Multiprocessors
2014/2015

Abstractions of parallel machines

Johan Lukkien
Overview

• Problem context
• Abstraction
• Operating system support
• Language / middleware support
Parallel / distributed processing: Scope

• Several active components (processors) *cooperate* to realize a *single* piece of work…. 

• …. processing occurs on a *single* parallel machine, characterized by:
  – low latency communication between nodes of the machine
  – privately controlled (single *managerial* domain)
  – spatial locality
Parallel / distributed processing: motivation

• Increase speed (absolute performance)
  – imposed by *application domain*
    • e.g., weather forecast, video processing, searching in large data bases (e.g. Google), gaming (compute next frame or scene)
    – can use specialized hardware for this purpose

• Improve price-performance ratio
  – distributed realization may be more cost-effective than a (fast) dedicated one

• Distribution is given fact, system property
  – sensor networks, a cluster, a plant with robots

• Use existing hardware more effectively
• Increase robustness
Concerns: scalability and portability

- **Scalability (which has several interpretations)**
  - good utilization of hardware when the system becomes larger
  - serve a broad range of loads (inputs) maintaining good performance
  - ability to effectively *balance problem increase with machine increase*
  - **Note:**
    - scalability pertains to the physical architecture, and to software, algorithms

- **Portability / programmability**
  - *safeguard investments in application development*
  - what is a reasonable *programming model* to write portable programs?
    - shared/distributed memory
    - Single Program Multiple Data
    - implicit models
  - *abstraction versus efficiency*
    - efficiency of *programmer* (abstraction) versus efficiency of *program*
    - in the performance the hardware will be visible anyway
Programming model

• Definition: set of *concepts, primitives and combiners* to express a computation

• Implemented in tools
  – a language, or language extensions + compiler
  – an API with a library, configuration tools

• The programming model abstracts from details
  – concurrency
  – detailed workings and organization of a machine

• C++, CUDA, MAP/REDUCE
This is what we do....

Programming model: set of concepts, primitives and combinators to express a computation

Problem

Designs [data, functional, result]

Machine model
(e.g. shared/distributed memory)

Machine(s)

dots: alternatives
arrows: abstractions
(no arrow: abstraction not possible)
Example: Map / Reduce on top of GFS

- **Map / Reduce**

- **Connected stations, local memory**

- **Distributed file system (GFS)**

- **cluster of workstations**

---

**dots**: alternatives

**arrows**: abstractions

(no arrow: abstraction not possible)
Example: MapReduce (Google, Apache)

- Map:
  - Accepts input key/value pairs
  - Emits intermediate key/value pairs
- Partition:
  - map intermediate key range to the set of $R$Reducers
    - e.g. $(k, v) \rightarrow \text{hash}(k) \mod R$
- Sort:
  - identical keys are taken together and their values are put in a single list; keys are sorted
- Reduce:
  - Accepts an intermediate key/(value-sequence) pair
  - Emits output key/value pair

Figure: data flow in MapReduce
Example: mapReduce for word count

• map:
  – generates list of (word, “1”) pairs

• reduce:
  – sum the values per key

• Programming model:
  – three (+ ..) API calls
    • map(), reduce(), compare()
  – underlying file system support
  – further tools to manage this

map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
What can you do with it?

- **Grep:**
  - *map*: emit line when pattern matches
  - *reduce*: copy

- **Frequency count (see before)**

- **Sort**
  - *map*: (word, "")
  - *partition*: include range-sort such that data going to different reducers is already sorted with respect to each-other
  - *reduce*: copy

- **Compute the pagerank (for Google search)**
Questions and issues

• Where is the gain?
  – there is a large overhead not visible in (smaller) sequential algorithms
    • writing to files, communication over the network, string interpretation, copying
• What are limitations?
  – size of problem? size of machine?
• What about load balancing?
  – how to achieve a good load balance?
• Can we do all algorithmic problems in this way?
  – does size matter?
  – examples of algorithms that won’t work?
• Required:
  – efficient underlying file system (GFS) & a high bandwidth network
• Is it new???
Overview

• Problem context
• Abstraction
• Operating system support
• Language / middleware support
Abstraction

- Abstraction hides details ("transparency")
  - two important places for abstraction: OS & language levels
  - abstraction is limited to functional aspects
    - extra-functional properties cannot be hidden easily
      - performance, memory use, ...
      - even with abstraction we may have ‘aware’ applications, e.g. cache-aware
    - abstraction makes efficient use of the parallel machine more difficult

- What details are visible in a programming model?
  - what should be visible/hidden by an OS?
  - what should be visible/hidden in a language?
    - or extension library, or API
Example aspects of transparency

- Multiple processors
  - OS: unified concurrency model (threads / processes)
  - Language: automatic extraction of concurrency
    - ‘data’ parallel computation, e.g. HP Fortran

- Interconnect (P-M or PM-PM) & memory model [shared / distributed]
  - OS: P-M: shared, linear memory [(N)UMA], message passing lib
  - OS: PM-PM: ‘distributed OS’
  - Language: PM-PM: fully connected network abstraction, message passing concepts

- Multiple ISA
  - OS: -
  - Language: compilation, virtual machine
Programming model types

• Explicit concurrency:
  – support creation of independent tasks that communicate by passing messages
  – each such task may have internal concurrency with shared memory
  – this maps easily to machine models, and allows optimization

• Implicit concurrency:
  – Domain-specific languages with run-time support
    • (e.g. Google MapReduce, Nvidia CUDA)
  – extracted concurrency and automated mapping (e.g. HPF)
  – this allows concurrency transparency and failure transparency, and increases programmer performance
Conclusion

• What is a preferred programming model?
  – the one that admits portable programs while not sacrificing too much performance
    • example: the Von Neumann machine has a 1-1 counterpart in imperative languages, which form therefore adequate abstractions
  – the one that admits high programmer performance while not sacrificing too much machine performance
Overview

• Problem context
• Abstraction
• Operating system support
• Language / middleware support
Uniprocessor Operating Systems

• Provide “virtual machine” to user:
  – realize transparencies (concurrency, private large memory)
  – factor out common functionality: system calls and libraries
  – virtualization of processor and memory + enforcement
  – unified machine view (portable)
  – ... abstraction, virtualization, resource management

• Ideally, standardize on the interface (system calls): the OS API
  – truly portable programs
  – POSIX initiative

• In practice, standardize on concepts
  – though POSIX is mostly supported
Threads & Processes

- **Thread**
  - unit of concurrency (virtualization of the processor)
  - unit of scheduling
    - though with one thread per process, a process is often said to be the unit of scheduling
  - operates in an address space
    - i.e., in a process; several threads may share this
  - has an associated *execution state*

- **Process**
  - defines a separate data space
  - has at least one thread
  - admits distribution, i.e., moving to another machine
    - depending on the communication primitives that are supplied/used
  - unit of fault containment

---

![Diagram of processes and threads]

- **Process 1**
  - Apache master
- **Process 2**
  - client
- **Process 3**
  - editor
- **Process 4**
  - another process

---

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

9-Mar-15
POSIX processes (1003.1)

- Unit of distribution, concurrency, scheduling
  - single address space: memory management
  - includes one ‘natural’ execution thread
- Usually implemented as *kernel entities*

- POSIX 1003.1 API: `fork()` – `exec()` – `exit()` – `wait()`
Before and after "\texttt{pid = fork()}"

Parent process about to perform \texttt{fork()}

Parent and child after \texttt{fork()}
child is a literal copy of parent

\texttt{pid} of child equals 0

\texttt{variable pid in parent points to child process}
Create new process: code fragment

- `fork()` creates an identical copy of the caller in a separate memory space; only the return value stored in variable ‘child’ differs between the two: 0 for the child, child-processid for the parent.

- `execlp()` overwrites the calling process with its argument (file containing an executable program).

- This means that code following `execlp` is never reached.

- `perror()` is a generic error printing routine.

```c
pid_t child;

......
child = fork();
if (child<0) /* error occurred */  { perror ("fork"); exit (-1); }

if (child == 0) /* the child */ {
    execlp ("child program", "own name", arg0, arg1, ..., NULL);
    /* this place is reached only in case of error */
    perror ("execlp"); exit (-1);
}
else /* the parent; child == process id of child */ {
    /* do whatever you want, e.g., just return from this routine */
}

......
```
Termination of children: fragment

- Use `exit(status)` to terminate

- Need to wait for children to free child’s resources
  - on many systems at least

- Functions `wait()`, `waitpid()`
  - `wait()` blocks until a child returns

- Alternative: (not shown here): use asynchronous notification: signals

child:

..... exit (23);

parent:

```c
pid_t child, terminated; int status;

/* blocking wait */
while (child != wait (&status)) /* nothing */;

/* or polling wait */
terminated = (pid_t) 0;
while (terminated != child) {
    terminated = waitpid (child, &status, WNOHANG);
    /* other useful activities */
}
/* both cases: status == 23 */
```
Multi-threading (pthreads, POSIX 1003.1c)

• Multiple threads within a process
  – introduces all the advantages and disadvantages of shared memory
    • direct access to shared data
    • interference and mutual exclusion

• Rule: blocking of a thread must not influence other threads (really independent and concurrent)
  – also not in system calls
  – avoids simple thread-simulation implementation
    • like run-to-completion, or explicit call-backs to thread library
    • not real-time!
  – some older UNIX (SOLARIS, Distributed Computing Environment) implementations do not abide by this
The life of a thread

• Main program (process): own thread
• Additional threads:
  – created on demand or on receipt of signal
  – thread code: a function in the program
  – behavior & limitations: determined by startup attributes

- initially
- scheduled
- released
- pre-empted
- done (function return), or cancelled
- wait (resource, condition, time, event)
- removed
- detached or joined
- terminated

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking
9-Mar-15
Example

```c
#include <stdio.h>
#include <pthread.h>

int x;

void Count_100 ()
{
    int i, s;
    for (i = 0; i < 100; i ++) {
        x = x+1;
    }
}

void main ()
{
    pthread_t thread_id;
    x = 0;
    pthread_create (&thread_id, NULL, Count_100, NULL);
    Count_100 ();
    pthread_join (thread_id, NULL);
    printf ("x = %d\n", x);
}
```

- Is statement ‘x=x+1’ atomic? Motivate your answer.
- Replace it with a sequence that can be regarded as atomic.
- What are possible final values of x?
Thread interface

- Start a function as a new thread
  - setting attributes, e.g. stacksize;
- This thread terminates when this function returns
- Extensive interface
  - detach: cleanup when thread terminates
  - join: pick up terminated thread
  - cancel: stop a thread

```c
pthread_t thread_id;

status = pthread_create (&thread_id, attr, func, arg_of_func);
/* func(arg_of_func) is started as a new thread; when attr == NULL some
 * internal defaults are chose */

status = pthread_join (thread_id, &result);
/* wait for thread to terminate */
```
Communication & synchronization facilities (primitives)

• Shared memory [with multi-threading]
  – just shared variables, e.g., event flags
  – semaphores, mutexes
  – condition variables
  – readers/writers locks
  – monitors

• Message passing
  – streaming: pipes, fifos, sockets
  – structured: message queues
  – asynchronous, buffered or synchronous

• Signals
  – asynchronous messages, i.e., interrupt the flow of control
    • notice: two interpretations of (a)synchronous
  – may regard the signal handler as a concurrent thread, that is
    (synchronously) waiting for an event
Overview

• Problem context
• Abstraction
• Operating system support
• Language / middleware support
Multiprocessors: extend single processor OS

- Multiprocessors: shared memory models (UMA, (cc)NUMA)

- In principle, same OS with shared code and shared kernel data structures
  - in practice significant reconstruction required
    - system calls need to take care of real competition on access

- Shared-memory synchronization primitives
  - semaphores, monitors
  - in principle similar as in single processor multi-tasking environment
    - but now with implementations that admit real concurrency

- Trend - for scalability and fault tolerance:
  - partition the machine
  - replicate the OS kernel
# Multicomputers: distributed memory

<table>
<thead>
<tr>
<th>System</th>
<th>Description</th>
<th>Main Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOS</td>
<td>Tightly-coupled operating system for homogeneous multicomputers</td>
<td>Hide and manage hardware resources</td>
</tr>
<tr>
<td>NOS</td>
<td>Loosely-coupled operating system for heterogeneous multicomputers (LAN and WAN)</td>
<td>Offer local services to remote clients</td>
</tr>
<tr>
<td>Middleware</td>
<td>Additional layer atop of NOS implementing general-purpose services</td>
<td>Provide distribution transparency</td>
</tr>
</tbody>
</table>

From: Tanenbaum & van Steen: Distributed Systems, Principle and Paradigms

- DOS: Distributed Operating Systems
- NOS: Network Operating Systems
Distributed Operating System

From: Tanenbaum & van Steen: Distributed Systems, Principle and Paradigms
Multicomputers: Distributed Operating System

- Multicomputers: Distributed memory
  - message passing is the basic primitive
  - no simple global synchronization available

- Generally, an OS (micro-)kernel per machine; symmetric: kernels are (almost) the same

- System view: kernels “know” about each other

- Services: transparently distributed

- May emulate shared memory
  - but usually message passing is provided only

- Few actual examples
  - Amoeba
  - QNX – perhaps
  - many projects, at universities, typically aimed at parallel computing
Network Operating System

- Machines independent:
  - own OS
  - heterogeneous

- Services: tied to individual machines

- Example:
  - the Internet, with applications using TCP/UDP
  - services: DNS, DHCP, NFS, ...

From: Tanenbaum & van Steen: Distributed Systems, Principle and Paradigms
Two clients and a server in a NOS

From: Tanenbaum & van Steen: Distributed Systems, Principle and Paradigms
Middleware

- OS’s don’t know each-other, can be entirely different
  - heterogeneous, multiple managerial domains
- Still: distributed, general services needed
  - Factor out common functionality
  - Transparency, in particular w.r.t. heterogeneity, distribution
Middleware services

• Communication
  – RPC, message oriented, ....

• Information
  – database access, directory, naming, ...

• Control
  – transaction processing, code migration

• Security
  – authentication, encryption, ...
# Comparison between Systems

<table>
<thead>
<tr>
<th>Item</th>
<th>Distributed OS</th>
<th>Network OS</th>
<th>Middleware-based OS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Multiproc.</td>
<td>Multicomp.</td>
<td></td>
</tr>
<tr>
<td>Degree of transparency</td>
<td>Very High</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Same OS on all nodes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Number of copies of OS</td>
<td>1 (or: a few)</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Basis for communication</td>
<td>Shared memory</td>
<td>Messages</td>
<td>Files, Messages</td>
</tr>
<tr>
<td>Resource management</td>
<td>Global, central</td>
<td>Global, distributed</td>
<td>Per node</td>
</tr>
<tr>
<td>Scalability</td>
<td>No</td>
<td>Moderately</td>
<td>Yes</td>
</tr>
<tr>
<td>Openness</td>
<td>Closed</td>
<td>Closed</td>
<td>Open</td>
</tr>
</tbody>
</table>

From: Tanenbaum & van Steen: Distributed Systems, Principle and Paradigms
Overview

• Problem context
• Abstraction
• Operating system support
• Language / middleware support
Language & library support

• First, need to understand typical parallel programs needs