1. Introduction to Parallel Processing

In the simplest sense, parallel computing is the simultaneous use of multiple computing resources to solve a problem.

a) Types of || machines and || computation.
A conventional computer consists of a processor executing a program stored in a main memory.

Shared memory multiprocessor system
A natural way to extend a single processor model is to have multiple processors connected to multiple memory modules such that each processor can access any memory module in a so-called shared memory configuration.

Message-Passing Multicomputer
Complete computers connected through an interconnection network.

Distributed shared memory
Each processor has access to the whole memory using a single memory address space. For a processor to access a location not in its local memory, message passing must occur to pass data from the processor to the location or from the location to the processor, in some automated way that hides the fact the memory is distributed.

SISD - Single instruction stream, single data stream
In a single processor computer, a single stream of instructions is generated from the program. The instructions operate upon a single stream of data items.

MIMD - Multiple instruction stream, multiple data stream
Each processor has a separate program and one instruction stream is generated from each program for each processor. Each instruction operates upon different data.

MPMD - Multiple program, Multiple data
MPMD applications typically have multiple executable object files (programs). While the application is being run in parallel, each task can be executing the same or different program as other tasks.

SIMD - Single Instruction Stream, Multiple Data Stream
A specially designed computer in which a single instruction stream is from a single program, but multiple data streams exist. The instructions from the program are broadcast to more than one processor. Each processor executes the same instruction in synchronisation, but using different data.

SPMD - Single program multiple data
Single source program is written and each processor will execute its personal copy of this program, although independently and not in sync.

What is granularity? (not sure if covered)

- Granularity is a qualitative measure of the ratio of computation to communication.
- Coarse: relatively large amounts of computational work are done between communication events
- Fine: relatively small amounts of computational work are done between communication events

What is load balance?

- Load balancing refers to the practice of distributing work among tasks so that all tasks are kept busy all of the time. It can be considered a minimization of task idle time.

What is memory overhead?

- Memory overhead refers to the excess / indirect memory required to perform a task.

What is workload imbalance?

- This is where tasks are not evenly distributed across processors.

b) Introduction - Important Laws of || computation.

Speedup Factor

$$S(n) = \frac{\text{Execution time using one processor (single processor system)}}{\text{Execution time using a multiprocessor with } n \text{ processors}} = \frac{t_s}{t_p}$$

where t_s is execution time on a single processor and t_p is execution time on a multiprocessor. S(n) tells you how much of an increase in speed you would gain if you used a multiprocessor. The maximum speedup is n with n processors (linear).

Efficiency

$$E = \frac{\text{Execution time using one processor}}{\text{Execution time using a multiprocessor} \times \text{number of processors}}$$

$$= \frac{t_s}{t_p \times n}$$

which leads to

$$E = \frac{S(n)}{n} \times 100\%$$

when E is given as a percentage.

Efficiency gives fraction of time that processors are being used on computation.

Superlinear speedup

Once in a while you may witness the speedup S(n) being larger than n. This is usually due to using a suboptimal sequential algorithm or some unique feature of the architecture that favours a parallel formation.

One common reason for superlinear speedup is the extra memory in the multiprocessor system which can hold more of the problem data at any instant, it leads to less relatively slow disk memory traffic. Superlinear speedup can occur in search algorithms.

Amdahl's law

$$S(n) = \frac{t_s}{ft_s + (1-f)t_s/n} = \frac{n}{1+(n-1)f}$$

Amdahl's Law states that for a problem of a fixed size(in terms of data), the speed up of a program executed on multiple processors is limited by the serial parts of the program. It is often used to predict the theoretical maximum speedup using multiple processors. (f is the serial part)

For example, Assume that a task has two independent parts, A and B. A takes 75% of the time of the whole computation and B takes 25%, where A is parallelizable and B is serial.

If part A is made to run twice as fast;

- n = 2
- timeA = 75
- timeB = 25
- f = timeB / (timeA + timeB) = 0.25

$$\text{maximum speedup} \leq \frac{2}{1+0.25\cdot(2-1)} = 1.60$$

As can be seen from above equation, no matter how many processors you may add to complete the computation, the maximum speedup achieved will always be limited by the serial part.

f=0 when no serial part → S(n)=n perfect speedup.
f=1 when everything is serial → S(n)=1 no parallel code.

S(n) is increasing when n is increasing
S(n) is decreasing when f is increasing.

We get perfect speedup S(n) = n when f = 0. There is no serial part of the program so all of the work can be done in parallel. Conversely there is no speedup S(n) = 1 when f = 1.

The speedup has an upper bound of 1 / f

$$S(n) = \frac{n}{1 + (n-1) \cdot f} \leq \frac{1}{f}$$

From the above we can see that no matter how many processors are being used, the speedup cannot increase above 1 / f. (the speedup has an upper bound of 1 / f)

This means the gain in speed achieved by adding another processor decreases as n becomes large.

Gustafson's Law

Rather than assume the problem size is fixed, assume that the parallel execution time is fixed. In increasing the problem size, Gustafson also makes the case that the serial section of the code does not increase as the the problem size increases. With increasing data size, the speedup obtained through parallelisation increases, because the parallel work increases(scales) with data size.

$$S(n) = \frac{Sequential\ Time}{Parallel\ Time} = \frac{s \cdot T + n \cdot p \cdot T}{T} = s + n \cdot p$$

When s + p = 1

$$S_s(n) = \frac{s + np}{s + p} = s + np = n + (1 - n)s$$

Assume that a task has a 5% serial part and that there are 20 processors working on this task. The speedup according to Gustafson, where n = 20 and s = 0.05 will be;

0.05 + 0.95(20) = 19.05

If this tasks proof was also tested with Amdahl's Law, the answer would be 10.26. Here the speedup obtained through parallelisation has increased as the number of processors has increased.

Important Consequences:

S(n) is increasing when n is increasing

S(n) is decreasing when n is increasing

There is no upper bound for the speedup - while increasing data to an unlimited size, the speedup obtained through parallelisation increases, because the parallel work increases(scales) with data size.

2. Programming with MPI

Environment management routines

MPI_Init - Initializes the MPI execution environment.

MPI_Init (&argc, &argv) - where argc and argv are the arguments of main()

MPI_Abort - Terminates all MPI processes associated with the communicator.

MPI_Abort (comm, errorcode)

MPI_Wtime Returns an elapsed wall clock time in seconds on the calling processor.

MPI_Wtime()

MPI_Finalize Terminates the MPI execution.

MPI_Finalize()

MPI_Comm_size Determines the number of processes from a communicator.

MPI_Comm_size (comm,&size)

MPI_Comm_rank Determines the rank of the calling process within the communicator.

MPI_Comm_rank (comm,&rank)

MPI_Get_processor_name Returns the processor name.

MPI_Get_processor_name (&name,&resultlength)

Communicators:

Communicators: set of processors that communicate each other. MPI routines require a communicator. **MPI_COMM_WORLD** is the default communicator. Within a communicator each process has a rank.

P2P Communication

MPI P2P operations involve message passing between two different MPI processors. The sender should have MPI_Send and the receiver a MPI_Recv.

The code should look like

```
if(rank==sender || rank == receiver)
{
        if (rank==sender)   MPI_Send(…);
        else if (rank==receiver) MPI_Recv();
}
```

Basic Operations

MPI_Send – Basic send routine returns only after the application buffer in the sending task is free for reuse.

MPI_Send (&buf,count,datatype,dest,tag,comm)

MPI_Isend - Identifies an area in memory to serve as a send buffer. Processing continues immediately (non blocking) without waiting for the message to be copied out from the application buffer.

int MPI_Isend(void sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int tag, MPI_Comm comm, MPI_request *request);*

MPI_Recv - Receive a message and block until the requested data is available.

MPI_Recv (&buf,count,datatype,source,tag,comm,&status)

MPI_Irecv - Identifies an area in memory to serve as a receive buffer. Processing continues immediately (non blocking) without actually waiting for the message to be received and copied into the application buffer.

int MPI_Irecv(void recvbuf, int recvcount, MPI_Datatype recvtype, int source, int tag, MPI_Comm comm, MPI_Request *request);*

MPI_Scatter - Distributes distinct messages from a single source task to each task in the group. This routine is the reverse operation of MPI_Gather.

int MPI_Scatter(void sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);*

MPI_Gather - Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.

int MPI_Gather(void sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);*

MPI_Bcast - Sends a message from the process with rank "root" to all other processes in the group.

int MPI_Bcast(void buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);*

MPI_Reduce - Applies a reduction operation on all tasks in the group and places the result in one task.

int MPI_Reduce(void sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);*

MPI_Cart_create - Creates a communicator containing topology information.

*int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart);*

MPI_Type_vector - Produces a new data type by making count copies of an existing data type, and allows for regular gaps (stride) in the displacements.

*int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);*

Variants of the collective communication:

1. All Collective Communication
   - MPI_Allgather
   - MPI_Allreduce
   - MPI_Alltoall
2. Variable Block Collective Communication
   - MPI_Scatterv
   - MPI_Gatherv
   - MPI_Allgatherv
3. MPI_Reduce_scatter

How to Evaluate Complexity

Times to work with

- Tcom - computation time of 1 floating point operation (how many operations do we have)
- Tcomm - communication time of 1 data (integer, double etc.)
- Tstartup - startup time for one comm operation

(All three of above needed to compute complexity)

For each communication operation

- Have one Tstartup
- Find the number of elements involved and multiply with Tcomm

For each calculation

- Find the number of elements involved and multiply with Tcom

Example; Sum of array

```
int MPI_Array_Sum(int n, double *a, double *sum, int root, MPI_Comm comm) {
        int i, rank, size;
        double local_sum, *local_a;

        MPI_Comm_Size(Comm, &size);

        MPI_Scatter ( &a[0], n/size, MPI_DOUBLE, &local_a[0], n/size, MPI_DOUBLE, 0,
        MPI_COMM_WORLD );

        for( i = 0; i < n/size; i++ ) { local_sum += local_a[i] }

        // want to reduce local_sum into sum
        MPI_Reduce ( &local_sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD );

        return MPI_SUCESS;
}
```

Complexity given by;

Computation : n/size * Tcom

Communication :

scatter : Tstartup + n/size * Tcomm

reduce : Tstartup + 1 * Tcomm

Total complexity is :

2 * Tstartup + (n/size + 1) * Tcomm + n/size * Tcom

Example: Split a matrix onto processors

```
int main(int argc, char ** argv){
        int size, rank, source=0, dest, tag=1, i, j,  n=100, m=100, **a,** b;
        MPI_Status stat;    MPI_Datatype columntype;
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Type_vector(n, 1,m,MPI_INT, &columntype);
        MPI_Type_commit(&columntype);

        if (rank == 0) {
                a=alloc_matrix(n,m);initialise_randomly(n,m,a);
        }
        b=alloc_matrix(n,m/size);
        MPI_Scatter(&a[0][0], m/size, columntype,
                        &b[0][0], m/size, columntype, 0, MPI_COMM_WORLD);

        MPI_Type_free(&columntype);
        MPI_Finalize();
}
```

Scatter n*n/size elements: $T_{startup} + \frac{n^2}{size} \cdot T_{comm}$

Bcast n elements: $T_{startup} + n \cdot T_{comm}$

Compute the product: $\frac{n^2}{size} \cdot T_{com}$

Gather n/size elements: $T_{startup} + \frac{n}{size} \cdot T_{comm}$

Total Complexity:

$$3 \cdot T_{startup} + \left(\frac{n^2 + n}{size} + n \right) \cdot T_{comm} + \frac{n^2}{size} \cdot T_{com}$$

MPI - Derived Datatypes.

MPI communication routines require MPI datatypes.

MPI offers several predefined datatypes:

MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE etc.

MPI_Datatype is used for new data types derived from the above ones.

MPI Derived datatypes:

Contiguous ==> contiguous elements
Vector ==> non-contiguous elements with regular gaps (strides)
Indexed ==> non-contiguous elements with non-regular gaps (strides)
Struct ==> non-contiguous elements that might have different types

MPI Derived datatypes transform non-contiguous elements into contiguous.

Fox's Matrix Multiplication

- The row i of blocks is broadcasted to the row i of processors in the order Ai,i Ai,i+1 Ai,i+2 …Ai,i-1
- The matrix b is partitioned on grid row after row in the normal order.
- In this way each processor has a block of A and a block of B and it can proceed to computation
- After computation roll the matrix b up

Consider the processor rank = (row, col).

1. Partition the matrix b on the grid so that Bi,j goes to Pi,j.
2. For i=0,1,2,..,p-1 times do
    - Broadcast $A_{row,\ row+i}$ to all the processors of the same row.
    - Multiply local_a by local_b and accumulate the product to local_c
    - Send local_b to (row-1, col)
    - Receive in local_b from (row+1, col)

Compare and Exchange (asked a lot)

- Develop an MPI routine for the compare and exchange operation. The prototype of the method can be void MPI_Comp_exchange(int n, int *a, int rank1, int rank2, MPI_Comm comm); (Note: You do not have to write the routine to merge two arrays)
- Evaluate the complexity of compare and exchange and explain why the routine is efficient.

Compare and Exchange Algorithms

Step 1. The array is scattered onto p sub-arrays.

Step 2. Processor rank sorts a sub-array.

At any time the processors keep the sub-arrays sorted.

Step 3. While is not sorted / is needed

compare and exchange between some processors

Step 4. Gather of arrays to restore a sorted array.

About

- Perform 2 blocking point-to-point transmissions of n items.
- Merge two lists of length n.
- Go around the for loop n times.
- Note: Only count complexity for one of the ranks. (the if or the else)

Complexity

Computation :

Merge : 2n * Tcom;

For loop : n * Tcom;

Communication :

MPI_Send : Tstartup + n * Tcomm;

MPI_Recv : Tstartup + n * Tcomm;

Total Complexity is :

(3n * Tcom) + (2 Tstartup + 2n * Tcomm)

Why efficient;

- Done in linear time
- Both the processors do similar work - load balancing
- C&E operations only between neighbours.

Odd-Even and related questions:

- Describe the odd-even parallel algorithm and evaluate its complexity.
- Develop an MPI routine for the odd-even sort and evaluate its complexity.
- Outline briefly how the odd-even deals with the compare and exchange phase and provide information about its complexity. (Odd even is the sort we did for lab 9)
- Outline briefly how the odd-even, shell and biton algorithms deal with compare and exchange and provide information about their complexities.
- Explain the shellsort algorithm and develop a MPI routine to illustrate it when the number of processors is a power of 2. (summer 2007)
- Develop a solution for the shell halving computation when the number of processors is not a power of 2. You can use two arrays ((l[i], u[i]), i = 0, 1, ..., pow(2, l) - l), where l[i] and u[i] represent the lower and upper bounds of the shell i, i = 0,...,pow(2, l) - l at the level l.
- Best is shell then odd-even then merge

Odd-Even:

Total computation complexity ➔

$$\left(\frac{n}{p}\log\frac{n}{p}+2n\right)\cdot T_{com}+\left(2\frac{n}{p}+2n\right)\cdot T_{comm}$$

- Step 1.The array is scattered onto *p* sub-arrays.
- Step 2. Processor rank sorts a sub-array.
- Step 3. While array is not sorted, compare and exchange between some processors
- Step 4. Gather of arrays to restore a sorted array.

- Odd-Even sort uses size rounds of exchange
- Odd-Even sort keeps all processors busy... or almost all.
- Simple and quite efficient.
- The number of steps can be reduced if you test ‘array sorted’.
- C&E operations only between neighbours.
- Operates in two alternating phases, an even phase and an odd phase
  - Even-phase: even-numbered processes exchange numbers with their right neighbour
  - Odd-phase: odd-numbered processes exchange numbers with their right neighbour

```c
int MPI_OddEven_Sort(int n, double *a, int root, MPI_Comm comm) {
	int rank, size, i, sorted_result;
	double *local_a;

	// get rank and size of comm
	MPI_Comm_rank(comm, &rank); //&rank = address of rank
	MPI_Comm_size(comm, &size);

	local_a = (double *) calloc(n/size, sizeof(double));

	// scatter the array a to local_a
	MPI_Scatter( a, n/size, MPI_DOUBLE, local_a, n/size, MPI_DOUBLE, root, comm );

	// sort local_a
	merge_sort(n/size, local_a);

	//do odd-even stages (as in the slide - get the same code, it will help a lot)
	for(i = 0; i < size; i++) {
		if( (i + rank) % 2 == 0 ) { // means i and rank have same nature
			if( rank < size-1 ) {
				MPI_Compare(n/size, local_a, rank, rank+1, comm);
			}
		}
		else if( rank > 0 ) {
			MPI_Compare(n/size, local_a, rank-1, rank, comm);
		}
		MPI_Barrier(comm);

		// test if array is sorted
		MPI_Is_Sorted(n/size, local_a, root, comm, &sorted_result);

		// is sorted gives integer 0 or 1, if 0 => array is sorted
		if(sorted_result == 0) { break; } // check for iterations
	}

	// gather local_a to a
	MPI_Gather( local_a, n/size, MPI_DOUBLE, a, n/size, MPI_DOUBLE, root, comm );

	return MPI_SUCCESS;
}
```

Shell Sort

Total computation complexity ➔

$$\left(\frac{n}{p}\log\frac{n}{p}+2\frac{n}{p}\cdot\log^2 p\right)\cdot T_{com}+\left(2\frac{n}{p}+2\frac{n}{p}\cdot\log^2 p\right)\cdot T_{comm}$$

The idea of Shellsort is the following:

1. arrange the data sequence in a two-dimensional array
2. sort the columns of the array

The effect is that the data sequence is partially sorted. The process above is repeated, but each time with a narrower array, i.e. with a smaller number of columns. In the last step, the array consists of only one column. In each step, the sortedness of the sequence is increased, until in the last step it is completely sorted.

First divide shells, then do odd-even sort.

Can compare with processors that are not immediate neighbours.

```
int MPI_Shell_Sort(int n, double *a, int root, MPI_Comm comm) {
      int rank, size, i, l, k, pair, sorted_result;
      double *local_a;

      // get rank and size of comm
      MPI_Comm_rank(comm, &rank);
      MPI_Comm_size(comm, &size);

      local_a = (double *) calloc(n/size, sizeof(double));

      //Stage 1. Divide the shells
      for(l = 0; l < log(size); l++){
           k = (rank*pow(2, l)) / size;
           pair = (2*k +1)*(size/pow(2, l)) -1 -rank;

            if(rank < pair) {
                  MPI_Compare(n/size, local_a, rank, pair, comm);
            }
            if(rank > pair) {
                  MPI_Compare(n/size, local_a, pair, rank, comm);
            }
      }

      // scatter the array a to local_a
      MPI_Scatter( a, n/size, MPI_DOUBLE, local_a, n/size, MPI_DOUBLE, root, comm );

      // sort local_a
      merge_sort(n/size, local_a);

      // do the odd-even stages
      for(i = 0; i < size; i++) {
```

```
            if( (i + rank) % 2 == 0 ) {
                  if( rank < size-1 ) {
                        MPI_Compare(n/size, local_a, rank, rank+1, comm);
                  }
            }
            else {
                  if( rank > 0 ) {
                        MPI_Compare(n/size, local_a, rank-1, rank, comm);
                  }
            }
            MPI_Barrier(comm);

            // test if array is sorted
            MPI_Is_Sorted(n/size, local_a, root, comm, &sorted_result);

            // is sorted gives integer 0 or 1, if 0 => array is sorted
            if(sorted_result == 0) { break; } // check for iterations
      }

      // gather local_a to a
      MPI_Gather( local_a, n/size, MPI_DOUBLE, a, n/size, MPI_DOUBLE, root, comm );

      return MPI_SUCCESS;
}
```

Linear Sort

Complexity is $O\left(m+n+\sum_j count[j]\right)=O(m+n+n)=O(m+2n)$

- Take an array of numbers
- Count how many times each number occurs in the array
- Re-establish the array with the appropriate amount of copies for each number
- The linear complexity makes this computation perhaps unsuitable for parallel computation.

```
MPI_Linear_sort(int n, int *a, int m, int root, MPI_Comm comm) {
      // The array a is scattered on processors.
      // The count is done on the scattered arrays.
      // The count arrays are all sum-reduced on processors
      // If root then restore the array

      int rank, size, i, sorted_result;
      double *local_a;
```

```
	// get rank and size of comm
	MPI_Comm_rank(comm, &rank); //&rank = address of rank
	MPI_Comm_size(comm, &size);

	local_a = (double *) calloc(n/size, sizeof(double));

	// Scatter the array a to local_a
	MPI_Scatter( a, n/size, MPI_DOUBLE, local_a, n/size, MPI_DOUBLE, root, comm );

	// The count is done on the scattered arrays.
	//for( i = 0; i < n/size; i++ ) { local_sum += local_a[i] }

	// reset the counters
	for(j=0;j<m;j++) count[j] = 0;
	// generate the counters
	for(i=0;i<n;i++) count[a[i]] ++;
	// restore the array order based on counters
	for(j=0;j<m;j++)
		for(k=0;k<count[j];k++)
			a[i++] = j;

	// Reduce local_sum into sum
	MPI_Reduce ( &local_sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD );

	// If root then restore the array
	if(rank == 0) {
		for(i=0; i<n; i++) {
			a[i] = sum[i];
		}
	}
}
```

Bucket Sort

$$(n+\frac{n}{p})T_{comm}+(\frac{n}{p}\cdot log\frac{n}{p}+n+p)\cdot T_{com}$$

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

- Drawback: the whole array is bcasted.
- Drawback: can have uneven distribution in the buckets.
- Benefit: no extra operation needed after gathering.

```
MPI_Bucket_sort(int n, double *a, double m, int root, MPI_Comm comm) {
	int rank, size, *bucketSize, *bucketSizes, n = 100000;
	double m = 1000.0, *a, *bucket;
	MPI_Comm_rank(comm, &rank);
	MPI_Comm_size(comm, &size);

	// Initialise the array a with random values
	a = (double *) calloc(n, sizeof(double));
	bucket = (double *) calloc(n, sizeof(double));
	bucketSizes = (int *) calloc(size, sizeof(int));

	// Braodcast the array to the processor
	MPI_Bcast( &a[0], n, MPI_DOUBLE, root, comm );

	// Collect the elements of bucket rank from array
	bucketSize = 0;
	for ( i = 0; i < n; i++ ) {
		// when a[i] is in the bucket
		if ( a[i] >= m*rank/size && a[i] < m*(rank + 1)/size ) {
			bucket[bucketSize++] = a[i];
		}
	}
```

```
	// Sort the bucket
	merge_sort( bucketSize, bucket );
	// Gather the buckets i.e gather bucketSize to bucketSizes
	MPI_Gather( &bucketSize, 1, MPI_INT, &bucketSizes[0], 1, MPI_INT, root, comm );
	// calculate the displacements
	if(rank==0) {
		for(disp[0]=0; i<size-1; i++) {
			disp[i+1]=disp[i] + bucketSize[i];
		}
	}
	// Gatherv the array
	MPI_Gatherv(&bucket[0], bucketSize, MPI_DOUBLE, &a[0], bucketSizes, disp, 0,
	MPI_COMM_WORLD);
	return MPI_SUCCESS;
}
```

Bitonic Sort

$$\text{Steps} = \sum_{i=1}^{k} i = \frac{k(k+1)}{2} = \frac{\log n(\log n + 1)}{2} = O(\log^2 n)$$

- A bitonic sequence has two sequences, one increasing and one decreasing.
- When we use compare and exchange on a sequence of size n, we get two bitonic sequences, where the numbers in one sequence are all less than the numbers in the other sequence.
- Compare and exchange operation moves smaller numbers from each pair to the left sequence and larger numbers to the right sequence.

```
int MPI_Bitonic(int n, double *a, int rank, MPI_Comm comm) { // Lecture 9
	int rank, size, i;
	double *local_a;

	MPI_Comm_rank(comm, &rank);
	MPI_Comm_size(comm, &size);

	local_a = (double *) calloc(n/size, sizeof(double));

	MPI_Scatter( a, n/size, MPI_DOUBLE, local_a, n/size, MPI_DOUBLE, root, comm );

	bitonic_sort(n/size, local_a);

	for(i = log(size)-1; i >= 0; i--){
		pair = rank^(1<<(i+1));
		if(rank < pair) {
			MPI_Compare(n/size, local_a, rank, pair, comm);
```

```
            }
            if(rank > pair) {
                  MPI_Compare(n/size, local_a, pair, rank, comm);
            }
      }

      MPI_Gather( local_a, n/size, MPI_DOUBLE, a, n/size, MPI_DOUBLE, root, comm );

      return MPI_SUCCESS;
}
```

Rank Sort

The number of numbers that are smaller than each selected number is counted. This count provides the position of selected number in sorted list, i.e. its rank. First a[0] is read and compared to each of the numbers a[1]...a[n-1], recording the number of numbers less than a[0]. This number is the index of the location in the final sorted list.

Sequential sorting time complexity: $O(n^2)$

Potential speedup

O(n log n) is optimal for any sequential sorting algorithm. Best we can expect based upon a sequential sorting algorithm but using n processors is..

$$\text{Optimal parallel time complexity} = \frac{O(n \log n)}{n} = O(\log n)$$