BSP Parallel Sorting by Regular Sampling – Algorithm and Implementation

Ronald Sujithan[*]

Thu Jul 25 1996

Contents

[*]krs@comlab.ox.ac.uk

1 Introduction

Sorting a collection of elements into ascending or descending order is a fundamental operation that arises as a sub-step in many applications. Because of the fundamental nature of this problem, several sorting methods (see [2, 3] for a comparative surveys) have been devised for a variety of parallel computers.

Essentially there are two approaches to (deterministic) parallel sorting [7]. In the *merge-based* approach, each processor sorts the sublist it holds locally by using some sequential sorting algorithm, and then all the sorted sublists are exchanged among all the processors, and finally the sublists are locally merged. Alternatively, in the *partition-based* (or quicksort based) approach the unsorted list is divided into a number of progressively smaller sublists using the selected pivots, and the sublists sorted sequentially. Furthermore, the merging or partitioning can be achieved in constant steps or recursively. After careful consideration of several sorting methods, we have selected and implemented a practical variant of the deterministic Parallel Sorting by Regular Sampling (PSRS) algorithm.

Our implementation closely follows the presentation of [10], which, in turn, is based on [9]. Furthermore, this algorithm can be considered as the simpler deterministic version of the ones presented in [5, 4], where the theoretical properties (in terms of "one-optimality" etc.) of the algorithm(s) are considered in detail.

Our choice was guided by the considerations of minimising communication costs [1, 6], and attaining load balance [10]. The constant step algorithms move each data element only once, thus minimise communication. On the other hand, recursive algorithms typically incur higher communication cost because elements may be moved several times between the processors. We attain the load balancing property by utilising global information about the data distribution from the regular samples.

2 BSP Algorithm

We introduce the algorithm, provide a cost analysis, and then discuss the load balancing properties. Our implementation,

[Step 1]

perform a local sort;
each processor selects p samples s which are gathered onto process zero;
(we choose a *regular* sample size of $\frac{n}{p^2}$);

[Step 2]

perform a local sort of p^2 samples;
pick $p-1$ regular pivots k from the sorted p^2 samples;
(pivots are picked at $ip + \frac{p}{2}$, $(1 < i \leq (p-1))$, intervals);
broadcast these pivot values to all the processors from processor zero;

[Step 3]

each processor produces p partitions of its local block using the $p-1$ pivots;
each processor sends it's partition i (marked by pivots k_i and k_{i+1} to processor i);

[Step 4]

perform a local sort to merge all the received partitions;
(produces the desired effect of a global sort).

3 Cost Analysis

Since the complexity of an optimal comparison-based sequential sort is $\textsc{SeqSort}(n) = n \log n$, then the cost of locally sorting p processors chunks of data in step 1 is $\textsc{SeqSort}(\frac{n}{p})$. The cost of sorting p^2 samples is likewise, $p^2 \log p^2$, and the final merging stage has an average cost of $\textsc{SeqSort}(\frac{n}{p})$. Thus, the total computation cost is[1]

$$\begin{aligned}
& \frac{n}{p}\log\frac{n}{p} + p^2\log p^2 + \frac{n}{p}\log\frac{n}{p} \\
= \;& 2\frac{n}{p}\log\frac{n}{p} + p^2\log p^2 \\
= \;& 2\frac{n}{p}\log n - 2\frac{n}{p}\log p + 2p^2\log p \\
= \;& 2\frac{n}{p}\left(\log n - \log p + \frac{p^3}{n}\log p\right)
\end{aligned}$$

Clearly, if $n \geq p^3$, then the computational cost is the optimal $O(\frac{n}{p}\log n)$. The communication cost can be estimated as follows. The cost of collecting the p^2 samples is $g(p^2-1)$, and the cost of broadcasting $(p-1)$ pivots is

[1] We provide a straightforward *average* BSP cost analysis of *our implementation* without resorting to detailed probabilistic analysis, which is beyond the scope of this report. Instead, we refer to the recent results reported in [5, 4, 6] for tighter theoretical bounds.

$g(p-1)$. The upper bound on the redistribution of data can be worked out by considering the data movement for partition i: there are at least

$$X_i \leq \left(ip + \frac{p}{2}\right)\frac{n}{p^2} = \frac{(2i+1)}{2}\frac{n}{p}$$

elements below partition i, and there are at least

$$X_{i+1} > \left(p^2 - \left((i+1)p + \frac{p}{2}\right)\right)\frac{n}{p^2} = n - \frac{(2i+3)}{2}\frac{n}{p}$$

elements after partition i. Thus, the amount of data communicated by processor i is at most $n - X_i - X_{i+1} = \frac{n}{p}$ and the BSP cost of redistributing the partitions is $g\frac{n}{p}$. Therefore, the total communication cost is

$$\begin{aligned} & g(p-1)^2 + g(p-1) + g\frac{n}{p} \\ = \;& g\frac{1}{p}\left(p^2(p-1)+n\right) \end{aligned}$$

The communication cost is $g\frac{n}{p}$ for the permute (i.e., at most the entire segment needs to be transferred to a different processor), and therefore the upper-bound BSP cost for our implementation of parallel sort is,

$$\textsc{ParSort}(n) \;=\; 2\frac{n}{p}\left(\log n - \log p + \frac{p^3}{n}\log p\right) + g\frac{1}{p}\left(p^2(p-1)+n\right) + 4l$$

This sorting procedure and the associated cost $\textsc{ParSort}(n)$ for n elements, forms the basis of many BSP collection algorithms we develop in the following sections.

4 Load Balancing

One of the advatages of our sorting algorithm is the load balancing property. The problem to be addressed with the block distribution scheme is to determine when the blocks are not load-balanced, that is, the distribution deviates significantly from $\frac{n}{p}$ from processor to processor.

To show a bound on load balancing, we make an observation that the upper-bound obtained for the redistribution of partitions, $\frac{n}{p}$, is independent

of the initial distribution. Thus during the final step, each processor will contain at most $\frac{n}{p}$ elements, and receive at most $\frac{n}{p}$ elements from other processors. Thus, no processor will contain more that $2\frac{n}{p}$ elements, after the final sort. Experimental results have shown that this bound is within a small percent of optimality in practice, instead of the factor of two derived here [8].

Clearly there is a trade-off between step 2 and step 4. Instead of the p regular samples, if oversampling is used ($s > p$) then an improvement on the load balance can be attained during step 4, at the expense of step 2. On the other hand, undersampling ($s < p$) results in significantly poor load balance in step 4, but decreases the size of step 2. Further work is required to study the oversampling approach to attain tighter load balance bounds [7].

5 Implementation

```
#include "bsp.h"
#include "bsp_level1.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <limits.h>

void bsp_samplesort(int*,int, int **,int *);
void quicksort(int*,int);

char usage_str[] = "main: -v -p <number> -dist <number> -size <number>";
int verbose=0, procs=4, sort_size = 20, sort_dist = 10;
```

The procedure `bsp_samplesort` is a collective communication operation that is to be called by all processes. It takes four arguments:

1. `block` is the array of integers local to each process to be globally sorted.
2. `blocksize` is the number of integers in the local block.
3. `res_block` is a pointer to an array of integers which will be updated with a freshly created array that contains the sorted data
4. `res_blocksize` is a pointer to an integer that will contain the size of the result local to each process.

```
void bsp_samplesort(int *block, int blocksize,
                   int **res_block, int *res_blocksize) {
   int nprocs, mypid,offset,i,j,k,l,NKEYS,
       *Samples,*Pivots,
       sample_stride, pivot_stride;

   int left,right,*chunks_size, **chunks;
   int oldtag_size, *temptag,*temppayload;
   int *mergeleft,   mergeleft_size,
       *mergeright,  mergeright_size,
       *mergeres, *mergetmp;

   oldtag_size=sizeof(int);
   bsp_set_tag_size(&oldtag_size);

   nprocs     = bsp_nprocs();
   mypid      = bsp_pid();
```

Allocate and register temporary data structures.

```
   Samples        = calloc(nprocs*nprocs,sizeof(int));
   Pivots         = calloc(nprocs,sizeof(int));
   if (Samples==NULL || Pivots==NULL)
     bsp_abort("{bsp_samplesort} unable to allocate storage");

   bsp_pushregister(block,blocksize*sizeof(int));
   bsp_pushregister(Samples,nprocs*nprocs*sizeof(int));
   bsp_pushregister(Pivots,(nprocs-1)*sizeof(int));
   bsp_sync();
```

Step 0: Calculate the number of samples from each processor

```
   for(i=0;i<nprocs;i++)
     bsp_hpput(i,&blocksize,Samples,mypid*sizeof(int),sizeof(int));
   bsp_sync();

   for((i=0,NKEYS=0);i<nprocs;i++) NKEYS+=Samples[i];
   for(i=0;i<nprocs;i++)
     Samples[i] = (int) (( ((float) (Samples[i]*nprocs*nprocs)) /
                           ((float) NKEYS)) - 0.5);

```



```
   for((i=0,j=0);i<nprocs;i++) j+= Samples[i];
   for(i=0;
       (i<nprocs) && (j<(nprocs*nprocs));
       (i++,j++)) Samples[i]++;

#ifdef DEBUG
   for(i=0;i<bsp_nprocs();i++) {
     if (bsp_pid()==i) {
       printf("\nStep 0 [pid %d]: Sample sizes:",bsp_pid());
       for(j=0;j<nprocs;j++)
         printf("%d ",Samples[j]);
       printf("\n");
     }
     fflush(stdout);
     bsp_sync();
   }
#endif
```

```
   sample_stride = (int) ((float)NKEYS / (float)(nprocs*nprocs));
   pivot_stride  = nprocs/2;
```

Step 1: locally sort each block of data

```
   quicksort(block, blocksize);
#ifdef DEBUG
   for(i=0;i<nprocs;i++) {
     if (mypid==i) {
       printf("\nStep 1 [pid %d]: after local sort:",mypid);
       for(j=0;j<blocksize;j++)
         printf("%d ",block[j]);
       printf("\n");
     }
     fflush(stdout);
     bsp_sync();
   }
#endif
```

Step 2: pick regular samples and send to proc 0

```
   for((offset=0,i=0);i<mypid;i++) offset+=Samples[i];

   /* Calculate number of samples from me */
   k = Samples[mypid];
```

```
  for ((i=0,j=0); j<k; (i+=sample_stride,j++)) {

    bsp_put(0,
            (void*)&block[i],
            (void*)Samples,
            (offset+j)*sizeof(int),
            sizeof(int));
  }
  bsp_sync();
#ifdef DEBUG
  if (bsp_pid()==0) {
    printf("\nStep 2 [pid %d]: Regular samples:",bsp_pid());
    for(i=0;i<nprocs*nprocs;i++) printf("%d ",Samples[i]);
    printf("\n");
  }
#endif
```

Step 3: sort gathered samples in processor 0

```
  if (bsp_pid() == 0) {
    quicksort(Samples, nprocs*nprocs);

#ifdef DEBUG
    printf("\nStep 3 [pid %d]: Sorted samples:",bsp_pid());
    for(i=0;i<nprocs*nprocs;i++) printf("%d ",Samples[i]);
    printf("\n");
#endif
  }
```

Step 4: broadcast pivots

```
  if (mypid==0) {
    for (i=0; i<nprocs; i++) {
      for(j=1;j<=(nprocs-1);j++) {
        bsp_put(i,
                (void*)&Samples[(j*nprocs + pivot_stride)-1],
                (void*)Pivots,
                (j-1)*sizeof(int),
                sizeof(int));
        }
     }
  }
```



```
  bsp_sync();
#ifdef DEBUG
  for(i=0;i<bsp_nprocs();i++) {
    if (bsp_pid()==i) {
      printf("\nStep 4 [pid %d]: pivots:",bsp_pid());
      for(j=0;j<(nprocs-1);j++)
        printf("%d ",Pivots[j]);
      printf("\n");
    }
    fflush(stdout);
    bsp_sync();
  }
#endif
```

Step 5: communicate each partition

```
  left = 0;
  for(right=0;right<blocksize;right++)
    if (block[right]>=Pivots[0]) break;

  if (right>left)
    bsp_send(0,&mypid,block,(right-left)*sizeof(int));

  left=right;
  for(i=1;i<nprocs-1;i++) {
    while ((block[right]>=Pivots[i-1]) &&
           (block[right]<Pivots[i])) right++;
    if (right>left)
      bsp_send(i,&mypid,&block[left],(right-left)*sizeof(int));
    left=right;
  }

  if (left<blocksize)
    bsp_send(nprocs-1,&mypid,&block[left],(blocksize-left)*sizeof(int));
  bsp_sync();
```

Step 5: Allocate memory at each node for the result

```
  chunks_size = calloc(nprocs,sizeof(int));
  chunks      = calloc(nprocs,sizeof(int*));
  if (chunks_size==NULL || chunks==NULL)
    bsp_abort("Bang");

```

```
  for(i=0;i<nprocs;i++) chunks_size[i]=0;
  *res_blocksize=0;
  for(i=0;i<nprocs;i++) {
    j = bsp_hpmove((void**) &temptag,(void**) &temppayload);
    if (j==-1) break;
    else {
      chunks_size[*temptag]= j;
      chunks[*temptag]     = temppayload;
      *res_blocksize      += j / sizeof(int);
    }
  }

  if (*res_blocksize>0) {
    mergeleft = calloc(*res_blocksize,sizeof(int));
    mergeres  = calloc(*res_blocksize,sizeof(int));

    if (mergeleft==NULL || mergeres==NULL)
      bsp_abort("{bsp_samplesort} unable to alloc %d elements for result",
                 *res_blocksize);
  }
```

Step 6: Merge each sorted partition in the result

```
  mergeleft_size=0;
  for(i=0;i<nprocs;i++)
    if (chunks_size[i]>0) {
      memcpy( mergeleft,
              chunks[i],
              chunks_size[i]);
      mergeleft_size= chunks_size[i]/sizeof(int);
      break;
  }


  for(i++;i<nprocs;i++)
    if (chunks_size[i]>0) {
      mergeright_size = chunks_size[i]/sizeof(int);
      mergeright      = chunks[i];
      for((j=0,k=0,l=0);
          (j<mergeleft_size) && (k<mergeright_size);
          l++) {
        if (mergeleft[j]<mergeright[k])
          mergeres[l] = mergeleft[j++];
        else
          mergeres[l] = mergeright[k++];
      }
      for(;j<mergeleft_size;j++)  mergeres[l++]=mergeleft[j];
      for(;k<mergeright_size;k++) mergeres[l++]=mergeright[k];

      mergetmp = mergeleft;
      mergeleft = mergeres;
      mergeres  = mergetmp;
      mergeleft_size += mergeright_size;
    }
  if (*res_blocksize != mergeleft_size)
    bsp_abort("{bsp_samplesort} internal error with merging");
```

Step 7: Free temprary arrays and return result

```
  bsp_popregister(Pivots);
  bsp_popregister(Samples);
  bsp_popregister(block);
  free(mergeres);
  free(chunks);
  free(chunks_size);
  free(Pivots);
  free(Samples);
  bsp_set_tag_size(&oldtag_size);

  *res_block = mergeleft;
}
```

5.1 Local Quick sort

```
void qs(int *array, int left,int right) {
  register int i,j;
  int        x,y;

  i = left; j = right;
  x = array[(left+right)/2];

  do {
     while(array[i]<x && i<right) i++;
```

```
     while(x<array[j] && j>left)  j--;

     if (i<=j) {
        y = array[i];
        array[i] = array[j];
        array[j] = y;
        i++; j--;
     }
  } while(i<=j);

  if (left<j)  qs(array, left, j);
  if (i<right) qs(array, i, right);
 }

void quicksort(int *array, int size) {
  if (size>1) qs(array,0,size-1);
 }
```

5.2 Start of the parallel program

```
static int integer_compare(int *i, int *j)  {
   return(*i - *j);  /* return a +, - or 0 value */
 }

void spmd_start() {
  int i,j,chunk_size, new_chunk_size,*array, *result;

  bsp_begin(procs);
    srand(bsp_pid());
    if (bsp_pid()==bsp_nprocs()-1)
      printf("I am the last process (%d) out of %d\n",
             bsp_pid(),bsp_nprocs());
    bsp_pushregister(&sort_size,sizeof(int));
    bsp_pushregister(&verbose,sizeof(int));
    bsp_sync();
    bsp_get(0,&verbose,0,&verbose,sizeof(int));
    bsp_get(0,&sort_size,0,&sort_size,sizeof(int));
    bsp_sync();
```

Perform sequential sort on process zero.

```
    if (bsp_pid() == 0) {
      array = calloc(sort_size,sizeof(int));
```



```
      if (array==NULL)
         bsp_abort("Unable to allocate %d elements",sort_size);

      for(i=0;i<sort_size;i++) array[i]=rand() % sort_dist;
      bsp_time();
      qsort(array,sort_size,sizeof(int),
            (int (*)(const void *, const void *)) integer_compare);
      printf("Unix qsort     : %f secs to sort %d elements on 1 proc\n",
             bsp_dtime(),sort_size);
      for(i=0;i<sort_size;i++) array[i]=rand() % sort_dist;
      bsp_time();
      quicksort(array,sort_size);
      printf("C qsort        : %f secs to sort %d elements on 1 proc\n",
              bsp_dtime(),sort_size);
      free(array);
   }
   bsp_sync();
```

Perform parallel sort on process zero.

```
    chunk_size = sort_size / bsp_nprocs();
    if (bsp_pid()==0) chunk_size += sort_size - (chunk_size*bsp_nprocs());
    array = calloc(chunk_size,sizeof(int));
    if (array==NULL)
      bsp_abort("Unable to allocate %d elements",chunk_size);

    for(i=0;i<chunk_size;i++) array[i]=rand() % sort_dist;

    if (verbose) {
      for(i=0;i<bsp_nprocs();i++) {
        if (bsp_pid()==i) {
          printf("Before sort on %d array segment of %d elems: ",i,chunk_size);
          for(j=0;j<chunk_size;j++)
            printf("%d, ",array[j]);
          printf("\n");
          fflush(stdout);
        }
        bsp_sync();
      }
    }
    bsp_time();
    bsp_samplesort(array,chunk_size,&result,&new_chunk_size);
```

```
    if (bsp_pid()==0)
      printf("BSP sample sort: %f secs to sort %d elements on %d procs\n",
             bsp_dtime(),sort_size,bsp_nprocs());
    if (verbose) {
      for(i=0;i<bsp_nprocs();i++) {
        if (bsp_pid()==i) {
          printf("After sort on %d array segment of %d elems: ",
                 i,new_chunk_size);
          for(j=0;j<new_chunk_size;j++)
            printf("%d, ",result[j]);
          printf("\n");
          fflush(stdout);
        }
        bsp_sync();
      }
    }

  bsp_end();
}
```

5.3 Main program

```
void main(int argc,char **argv) {
  int i,temp, command_line_opts=1;
  double trash;

  bsp_init(spmd_start,argc,argv);
  i=1;
  while (command_line_opts) {
    if (i >= argc) {
      command_line_opts = 0;
    } else if ( (strcmp(argv[i],"-size")==0) &&
                (sscanf(argv[i+1],"%d",&temp))) {
      sort_size = temp;
      i += 2;
    } else if ( (strcmp(argv[i],"-dist")==0) &&
                (sscanf(argv[i+1],"%d",&temp))) {
      sort_dist = temp;
      i += 2;
    } else if ( (strcmp(argv[i],"-p")==0) &&
                (sscanf(argv[i+1],"%d",&temp))) {
      procs = temp;
      i += 2;
    } else if (strcmp(argv[i],"-v")==0) {
      verbose =1;
      i += 1;
    } else {
      bsp_abort("{%s}: unknown option \"%s\"\n\t%s\n",
                argv[0],argv[i],usage_str);
    }
  }


  spmd_start();
  exit(0);
}
```


References

[1] M. Adler, J. W. Byers, and R. M. Karp. Scheduling Parallel Communication: The h-relation Problem. Technical Report TR-95-032, International Computer Science Institute, Berkeley, CA, July 1995.

[2] G. E. Blelloch, C. Leiserson, C. Maggs, B.and Plaxton, S. Smith, and M. Zagha. A Comparison of Sorting Algorithms for the Connection Machine CM-2. In *Proceeding of the Third ACM Symposium on Parallel Algorithms and Architectures*, July 1991.

[3] D. E. Culler, A. Dusseau, R. Martin, and K. E. Schauser. Fast Parallel Sorting under LogP: from Theory to Practice. In *Proceedings of the Workshop on Portability and Performance for Parallel Processing*. John Wiley & Sons, Ltd., July 1993.

[4] A. V. Gerbessiotis and S. Siniolakis. Deterministic Sorting and Randomized Median Finding on the BSP Model. In *8th ACM Symposium on Parallel Algorithms and Architectures*, 1996.

[5] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. Technical Report TR-10-92, Aiken Computation Laboratory, Harvard University, 1992.

[6] M. T. Goodrich. Communication-Efficient Parallel Sorting (Preliminary Version). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, pages 247–256, 22–24 May 1996.

[7] H. Li and K. C. Sevcik. Parallel Sorting by Overpartitioning. In *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 46–56, Cape May, New Jersey, June 27–29, 1994. SIGACT and SIGARCH.

[8] X. Li, P. Lu, J. Schaeffer, J. Shillington, P. S. Wong, and H. Shi. On the Versatility of Parallel Sorting by Regular Sampling. *Parallel Computing*, 19:1079–1103, October 1993.

[9] J. H. Reif and L. G. Valiant. Logarithmic Time Sort for Linear Size Networks. *Journal of the ACM*, 34(1):60–76, 1987.

[10] H. Shi and J. Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14(4):361–372, 1992.