Chapter 1
Chapter 1

1. **Parallel machine models**  
   (hardware abstractions)

2. **Parallel programming models**  
   (software abstractions)
Chapter 1

• Overview of
  – parallel machines (~hardware)
    ➢ Von Neumann
    ➢ SISD, SIMD, MISD, MIMD
    ➢ Shared memory; Distributed memory
  – parallel programming models (~software)
    ➢ Shared memory
    ➢ Shared address space
    ➢ Message passing
    ➢ Data parallel
    ➢ Hybrid: Clusters of SMPs or GPUs
    ➢ Grid

• Note: Confusion!
  Parallel machine models may or may not be tightly coupled to programming model!
  – Historically, tight coupling (new machine => new programming model!)
  – Today, not so much; abstraction for portability is important
There are many ways of classifying parallel machine architectures.

For completeness, start with *old skool* look at the world ....

Characterisation by **CONTROL MECHANISM**, in particular
- Instruction stream
- Data stream
Parallel Machine Models: von Neumann

von Neumann computer

- (named after Hungarian mathematician John von Neumann)
- “stored program” concept
- Memory stores both program and data
- Program consists of coded instructions for operation
- Control unit fetches instructions from memory, decodes, and performs SEQUENTIALLY on data also fetched from memory
- Arithmetic unit performs basic ops
- Stuff can be imported or exported to memory from the outside (loaded)

ALL COMPUTERS FOLLOW THIS BASIC DESIGN, EVEN PARALLEL!
Parallel machine models

Characterisation by CONTROL MECHANISM in particular
- Instruction stream
- Data stream

=> FLYNN’S TAXONOMY (1966)
Flynn’s Taxonomy: **SISD**

**Single Instruction, Single Data:**

- A serial (non-parallel) computer!
- Single instruction: only one instruction stream is being acted on by the CPU during any one clock cycle
- Single data: only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest and until recently, the most prevalent form of computer
- Examples: old PCs, single CPU workstations and mainframes
Flynn’s Taxonomy: **SIMD**

**Single Instruction, Multiple Data:**

- **Parallel computer:** Data parallel
- **Single instruction:** All processing units execute the same instruction at any given clock cycle
- **Multiple data:** Each processing unit can operate on a different data element
- This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units.
- Best suited for specialized problems characterized by a high degree of regularity, such as image processing.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines
- Examples:
  - Processor Arrays: Connection Machine CM-2, Maspar MP-1, MP-2
  - Vector Pipelines: IBM 9000, Cray C90, Fujitsu VP, NEC SX-2
- These were SIMD machines
- **Now SIMD at chip level:**
  - SSE instructions
  - GPUs are *sorta* SIMD
Flynn’s Taxonomy: MISD

Multiple Instruction, Single Data:

- A single data stream is fed into multiple processing units.
- Each processing unit operates on the same data independently using independent instruction streams.
- Very few actual examples of this class of parallel computer have ever existed. One is the experimental Carnegie-Mellon C.mmp computer (1971).
- Some conceivable uses might be:
  - multiple frequency filters operating on a single signal stream
  - multiple cryptography algorithms attempting to crack a single coded message.
- IGNORE!
Flynn’s Taxonomy: MIMD

Multiple Instruction, Multiple Data:

- **Multiple Instruction:** every processor may be executing a different instruction stream
- **Multiple Data:** every processor may be working with a different data stream
- **TASK parallel**
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Currently, the most common type of parallel computer: Most modern computers fall into this category.
- **Examples:** most current supercomputers, networked parallel computer "grids" and multi-processor SMP computers - including some types of PCs.
- But note many MIMD machines contain SIMD sub-components now
Parallel machine models

MIMD basically “won”!

These days it then becomes more useful to characterise machines by MEMORY STRUCTURE

Possibilities:
- Shared memory
- Distributed memory
- Hybrid
Parallel Machine Models: Shared Memory

- Multiple processors operate independently but access all memory via a global address space
- Changes in a memory location visible to all processors
- UMA – uniform memory access
  - Everything identical
  - Equal memory access
  - SMP (symmetric multiprocessor)
- NUMA – non-uniform
  - Unequal access to memory
  - Linked SMPs

- User friendly
- Fast access: O(1) connectivity
- Lack of scalability (buses)
- Synchronisation – cache coherence
- Expense for large machines
Parallel Machine Models: Distributed Memory

- Processors have local memory, no global address space
- Require communication network to connect processor memory
- Data from one processor must be communicated to another if required: message passing
- Synchronisation/coherence is programmer’s responsibility
- Network fabric varies greatly! (At this point we could go into a long aside about network hardware topologies which I think I will leave alone for the moment 😃)

✔ VERY scalable
✔ LOCAL memory access rapid
✔ No cache coherence issues
✔ Use commodity processors
✗ Lot of programmer responsibility!
✗ NUMA times
Parallel Machine Models: Hybrid Memory

- Biggest machines today have both!
- Shared memory component can be
  - (cc-UMA) SMP
  - GPU
- Machine is a distributed network of these
- Advantages and disadvantages are those of the individual parts!
  - Scalability of distributed
  - Speed and simplicity of memory access within SMP
  - BUT increased complexity for programmer
Generic modern machine model

![Diagram of a modern machine model with multiple processors (Proc) connected to an interconnection network, with memories (Memory) linked to the processors.](image-url)
Parallel machine models:

Old skool:
- Von Neumann
- SISD
- SIMD
- MISD
- MIMD

New skool:
- Shared memory
- Distributed memory
- Hybrid
Terminal terminology!

Core – logical execution unit comprising of L1 cache and functional units.

Chip – physical integrated circuit (IC) execution unit. Chips can be single-core or multi-core.

Processor/Central Processing Unit (CPU) – context specific! Can either be a chip or a core! Confusing.

Sockets – physical connector on a motherboard that accepts a single physical chip. Many motherboards/nodes have multiple sockets that in turn can accept multi-core chips.

Node – motherboard containing multiple sockets and therefore possibly multiple possibly multi-core chips.

e.g. grape: node = Dell PowerEdge R420 with 2 sockets; each socket has an Intel Xeon SandyBridge E5-2470 processor/chip which contains 8 cores
Parallel programming models
Parallel programming models

Von Neumann machine model:

- A processor and its memory
- "program" = list of stored instructions
- Processor loads program (reads from memory), decodes, executes instructions (basic arithmetic, 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

Parallel programming models exist as an abstraction above hardware and memory architectures

- Although not immediately apparent, these are NOT specific to a particular type of machine. Can be implemented on ANY hardware.
  - e.g. shared memory programming on distributed memory hardware (KSR Allcache)
  - e.g. Message passing on shared memory (on SGI Origin)
- No “best”, although there are some better implementations of some than others.
Hierarchical, modular design facilitates abstractions.

We already know that some of the things we are looking for are:

- **Concurrency** – things that can be done in parallel
- **Scalability** – things that can be scaled up to large numbers of processors
- **Locality** – remote memory access is always expensive!
- **Modularity** – plug and play = easy programming

Our 4 main weapons are such elements as …
Parallel programming abstractions

Can think about different abstractions in terms of:

- **Control**
  - How is parallelism created?
  - What orderings exist between operations?
- **Data**
  - What data is private vs. shared?
  - How is logically shared data accessed or communicated?
- **Synchronization (communication)**
  - What operations can be used to coordinate parallelism?
  - What are the atomic (indivisible) operations?
- **Cost (locality)**
  - How do we account for the cost of each of the above?
  - Locality
Parallel programming abstractions

Need to facilitate the *four main weapons*

Need to match the *multicomputer/multiprocessor/hybrid* machine models

**Several programming models in common use:**

- Shared memory/address
- Threads
- Message passing
- Data parallel
- Hybrid
Parallel abstractions: Task-Channel Model

**TASK AND CHANNEL MODEL:** The most general parallel abstraction of the multicomputer

**Tasks:** A sequential piece of computation and its local memory (i.e. a virtual von Neumann computer)

- A parallel computation consists of one or more tasks
- Tasks can 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 *IS* 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

Task-Channel model is an very basic example of

1. **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 local memory.
- Multiple tasks can reside on same physical machine
- Task communicate by sending messages
  - Communication is a co-operative protocol (sends need an associated receive)
  - 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 from my_task to task 10
- Message-passing is actually more restrictive:
  - does not actually preclude dynamic creation of tasks, multiple task per process (in the standard)
  - 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. **Message Passing Interface (MPI) has become the standard** (lots more later!!)
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

\[ \text{e.g. add 2 to all elements of an array} \]

Not so different from previous?

✓ Task-Channel can be applied
✓ Address space is global
✓ Tasks work collectively on same data structure
✓ Need to provide info on how to divide data into tasks
✓ Can think of as a separate task for each data element
✓ Much finer granularity version of Task-Channel
× No natural concept of locality


HPF & UPC are SPMD implementations. 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 (memory could be physically shared or distributed)

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.

Distributed shared memory implementations require special hardware and software (e.g. Cray T3D shmem; Co-Array Fortran).
4. **THREADS**  (subset of shared address space really)

**Single** ("heavyweight") process with **multiple, concurrent** ("lightweight") execution paths

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

- Heavyweight process acquires all resources
- Some serial work then creates sub-tasks = threads
- 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

5. 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. Threads perform computationally-intensive kernels on local data.

e.g. Combination of data parallel and message-passing: MPI with GPU programming
Parallel abstractions: Other models

6. **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

*MOST FLEXIBLE MODEL => MOST COMMONLY USED MODEL?*
Parallel abstractions: Other models

7. MPMD

MPMD – Multiple Program Multiple Data

Also can be built on top of the previous models

Multiple executables (programs)

All tasks use different data.
GPU’s big performance opportunity is data parallelism
- Most programs have a mixture of highly parallel operations, and some not so parallel
- GPUs provide a threaded programming model (CUDA) for data parallelism to accommodate both
- Current research attempting to generalize programming model to other architectures, for portability (OpenCL)
- Guest lecture later in the semester?

Cloud computing lets large numbers of people easily share $O(10^5)$ machines
- MapReduce was first programming model: data parallel on distributed memory
- More flexible models (Hadoop…) invented since then
- Guest lecture later in the semester?
Terminal terminology!

Core – logical execution unit comprising of L1 cache and functional units.

Chip – physical integrated circuit (IC) execution unit. Chips can be single-core or multi-core.

Processor/Central Processing Unit (CPU) – context specific! Can either be a chip or a core! Confusing.

Sockets – physical connector on a motherboard that accepts a single physical chip. Many motherboards/nodes have multiple sockets that in turn can accept multi-core chips.

Node – motherboard containing multiple sockets and therefore possibly multiple possibly multi-core chips.

e.g. grape: node = Dell PowerEdge R420 with 2 sockets; each socket has an Intel Xeon SandyBridge E5-2470 processor/chip which contains 8 cores

Process – independent program with its own stack memory running on a computer. MPI runs multiple processes.

Thread – process that essentially does not have a full stack of memory, i.e. it is tied to a parent process. Threads run on the same computer or must at least share an address space (e.g. cores in a multi-core chip). OpenMP runs multiple threads.

Hyperthreading – makes a single core look like multi-core, or more cores than physically exist. OS schedules more than one process at a time.
1. **Parameter study**

Example of *embarrassingly parallel*

⇒ (Virtually?) no communication; completely independent problems

e.g. same computation, different parameters only.

Task? = a computation with it’s parameters

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

Channels? : None!

**Workers task logic:**
Request parameters from input processor
Compute
Output to output processor

---

**Anything interesting??**

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

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

2. 1-D Finite-Differences

Update array values

\[ x_i^{(t+1)} = \frac{(x_{i-1}^{(t)} + 2x_i^{(t)} + x_{i+1}^{(t)})}{4} \]

\[ t=0,1,2, \ldots, T \]

\[ i = 1,2, \ldots, N \]
2. **1-D Finite-Differences**

Update array values

\[ x_i^{(t+1)} = \frac{(x_{i-1}^{(t)} + 2x_i^{(t)} + x_{i+1}^{(t)})}{4} \]

\[ t=0,1,2, \ldots, T, \quad i = 1,2,\ldots, N \]

**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

**Deterministic?**

⇒ iff synchronisation enforced by blocking receive
3. **Pairwise interactions**

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

\[ \Rightarrow N \times (N-1) \text{ total interactions} \]

Challenge: use only \( N \) channels!
3. Pairwise interactions

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

$\Rightarrow$ $N*(N-1)$ total interactions

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

**Naive**: all-to-all,

$N*(N-1)$ channels
Parallel algorithm examples

3. Pairwise interactions

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

$\implies N \cdot (N-1)$ total interactions

e.g. $I(x_1, x_2) = x_1 + x_2$

**Naive:** all-to-all, $N \cdot (N-1)$ 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)
3. **Pairwise interactions**

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

\[ \Rightarrow N(N-1) \text{ total interactions} \]

e.g. \( I(x_1, x_2) = x_1 + x_2 \)

**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

4. 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)
5. (a) Simple example: Sum evaluation – Shared memory

- Consider applying a function $f$ to the elements of an array $A$ and then computing its sum:

\[
\sum_{i=0}^{n-1} f(A[i])
\]

$A = $ array of all data

$fA = f(A)$

$s = \text{sum}(fA)$
5. (a) **Simple example: Sum evaluation – Shared memory**

**Remember: Determining questions:**
- Where does $A$ live? All in single memory? Partitioned?
- What work will be done by each processors?
- How do they need to coordinate to get a single result?
- Locality?

- Shared memory: small number $p << n=\text{size}(A)$ processors attached to single memory

**Strategy**
- Parallel Decomposition:
  - Each evaluation and each partial sum is a task.
- Assign $n/p$ numbers to each of $p$ procs
  - Each computes independent “private” results and partial sum.
  - Collect the $p$ partial sums and compute a global sum.
Parallel algorithm examples

5. (a) Simple example: Sum evaluation – Shared memory

```c
fork(sum,a[0:n/2-1]); sum(a[n/2,n-1]);
```

```c
static int s = 0;
```

High level code

**Thread 1**

```c
for i = 0, n/2-1
s = s + f(A[i])
```

**Thread 2**

```c
for i = n/2, n-1
s = s + f(A[i])
```

What is the problem with this program?

Two Classes of Data:

- Logically Shared
  - The original n numbers, the global sum.
- Logically Private
  - The individual function evaluations.
  - What about the individual partial sums?
Parallel algorithm examples

5. *(a) Simple example: Sum evaluation – Shared memory*

```
static int s = 0;
fork(sum,a[0:n/2-1]); sum(a[n/2,n-1]);
```

High
level code

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>for i = 0, n/2-1</td>
<td>for i = n/2, n-1</td>
</tr>
<tr>
<td>s = s + f(A[i])</td>
<td>s = s + f(A[i])</td>
</tr>
</tbody>
</table>

What is the problem with this program?

A *race condition* or *data race* occurs when:

- Two processors (or two threads) access the same variable, and at least one does a write.
- The accesses are concurrent (not synchronized) so they could happen simultaneously.
5. *(a) Simple example: Sum evaluation – Shared memory*

Example: Assume $A = [3,5]$, $f(x) = x^2$, and $s=0$ initially

$$A = \begin{bmatrix} 3 & 5 \end{bmatrix} \quad f(x) = x^2$$

**Machine code**

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>....</td>
<td>...</td>
</tr>
<tr>
<td>compute $f([A[i]])$ and put in reg0</td>
<td>compute $f([A[i]])$ and put in reg0</td>
</tr>
<tr>
<td>reg1 = s</td>
<td>9</td>
</tr>
<tr>
<td>reg1 = reg1 + reg0</td>
<td>0</td>
</tr>
<tr>
<td>s = reg1</td>
<td>9</td>
</tr>
<tr>
<td>...</td>
<td>9</td>
</tr>
</tbody>
</table>

Answer = 9!
5. (a) Simple example: Sum evaluation – Shared memory

Example: Assume $A = [3, 5]$, $f(x) = x^2$, and $s=0$ initially

$A= \begin{bmatrix} 3 & 5 \end{bmatrix}$  \hspace{1cm} f(x) = x^2

Machine code

Thread 1

....
compute $f([A[i]]$ and put in reg0
reg1 = s
reg1 = reg1 + reg0
s = reg1
....

Thread 2

....
compute $f([A[i]]$ and put in reg0
reg1 = s
reg1 = reg1 + reg0
s = reg1
....

Answer=25!
Parallel algorithm examples

5. (a) Simple example: Sum evaluation – Shared memory

Example: Assume $A = [3, 5]$, $f(x) = x^2$, and $s=0$ initially

\[
\begin{array}{c}
A = \begin{bmatrix} 3 & 5 \end{bmatrix} \\
\end{array}
\]
\[
\begin{array}{c}
f(x) = x^2 \\
\end{array}
\]

Machine code

\[
\begin{array}{ll}
\text{Thread 1} & \text{Thread 2} \\
\text{compute } f([A[i]]) \text{ and put in reg0} & 9 \\
\text{reg1} = s & 0 \\
\text{reg1} = \text{reg1} + \text{reg0} & 9 \\
\text{s} = \text{reg1} & 9 \\
\text{...} & \text{...} \\
\end{array}
\]

\[
\begin{array}{ll}
\text{...} & \text{...} \\
\end{array}
\]

Answer=34!
5. **Simple example: Sum evaluation – Shared memory**

Example: Assume \( A = [3, 5] \), \( f(x) = x^2 \), and \( s=0 \) initially

\[
A = \begin{bmatrix} 3 & 5 \end{bmatrix} \quad f(x) = x^2
\]

The **atomic** operations are reads and writes

+= operation is *not* atomic

All computations should happen in (private) registers

---

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>compute ( f([A[i]]) ) and put in reg0</td>
<td>compute ( f([A[i]]) ) and put in reg0</td>
</tr>
<tr>
<td>reg1 = s</td>
<td>reg1 = s</td>
</tr>
<tr>
<td>reg1 = reg1 + reg0</td>
<td>reg1 = reg1 + reg0</td>
</tr>
<tr>
<td>( s = \text{reg1} )</td>
<td>( s = \text{reg1} )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

So … for this program to work, \( s \) should be \( 3^2 + 5^2 = 34 \) at the end
but it may be 34, 9, or 25!!
5. (a) Simple example: Sum evaluation – Shared memory

Thread 1

```
local_s1 = 0
for i = 0, n/2-1
    local_s1 = local_s1 + f(A[i])
    lock(lk);
    s = s + local_s1
    unlock(lk);
```

Thread 2

```
local_s2 = 0
for i = n/2, n-1
    local_s2 = local_s2 + f(A[i])
    lock(lk);
    s = s + local_s2
    unlock(lk);
```

Better?

- Since addition is associative, it’s OK to rearrange order
- Most computation is on private variables
- Sharing frequency is also reduced, which might improve speed

Still got race condition on write!

Add locks …
Parallel algorithm examples

5. (a) Simple example: Sum evaluation – Shared memory

Here is the code for calculating the sum of a function `f` applied to elements of an array `A` using parallel threads in shared memory:

```c
static int s = 0;
static lock lk;

Thread 1
local_s1 = 0
for i = 0, n/2-1
    local_s1 = local_s1 + f(A[i])
    lock(lk);
    s = s + local_s1
    unlock(lk);

Thread 2
local_s2 = 0
for i = n/2, n-1
    local_s2 = local_s2 + f(A[i])
    lock(lk);
    s = s + local_s2
    unlock(lk);
```

Why not do lock inside loop in original program?

```c
Thread 1
for i = 0, n/2-1
    lock(lk);
    s = s + f(A[i])
    unlock(lk);

Thread 2
for i = n/2, n-1
    lock(lk);
    s = s + f(A[i])
    unlock(lk);
```

Makes code sequential! PLUS lock overheads!
5. (b) Simple example: Sum evaluation – Message Passing

Same Questions:
- Where does A live? All in single memory? Partitioned?
- What work will be done by each processors?
- How do they need to coordinate to get a single result?

• Distributed memory: small number $p << n=\text{size}(A)$ processors SAME, BUT each with its own memory now => all variables are “private”

Strategy:
• Parallel Decomposition SAME:
  - Each evaluation and each partial sum is a task.
• Assign $n/p$ numbers to each of $p$ procs SAME
  - Each computes independent “private” results and partial sum.
  - Collect the $p$ partial sums and compute a global sum: DIFFERENT:
    • send the partial sums to the other processors
    • Receive from neighbours
    • Everyone calculates the full sum
Parallel algorithm examples

5. **(b) Simple example: Sum evaluation – Message Passing**

**Processor 1**

- $x_{local} = f(A[1])$
- send $x_{local}$, proc2
- receive $x_{remote}$, proc2
- $s = x_{local} + x_{remote}$

**Processor 2**

- $x_{local} = f(A[2])$
- send $x_{local}$, proc1
- receive $x_{remote}$, proc1
- $s = x_{local} + x_{remote}$

What happens if send/receive acts like the telephone system? **DEADLOCK!**
The post office? **OK!**  => *Different types of send/receives needed.*

**Second possible solution**

**Processor 1**

- $x_{local} = f(A[1])$
- send $x_{local}$, proc2
- receive $x_{remote}$, proc2
- $s = x_{local} + x_{remote}$

**Processor 2**

- $x_{local} = f(A[2])$
- receive $x_{remote}$, proc1
- send $x_{local}$, proc1
- $s = x_{local} + x_{remote}$

**BUTTTT … what if there are more than 2 processors?**
Parallel algorithm examples

5. (c) Simple example: Sum evaluation – Data parallel

Not all problems fit into this model!
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 ratio 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

More precise models: shared memory, distributed memory, data parallel (GPU etc) (+hybrids)

Programming model: TASK-CHANNEL

✓ Provides simple abstractions that allow us to talk about the desirable attributes
✓ More detailed models: shared address space, message passing, data parallel (+hybrids)
✓ Almost everything (in science world) is SPMD mode
Redo the task-channel model for the pairwise interaction problem but

- **Assume that the interactions are symmetric** i.e. \( I(x_1,x_2) = I(x_2,x_1) \) and therefore you don’t need to calculate both of them

- Make it so that no redundant interactions are calculated

- **Hint:** you can change the channel structure