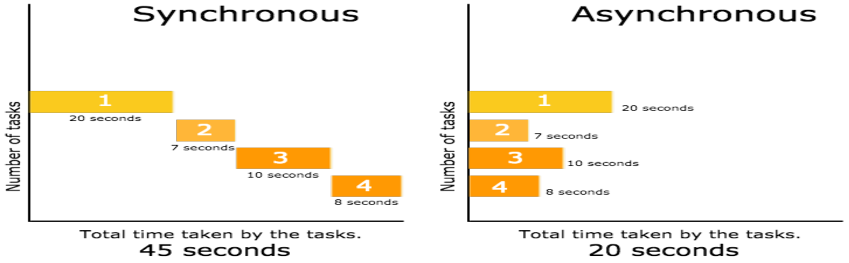
Big Data and Cloud Computing, 23/24

Inês Dutra

DCC-FCUP
room 1.31
ines@dcc.fc.up.pt

Apr 17th, 2024



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)

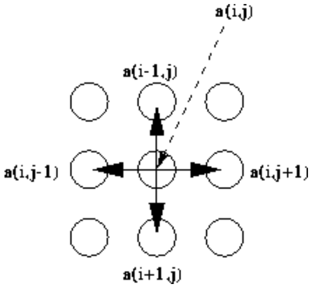
Message passing model

```
Proc pid:

chunk = N/NPROCS
for i = 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 $\rightarrow$ max concurrency; flow and error problems
- blocking send waits till receptor is ready
- blocking receive waits till a message appears
- non-blocking Send completes immediately, except when there is no buffer
- non-blocking Receive completes immediately even if there is nothing in the buffer

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

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
```

Iteration space

- Graphically: 1 dot per iteration
- Directed edge if the command of one iteration depends on the previous iteration (dependence graph)
- Graph difficult to build because dependencies can become crossed in space

Example

```
(1) do I=3,7
(2)    do J=6,2 by -2
(3)       A[I,J] = A[I,J+2] + 1
(4)    enddo
(5) enddo
```

```
   J
   ^
   |
6  |           *   *   *   *   *
   |           |   |   |   |   |
5  |           |   |   |   |   |
   |           V   V   V   V   V
4  |           *   *   *   *   *
   |           |   |   |   |   |
3  |           |   |   |   |   |
   |           V   V   V   V   V
2  |           *   *   *   *   *
   |
1  |
   |
0  +---------------------------- > I
   0   1   2   3   4   5   6   7
```

Dependence in parallel loops

- Two statements or iterations are in conflict when they may refer to the same memory address
- Conflicts need to be solved so that semantics is correct
- List of statements: conflicts solved completing first memory access before initiating the second access: conflicts solved as anti-dependences
- Loop: conflict solved according to loop rules

Dependende among parallel loops

- do: conflicts between iterations are solved completing the memory access of the previous iteration first
- dopar: values computed in one iteration can not be used by other iteration $\rightarrow$ conflicts solved as anti-dependencies or output dependencies
- doall: there are no dependencies between iterations $\rightarrow$ conflicts solved as if loops are independent

Example

```
    (1) dopar I=2,20
    (2)     X[I] = Y[I] + 1
    (3)     Z[I] = X[I-1] + X[I] + X[I+1]
    (4) enddopar
```

```
         I=2                         I=3
(2) X[2]=Y[2]+1                 X[3]=Y[3]+1
(3) Z[2]=X[1]+X[2]+X[3]   Z[3]=X[2]+X[3]+X[4]
```

flow dependency between (2) and (3), distance (0) - X[I]
Anti-dep between iterations with distance (-1) and (1) - X[I-1], X[I+1]

Loop restructuring

Reorder statements and iterations, but preserving semantics

Techniques:

- Peeling
- Splitting
- Scalar expansion
- Fusion
- Fission
- Interchanging
- Strip mining
- Tiling
- Unrolling

Peeling

Isolate 1st or last iterations from the remaining
Often used to adjust number of iterations (to allow for fusion, for example) or to remove a condition tested on an index

```
Before: do I=1,N
            A[I] = (X + Y) * B[I]
        enddo

After:  if N >= 1 then
            A[1] = X + Y * B[1]
            do I=2,N
                A[I] = X + Y * B[I]
            enddo
        endif
```

Splitting

Divide set of indice in two subsets
Used by the same reasons as peeling

```
Before: do I=1,100
              A[I] = B[I] + C[I]
              if I > 10 then
                  D[I] = A[I] + A[I-10]
              endif
        enddo

After:  do I=1,10
              A[I] = B[I] + C[I]
        enddo
        do I=11,100
              A[I] = B[I] + C[I]
              D[I] = A[I] + A[I-10]
        enddo
```

Scalar Expansion, Fusion, Fission, Interchanging

- SE: scalars become arrays to prevent anti-deps and output dependencies
- Fusion: join 2 loops of same limits in just one loop. Used to reduce costs of branch and test instructions, to improve temporal locality and allow for scalar optimizations such as elimination of subexpressions in mathematical calculations
- Fission: opposite of fusion. Used to improve cache hit ratios
- Interchanging: change nested loops. Used to reduce the cost os starting loops, to help uncovering automatic parallelism (move loops without dependencies inside the main loop) and improve temporal and spatial locality

Strip Mining

Convert one loop in two nested loops
Used to improve data locality

```
Before: do I=1,N
            A[I] = B[I] + C[I]
        enddo

After (strip of size s):
        do Is=1,N by s
             do I=Is,min(N,Is+s-1)
                 A[I] = B[I] + C[I]
             enddo
        enddo
```

Tiling and Unrolling

- Tiling: similar to strip mining, but with nested loops. Used to created blocked versions of code with better data locality
- Unrolling: Used to increase amount of instruction parallelism

Work distribution

- Master/slave paradigm: main part of code executed by just one process (master - controller)
- When parallel region is reached, master creates multiple slaves
- When slaves finish execution synchronize using a barrier

Implementation:

- Slaves: processes or threads?
- Should Master participate of the parallel execution?
- Or should it be doing some distinct work?
- Does last slave in the barrier become master?

Scheduling

- Static: N/P consecutive iterations per slave (blocks) or one iteration per slave (round-robin or cyclic).
- Static scheduling does not work well when there is load imbalance
- Dynamic:
  - Self-scheduling: each slave takes a task from a queue
  - Guided self-scheduling: 1/P tasks each time
  - Affinity scheduling: each slave continues executing iterations that has executed before (parallel loop inside sequential loop)

Synchronization

- Needed whenever there is dependence between iterations or tasks
- Implemented through ordered **critical sections** (CS)
- Processes enter CS in order
- Implemented using primitives await and advance

Example

```
do I=2,n
    D[I] = D[I-1] + A[I]
enddo

Each slave:

await(I-1)          await and advance can be
fetch D[I-1]->r1   implemented in different forms.
fetch A[I]->r2     e.g., bit vector (1 per iteration);
add r1, r2->r3     advance sets i-th bit,
store r3->D[I]     await waits till
advance            (i-1)-th bit is set (atomic operation).
```

Python multiprocessing.Pool

- synchronous: parent process blocks and only proceeds to the next statement after the call finishes: `pool.apply`, `pool.map`, `pool.starmap`, `pool.imap`, `pool.imap_unordered`
- asynchronous: parent process proceeds as soon as the call executes: `pool.apply_async`, `pool.map_async`, `pool.starmap_async`

Python multiprocessing.Pool

(source: `superfastpython.com` cheat sheet for multiprocessing.Pool)

- multiprocessing.Pool execute functions that perform CPU-bound tasks asynchronously in new child processes

Python multiprocessing.Pool

- Create, configure and use
  - Import module
    ```
    from multiprocessing import Pool
    ```
  - Create default config
    ```
    pool = Pool()
    ```
  - Config number of workers
    ```
    pool = Pool(processes=8)
    ```
  - Config worker initializer function
    ```
    pool = Pool(initializer=init, initargs=(a1,a2,...,))
    ```
  - Config max tasks per child worker
    ```
    pool = Pool(maxtasksperchild=10)
    ```

Python multiprocessing.Pool

- Create, configure and use
  - Config multiprocessing context
    ```
    ctx = get_context('spawn')
    pool = Pool(context=ctx)
    ```
  - Close after tasks finish, prevent further tasks
    ```
    pool.close()
    ```
  - Terminate, kill running tasks
    ```
    pool.terminate()
    ```
  - Join, after close, wait for workers to stop
    ```
    pool.join()
    ```
  - Context manager, terminate automatically
    ```
    with Pool() as pool:
        # ...
    ```

Issue tasks synchronously

- Issue tasks, block until complete
  - Issue one task

    ```
    value = pool.apply(task, (a1,a2))
    ```
  - Issue many tasks, one argument/iterable

    ```
    for val in pool.map(task, items):
        # ...
    ```
  - Issue many tasks, lazy

    ```
    for val in pool.imap(task, items):
        # ...
    ```
  - Issue many tasks, lazy, unordered results

    ```
    for val in pool.imap_unordered(task, items):
        # ...
    ```
  - Issue many tasks, multiple arguments/iterables

    items = [(1,2), (3,4), (5,6)]

    ```
    for val in pool.starmap(task, items):
        # ...
    ```

Issue tasks Asynchronously

- Issue tasks, return control to parent process immediately
  - Issue one task
    ```
    value = pool.apply_async(task, (a1,a2))
    ```
  - Issue many tasks, only one argument/iterable for the target function
    ```
    for val in pool.map_async(task, items):
        # ...
    ```
  - Issue many tasks, multiple arguments/iterables for the target function
    items = [(1,2), (3,4), (5,6)]
    ```
    for val in pool.starmap_asynch(task, items):
        # ...
    ```

Chunksize

- All versions of map() functions

```
for val in pool.map(task, items, chunksize=5):
    # ...
```

Results

- Get result (blocking)

```
result = value.get()
```

- Get result with exception

```
try:
    result = value.get()
except Exception as e:
    # ...
```

- Get result with timeout

```
result = value.get(timeout=5) # 5 seconds
```

- for all asynch functions: apply callback to collect results

Wait and task status

- Wait for task to complete

```
value.wait()
```

- Wait for task, with timeout

```
value.wait(timeout=5)
```

- Check if task is finished (not running)

```
if value.ready():
    # ...
```

- Check if task was successful (no exception)

```
if value.successful():
    # ...
```