Review of course material for Intro to Parallel Computing

1 Fundamental concepts in parallel computing

1.0.1 Basic notation

Notation	Description
n	input size
p	number of processes (alternatively, number of threads in multithreaded programming)
$T(n,1)$	time taken by the parallel code on 1 process/thread
$T(n,p)$	time taken by the parallel code using p processes/threads
ω	serial work (assumes the use of a best serial algorithm)

1.1 Parallel performance metrics

- **Speedup** is the ratio between serial and parallel run-times. Real speedup $S = \frac{\omega}{T(n,p)}$; Relative speedup $S = \frac{T(n,1)}{T(n,p)}$; By default, the term speedup refers to real speedup.
- **Efficiency** is the ratio between serial and parallel work. It is given by the formula: $E = \frac{S}{p}$. Recall that efficiency measures the % utilization of the parallel system. In other words, it is a better metric for measuring throughput.
- **Parallel work** is given by: $p \times T(n,p)$.
- **Parallel overhead** is denoted by $T_o(n,p)$ and is equal to the difference between the parallel work and serial work — i.e., $T_o(n,p) = p \times T(n,p) - \omega$
- **Amdahl's law** specifies an upperbound on achievable speedup (for a fixed input size)
- **Gustafson's law** releases the speedup barrier posed by Amdahl's law by proposing to increase input work proportional *along with* the increase in parallel resources.
- **Isoefficiency** is an equation expressesing the relationship between serial work and parallel overhead. $\omega = K \times T_o(n,p)$, where $K = \frac{E}{1-E}$. The isoefficiency metric provides a way to compute the factor by which input work should be increased with the number of processors so as to maintain efficiency.

Things to know/ponder:

- All definitions, and conceptual differences between speedup and efficiency, and related properties (e.g., what happens to efficiency when the number of processors is decreased? what can be said about speedup? how to compare algorithms using speedup and efficiency? etc.)

- How to use isoefficiency metric to calculate required increase in workload so as to maintain efficiency at a larger system size? or by what factor should you increase the input work by in order to maintain the same time at more processors? etc.

1.2 Models of parallel computing

1.2.1 PRAM theoretical models

Concurrent/exclusive read, concurrent/exclusive write models — CRCW, CREW, ERCW, EREW. CREW is the most realistic of the PRAM models. PRAM models are generally useful in proving lowerbounds in parallel complexities. We didn't cover much of this in the class but just knowing these is good.

1.2.2 Models based on the concurrency in instructions

These include SISD, MISD, SIMD, MIMD.

Single Instruction: synchronous execution (all processing units execute the same instruction.

Multiple Instruction: asynchronous execution (processing units could be executing different instructions.

Single Data: all processing units are working on the same data.

Multiple Data: processing units can be working on different data.

1.2.3 Models based on memory access

Shared memory model: all processing units share an address space.

Distributed memory model: each processing unit has its own distinct address space.

2 MPI communication

Communication complexity analysis using the Hockney model: Communication time = $O(\tau + \mu \times m)$, where m is the message size.

2.1 Communication patterns

Shift permutation: uses a linear ordering of processors

Ring permutation: uses a circular ordering of processors

Hypercubic permutation: uses a hypercubic ordering of processors (i.e., each processor communicates with a distinct subgroup of $\lg p$ other processors)

Things to ponder/know: For these communication patterns think of ways to express the algorithm in the form of a pseudocode. Basically you need to know how to calculate the communicating partner(s) for a given rank i at a given time step t. For shift and ring permutations this is straightforward. For the hypercubic permutation also this is easy if you model it as a bit shift operation -

i.e., if the destination rank is denoted by $dest$, then $dest$ rank is obtained by toggling the t^{th} least significant bit of my rank i (in binary representation).

2.2 Communication primitives

2.2.1 Point to point

- Send, Recv, Isend, Irecv
- Know the differences between blocking and nonblocking versions

2.3 Collective

- **Reduce and broadcast** operations (Reduce, Allreduce, Bcast). Complexity, assuming m is the message size within each step of communication: $O(\tau + \mu m) \lg p$.
- **Parallel prefix** operation (Scan). Complexity, assuming m is the message size within each step of communication: $O(\tau + \mu m) \lg p$.
- **Gather** operations (Gather, Allgather). Complexity, assuming m is the message size at *each* processor: $O(\tau \lg p + \mu \times m \times p)$.
- **Scatter** operation (Scatter). Complexity, assuming m is the message size at the root processor prior for scattering: $O(\tau \lg p + \mu \times m)$.
- **All to all** operations (Alltoall, Alltoallv). Alltoall complexity, assuming m is the message size that each processor has to send to every other processor: $O(\tau p + \mu \times m \times p)$.
  Alltoallv can be implemented using two Alltoall communications.

Things to ponder/know: You need to know when to use what type of communication primitive. You need to be familiar with these run-time complexities, and be able to derive them (if needed) on demand.

3 Network interconnect topologies

3.1 Network measures

Network Diameter is the length of the longest shortest path between any two nodes in the network. Smaller diameters are preferred.

Links per node denotes the number of physical links connected to each node on the network. Ideally, this should be fixed.

Bisection bandwidth is the minimum number of links needed to be cut in order to divide the parallel system into roughly two equal halves. Larger bisection bandwidth offers better parallelism in communication among halves.

Things to ponder/know:

Given a particular network topology, you should be able to derive these network measures. Recall that in the class we derived most of these measures for array (bus), ring, mesh, torus, and hypercube topologies. The Georgia Tech lecture notes also have these.

3.2 Topologies and embedding

Traditional network topologies discussed in class (in order of complexity):

1. Bus, Ring
2. Mesh, Torus (2D, 3D)
3. Hypercube (d-dimensional)

Embedding is a way to impose the ordering of processors along one network on another network so that adjacency is maintained. For example a bus can be embedded into a mesh by linear ordering the processors along the mesh in a row major order, with alternating rows getting traversed in the same direction. PS: We didn't do any embedding in class and so no questions will be asked in the test but it is good to know.
Things to ponder/know:

- how the different topologies compare by the different network performance metrics
- how to embed one network into another (if that is possible). You can show embedding pictorially. We went over some examples in class - for instance, how to embed an array into a ring, or a ring into mesh. Those are easier to illustrate. We used reflected binary Gray code encoding to relabel the nodes (ranks) of a hypercube into a ring (i.e., to embed a ring into a hypercube).

4 Matrix algorithms

Data partitioning schemes:

- Block partitioning
- Cyclic partitioning
- Block-Cyclic partitioning

Dense Matrix Vector product (MxV): The MxV operation can be parallelized in a couple of different ways, either by block partitioning the rows or block partitioning the columns of the input matrix. If the rows are partitioned then no communication is necessary. If the columns are partitioned then a reduction is necessary at the end to compute the output vector values.

Dense Matrix Matrix product (MxM):

We discussed the Cannon's algorithm to perform MxM in parallel. Be familiar with the overall layout of that algorithm and the complexity results.

5 Parallel prefix

Topics covered:

- Parallel prefix sum
- Algorithm for implementing the parallel prefix operation using any binary associative operator. (Keep track of a local value and global value within each processor; communicate the global value in $\lg p$ steps using a hypercubic permutation.
- Applications of parallel prefix: Polynomial evaluation, Linear recurrences (e.g., Fibonacci number generation), linear congruential random series generator, sequence alignment using dynamic programming, list ranking

Things to ponder/know:

- How to parallelize prefix operations. Technique/approach is key here. For instance, for parallelizing the random series generation, we first represented the problem as a vector matrix product calculation. This helped us use parallel prefix.
- Being able to use parallel prefix as a routine within other applications
- Deriving complexities (analysis)

6 Multithreaded programming in OpenMP

Topics covered:

- Basic concepts in shared memory multicore programming: scoping, scheduling, synchronization, avoiding race conditions
- Using atomic vs. critical section vs. locks for implementing mutual exclusion.
- OpenMP scheduling schemes (static, dynamic, guided) and how it applies to load balancing
- Applications: Parallelization of matrix algorithms (MxV, MxM) using OpenMP, Pagerank estimation

Things to know/ponder:

- Conceptual differences between shared and distributed memory programming models. What are the performance considerations in both?
- Think of algorithms we discussed for distributed memory settings and see how they can be modified/reengineered to make them efficient under shared memory settings? Also, think of the converse.

7 About the Test

The midterm test will be held in class, on the day stated on the course schedule. It will be closed book, closed notes. Use of any computing devices (including calculators) is *NOT* allowed. They won't be necessary.

Be ready to answer analytical questions (and may be a couple of objective questions). Be ready to work out examples. Be ready to design an algorithm for a new problem we didn't discuss in class, using knowledge that you have from algorithms we designed in class already for similar problems.

8 Sorting

This topic was not covered and is therefore not included in the exam syllabus

Topics covered:

- Identifying parallel bottlenecks within traditional sorting methods like quicksort and mergesort
- Sample sort (extending ideas from quicksort under parallel setting): how to select local pivots, how to converge on global pivots, how to use the global pivots to repartition elements across the processors.
- Bitonic sort (extending ideas from mergesort under parallel setting): Bitonic sequence definition, bitonic split property, bitonic merge algorithm, recursively apply bitonic merge algorithm to implement bitonic sort in $O((\lg p)^2)$ steps.

Things to know/ponder:

- Parallel techniques for other sorting techniques
- Variations to sample sort and bitonic sort methods
- Deriving complexities (analysis)
- Be prepared to work out examples to illustrate the sorting algorithms on a need basis.