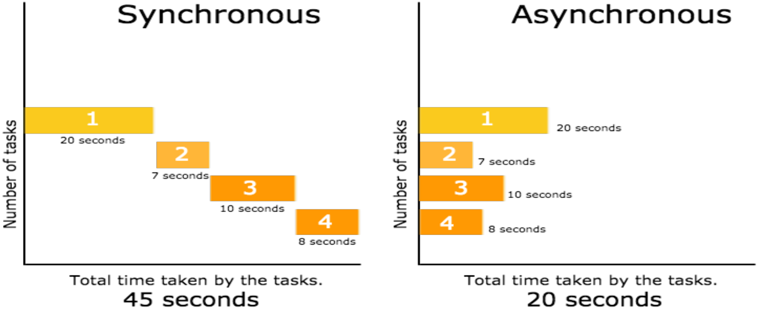
Recalling Topics on Parallel Programming

April 22nd



Recalling Topics on Parallel Programming

1. Introduction
2. Parallel Programming Models
3. Parallel Architectures
4. Synchronization
5. Message Passing
6. Parallel Constructs and Techniques
7. Languages and runtime systems for parallel programming
8. Performance Issues

Introduction

- Why Parallelism?
- Dimensions of parallel programming
- Design and Verification of Parallel programs

Why parallelism?

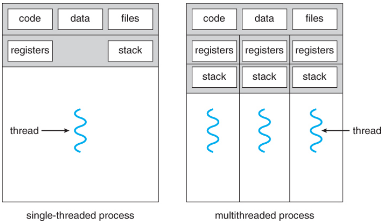
- Physical limits of sequential processor speed
- Natural parallelism in some applications
- System software
- Intrinsic interest

Dimensions of Parallelism

- Processes and threads
- Programming Models
- Concurrent x parallel x distributed
- Parallel and distributed systems
- Parallel architectures
- Languages and runtime
- Performance metrics

Processes and Threads

- Process → workspace
- Thread → same workspace as parent process
- Process (logical) != processor (physical hardware)
- Process is an abstraction of a processor
- Von Neumann model → one control flow
- concurrent program → 1+ flow



https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/4_Threads.html

Programming Models

- Define interface used by the programmer
- Parallelism, communication, synchronization etc
- Examples: sequential, shared/centralized memory, message passing

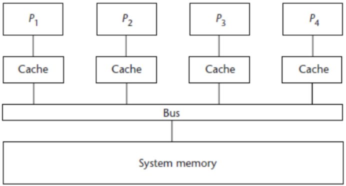
Concurrent x Parallel x Distributed

- Concurrent: 1+ control flow
- Parallel: concurrent with shared memory
- Distributed: concurrent with message passing

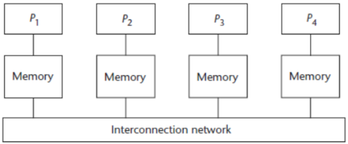
Parallel and Distributed Systems

- Parallel: hardware with just one memory space
- Distributed: multiple memories
- It is possible to run distributed programs in parallel systems and vice-versa

Shared memory

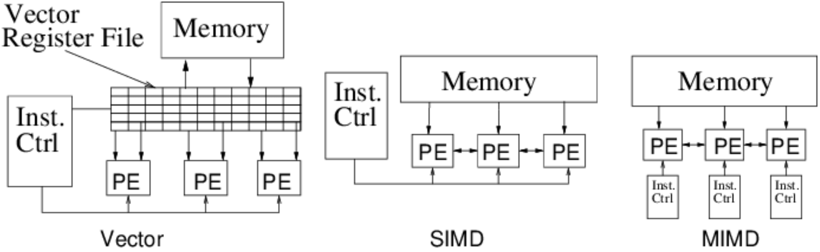


Distributed memory



Parallel Architectures

Most common: **SIMD** and **MIMD**



https://www.researchgate.net/publication/4049051_Universal_mechanisms_for_data-parallel_architectures/figures?lo=1

Languages, Compilers and Libraries

- Languages: special syntax, side effects and implicit context, type verification, threads, exception handling etc
- Compilers: makes the programming model simpler
- Library: easy to modify, use with existing languages, use with several languages

Performance Metrics

Amdahl's Law:

- Speedup $s = \frac{T(1)}{T(p)}$
- Total work $c = T_s + T_p = T(1)$
- $T(p) = T_s + \frac{T_p}{p}$
- $s = \frac{(T_s + T_p)}{(T_s + \frac{T_p}{p})} =$
  $= \frac{c}{(T_s + \frac{T_p}{p})} \rightarrow \frac{c}{T_s}$ when $p \rightarrow \inf$

Design and Verification of parallel programs

- Important: guarantee liveness and safety
  $\rightarrow$ Liveness: good things eventually happen
  $\rightarrow$ Safety: bad things never happen!
- Examples of liveness: no process waits forever, the program terminates
- Examples of safety: mutual exclusion, no buffer overflow

Most common parallel programming models

- Sequential
- Shared memory
- Message passing
- SPMD vs. MPMD or data parallelism vs. task parallelism

Other programming models

- Linda
- Actors
- Dataflow
- Logic
- Functional
- Constraints

Sequential programming model

- The simplest of all...
- Parallelism implemented by the compiler or by the system

```
for i = 1 to N
    a[i] = 1
```

- e.g.: HPF and other Fortran versions (compiler); some declarative languages (runtime)

Shared memory model

- More complex, but close to sequential
- Parallelism implemented by the programmer with constructs and calls to functions provided by a language or by the system software
- Synchronization is needed
- Transparent communication (implemented in sw or hw)
- e.g: C#, Java (language), OpenMP (runtime, library), PFS (file system, OS)

Shared memory model

```
doall i = 1 to N
    a[i] = 1

for j = 1 to NPROCS-1
    fork(compute,j)
compute(0)

lock(mutex)
    x = x + 1
unlock(mutex)
```

Message Passing model

- Very complex model
- parallelism implemented by the programmer using language or system calls
- Explicit process communication
- Synchronization associated with the messages
- e.g.: SR and Occam (language), MPI (runtime library)

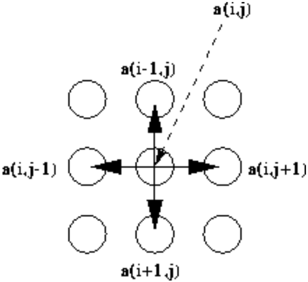
Message passing model

```
Proc pid:

chunk = N/NPROCS
for j = pid*chunk to (pid+1)*chunk-1
    a[i] = 1
send(dest,&a[pid*chunk],chunk*sizeof(int))
```

Example: Successive Over Relaxation - SOR

- Computing over a matrix
- Group of consecutive lines per process
- Each new cell value is calculated using neighbor cells
- Communication on the borders



http://www2.phys.canterbury.ac.nz/dept/docs/manuals/Fortran-90/HTMLNotesnode193.html

Example: SOR

Sequential

```
for num_iters
    for num_linhas
        compute
```

Shared memory

```
for num_iters
    for num_linhas in //
        compute


or


for num_iters
    for num_minhas_linhas
        compute
    barreira
```

Example: SOR

Message passing with non-blocking send

```
define submatriz local
for num_iters
    if pid != 0
        send first line to process pid-1
        receive last line from process pid-1
    if pid != P-1
        send last line to pid+1
        receive first line from pid+1
    for num_linhas
        compute
```

Comparing models

- Sequential ideal, but it depends on sophisticated software
- Shared memory model yields simpler programs, but requires explicit synchronization
- Message passing yields efficient communication and implicit synchronization, but it makes the programming model more difficult

SPMD vs. MPMD

- Classification of programs
- SPMD = data parallelism = program for SIMD running over MIMD
- MPMD = task parallelism; example: master-slave

Classical topics in shared memory

Race conditions

- when actions are not synchronized and behavior depends on their order
- sometimes it does not cause problems. For example, master-slave or task queue
- in general, we want to avoid race conditions

Example that we need to prevent:

```
  Proc 1              Proc 2
load X,reg          load X,reg
inc reg             inc reg
store reg,X         store reg,X
```

Classical topics in shared memory

Synchronization

- Used to avoid race conditions
- two types: mutual exclusion and conditional sync
- need atomic instructions
- need to care to not over synchronize

Example of synchronization

Proc 1	Proc 2
`mutex L`	`mutex L`
`load X,reg`	`load X,reg`
`inc reg`	`inc reg`
`store reg,X`	`store reg,X`
`demutex L`	`demutex L`

Synchronization

- iteratively read a variable till some value: busy waiting
- busy waiting spends precious processor cycles
- sync needs to interact with the scheduler to block: semaphores and monitors
- Tradeoff: spin when waiting time is lower than the overhead of rescheduling

Classical topics of message passing

- blocking and non-blocking communication
- Naming and collective communication
- Messaging overhead

Blocking and non-blocking

- blocking comm. does not require buffers
- non-blocking communication → max concurrency; flow and error problems
- blocking send waits till receptor is ready
- blocking receive waits till a message appears
- non-blocking Send complete immediately, except when there is no buffer
- non-blocking Receive completes immediately

Naming and Collective Communication

- Channel, port, or process used to specify a receptor in a 1-to-1 communication
- Other forms of communication for collective communication 1-to-many, many-to-1 and many-to-many
- Exs: links em Demos, Demos-MP, and Arachne; Linda tuple space

Messaging Overhead

- message passing generally costly (done in sw and with the intervention of the OS)
- Modern systems avoid calling the OS (only napping and verification of protection)
- Exs: Active msgs, Fast msgs

Data Parallelism

Decomposing and distributing data

	P0	P1	P2	P3
	x(1)	x(4)	....	....
block	x(2)	x(5)		
	x(3)	....		

	P0	P1	P2	P3
	y(1)	y(2)	y(3)	y(4)
cyclic	y(5)	y(6)	....	....
	y(9)	....		

	P0	P1	P2	P3
	z(1)	z(3)	z(5)	....
cyclic	z(2)	z(4)	z(6)	
(2)	....	....	....	

Data Parallelism

Different types of loops:

- Array assignment – Ex: `a(1:n) = b(0:n-1)*2 + c(2:n+1)`
- `do` (seq) – one iteration only starts after the previous one finishes
- `dopar` (par) – iterations are executed by different processes/threads and data is the same as when the loop started in each proc
- `doall` (special dopar) – there are no dependencies between iterations
- `doacross` (par) – there are dependencies and assignments of each iteration will be seen by the others

Dependence Relations

- Relations are used to represent ordering constraints between the commands in a program
- In the example below: Moving (2) above (1) or changing the order of (3) and (4) modify the semantics. But changing the order of (2) and (3) does not cause problems.

```
(1) A = 0
(2) B = A
(3) C = A + D
(4) D = 2
```

Dependence Relations

Data dependence graph: nodes = statements or blocks, edges = constraints
Constraints:

- Flow dependence: variable assigned in a statement and used in the next
- Anti-dep: variable used in a statement and assigned in the next
- Output dependence: variable assigned in a statement and reassigned in the next

Example

```
(1) A = 0        S1 --+
(2) B = A        |    |
(3) C = A+D      V    | flow
(4) D = 2        S2   |
                      |
                 S3 <-+
                 |
                 - anti
                 |
                 V
                 S4
```

precedence graph: directed acyclic

Dependence in sequential loops

- loop-carried dependence: dependence between statements in different loop iterations
- loop independent dependence: dependence between statements of the same iteration
- Forward (backward) dependence: source precedes destination (destination precedes source)

Example

```
(1) do I=2,9
(2)    X[I] = Y[I] + Z[I]
(3)    A[I] = X[I-1] + 1
(4) enddo
```

```
Dependence relations caused by X:

         I=2                I=3
(2) X[2]=Y[2]+Z[2]   X[3]=Y[3]+Z[3]
(3) A[2]=X[1]+1      A[3]=X[2]+1

Forward dep from (2) to (3):

    S2
    |
    | (1)
    V
    S3
```