1.3 Parallel Programming Models

(Thanks #1) 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 that can execute sequences of instructions

→ "instruction" = arithmetic operations (basic)
+ memory read / writes
+ address of next instruction

Big codes ⇒ 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

Ease due to e.g. procedures → can ignore internal structure

So...

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

(Thanks #2) Thank goodness for compilers!! (a huge tough subject)

Parallel programming ⇒ additional complexity.
If programmed at low level ... (wow)
- # of instructions increases
- manage more processors
- co-ordinate interprocessor interactions

⇒ abstraction & modularity even more important.

Monty Python

SO ... main weapons so far are:
- concurrency
- scalability
- locality
- modularity

} requirements for parallel software.

1.3.1 Parallel abstractions: tasks & channels

Need to facilitate the 4 paradigms above,
and need to match the multicomputer model

Concepts of TASKS and CHANNELS

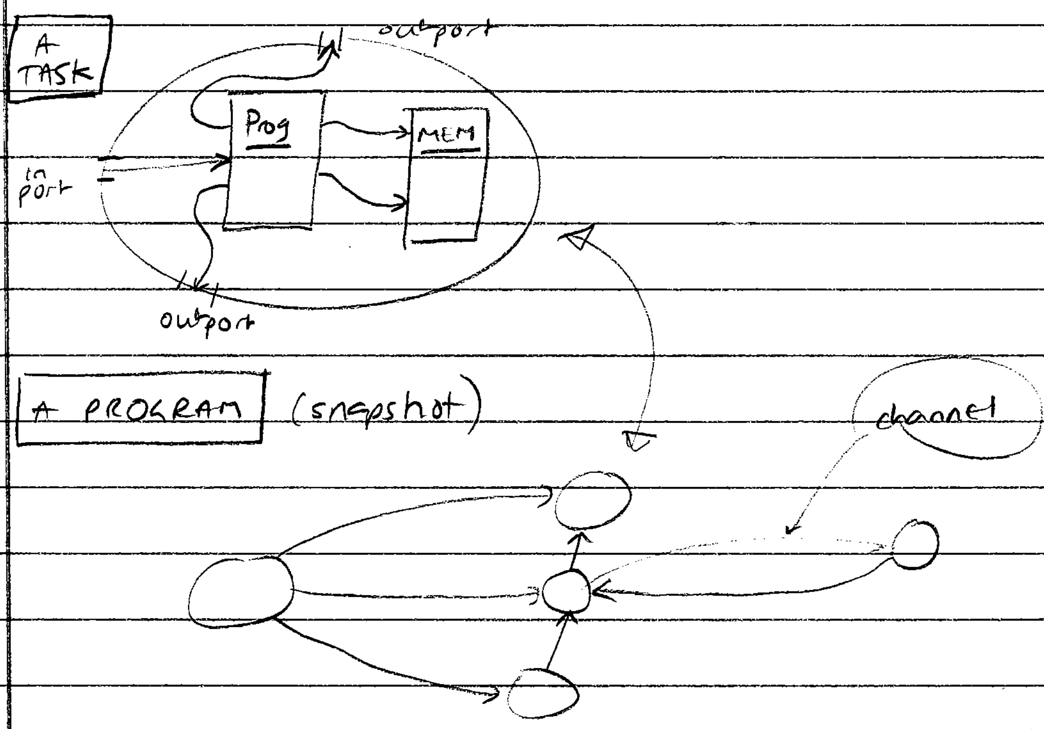
TASKS:
→ a sequential piece of computation and it's local memory
(= a VIRTUAL von Neumann computer)

→ a parallel computation consists of one or more tasks
→ tasks execute CONCURRENTLY
→ no. of tasks can vary during computation

In addition to reading / writing it's local memory, a task can perform 4 basic actions:

1/ terminate
2/ create new tasks
3/ send info (asynchronous : completes immediately)
4/ receive info (synch : blocks [waits] until msg received)

Tasks have PORTS – an INport and a OUTport
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
etc

= decomposition (see later)

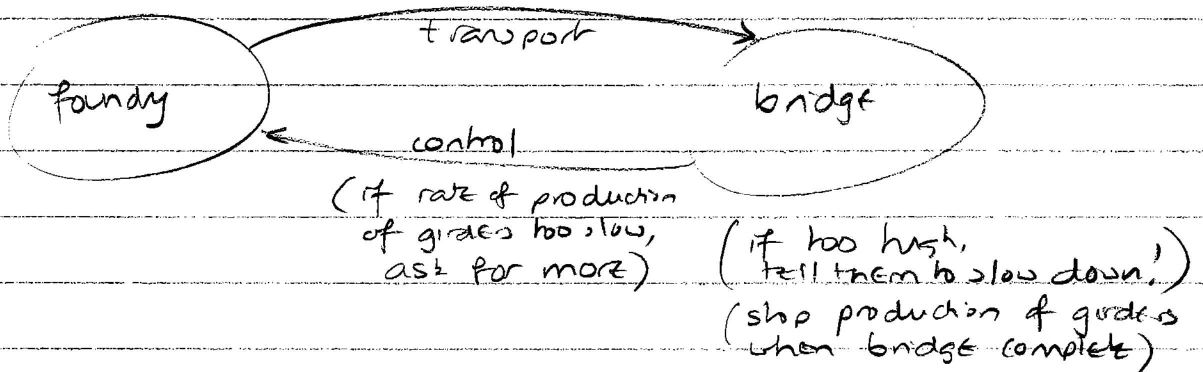
Mapping does NOT affect semantics of program

"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 → bridge

Tasks can proceed independently if # girders @ bridge are sufficient
⇒ need channel back to control output of foundry



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 mask communication delays (LATENCY) too – execute more computation while accessing remote data.

BUT ...

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**

Modular ⇒ pieces can be developed SEPARATELY and then incorporated into complete program.
→ can change module without changing other components
→ interaction between modules restricted to well-defined interfaces

modular ⇒ reduces complexity
⇒ encourages code re-use

Task/channel model:
Task encapsulates both data and code that operates on data.
The inports/outports are the interface
⇒ have the advantages 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 completes

e.g. Bridge construction:

Want same bridge regardless of rates of girder and bridge construction!

→ If bridge assembly crew 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 too fast, girders simply accumulate ready to be used.

→ If multiple bridges being built, all is fine as long as girders destined for different bridges travel on different channels.

Other programming models

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 task 10" rather than "send on channel 5" (channel 5 goes to task 10)

message passing in general a bit more restrictive:
→ 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

2/ Data Parallelism

(not so different: smaller granularity)

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/ <u>shared memory</u>

→ Tasks share a <u>common</u> address space

→ Read and write to this asynchronously

→ Locks / semaphores must be used to control access to shared memory (synchronisation)

Advantage → no concept of "ownership" of data
Everyone has everything.

Disadvantages → concept of locality <u>hidden</u> ⇒ difficult to manage (e.g. cache coherence)

→ also can be more difficult to make deterministic (due to asynch write to memory)

1.4 Parallel Algorithm Examples

1.4.1 1-D finite differences

Update array values by

$$x_i^{(t+1)} = \frac{x_{i-1}^{(t)} + 2x_i^{(t)} + x_{i+1}^{(t)}}{4} \qquad t = 0, \ldots, T$$

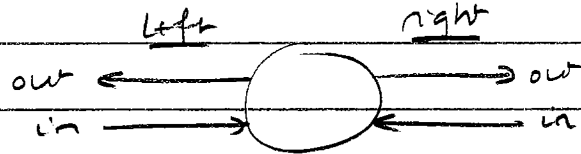
($t=0$: initial value; $\ldots, T$: steps)

$x_i, \quad i = 1, \ldots, 8$ ← 8 points

Parallel algorithm: Assign tasks →

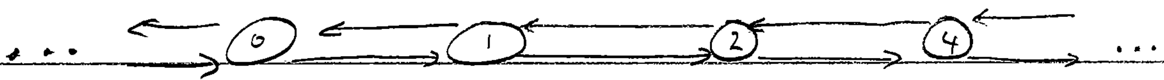
8 tasks — 1 for each x

i^{th} task — update x_i from $t=0$ to $t=T$



or cleaner looking

ignoring "end effects"



Data transfer by channels "left" and "right"

At each step t, task i

1) send data $x_i^{(t)}$ on outports to left and right
2) receive $x_{i-1}^{(t)}$ from left $x_{i+1}^{(t)}$ from right inports
3) compute $x_i^{(t+1)}$

Exercise HW: convince yourself of this

⇒ N independent tasks

⇒ synchronisation enforced by blocking receive ⇒ **DETERMINISTIC**

(cannot do step $t+1$ until info from t received)

1.4.2 Pairwise Interactions

e.g. N-body forces

All pairwise interactions $I(x_i, x_j)$ $i \neq j$
on N data $\Rightarrow$ $N(N-1)$ interactions
$x_0, \ldots, x_{n-1}$

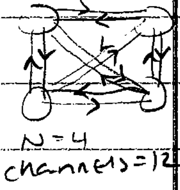
SKIP? PROBLEM?

(If interactions symmetric $I(x_i, x_j) = (\pm) I(x_j, x_i)$
$\Rightarrow$ $N(N-1)/2$)

Parallel algorithm:

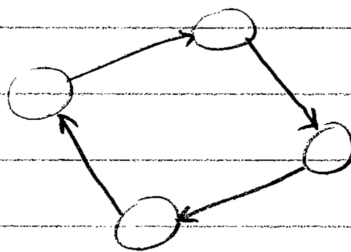
N tasks : task i for x_i
computes $I(x_i, x_j)$ $j = 0, \ldots n-1$
$j \neq i$

Channels :



niaive : each x_i interacts with each x_j
$\Rightarrow$ $N(N-1)$ channels

sensible : N channels !



unidirectional ring
Each task →
1 inport
1 outport

Task has two variable buffers

1/ datum buffer initialised with x_i
2/ accumulator

Then repeatedly (N-1) times

1/ send buffer on outport
2/ receive datum on inport into buffer
3/ interact contents of buffer with local x_i
4/ update local accumulator

(*) passes same # of messages just uses less channels & more local

SKIP or does ? PROBLEM:

If interactions are symmetric, can halve no. of communications and no. of interactions computed of course but requires refinement of channels:

→ link each task i to task $i + N/2$ as well
→ N more channels ($2N$ total)
→ continue as before, but each time

$I(x_i, x_j)$ x_i local datum
  x_j 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.

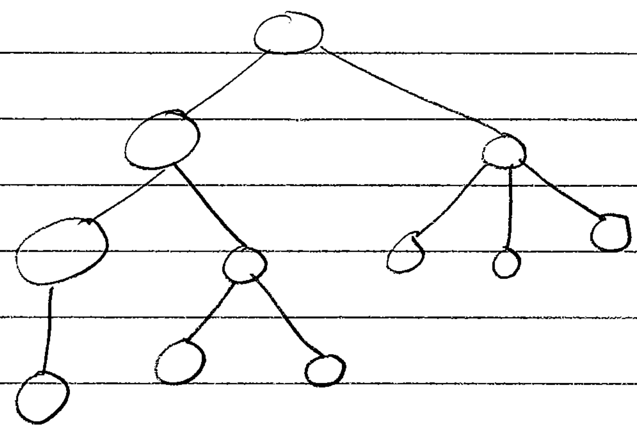
⇒ each $I(x_i, x_j)$ only calculated once //

1.4.3 Search

Example of dynamic task & channel allocation

Search tree that looks for nodes corresponding to a solution

```
procedure search(A)
begin
    if (solution(A)) then report solution
    else
        foreach (child A_i of A) search(A_i)
    end if
end.
```



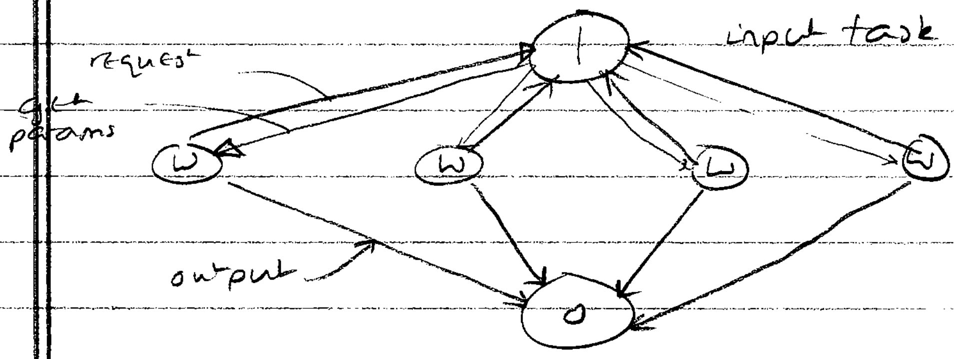
Initiates 1 task.
If solution stop
Else search children.

At any time, some tasks terminating at solns, 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 ⇒ (virtually) <u>no</u> communication
⇒ <u>completely</u> 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 problems to each processor



input task: parameter values read in

Workers:
1, request param from ①
2, compute
3, output to ⓞ

Many-to-one request process ⇒ <u>NOT</u> DE<u>TERMINISTIC</u>
If processes 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.

→ notice also:
worker requests parameter then waits for it to arrive.
Can improve efficiency by PREFETCHING
i.e. 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 ratio of local to remote memory access ⇒ efficiency.
4/ Modularity – decomposition of complex entities into simpler components

B/ Parallel machine model → multicomputer SIMPLE REALISTIC
 = one or more von Neumann computers joined by interconnect

C/ Programming model → task/channel → provides abstractions that allows us to talk about 1→4