CS3211

AY23/24 Sem 1

github.com/SeekSaveServe

Lectures

Introduction

L0 and L1

Program Parallelization

Decomposition: Decompose a sequential algorithm into tasks (programmer)

- Granularity of tasks are important
- Tasks have dependencies (data or control) between each other which defines the execution order

Scheduling: Assign tasks to processes (programmer / compiler)

Mapping - Map processes to cores (OS)

Von Neumann Computation Model instruction and data are stored in memory, and processors computes.

Memory Wall disparity between memory speed and processor speed ($\leq$ 1 ns VS $\geq$ 100 ns)

Processing unit refers to a core that can execute a kernel thread

Interconnect busses betwen different components in the machine

Node Machine in a distributed system

Why Parallel

Primary Reasons

1 OVercome limits of serial computing
2 Solve larger problems
3 Save (wall-clock) time

Other Reasons

- Take advantage of non-local resources
- Cost/energy saving - use multiple cheaper computing resourcees
- Overcome memory constraints

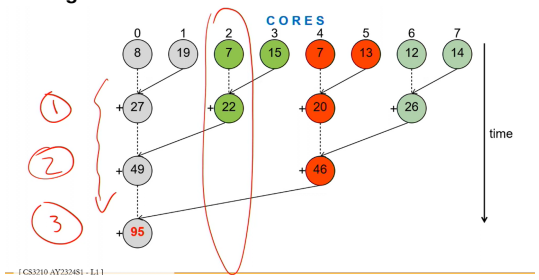
Computational Model Attributes

- **Operation mechanism** Primitive units of computation or basic actions of the computer on a specific Architecture
- **Data Mechanism** How we access and store data in address space
- **Control Mechanism** How primtive units of computation are scheduled
- **Communication Mechanism** Modes and patterns of exchanging information between parallel tasks (e.g message passing, shared memory)
- **Synchronization Mechanism** ensures to ensure needed information arrives at the right time

Dependencies and Coordination

- Dependencies among tasks impose constraints on scheduling
- Memory organizations: Shared-memory (threads), distributed-memory (processes)
- Coordination (synchronisation) imposes additional overheads

Two algorithms



- Core 0 is active throughout the execution
- Some cores are idle
- This is a lot better than having all cores idle while the master core is executing

Parallel Performance

- Execution time Vs Throughput
- Parallel execution time = computation time + parallelization overheads
- Overheads: Distribution of work(tasks) to porocesses, information exchange, synchronisation, idle time, etc

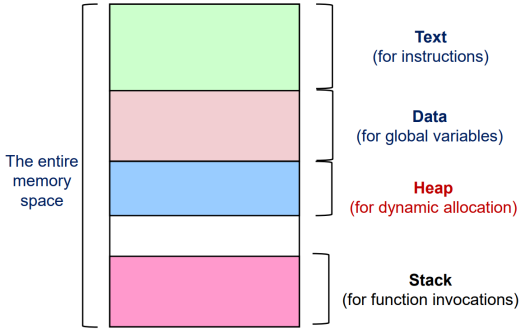
Background on Parallelism

L2: Processes and Threads

Process

- Identified by PID
- Program counter, global data (open files, network connections), stack or heap, current values of the registers (GPRs and Special)
- These information are abstracted in the PCB, and each proecss can be viewed as having exclusive access to tis address space
- Explicit communication is needed
- **Disadvantage**

1. High overhead of system calls
2. Potential re-allocation of data-structures
3. Communication goes through OS (system calls) and context switch is costly



Multi tasking

- Overhead: Context switching (PCB change) is needed and states of suspended process must be saved
- Time slicing: Pseudo-parallelism
- Child processes can use parent's data

Inter-process communication (IPC)

- Shared memory: need to protect access with locks
- Message passing: Blocking, unblocking, Synchronous, unsynchronous

Exceptions

- Executing a **machine level instruction** can cause exception
- For example: Overflow, Underflow, Division by Zero, Illegal memory address, Mis-aligned memory access

Interrupts

- External events can interrupt the execution of a program
- Usually hardware related: Timer, Mouse Movement, Keyboard Pressed etc

- **Synchronous**
  - Occur due to program execution
  - Have to execute an **exception handler**
- **Asynchronous**
  - Occur **independently** of program execution
  - Have to execute an **interrupt handler**

Threads

- A process may have multiple indepedent control flows called threads
- Each thread has its own stack and registers (PC, SP, registers), but share the same address space
- Shared memory model and Shared memory architecture
- Faster thread generation- no copy of address space
- Different process can be assigned to run on different cores of a multicore processor
- **User threads**
  - Managed by library
  - Context siwtch is fast, OS not involved
  - **Disadvantage**
    1. OS cannot map different threads of the same process to different resources $\Rightarrow$ No parallelism
    2. OS cannot switch to another thread if one thread blocks
- **Kernel threads**
- OS is aware of the threads and can manage accordingly
- Efficient in a multicore system
- Potential synchornisation issues

Many to one mapping

- All user-level threads mapped to one process.
- Efficiency depends on threading library

One to one mapping

- Each user-level thread is mapped to one kernel thread
- OS schedules

Many to many mapping

- Many user-level threads mapped to many kernel threads
- Library threads has overheads, and kernel threads has overheads
- At different points in time, different user threads are mapped to different kernel threads
- Number of threads must be suitable to the degree of parallism and the resources available

Locks

- Spinlock: busy wait
- Blocking: mutex
- Using more locks increasese the number of context switches
- DO NOT wait in the critical section

Semaphores

- Essentially shared global variables
- Can be potentially accessed anywhere in program
- No connection between semaphone and the data being protected

Barrier

- All threads must reach the barrier before any thread can proceed

Deadlock

- Deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set
- **iff these conds are met**

1. Mutual exclusion-at least one resource is not shareable
2. Hold and wait - at least one process holding a resource and waiting for another
3. No preemption - crticial section cannot be aborted externally
4. Circular wait

- **Dealing with deadlock**
- Ignore it, prevent it, avoid it by controlling resource allocation, detection and recovery by breaking cycles

Starvation

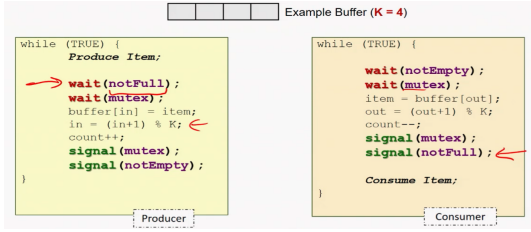
- Side effect of the scheduling algorithm. Lower priority processes might starve

Livelock

- Active acquire release but no useful work done

Producer-Consumer Problem

- Specifications:
  - Producers put in a shared bounded buffer if not full, consumers read from it if not empty
- Solution:



- Concurrent read, exclusive write. Categorical starvation of writer is possible

```
Writers                          Readers
roomEmpty.wait ()                mutex.wait ()
    #critical section for writers readers += 1
roomEmpty.signal ()              if readers == 1:
                                     roomEmpty.wait () # first in locks
                                 mutex.signal ()
                                 # critical section for readers

                                 readers -= 1
                                 if readers == 0:
                                     roomEmpty.signal () # last out unlocks
```

- Light switch: Abstracts out the shared lock for the reader

```
counter = 0
mutex = Semaphore (1)
lock (semaphore):
    mutex.wait ()
    counter += 1
    if counter == 1:
        semaphore.wait ()
    mutex.signal ()
unlock (semaphore):
    mutex.wait ()
    counter -= 1
    if counter == 0:
        semaphore.signal ()
    mutex.signal ()
```

- Starvation free solution (block out readers):

Writers	Readers
turnstile.wait ()	turnstile.wait ()
roomEmpty.wait ()	turnstile.signal ()
# critical section for writers	readSwitch.lock (roomEmpty)
turnstile.signal ()	# critical section for readers
roomEmpty.signal ()	readSwitch.unlock (roomEmpty)

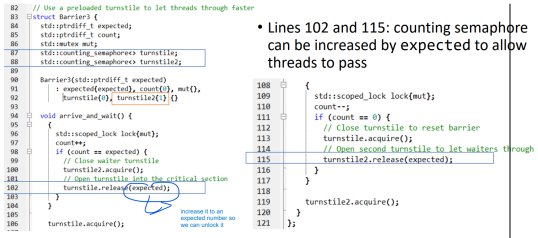
- Prioritise Writer:

```
readSwitch = lightswitch();
writeSwitch = lightswitch();
noReaders = semaphore(1);
noWriters = semaphore(1);

reader() {
/* Waiting for writers to be done */
wait(noReaders);
/* Writers cannot enter */
    readswitch.lock(noWriters);
signal(noReaders);
    # critical section
readSwitch.unlock(noWriters);
}

writer() {
/* Immediately acquires no readers so
    writers have priority */
writeSwitch.lock(noReaders);
    wait(noWriters);
    # critical section
    signal(noWriters);
writeSwitch.unlock(noReaders);
}
```

- This is implemented in C++ as a **shared_lock** and **unique_lock**
- GO has something similar: readLock and writeLock

Barrier

- All threads must stop at a common point before proceeding, can be reusable (barrier) or single use (latch)
- std::barrier, std::latch in C++
- E.g. std::barrier arrivalPoint(size) ... arrivalPoint.arrive_and_wait()
- sync.WaitGroup in GO is a latch, we can use 2 of them to make a barrier
- C++ implementation
  - The naive version fails because context switch can happen right before counter == N, which causes multiple threads (that were context switched out after counter ++) to signal the switch (another way to fail is to have 1 thread lap everyone else between the first barrier unlocks and second barrier unlocks)
  - The solution is to add a second turnstile (initialised as 1) to guard the turnstile1.signal, such that only one thread can signal it
  - But using mutex to increment turnstile1 one-by-one is slow
  - So we use a counting semaphore instead so we can raise the barrier by 1 thread!



- Lines 102 and 115: counting semaphore can be increased by expected to allow threads to pass

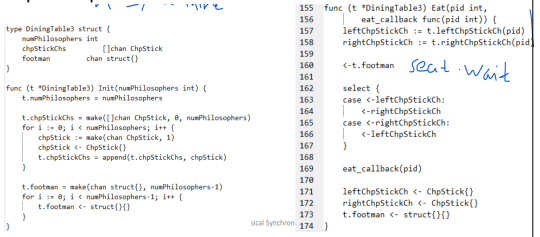
Dining Philosophers

- Specifications: N philosophers, N chopsticks
- Deadlock: All pick up left simultaneously
- Livelock: Put down left if right cannot be acquired
- Slap a mutex: Becomes sequential
- Scoped Lock(left, right): Acquire multiple mutexes in a deadlock free manner (deadlock avoidance), but as we have seen in CS3223, deadlock avoidance can lead to livelock
- GO's Mutex Free Solution: Use odd-even ring communication, odd numbered philosophers pick up left first, even numbered philosophers pick up right first
- This is the same as the right hander argument
- Tanenbaum's solution:

```
#define N 5
#define LEFT ((i+N-1)% N)
#define RIGHT ((i+1) % N)
#define THINKING 0
#define HUNGRY 1
#define EATING 2
int state[N];
Semaphore mutex = 1;
Semaphore s[N];

void philosopher( int i ){
    while (TRUE){
        Think( );
        takeChpStcks( i );
        Eat( );
        putChpStcks( i );
    }
}
void safeToEat(i)
{
    if( (state[i] == HUNGRY) &&
    (state[LEFT] != EATING) &&
    (state[RIGHT] != EATING) ) {
    state[i] = EATING;
    signal(s[i]);
    }
}

void takeChpSticks( i )
{
    wait(mutex);
    state[i] = HUNGRY;
    safeToEat(i);
    signal(mutex);
    wait(s[i]);
}
```

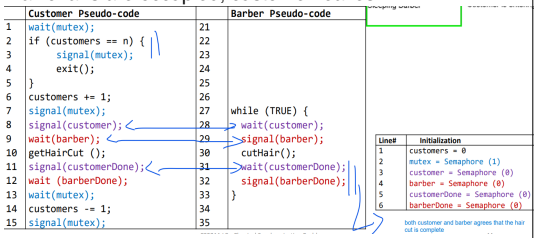
```
void putChpSticks(i)
{
    wait(mutex);
    state[i] = THINKING;
    safeToEat(LEFT);
    safeToEat(RIGHT);
    signal(mutex);
}
```

- Limited seats: Use a semaphore(N-1) to limit the number of philosophers that can eat at the same time

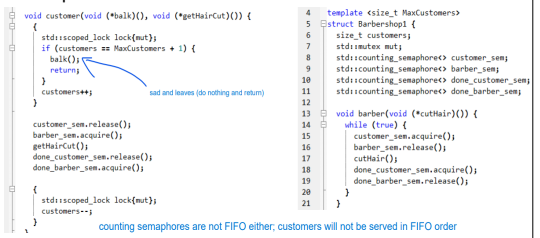


Barber Shop

- Barbershop consists of a waiting room with n chairs and the barber chair
- If there are no customers to be served, the barber goes to sleep
- If the barber is busy, but waiting room is available, customer seats on one of the chairs
- If barber is sleeping, customer wakes him up
- If all chairs are occupied, customer leaves



- line 31-32 are important since we need customer and barber need to agree that the haircut is done
- GO implementation:



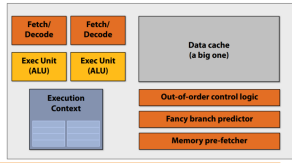
Architecture

L3: Processor and memory organization

Single Processor Parallelism

- Bit level - we work with word (multiple bits), data parallelism
- Instruction level (from same thread)

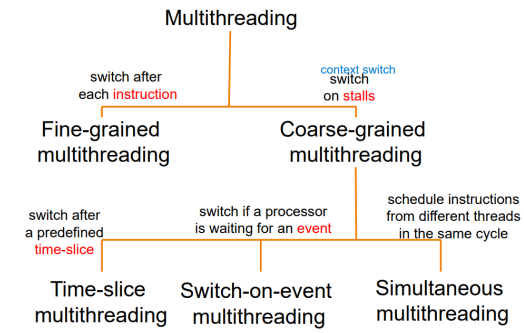
1. Pipelining - parallelism across time
   - Multiple instructions to occupy different stages in the same clock cycle - assuming no control or data dependencis
   - **Disadvanatges**
     1.1. Independence
     1.2. Bubbles - idle stages
     1.3. Data and control flow hazard
   - Wrong speculation of if-else branches can lead to wasted cycles
   - Synchronisation - need to preserve read-after-write
   - no more benefit to improving ILP now
2. Superscalar - parallelism across space



   - Duplicate pipelines and allow multiple instructions to pass through the same stage
   - Scheduling tough - which ones to execute together?
   - E.g Multiple ALUs
     2.1. Static - compiler decides
     2.2. Dynamic - hardware decides
- Thread level
  - Motivated by the limitation of ILP
  - SMT: Duplicate hardware context (PC, registers etc)
  - By convention, SMT is limited to 2 threads to reduce overhead and memory contention
  - Logical cores: hyperthreads

Processor level parallism

- Add more cores to processors to enable **multiple execution flows**
- Each core can be hyperthreaded
- Shared Memory
- Distributed Memory



Flynn's Taxonomy

- Describes parallel architecture based on instruction stream (execution flow - PC) and data stream

Single Instruction Single Data

- Single stream of instructions with each working on a single data

Single Instruction Multiple Data

- Single stream of instructions with each working on multiple data
- Exploit data parallelism (vector processor)

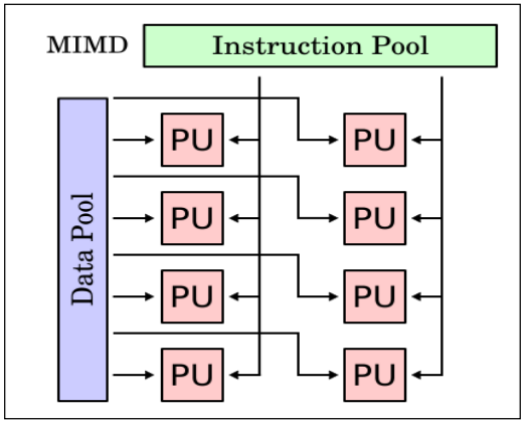
- Same instruction broadcasted to all ALUs
- AVX: intrinsic functions operatie on vectors of 4 64 bit values

Multiple Instruction Single Data

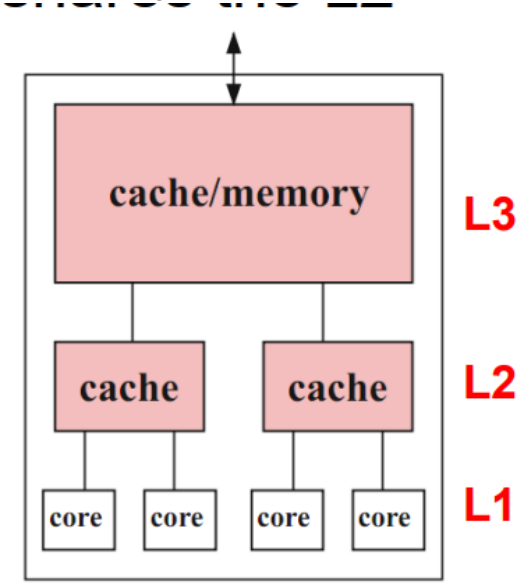
- Multiple instructions operating with a single data

Multiple Instructions Multiple Data

- Each PU fetches its own instructions
- Each PU operates its own data
-

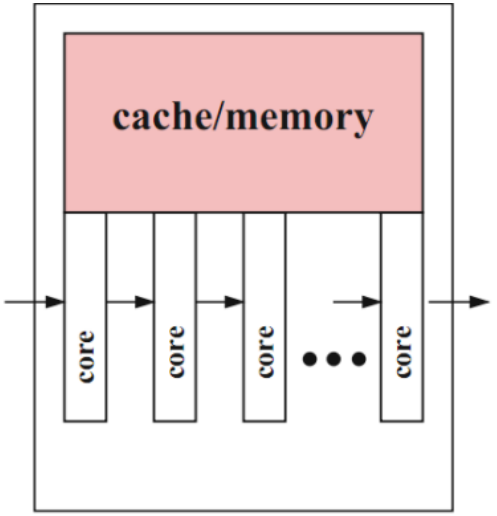


Hierarchical designs



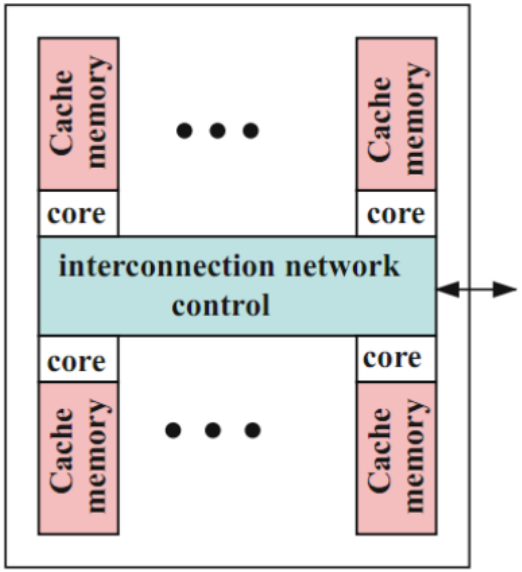
- Each core can have a separate L1 cache and shares the L2 cache
- All cores share common external memory

Pipelined design



- Multiple packets being processed in a pipelined fashion
- Cores connected linearly, shares the same cache, memory
- Useful if the same computation has to be applied to a long sequence of data elements

Network-based design



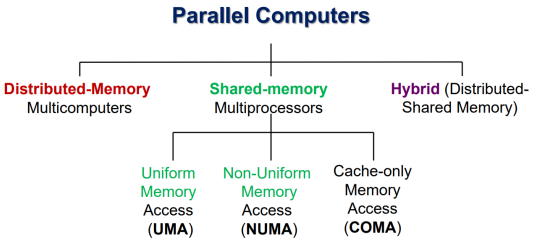
- Cores and their local memory and memories are connected via an interconnection network

Why cache

- cache provides high bandwidth data transfer to CPU and reduce latency in data access
- Memory latency: Amount of time for a memory request from a procesor to be serviced
- Bandwidtdh: Rate at which the memory system can provide data to a processor
- A stall happens when the next instruction depends on previous instructions
- Bandwidth and latency affects stalls, since instructions (sw, lw) needs to wait for the memory system to become available

Performant parallel programs

- Try not to overload the memory system with too many requests
- Share data across threads (inter-thread cooperation)
- Reuse data fetched previously (temporal locality)
- **Favor additional arithmatic over load / store**



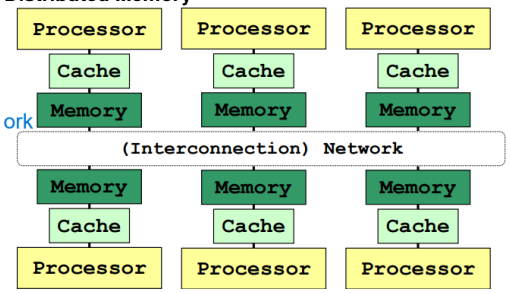
Cache coherence

- Multiple copies of data exist on different caches
- Local updates should not be seen by other processes
- Maintained by additional insructions
- Instructions that mess up cache coherence hence presents severe overheads

Memory consistency

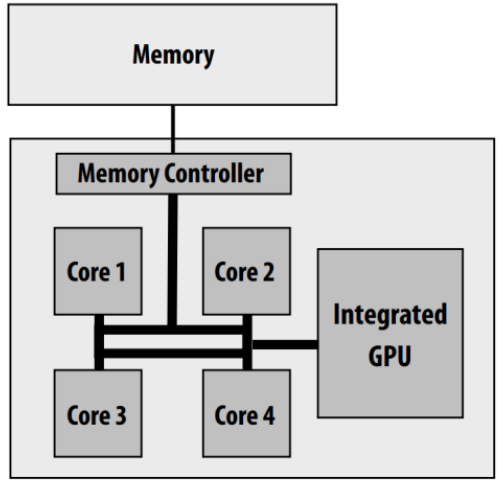
- Memory consistency depends on the PL and architecture
- A seq consistent architecture makes a PL with seq const memory model run faster since fewer instructions are needed to ensure memory consistency

Distributed Memory



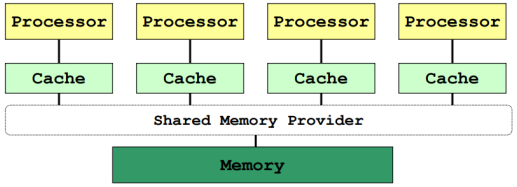
- Each node is an independent unit with processor and memory
- Memory in each node is private
- Nodes communicate through a network

Shared memory



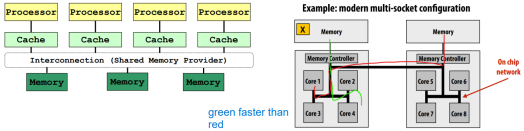
Intel Core i7 (quad core) (interconnect is a ring)

- Parallel programmes share memory through controller / provider
- Cache coherence and memory consistency is ensured



Uniform Memory Access

- Latency of accessing main memory is the same for processors
- Suitable for samll number of processors. Contention over memory can be high for large number of processes

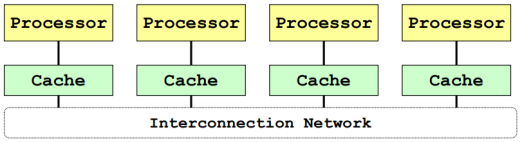


Non-uniform Memroy Access

- Physically distributed memory of all processing elements are combined to form a global shared memory
- Local memory access has lower latency
- Reduce contention since each processor tend to access local memory
- Adding more processes does not increase contention as much as UMA
- Data consistency is easier too

Cache Coherent NUMA (CCNUMA)

- EAch node has cache to reduce contention



Cache only Memory Architecture (COMA)

- Each memory blocks works as cache memory. This means that no fixed space stores data permanently and cache block with data can be moved around dynamically.
- Data migrates dynamically to keep data as close as possible to the processors
- Cache coherence is harder since data may not just be copied, they can also be shifted around.

L7: Cache coherence and memory consistency

L11: Interconnection networks

Parallel Computation Models

L4: Shared-memory programming models

Parallelism

- Average number of units of work that can be performed in parallel per unit time.
- E.g. MIPS, MFLOPS
- Limitation: Program dependencies - data, control
- Runtime delays - memory contention, communication overheads, thread overhead, synchronisation
- We cannot reorder them however we like
- Work = Task + dependencies (limitations)

Data parallelism

- If iterations are **independent**, they can be executed in arbitrary order on multiple cores
- Patition data among processing units, each doing similar work
- Commonly expressed as a loop, if the iterations are independent and can be executed in arbitrary order
- E.g. SIMD computers
- **OpenMP - matrix multiplication**

```
// parallelize result = a x b
// each thread works on one itreration of
     the outer-most loop
// vars (a, b ,result) are shared
#pragma omp parallel for num_thread(8)
     shared(a, b, result) private(i, j ,k)
     ...
```

- **Same as**

```
for (i=0; i < size; i ++)
     for (j=0; j < size; j ++)
          for (k=0; k < size, k ++)
               result[element][i][j] +=
                    a.element[i][k] *
                    b.element[k][j]
```

- **Single Program Multiple Data (SPMD)**
  - Same programme may behave differently based on the data
  - E.g. Scalar product of $x.y$ on p processing units

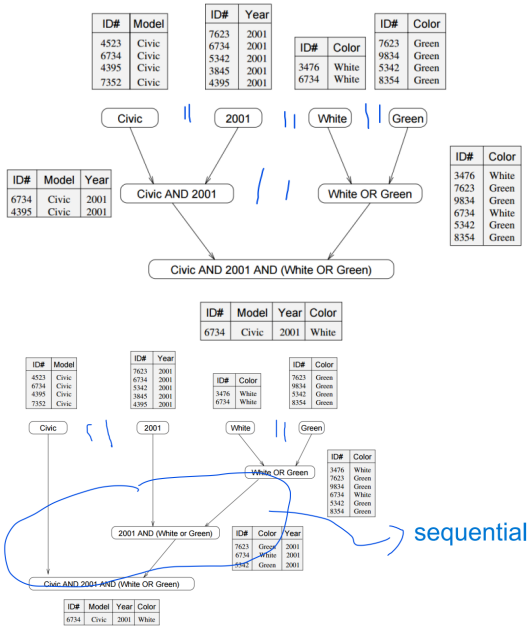
```
local_size = size/p;
local_lower = me * local_size;
local_upper = (me+1) * local_size - 1;
local_sum = 0.0;

for (i=local_lower; i<=local_upper; i++)
  local_sum += x[i] * y[i];

Reduce(&local_sum, &global_sum, 0, SUM);
```

Same program executed by **p** processing units.

"me" is the processing units index (0 to p-1)

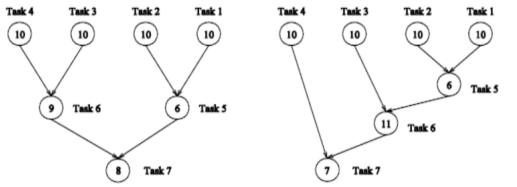
Task parallelism

- Partition the tasks among the processing units
- indepedent program tasks/ parts can be executed in parallel
- Granularity: statement, loop, function
- More complexed than data parallelism → needs to schedule, map, take care of dependencies ...
- **Decomposition**
  - The room for parallelism in a task depends on how the task is decomposed



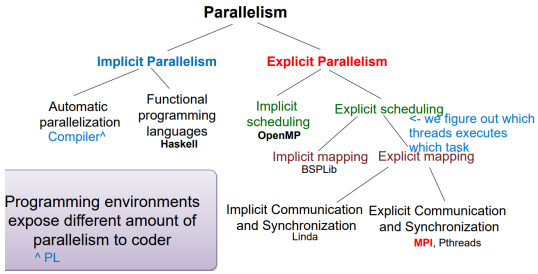
Task dependence graph

- DAG: node=tasks, value=expected execution time, edge=condtrol dependency
- Bad for one process to take disproportionately more data → idle time
- Critical path length: maximum slowest completion time
- Degree of concurrency=total work/critical path length



Critical Path = (Task 4 → 6 → 7)
Critical Path Length = **27**
Degree of concurrency = 63 / 27 = **2.33**

Critical Path = (Task 1 → 5 → 6 → 7)
Critical Path Length = **34**
Degree of concurrency = 64 / 34 = **1.88**



Programming environments expose different amount of parallelism to coder
^ PL

Coordination: Shared memory

- Protect access to shared address space, mutex.
- Needs hardware support to implement efficiency. NUMA makes it easier but it is still costly to scale due to contention (any processor can load/ store to any address)
- Can be done without a shared memory system (NUMA, UMA)
- Any type of coordination can be used in any hardware via software

Coordination: Data-parallel

- SIMD, vector processors
- Traditional: Map a function onto a large collection of data
- Side effect free execution
- Modern: Data-parallel languages do not enforce this structure
- SPMD model used in CUDA, OpenCL, ISPC instead

Coordination: Message passing

- Tasks operate within their own private address space and communicate by explicitly sending / receiving messages
- E.g. MPI, GO
- Hardware does not implement system wide loads and stores, can connect commodity systems toegther to form large parallel machines
- Many many computers, not a very big one
- Compatible with distributed memory systems

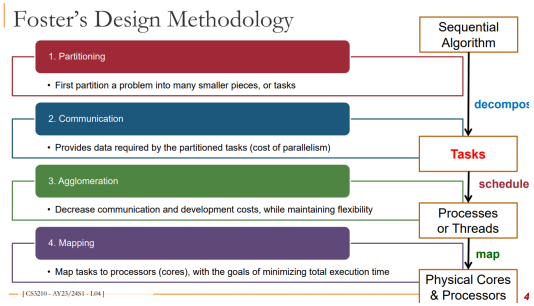
Coordination and hardware

- Shared memory: UMA, NUMA. Copies of messages and sent / received from library buffers
- Message passing: distributed systems, clusters, supercomputers
- Any abstraction can be implemented with any hardware but it will be more costly
- Shared address space on incompatible hardware
  - Write: Send message to all cores to invalidate value
  - Read: page fault handler issues appropriate network requests

Summary of Coordination Models

- Shared address space: very little structure
  - All threads can read and write to all shared variables
  - Drawback: not all reads and writes have the same cost (and that cost is not apparent in program text)
- Data-parallel: very rigid computation structure
  - Programs perform the same function on different data elements in a collection
- Message passing: highly structured communication
  - All communication occurs in the form of messages

Foster's Design Methodology



Foster's Design methodology

1. Partitioning
   - Divide computation and data into independent pieces to discover maximum parallelism
   - Two approaches:
     - 1.1. Domain decomposition: divide data into smaller, equal pieces. Associate computation with data.
     - 1.2. E.g. 24 tasks with 3 grids each → 6 tasks with 12 grids each
     - 1.3. Functional decomposition: Divide computation into piece. Associate data with computation.
     - 1.4. E.g. Climate model → Atmospheric model, hydrology model ...
   - Rule of thumb:
     - 10x more primitive tasks than cores in target computer
     - Minimize redundant computations and redundant data storage
     - Primitive data should be of roughly the same size
     - Number of tasks an increasing function of problem size
2. Communication (coordination)
   - Dependencies between tasks necessitates communication
   - Overlap computation and communication such that when some tasks are communicating, others are computing (improve utilisation)
   - **Local Communication**
     - Tasks needs data from a small number of other tasks (neighbors)
     - Use channel
   - **Global Communication**
     - Significant number of tasks contribute to perform a computation
     - Do not create channels early on in the execution

Parallel Programming Patterns

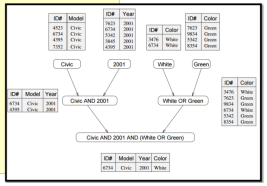
- Pattens are not mutually exclusive, use the best match

Fork Join

- Children run in parallel but are independent
- Children execute the same or different program

- Children join the parent at different points
- **Implementation:** Processes, threads etc

```
P1 = Fork {
     P3 = Fork { return Model = "civic" }
     P4 = Fork { return Year = "2001" }
     Join P3, P4
     Return P3 AND P4
}
P2 = Fork {
     P5 = Fork { return Color = "green" }
     P6 = Fork { return Color = "white" }
     Join P5, P6
     Return P5 OR P6
}
Join P1, P2
Return P1 AND P2
```

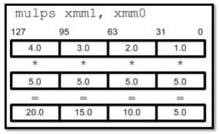


Parbegin - Parend

- most relaxed, code is structured into sequential segments and parallel segments
- Programmer specifies a sequence of statements to be executed in parallel
- A set of threads is created and the statement of the construct are assigned to these threads
- All the forks are done at the same time and all the joins are done at the same time
- Statements after parbegin and parend are only executed after all threads joins (barrier)
- **Implementation:** OpenMP or compiler directives
- E.g Matrix multiplication using openMD

SIMD (not the Architecture)

- Single instructions are executed synchronously by diferent threads on different data
- Similar to parbegin-parend but al threads execute the same instruction at the same time (synchronous)
- Parallel but synchronous
- **Implementation:** AVX / SSE instruction on intel processor



xmm registers are 128 bits long

SSE instruction treats the xmm registers as 4 individual 32-bit floating point value

SPMD

- Same program executed on different cores but operate o different data
- Different threads might execute on different instructions of the same program due to control flow (ifs) and speed of cores
- Similar to parbegin-parend but there is no implicit synchronization (lack of barrier)
- E.g. programs on GPGPU

Master-Worker

- Single program controls the execution of the program
- Master executes main function, assigns work to worker threads
- Initialisation, output and Coordination is done by master
- Worker waits for instruction

L6: Data parallel models (GPGPU)

L9,10: Distributed-programming models

Performance and Scalability of Parallel Programs

L5: Performance of parallel systems

Two Views

- Response Time (user): duration of a program is reduced (start - end time)
- Throughput (computer manager): more work to be done in the same time (jobs per second)

Performance Factors

1. Programming Model
2. Computational Mode: How well the given program runs in the given architecture
3. Architectural Model: interconnnection network, memory organization, execution mode,sync or async processing

Response time in sequential programs

- Wall-clock time
- Comprise of
  - User CPU time: time CPU spends executing program
    - Know that read and write cycles take different time
  - System CPU time: time CPU spends on system instructions. Depends on OS.
  - Waiting time: IO waiting time and execution of ther programs due to time sharing. Depends on the load of the system.

L8: performance instrumentation

New Trends

L12: Energy efficient computing