

- Backtracking, branch & bound and alpha-beta pruning: how to assign work to idle processes without much communication?
- Additionally for alpha-beta pruning: implementing the young-brothers-wait concept. How to get the most urgent tasks done first?
- Another example: approximating the **Mandelbrot set** \mathcal{M} .

Load Balancing: The Mandelbrot Set

$c \in \mathbb{C}$ belongs to \mathcal{M} iff the iteration $z_0(c) = 0$, $z_{k+1}(c) = z_k^2(c) + c$ remains bounded, i.e., iff $|z_k(c)| \leq 2$ for all k .
(One can show that $|z_k(c)| \leq 2$ holds for all k whenever $c \in \mathcal{M}$.)

If $c \notin \mathcal{M}$, then the number of iterations required to “escape” varies considerably:

- $-2 \in \mathcal{M}$, but $c = -2 - \varepsilon$ escapes after one iteration for every $\varepsilon > 0$,
- $\frac{1}{4} \in \mathcal{M}$ and the number of iterations for $c = \frac{1}{4} + \varepsilon$ grows, when $\varepsilon > 0$ decreases.

How to balance the load?

The Mandelbrot Set: Load Balancing

We are given a rectangle R within \mathbb{C} .

- Color pixels in $R \cap \overline{\mathcal{M}}$ in dependence on the number of iterations required to escape.
 - ▶ **Dynamic load balancing** (idle processes receive pixels during run time) versus **static load balancing** (assign processes to pixels ahead of time).
 - ▶ Static load balancing superior, if done **at random**.
- If we only have to display \mathcal{M} within the rectangle R : use that \mathcal{M} is connected. Dynamic load balancing wins.
 - ▶ We work with a master-slave architecture. Initially a single slave receives R . If the slave finds that all boundary pixels belong to \mathcal{M} , then it “claims” that the rectangle is a subset of \mathcal{M} .
 - ▶ Otherwise the rectangle is returned to the master who partitions it into two rectangles and assigns one slave for each new rectangle.
 - ▶ This procedure continues until all slaves are busy.

Static Load Balancing I

We have used **static load balancing** for all applications in parallel linear algebra, since we could predict the duration of tasks.

When executing **independent** tasks of unknown duration:
assign tasks at random.

Static Load Balancing II

- The static load balancing problem in the Mandelbrot example was easy, since the tasks are independent.
- In general we are given a task graph $\mathcal{T} = (T, E)$.
 - ▶ The nodes of \mathcal{T} correspond to the tasks and
 - ▶ there is a directed edge (s, t) from task s to task t whenever task s has to complete before task t can be dealt with.
 - ▶ We assume an ideal situation in which we know the duration w_t for each task t .
 - ▶ Partition T into p disjoint subsets T_1, \dots, T_p such that processes
 - ★ carry essentially the same load, i.e., $\sum_{t \in T_i} w_t \approx (\sum_{t \in T} w_t)/p$, and
 - ★ communicate as little as possible, i.e., the number of edges connecting tasks in different classes of the partition is minimal.
- The static load balancing problem is \mathcal{NP} -complete and hence computationally hard. Use heuristics (Kernighan-Lin, Simulated Annealing).
- Also, the assumption of known durations is often unrealistic.

Dynamic Load Balancing

- In **centralized load balancing** there is a centralized priority queue of tasks, which is administered by one or more masters assigning tasks to slaves. (cp. APHID).
 - ▶ This approach normally assumes a relatively small number of processes.
 - ▶ Rules of thumb: try to assign larger tasks at the beginning and smaller tasks near the end to even out finish times.
 - ▶ Take different processor speeds into account.
- In **distributed dynamic load balancing** one distinguishes
 - ▶ methods based on **work stealing** or task pulling (idle processes request work) and
 - ▶ **work sharing** or task pushing (overworked processes assign work).
- We concentrate on distributed dynamic load balancing.

Work Stealing

Three methods.

- **Random Polling**: if a process runs out of work, it requests work from a randomly chosen process.
- **Global Round Robin**: whenever a process requests work, it accesses a **global target variable** and requests work from the specified process.
- **Asynchronous Round Robin**: whenever a process requests work, it accesses its **local target variable**, requests work from the specified process and then increments its target variable by one modulo p , where p is the number of processes.

Which method to use?

Comparing the Three Methods

The model:

- Assume that total work W is initially assigned to process 1.
- Whenever process i requests work from process j , then process j **donates** half of its current load and **keeps** the remaining half, provided it still has at least W/p “units of work”.

The question: how long does it take to achieve perfect parallelization, i.e., work $O(W/p)$ for all processes.

An Analysis of Random Polling

Assume that exactly j processes have work.

Let $V_j(p)$ be the expected number of requests such that **each of the j processes** receives at least one request.

- The load of a process is halved after it serves a request.
- After $V_j(p)$ requests, the peak load is at least halved.
- We show that $V_j(p) = O(p \cdot \ln j)$ and hence $V_j(p) = O(p \cdot \ln p)$.
- The peak load is bounded by $O(W/p)$ after $O(p \cdot \ln^2 p)$ requests and hence the communication overhead is bounded by $O(p \cdot \ln^2 p)$.
- We have to determine $V_j(p)$.

Determining $V(p)$

- Assume that exactly i of the j processes with work have received requests. Let $f_j(i, p)$ be the expected number of requests such that each of the remaining $j - i$ processes receives a request.
- Our goal is to determine $f_j(0, p)$.
 - ▶ $f_j(i, p) = (1 - \frac{i}{p}) \cdot (1 + f_j(i, p)) + \frac{i}{p} \cdot (1 + f_j(i + 1, p))$ holds.
 - ▶ Why? With probability $(j - i)/p$ we make progress and one more process has received work, whereas with probability $1 - \frac{i}{p}$ we fail.
 - ▶ Thus $\frac{i}{p} \cdot f_j(i, p) = 1 + \frac{i}{p} \cdot f_j(i + 1, p)$ and hence $f_j(i, p) = \frac{p}{j-i} + f_j(i + 1, p)$.
 - ▶ $f_j(0, p) = \frac{p}{j-0} + \dots + \frac{p}{j-1} + f_j(i + 1, p)$ and as a consequence $f(0, p) = p \cdot \sum_{i=0}^{j-1} \frac{1}{j-i} = p \cdot \sum_{i=1}^j \frac{1}{i}$ follows.
- Hence $V_j(p) = O(p \cdot \ln(j)) = O(p \cdot \ln p)$ and after $O(p \cdot \ln^2 p)$ work requests the peak load is bounded by $O(W/p)$.
- To achieve constant efficiency, $W = \Omega(p \cdot \ln^2 p)$ will do.

Global and Asynchronous Round Robin

- When does global Round Robin achieve constant efficiency?
 - ▶ At least a constant fraction of all processes have to receive work, otherwise the computing time is not bounded by $O(W/p)$.
 - ▶ The global target variable has to be accessed for $\Omega(p)$ steps.
 - ▶ To achieve constant efficiency $\frac{W}{p} = \Omega(p)$ or equivalently $W = \Omega(p^2)$ has to hold.
- The performance of asynchronous Round Robin.
 - ▶ The best case: $p \cdot \log_2 p$ requests suffice and $W = \Omega(p \cdot \log_2 p)$ guarantees constant efficiency.
 - ▶ The worst case: show that $\Theta(p)$ rounds are required. Hence $W = \Omega(p^2)$ guarantees constant efficiency.
- The performance of asynchronous Round Robin is in general better than global Round Robin, since it avoids the bottleneck of a global target variable. However, to avoid its worst case, randomization and therefore random polling is preferable.

Random Polling for Backtracking

We assume the following model:

- An instance of backtracking generates a tree T of height h with N nodes and degree d .
- T has to be searched with p processes. Initially only process 1 is active and it inserts the root of T into its empty stack.
- If at any time an active process takes the topmost node v off its stack, then it **expands** v and pushes all children of v onto the stack. (We use depth-first search to traverse T .)

Thus each stack is composed of generations with the current generation on top and the oldest generation at the bottom.

- An idle process p uses random polling to request work from a randomly chosen process q .

Which tasks should a donating process hand over to the requesting process?

Donating Work

If the donating process q has work, then an arbitrarily chosen request is served and q sends one half of its oldest generation.

- Whenever a node v is donated, then it migrates together with one half of its current siblings.
 - ▶ The generation of v is halved in each donation step involving v .
 - ▶ v participates in at most $\lceil \log_2 d \rceil$ donation steps, where d is the degree of T .
- Thus the communication overhead consists of
 - ▶ at most $O(N \cdot \log_2 d)$ node transfers and
 - ▶ all work requests.
- Any parallel algorithm requires time $\Omega(\max\{N/p, h\})$ to search N nodes with p processes, assuming the tree has height h .

Donating Half of the Oldest Generation: An Analysis I

If less than $p/2$ processes are idle in some given step, then we say that the step **succeeds** and otherwise that it **fails**.

- How many successful steps are performed?
 - ▶ If a process is busy, then it participates in expanding a node or in a node transfer: there are at most $O(N + N \cdot \log_2 d)$ such operations.
 - ▶ There are at most $O(\frac{N}{p} \cdot \log_2 d)$ successful steps.
- We show that there are at most $O((\frac{N}{p} + h) \cdot \log_2 d)$ failing steps.
 - ▶ Fix an arbitrary node v of T .
 - ★ At any time there is a unique process which stores v or the lowest ancestor of v in its stack.
 - ★ We say that v **receives a request**, if “its” process receives a request for work.
 - ▶ After at most $\lceil \log_2 d \rceil \cdot h$ requests for work, v belongs to the oldest generation of its process.
 - ▶ v is expanded after at most $\lceil \log_2 d \rceil$ further requests.

Donating Half of the Oldest Generation: An Analysis II

After how many failing steps does v receive $\lceil \log_2 d \rceil \cdot (h + 1)$ requests?

- Determine the probability q that v receives a request in a failing step.
 - ▶ In a failing step there are exactly k idle processes with $k \geq p/2$.
 - ▶ The probability that none of them requests node v is $(1 - \frac{1}{p})^k \leq (1 - \frac{1}{p})^{p/2} \leq e^{-1/2}$ and $q \geq 1 - e^{-1/2} \geq 1/3$ follows.
- Random polling performs in each failing step a random trial with success probability at least $1/3$ and the expected number of successes in t trials is at least $t/3$.
 - ▶ The Chernoff bound: $\text{prob}[\sum_{i=1}^t X_i < (1 - \beta) \cdot t/3] \leq e^{-(t/3) \cdot \beta^2/2}$.
 - ▶ For $\beta = 1/2$: there are less than $t/6$ successes in t trials with probability at most $e^{-t/24}$.
 - ▶ Set $t = 6 \lceil \log_2 d \rceil \cdot (h + 1 + N/p)$. There are less than $\lceil \log_2 d \rceil \cdot (h + 1)$ requests for v with probability at most $e^{-\Omega(\log_2 d \cdot (h+1+N/p))} = d^{-\Omega(h+N/p)}$.

Summary

- There are at most $O(\frac{N}{p} \cdot \log_2 d)$ successful steps.
- v is expanded after at most $(h + 1) \cdot \lceil \log_2 d \rceil$ requests.
- In $6 \lceil \log_2 d \rceil \cdot (h + 1 + N/p)$ failing steps there are less than $\lceil \log_2 d \rceil \cdot (h + 1)$ requests for v with probability at most $d^{-\Omega(h+N/p)}$.
- We have fixed one node v out of N nodes:

With probability at least $1 - N \cdot d^{-\Omega(h+N/p)}$, random polling runs for at most

$$O(\max\{\frac{N}{p}, h\} \cdot \log_2 d)$$

steps. Its speedup is at least $\Omega(\frac{p}{\log_2 d})$.

Work Sharing

In **randomized work sharing** a busy process assigns work to a randomly chosen process.

- If many processes are busy:
 - ▶ The probability is high that a busy process receives more work in randomized work sharing. Even if the work assignment can be rejected, the communication overhead increases.
 - ▶ In random polling however a busy process profits, since it may get rid of part of its work with minimal communication.
- If few processes are busy:
 - ▶ The probability increases that a busy process finds an idle process to assign work to. The performance of work sharing improves.
 - ▶ Work stealing decreases peak loads as well, but not as efficient as in work sharing.
- “Normally” only few processes are idle and random polling wins. But work sharing is required if urgent tasks have to complete fast.

Extreme Work Sharing

The model: Traverse a backtracking tree of height h . The time to expand a node is unknown.

- Any process has its own pool of nodes.
- Whenever a process needs more work it picks a node from its pool and expands the node. Children of the expanded node are assigned to pools of randomly chosen processes.
- Assume p processes share the total compute time W .
- Let $d = \max_v D(v) / \min_v D(v)$ be the ratio of the longest and shortest duration of a node. Assume $\min_v D(v) = 1$.
- One can show: the total time is proportional to $\frac{W}{p} + h \cdot d$.
 - ▶ Extreme work sharing guarantees a uniform load at the expense of a large communication overhead.
 - ▶ Compare with the total time $O(\max\{\frac{N}{p}, h\} \cdot \log_2 d)$ of random polling. (Here d is the degree of the backtracking tree.)

How to Assign Work?

A model scenario:

- We observe p clients, where the i th client assigns its task at time i to one of p servers.
- All tasks require the same run time, but clients do not know of each others actions.
- We are looking for an assignment strategy that keeps the **maximum load** of a server as low as possible.

- Should we use randomized work sharing?
- Does it pay off to ask more than one server and to choose the server with minimal load?
- Are there even better methods?

The Performance of Randomized Work Sharing

- What is the probability q_k that **some** server receives k tasks?
 - ▶ Assume that k is not too large.
 - ▶ Fix a server S . What is the probability that S receives exactly k tasks?
$$\binom{p}{k} \cdot \left(\frac{1}{p}\right)^k \cdot \left(1 - \frac{1}{p}\right)^{p-k} \approx \binom{p}{k} \cdot \left(\frac{1}{p}\right)^k \approx \left(\frac{p}{k} \cdot \frac{1}{p}\right)^{\Theta(k)} = \left(\frac{1}{k}\right)^{\Theta(k)}.$$
 - ▶ Thus $q_k \leq p \cdot k^{-\Theta(k)}$, since we have fixed one of p servers.
- Choose k such that $p \approx k^{\Theta(k)}$, i.e., $k = \alpha \cdot \frac{\log_2 p}{\log_2 \log_2 p}$ for some constant α .
- Our analysis is tight: the server with the heaviest burden has $\Theta\left(\frac{\log_2 p}{\log_2 \log_2 p}\right)$ tasks with high probability. Compare this with the expected load 1.

Uniform Allocation

Let d be a natural number.

- Whenever a task is to be assigned, choose d servers at random, enquire their respective load and assign the task to the server with smallest load.
 - If several servers have the same minimal load, then choose a server at random.
-
- The maximum load of the uniform allocation scheme is bounded by $\frac{\log_2 \log_2(p)}{\log_2(d)} \pm \Theta(1)$. This statement holds with probability at least $1 - p^{-\alpha}$ for some constant $\alpha > 0$.
 - A significant reduction in comparison to the maximum load $\Theta\left(\frac{\log_2 p}{\log_2 \log_2 p}\right)$ of randomized work sharing.
 - A significant reduction already for $d = 2$: the **two-choice paradigm**.

Non-uniform Allocation

Partition the servers into d groups of same size and assign the task according to the “**always-go-left**” rule:

Choose one server at random from each group and assign the task to the server with minimal load. If several servers have the same minimal load, then choose the **leftmost** server.

- The maximum load is bounded by $\frac{\log_2 \log_2(p)}{d \cdot \log_2(\phi_d)} \pm \Theta(1)$ with $\phi_d \approx 2$. This statement holds with probability at least $1 - p^{-\alpha}$, where α is a positive constant.
- Again a significant improvement compared with the the maximum load $\frac{\log_2 \log_2(p)}{\log_2(d)} \pm \Theta(1)$ for uniform allocation.

Why is Non-uniform Allocation So Much Better?

- Non-uniform allocation seems nonsensical, since servers in left groups seem to get overloaded.
- But in subsequent attempts, servers in right groups will win new tasks and their load follows the load of servers in left groups.
- The combination of the group approach with always-go-left enforces therefore
 - ▶ on one hand a larger load of left servers
 - ▶ with the consequence that right servers have to follow suit.
- The preferential treatment of right groups enforces a **more uniform load distribution**.

- We have used **static load balancing**
 - ▶ for all applications in parallel linear algebra, since we could predict the duration of tasks, and
 - ▶ when assigning independent tasks of unknown duration at random.
- **Dynamic load balancing:**
 - ▶ **work stealing:**
 - ★ idle processes ask for work.
 - ★ Random polling is a good strategy.
 - ▶ **work sharing:**
 - ★ a busy process assigns work to another process.
 - ★ non-uniform allocation (partition into d groups, pick one process per group and apply the “always-go-left” rule) is the most successful strategy.
 - ▶ Work stealing is superior, if there are relatively few idle processes, the typical scenario.