1.3 Parallel Programming Models

Thank goodness for modular, hierarchical design techniques or else we'd all be programming in machine language!

Low level: machine language.

- von Neumann machine model = just a processor
  - can execute sequence of instructions
  - "instruction" = arithmetic operations (basic)
    + memory read/writes
    + address of next instruction

Big code = keep track of millions of memory locations & thousands of instructions!

Modular design: complex programs constructed from simple components.

Components: structured in terms of higher-level abstractions:
- data structures, iterative loops, procedures etc.

E.g., due to e.g., procedures ➔ can ignore internal structure

High level: languages (Fortran, C, etc.): allow design expressed in terms of abstractions to be automatically translated into executable code.

Thank goodness for compilers!! (a huge topic subject)
Parallel computation consists of one or more tasks that execute concurrently. A parallel computer consists of a shareable piece of computation and its local memory. No task can exist concurrently. Concepts of tasks and channels are needed to facilitate the processors above.

13.1 Parallel abstractions: tasks & channels needed to facilitate the processors above. Parallel programming is needed to control the multiprocessor model.

So... main concern: so far... concurrency - scalability - modularity - modularity

- more processors - more instructions per second - more instructions per second - additional complexity

- n instructions at low level... (low)
In addition to reading/writing to local memory, a task can perform 4 basic actions:

1. terminate
2. create new tasks
3. send info (asynchronous: complete immediately)
4. receive info (synchronous: blocks [waits] until msg received)

Tasks have ports - an input and an output
Communications (3 & 4) are done by connecting ports via CHANNELS

Tasks can be MAPPED into physical processors in numerous ways:

- multiple tasks → one processor
- single task → one processor

These mapping does NOT affect semantics of program
"Task" abstraction provides concept of [LOCALITY]:
- data in task's local memory are "close"
- data elsewhere are "remote"

"Channel" abstraction provides concept of DEPENDENCY of one task on another i.e. one task needs data from another task in order to proceed.

e.g. Building a bridge out of steel girders

2 tasks: making girders
making bridge out of girders

1 channel: transporting girders from foundry to bridge

Tasks can proceed independently if # girders @ bridge are sufficient

⇒ need channel back to control output of foundry

(foundry) ➔ [transport] ➔ [control] ➔ [bridge]

(if rate of production of girders too slow, ask for more)
(if too high, tell them to slow down) (ship production of girders when bridge complete)
Other properties of TASK/CHANNEL model:

1. **Mapping independence**: End result should NOT depend on mapping!
   - In task/channel model, tasks interact via channels always regardless of location of task, so result does not depend on where tasks execute.
   - Algorithms can be designed & implemented in terms of tasks & channels **without concern for** no. of processors on which they will execute.
   - **Notice**: often create more tasks than processors
     - **scalability**: (add more processors, algorithm does not change, just mapping).
     - can make communication delays (**latency**) less - execute more computation while accessing remote data

2. **Performance**
   - Two tasks sharing a channel
     - mapped to **different** processors or
     - mapped to **same** processor
     - may mean different efficiencies
   - Generally, **interprocessor communication** less efficient than **intra-processor**
3. Modularity

- Pieces can be developed separately and then incorporated into a complete program.
- Can change module without changing other components.
- Interaction between modules restricted by well-defined interfaces.
- Reduces complexity.
- Encourages code re-use.

Task (channel model):

- Task encapsulates both data and code that operate on data.
- The inputs/outputs are the interface.
- Have the advantage of modular design.

(Similar to "object-oriented" programming paradigm:
- tasks = objects = encapsulate data & code
- but tasks = concurrency
  (but no inheritance)

4. Determinism

Deterministic: particular input always yields particular output.

Good thing!

In the task/channel model, determinism likely if:
- Each channel has a single sender and single receiver.
- Receiving channel blocks until operation complete.
e.g. Bridge construction:

- War some bridge regardless of rate of girder and bridge construction!

  - If bridge assembly crews are fast and run ahead of girder production, they block (wait) for girders to arrive (rather than trying to use half-completed girders).
  - If girder production is fast, girders simply accumulate ready to be used.
  - If multiple bridges being built, all is fine as long as girders designed for different bridges travel on different channels.

Other programming models

1. Task/channel model is an example of "task parallelism": Different tasks execute concurrently.

Message passing

- Most widely-used parallel programming model
- It is a "task parallel" model — a subset of task/channel
- Each task has a unique name (ID)
- Instead of specifying channels, just specifies IDs for "to" and "from"
- I.e., "Send to task10" rather than "Send on channel 5" (channel 5 goes to tank 18).
Message passing in general a bit more overhead:

- does not preclude
  - dynamic creation of tasks
  - multiple tasks per processor

but in practice, most message passing programs create fixed no of identical tasks.

This is called SPMD - Single Program Multiple Data

Data Parallelism

Concurrency derives from doing same operation over and over on different data within a data structure.

- e.g. add 2 to all elements of an array
- increase salary of all associate professors

Can think of as separate task for each data element

- granularity very small
- no natural concept of locality.

Need to provide information as to how to distribute data over processors, i.e. how to partition into TASKS then compiler can work in SPMD mode.

- task/channel model can still be applied
  (not so different)
3) **Shared memory**

- Tasks share a common address space
- Read and write to this asynchronously
- Locks / semaphores must be used to control access to shared memory (synchronization)

**Advantages**
- No concept of "ownership" of data
  - Everyone has everything.

**Disadvantages**
- Concept of locality hidden = difficult to manage (e.g., cache coherence)
  - Also can be more difficult to make determinisitic (due to async write to memory)
1.4 Parallel Algorithm Examples

1.4.1 1-D Finite Differences

Update array values by:

\[ x_i(t+1) = x_{i-1}(t) + 2x_i(t) + x_{i+1}(t) \]
\[ t = 0, \ldots, T \]

\[ x_i, i = 1, \ldots, 8 \] \( \leftrightarrow \) 8 points

Parallel algorithm: Assign tasks →

8 tasks = 1 for each \( x \)

\( i^{th} \) task - update \( x_i \) from \( t=0 \) to \( t=T \)

\[ \text{left} \rightarrow \text{our} \rightarrow \text{right} \]

\[ \text{in} \rightarrow \text{our} \rightarrow \text{in} \]

or cleaner looking

\[ \ldots \rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow \ldots \]

Data transfer by channels "left" and "right"

At each step \( t \), task \( i \)

1. Send data \( x_i^{(t)} \) on outputs to left and right

2. Receive \( x_{i-1}^{(t)} \) from left, \( x_{i+1}^{(t)} \) from right inputs

3. Compute \( x_i^{(t+1)} \)

\( N \) independent tasks

Synchronisation enforced by blocking receive \( \Rightarrow \) deterministic (cannot do step \( t+1 \) until info from \( t \) received)
14.2 Pairwise Interactions

All pairwise interactions \( I(x_i, x_j) \), \( i \neq j \)
on \( N \) data \( \Rightarrow N(N-1) \) interactions

- If interaction symmetric \( I(x_i, x_j) = (\pm) I(x_j, x_i) \)
  \[ \Rightarrow N(N-1)/2 \]

Parallel algorithm:
- \( N \) tasks: task \( i \) for \( x_i \)
  - compute \( I(x_i, x_j) \), \( j = 0, \ldots, n-1 \)
    - \( j \neq i \)

Channels:

- **naive**: each \( x_i \) interacts with each \( x_j \)
  \[ \Rightarrow N(N-1) \] channels
- **sensible**: \( N \) channels!

Undirected ring
- Each task:
  - 1 input
  - 1 output

Task has two receive buffers
- 1 datum buffer initialized with \( x_i \)
- 1 accumulator

Then repeatedly \( (N-1) \) times

1. send buffer on output
2. receive datum on input into buffer
3. interact contents of buffer with local \( x_i \)
4. update local accumulator
if interactions are symmetric, can halve no of communications and no of interactions compared to case but requires refinement of channels:

→ link each task i to task i + N \%2 N \% as well

→ N more channels (2N total)

→ continue as before, but each time

\[ I(x_i, x_j) \quad x_i \text{ local datum} \]

\[ x_j \text{ incoming datum} \]

is computed, accumulate in accumulator for \( x_i \) and in another accumulator that circles with \( x_j \).

→ after N/2 steps, send \( x_j \) and accumulated interactions back to home task via new paths.

→ each \( I(x_i, x_j) \) only calculated once

1.4.3 Search

Example of dynamic task allocation

Search tree that looks for nodes corresponding to a solution

\[
\text{procedure search}(A) \]

begin

\[
\text{if} \ (\text{solution}(A)) \ \text{then report solution}
\]

\[
\text{else}
\]

\[
\text{foreach} \ (\text{child of } A) \ \text{search}(A_i)
\]

\[
\text{end if}
\]

end.
Initiate 1 task.
If solution yes
else search children.

At any time, some tasks terminating at leaves, some tasks dynamically creating new tasks to continue search. Solutions reported back up tree (up "branch" channels).

1.4.4 Parameter study - Embarassingly Parallel

Embarassingly parallel \(\Rightarrow\) (virtually) no communication
\(\Rightarrow\) completely independent problems
E.g. a parameter study using same code, different params.
If execution time independent of parameters, all processors same speed, just allocate a partition set of the problem to each processor.

Many-to-one request policy \(\Rightarrow\) NOT DETERMINISTIC
If processors vary in speed, results output in different order.
non-determinism not a problem here
- only affects order results collected in
- does not affect results themselves

Note also:
- worker requests parameter then works for
- it to arrive.

Can improve efficiency by prefetching
- request next parameter before it is needed.

≡ overlap of communication & computation
   (Key idea!)

1.5 Summary Chapter 1

A/ Four desirable attributes of parallel algorithms and software:
1/ concurrency - perform actions simultaneously
2/ scalability - resilience to changing (increasing) processor counts
3/ locality - high rate of local to remote memory access
   ⇒ efficiency
4/ modularity - decomposition of complex entities into simpler components

B/ Parallel machine model - multi computer
   = one or more von Neumann computers joined by interconnect

C/ Programming model - task/channel - provides abstractions
   that allows to talk about