

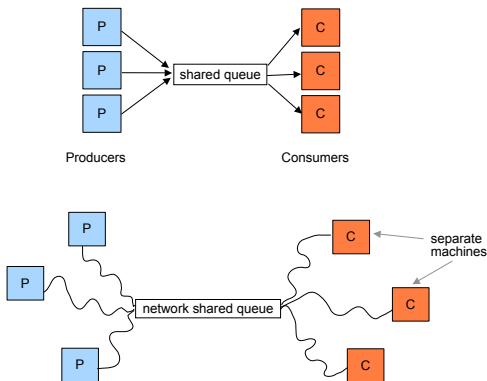
Principles of High Performance Computing (ICS 632)

Programming Model And Map Reduce

Programming Models

- In this class we have used MPI
 - Very low-level
 - Very flexible
 - Very bug-prone
- An never-ending goal of researchers in the area of high-performance and parallel computing is to come up with programming models
- A **programming model** is simply an abstraction
 - A few concepts
 - An API
- The idea is to replace the MPI programming model by simpler programming model
- We know that **_all_** applications can be written with the MPI programming model (or close)
- But aren't there simple programming models that would be useful for **_many_** applications?
- If yes, then there is a market for these models
- A typical model is the workqueue model

Workqueues (Abstract)



Loosely-coupled Models

- Example programming models:
 - Specialized for “mesh computations” as in PDE solvers
 - KeLP, Baden et al.
 - Specialized for Master-Worker applications
 - Condor MW, Livny et al.
 - Specialized for Divide-and-Conquer applications
 - Satin, Bal et al.
 - Etc.
- A very famous programming model is: Remote Procedure Call (RPC)

Remote Procedure Call

- Process A on some host places a call to a function
- This call is executed by process B on some (other) host
- Requires
 - Marshalling and un-marshalling of arguments and returned values
 - Some clever “stub” scheme to be able to compile the caller
- Can be synchronous or asynchronous
- Implemented in many systems
 - CORBA, JRMI, DCOM, etc.

The Map-Reduce Model

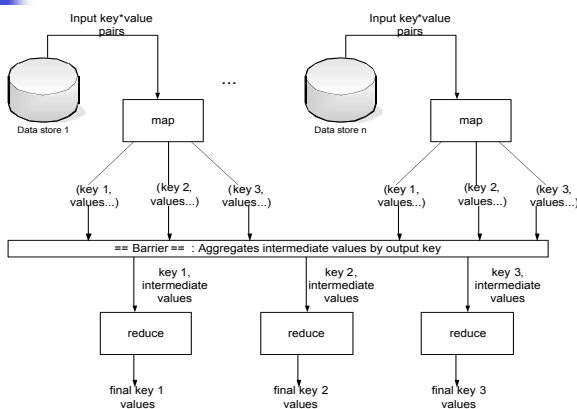
- Inspired by functional programming
 - Functions without side effects
 - LISP Map: take a list and create a new list by applying a function to all elements of it
 - All function evaluations are independent, hence it's possible to do them in parallel
- Map-Reduce's simple abstraction:
 - `map (in_key, in_value) -> (out_key, intermediate_value) list`
 - `reduce (out_key, intermediate_value list) -> out_value list`

Map

- Records from a source (lines out of files, rows of a database, etc) are fed into the map function as key*value pairs: e.g., (filename, line)
- map() produces one or more *intermediate* values along with an output key from the input

Reduce

- After the map phase is over, all the intermediate values for a given output key are combined together into a list
- reduce() combines those intermediate values into one or more *final values* for that same output key
- (in practice, usually only one final value per key)



Example: Count word occurrences

```
map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
        EmitIntermediate(w, "1");

reduce(String output_key, Iterator
intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
        result += ParseInt(v);
    Emit(AsString(result));
```

Locality

- Master program divvies up tasks based on location of data: tries to have map() tasks on same machine as physical file data, or at least same rack
- map() task inputs are divided into 64 MB blocks: same size as Google File System chunks

Fault Tolerance

- Master detects worker failures
 - Re-executes completed & in-progress map() tasks
 - Re-executes in-progress reduce() tasks
- Master notices particular input key/values cause crashes in map(), and skips those values on re-execution.
 - Effect: Can work around bugs in third-party libraries!



Map-Reduce Paper

In-class Discussion and Questions