Introduction

• Moler’s law, Sullivan’s theorem give upper bounds on the speed-up that can be achieved using multiple processors.
• But to get these need to “efficiently” assign the different concurrent processes that make up a concurrent program on the available processors.
• This is called Load Balancing.
• Load balancing is a special case of more general Resource Allocation Problem in a parallel/distributed system.
• In the load balancing situation, resources are processors.
• Before clarifying load balancing problem need to formalise models of the concurrent program and concurrent system.
• To do this, we can use methods such as Graph Theory.
Sources of Parallel Imbalance

• Individual processor performance
  – Typically in the memory system

• Too much parallelism overhead
  – Thread creation, synchronization, communication

• Load imbalance
  – Different amounts of work across processors (comp: comms ratio)
  – Processor heterogeneity (maybe caused by load distribution)

• Recognizing load imbalance
  – Time spent at synchronization is high/uneven across processors

Aside: Graph Theory

• Directed graph are useful in the context of load balancing
• Nodes can represent tasks and the links representing data or communication dependencies
• Need to partition graph so that to minimize execution time.
• The graph partition problem is formally defined on data represented in the form of a graph

\[ G = (V, E) \]  with \( V \) vertices and \( E \) edges

• It is possible to partition \( G \) into smaller components with specific properties.
• For instance, a \( k \)-way partition divides the vertex set into \( k \) smaller components.
• A good partition is defined as one in which the number of edges running between separated components is small.
Graph Theory (cont’d)

• Partition $G$ such that
  – $\{V\} = \{V_1\} \cup \{V_2\} \cup \cdots \cup \{V_n\}$ with $|V_i| \approx |V|/n$
  – As few of $\{E\}$ connecting $|V_i|$ with $|V_j|$ as possible

• If $\{V\} = \{\text{tasks}\}$, each unit cost, edge $e = (i, j)$ (comms between task $i$ and task $j$), and partitioning means
  – $\{V\} = \{V_1\} \cup \{V_2\} \cup \cdots \cup \{V_n\}$ with $|V_i| \approx |V|/n$ i.e. load balancing
  – Minimize $\{E\}$ i.e. minimize comms

• As optimal graph partitioning is NP complete, so use heuristics
• Trades off between partitioner speed & with quality of partition
• Better load balance costs more and law of diminishing returns?

Formal Models in Load Balancing: Task Graphs

• A task graph is a directed acyclic graph where
  – nodes denote the concurrent processes in a concurrent program
  – edges between nodes represent process comms/synchronisation
  – nodal weight is the computational load of the process the node represents
  – edge weight between two nodes is the amount of comms between two processes represented by the two nodes.
Formal Models in Load Balancing: Processor Graphs

- The processor graph defines the configuration of the parallel or distributed system.
- Each node represents a processor & the nodal weight is the computation speed of this processor.
- The edges between nodes represent the communication links between the processors represented by the nodes.
- Edge weight is the speed of this communications link.

Load Balancing Based on Graph Partitioning: Typical Example

- The Nodes represent tasks
- The Edges represent communication cost
- The Node values represent processing cost
- A second node value could represent reassignment cost
Load Balancing: The Problem

• To partition a set of interacting tasks among a set of interconnected processors to maximise “performance”.
• Basically the idea in load balancing is to balance the processor load so they all can proceed at the same rate.
• However formally can define maximising “performance” as:
  – minimising the makespan, $C_{\text{max}}$:
    \[
    \min(C_{\text{max}}) = \min(\max_{i \in \{1, \ldots, n\}} G_i)
    \]
  – minimising the response time, the total idle time, or
  – any other reasonable goal.
• A general assumption that is made is that the comm between tasks on the same processor is much faster than that between two tasks on different processors.
• So intra-processor comm is deemed to be instantaneous.

\[^{1}\text{where makespan is defined as the maximum completion time of any of the } n \text{ tasks}\]

Load Balancing: Allocation & Scheduling

• Load Balancing has two aspects:
  – the allocation of the tasks to processors, and
  – the scheduling of the tasks allocated to a processor.
• Allocation is usually seen as the more important issue.
  – As a result some load balancing algorithms only address allocation.
• Complexity of the problem:
  – Find an allocation of $n$ arbitrarily intercommunicating tasks,
  – constrained by precedence relationships,
  – to an arbitrarily interconnected network of $m$ processing nodes,
  – meeting a given deadline
    this is an NP complete problem.
• Finding $\min(C_{\text{max}})$ for a set of tasks, where any task can execute on any node and is allowed to pre-empt another task, is NP complete even when the number of processing nodes is limited to two.
A hierarchical taxonomy of algorithms is by Casavant and Kuhl.

- **Static Algorithms:**
  - Nodal assignment (once made to processors) is fixed
  - Use only info about the average behaviour of the system.
  - Ignore current state/load of the nodes in the system.
  - Are obviously much simpler.

- **Dynamic Algorithms:**
  - Use runtime state info to make decisions
  - I.e. can tasks be moved from one processor as system state changes?
  - Collect state information and react to system state if it changed
  - Are able to give significantly better performance
Casavant & Kuhl (cont’d):
Centralized V Distributed

- Centralized Algorithms:
  - collect info to server node and it makes assignment decision
  - can make efficient decisions, have lower fault-tolerance
  - must take account of info collection/allocation times

- Distributed Algorithms:
  - contains entities to make decisions on a predefined set of nodes
  - avoid the bottleneck of collecting state info and can react faster
  - don’t have to take account of info times

Load Balancing: Coffman’s Algorithm

- This is an optimal static algorithm that works on arbitrary task (program) graphs.
- Since generally, the problem is NP-complete, some simplifying assumptions must be made:
  1. All tasks have the same execution time.
  2. Comms negligible versus computation. Precedence ordering remains.

- The Algorithm
  1. Assign labels 1, ..., t to the t terminal (i.e. end) tasks.
     a) Let labels 1, ..., j − 1 be assigned, and let S be the set of tasks with no unlabelled successors.
     b) For each node x in S define l(x) as the decreasing sequence of the labels of the immediate successors of x.
     c) Label x as j if l(x) ≤ l(x')(lexicographically) for all x' in S.
  2. Assign the highest labelled ready task to the next available time slot among the two processors.
Scheduling Algorithms

- Concepts of *load balancing* & *scheduling* are closely related.
- The goal of scheduling is to maximize system performance, by switching tasks from busy to less busy/ idle processors.
- A scheduling strategy involves two important decisions:
  1. determine tasks that can be executed in parallel, and
  2. determine where to execute the parallel tasks.
- A decision is normally taken either based on prior knowledge, or on information gathered during execution.
Scheduling Algorithms: Difficulties

- A scheduling strategy design depends on the tasks’ properties:
  a) Cost of tasks
     - do all tasks have the same computation cost?
     - if not, when are costs known? before execution, on creation, or on termination?
  b) Dependencies between tasks
     - can we execute the tasks in any order?
     - if not, when are task dependencies known?
     - again, before execution, when the task is created, or only when it terminates?
  c) Locality
     - is it important that some tasks execute in the same processor to reduce communication costs?
     - when do we know the communication requirements?

- Have come up against a lot of these ideas already in MPI Lectures

Scheduling Algorithms: Differences

- Like Allocation Algorithms, Scheduling Algorithms can be either **Static** or **Dynamic**.
- A key question is when certain information about the load balancing problem is known.
- Leads to a spectrum of solutions:
  1. **Static scheduling**:
     - In this all info is available to the job scheduling algorithm
     - Then this is able to run before any real computation starts.
     - For this case, we can run off-line algorithms, eg graph partitioning algorithms.
Scheduling: Semi-Static Algorithms

2. Semi-Static Scheduling:
   • In this case, info about load balancing may be known
     – program startup, or
     – beginning of each timestep, or
     – at other well-defined points in the execution of the program.
   • Offline algorithms may be used even though the problem has
dynamic aspects. eg Kernighan-Lin Graph Partitioning Algorithm
   • Kernighan-Lin (KL) is a \( O(n^2 \log n) \) heuristic algorithm for
solving the graph partitioning problem.
   • It is commonly applied as a solution to the Travelling Salesman
Problem (TSP) which, ordinarily, is NP complete.

Scheduling: Semi-Static Algorithms (cont’d)

• KL tries to split \( V \) into two disjoint subsets \( A, B \) of equal size.
• Partitioned such that sum \( T \) of the weights of the edges
between nodes in \( A \) and \( B \) is minimized.
• Proceeds by finding an optimal set of interchanges between
elements of \( A, B \) maximizing \( T_{old} - T_{new} \) (iterating as necessary)
• It then executes the operations, partitioning \( V \) into \( A \) and \( B \).
• Kernighan-Lin has many applications in such areas as diverse as:
  – Circuit Board Design (where edges represent solder on a circuit board
    and need to minimize crossings between components represented by
    vertices) and
  – DNA sequencing (where edges represent a similarity measure between
    DNA fragments and the vertices represent DNA fragments themselves).
3. **Dynamic Scheduling:**

- Here load balancing info is only known mid-execution.
- This gives rise to sub-divisions under which dynamic algorithms can be classified:
  - *source-initiative algorithms*, where the processor that generates the task decides which processor will serve the task, and
  - *server-initiative algorithms*, where each processor determines which tasks it will serve.
- Examples of source-initiative algorithms are *random splitting*, *cyclical splitting*, and *join shortest queue*.
- Examples of server-initiative algorithms are *random service*, *cyclical servicing*, *serve longest queue* and *shortest job first*.

**Scheduling: Dynamic Algorithms (cont’d)**

- Server-initiative algorithms tend to out-perform source-initiative algorithms, with the same information content if the communications costs are not a dominating effect.
- However, they are more sensitive to distribution of load generation, and deteriorate quickly when one load source generates more tasks than another.
- But in heavily loaded environments server-initiative algorithms dominate source-initiative algorithms.
Scheduling in Real Time Systems (RTS)

• The goal of scheduling here is to guarantee:
  – that all critical task meet their deadlines and
  – that as many as possible essential tasks meet theirs.
• RTS Scheduling can be synchronous or asynchronous.

1. Synchronous Scheduling Algorithms

• These are static algorithms in which the available processing time is divided by hardware clock into frames.
• Into each frame a set of tasks are allocated which will be guaranteed to be completed by the end of the frame.
• If a task is too big for a frame it is artificially divided into highly dependent tasks such that the smaller tasks can be scheduled into the frames.

RTS Scheduling (cont’d)

2. Asynchronous Scheduling

• This can be either static or dynamic.
• In general dynamic scheduling algorithms are preferred as static algorithms cannot react to changes in state such as h/w or s/w failure in some subsystem.
• Dynamic Asynchronous Scheduling Algorithms in a hard real time system must still guarantee that all critical tasks meet their deadlines under specified failure conditions.
• So critical tasks are scheduled statically and replicates of them are statically allocated to several processors and that the active state information of the task is also duplicated.
• In the event of a processor failure the state information is sent to a duplicate of the task and all further inputs are rerouted to the replicate task.