Parallel algorithms

Parallel sort
Divide and Conquer approach - review

- Recursively partition a problem into subprogram of roughly equal size.

- If subprogram can be solved independently, there is a possibility of significant speed up by parallel computing.

\[
\sum_{i=0}^{2^n-1} = \sum_{i=0}^{2^{n-1}-1} + \sum_{i=2^{n-1}}^{2^n-1}
\]

Nb of levels \( n = \log_2 N \)

Operations on the same level can be realized concurrently

Nb of steps \( \sim n = \log_2 N \)
Example - MergeSort

Diagram of merge sort process.
Parallel merge sort – simple approach (tree)

- Memory requirements at root of the tree
- Only tasks at the same level can run concurrently
Parallel mergesort – agglomerated tree

- Still remains memory problem ...
- the sequence of comparisons is data-dependent.
Problem

• Precondition:
  – The order of tasks is defined
  – Each task contains part of the unsorted list

• After some exchange between tasks each task should contain part of sorted list according to tasks’ defined order:
  – The exchange should not force the tasks to allocate too much memory in comparison to the initial list requirements
  – The amount of communication between tasks should be optimized.

• Mergesort is difficult to directly parallelize as the sequence of comparisons in this algorithm is data-dependent
  – Difficult to define a network of exchanges
Modification – Bitonic mergesort

• Based on the features of bitonic sequence (bitonic list)

• Bitonic list:
  – A list with no more than one local maxima and no more than one local minima.
  – Important type of bitonic list:
    • the first half is sorted in ascending (descending) order and the second half in descending (ascending) order
    • example: 2, 4, 6, 8, 10, 9, 7, 5, 3, 1
Bitonic split

- Each element in the first half of the list is assigned a partner, which is the same relative position from the second half of the list.
- Each pair of partners compares themselves. If the first half of the list has a larger (smaller) item then exchange them.
- The results:
  - Each item in the first half of the list is less than every item in the second half.
  - The first half and the second half of the list are each a bitonic list of length N/2.
- the sequence of comparisons is not data-dependent!
Example: bitonic split

Original bitonic list

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Crossover

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Split bitonic list (ascending)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Split bitonic list (descending)

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>45</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recursive bitonic split = bitonic merge

- Input: bitonic list
- Recursive application of bitonic split
  - Not data-dependent sequence of comparisons
- Output: the list is fully sorted after log\(N\) levels!
Bitonic mergesort

- Recursive bitonic split sorts bitonic list
  - How to apply it to unsorted sequence?
- Devide and conquer:
  - Consider an unsorted list with \( N = 2^{\text{dim}} \) items.
  - Any list with only two items is a bitonic list.
  - The unsorted list consists of \( N/2 \) bitonic lists! (of length 2)
  - By applying the bitonic merge to pairs of adjacent lists, the result is \( N/4 \) bitonic lists of length 4.
  - After \( \log N \) repetitions of the bitonic merge, the list is completely sorted.
Bitonic mergesort - analysis

- Each bitonic split requires n/2 comparisons
- Log n levels of recursive bitonic split
- Log n repetitions of bitonic merge
- Nb of comparisons O(n* log (n)^2)
Making it parallel

- As a communication structure we use hypercube
Hypercube topology

• constructed recursively
  – a one-dimensional hypercube has two connected processes 0 and 1.
  – A (d+1)-dimensional hypercube is defined from a d-dimensional hypercube as follows:
    • Duplicate the d-dimensional hypercube including process numbers.
    • Create links between processes with the same number in the original and duplicate.
    • Append a binary 1 to the left of each process number in the duplicate, and a binary 0 to left of each process number in the original.
Hypercube - features

- A d-dimensional hypercube consists of $n = 2^d$ nodes.
- Each node has a number whose binary representation has $d$ digits.
- **Hamming distance**: The total number of bit positions at which two binary numbers differ.
- **In a hypercube, two processes are connected if their Hamming distance is 1.**
  - Binary representations of numbers of connected nodes differ only on one position
  - The connectivity of a d-dimensional hypercube is d.
**XOR and AND Bitwise operations - review**

**XOR**: 1,1=0; 0,0=0; 1,0=1 0,1=1  
**AND**: 1,1=1; 0,0=0; 1,0=0; 0,1=0  

E.g. for m=3  

\[
\begin{align*}
xx1xxx \text{ XOR } 001000 &= xx0xxx \\
xx0xxx \text{ XOR } 001000 &= xx1xxx
\end{align*}
\]

So: \textit{nodeA and nodeB= nodeA XOR } 2^m \textit{ are neighbours in hypecube in dim m}  

\[
\begin{align*}
xx0xxx \text{ AND } 001000 &= 000000 \\
xx1xxx \text{ AND } 001000 &= 001000
\end{align*}
\]

So: \textit{AND with } 2^m \textit{ indicates the value of the bit m-th.}
Making it parallel

- n - the number of elements to be sorted
- \( p = 2^d \) be the number of processes in a \( d \)-dimensional hypercube
- Each process is assigned a block of \( n/p \) elements
- Comparing of single values => comparing of blocks of \( n/p \) elements using simple mergesort
Mergesort between nodes

• The whole merge is done in the local memory of the single process
• The partial lists are already sorted
• Optimisation can be done – e.g merging sorted sequences algorithm can be used.
Making it parallel

for L := 1 to dim do
    for m := L-1 downto 0 do
        begin
            partner := myid XOR 2^m;
            if me AND 2^L = 0 then
                ascending merge sort
                for me and partner
            else
                descending merge sort
                for me and partner;
            if me AND 2^m = 0 then
                retain the first half
                of the merged list
            else
                retain the second half
                of the merged list
        end;
Example

level  |  node ID
-------|---------
 1  m  | 000  001  010  011  100  101  110  1
 1  0  | 3  4  8  7  2  6  9  1  3  5  9  6  2  4  3
 2  1  | 3  4  1  2  6  9  7  8  5  4  9  6  3  3  2
 2  0  | 1  2  3  4  6  7  8  9  9  6  5  4  3  3  2
 3  2  | 1  2  3  4  3  3  2  1  6  9  4  5  6  7  8
 3  1  | 1  2  1  2  3  3  3  4  6  6  4  5  7  9  8
 3  0  | 1  1  2  2  3  3  3  4  4  5  6  6  7  8  9
Hypercube Quicksort
Quicksort - review

- Quicksort is a “divide-and-conquer” method for sorting.
- Complexity O(n*\log n), worst case O(n^2)
- It works by partitioning a file into two parts, then sorting the parts independently.

```c
quicksort(int list[], int left, int right){
    int k;
    if (left < right) {
        k = partition(list, left, right);
        quicksort(list, left, k-1);
        quicksort(list, k+1, right);
    }
}
```
Quicksort - review

• The partition procedure works as follows:
  – Given a sublist list[left:right], it first chooses one of the elements as a pivot
    • e.g a[left] or a[(right+left)/2]
  – the pivot element is placed at the k-th position, and:
    • a[left],...,a[k-1] are less than or equal to a[k];
    • a[k+1],...,a[right] are greater than or equal to a[k].
Quicksort partition - example

<table>
<thead>
<tr>
<th></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>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>j</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>i</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>j</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>i</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>j</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>i</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>8</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>j</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>7</td>
</tr>
</tbody>
</table>
Parallel Quicksort

• $n$ - the number of elements to be sorted
• $p = 2^d$ be the number of processes in a $d$-dimensional hypercube
• Each process is assigned a block of $n/p$ elements.
• The algorithm:
  – Select a common pivot value, which is broadcast to all processes.
  – Each process partitions its local elements into two blocks, one with elements smaller than the pivot, and the other with elements larger than the pivot
Parallel Quicksort

The algorithm (cont’d):

– the processes connected along the d-th communication link exchange blocks:
  • Each process with a 0 in the d-th bit retains the smaller elements,
  • Each process with a 1 in the d-th bit retains the larger elements.
  • After this step:
    – each process in the (d-1)-dimensional hypercube whose d-th bit is 0 has elements smaller than the pivot,
    – each process in the other (d-1)-dimensional hypercube has elements larger than the pivot.
Parallel Quicksort

The algorithm (cont’d):

– At the next level, a pivot is chosen in each (d-1)-dimensional hypercube separately.
– Pivot is broadcast to all the processes in each sub hypercube.
– Each process partitions its local elements into two blocks, one smaller and the other larger than the pivot.
– Appropriate blocks are exchanged through the (d-1)-th communication link:
  • each process with a 0 in the (d-1)-th bit retains the smaller elements than the pivot,
  • each process with a 1 in the (d-1)-th bit retains the larger.
Parallel Quicksort

- This procedure is performed recursively.
- After $d$ such splits, the sequence is sorted with respect to the global ordering imposed on the processes.
- Then each process sorts its local elements by using sequential quicksort.
- Dimension Master of Subcube
  - $3 \ 000$
  - $2 \ x00$
  - $1 \ xx0$
- Partner is obtained by flipping the $d$-th bit
- Master of each subcube determines the pivot value and broadcasts it to all the processes in the subcube.
- Bad choice of pivot at early stages degrades the performance significantly
Parallel Bucket Sort
Bucket sort

- Set up an array of initially empty "buckets."
- Scatter: Go over the original array, putting each object in its bucket.
- Sort each non-empty bucket.
- Gather: Visit the buckets in order and put all elements back into the original array.
Parallel bucket sort

- Calculation of buckets’ borders based on small data sample
- Putting data in the buckets
- Exchanging buckets
  - Each process has its own bucket
- Sorting data in the buckets
- Correction
Calculation of buckets’ borders

- based on small data sample
- each process:
  - chooses small number of data
  - sends it to process 0 that:
    - sorts the sample
    - determine borders
    - send borders back
  - This can be done using e.g. tree communication scheme
Exchanging buckets

• Before: every process has all buckets belonging to others
• After: each process has its own bucket composed from buckets from others

<table>
<thead>
<tr>
<th>Process 1</th>
<th>Process 2</th>
<th>Process 3</th>
<th>Process 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCD</td>
<td>EFGH</td>
<td>IJKL</td>
<td>MNOP</td>
</tr>
<tr>
<td>AEIM</td>
<td>BFJN</td>
<td>CGKO</td>
<td>DHLP</td>
</tr>
</tbody>
</table>
Correction

• The borders were estimated based on the sample.

• Therefore the number of data in each bucket (process) can vary.

• Correction can be done e.g.:
  – Process 0 shifts data to/from process 1
  – Process 1 shifts data to/from process 2
  – ...
Parallel bucket sort

- Can be used by partitioners, where:
  - distribution between processes is important,
  - full sort is often not necessary