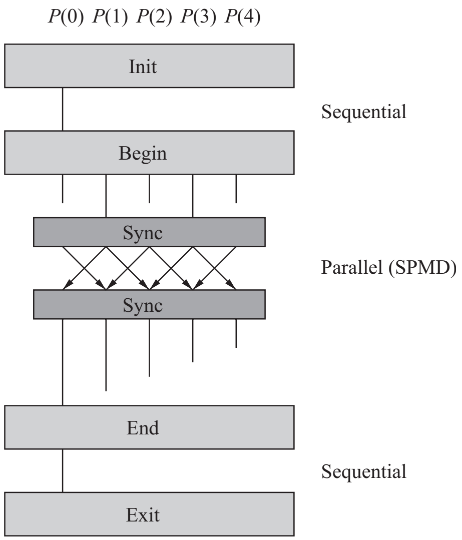
- The 'put' operation involves an active sender and a passive receiver. We assume all puts are accepted. Thus we can define each data transfer by giving only the action of **one side**.
- No clutter in programs: **shorter and simpler** texts.
- No danger of the dreaded **deadlock**. Deadlock can easily occur in two-sided message passing, with an active sender and an active receiver that must shake hands. This may cause lots of problems, e.g., What happens if both processors want to receive first?
- Another one-sided operation is the 'get'. The name says it all.
- One-sided communications are **more efficient**.

Summary

- We design algorithms in Single Program, Multiple Data style. Each processor runs its own copy of the same program, on its own data.
- The block and cyclic distributions are commonly used in parallel computing. Both are suitable for an inner product computation.
- The BSP style encourages balancing the communication among the processors. Sending all data to one processor is discouraged. Better: all to all.
- One-sided communications such as puts and gets are easy to use and efficient.
- The BSP cost is transparently calculated.

BSPlib, the BSP interface

BSPlib program: sequential, parallel, sequential



BSPlib programs: top-level structure

- A BSPlib program starts with a sequential part, mainly intended for input. Motivation:
  - Desired number of processors of the parallel part may depend on the input.
  - Input of data describing a problem is often sequential.
- A BSPlib program ends with a sequential part, mainly intended for output. Motivation:
  - Reporting the output of a computation is often sequential.
- Sequential I/O in a parallel program may be inherited from a sequential program.
- The sequential parts may also be empty.

Main function of BSPlib program

```
int P;
int main(int argc, char **argv){
    bsp_init( bspinprod, argc, argv );

    printf("How many processors?\n"); /* sequential part */
    scanf( "%d", &P );
    if( P > bsp_nprocs() ) {
        printf( "Sorry, not enough available.\n" );
        exit( 1 );
    }

    bspinprod();                        /*  parallel part  */

    exit( 0 );                          /* sequential part */
}
```

Primitive bsp_init

```
bsp_init( spmd, argc, argv );
```

- The BSPlib primitive `bsp_init` initialises the program. It must be the first executable statement in the program.
- `spmd` is the name of the function that comprises the parallel part . In our example, the name is `bspinprod`.
- The primitive `bsp_init` is needed to circumvent restrictions of certain machines.
- It is ugly and often misunderstood. (But then, what happened to Quasimodo in the end?)
- `int argc` is the number of command-line arguments and `char **argv` is the array of arguments. These arguments can be used in the sequential input part, but they **cannot** be transferred to the parallel part.

Structure of SPMD part

```
void bspinprod() {
    int p, s, n;

    bsp_begin( P );
    p = bsp_nprocs(); /* p = number of procs  */
    s = bsp_pid();    /* s = processor number */
    if( s == 0 ) {
        printf( "Please enter n:\n" );
        scanf( "%d", &n );
        if( n < 0 )
            bsp_abort( "Error in input: n < 0" );
    }
    ...
    bsp_end();
}
```

Primitives bsp_begin, bsp_end

```
bsp_begin( reqprocs );
bsp_end();
```

- The BSPlib primitive bsp_begin starts the parallel part of the program with reqprocs processors. It must be the first executable statement in the SPMD function.
- The BSPlib primitive bsp_end ends the parallel part of the program. It must be the last executable statement in the SPMD function.
- If the sequential parts of the program are empty, main can become the parallel part and bsp_init can be removed.
- $P(0)$ inherits the values of the variables from the sequential part and can use these in the parallel part. Other processors do not inherit any values and must obtain needed values by explicit communication.

Primitives `bsp_nprocs`, `bsp_pid`

```
bsp_nprocs();
bsp_pid();
```

- The BSPlib primitive `bsp_nprocs` gives the number of processors. In the parallel part, this is the actual number p of processors involved in the parallel computation. In the sequential parts, it is the maximum number available.
- Thus, we can ask how many processors are available and then decide not to use them all. **Sometimes, using fewer processors gives faster results!**
- The BSPlib primitive `bsp_pid` gives the processor identity s, where $0 \leq s < p$.
- Both primitives can be used anywhere in the parallel program, so you can always get an answer to burning questions such as: How many are we? Who am I?

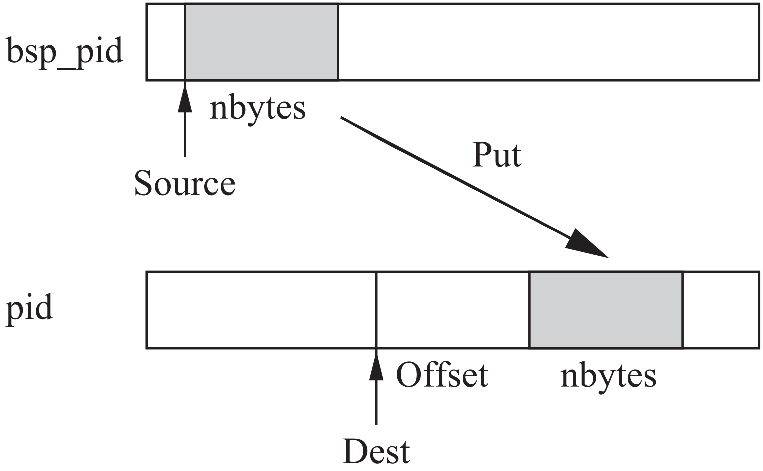
Primitive bsp_abort

```
bsp_abort( error_message );
```

- If one processor detects that something is wrong, it can bring all processors down in a graceful manner and print an error message by using `bsp_abort`.
- The message is in the standard format of the C-function `printf`.

Putting data into another processor

```
bsp_put(pid, source, dest, offset, nbytes);
```



Primitive bsp_put

```
bsp_put(pid, source, dest, offset, nbytes);
```

- The `bsp_put` operation **copies** `nbytes` of data from the local processor `bsp_pid` into the specified destination processor `pid`.
- The pointer `source` points to the start of the data to be copied.
- The pointer `dest` specifies the start of the memory area where the data is written.
- The data is written at `offset` bytes from the start.
- This is the most-often used one-sided communication operation.

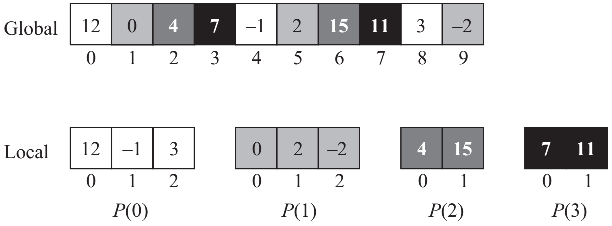
Inner product function

```
double bspip( int p, int s, int n, double *x, double *y ) {
    double inprod, *Inprod;                 /* Initialisation */
    int i, t;
    Inprod = vecallocd( p );
    bsp_push_reg( Inprod, p * SZDBL );
    bsp_sync();

    inprod = 0.0;                           /*  Superstep 0   */
    for ( i = 0; i < nloc( p, s, n ); i++ )
        inprod += x[i] * y[i];

    for(t = 0; t < p; t++ )                 /*  Superstep 1   */
        bsp_put( t, &inprod, Inprod, s*SZDBL, SZDBL );
    bsp_sync(); ...
}
```

Local and global indices for cyclic distribution



Global index: i, local index on $P(s)$: i. Their relation is:

$$i = \mathtt{i} \cdot p + s.$$

Use local indices in programs:

```
for( i = 0; i < nloc(p, s, n); i++ )
    inprod += x[i] * y[i];
```

Primitive bsp_get

```
bsp_get(pid, source, offset, dest, nbytes);
```

- The `bsp_get` operation copies `nbytes` of data from the specified remote source processor `pid` into the local processor `bsp_pid`.
- The pointer `source` points to the start of the data in the remote processor to be copied.
- The pointer `dest` specifies the start of the local memory area where the data is written.
- The data is read starting at `offset` bytes from the start of `source`.
- Remember for both puts and gets: the source parameter comes first and the offset is in the remote processor.

Getting n from $P(0)$

```
void bspinprod() {

    int p, s, n;
    ...
    if( s == 0 ) {
        printf( "Please enter n:\n" );
        scanf( "%d", &n );
    }
    bsp_push_reg( &n, SZINT);
    bsp_sync();

    bsp_get( 0, &n, 0, &n, SZINT );
    bsp_sync();
    ...
}
```

Primitive bsp_sync

```
bsp_sync();
```

- The `bsp_sync` operation terminates the current superstep. It causes all communications initiated by puts and gets to be actually carried out. It synchronises all the processors.
- After the `bsp_sync`, the communicated data can be used.

Safety first: no interference

- The regular `bsp_put` and `bsp_get` operations are doubly buffered, at the source and at the destination. This provides safety:
- a data word that is put is first copied into a local send buffer;
- the original data word can be reused immediately.
- All received data are first stored in a receive buffer, and made available after a barrier.

Safety first: no interference

- The regular `bsp_put` and `bsp_get` operations are doubly buffered, at the source and at the destination. This provides safety:
- a data word that is put is first copied into a local send buffer;
- the original data word can be reused immediately.
- All received data are first stored in a receive buffer, and made available after a barrier.
- All communication is postponed until all computations are finished (in the current superstep). The value obtained by a get thus is the value at the moment computations were finished.
- If you like living on the edge: the `bsp_hpput` primitive is **unbuffered**, more efficient than `bsp_put`, and requires less memory. There is also a `bsp_hpget`.
- Unbuffered communicatoins are considered dangerous.

Your x is my x

```
bsp_push_reg( variable, nbytes );
bsp_pop_reg( variable);
```

- A variable called `x` may have the same name on different processors, but not the same address in memory;
- therefore, variables must be **registered** first.
- All processors participate in the registration procedure by pushing their variable and its memory size onto a stack. (The unwilling ones can register `NULL`.)
- Variables must be registered in the same order across all processors.
- Registration takes effect only in the next superstep.
- Deregistration is also done by all processors collectively.

Registration is expensive

To register, all processors have to talk to each other, which takes some time; it is a *p*-relation in the worst case.

- Try to register sparingly.
- Register once, put many times.

BSP timer measures elapsed time

```
...
bsp_sync();
time0 = bsp_time();

alpha = bspip( p, s, n, x, x );
bsp_sync();
time1=bsp_time();

if( s==0 )
    printf( "This took only %.6lf seconds.\n",
            time1 - time0 );
...
```

Summary

- BSPlib is a small library of 20 primitives for writing parallel programs in bulk synchronous parallel style.
- We have learned 12 primitives and with them, we are perfectly equipped for parallel programming.
- The put and get primitives provide Direct Remote Memory Access (DRMA).
- Registration allows direct access to dynamically allocated memory.

Programming using BSPlib

Available BSP libraries

Distributed-memory computers:

1. Oxford BSP library, see `www.bsp-worldwide.org`
2. BSPonMPI, see `bsponmpi.sourceforge.net`

Shared-memory computers, see `www.multicorebsp.com`:

1. MulticoreBSP for Java
2. **MulticoreBSP for C**

Compilation of BSPlib programs

Oxford BSP Toolset (Hill, McColl, Stefanescu, Goudreau, Lang, Rao, Suel, Tsantilas, Bisseling; 1998)

```
bspcc bspinprod.c;
bsprun -npes <P> ./a.out
```

BSPonMPI (van Suijlen; 2006)

```
mpcc bspinprod.c -lbsponmpi;
./a.out
```

MulticoreBSP for C (Yzelman, Bisseling, Roose, Meerbergen; 2012)

```
cc bspinprod.c -lmcbsp -pthread -lrt;
./a.out
```

MulticoreBSP for Java (Yzelman, Bisseling; 2010)

```
javac -cp MulticoreBSP.jar bspinprod.java;
java -cp MulticoreBSP.jar bspinprod
```

Other BSP libraries

Other (distributed memory) BSP-style libraries (incompatible API):

- MapReduce

  Google Inc.; 2004.
- Pregel

  Malewicz, Austern, Bik, Dehnert, Ilan Horn, Czajkowski (Google Inc.); 2010.
- Apache Hama

  Yoon et al.; 2010
- Paderborn University BSP library (PUB)

  Bonorden, Juurlink, von Otte, Rieping; 1998.
- Bulk Synchronous Parallel ML (BSMLlib)

  Gava, Gesbert, Hains, Tesson; 2000.
- Python/BSP

  Hinsen, Sadron; 2003.
- Cloudscale BSP

  McColl et al. (Cloudscale Inc.); 2012.

Hello World!

```
#include "bsp.h" / "mcbsp.h"

int main(int argc, char **argv) {
    bsp_begin( 4 );

    printf( "Hello world from thread %d out of %d!\n",
        bsp_pid(), bsp_nprocs() );

    bsp_end();
}
```

Hello World!

An example using MulticoreBSP for C and static linkage:

```
$ gcc -o hello hello.c lib/libmcbsp1.1.0.a -pthread -lrt
$ ./hello
Hello world from thread 3 out of 4!
Hello world from thread 2 out of 4!
Hello world from thread 0 out of 4!
Hello world from thread 1 out of 4!
$ ...
```

Hello World!

```
#include "mcbsp.h"
int P;

void hello() {
    bsp_begin( P );
    printf( "Hello world from thread %d!\n",
        bsp_pid() );
    bsp_end();
}

int main(int argc, char **argv) {
    bsp_init( hello, argc, argv );
    scanf( "%d", &P );
    hello();
}
```

Summary

- There are multiple implementations of the BSPlib interface.
- Other BSP-style libraries for parallel programming exist; some stay close to the BSP model, others do not.
- We have seen the syntax for compilation for various BSPlib libraries, and applied these on a 'Hello world!' example.
- The complete program `bspinprod` should now be clear from the last section. Try to compile it and to run it!