Sorting Methods in HPC

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

High Performance Computing I, 2013

Part I

Parallel Sorting

Importance of Sorting

- Dry, but necessary topic
- Sorting fundamental to many applications
- Sorting: placing items in non-decreasing/non-increasing order
- Not necessarily numerical - lexical/binary
- Most successful sequential sorting algorithms use compare and exchange method on pairs of data elements

Potential Parallel Speedup

Ok, sequential sorting algorithms (which we will briefly review) are well studied - **quicksort** and **mergesort** are popular:

- Both are based on compare-and-exchange
- Worst-case mergesort is $\mathcal{O}(N \log N)$
- Average-case quicksort is $\mathcal{O}(N \log N)$
- Argue that best-case is $\mathcal{O}(N \log N)$ without using special properties of data elements
- Optimal parallel is $\mathcal{O}(\log N)$ for $P = N$ processors
  - In practice, difficult to achieve

Compare & Exchange

Compare and exchange:

- Forms basis of most classical sequential sorting algorithms
- Basic step, two elements compared and, if necessary, exchanged:

```
if ( A > B ) then
   temp = A
   A = B
   B = temp
end if
```

- In parallel, easy to implement a similar step using message-passing, assuming processor P_1 has A, P_2 has B:

```
if (myID == 1) then
   SEND( A to P_2 )
   RECV( A from P_2 )
if (myID == 2) then
   RECV( A from P_1 )
   if ( A > B ) then
      SEND( B to P_1 )
      B = A
   else
      SEND( A to P_1 )
end if
```

or in a slightly more symmetric way:

```
if (myID == 1) then
   SEND( A to P_2 )
   RECV( B from P_2 )
   if ( A > B ) A=B
if (myID == 2) then
   RECV( A from P_1 )
   SEND( B to P_1 )
   if ( A > B ) B=A
end if
```

Data Partitioning

Consider data partitioning in the context of the preceding compare-and-exchange scheme:

- Instead of 1 data element per processor ($N = P$) we can easily treat the case of N/P data elements per processor
- Can still apply the first algorithm, in which one processor sends its partition to the other, which performs comparisons, merges results and returns the lower half
- Can also apply the second algorithm, in which both partner processors exchange partitions, one keeps smaller half, the other the larger half

Sequential Bucket Sort

Let's begin with **bucket** sort:

- Similar to quicksort
- Not based on compare-and-exchange
- Based on partitioning, which will be important for parallel sorting
- Optimal in case where data uniformly distributed in some interval
- Parallelization using divide-and-conquer approach

Sequential Bucket Sort Algorithm

Sequential bucket sort (N data items):

- Identify region in which number lies, say if $x \in [0, a]$, $\mathrm{INT}(x/(M/a))$ gives interval ("bucket") in which x lies this could be quite fast if N is a power of 2
- N steps for M buckets
- Sort buckets using quicksort or mergesort
- Time complexity,

$$\tau_S = N + M\left[(N/M)\log(N/M)\right] \simeq \mathcal{O}(N\log(N/M))$$

For $N/M = \mathrm{constant}$, this is $\mathcal{O}(N)$, better than most sequential algorithms, but it requires uniform data distribution.

Parallel Buckets

For a simple parallel bucket sort:

- Assign one processor/bucket, $\mathcal{O}(N/P \log(N/P))$
- Partition data into $M = P$ sets
- Each processor keeps $P = M$ "small" buckets, to be exchanged with other processors
- "small" buckets exchanged to processors
- "large" buckets sorted on each processor (see figure)

Bucket Sort Time Complexity

Time complexity for bucket sort:

- Step 1: communication via broadcast, $\tau_{comm1} \sim \tau_{lat} + N\tau_{data}$
- Step 2: computation,
  $\tau_{comp2} \sim N/P$
- Step 3: communication, assuming each small bucket has N/P^2 numbers,
  $\tau_{comm3} \sim P(P-1)\left[\tau_{lat} + (N/P^2)\tau_{data}\right]$
- Step 4: computation,
  $\tau_{comp4} \sim (N/P)\log(N/P)$

- Full time complexity:

$$\begin{aligned}\tau_P &= \tau_{comm1} + \tau_{comp2} + \tau_{comm3} + \tau_{comp4},\\ &= \frac{N}{P}\left[1 + (N/P)\log(N/P)\right] + P\tau_{lat} + \left[N + (P-1)(N/P^2)\right]\tau_{dat},\end{aligned}$$

Noting that

$$\tau_{comp}/\tau_{comm} = \frac{(N/P)\left[1 + \log(N/P)\right]}{P\tau_{lat} + \left[N + (P-1)(N/P^2)\right]\tau_{data}},$$

the speedup factor is given by

$$\begin{aligned}S = \tau_S/\tau_P &= \frac{N + N\log(N/P)}{(N/P)\left[1 + (N/P)\log(N/P)\right] + P\tau_{lat} + \left[N + (P-1)(N/P^2)\right]\tau_{dat}},\\ &= P\frac{1}{1 + P\tau_{comm}/\tau_{comp}},\end{aligned}$$

a familiar result.

Sequential Bubble Sort

Consider a sequence of data, $\{x_0, x_1, \ldots, x_{N-1}\}$. Bubble sort is a simple algorithm:

- x_0 and x_1 compared, larger shifted to x_1.
- x_1 and x_2 compared, larger shifted to x_2, ...
- ... repeat from previous largest data member
- Total of $N-1$ compare-and-exchange steps, then $N-2$, $N-3$, ...
- Summation:

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

so $\mathcal{O}(N^2)$, not so great (but very simple).

Sequential Bubble Sort Pseudo-code

Pseudo-code for sequential bubble sort:

```
do i=N-1,1,-1
   do j=0,i-1
      k=j+1
      if ( a(j) > a(k) ) then
         temp = a(j)
         a(j) = a(k)
         a(k) = temp
      end if
   end do
end do
```

Odd-Even Transposition Sort

Odd-even transposition sort is a reformulation of bubble for parallel computation. It has (unsurprisingly!) two phases - corresponding to odd-numbered and even-numbered processors:

even phase even-numbered processor i, $i \in [0, 2, 4, ...]$:

```
RECV( A from P_{i+1} )
SEND( B to P_{i+1} )
if ( A < B ) B = A
```

odd-numbered processor i, $i \in [1, 3, 5, ...]$:

```
SEND( A to P_{i-1} )
RECV( B from P_{i-1} )
if ( A < B ) A = B
```

odd phase for odd-numbered processor i, $i \in [1, 3, 5, ...]$:

```
SEND( A to P_{i+1} )
RECV( B from P_{i+1} )
if ( A > B ) A = B
```

even-numbered processor i, $i \in [2, 4, 6, ...]$:

```
RECV( A from P_{i-1} )
SEND( B to P_{i-1} )
if ( A > B ) B = A
```

or we can combine the above steps for a slightly cleaner presentation:

```
if ( mod(myID,2) == 1 ) ! odd-numbered proc
   SEND( A to P_{i-1} )
   RECV( B from P_{i-1} )
   if ( A < B ) A = B
   if ( myID <= N-3 ) then
      SEND( A to P_{i+1} )
      RECV( B from P_{i+1} )
      if ( A > B ) A = B
   end if
else ! even-numbered proc
   RECV( A from P_{i+1} )
   SEND( B to P_{i+1} )
   if ( A < B ) B = A
   if ( myID >= 2 ) then
      RECV( A from P_{i-1} )
      SEND( B to P_{i-1} )
      if ( A > B ) B = A
   end if
end if
```

Odd-even transposition sort illustrated:

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**

time

- Odd-even transposition bubble sort is $\mathcal{O}(N)$ when run in parallel
- Not too shabby - can we do better?

Mergesort

Mergesort is fundamentally amenable to parallel computation:

- Based on (you guessed it) divide-and-conquer approach
- Preserves input order of equal elements (a.k.a. **stable**)
- Requires $\mathcal{O}(N)$ auxiliary storage space

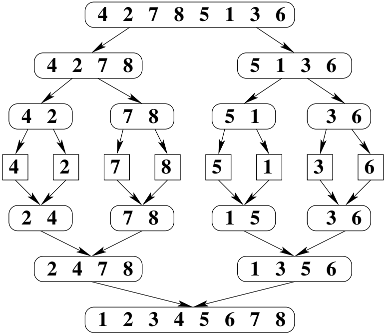
Mergesort Algorithm

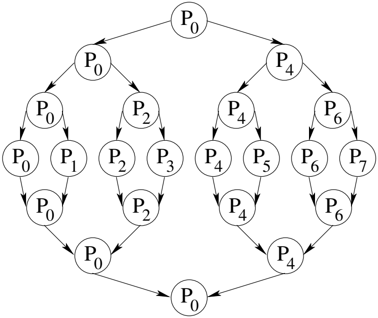
Mergesort algorithm:

- Originated with John Von Neumann (1945)
- List divided in two, then again repeatedly until reach single pairs of data elements
- Elements then merged in correct order
- Well suited to tree structure (but with usual reservations about parallel load-balancing)
- Sequential time complexity is $\mathcal{O}(N \log(N))$

Mergesort Illustrated

Illustration of mergesort (with tree-based parallel process allocation):





Parallel Mergesort Complexity

Time complexity for parallel mergesort

- Sequential time complexity is $\mathcal{O}(N\log(N))$ (average and worst-case)
- Communication: $2\log(N)$ steps,

$$\tau_{comm} \simeq 2(\log(P))\tau_{lat} + 2N\tau_{data}.$$

- Computation: only from merging sublists,

$$\tau_{comp} \simeq \sum_{i=1}^{\log P}(2^i - 1),$$

  or $\mathcal{O}(P)$ using $P = N$ processors.

Quicksort

Quicksort is a lot like **mergesort**:

- Based on divide-and-conquer approach
- Well-suited to recursive implementation
- Sequential time complexity is $\mathcal{O}(N \log(N))$
- Divide into two sublists, all elements in one list smaller than elements in other sublist, using **pivot** element against which others are compared (rinse and repeat!)

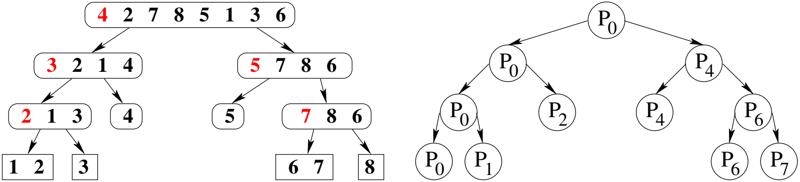
Quicksort Pseudo-code

Pseudo-code for a simple recursive implementation of quicksort (there are lots of variations):

```
Quicksort(list,start,end) {
  if (start < end) {
     Partition(list,start,end,pivot);
     Quicksort(list,start,pivot-1);
     Quicksort(list,pivot+1,end);
  }
}
```

Quicksort Illustrated

Our example processed using quicksort (pivot is first element, shown in red)



along with a naive parallel decomposition

Quicksort Parallel Time Complexity

For parallel quicksort we have time complexity:

- Computation:

$$\tau_{comp} = N + N/2 + N/4 + \ldots \simeq 2N$$

- Communication:

$$\tau_{comm} = (\tau_{lat} + (N/2)\tau_{data}) + (\tau_{lat} + (N/4)\tau_{data}) + \ldots \simeq (\log P)\tau_{lat} + N\tau_{data}$$

- Assuming sublists of approximately equal size, therefore near-perfect load balance
- Worst-case: pivot is the largest element, in which case $\mathcal{O}(N^2)$! (pivot selection is important and tricky)

Batcher's Parallel Sorting Algorithms

The problem is that neither mergesort nor quicksort are very well suited to running in parallel - both suffer from load-balancing issues (negative ones at that). You can certainly apply traditional load balancing methods to both - e.g. master/worker.

Batcher developed several algorithms in the 1960s for sorting in the context of switched networks: **odd-even mergesort** and **bitonic mergesort**, both of which are $\mathcal{O}(\log^2 N)$.

Odd-Even Mergesort

Odd-even mergesort starts with two ordered lists, $\{a_i\}$ and $\{b_i\}$, each of which (for simplicity) has an equal number of elements which is a power of 2.

- Odd index elements are merged into one sorted list $\{c_i\}$
- Even index elements are merged into one sorted list $\{d_i\}$
- Final sorted list, $\{e_i\}$ given by interleaving, i.e.

$$\begin{aligned} e_{2i} &= \mathrm{MIN}(c_{i+1}, d_i) \\ e_{2i+1} &= \mathrm{MAX}(c_{i+1}, d_i) \end{aligned}$$

- These steps are applied recursively, $\mathcal{O}(\log^2 N)$ for $N = P$ processors

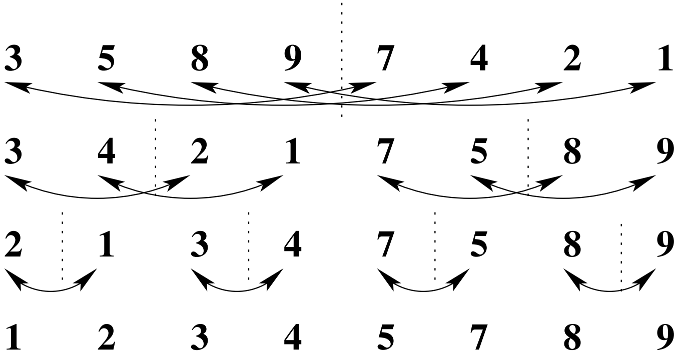
Bitonic Mergesort

Recall that a **bitonic sequence** monotonically increases, then monotonically decreases, i.e. $a_0 < a_1 < \ldots < a_i > a_{i+1} > \ldots > a_{N-1}$. The **bitonic mergesort**:

- If we compare-and-exchange a_i with $a_{i+N/2}$ for all i, we get two bitonic sequences in which all the members of one sequence are less than all the members of the other
- Smaller elements move to the left, larger to the right
- Apply recursively to sort the entire list

Bitonic Mergesort Illustrated

Example of bitonic mergesort:

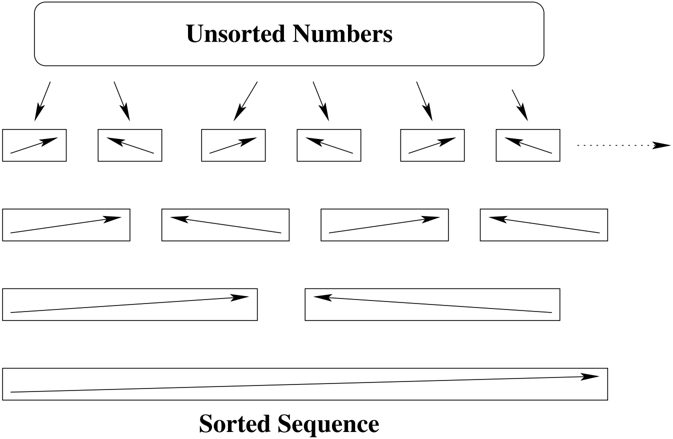


Sorting with Bitonic

Starting from an unordered sequence of numbers:

- First build a bitonic sequence starting from compare-and-exchange on neighboring pairs
- Join into pair of increasing/decreasing sequences
- Repeat to create longer sequences
- End result is single bitonic sequence

Schematic of Bitonic mergesort



Bitonic Example

Bitonic mergesort on eight numbers:

Step								
1	**8**	**3**	**4**	**7**	**9**	**2**	**1**	**5**
	3	**8**	**7**	**4**	**2**	**9**	**5**	**1**
2	**3**	**4**	**7**	**8**	**5**	**9**	**2**	**1**
3	**3**	**4**	**7**	**8**	**9**	**5**	**2**	**1**
4	**3**	**4**	**2**	**1**	**9**	**5**	**7**	**8**
5	**2**	**1**	**3**	**4**	**7**	**5**	**9**	**8**
6	**1**	**2**	**3**	**4**	**5**	**7**	**8**	**9**

Bitonic Mergesort Time Complexity

In this example, we have 3 phases:

- phase 1: (step 1) form pairs into increasing/decreasing sequences
- phase 2: (steps 2/3) split each sequence into two halves, larger sequence at center; sort into increasing/decreasing sequence and merge to form single bitonic sequence
- phase 3: (steps 4-6) sort bitonic sequence

With $N = 2^k$ there are generally k phases, each with $1, 2, ..., k$ steps. Thus the total number of steps is

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2} = \frac{\log N(\log N + 1)}{2} \simeq \mathcal{O}(\log^2 N)$$

Summary of Parallel Sorting Methods

In a nutshell, what we have found thus far:

- Bucket sorting - highly specialized for uniform distribution, $\mathcal{O}(N)$
- Odd-even transposition - $\mathcal{O}(N)$
- Parallel mergesort - load balancing difficult, $\mathcal{O}(N \log N)$
- Parallel quicksort - load balancing difficult, $\mathcal{O}(N \log N)$
- Odd-even and bitonic mergesort, $\mathcal{O}(\log^2 N)$

Bitonic mergesort has a lot of traction as the parallel sorting method of choice.

More Sorting Resources

More sorting resources:

- Donald E. Knuth, *The Art of Computer Programming*, Addison-Wesley, volumes 1, 2, and 3, 3rd edition, 1998.
- W. H. Press et. al., *Numerical Recipes*, Cambridge, 2nd edition, 2002. (Not much parallel sorting, though)

Part II

Spatial Sorting

Spatial Sorting

Spatial sorting is motivated by neighbor-list calculations, which is $\mathcal{O}(N^2)$. Even with a cutoff radius, there is a necessary complication in computing and updating neighbor lists.

- Force calculation and pair-list extraction is $\mathcal{O}(N)$
- Sorting phase is best-case $\mathcal{O}(N \log N)$
- Spatially coherent data structure in which to organize calculation
  - Spatial volumes in which to bin atoms
  - Linked list for each volume element to contain resident atoms
  - Pair interactions can be picked directly from data structures

Spatial Binning

The idea behind spatial binning is a pretty simple one:

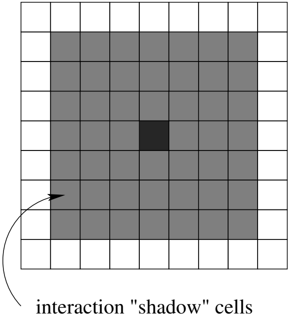
- Improve inter-processor data locality
- Increased scalability through reduced communication

In the following we will consider 2D as an example, but the generalization to 3D is straightforward.

Particle k has coordinates (x_k, y_k):

$$x_{ij} \leq x_k < x_{i,j+1}$$
$$y_{ij} \leq y_k < y_{i+1,j}$$

and the assignment of particles to "bins" is $\mathcal{O}(N)$.



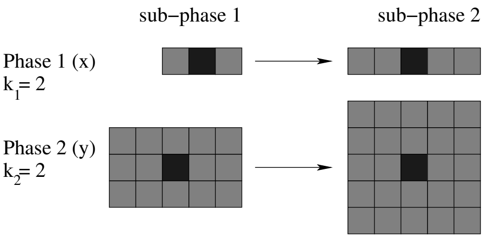
Shift Algorithm

The **shift algorithm** is a method for systematic communication using the data locality in spatial binning

- Clark et. al. "Parallelizing Molecular Dynamics Using Spatial Decomposition," Proc. of Scalable High Performance Computing Conference, IEEE (Knoxville, TN, 1994).
- All shared region data obtained from 4 orthogonal adjacent domains in 2D, 6 in 3D
- Phase d for dimension d ($x = 1, y = 2, z = 3$), k_d sub-phases $k_d = \mathrm{INT}(r_d/u_d)$, where r_d is the interaction distance, u_d the domain extent
- For all sub-phases , process communicates to each of the two neighbors, data received in preceding sub-phase from opposing neighbors

Shift Algorithm Example

2D schematic of shift algorithm:



Monotonic Lagrangian Grid

Monotonic Lagrangian grid (MLG) uses a Lagrangian method, i.e. no fixed grid, everything moves with the particles.

- J. P. Boris, “A Vectorized Near-Neighbor Algorithm of Order N Using a Monotonic Logical Grid,” J. Comp. Phys. **66**, 1-20 (1986).
- Contrast to spatial binning, which is **Eulerian**, or based on a fixed grid
- Also facilitates vectorization for *N*-body methods (improves data locality)
- MLG follows particles as they move

MLG Properties

Properties of the MLG algorithm:

- Sort entire MLG in increasing z, traversing x, then y, then z
- Then sort each $x - y$ plane independently in order of increasing y, traversing x, then y
- Then sort each x row independently in order of increasing x
- Each sort operates on 1D list, $\mathcal{O}(N \log N)$, where $N = N_x N_y N_z$
- Slow moving particles resorted less often (bubblesort variant more effective for nearly ordered list)
- Parallelized using decomposition (preferred is by rows over slabs or blocks, but that increases communication cost), parallel sorts

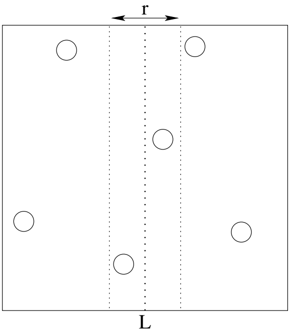
Bentley's Divide & Conquer

- J. L. Bentley, "Multidimensional Divide-and-conquer," Comm. of the ACM **23**, 214-229 (1980).
- Well suited to dynamically adapting size and number of domains
- Consider *N* particles in 2D with interaction radius *r*
- Segment space in A and B, recursively applied for each subspace
- near-neighbor problem is to find all pairs with one partner in A and one in B for each step in recursion

Bentley's Divide & Conquer Algorithm

- Initially bisect along median x-value
- Particles sorted in increasing x
- Pairs along bisector , particles within r in A and B
- Sorted in increasing y, for all particles check within distance r of bisector
- $\mathcal{O}(N)$ if particles sparse along bisector

Bentley's Divide & Conquer Illustrated



Efficiency depends on sparsity along bisector - perverse case is all particles within $r/2$ of bisector, in which $\mathcal{O}(N^2)$.