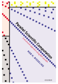
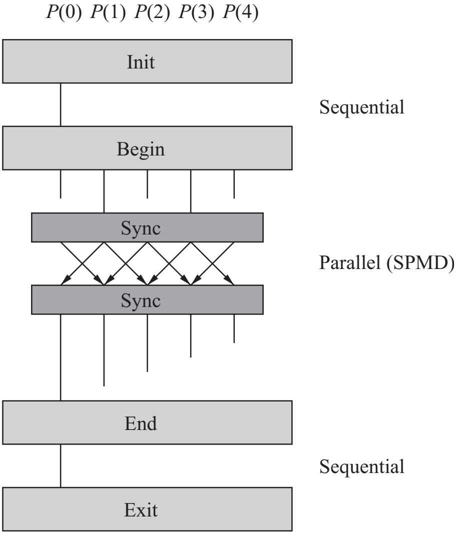
BSPlib, the BSP library
(PSC §1.4)



BSPlib program: sequential, parallel, sequential



$P(0)$ $P(1)$ $P(2)$ $P(3)$ $P(4)$

Init

Sequential

Begin

Sync

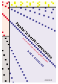
Parallel (SPMD)

Sync

End

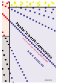
Sequential

Exit



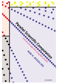
Sequential I, parallel computation, sequential O

- 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

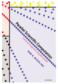
```
int P;
int main(int argc, char **argv){
    bsp_init(bspinprod, argc, argv);

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

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



Primitive bsp_init

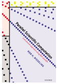
```
bsp_init (spmd, argc, argv) ;
```

- 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 (written in SPMD style: Single Program, Multiple Data). 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

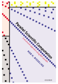
```
void bspinprod(){
    int p, s, n;

    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 () ;
```

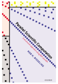
- The BSPlib primitive bsp_begin starts the parallel part of the program with reqprocs processors. It must be the first executable statement in the SPMD function.
- The BSPlib primitive bsp_end ends the parallel part of the program. It must be the last executable statement in the SPMD function.
- If the sequential parts of the program are empty, main can become the parallel part and bsp_init can be removed, but it is better to leave it in.
- $P(0)$ 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 () ;
```

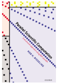
- The BSPlib primitive bsp_nprocs gives the number of processors. In the parallel part, this is the actual number p of processors involved in the parallel computation. In the sequential parts, it is the maximum number available.
- Thus, we can ask how many processors are available and then decide not 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?



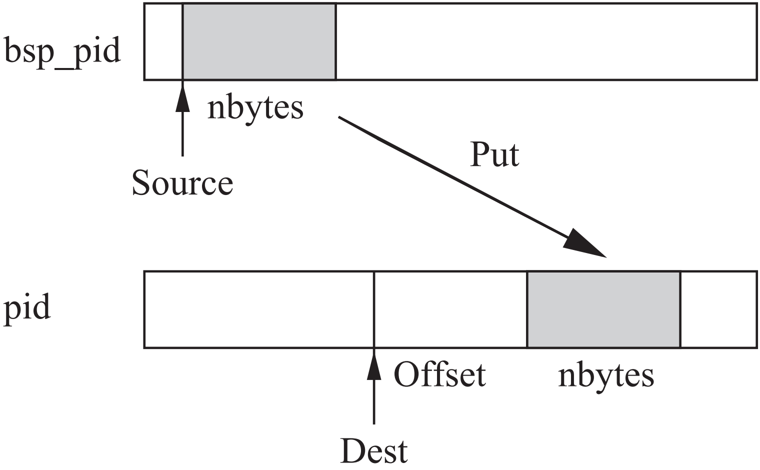
Primitive bsp_abort

```
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



Primitive bsp_put

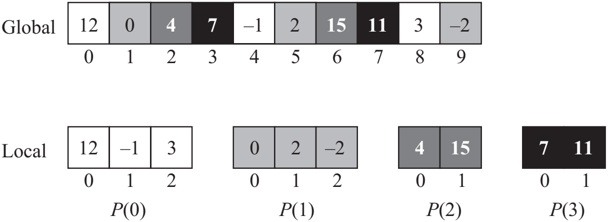
```
bsp_put(pid, source, dest, offset, nbytes);
```

- 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 important one-sided communication operation.

Inner product function

```
double bspip(int p, int s, int n, double *x, double *y){

    double inprod, *Inprod;
    int i, t;

    Inprod= vecallocd(p);
    bsp_push_reg(Inprod,p*SZDBL);
    bsp_sync();

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

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

Local and global indices for cyclic distribution



Global	12	0	4	7	–1	2	15	11	3	–2
	0	1	2	3	4	5	6	7	8	9

	$P(0)$			$P(1)$			$P(2)$		$P(3)$	
Local	12	–1	3	0	2	–2	4	15	7	11
	0	1	2	0	1	2	0	1	0	1

Global index: i
Local index on $P(s)$: $\mathtt{i}$
Relation: $i = \mathtt{i} \cdot p + s$

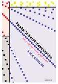
Use local indices in programs:

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

Primitive bsp_get

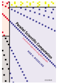
```
bsp_get(pid, source, offset, dest, nbytes);
```

- 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();
    ...
}
```



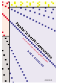
Primitive bsp_sync

```
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

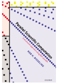
- The regular `bsp_put` and `bsp_get` operations are doubly buffered, at the source and at the destination. This provides safety.
- A data word that is put is first copied into a local send buffer. The space occupied by the original data word can be reused immediately.
- All received data are first stored in a receive buffer.
- All communication is postponed until the moment all computations of the current superstep are finished. The value obtained by a get is the value at the moment computations are finished.
- If you like living on the edge: the `bsp_hpput` primitive is unbuffered, more efficient than `bsp_put`, uses less memory, but is 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 this does not guarantee that it has the same actual address in memory.
- To provide access to the proper addresses, the names 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.
- The SPMD style suggests registering the same variable name on all processors, but this is not strictly necessary.
- Registration takes effect only in the next superstep.
- Deregistration is done by all processors together popping the variable from the stack.

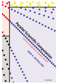


Registration is expensive

- To register, all processors have to talk to each other, which takes some time.
- 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

- SMALL IS BEAUTIFUL
- BSPlib is a small library of 20 primitives for writing parallel programs in bulk synchronous parallel style.
- We have learned 12 primitives and are ready to start programming in parallel.
- The put and get primitives provide RDMA (Remote Direct Memory Access, also called DRMA).
- Registration allows direct access to dynamically allocated memory.
- The complete program `bspinprod` should now be clear. Try to compile it using `bspcc` (or an equivalent command) and run it on 4 processors using `bsprun -npes 4`.