All partial sums of n values, where $n > p$

Jonathan M.D. Hill

```
#include "bsp.h"
#include <stdio.h>
#include <stdlib.h>
```

1 All sums: version 1

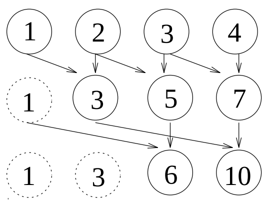

Figure 1: All sums using the parallel prefix technique

The function `bsp_allsums1` calculates the running sums of p integers stored on p processors. That is, given $x_0, \ldots, x_{p-1}$ (where x_i is stored on process i), the result on process i is $x_0 + \cdots + x_i$. The algorithm uses the parallel prefix technique that performs $\lceil \log p \rceil$ supersteps, such that during the k^{th} superstep, the processes in the range $2^{k-1} \leq i < p$ combine there local partial sums with process $i - 2^{k-1}$. Figure **??** shows the steps involved in summing the values `1+bsp_pid()` stored among 4 processors.

```
int bsp_allsums1(int x) {
  int i, left, right;
  bsp_pushregister(&left,sizeof(int));
  bsp_sync();

  right = x;
  for(i=1;i<bsp_nprocs();i*=2) {
```

```
    if (bsp_pid()+i < bsp_nprocs())
      bsp_put(bsp_pid()+i,&right,&left,0,sizeof(int));
    bsp_sync();
    if (bsp_pid()>=i) right = left + right;
  }
  bsp_popregister(&left);
  return right;
}
```

The procedure `bsp_pushregister` allows all processors to declare that the variable `left` is willing to have data put into it during a DRMA operation. The reason why registration is required is that each processors copy of the data-structure `left` isn't necessarily stored at the same address on all processors. Registration therefore creates a correspondence between data-structures that have the "same name" on different processors.

When `bsp_put(bsp_pid()+i,&right,&left,0,sizeof(int))` is executed on process *bsp_pid*(), then a single integer `left` is copied into the memory of processor `bsp_pid()+i` at the address `&left+0` (where `left` is a previously registered data-structure).

The cost of the algorithm is $\lceil \log p \rceil (1+g+l)+l$ as there are $\lceil \log p \rceil + 1$ supersteps (including one for registration); such that during each superstep a local addition is performed (which costs 1 flop), and at-most one message of size 1 word enters and exits each process (this attributes a cost of g to each superstep).

2 All sums: version 2

An alternative implementation of the all partial sums function can be achieved in a single superstep by using a temporary data-structure containing at-most p integers. Each process i puts the data to be summed into the i^{th} element of the temporary array on processes j (where $0 \leq j \leq$ `bsp_pid`). After all communications has been completed, a local sum is then performed on the accumulated data. The cost of the algorithm is $(p-1)+(p-1)g+2l$.

```
int bsp_allsums2(int x) {
  int i, result,*array = calloc(bsp_nprocs(),sizeof(int));
  if (array==NULL)
    bsp_abort("Unable to allocate %d element array",bsp_nprocs());
  bsp_pushregister(array,bsp_nprocs()*sizeof(int));
```

```
  bsp_sync();

  for(i=bsp_pid();i<bsp_nprocs();i++)
    bsp_put(i,&x,array,bsp_pid()*sizeof(int),sizeof(int));
  bsp_sync();

  result = array[0];
  for(i=1;i<=bsp_pid();i++) result += array[i];
  free(array);
  bsp_popregister(array);
  return result;
}
```

Comparing the costs of the two algorithms, the first algorithm performs a logarithmic number of additions and supersteps, whereas the later algorithm performs a linear number of additions but a constant number of supersteps. If the operation being performed at each iteration of the algorithm were changed from addition to another, more costly associative operator, then BSP cost analysis provides a simple mechanism to determine which is the optimal algorithm on a given architecture.

3 All sums on an array

Either of the routines defined above can be used to sum n values held in n/p blocks distributed among p processors. The algorithm proceeds in four phases:

1. The running sums of each n/p block of integers is performed locally on each processor.

2. As the last element of each n/p block will contain the sum of each n/p element segment, then `bsp_allsums1` can be used to calculate what the running sums of the last element in each block is (we call this `last`).

3. Each processor gets the value of `last` from their left neighbouring processor (we call this `left_last`).

4. Adding `left_last` to each of the locally summed n/p elements produces the desired effect of the running sums of all n elements.

```
void bsp_allsums(int *array, int n_over_p) {
  int i, last, lefts_last;
  bsp_pushregister(&last,sizeof(int));

  for (i=1;i<n_over_p;i++)
    array[i] += array[i-1];

  last = bsp_allsums1(array[n_over_p-1]);

  if (bsp_pid()==0) lefts_last=0;
  else
    bsp_get(bsp_pid()-1,&last,0,&lefts_last,sizeof(int));
  bsp_sync();

  for(i=0;i<n_over_p;i++)
    array[i] += lefts_last;
  bsp_popregister(&last);
}
```

```
void main() {
  int i,j,n_over_p,*xs;
  bsp_begin(bsp_nprocs());
  n_over_p  = 4;
  xs = calloc(n_over_p,sizeof(int));
  for (i=0;i<n_over_p;i++) xs[i]=1;
  bsp_sync();
  bsp_allsums(xs,n_over_p);
  bsp_sync();
  for(i=0;i<bsp_nprocs();i++) {
    if (bsp_pid()==i) {
      printf("On process %d: ",bsp_pid());
      for(j=0;j<n_over_p;j++) printf("%d ",xs[j]);
      printf("\n");
      fflush(stdout);
    }
    bsp_sync();
  }
  bsp_end();
}
```