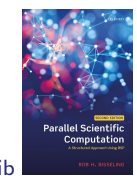
Starting with BSPlib

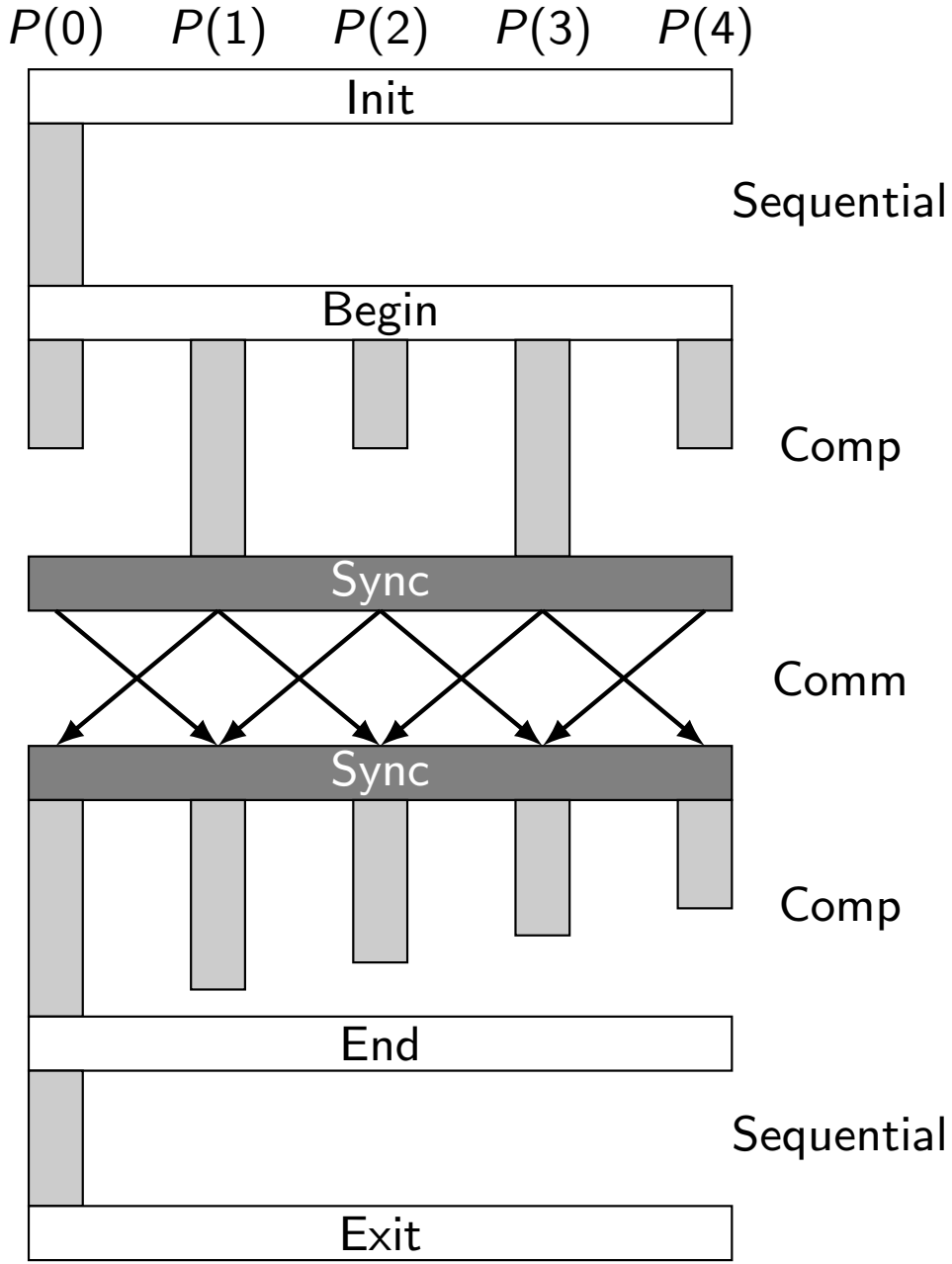
Section 1.4 of Parallel Scientific Computation, 2nd edition

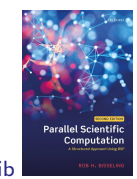
Rob H. Bisseling

Utrecht University



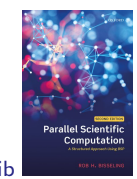
BSPlib program: sequential, parallel, sequential





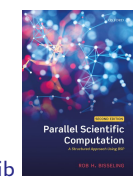
Sequential input, parallel computation, sequential output

- A BSPlib program starts with a sequential part, mainly intended for input. Motivation:
  - The desired number of processors of the parallel part may depend on the input.
  - The 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

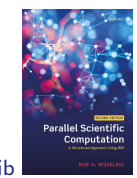
```
long P;
int main(int argc, char **argv){
    bsp_init(bspinprod, argc, argv);

    /* Sequential part */
    printf("How many processors do you want to use?\n");
    fflush(stdout);
    scanf("%ld",&P);
    if (P > bsp_nprocs()){
        printf("Sorry, only %u processors available.\n",
                bsp_nprocs());
        exit(EXIT_FAILURE);
    }

    /* SPMD part */
    bspinprod();

    /* Sequential part */
    exit(EXIT_SUCCESS);
}
```



Primitive bsp_init

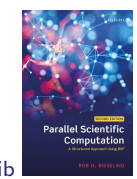
```
bsp_init (spmd, argc, argv);
```

- The BSPlib primitive `bsp_init` initializes 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 a bit ugly and often misunderstood.
- `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

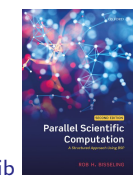
```
void bspinprod(){

    bsp_begin(P);
    long p= bsp_nprocs();
    long s= bsp_pid();
    long n;
    if (s==0){
        printf("Please enter n:\n");
        fflush(stdout);
        scanf("%ld",&n);
        if(n<0)
            bsp_abort("Error in input: n is negative");
    }
    ...
    bsp_end();
}
```



Primitives bsp_begin, bsp_end

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

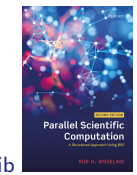
- The BSPlib primitive `bsp_begin` starts the parallel part of the program with the requested `reqprocs` processors.
- The BSPlib primitive `bsp_end` ends the parallel part of the program.
- `bsp_begin` and `bsp_end` must be the first and last executable statements, respectively, in the SPMD function.
- $P(0)$ 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();
```

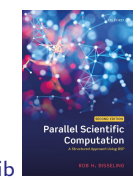
- The BSPlib primitive `bsp_nprocs` gives the number of processors. In the parallel part, this is the actual number p of processors involved in the parallel computation. In the sequential parts, it is the maximum number available.
- Thus, we can ask how many processors are available and then decide not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



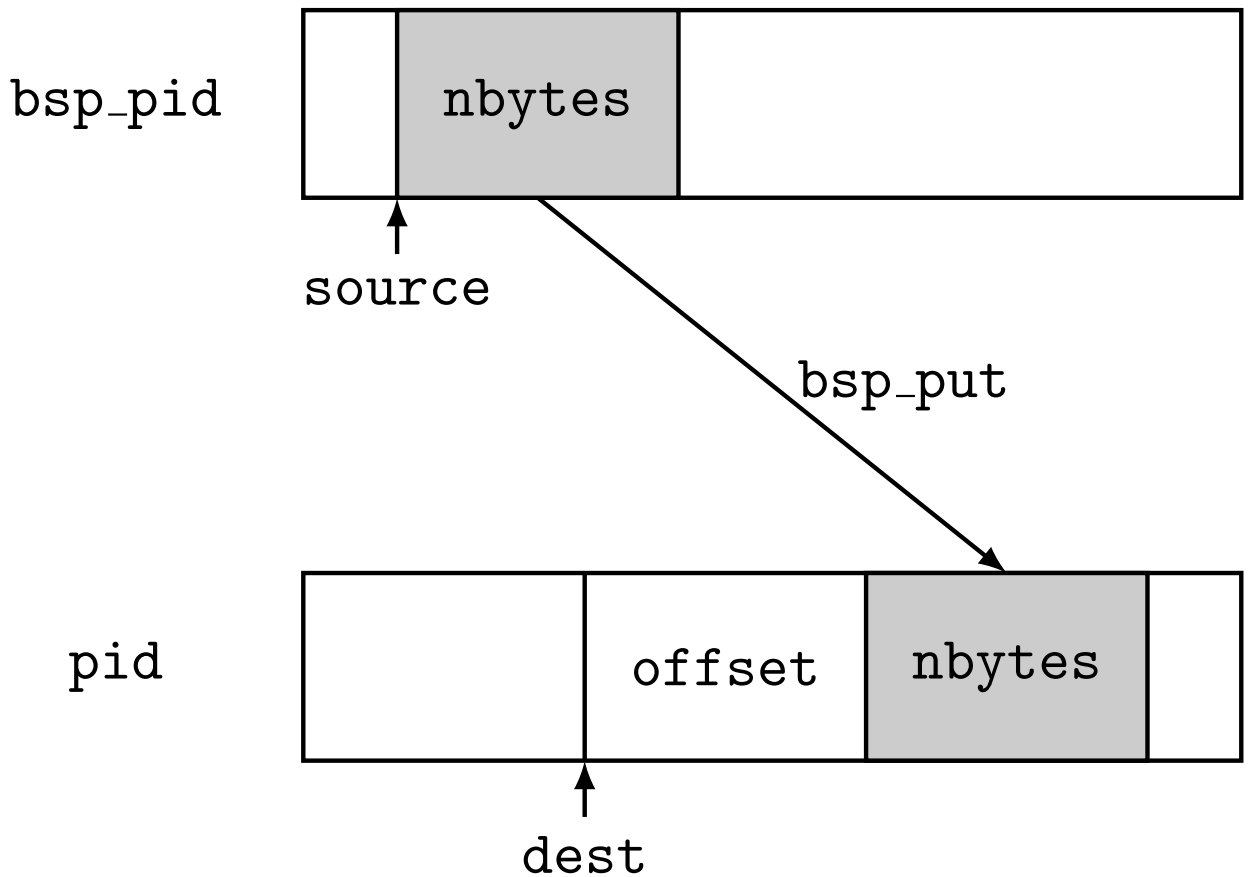
Primitive bsp_abort

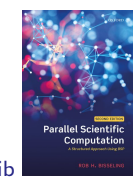
```
bsp_abort(error_message);
```

- 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

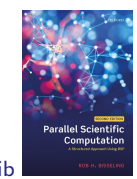




Primitive bsp_put

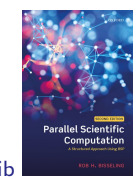
```
bsp_put(pid, source, dest, offset, nbytes);
```

- 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 will be written.
- The data is written at `offset` bytes from the start.
- This is the most important one-sided communication operation.

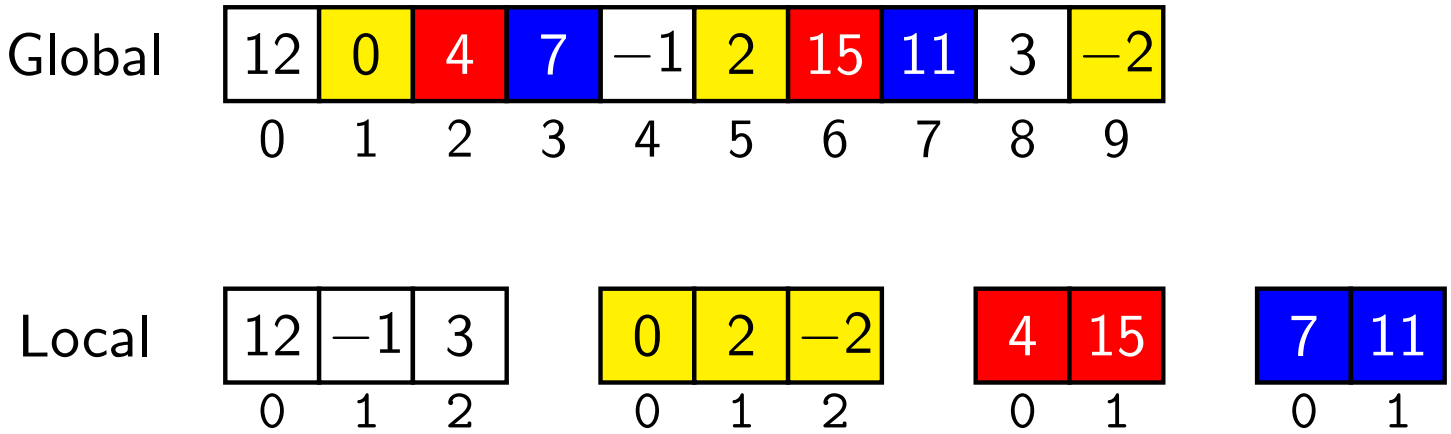


Inner product function

```
double bspip(long n, double *x, double *y){

    long p= bsp_nprocs();
    long s= bsp_pid();

    double *lnprod= vecallocd(p);
    bsp_push_reg(lnprod,p*sizeof(double));
    bsp_sync();

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

    for (long t=0; t<p; t++)
        bsp_put(t,&inprod,lnprod,s*sizeof(double),
                sizeof(double));
    bsp_sync();
    ...
}
```



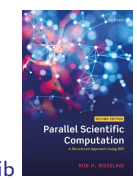
Local and global indices for cyclic distribution



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

Use local indices in programs:

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



Parallel algorithms and parallel programs

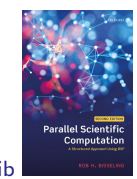
Parallel algorithms:

- are meant for humans;
- give just enough detail to be understood;
- use global variables in unambiguous mathematical notation.

Parallel programs:

- are meant for compilers;
- give all the gory details;
- use local variables valid for a processor $P(s)$.

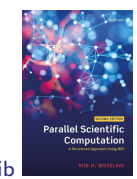
Advice: develop a parallel algorithm first, then implement it as a parallel program!



Primitive bsp_get

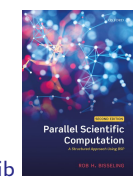
```
bsp_get(pid, source, offset, dest, nbytes);
```

- 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 will be written.
- The data is read starting at offset bytes from source.
- Remember for both puts and gets: the source parameter comes first and the offset is in the remote processor.



Getting n from $P(0)$

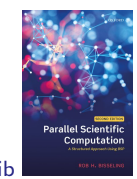
```
void bspinprod(){

    long n;
    ...
    if (s==0){
        printf("Please enter n:\n");
        scanf("%ld",&n);
    }
    bsp_push_reg(&n,sizeof(long));
    bsp_sync();

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



Primitive bsp_sync

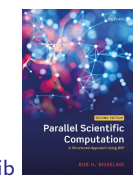
```
bsp_sync();
```

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



Safety first: no interference

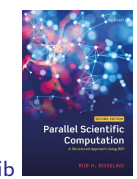
- The regular `bsp_put` and `bsp_get` operations are doubly buffered, at the source and the destination, to provide safety.
- A data word that is put is first copied into a local send buffer. The space occupied by the original data word can be reused immediately.
- All received data are first stored in a receive buffer.
- All communication is delayed until the moment all computations of the current superstep are finished. The value obtained by a get is the value at that moment.
- If you like living on the edge: the high-performance primitive `bsp_hpput` is unbuffered and more efficient than `bsp_put`, and it uses less memory, but it is considered dangerous.



Registration: your x is my x

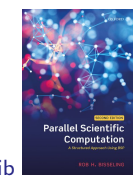
```
bsp_push_reg(variable, nbytes);
```

- 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, for instance when memory is allocated dynamically.
- To link the 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. Unwilling processors 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.



Registration is expensive

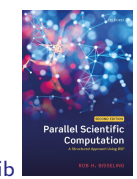
- To register, all processors have to talk to each other, which takes some time.
- Try to register sparingly.
- Register once, put many times.



Deregistration

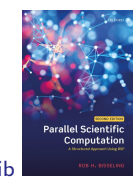
```
bsp_pop_reg(variable);
```

- Deregistration is done by all processors together popping the variable from the stack.



BSP timer measures elapsed time

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

double alpha= bspip(n,x,x);

bsp_sync();
double 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 22 primitives for writing parallel programs in bulk synchronous parallel style.
- We have learned 12 primitives (not counting `bsp_hpput`), and are now ready to start programming in parallel.
- The put and get primitives provide Remote Direct Memory Access (RDMA, 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` and run it using `bsprun`.