PPD: Scheduling and Load Balancing

Fernando Silva

Computer Science Department
Center for Research in Advanced Computing Systems (CRACS)
University of Porto, FCUP
http://www.dcc.fc.up.pt/~fds

(Some slides are based on slides from Kathy Yelick, www.cs.berkeley.edu/~yelick)
Goals of Scheduling

The concepts of load balancing and scheduling are very closely related, and are often used with the same meaning.

The goal of scheduling strategy
is to maximize the performance of a parallel system, by transferring tasks from busy processors to other processors that are less busy, or even idle.

A scheduling strategy involves two important decisions:

- determine tasks that can be executed in parallel, and
- determine where to execute the parallel tasks.

A decision is normally taken either based on prior knowledge, or on information gathered during execution.
Difficulties in Devising a Scheduling Strategy

The design of a scheduling strategy depends on tasks properties:

- **Cost of tasks**
  - do all tasks have the same computation cost?
  - if they do not, then when are those costs known?
    - before the execution, when a task is created, or only when it terminates?

- **Dependencies between tasks**
  - can we execute the tasks in any order?
  - if not, when are the dependencies between tasks known?
    - before the execution, when the task is created, or only when it terminates?

- **Locality**
  - is it important that some tasks execute in the same processor to reduce communication costs?
  - when do we know the communication requirements?
Costs of Tasks

Scheduling a set of tasks in the following cases:

*Easy*: The tasks all have equal (unit) cost.

*Harder*: The tasks have different, but known, times.

*Hardest*: The task costs unknown until after execution.
Dependencies between tasks

Scheduling a graph of tasks in the following cases:

**Easy:** The tasks can execute in any order.

**Harder:** The tasks have a predictable structure.
- wave-front
- out-tree
- in-tree
- general dag

**Hardest:** The structure changes dynamically (slowly or quickly)

dependence
free loops

matrix
computations
(dense, and some sparse, Cholesky)

search, sparse LU
Locality of tasks

Scheduling a set of tasks in the following cases:

**Easy:** The tasks, once created, do not communicate.

**Harder:** The tasks communicate in a predictable pattern.

- **Regular:**
  - embarrassingly parallel

- **Irregular:**
  - PDE solver

**Hardest:** The communication pattern is unpredictable.

- **Discrete event simulation**
Scheduling Solutions

- **Static Scheduling** - decisions are taken at compilation time.
  - static analysis of programs is used to estimate size of tasks; this information is hard to obtain and often incomplete.
  - static mapping of the search tree on to the parallel architecture (optimal mapping is NP-complete).
  - build a directed graph with the nodes representing the tasks and the links representing data or communication dependencies; determine the execution order for tasks so that execution time is minimized.

- **Dynamic Scheduling** (or adaptive work sharing) - makes use of computational state information during execution to make decisions.
  - Example: verifies the load of each processor and ensures a dynamic load balancing among all processors.
Why Dynamic Scheduling?

A large class of problems have a solution space that corresponds to a search tree.
These problems are commonly:

- computationally demanding
- allow different parallelization strategies
- require dynamic load balancing

Examples:

- enumerating subgraphs of size k of a given graph
- pattern mining in social and biological networks
- problem of placing a queens in a chessboard
- divide-and-conquer and branch-and-bound problems
- Prolog execution tree
Search Tree

- The tree is dynamically built during or with execution.
- There can be common sub-problems in different paths.

![Search Tree Diagram]

- Terminal node (non-goal)
- Non-terminal node
- Terminal node (goal)
Parallel search

Consider:

- a tree depth-first search
- static scheduling: for each non-busy processor, assign the next new task.

![Load balance on 2 processors](image)

![Load balance on 4 processors](image)

We can and must do better then this!
Strategies for Dynamic Scheduling

The dynamic scheduling strategies can be:

- **centralized**
  - a central scheduler holds all information about the system, namely work-load, and takes decisions for work sharing and transfer.
  - works quite well in shared memory, but with a reduced number of processors.
  - it is inefficient in distributed memory, as it requires much communication to keep scheduling information up to date.

- **distributed**
  - there is one scheduler per processor taking autonomous decisions regarding work sharing.
  - the goal is to maintain its processor busy and balance the workload in the system.
Centralized Scheduling

- The scheduler has a unique queue of tasks
- 1. answers to requests from processors (or workers), or
- 2. the workers access autonomously to the centralized work-queue, synchronizing the access using locks.
Master-Worker

- Centralized strategy normally for distributed memory systems
- Work distribution decisions is done only by the master
- Worker execute a loop:
  - ask for work (task)
  - receive work (task)
  - execute work (task)
  - send results
Difficulties with a centralized strategy

- Avoiding contention in accessing the shared work-queue
- Suppose that tasks correspond to independent iteration of a loop. Let $K$ be the size of a task:
  - If $K$ is large, contention in accessing the work-queue is reduced
  - If $K$ is small, load balancing workload is simpler and more efficient
- Ideas:
  - Use larger tasks at the beginning of execution (more iterations in one task) to avoid excessive overheads and use smaller task sizes near the end of the computation.
  - On access order $i$ to the work-queue, select a task with size $\lceil R_i/p \rceil$, where $R_i$ is the number of processors.
  - The size $K_i$ is a function of the remaining work, but it is also a variance of the cost of the task.
    - the variance is estimated using historical information
    - larger variance $\Rightarrow$ smaller sizes
    - reduced variance $\Rightarrow$ larger sizes
  - Accommodate differences in processing capacity (heterogeneity).
Distributed work-queues (or work-pools)

- Extends naturally for distributed architectures, but also works for shared memory
- Allows for idle workers to take initiative in searching for work, or for busy workers to voluntarily share tasks.
- Useful when:
  - for distributed memory architectures
  - there is much synchronization or many small tasks
Distributed Scheduling

The work sharing decisions can be:

- **sender initiated (or work distribution)** - the busy workers search for less busy workers to give them more tasks.
  - better for smaller workloads.

- **receiver-initiated (or work-stealing)** - the workers that become idle, without work, search for a worker with work and requests or steals work.
  - better for greater workloads in the system
How do we select a worker to request work from?

- Round-robin:
  - $target_k = (target_k + 1) \mod \text{procs.}$

- random polling/stealing
  - When a worker needs work, selects randomly another worker and sends a work request.

- Ask to the last worker:
  - locality might favor requesting to the same worker from whom received work.
  - if this does not have work, apply one of the previous strategies.
How to divide work?

- Number of tasks to distribute? Divide in half?
- Which tasks:
  - topmost: tasks from the beginning of the queue (older tasks)
  - bottommost: tasks from the end of the queue (most recent)
Sharing strategies 1/2

- **Sender-initiated**
  When a worker produces a new task, its scheduler searches for a free (idle) processor and assigns it the task for execution. If it does not succeed, keeps the task in its queue.

- **Receiver-initiated**
  Tasks produced by workers are stored in their local work-queues. When a worker searches for a task for execution, its scheduler searches first in its local queue. If it is empty, then requests or steals work from another worker.
Sharing strategies 2/2

- **Adaptative I**
  Combines the strategies sender-initiated and receiver-initiated. The workers are classified as *Senders* or *Receivers* depending on the value of a *Threshold* parameter. A worker is sender if the number of tasks in its work-queue is above the threshold, it is receiver otherwise.

- **Adaptative II**
  Improves the heuristic Adaptative I, by introducing two parameters *Low* and *High* to classify the workers as *Senders*, *Receivers* or *Neutrals*. A worker changes dynamically its behavior, function of the amount of work in its work-queue:
  - \( \#tasks < high \Rightarrow \) it behaves as sender
  - \( \#tasks < low \Rightarrow \) it behaves as receiver
  - neutral if \( low \leq \#tasks \leq high \)
Example of the evolution in time of the work-queues