RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

ONERA
THE FRENCH AEROSPACE LAB

Parallel sorting algorithms

Theory and implementation

Xavier JUVIGNY, SN2A, DAAA, ONERA
xavier.juvigny@onera.fr

Course Parallel Programming
- January 8th 2023 -

[1] ONERA, [2] DAAA

Table of contents

Overview

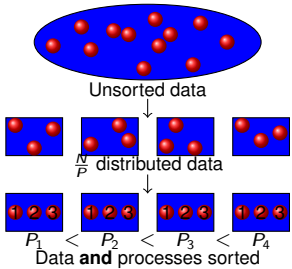
Complexity of sorting algorithms

Basic operations

- **Compare algorithm** : Comparison algorithm complexity is supposed $\mathcal{O}(1)$. But in distributed parallel context, one must consider the distribution of the initial data to account for the cost of data exchange between processes !
- **Exchange algorithm** : Exchange algorithm complexity is supposed $\mathcal{O}(1)$. But same consideration to do as **compare algorithm** ;
- Sequential “compare–and–exchange” algorithm :

```
if (a>b) { // Comparison
  // Exchange
  tmp = a;
  a   = b;
  b   = tmp;   }
```

Potential speed-up



- Best sequential sorting algorithms (for arbitrary sequences of numbers) have average time complexity $O(n \log n)$
- hence, the best speedup one can expect from using n processors is $\frac{O(n \log n)}{n} = O(\log n)$
- there are such parallel algorithms, but the hidden constant is very large (F. Thomson Leighton : Introduction to parallel algorithms and architectures (1991))
- In general, a practical useful $O(\log n)$ algorithm may be difficult to find.

Beware, it may be a bad idea to take n processes to sort n data (granularity).

Parallelization of a naive algorithm

Naive algorithm

- Count the number of numbers that are smaller than a number *a* in the list
- this gives the position of *a* in the sorted list
- this procedure has to be repeated for all elements of the list ; hence the time complexity is $n(n-1) = O(n^2)$ (not so good sequential algorithm)

Implementation

```
for ( i = 0; i < n; i++ ) {// For each value
  x = 0;
  for ( j = 0; j < n; j++ )// Computing the new pos.
    if (a[i] > a[j]) x++;
  b[x] = a[i];
}
```

Works well if there are no repetitions of the numbers in the list (in case of repetitions the code must be changed slightly).

Rank sort : Parallel code

Embarrassingly “ideal” algorithm

Parallel code, using n processes (for n values to sort)

```
x = 0;
for ( j = 0; j < n; j++ )
  if ( a[rank] > a[j] ) x++;
b[x] = a[rank];
```

Complexity

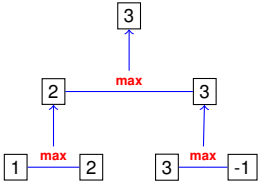
- n processors work in parallel to find the ranks of all numbers of the list ;
- Parallel time complexity is $O(n)$, better than any sequential sorting algorithm !
- Usable for GPGPU units.

More parallelization...

Parallel code using n^2 processes (for n values to sort)

Parallel algorithm

- In the case n^2 processes may be used, the comparison of each `a[0],...,a[n-1]` with `a[i]` may be done in parallel as well
- Incrementing the counter is still sequential, hence the overall computation requires $1 + n$ steps;
- If a tree structure is used to increment the counter, then the overall computation time is $O(\log_2 n)$



(but, as one expects, processor efficiency is very low)

These are just theoretical results : it is not efficient to use n or n^2 processors to sort n numbers.

Data partitioning

Context

- Usually the number n of values is much larger than the number p of processes ;
- In such cases, each process will handle a part of the data (a sublist of the data)

Distributed sorted container

- local container is sorted ;
- if $p_i < p_j$ then $\forall a_i \in p_i, \forall a_j \in p_j, a_i \leq a_j$

Global scheme of parallel sort algorithm

For a process :

- Sort his local data ;
- Run a merge sort algorithm to concatenate its list with that received from another process ;
- Keep the bottom half (or the top half) of the sorted list.

Parallel compare and exchange operations

Asymmetric algorithm

- Process p_i sends local value A to process p_j ;
- Process p_j compares value A with some local values B_j ;
- Send the B_j which are larger (or lesser) than A. If no B_j is larger (or lesser) than A, send back A ;

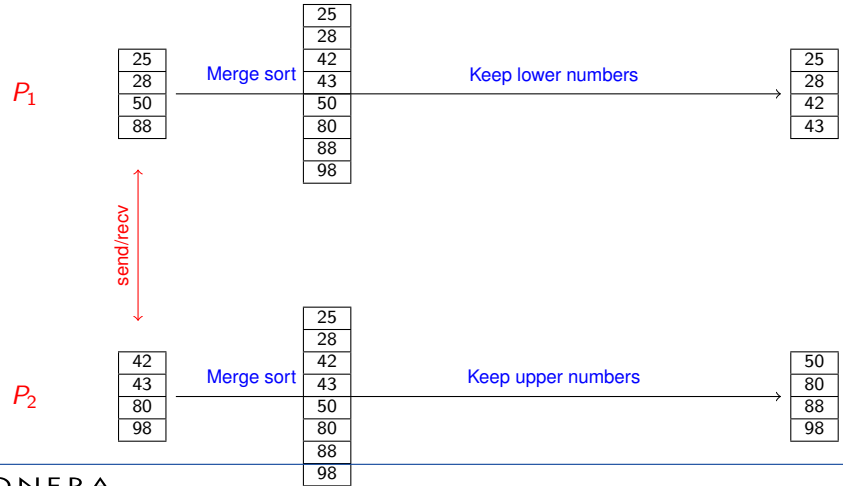
Symmetric algorithm

- Processes p_i and p_j send some value to the other ;
- Each process compares his value with the received value ;
- Each process keeps his value or the received value relative to the comparison result ;

Remarks

- Data exchanges between processes is very expensive, so find some algorithms which minimize data exchanges ;
- In general, the receive operation doesn't know the number of values to receive $\Rightarrow$ one must probe the received message to get the number of data to receive, allocate the relative buffer and receive the data !

Scheme of a general algorithm for parallel sort algorithm



Overview

Sequential bubble sort algorithm

Bubble sort algorithm

- Simplest, but not an efficient sequential sorting algorithm ;
- Compare/exchange complexity :

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

FIGURE – Analogic bubble sort

Sequential code

```
for (int i=n-1; i>0; --i)
    for (int j=0; j<i; ++j) {
      k = j+1;
      if (a[j]>a[k]) std::swap(a[j],a[k]);
    }
```

Odd-Even sort algorithm

- Parallelized bubble sort
- Based on idea that the bodies of the main loop may be overlapped

“scalar” Algorithm : Iteration between even and odd phase

- Even phase


```
if (rank%2==0) {
  recv(&temp, (rank+1)%nbp);
  send(&value,(rank+1)%nbp);
  if (temp < A) A = temp;   }
```

```
if (rank%2==1) {
  send(&value,rank-1);
  recv(&temp, rank-1);
  if (temp > A) A = temp;   }
```

- Odd phase


```
if (rank%2==0) {
  recv(&temp, (rank+nbp-1)%nbp);
  send(&value,(rank+nbp-1)%nbp);
  if (temp > A) A = temp;   }
```

```
if (rank%2==1) {
  send(&value,(rank+1)%nbp);
  recv(&temp, (rank+1)%nbp);
  if (temp < A) A = temp;   }
```

Example of even–odd parallel bubble sort

Example : Sorting 8 numbers on 8 processes

Step	P_0		P_1		P_2		P_3		P_4		P_5		P_6		P_7
0	4	↔	2		7	↔	8		5	↔	1		3	↔	6
1	2		4	↔	7		8	↔	1		5	↔	3		6
2	2	↔	4		7	↔	1		8	↔	3		5	↔	6
3	2		4	↔	1		7	↔	3		8	↔	5		6
4	2	↔	1		4	↔	3		7	↔	5		8	↔	6
5	1		2	↔	3		4	↔	5		7	↔	6		8
6	1	↔	2		3	↔	4		5	↔	6		7	↔	8
7	1		2	↔	3		4	↔	5		6	↔	7		8

Odd-even parallel algorithm per block

Per block algorithm

- Replace a value per process with a sorted set of values per process
- Use sort-fusion algorithm to exchange values
- Data comparison complexity :
  $\frac{N}{nbp} \log_2 \left(\frac{N}{nbp} \right) + (nbp - 1). \frac{2N}{nbp}$
- Data communication complexity :
  $(nbp - 1). \frac{2N}{nbp}$

Implementation

```
sort(values);// Quick sort of local values
for (iter=0; iter<nbp-1; ++iter) {// Odd-even algorithm
  if (iter is odd) {
    if (rank is even and rank > 0) {
      recv(buffer, rank-1); send(values, rank-1);
      values = fusionSort(buffer, values, keepMax);
    } else if (rank is odd and rank < nbp-1) {
      send(values, rank+1); recv(buffer, rank+1);
      values = fusionSort(buffer, values, keepMin);
    }
  } else if (iter is even) {
    if (rank is even and rank < nbp-1) {
      recv(buffer, rank+1); send(values, rank+1);
      values = fusionSort(buffer, values, keepMin);
    } else if (rank is odd) {
      send(values, rank-1); recv(buffer, rank-1);
      values = fusionSort(buffer, values, keepMax);
    }
  }
}
```

Shear sort algorithm

Two dimensional sorting

Basic idea

- Look at the array as a two-dimensional array (one row per process)
- The goal is to sort this 2D array in snakelike style : even rows increasing, odd rows decreasing ;
- Two phases : In **even phase**, sort per row, in **odd phase**, sort per column increasing from top to bottom ;
- After $\log_2(N) + 1$ phases, the array is snakelike-style sorted.

Example

4	14	8	2
10	3	13	16
7	15	1	5
12	6	11	9

Original number

Remarks

- Embarrassingly parallel algorithm for shared memory !
- But not well adapted for distributed parallel architecture as is ;
- How to change this algorithm for distributed parallel architecture ?

Shear sort algorithm

Two dimensional sorting

Basic idea

- Look at the array as a two-dimensional array (one row per process)
- The goal is to sort this 2D array in snakelike style : even rows increasing, odd rows decreasing ;
- Two phases : In **even phase**, sort per row, in **odd phase**, sort per column increasing from top to bottom ;
- After $\log_2(N) + 1$ phases, the array is snakelike-style sorted.

Example

2	4	8	14
16	13	10	3
1	5	7	15
12	11	9	6

Phase 1 – row sort

Remarks

- Embarrassingly parallel algorithm for shared memory !
- But not well adapted for distributed parallel architecture as is ;
- How to change this algorithm for distributed parallel architecture ?

Shear sort algorithm

Two dimensional sorting

Basic idea

- Look at the array as a two-dimensional array (one row per process)
- The goal is to sort this 2D array in snakelike style : even rows increasing, odd rows decreasing ;
- Two phases : In **even phase**, sort per row, in **odd phase**, sort per column increasing from top to bottom ;
- After $\log_2(N) + 1$ phases, the array is snakelike-style sorted.

Example

1	4	7	3
2	5	8	6
12	11	9	14
16	13	10	15

Phase 2 – col sort

Remarks

- Embarrassingly parallel algorithm for shared memory !
- But not well adapted for distributed parallel architecture as is ;
- How to change this algorithm for distributed parallel architecture ?

Shear sort algorithm

Two dimensional sorting

Basic idea

- Look at the array as a two-dimensional array (one row per process)
- The goal is to sort this 2D array in snakelike style : even rows increasing, odd rows decreasing ;
- Two phases : In **even phase**, sort per row, in **odd phase**, sort per column increasing from top to bottom ;
- After $\log_2(N) + 1$ phases, the array is snakelike-style sorted.

Example

1	3	4	7
8	6	5	2
9	11	12	14
16	15	13	10

Phase 3 : row sort

Remarks

- Embarrassingly parallel algorithm for shared memory !
- But not well adapted for distributed parallel architecture as is ;
- How to change this algorithm for distributed parallel architecture ?

Shear sort algorithm

Two dimensional sorting

Basic idea

- Look at the array as a two-dimensional array (one row per process)
- The goal is to sort this 2D array in snakelike style : even rows increasing, odd rows decreasing ;
- Two phases : In **even phase**, sort per row, in **odd phase**, sort per column increasing from top to bottom ;
- After $\log_2(N) + 1$ phases, the array is snakelike-style sorted.

Example

1	3	4	2
8	6	5	7
9	11	12	10
16	15	13	14

Phase 4 : col sort

Remarks

- Embarrassingly parallel algorithm for shared memory !
- But not well adapted for distributed parallel architecture as is ;
- How to change this algorithm for distributed parallel architecture ?

Shear sort algorithm

Two dimensional sorting

Basic idea

- Look at the array as a two-dimensional array (one row per process)
- The goal is to sort this 2D array in snakelike style : even rows increasing, odd rows decreasing ;
- Two phases : In **even phase**, sort per row, in **odd phase**, sort per column increasing from top to bottom ;
- After $\log_2(N) + 1$ phases, the array is snakelike-style sorted.

Example

1	2	3	4
8	7	6	5
9	10	11	12
16	15	14	13

Final 5 : row sort

Remarks

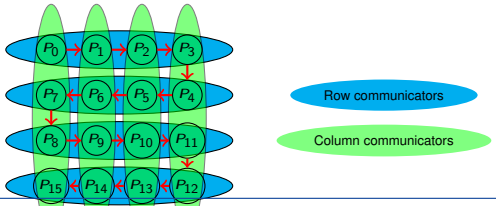
- Embarrassingly parallel algorithm for shared memory !
- But not well adapted for distributed parallel architecture as is ;
- How to change this algorithm for distributed parallel architecture ?

Shear sort algorithm for parallel distributed memory architecture

Implementation ideas

- Same principal as odd-even algorithm : replace a value with some sets of values S_i (one set per process) ;
- Define a relation order : $S_i < S_j$ iff $\max(S_i) < \min(S_j)$ (In set, values ordered as increasing order)
- Use odd-even algorithm to parallelize the phase of sorting per row or column ;
- Grouping processes in new communicators per rows and per columns ;
- Play with rank numbering to alternate between increasing order and decreasing order for rows ;

Processes repartition



- Use `MPI_Comm_split(comm,color,key,&newcomm)` to define row and columns communicators ;
- Processes calling this function with same color are inside the same new communicator ;
- key is a value to number the processes inside the new communicator.

Overview

Sequential quick-sort algorithm

Reminder

- Optimal sequential sorting algorithm ($\mathcal{O}(n \log_2(n))$) based on divide-and-conquer algorithm class ;
- Select a number r called pivot and split the list into two sublists : one with all elements at most equal to r, the other holding all elements greater than r ;
- This procedure is recursive, applied until one element lists are obtained (which are sorted)
- **Example :**

4 2 7 8 5 1 3 6

Sequential code

```
void quicksort( T* list, T* start, T* end ) {
    auto pivot = choosePivot(start,end);
    if (start < end) {
        split(list, start, end, pivot);
        quicksort(list, start, pivot-1);
        quicksort(list,pivot+1, end   );
    }
}
```

Sequential quick-sort algorithm

Reminder

- Optimal sequential sorting algorithm ($\mathcal{O}(n \log_2(n))$) based on divide-and-conquer algorithm class ;
- Select a number r called pivot and split the list into two sublists : one with all elements at most equal to r, the other holding all elements greater than r ;
- This procedure is recursive, applied until one element lists are obtained (which are sorted)
- **Example :**

4 2 7 8 5 1 3 6

Sequential code

```
void quicksort( T* list, T* start, T* end ) {
    auto pivot = choosePivot(start,end);
    if (start < end) {
        split(list, start, end, pivot);
        quicksort(list, start, pivot-1);
        quicksort(list,pivot+1, end   );
    }
}
```

Sequential quick-sort algorithm

Reminder

- Optimal sequential sorting algorithm ($\mathcal{O}(n \log_2(n))$) based on divide-and-conquer algorithm class ;
- Select a number r called pivot and split the list into two sublists : one with all elements at most equal to r, the other holding all elements greater than r ;
- This procedure is recursive, applied until one element lists are obtained (which are sorted)
- **Example :**

4 2 5 1 3 6 7 8

Sequential code

```
void quicksort( T* list, T* start, T* end ) {
    auto pivot = choosePivot(start,end);
    if (start < end) {
        split(list, start, end, pivot);
        quicksort(list, start, pivot-1);
        quicksort(list,pivot+1, end   );
    }
}
```

Sequential quick-sort algorithm

Reminder

- Optimal sequential sorting algorithm ($\mathcal{O}(n \log_2(n))$) based on divide-and-conquer algorithm class ;
- Select a number r called pivot and split the list into two sublists : one with all elements at most equal to r, the other holding all elements greater than r ;
- This procedure is recursive, applied until one element lists are obtained (which are sorted)
- **Example :**

4 2 5 1 3 6 7 8

Sequential code

```
void quicksort( T* list, T* start, T* end ) {
    auto pivot = choosePivot(start,end);
    if (start < end) {
        split(list, start, end, pivot);
        quicksort(list, start, pivot-1);
        quicksort(list,pivot+1, end   );
    }
}
```

Sequential quick-sort algorithm

Reminder

- Optimal sequential sorting algorithm ($\mathcal{O}(n \log_2(n))$) based on divide-and-conquer algorithm class ;
- Select a number r called pivot and split the list into two sublists : one with all elements at most equal to r, the other holding all elements greater than r ;
- This procedure is recursive, applied until one element lists are obtained (which are sorted)
- **Example :**

2 1 3 4 5 6 7 8

Sequential code

```
void quicksort( T* list, T* start, T* end ) {
    auto pivot = choosePivot(start,end);
    if (start < end) {
        split(list, start, end, pivot);
        quicksort(list, start, pivot-1);
        quicksort(list,pivot+1, end   );
    }
}
```

Sequential quick-sort algorithm

Reminder

- Optimal sequential sorting algorithm ($\mathcal{O}(n \log_2(n))$) based on divide-and-conquer algorithm class ;
- Select a number r called pivot and split the list into two sublists : one with all elements at most equal to r, the other holding all elements greater than r ;
- This procedure is recursive, applied until one element lists are obtained (which are sorted)
- **Example :**

2 1 3 4 5 6 7 8

Sequential code

```
void quicksort( T* list, T* start, T* end ) {
    auto pivot = choosePivot(start,end);
    if (start < end) {
        split(list, start, end, pivot);
        quicksort(list, start, pivot-1);
        quicksort(list,pivot+1, end   );
    }
}
```

Sequential quick-sort algorithm

Reminder

- Optimal sequential sorting algorithm ($\mathcal{O}(n \log_2(n))$) based on divide-and-conquer algorithm class ;
- Select a number r called pivot and split the list into two sublists : one with all elements at most equal to r, the other holding all elements greater than r ;
- This procedure is recursive, applied until one element lists are obtained (which are sorted)
- **Example :**

1 2 3 4 5 6 7 8

Sequential code

```
void quicksort( T* list, T* start, T* end ) {
    auto pivot = choosePivot(start,end);
    if (start < end) {
        split(list, start, end, pivot);
        quicksort(list, start, pivot-1);
        quicksort(list,pivot+1, end   );
    }
}
```

Naive parallelization of quicksort algorithm

Ideas

- The class of quick-sort algorithm suggests to apply a divide-and-conquer parallelization method ;
- The main problem of this approach is that the tree distribution (induced by the lengths of the sublists) heavily depends on pivot selection ; in the worst case, the tree may consists of a single path (as a list) ;
- **Analysis** : provided an equal distribution of values within sublists is ensured, one gets :
  - Comparisons : $n + \frac{n}{2} + \frac{n}{4} + \cdots \approx 2n$
  - Communications : $t_s + \frac{n}{2} t_d + t_s + \frac{n}{4} t_d + \cdots \approx \log_2(n) t_s + n t_d$ where t_s is time to start a communication and t_d the time to transfer one element to another process.
- But only last iteration uses full parallelization !

Hyperquick sort algorithm

Binary numbering of Hypercube

FIGURE – Dimension 0

Hyperquick sort algorithm

Binary numbering of Hypercube

FIGURE – Dimension 1

Hyperquick sort algorithm

Binary numbering of Hypercube

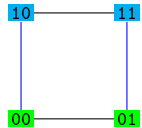


FIGURE – Dimension 2

Hyperquick sort algorithm

Binary numbering of Hypercube

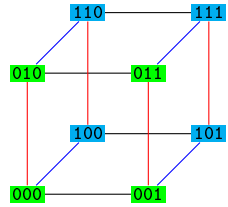


FIGURE – Dimension 3

Hyperquick sort algorithm

Binary numbering of Hypercube

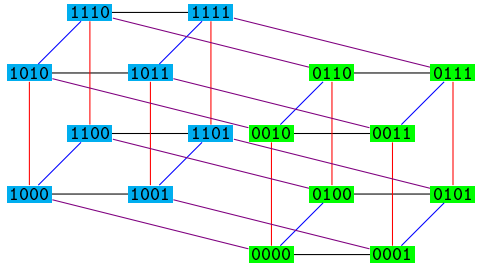


FIGURE – Dimension 4

Hyper quick sort algorithm (2)

Numbering vertices of hyper-cube

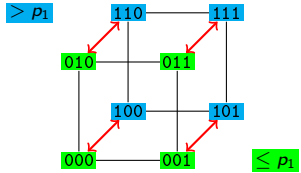
- The binary numbers of two linked nodes have a difference of one bit ;
- The distance between two nodes in a hypercube (minimal number of nodes to access to go from first node to second node) is the number of bit which differ in their binary number ;
- It's the **Gray code** numbering.

Ideas of the hyper quick sort algorithm

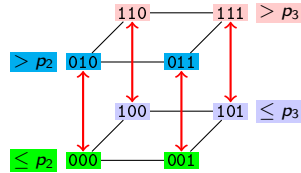
- Initially, Data are distributed across all processes ;
- Each process sorts its local data ;
- Loop on dimension of the hypercube and for dimension d, consider pair of processes $\left(p; p + 2^d\right)$
- Process p chooses his median value as pivot and sends value lesser than pivot to process $p + 2^d$;
- Process $p + 2^d$ receives pivot and data from process p and keeps value greater than pivot ;
- Each process keeps sorted value, using fusion sort algorithm to keep sorting.

Illustration of hyper quick sort

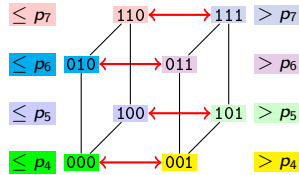
Phase 1 :



Phase 2 :



Phase 3 :



Hyperquicksort : complexity analysis

- Suppose we run algorithm on $nbp = 2^d$ processes (hypercube dimension d) ;
- Each process holds initially $Nl = \frac{N}{nbp}$ values ;
- **Initial sorting :** $Nl.\log_2(Nl)$ comparisons ;
- **Pivot selection :** $\mathcal{O}(1)$ (one takes middle list element) for each dimension ;
- **Pivot broadcasting :**
  - Broadcast one pivot in a k hypercube : $k(t_s + t_d)$;
  - For all iterations : $(d + \cdots + 1)(t_s + t_d) = \frac{d(d-1)}{2}(t_s + t_d)$ comparisons ;
- **Split list from pivot value :** For sorted list of size x, $\log_2(x)$ comparisons ;
- **Exchange part of list :** To exchange $\frac{x}{2}$ data : $2(t_s + \frac{x}{2}t_d)$
- **Fusion merge sort :** $\frac{x}{2}$ comparisons

Total (for ideal balance)

- $Nl.\log_2(Nl) + \left(\log_2(Nl) + \frac{Nl}{2}\right).d$ comparisons
- $\left(\frac{d(d-1)}{2} + 2d\right)t_s + \left(\frac{d(d-1)}{2} + d.Nl\right).t_d$ for message ;

Overview

Bitonic sequences

Definition of a bitonic sequence

- A sequence of values $\{a_i\}_{i\in[1;N]}$ than can be split in two subsequences (of consecutive numbers), one increasing and one decreasing ; e.g :

$$\exists i \in [1; N] \text{ verifying } \begin{cases} a_1 \leq a_2 \leq \cdots \leq a_i \\ a_i \geq a_{i+1} \geq \cdots \geq a_N \end{cases}$$

  Example : 3, 5, 8, **19**, 17, 14, 12, 11

- **Or** a sequence which may be brought to this form by a circular shifting of the elements of the sequence

  Example : 12, 11, 3, 5, 8, **19**, 17, 14

Remarks

- The first subsequence can be increasing **or** decreasing.
- So a bitonic sequence (without considering circular shifting) can be increasing-decreasing or decreasing-increasing.
- A monotone bitonic sequence is a sorted sequence (increasing or decreasing) ;
- All sequences with three or less elements are bitonic.

Splitting a bitonic sequence

Bitonic split

Let $\{a_i\}_{i\in[1;N]}$ be a bitonic sequence. So the subsequences

$$\begin{cases} \{b_i\}_{i\in\left[1,\frac{N}{2}\right]} & = & \left\{\min(a_1, a_{1+\frac{N}{2}}), \min(a_2, a_{2+\frac{N}{2}}), \cdots, \min(a_{\frac{N}{2}-1}, a_N)\right\} \\ \{c_i\}_{i\in\left[1,\frac{N}{2}\right]} & = & \left\{\max(a_1, a_{1+\frac{N}{2}}), \max(a_2, a_{2+\frac{N}{2}}), \cdots, \max(a_{\frac{N}{2}-1}, a_N)\right\} \end{cases}$$

- are bitonic sequences ;
- $\forall i \in \left[1; \frac{N}{2}\right]$; $b_i \leq c_i$

Example

1 5 8 7 6 4 3 2 $\overset{\text{split}}{\Longrightarrow}$ 1 4 3 2 6 5 8 7

Sorting a bitonic sequence (SBS algorithm)

Algorithm

Apply split procedure on bitonic sequence and repeat this splitting procedure on subsequences until having one element per subsequence to obtain sorted sequence.

Example

1. 1 5 8 7 6 4 3 2
2. 1 4 3 2 6 5 8 7
3. 1 2 3 4 6 5 8 7
4. 1 2 3 4 5 6 7 8

Sorting a bitonic sequence (SBS algorithm)

Algorithm

Apply split procedure on bitonic sequence and repeat this splitting procedure on subsequences until having one element per subsequence to obtain sorted sequence.

Example

1. 1 5 8 7 6 4 3 2
2. 1 4 3 2 6 5 8 7
3. 1 2 3 4 6 5 8 7
4. 1 2 3 4 5 6 7 8

Sorting a bitonic sequence (SBS algorithm)

Algorithm

Apply split procedure on bitonic sequence and repeat this splitting procedure on subsequences until having one element per subsequence to obtain sorted sequence.

Example

1. 

2. 

3. 

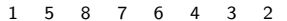
4.

Sorting a bitonic sequence (SBS algorithm)

Algorithm

Apply split procedure on bitonic sequence and repeat this splitting procedure on subsequences until having one element per subsequence to obtain sorted sequence.

Example

1. 

2. 

3. 

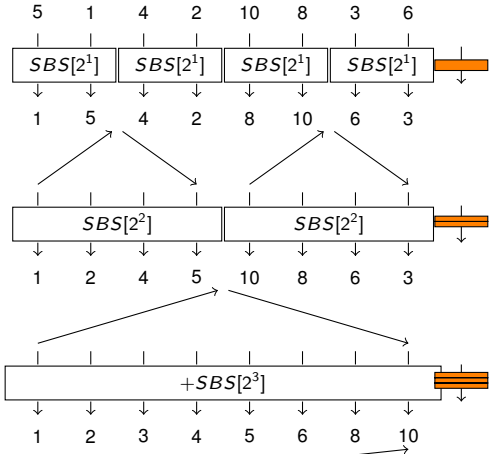
4. 

Building a bitonic sequence

Procedure

1. Split the sequence in two-elements subsequences (which are bitonics !) ;
2. Sort subsequences alternating increasing and decreasing sorting (using SBS algorithm) ;
3. Concatenate two adjacent lists to get a longer bitonic sequence ;
4. Repeat from step 2 until the full list becomes a bitonic sequence.

Example on 8 elements



One step of SBS

Two steps of SBS

Three steps of SBS

Bitonic sort analysis

Complexity of bitonic sort :

- With $n = 2^k$, there are k phases, each involving $1, 2, \ldots, k$ steps, respectively ;
- The total number of steps is

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2} = O(k^2) = O(\log_2^2 n) \tag{1}$$

- Total complexity is $O(n \log_2^2 n)$

Adapt bitonic sort to distributed parallel architecture

Main ideas

- First sort local data with the fastest sort algorithm in increasing values if rank is even and decreasing values if rank is odd ;
- Pairing inside a subcommunicator processes to define a bitonic sequence ;
- And apply bitonic sort algorithm, grouping processes per four, height and so on. . .
- Sub-communicators built here are very similar to the sub-communicators build with hyperquick sort algorithms !

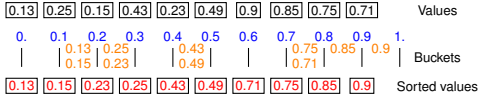
Overview

Bucket Sort algorithm

Algorithm

Distribute elements of a list into a fixed number of buckets, and sort the elements inside each buckets.

- Setup an array of initially empty "buckets"
- **Scatter** : Put each element of the list in the right bucket
- Sort the elements inside each bucket
- **Gather** : Visit each bucket in order and put all elements back in original list



- Complexity at best : $N + \frac{N}{k} \log_2(\frac{N}{k})$ comparisons (k = numbers of buckets)
- Difficulty : How to choose the intervals for the buckets ?

Parallel bucket sort

Main ideas for the parallel algorithm

- Each process is seen as one bucket ;
- Compute intervals for buckets :
  - Sort local data
  - Take $nbp + 1$ values at regular intervals
  - Gather values in bucket array
  - Extract $nbp + 1$ values to find intervals for buckets

Analysis

- Local sort : $\frac{N}{nbp} \log_2(\frac{N}{nbp})$ comparisons ;
- Gather bucket array : $nbp(t_s + (nbp + 1)t_d)$
- Sort bucket array : $\frac{nbp}{\log_2}(nbp)$ comparisons ;
- Distribute data in buckets : $(nbp - 1)\left(t_s + \frac{N}{nbp^2} t_d\right)$
- Sort local data : $2(nbp - 1)\frac{N}{nbp}$ comparisons (fusion sort)