BSPlib
The BSP library

PSC §1.4
BSPlib program: sequential, parallel, sequential

\[ P(0) \ P(1) \ P(2) \ P(3) \ P(4) \]

Sequential (input)

Parallel (SPMD)

Sequential (output)
BSP Program Outline

• A BSPlib program starts with a sequential part
  • Mainly intended for input
  • Desired number of processors of the parallel part may depend on the input
  • Input of data describing a problem is often sequential
• Parallel part: SPMD
• A BSPlib program ends with a sequential part
  • Mainly intended for output
  • 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

```c
int P;
int main(int argc, char **argv) {
    bsp_init(bspinprod, argc, argv);
    /* sequential part (input) */
    printf("How many processors?\n");
    scanf("%d", &P);
    if (P > bsp_nprocs()) { /* real #procs */
        printf("Not enough available.\n");
        exit(1);
    }
    /* parallel part */
    bspinprod();
    /* sequential part (output) */
    exit(0);
}
```
Primitive `bsp_init`

`bsp_init(spmd, argc, argv)`
- The BSPlib primitive `bsp_init` initializes the BSP environment
- 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`
- It is ugly and often misunderstood
  - But then, what happened to Quasimodo in the end?
- `argc` is the number of command-line arguments and `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

```c
void bspinprod() {
    int p, s, n;
    bsp_begin(P);
    p = bsp_nprocs(); /* number of procs */
    s = bsp_pid(); /* 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)`
- 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

`bsp_end()`
- 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
## BSP Initiation Summary

<table>
<thead>
<tr>
<th>Program</th>
<th>Main Process</th>
<th>Sub Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>bsp_init</td>
<td>Implementation dependent initialization</td>
<td>Calling SPMD function and then exit(0)</td>
</tr>
<tr>
<td>Sequential code before the SPMD</td>
<td>Runs only here</td>
<td>Does not run here</td>
</tr>
<tr>
<td>Direct call to SPMD function</td>
<td>Runs only here</td>
<td>Does not run here</td>
</tr>
<tr>
<td>bsp_begin</td>
<td>Creates sub-processes And waits on barrier</td>
<td>Wait on barrier</td>
</tr>
<tr>
<td>SPMD implementation</td>
<td>Runs</td>
<td>Runs</td>
</tr>
<tr>
<td>bsp_end</td>
<td>Waits all of sub-processes</td>
<td>Notify main process</td>
</tr>
<tr>
<td>Sequential code after the SPMD</td>
<td>Runs only here</td>
<td>Does not run here</td>
</tr>
</tbody>
</table>
Primitives `bsp_nprocs`, `bsp_pid`

**`bsp_nprocs()`**
- 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!

**`bsp_pid()`**
- The BSPlib primitive `bsp_pid` gives the processor identity $s$
  - where $0 \leq s < p$
  - The master process is always given the id 0
- Both primitives can be used from 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

\texttt{bsp_abort(error\_message, \ldots)}

- 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 \texttt{bsp\_abort}
- The message is in the standard format of the C-function \texttt{printf}
Your \textit{x} is my \textit{x}

\textbf{\texttt{bsp\_push\_reg}(variable, nbytes)}

- A variable called \textit{x} may have the same name on different processors, but this does not guarantee that it has the same actual address in memory
- To guarantee correctness, the variables must be registered first
- \textbf{All processors participate} in the registration procedure by pushing their variable and its memory size onto a stack
- The unwilling ones can register \texttt{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 \texttt{superstep}

\textbf{\texttt{bsp\_pop\_reg}(variable)}

- Deregistration is done by \textbf{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, use many times.
Putting data into another processor

\[ \text{bsp\_put}(\text{pid}, \text{source}, \text{dest}, \text{offset}, \text{nbytes}) \]
Primitive bsp_put

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

- The `bsp_put` operation copies `nbytes` of data from the local processor `my_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
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 my_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
  • The offset is in the remote processor
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
Getting $n$ from $P(0)$

```c
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();
    ...
}
```
Inner product function

def double bspip(int p, int s, int n, double *x, double *y) {
    double inprod_res=0.0, *inprod_vec;

    inprod_vec = vecallocated(p);
    bsp_push_reg(inprod_vec, p * SZDBL);
    bsp_sync();
    for (int i=0; i < nloc(p,s,n); i++)
        inprod_res += x[i]*y[i];
    for (int t=0; t < p; t++)
        bsp_put(t,&inprod_res, inprod_vec, s * SZDBL, SZDBL);
    bsp_sync();
    ...
}
Local and global indices for cyclic distribution

Global Data

<table>
<thead>
<tr>
<th>Global Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Data</td>
<td>12</td>
<td>-1</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Global Data:

<table>
<thead>
<tr>
<th>Global Data</th>
<th>12</th>
<th>0</th>
<th>4</th>
<th>7</th>
<th>-1</th>
<th>2</th>
<th>15</th>
<th>11</th>
<th>3</th>
<th>-2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Index</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

Local Data:

<table>
<thead>
<tr>
<th>Local Data</th>
<th>12</th>
<th>-1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local Index</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Global index: \( i \)

Local index on \( P(s) \): \( \text{local}_i = i \cdot P + s \)

Use local indices in programs:

```c
for (i=0; i < nloc(p,s,n); i++)
    inprod += x[i]*y[i];
```
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
BSP timer measures elapsed time

...*

```c
bsp_sync();
time0 = bsp_time();
alpha = bspip(p, s, n, x, x);
bsp_sync();
timel = bsp_time();
if (s == 0)
    printf("bspip took %.6lf seconds.\n", timel-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
• Run it on 4 processors using bsprun -npes 4