	Title : CS4402 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4402
	Exam Date: Thursday 11th December 2014 @ 16.30

Review / Exam Lecture Notes

- Introduction section: Needs to be learned very well
- Programming with MPI section: Needs to be understood decently

- Question 1 will contain a question on either Amdahl's law, or Gustafson's law
- MUST know consequences of laws; i.e: results of more serial parts/more processors added

- Question 2 will be on MPI Methods

- Question 3 will be on Sorting Algorithms.

- MUST know Core MPI functions: send, recv, scatter, gather, bcast etc. Need to memorise methods with appropriate arguments.
- Will need to write an MPI function with the methods studied
- MUST be able to solve simple routines as done in the labs
- MUST know Compare/Exchange.
- Hinted that Odd/Even sort will be on the exam
- Need to understand/be able to write MPI methods for max, min, product, comp/exch etc.
- Will be a question on examining the complexity of a sorting method (number of operations, evaluated into an expression)
- Possible sorting methods: simplistic, linear, bucket, ranking, odd/even, shell.

- Questions that will NOT appear:
  - Merge sort w/ Divide & Conquer
  - Canon/Matrix Multiplication
  - Fractals
  - Virtual Topologies
  - Bitonic sorting
  - Cloud Computing

- On answering questions:
  - "Briefly describe": Sabin wants 2 sentences, max.
  - "Describe": Sabin wants 4 sentences, max.

	Title : CS4402 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4402
	Exam Date: Thursday 11th December 2014 @ 16.30

2014 Exam with answers

Question 1. Parallel Computing Models

(a) Explain briefly the following terms: Shared Memory Machine, Distributed Memory Machine, SPMD and Load Balancing.

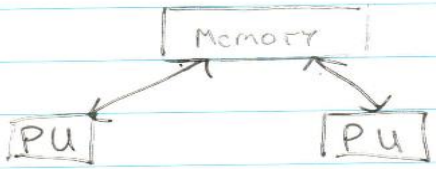
(10 marks)

Q1 = 17

Q1

A(i) Shared Memory Machine

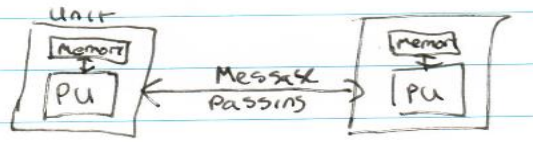
A machine in which each individual processing unit all access a common shared memory



The processing units do not have their own memory

(ii) Distributed Memory Machine

A machine in which each individual processing unit has its own local memory. Data must be shared through message passing



A hybrid machine combines both of these models

UCC Coláiste na hOllscoile Corcaigh, Éire University College Cork, Ireland	Title : CS4402 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4402
	Exam Date: Thursday 11th December 2014 @ 16.30

Q1 a (ii) SPMD. Single Program Multiple Data streams One OF the sub classifications OF MIMD in which a single program acts on Multiple data streams. ~

Flynn's taxonomy

	Single Instruction Stream	Multiple Instructions
Single Data Stream	SISO	MISD
Multiple Data Stream	SIMD	MIMD (SPMD | MPMD)

SPMD

Q1 @ (ii) Load balancing is the act OF attempting to balance the computational burden OF a multiprocessor system evenly across the individual processors in the system

10pts

ucc Coláiste na hOllscoile Corcaigh, Éire University College Cork, Ireland	Title : CS4402 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4402
	Exam Date: Thursday 11th December 2014 @ 16.30

(b) State the Gustafson law and provide a proof for it.

(10 marks)

Q1

b Gustafson's Law states that;

$S(n) = n.\text{SerialPart} + (1 - \text{SerialPart})$ 1pt

(c) Give and explain briefly four consequences of this law.

(10 marks)

Q1

c. It Provides no upperbound on speedup (as you increase the number of processors the speedup also increases

6pts The greater the serial Part, the smaller the speedup

Question 2. MPI Programming and Parallel Algorithms

(a) Explain briefly and give the full prototype for the following MPI functions: MPI_Bcast(), MPI_Gather(), MPI_Gatherv(), MPI_Comm_size().

(10 marks)

	Title : CS4402 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4402
	Exam Date: Thursday 11th December 2014 @ 16.30

Q2 $Q_2 = 50$

A MPI_Bcast. Broadcast the value of of a buffer to all processors within a communication group

```
int MPI_Bcast ( void* buffer, // buffer to send
                int count,    // the size of buffer
                MPI_Data type, // the type of the items in the buffer
                int root,     // the rank of the proc sending buffer
                MPI_Comm comm_group // comm group of processors taking part in Bcast
);
```

Returns response code, MPI_Success if successfull (as do all the following methods)

MPI Gather. Gather a buffer of elements from many processors to one processor

```
int MPI_Gather ( void* buffer,  // Send buffer
                 int count,     // Send count
                 MPI_Data type, // type of items in buffer
                 void* buffer   // receive buffer
                 int count,     // receive count
                 MPI_Data type, // type of receive items
                 int root,      // root of proc gathering to
                 MPI_Comm comm_group // group of processors taking part
);
```

	Title : CS4402 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4402
	Exam Date: Thursday 11th December 2014 @ 16.30

MPI_Gatherv gather data from remote Processors to a root where the length of each set of items a processor can send varies

```
int MPI_Gatherv ( void* buffer, // send buffer
                  int count, // count of send buff
                  MPI_Data type, // type of send buffe
                  void* buffer, // receive buffer
                  int count, // count of receive buffe
                  int* displs, // Array instructing how
                  MPI_Data type, // type of receive bu
                  int root, // destination of gather
                  MPI_Comm commgroup // comm group
                  // of procs performing gather
);
```

one of MPI_Double MPI_INT etc

to stick back the overall array

MPI_Comm_size get the size of a particular comm group

```
int MPI_Comm_size(MPI_Comm comm_group,
                  *int size. // where size will be
                  placed
);
```

10pts

UCC Coláiste na hOllscoile Corcaigh, Éire University College Cork, Ireland	Title : CS4402 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4402
	Exam Date: Thursday 11th December 2014 @ 16.30

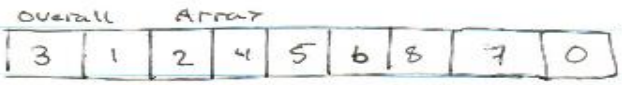
(b) Give an explanation of how the parallel bucket sort works and write an MPI function for it. The prototype of this function may be:

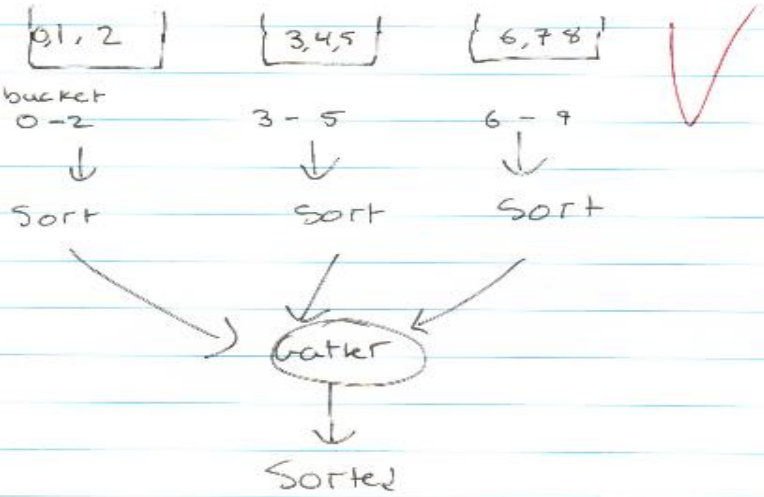
*int MPI_Sort(int n, int *a, int root, MPI_Comm comm)*

(20 marks)

Q2 B

The entire array is broadcast to all Processors in the sorting group each Processor gets the items for their particular bucket:
Once a Processor gets their bucket it sorts the bucket internally.
The buckets are then gathered up in order into the overall sorted array





Assume exists an M such that
M is max item in list

```
int MPI_Sort (int n, int * a, int root,
                MPI_Comm comm)
{
  int size, rank, i, rc;
  MPI_Comm_size (comm, & size);
  MPI_Comm_rank (comm, & rank);
  // Broadcast entire array to all processors
  rc = MPI_Bcast (a, n, MPI_INT, root, comm);
  IF (rc != MPI_Success) {
      MPI_Abort (0, comm);
  }
  int * bucket = (int *) calloc (n, sizeof(int));
  int bucketCount = 0;
  int bucketRangeLength = M / size;
  for (i = 0; i < n; i++) {
      IF (a[i] > rank * bucketRangeLength &&
          a[i] < (rank + 1) * bucketRangeLength) {
          bucket [bucketCount++] = a[i]; }
  }
  Merge_sort (bucket, bucket_count);
  int * overallBucketCount = (int *) calloc (size, sizeof(int));
rc= MPI_Gather (bucketCount, 1, MPI_INT, overallBucketCount,
                1, MPI_Int, root, comm);
IF (rc != MPI_Success) {
    MPI_Abort (0, comm);
}
```

ucc Coláiste na hOllscoile Corcaigh, Éire University College Cork, Ireland	Title : CS4402 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4402
	Exam Date: Thursday 11th December 2014 @ 16.30

```
int * displs ;
IF (rank = root){
    displs = (int *) calloc ( size, sizeof(int));
    displs[0] = 0;
    For (i = 1; i ≤ size; i ++) {
        displs[i] = displs[i-1] + overall BucketCount[i-1];
    }
}

rc= MPI_Gatherv (&bucket, bucketCount, MPI_INT,
                 ra, n, &displs, MPI_INT, root, comm);

return rc;
}
```

20pts

Title : CS4402 Study	
Student Name : Brian O Regan	
Student Number : 110707163	
Module : CS4402	
Exam Date: Thursday 11th December 2014 @ 16.30	

(c) Evaluate the speedup of the *MPI_Sort* function assuming that all buckets are similarly sized. For simplicity consider that the complexity of the sequential sort is quadratic and $T_{startup}=0$.

(10 marks)

Q2

c Speedup = $\frac{T(1)}{T(n)}$

$T(1) = n^2$

$T(n) = T_{comm}(n) + T_{compute}(n)$

$T_{comm}(n) =$

T_{comm} For Bcast = $n.T_{comm}$

T_{comm} for bucketCount gather = $size.T_{comm}$

T_{comm} For gatherv = $n.T_{comm}$

$\Rightarrow T_{comm}(n) = 2n + size.T_{comm}$

$T_{compute}(n) =$

$T_{compute}$ Fill buckets = $n.T_{compute}$

$T_{compute}$ Sort interal buckets = $n/size \log n/size$

From assumption of similarly sized buckets Sorting using normal merge sort

$T_{compute}$ displacements = $size.T_{compute}$

$\Rightarrow T_{compute}(n) = n + \frac{n}{size} \log \frac{n}{size} + size.T_{compute}$

$\Rightarrow T(n) = 2n + size.T_{comm} + n + \frac{n}{size} \log \frac{n}{size} + size.T_{compute}$

$\Rightarrow$ Speedup = $\frac{n^2}{2n + size.T_{comm} + n + \frac{n}{size} \log \frac{n}{size} + size.T_{compute}}$

10

UCC Coláiste na hOllscoile Corcaigh, Éire University College Cork, Ireland	Title : CS4402 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4402
	Exam Date: Thursday 11th December 2014 @ 16.30

(d) Provide and briefly discuss two negative and two positive facts about the *MPI_Sort* function.

(10 marks)

Q2.

D Negative Facts

1 entire array must be broadcast to all processors taking part in sort
⇒ This increases communication complexity substantially

2. Each processor must iterate the entire array to find the contents of its buckets.
⇒ increases linearly with length of array

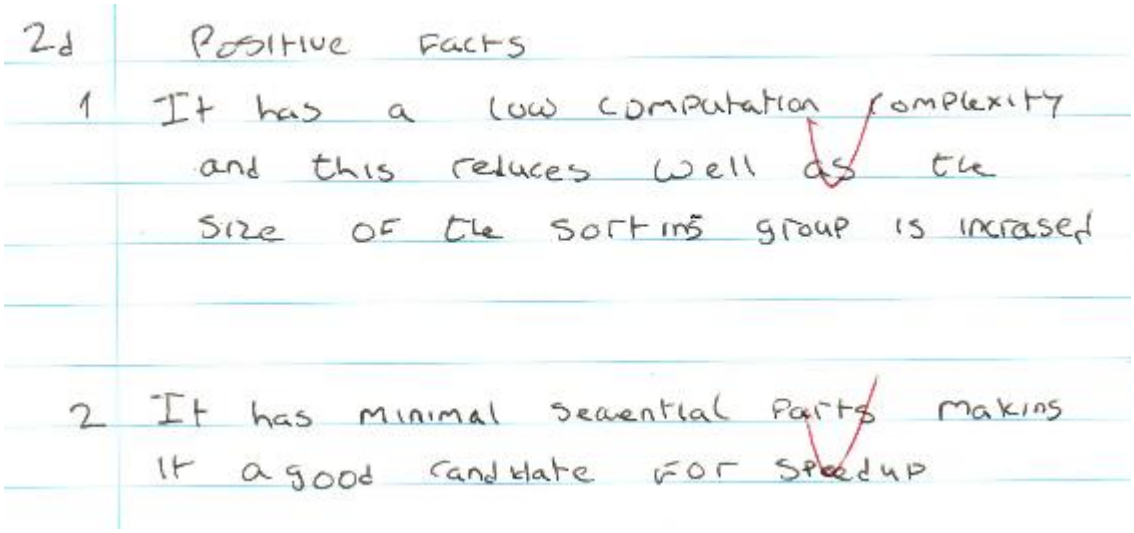
3 A processor does not know how many items are going to be in its bucket. this means it must allocate a bucket of size n
⇒ increases space complexity

10pts

4 An upper and lower bound must be known on the contents of the array, in this case it was assumed m and 0.

5 ⇒ Not all arrays are good candidates for bucket sort. The must be evenly distributed
eg [1, 200042, 200043, 200000, ...]

	Title : CS4402 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4402
	Exam Date: Thursday 11th December 2014 @ 16.30



Terms & Definitions

SISD	Short for single instruction, single data. A type of parallel computing architecture that is classified under Flynn's taxonomy. A single processor executes a single instruction stream, to operate on data stored in a single memory. There is often a central controller that broadcasts the instruction stream to all the processing elements.
MISD	Short for multiple instruction, single data. A type of parallel computing architecture that is classified under Flynn's taxonomy. Each processor owns its control unit and its local memory, making them more powerful than those used in SIMD computers. Each processor operates under the control of an instruction stream issued by its control unit: therefore the processors are potentially all executing different programs on different data while solving different sub-problems of a single problem. This means that the processors usually operate asynchronously.
SIMD	Short for single instruction, multiple data. A type of parallel computing architecture that is classified under Flynn's taxonomy. A single computer instruction perform the same identical action (retrieve, calculate, or store) simultaneously on two or more pieces of data. * Typically this consists of many simple processors, each with a local memory in which it keeps the data which it will work on. Each processor simultaneously

	performs the same instruction on its local data progressing through the instructions in lock-step, with the instructions issued by the controller processor. The processors can communicate with each other in order to perform shifts and other array operations.
MIMD	Short for multiple instruction, multiple data. A type of parallel computing architecture that is classified under Flynn's taxonomy. Multiple computer instructions, which may or may not be the same, and which may or may not be synchronized with each other, perform actions simultaneously on two or more pieces of data. The class of distributed memory MIMD machines is the fastest growing segment of the family of high-performance computers
MPI	M P I = Message Passing Interface An Interface Specification: MPI is a specification for the developers and users of message passing libraries. By itself, it is NOT a library - but rather the specification of what such a library should be. Simply stated, the goal of the Message Passing Interface is to provide a widely used standard for writing message passing programs. The interface attempts to be: practical, portable, efficient, flexible. Interface specifications have been defined for Fortran, C/C++ and Java programs.
SPMD	In computing, SPMD (single program, multiple data) is a technique employed to achieve parallelism; it is a subcategory of MIMD. Tasks are split up and run simultaneously on multiple processors with different input in order to obtain results faster. SPMD is the most common style of parallel programming.

	Title : CS4402 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4402
	Exam Date: Thursday 11th December 2014 @ 16.30

Shared Memory Multiprocessor System

Shared Memory Multiprocessor System

A natural way to extend the 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:

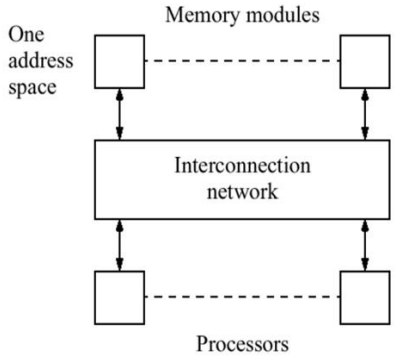


Figure 1.3 Traditional shared memory multiprocessor model.

Message-Passing Multicomputer

Message-Passing Multicomputer

Complete computers connected through an interconnection network:

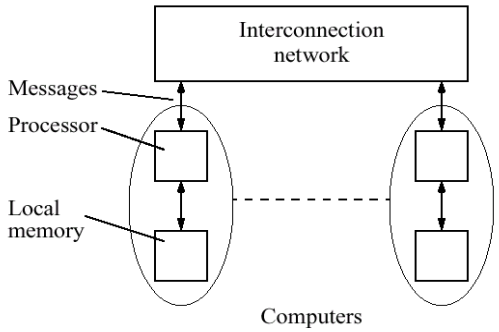


Figure 1.4 Message-passing multiprocessor model (multicomputer).

	Title : CS4402 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4402
	Exam Date: Thursday 11th December 2014 @ 16.30

Distributed Shared Memory

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 that the memory is distributed.

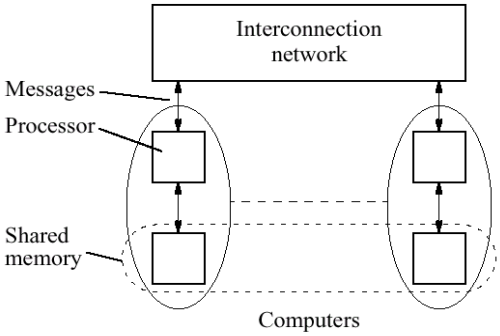


Figure 1.5 Shared memory multiprocessor implementation.

Multiple Program Multiple Data (MPMD)

Multiple Program Multiple Data (MPMD) Structure

Within the MIMD classification, which we are concerned with, each processor will have its own program to execute:

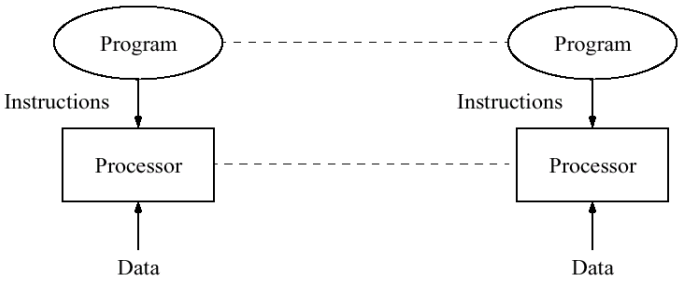


Figure 1.6 MPMD structure.

	Title : CS4402 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4402
	Exam Date: Thursday 11th December 2014 @ 16.30

Single Program Multiple Data (SPMD)	**Single Program Multiple Data (SPMD) Structure** Single source program is written and each processor will execute its personal copy of this program, although independently and not in synchronism. The source program can be constructed so that parts of the program are executed by certain computers and not others depending upon the identity of the computer.

Laws

Flynn's Taxonomy

First proposed by Michael J. Flynn in 1966, Flynn's taxonomy is a specific classification of parallel computer architectures that are based on the number of concurrent instruction (single or multiple) and data streams (single or multiple) available in the architecture. The four categories in Flynn's taxonomy are the following:

- (SISD) single instruction, single data
- (MISD) multiple instruction, single data
- (SIMD) single instruction, multiple data
- (MIMD) multiple instruction, multiple data

Gustafson's Law

This law says that increase of problem size for large machines can retain scalability with respect to the number of processors.

Assume that the workload is scaled up on an n-node machine as, s

$$W' = \alpha W + (1-\alpha)nW$$

Speedup for the scaled up workload is then,

$$S_n^{'} = \frac{Single\,\mathrm{Pr}\,ocessorExeutionTime}{n - ProcessorExecutionTime}$$

$$S_n^{'} = \frac{(\alpha W + (1-\alpha)nW)/1}{\frac{\alpha W}{1} + \frac{(1-\alpha)nW}{n}} \qquad (3)$$

Simplifying Eq.(3) produces the Gustafson's law:

$$S_n^{'} = \alpha + (1-\alpha)n \qquad (4)$$

Notice that if the workload is scaled up to maintain a fixed execution time as the number of processors increases, the speedup increases linearly. What Gustafson's law says is that the true parallel power of a large multiprocessor system is only achievable when a large parallel problem is applied.

Important Consequences:

1) S(n) is increasing when n is increasing
2) S(n) is decreasing when n is increasing
3) There is no upper bound for the speedup.

	Title : CS4402 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4402
	Exam Date: Thursday 11th December 2014 @ 16.30

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 t he 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 t he 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 par allel 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 wor k 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.

Computational workload W is fixed while the number of processors that can work on W can be increased.

In Amdahl's law, computational workload W is fixed while the number of processors that can work on W can be increased.

Denote the execution rate of i processors as R_i, then in a relative comparison they can be simplified as $R_1 = 1$ and $R_n = n$. The workload is also simplified. We assume that the workload consists of sequential work αW and n parallel work $(1-\alpha)W$ where α is between 0 and 1. More specifically, this workload can be written in a vector form as, $W = (\alpha, 0, ..., 0, \alpha - 1)W$, or, $W_1 = \alpha W$, $W_n = (1-\alpha)W$, and $W_i = 0$ for all $i \neq 1, n$.

The execution time of the given work by n processors is then computed as,

$$T_n = \frac{W_1}{R_1} + \frac{W_n}{R_n}$$

Speedup of n processor system is defined using a ratio of execution time, i.e.,

$$S_n = \frac{T_1}{T_n}$$

Substituting the execution time in relation W gives,

$$S_n = \frac{W/1}{\frac{\alpha W}{1} + \frac{(1-\alpha)W}{n}} = \frac{n}{1+(n-1)\alpha} \qquad (1)$$

Eq.(1) is called the Amdahl's law. If the number of processors is increased infinity, the speedup becomes,

$$S_\infty = \frac{1}{\alpha} \qquad (2)$$

Notice that the speedup can NOT be increased to infinity even if the number of processors is increased to infinity. Therefore, Eq.(2) is referred to as a sequential bottle neck of multiprocessor systems.

Important Consequences

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

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

Important Consequences

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

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

Important Consequences

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

no matter how many processors are being used the speedup cannot increase above

Examples:

f = 5% ➔ S(n) < 20
f = 10% ➔ S(n) < 10
f = 20% ➔ S(n) < 5.

Code

Odd-Even

This a variation of Bubble Sort and operates in two alternating phases, an even phase and an off phase.

Even – Even-numbered processes exchange numbers with their right neighbour.
Odd - Odd-numbered processes exchange numbers with their right neighbour.

```
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 the 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;
}
```

Compare

Take place between processors rank1, rank2. Each processor keeps the array a=(a[i],i=0,1,…,n).

Step 1. The array is scattered onto p smaller arrays.
Step 2. Processor rank sorts its local array.
Step 3. While is not sorted / is needed compare and exchange between some processors
Step 4. Gather of arrays to restore the sorted array.

```
int MPI_Compare(int n, double *a, int rank1, int rank2, MPI_Comm comm) {

        int rank, size, i, tag1 = 0, tag2 = 2;

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

        double *b = (double *) calloc(n, sizeof(double));
        double *c;


        //do pingpong between rank 1 and rank 2
        if(rank == rank1) {
           MPI_Send( &a[0], n, MPI_DOUBLE, rank2, tag1, comm );

           MPI_Recv( &b[0], n, MPI_DOUBLE, rank2, tag2, comm, &status );

           c = merge_array(n,a,n,b);
           for(i = 0; i < n; i++) {
                        a[i] = c[i];
           }
        }
        else if(rank == rank2) {
           MPI_Recv( &b[0], n, MPI_DOUBLE, rank1, tag1, comm, &status );

           MPI_Send( &a[0], n, MPI_DOUBLE, rank1, tag2, comm );

           c = merge_array(n,a,n,b);
           for(i = 0; i < n; i++) {
                        a[i] = c[i+n];
           }
        }
        return MPI_SUCCESS;
```

```
}
```

Bucket Sort

Suppose that array=(array[i], i=0,…,n-1) has all elements in the interval [0, a]. Use multiple buckets / collectors to filter the elements in the buckets. Then sort the buckets.

```
MPI_Bucket_sort(int n, double *a, double m, int root, MPI_Comm comm) {

        int rank, size, *bucketSize, *bucketSizes;
        int n = 100000;
        double m = 1000.0;
        double *a, *bucket;

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

        a = (double *) calloc(n, sizeof(double));
        bucket = (double *) calloc(n, sizeof(double));
        bucketSizes = (int *) calloc(size, sizeof(int));

        // Initialise the array a with random values

        // 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;
}
```

Shell Sort

Shell Sort is based on two stages:

Stage 1. Divide the shells

for l=0,1,2, log(p)

- Exchange in parallel between extreme processors in each shell.

Stage 2. Odd-Even

for l=0,1,2, …,p

- if rank and l are both even then exchange in parallel betw rank and rank+1
- if rank and l are both odd then exchange in parallel betw rank and rank+1
- test "array sorted"

Shell Sort Complexity

Shell Sort Complexity

Stage 0. To sort out the scattered array ➔ $\frac{n}{p}\log\frac{n}{p}T_{com}$

Stage 1. Odd-Even for *l* levels ➔

$$2\cdot l\cdot T_{startup}+2\cdot\frac{n}{p}\cdot l\cdot T_{comm}+2\cdot\frac{n}{p}\cdot l\cdot T_{com}=2\cdot\log^2 p\cdot T_{startup}+2\cdot\frac{n}{p}\cdot\log^2 p\cdot T_{comm}+2\cdot\frac{n}{p}\cdot\log^2 p\cdot T_{com}$$

Catch ➔ the average complexity of *l* is in this case O(log^2(p)) so that in average the shell can be

Scatter and Gather ➔ $2\cdot\frac{n}{p}\cdot T_{comm}$

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

```
int MPI_Shell_Sort(int n, double *a, int root, MPI_Comm comm) { // Odd-Even Sort

        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=0,1,2, log(p)
	//	- exchange in parallel betw extreme processors in each shell.
	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

Suppose that the array a=(a[i], i=0,…,n-1) has only integers in 0,1,…,m-1. In this case we can count how many times j=0,1,…,m-1 occurs in a. Then this information is reused to generate the array.

Example:

a=(2,1,3,2,1,3,0,1,1,2,0,3,1)
count[0]=2, count[1]=5, count[2]=3, count[3]=3
a is restore with 2 0-s, 5 1-s, 3 2-s and 3 3-s.
a=(0,0,1,1,1,1,1,2,2,2,3,3,3)

```
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];
                }
        }
}
```

Rank Sort

The number of numbers that are smaller than each of the selected number counted. This count provides the position of the selected number in the sorted list, that is, its rank.

```
MPI_Rank_sort(int n, int * a, int * b, int root, MPI_Comm comm)
{
        int rank, size, * ranking, *overallRanking;
        MPI_Comm_size(comm, &size); MPI_Comm_rank(comm, &rank);
        ranking = (int *) calloc(n/size, size(int)); overallRanking = (int *) calloc(n, size(int));
        // bcast the array a
        MPI_Bcast(&a[0], n, MPI_INT, root, comm);
        // generate the array ranking
        for(i=0; i<n/size; i++)
                for(ranking[i]=j=0; j<n; j++)
                        if(a[j]>a[i+rank*n/size])ranking[i]++;
        // gather ranking
        MPI_Gather(&ranking[0], n/size, MPI_INT, &overallRanking[0], n/size, MPI_INT, root, comm);
        // restore the order
        if(rank==0){
                for(i=0;i<n;i++)b[overallRanking[i]]=a[i];
        }
        return MPI_SUCCESS;
}
```

Sub-Routines

MPI_Bcast()
Sends a message from the process with the rank 'root' to all other processes in the group.

```
MPI_Bcast( &a[0], n, MPI_DOUBLE, root, comm );
```

	Title : CS4402 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4402
	Exam Date: Thursday 11th December 2014 @ 16.30

MPI_Reduce() Applies a reduction operation on all tasks in the group and places the result in one task. *MPI_Reduce (&local_sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);*
MPI_Send() Basic send routine returns only after the application buffer in the sending task is free for reuse. *MPI_Send(&a[0], n, MPI_DOUBLE, rank1, tag1, comm);*
MPI_Recv() Receive a message and block until the requested data is available. *MPI_Recv(&b[0], n, MPI_DOUBLE, rank1, tag1, comm, &status);*
MPI_Init() Initalises the MPI execution environment. *MPI_Init (&argc, &argv)*
MPI_Gather() Gathers direct messages from each task in the group to a single destination task – Opposite of Scatter. *MPI_Gather(&bucketSize, 1, MPI_INT, &bucketSizes[0], 1, MPI_INT, root, comm);*
MPI_Is_Sorted() Test if the array is sorted. Is sorted gives integer 0 or 1. If 0 => array is sorted. *MPI_Is_Sorted(n/size, local_a, root, comm, &sorted_result);*
MPI_Compare() *MPI_Compare(n/size, local_a, rank, pair, comm);*

	Title : CS4402 Study
	Student Name : Brian O Regan
	Student Number : 110707163
	Module : CS4402
	Exam Date: Thursday 11th December 2014 @ 16.30

MPI_Scatter()
Distributes from a single task to each task in the group.

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

MPI_Max() ???

```
MPI_Max_array(int *a, int *max, MPI_Comm comm) {
        If (rank == 0)  {
                max = mymax;

                for(i=1;i<numprocs;i++)  {
                        MPI_Recv(&mymax, 1, MPI_INT, MPI_ANY_SOURCE, 1, MPI_COMMM_WORLD,
&status);
                        if(max<mymax) max = mymax;
                }
                printf(max);
        }
}
```

Quick Revision

Odd-Even = so digest	S	Sort
	D	Do
	I	Is Sorted
	G	Get
	E	Even
	S	Scatter
	T	Test
Bucket Sort = wc bc gigs	W	When
	C	Compare
	B	Broadcast
	C	Calculate
	G	Gather
	I	Initialise
	G	Gather
	S	Sort