14. Modely a architektury paralelních a distribuovaných systémů; prostředky pro jejich implementaci a základní algoritmy.

https://cw.fel.cvut.cz/b172/courses/b4b36pdv/lectures/start

Modely a architektury paralelních a distribuovaných systémů

Paralelní programování

1 program, vícero úloh pro vyřešení úlohy, sdílení paměti a synchronizace
Chceme zrychlit řešení problému.

Distribuované systémy

Vícero programů, výpočetní procesy, každý svou paměť.
Chceme mít robustnější systém.

Prostředky pro implementaci paralelních a distribuovaných systémů

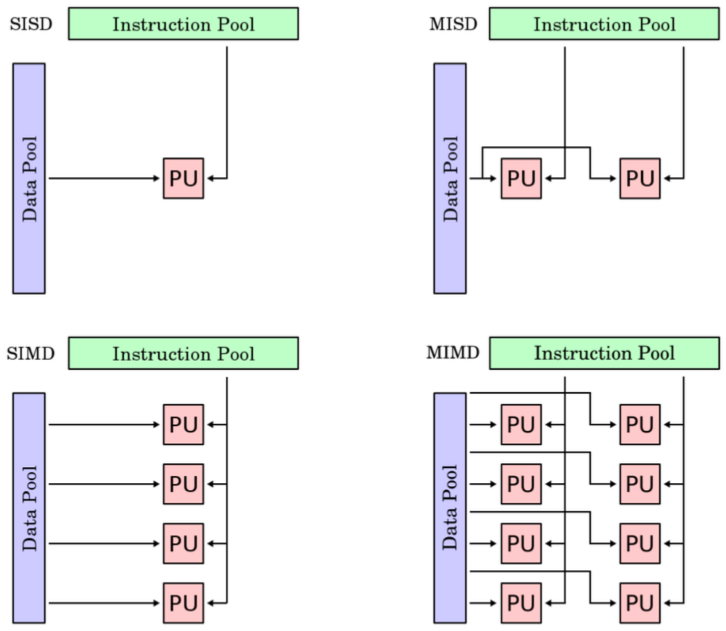
Paralelizace na úrovni instrukci (Instruction Level Paralelization - ILP)

SIMD - single instruction multiple data

- jedna řídící jednotka vícero ALU (arithmetic logical unit) jednotek
- datový paralelismus
- vektorové procesory, GPU

MISD - multiple instruction single data
MIMD - multiple instruction multiple data

- více jádrové procesory
- různé jádra různé instrukce



Vlákna

- create
- join
-

Synchronizace

Mutexy

- Mutex (lock, unlock)
- Semafor (post, wait)
- Monitor (účastníci mohou mezi sebou komunikovat)
- Podmínková proměnná (notify, notify all, wait)

OpenMP

- omp parallel for
- pragma task - await all and stuff
- reduction (+:a) převede normální sčítání na vektorové

Compare and swap

Využití u atomických proměnných. Atomické proměnné zajišťují, že operaci nad nimi provedené jsou atomické (princip makové buchty). Při použití compare_strong_Exchange - kromě hodnoty, kterou chceme vložit, ji posíláme i s hodnotou o které se myslíme že v proměnné je. Pokud máme pravdu tak se to normálně vloží. Pokud se nám to ale pod rukama změní, tak operace skončí s návratovou hodnotou false, a do proměnné s expected hodnotou se přiřadí aktuální hodnota, která nám tam proklouzla z jiného vlákna.

https://cw.fel.cvut.cz/wiki/_media/courses/b4b36pdv/lectures/lecture_4_2019.pdf

Rozdělené práce

Fixní a statické: ne nutně při kompilaci ale jakmile jednou vlákno dostane přiřazenou úlohu už mu zůstane
Dynamické: program rozděluje úlohy dynamicky podle vytíženosti jednotlivých vláken

- jedna fronta úkolů z které vlákna berou → dobré pro velké úlohy, při vícero malých úkolech to bude bottleneck
- každé vlákno se svou frontou, jakmile nemá tasky tak bere od dalších vláken, tím se balancuje

Jednotlivé úlohy na sobě mohou záviset, v openMP to pak lze řešit pomocí

```
#pragmaomptasksdepend([in/out/inout]:variables)
```

Základní paralelní a distribuované algoritmy

https://www.cs.cmu.edu/~scandal/nesl/algorithms.html

Parallel algorithms on sequences and strings

- Scan (prefix sums)
- List ranking
- Sorting
- Merging
- Medians

- Searching
- String matching

Parallel sorts

Obecně při práci s tasky: mám rekurzi, kterou pouštím na dvě poloviny. První polovinu hodím do tasku, druhou polovinu počítám sám na svém vlákně. Pak počkám na tu první polovinu.

Quicksort

Quicksort dělí vždy řazenou array na 2 podčásti. My můžeme paralelizovat vykonávání každé části. Nutné je však zabránit paralelizace ve větší hloubce abychom nestrávili více času správou paralelního vykonávání než samotným řazením.

Merge sort

Stejný případ

Binary search

Stejným způsobem jako quicksort. Do jisté hloubky můžeme při každém postupu do potomků nodu spustit úlohu paralelně. Opět je však nutné si hlídat množství paralelních úloh.

Parallel algorithms on trees and graphs

- Trees
- Connected components
- Spanning trees
- Shortest paths
- Maximal independent set

Parallel algorithms for Computational Geometry

- Convex hull
- Closest pairs
- Delaunay triangulation

Parallel algorithms for Numerical/Scientific Computing

- Fourier transform
- Dense matrix operations

- Sparse matrix operations

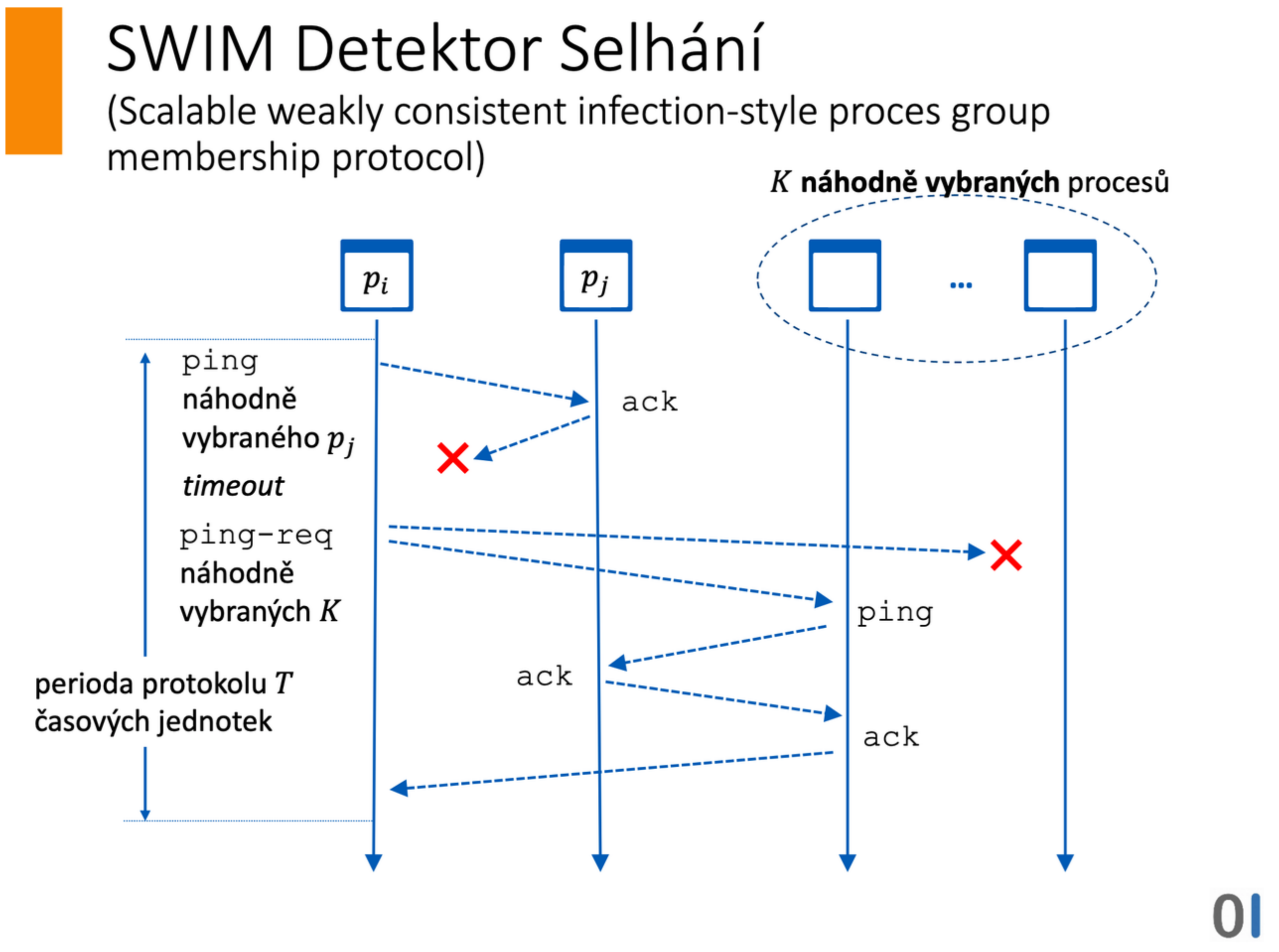
SWIM Protocol

SWIM full name is Scalable, Weakly-Consistent, Infection-Style, Processes Group Membership Protocol. Lets break this down:

- By *scalable* we mean that the protocol is applicable for large networks, consistings of thousands or tens of thousands of nodes. When dealing with networks of this magnitude, we must take into consideration the amount of messages sent among the nodes.
- By *Weakly Consistent* we mean that each node will have a somewhat different view of its surrounding environment. Over time, as nodes communicate and exchange information, we expect these conflicting views to converge.
- *Infection Style* is a synonym for gossip. Each node is expected to exchange information with a **subset** of its peers, so information flows in the network similar to the spread of an epidemic in the general population.
- Lastly, *Membership Protocol* means that we expect SWIM to answer the basic question - *who are my live peers?*

Failure detection in SWIM

Each node in the cluster will choose a node at random (say, `pj`) and will send a `ping` message, expecting to receive an `ack` back. This is simply a probe message and in normal circumstances it would receive this `ack` message and confirm that `pj` is still alive.
When that doesn't happen, though, instead of immediately marking this node as dead, it will try to probe it *through* other nodes. It will randomly select `k` other nodes from its membership list and send a `ping-req(N2)` message.



SWIM Detektor Selhání

(Scalable weakly consistent infection-style proces group membership protocol)

This helps to prevent false-positives when for some reason `pi` cannot get a response directly from `pj` (maybe because there's a network congestion between the two), but the node is still alive and accessible by others.
If the node cannot be accessed by any of the `k` members, though, it's marked as dead.

Raft

Raft is a consensus algorithm designed as an alternative to Paxos. It was meant to be more understandable than Paxos by means of separation of logic, but it is also formally proven safe and offers some additional features. Raft offers a generic way to distribute a state machine across a cluster of computing systems, ensuring that each node in the cluster agrees upon the same series of state transitions. It is named after Reliable, Replicated, Redundant, And Fault-Tolerant.

Raft achieves consensus via an elected leader. A server in a raft cluster is either a *leader* or a *follower*, and can be a *candidate* in the precise case of an election (leader unavailable). The leader is responsible for log replication to the followers. It regularly informs the followers of its existence by sending a heartbeat message. Each follower has a timeout (typically between 150 and 300 ms) in which it expects the heartbeat from the leader. The timeout is reset on receiving the heartbeat. If no heartbeat is received the follower changes its status to candidate and starts a leader election.
Raft guarantees each of these safety properties :

- **Election safety:** at most one leader can be elected in a given term.
- **Leader Append-Only:** a leader can only append new entries to its logs (it can neither overwrite nor delete entries).
- **Log Matching:** if two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.
- **Leader Completeness:** if a log entry is committed in a given term then it will be present in the logs of the leaders since this term
- **State Machine Safety:** if a server has applied a particular log entry to its state machine, then no other server may apply a different command for the same log.

Bully

The process with the highest process ID number from amongst the non-failed processes is selected as the coordinator.
The algorithm uses the following message types:

- Election Message: Sent to announce election.
- Answer (Alive) Message: Responds to the Election message.
- Coordinator (Victory) Message: Sent by winner of the election to announce victory.

When a process P recovers from failure, or the failure detector indicates that the current coordinator has failed, P performs the following actions:

1. If P has the highest process id, it sends a Victory message to all other processes and becomes the new Coordinator. Otherwise, P broadcasts an Election message to all other processes with higher process IDs than itself.
2. If P receives no Answer after sending an Election message, then it broadcasts a Victory message to all other processes and becomes the Coordinator.
3. If P receives an Answer from a process with a higher ID, it sends no further messages for this election and waits for a Victory message. (If there is no Victory

message after a period of time, it restarts the process at the beginning.)

4. If P receives an Election message from another process with a lower ID it sends an Answer message back and starts the election process at the beginning, by sending an Election message to higher-numbered processes.
5. If P receives a Coordinator message, it treats the sender as the coordinator.