Parallel programming models

Von Neumann machine model:

- A processor and it's memory
- "program" = list of stored instructions
- Processor loads program (reads from memory), decodes, executes instructions (basic aritmetic, reads/writes memory, gets address of next instruction, …)

Big codes => keep track of

- Millions of memory locations
- Thousands of instructions

Parallel programs => complexity increases significantly => high-level abstractions vital. e.g. high-level languages to automatically translate concepts into nasty machine language

Further abstraction required for parallel programming to overcome extra complexity

Hierarchical, modular design of underlying work facilitates abstractions.

Main weapons

Our main weapons are therefore …

- **Concurrency**
- **Scalability**
- **Locality**
- **Modularity**

Amongst our weapons are such elements as …

Parallel programming abstractions

Need to facilitate the ***four main weapons***

And need to match the ***multicomputer/multiprocessor/hybrid*** machine models

Several programming models in common use:

- ✓ Shared memory
- ✓ Threads
- ✓ Message passing
- ✓ Data parallel
- ✓ Hybrid

Models ***are not*** and ***should not*** be specific to hardware. All should be able to be implemented on any hardware (theoretically).

e.g. shared memory programming on distributed memory hardware (KSR Allcache)

e.g. Message passing on shared memory (on SGI Origin)

Parallel abstractions: Task-Channel Model

TASK AND CHANNEL MODEL A general parallel abstraction of the multicomputer

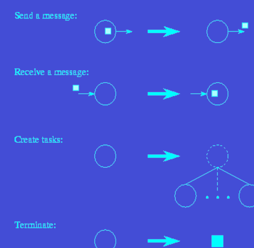
Tasks:

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***
- ✓ Number of tasks can ***vary*** during computation

In addition to reading/writing local memory, a task can perform 4 basic functions:

1. terminate
2. create new tasks
3. send information (an asynchronous operation: completes immediately)
4. receive information (a synchronous operation: blocks/waits until msg received)



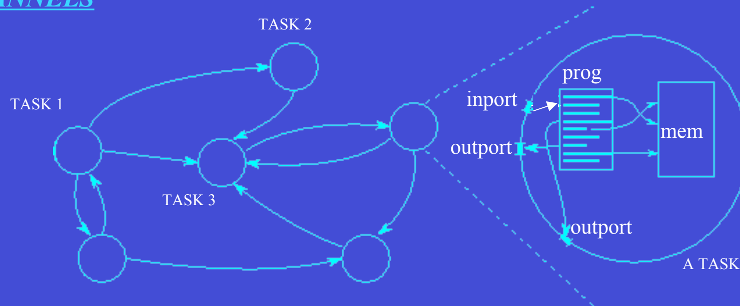
Parallel abstractions: Task-Channel Model

Tasks (continued):

Tasks have ***PORTS:***

- ***IN*port** – for receiving information
- ***OUT*port** – for sending information

Communications (actions 3 and 4 on previous) are achieved by connecting ports via ***CHANNELS***



Parallel abstractions: Task-Channel Model

Tasks (continued):

Tasks can be ***MAPPED*** into physical processors in numerous ways

- ✓ Single task per processor
- ✓ Multiple tasks to one processor
- ✓ Mapping does NOT affect semantics of program

Task abstraction provides concept of ***LOCALITY :***

- ✓ Data in task's local memory is "close"
- ✓ Data elsewhere is "remote"

Channel abstraction provides concept of ***DEPENDENCY:***

- ✓ One task needs data from another task in order to proceed

Parallel abstractions: Task-Channel Model

Example: Building a bridge out of steel girders

Two tasks:

- ✓ Making the girders out of steel at a foundry
- ✓ Building the bridge

One channel:

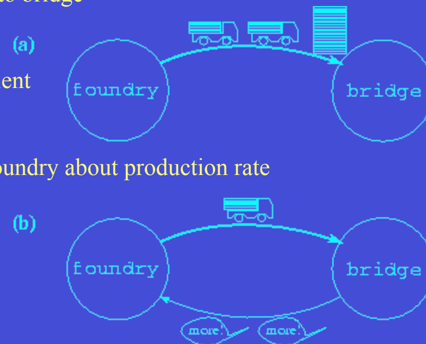
- ✓ Trucks transporting girders from foundry to bridge

Tasks can proceed independently if …

- ✓ Number of girders at the bridge are sufficient

Maybe need another channel:

- ✓Control channel: communicate BACK to foundry about production rate



Parallel abstractions: Task-Channel Model

Other properties of the Task-Channel Model:

MAPPING INDEPENDENCE

End result does ***not*** depend on the mapping:

- ✓ Tasks interact via channels *regardless of location of the task* so result does not depend on where task executes
- ✓ Algorithms can be designed *without concern for number of processors* on which they will execute

Often create ***more tasks than processors*** :

⇒ ***scalability*** (add more processors, algorithm does not change, just mapping)

⇒ Can ***mask communication delays*** by doing some computation (another task) while accessing remote data (overlapping communication and computation)

BUT …

Parallel abstractions: Task-Channel Model

Other properties of the Task-Channel Model:

PERFORMANCE

Two tasks communicating via a channel

- ✓ mapped to the same processor
- ✓ mapped to different processors

may => different efficiencies

Interprocessor communication generally less efficient than intra processor communication

Parallel abstractions: Task-Channel Model

Other properties of the Task-Channel Model:

MODULARITY

Modular =>

- ✓ pieces can be developed separately
- ✓ then incorporated into the complete program
- ✓ Can change module without changing other component
- ✓ Interaction between modules restricted to well-defined interfaces
- ✓ Reduces complexity
- ✓ Encourages code re-use

Task-Channel model encapsulates both data and code that operates on data and the inports/outports are the interface

⇒ Have advantages of modular design

Similar to object-oriented programming paradigm (tasks ~ objects, in that they capture data structures and code, BUT tasks => concurrency … and no inheritance blah blah)

Parallel abstractions: Task-Channel Model

Other properties of the Task-Channel Model:

DETERMINISM

Deterministic => a particular input ALWAYS yields a particular output

Determinism seems to be a good thing but is not always necessary.

In the Task-Channel model, determinism is ***likely*** if

- ✓ Each channel has a single sender and a single receiver (one-to-one)
- ✓ Receiving channel blocks (waits) until operation completes

e.g. Bridge construction. Would be nice to get the same bridge regardless of rates of girder and bridge construction! (determinism important here)

If girder production is too fast, girders simply accumulate ready to be used (send does not need to be blocked)

If bridge assembly crew are too fast and run ahead of girder production, they must block (wait) for more girders to arrive rather than use half-made girders! (receive needs to be blocked so co-ordinates with a send)

If multiple bridges are being built at the same time, all is fine as long as the girders destined for different bridges travel on different channels (trucks)

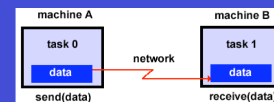
Parallel abstractions: Other models

1. Task-Channel model is an example of ***TASK PARALLELISM***

⇒ different ***tasks*** execute concurrently

MESSAGE-PASSING

is a ***subset*** of the task parallel Task-Channel programming model:



- ✓ Most widely used parallel programming model
- ✓ Each task has a unique ID
- ✓ Instead of specifying channels, specify sender ID and receiver ID
- ✓ e.g. instead of “send on channel 5” -> send to task 10
- ✓ Message-passing is actually more restrictive:
  - ⇒ does not preclude dynamic creation of tasks, multiple task per process
  - ⇒ But in practice, most message-passing programs tend to create a fixed number of identical tasks – this is SPMD mode (more in a moment).

Implementation: Target – MIMD. Library of subroutines that you embed in source code. MPI, PVM.

Parallel abstractions: Other models

2. DATA PARALLELISM

Set of tasks works collectively on a partition of the data

Concurrency derives from doing ***same*** operation over and over again on ***different*** data

e.g. add 2 to all elements of an array

e.g. increase the salary of all associate professors ☺

Not so different from previous?

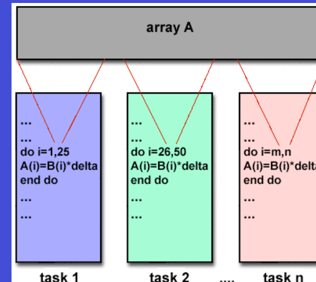
✓ Task-Channel can be applied

✓ Can think of as a separate task for each data element

✓ Much finer granularity version of Task-Channel

✓ Need to provide info on how to divide data into tasks (multicomputer, distributed memory)

× No natural concept of locality ☹



Implementation: Target – SIMD. Libraries, compiler directives. HPF (-> Fortran 95). On distrib mem, usually invisibly converts to MPI!)

Parallel abstractions: Other models

3. SHARED ADDRESS SPACE

("address" => programming model, "memory" => hardware architecture. Remember should work on ANY hardware)

Tasks share a common address space

Read and write to this address space asynchronously

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

Advantage: no concept of "ownership" of memory/data.

⇒ everyone has everything

⇒ simplified program development

Disadvantage: concept of locality is hidden

⇒ difficult to manage (e.g. cache coherence)

⇒ difficult to make deterministic (due to asynchronous writing to memory)

Implementation: Target - shared memory machines. Compilers translate variables to global memory addresses. No common distributed memory implementations currently exist.

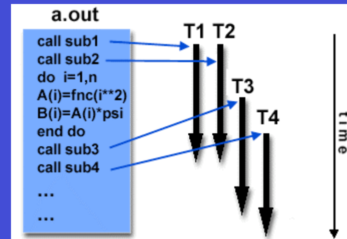
Parallel abstractions: Other models

3. THREADS

Single process with multiple, concurrent execution paths

Process runs in serial mode then forks to produce multiple threads that can run concurrently

- ✓ Think of thread as a subroutine
- ✓ Each thread has local data
- ✓ Each thread has access to global resources
- ✓ Threads communicate through global memory
- ✓ Requires synchronisation
- ✓ Threads come and go but main program remains.



Implementation: Target – shared memory machines. Libraries of subroutines, compiler directives. Standards – POSIX threads (Pthreads, C only, library); OpenMP (Fortran, C, C++, compiler directives)

Parallel abstractions: Other models

3. HYBRID

e.g. Combination of message-passing model (MPI) with either threads model (Pthreads or OpenMP) might lend itself well to hybrid machines that are clusters of SMPs (although not yet proven to be better than plain MPI ☺)

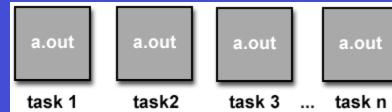
e.g. Combination of data parallel and message-passing. Data parallel languages e.g. HPF might use MPI transparently to the programmer.

Parallel abstractions: Other models

4. SPMD

SPMD – Single Program Multiple Data

(Note: MIMD/SIMD etc are hardware paradigms)



High-level programming model that can be built ***on top of*** any of the previous constructs

- ✓ Single program is executed by all tasks
- ✓ SPMD programs have branched logic OR execution dependent on LOCATION => at any point, tasks can be executing same or different instructions within the same program
- ✓ Tasks execute on different data

SPMD with conditional branching on DMP ~ task parallel

SPMD with no conditional branching, and data partition on DMP ~ data parallel

SPMD with conditional branching on SMP ~ threads

SPMD with no conditional branching, and data partition on SMP ~ shared address

Parallel abstractions: Other models

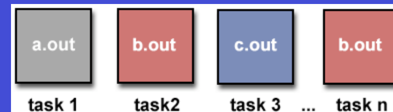
5. MPMD

MPMD – Multiple Program Multiple Data

Also can be built on top of the previous models

Multiple executables (programs)

All tasks use different data.



Parallel algorithm examples

1. 1-D Finite-Differences

Update array values

$x_i^{(t+1)} = (x_{i-1}^{(t)} + 2\, x_i^{(t)} + x_{i+1}^{(t)}) /4$ t=0,1,2, …, T , i = 1,2,…, N

Parallel algorithm examples

1. 1-D Finite-Differences

Update array values

$x_i^{(t+1)} = (x_{i-1}^{(t)} + 2\, x_i^{(t)} + x_{i+1}^{(t)}) /4$ t=0,1,2, …, T , i = 1,2,…, N

Tasks:

1 for each data point

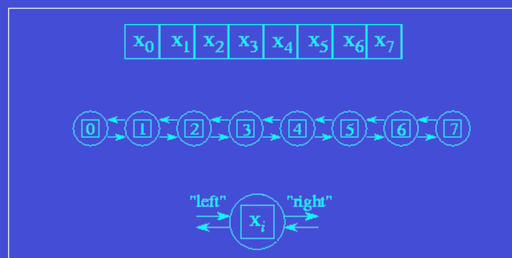
i^{th} task = update x_i

Channels:

Each task has 2 inports (left/right) and 2 outports (left/right)

Task logic:

1. Send my x to left and right
2. Receive data from left and right
3. Compute update



⇒ **N independent tasks**

⇒ **Synchronisation enforced by blocking receive**

⇒ **Deterministic**

Parallel algorithm examples

2. Pairwise interactions

Calculate some function (interaction) of two variables I(x(i),x(j)) for all pairs of variables in an array of variables x(i), i=1,…,N

⇒ N*(N-1) total interactions

Challenge: use only N channels!

Parallel algorithm examples

2. Pairwise interactions

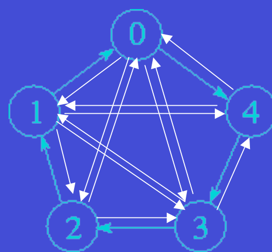
Calculate some function (interaction) of two variables I(x(i),x(j)) for all pairs of variables in an array of variables x(i), i=1,…,N

⇒ N*(N-1) total interactions

e.g. I(x1,x2) = x1+x2

Niaive: all-to-all,

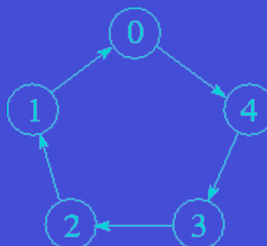
N*(N-1) channels



Smarter:

Uni-directional ring

N channels



Task logic:

Each task responsible for computing interactions with a particular x(i) (my_xi). Perform:

Initialise send buffer with my_xi

Do N-1 times:

Send buffer

Receive buffer

Interact buffer with local my_xi and store result

End Do

(same no of total communications, less channels, more local)

Parallel algorithm examples

3. Search

Search that explores a tree-structured database looking for nodes that are "solutions".

e.g. nameserver looking for an IP address

Dynamical task allocation!

Task logic:

procedure search(A)

If (solution(A) Then report_solution_found

Else

Foreach (child B of A) Begin

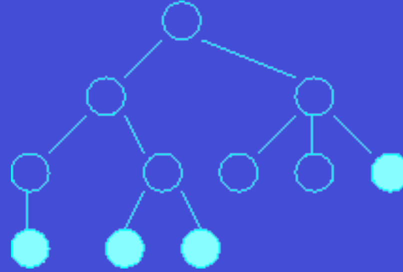
Create new task

Create return_report_channel

search(B)

End

End If



At any one time:

⇒ child tasks terminating at solutions and reporting

⇒ parent tasks waiting for reports

⇒ new tasks/channels being created (in a wavefront)

Parallel algorithm examples

4. Parameter study

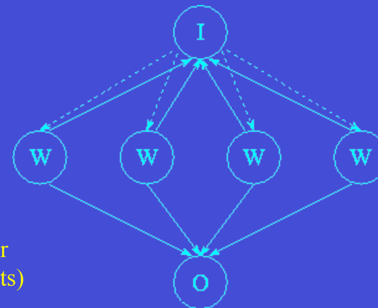
Example of ***embarrassingly parallel***

⇒ Virtually no communication; completely independent problems

e.g. same computation, different parameters only.

Each task = a computation with it's parameters

Assign (subset, probably 1, of) tasks to each processor and allocate input (parameters) and output (results) processors.



Workers task logic:

Request parameters from input processor

Compute

Output to output processor

Anything interesting??

Note:

1. Many-to-one communication structure for input and output

⇒ ***NOT DETERMINISTIC***! (not big deal here, only changes order of results output)

2. Workers must wait for parameters. ***Prefetching*** would help. Comm/comp overlap

Chapter 1 Summary

Four desirable attributes of parallel algorithms and software:

1. Concurrency – ability to perform actions simultaneously
2. Scalability – resilience to changing (increasing) processor counts
3. Locality – high ration of local to remote memory access => efficiency
4. Modularity – decomposition of complex entities into simpler components

Parallel machine model of major interest: MULTICOMPUTER

= one or more von Neumann computers connected together by an interconnect

- ✓ Simple
- ✓ Realistic

Programming model: TASK-CHANNEL

- ✓ Provides simple abstractions that allow us to talk about the desirable attributes.