Info on Exam

- Very similar to summer exam.
- Explain briefly means 1,2 sentences max
- Explain means 2,3,4 sentences (eg. explain scheduling)
- Elaborate means perhaps ½ page (eg. elaborate on scheduling - (master, slave))
- Look over
    - http://www.cs.ucc.ie/~stabirca/teaching/CS4402/llnlpp/index.htm
    - http://www.cs.ucc.ie/~stabirca/teaching/CS4402/llnlmpi/index.html

No Need To Learn

- Cannon (just be aware)
- isSorted
- MergeSort

Test Structure

- 1 => Intro to Parallel
- 2 => Code (max 15 lines)
- 3 => Sorting Algorithms (bullet points how it works, implementation code)

To Know (says the man himself)

- Basic MPI Routines (most important)
    - size, rank, send, receive, reduce, scatter, gather
    - Do not get into Cartesian, just classical MPI Routines.
- Fundamental Laws of Parallel Computing
    - Amhdall, Gustafson - Proofs, consequences
- General Concepts about Parallel Computing
    - shared memory in parallel computing
- MPI Programming - (Write MPI Code for doing something) (main 3 - oddEven, isSorted, C & E)
    - If you have array, write code for sum
    - If you have array, write code for C & E
    - Know general structure of programs (main 3 I would think)
    - Eventually to analyse and interpret complexity

Questions List (from papers + some slides[where indicated])

Explain briefly;

1. What is a distributed memory machine?
    - Distributed memory refers to a multiple-processor computer system in which each processor has its own private memory. Computational tasks can only operate on local data, and if remote data is required, the computational task must communicate with one or more remote processors.

2. What is a shared memory machine?
    - Multiple processors can operate independently but share the same memory resources. Changes in a memory location made by one processor are visible to all other processors.

3. What is SPMD? (S's asked in past not M's, but just in case learn both)
    - MPMD
        - 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.
        - All tasks may use different data
    - MIMD
        - Currently, the most common type of parallel computer
        - Multiple Instruction: every processor may be executing a different instruction stream
        - Multiple Data: every processor may be working with a different data stream
    - SPMD
        - A single program is executed by all tasks simultaneously.
        - At any moment in time, tasks can be executing the same or different instructions within the same program.
        - All tasks may use different data.
    - SIMD
        - A type of parallel computer
        - Single instruction: All processing units execute the same instruction at any given clock cycle
        - Multiple data: Each processing unit can operate on a different data element
        - Best suited for specialized problems characterized by a high degree of regularity,such as image processing.
    - SISD
        - A serial (non-parallel) computer
        - Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
        - Single data: only one data stream is being used as input during any one clock cycle
        - Examples: most PCs, single CPU workstations and mainframes

4. 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

5. 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.

6. What is memory overhead?
    - Memory overhead refers to the excess / indirect memory required to perform a task.

7. What is workload imbalance?
    - This is where tasks are not evenly distributed across processors.

8. Explain and give the full prototype for the following MPI Routines
    - MPI_Scatter()
        i. Distributes distinct messages from a single source task to each task in the group. This routine is the reverse operation of MPI_Gather.
        ii. int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);

    - MPI_Gather()
        i. Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter.
        ii. int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);

    - MPI_Bcast()
        i. Broadcasts (sends) a message from the process with rank "root" to all other processes in the group.
        ii. int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm);

    - MPI_Reduce()
        i. Applies a reduction operation on all tasks in the group and places the result in one task.
        ii. int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);

    - MPI_Send()
        i. Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse.
        ii. int MPI_Send(void* sendbuf, int sendcount, MPI_Datatype sendtype, int dest, int tag, MPI_Comm comm);

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

    - MPI_Recv()
        i. Receive a message and block until the requested data is available in the application buffer in the receiving task.
        ii. int MPI_Recv(void* recvbuf, int recvcount, MPI_Datatype recvtype, int source, int tag, MPI_Comm comm, MPI_Status *status);

    - MPI_Irecv()
        i. 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 the application buffer.

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

- MPI_Comm_rank()
  i. Determines the rank of the calling process within the communicator. Initially, each process will be assigned a unique integer rank between 0 and number of processors - 1 within the communicator MPI_COMM_WORLD.
  ii. int MPI_Comm_rank(MPI_Comm comm, int *rank);

- MPI_Comm_size()
  i. Determines the number of processes in the group associated with a communicator. Generally used within the communicator MPI_COMM_WORLD to determine the number of processes being used by your application.
  ii. int MPI_Comm_size(MPI_Comm comm, int *size);

- MPI_Cart_create()
  i. Creates a communicator containing topology information.
  ii. int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods, int reorder, MPI_Comm *comm_cart);

- MPI_Type_vector()
  i. Produces a new data type by making count copies of an existing data type, and allows for regular gaps (stride) in the displacements.
  ii. int MPI_Type_vector(int count, int blocklength, int stride, MPI_Datatype oldtype, MPI_Datatype *newtype);

int MPI_Init(&argc, &argv); Initializes MPI environment

Topology is useful as its very easy for a processor to find out its neighbouring cell.

Systolic array: good rhythm and synchronisation, the array executes in pulses.

9. State Gustafson's Law (more chance)
   - Assumes a fixed parallel execution time that can be achieved with arbitrarily large data size.
   - Gustafson's law states that , with increasing data size, the speedup obtained through parallelisation increases, because the parallel work increases(scales) with data size.

   Scaled speedup factor (when s + p = 1)

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

   Gustafson's speedup

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

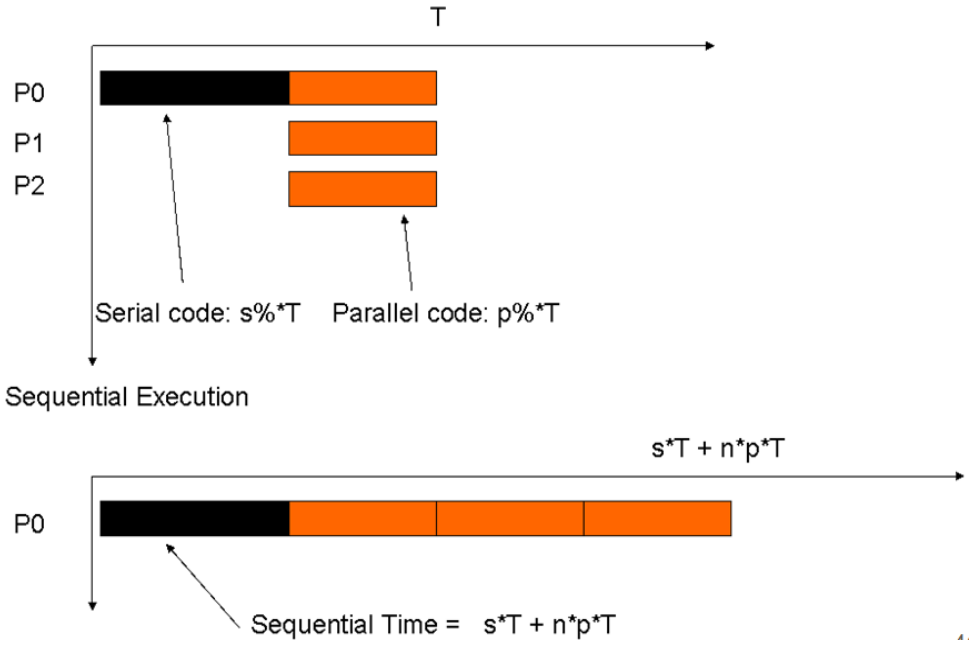
- Provide proof for it.
  (Proof 1)
  Assume that a task has a 5% serial part and that there are 20 processors working on this task. The speedup according to Gustafson, where n = 20 and s = 0.05 will be;

  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.

  (Proof 2)



- Give and explain two consequences of this law.
  i. S(n) is increasing when n is increasing
     S(n) is decreasing when n is decreasing

  ii. 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.

10. State Amdahl's Law
    - 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)

- Provide proof for it.
  (Proof 1)
  The maximum speedup of a program is limited by the inequality;

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

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.

Maximum Speedup

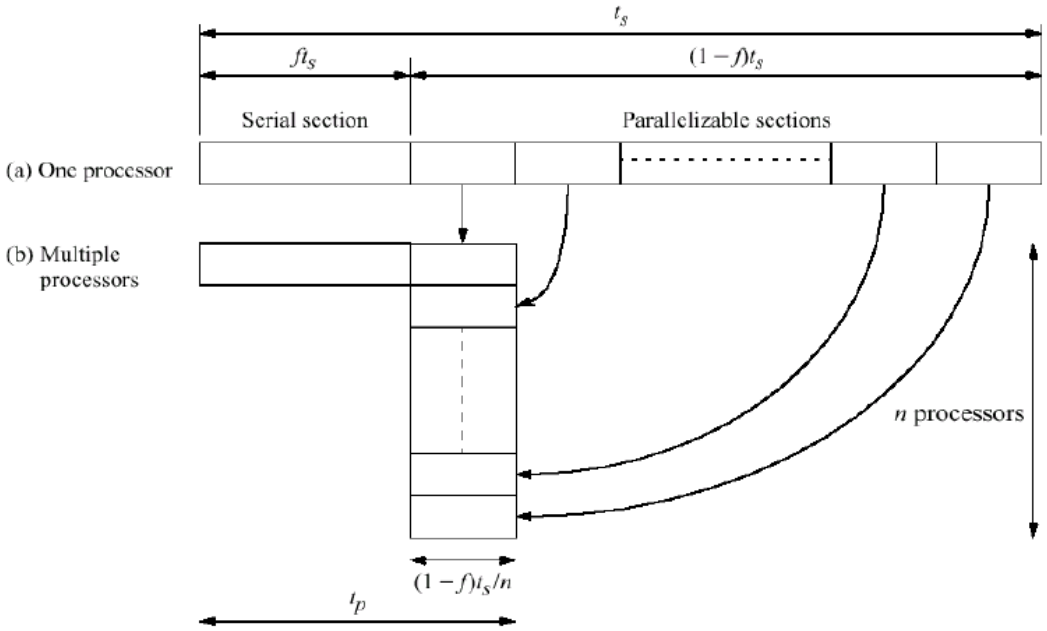


Figure 1.29 Parallelizing sequential problem — Amdahl's law.

where f is the fraction of serial code remaining. The larger f is, the smaller the speedup can be.

- Give and explain two consequences of this law.
  i. 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.

  Overall;
  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.

  ii. 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.

  Examples;

f = 5% -> S(n) < 20
f = 20% -> S(n) < 5
f = 50% -> S(n) < 2
f = 100% -> S(n) < 1

Possible Coding & Complexities

- Basic operations on arrays (sum, max etc.. know one, know em all)
- Basic operations on matrices (know one, know em all) (don't think really important)
- Compare & Exchange (includes point-to-point)
- Sorting Algorithms - odd-even, shell (variation of odd-even), linear, bitonic, bucket
- Divide & Conquer (extra) (code, how to approach, small chance, if you know summation it)
    - A divide and conquer algorithm works by recursively breaking down a problem into two or more subproblems of the same (or related) type, until these become simple enough to be solved directly.
- Note: if asks for routine / function.. don't write full code.

11. 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.

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.

12. Odd-Even & Others
    - 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 Sort ➔

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

- 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

Shell Sort

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

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.

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.

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.

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.

How to Evaluate Complexity

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

    (All three of above needed to compute complexity)

2. For each communication operation
    a. Have one Tstartup
    b. Find the number of elements involved and multiply with Tcomm

3. For each calculation
    a. 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;
}
```

(in for loop: could also be

```
prod *= scattered_array[i];
if(max < scattered_array[i]) {
        max = scattered_array[i];
}
if(min > scattered_array[i]) {
        min = scattered_array[i];
} )
```

(in Reduce: could be &sum, &prod, &max, &min)

Complexity given by;

1. Computation : n/size * Tcom
2. Communication :
    a. scatter : Tstartup + n/size * Tcomm
    b. reduce : Tstartup + 1 * Tcomm

Total complexity is :

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

Divide and Conquer

Characterized by dividing a problem into subproblems that are of the same form as the larger problem. Further divisions into still smaller sub-problems are usually done by recursion

A sequential recursive definition for adding a list of numbers is

```
int add(int *s)                    /* add list of numbers, s */
{
    if (number(s) =< 2) return (n1 + n2);/* see explanation */
    else {
      Divide (s, s1, s2); /* divide s into two parts, s1 and s2 */
      part_sum1 = add(s1);/*recursive calls to add sub lists */
      part_sum2 = add(s2);
      return (part_sum1 + part_sum2);
    }
}
```

Other Revision

Background to Parallel Computing

- In its simplest sense, parallel computing is the simultaneous use of multiple computing resources to solve a problem.
- Point to point only involves 2 processors.
- We need to minimise / avoid serial parts in program. ie. aim to split up program evenly.
- Parallel computing is the solution for the “Grand Challenge Problem”
    - A grand challenge problem is one that cannot be solved in a reasonable amount of time with todays computers. eg. global weather forecasting, modelling DNA structures
- Ultimately parallel computing is an attempt to minimize time.
- Computations must be completed within a ‘reasonable’ time period.
- Parallel computing saves costs by using multiple ‘cheap’ computing resources instead of paying for time on a supercomputer.
- A parallel system is cheaper than a better processor.
- A cluster of workstations (COWs), or a network of workstations (NOWs) offers a very attractive alternative to expensive supercomputers and parallel computer systems for high-performance computing.

Main aspects of Parallel Computing

- Speedup Factor

- S(n) gives the increase in speed in using a multiprocessor.

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

Speedup factor can also be cast in terms of computational steps:

$$S(n) = \frac{\text{Number of computational steps using one processor}}{\text{Number of parallel computational steps with } n \text{ processors}}$$

The maximum speedup is n with n processors (*linear speedup*).

- 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.

- One common reason for superlinear speedup is the extra memory in the multiprocessor system which can hold more of the problem at any instant.

- Cyclic partition - each processor gets even load.