Advanced R Programming - Lecture 6
Parallel programming

Krzysztof Bartoszek
(slides based on Leif Jonsson's and Måns Magnusson's)

Linköping University

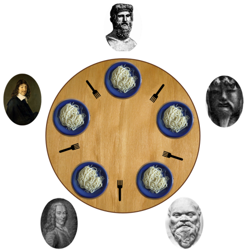
krzysztof.bartoszek@liu.se

29 IX, 1 X 2021 (Zoom)

Today

Questions since last time?

Classical introduction: dining philosophers problem



wait until left fork available
wait until right fork available
eat
release left fork
release right fork

Solutions:
synchronize: pick lower number (Dijkstra)
wait for permission (mutex)
release and random wait

https://en.wikipedia.org/wiki/Dining_philosophers_problem

Mostly we will not (a.s.) be concerned with such problems.

Naïve distributed leader election

1. Each processor has a unique id, the processors form a ring, ids can be ordered and are all comparable.
2. Each processor sends its id to its left neighbour.
3. If the received id is greater than one's own, then the received one is passed to the left neighbour. Otherwise the receiving processor becomes silent.
4. If a processor receives its own id back it declares itself the leader.

Complexity is $O(n^2)$ messages passed.
Distributed minimum, maximum algorithm.

See e.g. `https://hagit.net.technion.ac.il/da98/notes/` for more sophisticated approaches.

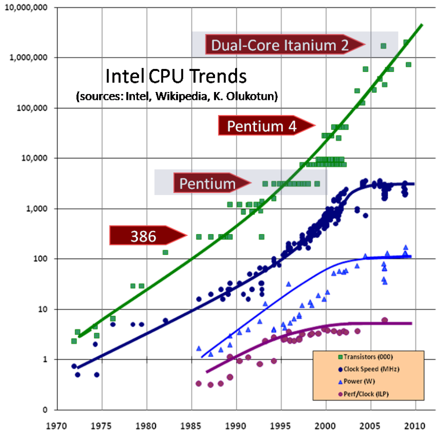
What is parallelism?

Multiple cores

Each core work with its own part

Cores can exchange information

Why parallelism?



http://www.gotw.ca/publications/concurrency-ddj.htm

Why parallelism?

Single core limits

Handling larger data

Solving problems faster

More robust (if one processor fails, others still can work)

More and more important

Is there any **but** . . . ?

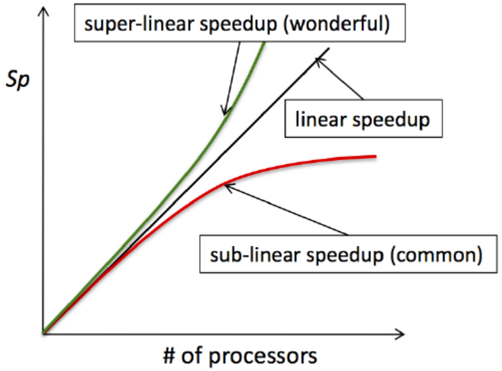
Types of parallelism

Multicore systems

Distributed systems

Graphical processing units (GPU)

Speedup



```
https://portal.tacc.utexas.edu/c/document_library/get_file?uuid=
e05d457a-0fbf-424b-87ce-c96fc0077099&groupId=13601
```

Theoretical limits

Strong scaling: Amdahl's law
Deals with *fixed problem size, increasing resources*

Weak scaling: Gustafsons law
Deals with *increasing size problem along with increasing resources*

Amdahl's law

$$\text{Speedup}: \quad S_p = \frac{\text{execution time on 1 processor}}{\text{execution time on } P \text{ processors}}$$

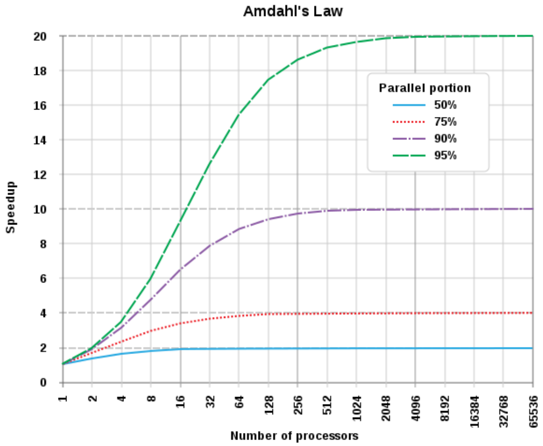
$$S_p \leq \frac{1}{f_s + \frac{f_p}{P}}$$

f_s = serial fraction of computations
f_p = parallel fraction of computations
P = number of cores

For a *fixed size problem*, single core computation time is fixed$= 1$!

$$\text{Efficiency}: \quad S_p P^{-1} = \frac{\text{execution time on 1 processor}}{P(\text{execution time on } P \text{ processors})}$$

Amdahl’s law



https://en.wikipedia.org/wiki/Amdahl’s_law

Gustafsons law

$$\text{Speedup}: \quad S_p = \frac{\text{execution time on 1 processor}}{\text{execution time on } P \text{ processors}}$$

$$S_p \leq \frac{\alpha + (1-\alpha)P}{\alpha + (1-\alpha)} = P - \alpha * (P-1)$$

Where:

$\alpha =$ fraction of time dedicated to serial computations
$P =$ number of cores

Problem size scales with P, parallel execution time is fixed$=1$!
if we only had one core, then the P parallel computations would have to be done on that core with time $(1-\alpha)P$

Practical problems

Costs of parallelism
communication
load balancing (NP–hard)
scheduling (NP–hard) but $4/3 = k$–approximate algorithm

Symmetry:

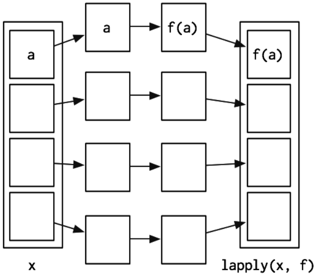
If $\forall_{P_1 \text{ in parallel}} \exists_{P_2 \text{ connected with } P_1} : v(P_1) = v(P_2)$ then $v(P_2)$++

Byzantine Generals Problem:
network must reach a decision but some nodes can be corrupt

fine-grained vs embarrassingly parallel (EP)

Parallelism in R (embarassingly parallel)

Based on `lapply()`
iterations inside a loop are indepenendent of each other



(H. Wickham, Advanced R, p. 201)

What about: `fctl<-1;for(i in 1:n){fctl<-i*fctl}` ?

parallel package

Two approaches:

1. mclapply()
2. parLapply()

mclapply()

Pros

Simple to use

Low overhead (startup)

Cons

Does not work on Windows

Only multi core

parLapply(type="psock")

Pros

Works everywhere

Good for testing/developing

Cons

Slow on multiple nodes

parLapply(type="mpi")

Pros

Good for multiple computers Good for production

Cons

Can be used interactively Needs Rmpi package

Load balance

`lapply(list,...)` does in order of `list`

load balancing is NP–hard
often we do not know running time for `list[[i]]`
submit to cores in order?
split into consecutive equal chunks?
Example:
all 2^p regression models on p predictors
generate all combinations: $\emptyset, \{1\}, \ldots, \{p\}, \{1,2\}, \ldots$

randomize order in `list`

`R`'s `par` family has load balancing capabilities

Subset methods (non–EP)

`glm()`: large number of observations

Chunk averaging (estimation)

break data into chunks of rows
to each chunk (in parallel) apply `glm()`
average the results to obtain single estimate

Observations i.i.d. and model is *decent*
Asymptotic equivalance for large samples

Chunks are not identically distributed: first cases then controls
randomly permute observations
(will not harm an initial random arrangement)

Subset methods (non–EP)

`glm()`: large number of predictors

Subsetting variables (prediction, *decent* model)

- create random subsets of predictors
- to each subset apply `glm()`
- do prediction for each subset
- combine predictions e.g. average, majority rule

prediction difficult in high dimensions—*curse of dimensionality*

Example

`https://github.com/STIMALiU/AdvRCourse/blob/master/Code/parallel_example.R`
Parallel code example

`https://github.com/STIMALiU/AdvRCourse/blob/master/Code/parallel_scalarproduct.R`
Parallel scalar product example

The End... for today.

Questions?

See you next time!