14

Load Balancing

Building a house comprises a large number of individual tasks such as digging a hole, laying the foundations, putting up walls, installing windows, wiring, and pipes, etc. Looking more closely, tasks consist of subtasks. For example, to put up a wall, you have to lay a large number of bricks. If you want to build fast, many workers have to cooperate. You have to plan which tasks can be done in parallel, what their dependencies are, and what resources (qualified workers, raw material, tools, space, ...) are needed. It is often difficult to estimate in advance how long a certain task is going to take. For example, raw materials may be delayed, a worker may fall ill, or a subtask may never have been done before. Hence, plans have to be revised as new information becomes available.

A parallel computation is quite similar to the construction site example above. When designing a parallel algorithm, we identify *tasks* (often also called *jobs*) that can be done in parallel. These tasks are assigned to PEs (workers) with possibly varying capabilities (qualifications of workers). The tasks may depend on each other, and they may require further resources such as memory or communication channels. We may or may not know how long a task is going to take (or what other resource requirements it has). *Load balancing* or *scheduling* is the process of assigning resources to tasks in order to make the overall computation efficient. Since the parallel algorithms in this book are fairly simple, we can usually get away with load-balancing algorithms that are also fast and simple and do not have to deal with all complications of the general problem. Nevertheless, we need several kinds of nontrivial load balancing techniques, which we describe in this chapter.

[0] The photograph shows a construction site in Hong Kong (Ding Yuin Shan `www.flickr.com/photos/90461913@N00/5363967194/in/photostream/`).

14.1 Overview and Basic Assumptions

Except in Sect. 14.6, we assume identical PEs and a set of m independent tasks. A justification for the independence assumption is that parallel algorithms often take dependencies into account explicitly. They split their computation into multiple phases so that the load balancer only has to deal with the independent tasks of a single phase. Section 14.2 deals with the case where we (pretend to) know exactly the work involved in each task. Then we can assign jobs very efficiently using prefix sums. The following three sections deal with the case where the size of the jobs is known only roughly or not at all. A straightforward solution is to choose a *master*, treat the other PEs as *workers* and let the master assign the jobs to the workers; see Sect. 14.3. However, this *master–worker scheme* does not scale well. When the number of jobs is large, Sect. 14.4 shows that it is quite effective to simply distribute the jobs randomly. Finally, Sect. 14.5 introduces a highly scalable dynamic load-balancing algorithm that can deal with widely varying and completely unknown task sizes provided that tasks can be split into subtasks when necessary. The idea is that idle PEs *steal* work from randomly chosen busy PEs. If an idle PE wants to steal work from a busy PE, the task at the busy PE is split into two subtasks, and the idle PE takes over one of the subtasks.

14.1.1 Independent Tasks

Many of the algorithms described below can be understood and analyzed as assigning m independent tasks to p PEs. Let t_j denote the size (or execution time) of task $j \in [m]$. We use the following notation in the discussion of the algorithms: the total execution time $T := \sum_i t_i$, the maximum task size $\hat{t} := \max_j t_j$, the average task size $\bar{t} := T/m$, and the *load* ℓ_i of PE i – the total size of the tasks assigned to PE i. The goal is to minimize the maximum load $\hat{\ell} := \max_i \ell_i$ (see also Sect. 12.2). Figure 14.1 illustrates these quantities.

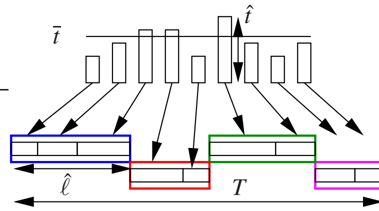


Fig. 14.1. An example of the quantities $\bar{t}$, $\hat{t}$, $\hat{\ell}$, and T that characterize sets of independent tasks. The large horizontal rectangles indicate PEs.

Table 14.1 compares the scalability of the load balancing algorithms presented below for independent jobs in the distributed-memory model, assuming that jobs are fully characterized by their ID number. We also ignore the cost for collecting the result. Clearly, it is easier to balance the load if there is more total work to be distributed. We want the algorithms to guarantee $T_{\text{par}} \leq (1+\varepsilon)T/p$, i.e., optimal

Table 14.1. Scalability of four load-balancing algorithms for independent tasks. The second column shows the minimum required total work $T = \sum_j t_j$ for the algorithm to guarantee $T_{\text{par}} \le (1+\varepsilon)T/p$. The startup overhead α is treated as a variable.

Algorithm	Sect.	$T = \Omega(\cdots)$	Remarks
prefix sum	14.2	$\frac{p}{\varepsilon}(\hat{t} + \alpha \log p)$	known task sizes
master–worker	14.3	$\frac{p}{\varepsilon} \cdot \frac{\alpha p}{\varepsilon} \cdot \frac{\hat{t}}{\bar{t}}$	bundle size $\sqrt{\frac{m\alpha}{\hat{t}}}$
randomized static	14.4	$\frac{p}{\varepsilon} \cdot \frac{\log p}{\varepsilon} \cdot \hat{t}$	randomized
work stealing	14.5	$\frac{p}{\varepsilon}(\hat{t} + \alpha \log p)$	randomized

balance up to a factor $1+\varepsilon$. We list for each algorithm the minimum total work required for this guarantee.

The prefix sum method is our "gold standard" – a simple method that is hard to improve upon if we know the task sizes. The other methods do not require task sizes but are more expensive in most cases. Work stealing comes within a constant factor of the gold standard in expectation. Randomized static load balancing behaves well when the tasks are very fine-grained, i.e., $\hat{t}/\varepsilon \ll \alpha$. Furthermore, randomized load balancing is very simple and can sometimes be done without explicitly moving the data. The master–worker scheme is the "ugly duckling" in this asymptotic comparison. It needs work quadratic in p, whereas the other methods only need work $p \log p$ and there are additional factors that can be big for large values of α, $1/\varepsilon$, or $\hat{t}/\bar{t}$. We include the master–worker scheme because it is simple and very popular in practice.

***Exercise 14.1.** Prove the bounds given in Table 14.1 using the results discussed in Sections 14.2–14.5.

14.1.2 A Toy Example – Computing the Mandelbrot Set

We shall use a popular example from recreational mathematics as an example in Sect.s 14.4 and 14.5. In part, the discussion is similar to that in the book [280] including some of the images.

For every complex number $c \in \mathbb{C}$, we define the sequence $z_c(m) : \mathbb{N} \to \mathbb{C}$ as $z_c(0) := 0$ and $z_c(m+1) := z_c(m)^2 + c$. The Mandelbrot set M is then defined as

$$M := \{c \in \mathbb{C} : z_c(m) \text{ is bounded}\} .$$

We want to know which points from a quadratic subset $A = \{z_0 + x + \mathrm{i}y : x, y \in [0, a_0]\}$ of the complex plane belong to M. We shall approximate this question in two respects. First, we sample $n \times n$ grid points from this set (see Fig. 14.2): $c(j,k) := z_0 + (j + \mathrm{i}k)a_0/n$ with $j,k \in 0..n-1$. Second, we consider only the first $\hat{t}$ members of the sequence $z_{c(j,k)}$. We exploit the mathematical fact that z_c is unbounded if $|z_c(t)| > 2$ for some t. Conversely, if $|z_c(t)| \le 2$ for all $t \in 1..\hat{t}$, we assume that $c \in M$. Figure 14.2 shows an approximation to M.

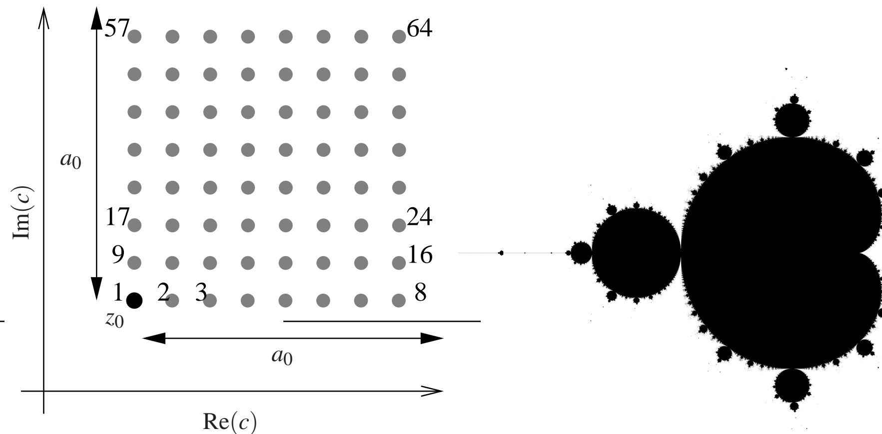


Fig. 14.2. *Left*: The points of the complex plane considered by our algorithm, and how they are numbered. *Right*: Approximation to the Mandelbrot set M.

There are quite sophisticated algorithms for computing the above approximation (e.g., [250]). We shall consider only a very simple method. We have $n \times n$ independent tasks. To simplify the description, we shall enumerate the tasks from 1 to $m = n^2$ as given in Fig. 14.2. The task for $c(j,k)$ iterates through $z_{c(j,k)}(t)$ and stops when this value becomes larger than 2 or when $t = \hat{t}$. Note that we have no a priori knowledge of how long a task will take.

14.1.3 Example – Parallel Breadth-First Search

An example, where we have a good estimate for the job size is parallel BFS in graphs (Sect. 9.2.4). In each iteration of the BFS algorithm, each node in the queue Q represents a job whose execution time is (roughly) proportional to the degree of the node. If some nodes have very high degree, we may have to design the implementation so that jobs are splittable, i.e., multiple PEs cooperate in exploring the edges out of a node.

14.2 Prefix Sums – Independent Tasks with Known Size

Suppose we have to assign m independent tasks with known sizes (execution times) t_j, $1 \leq j \leq m$. We first assume that tasks can be split between PEs and later remove this assumption. If tasks can be split, a simple idea works. We visualize task j as an interval of length t_j, arrange the intervals next to each other on a line segment of length $T = \sum_i t_i$, and cut the line segment into pieces of equal length, one for each PE. To get closer to an actual implementation, we assume from now on that the t_j's are integers and that we want to split only at integer positions.

Let $s = \lceil T/p \rceil$. We view the line segment as consisting of T unit intervals numbered starting at 1 and make PE i responsible for the unit intervals numbered $(i-1)\cdot s+1$ to $i\cdot s$. Task j corresponds to the unit intervals numbered T_j+1 to T_j+t_j, where $T_j = \sum_{k<j} t_k$ denotes the exclusive prefix sum up to task j. In this way, PEs $\lfloor T_j/s \rfloor + 1$ to $\lceil (T_j+t_j)/s \rceil$ are collectively responsible for task j. Note that $(i-1)s+1 \le T_j+1$ iff $i \le T_j/s+1$ iff $i \le \lfloor T_j/s \rfloor + 1$, where the last "iff" uses the fact that i is an integer. Similarly, $i\cdot s \ge T_j+t_j$ iff $i \ge (T_j+t_j)/s$ iff $i \ge \lceil (T_j+t_j)/s \rceil$.

Exercise 14.2. Show that task j is split into at most $\lceil t_j/s \rceil + 1$ pieces.

To compute the assignment in parallel, we essentially have to compute the prefix sums T_j. We can reduce this computation to local operations plus a prefix sum over one value per PE. This global prefix sum is discussed in Sect. 13.3. We use Brent's principle (see Sect. 2.10). Suppose PE i is responsible for assigning tasks $a_i..b_i$. PE i first locally calculates the local sum $s_i := \sum_{j\in a_i..b_i} t_j$. Then the global prefix sum $S_i := \sum_{k<i} s_k$ is computed. In a second pass over the local data, PE i computes $T_j = S_i + \sum_{a_i \le k < j} t_k$ for $j \in a_i..b_i$. Overall, this takes time $O(\alpha \log p + \max_i(b_i - a_i))$ on a distributed-memory machine. On top of that, we may need time to move the tasks to the PEs they are assigned to. This depends on the size of the task descriptions, of course. We can, however, exploit the fact that consecutive tasks are moved to consecutive PEs.

Atomic Tasks. When tasks are nonsplittable, we face the scheduling problem already discussed in Sect. 12.2. "Atomic" is a synonym for "nonsplittable". While the basic greedy algorithm is inherently sequential, we can obtain a scalable parallel algorithm with similar performance guarantees by interpreting the prefix sums calculated for the splittable case differently. We can simply assign all of task j to PE $\lfloor T_j/s \rfloor + 1$ (see Fig. 14.3 for an example), i.e., instead of assigning the task to an interval of PEs starting at PE $\lfloor T_j/s \rfloor + 1$, we assign the entire task to this PE. This leads to good

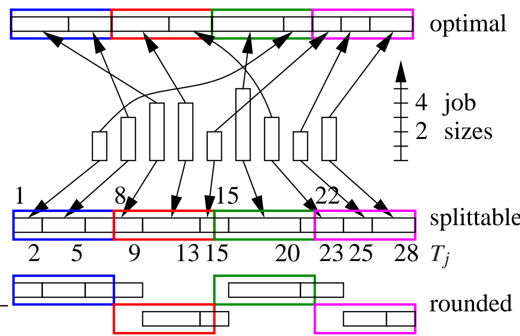


Fig. 14.3. Assigning 9 tasks to 4 PEs. We have $T = 28$ and $s = 28/4 = 7$. The yardstick on the right indicates the job sizes. Three assignments are shown. *Top*: optimal, with $\hat{\ell} = 7$. *Middle*: using prefix sums assuming jobs can be split, $\hat{\ell} = 7$. *Bottom*: using prefix sums and rounding, $\hat{\ell} = 9$. Note that the first three jobs are assigned to PE 1 since $\lfloor T_3/s \rfloor + 1 = \lfloor 5/7 \rfloor + 1 = 1$.

load balance when the task sizes are much smaller than the contingent s of each PE. Indeed, prefix-sum-based assignment represents a two-approximation algorithm for the problem of assigning tasks to identical PEs.

Exercise 14.3. Prove the above claim. Hint: Recall that ℓ_i denotes the load of PE i and that $\hat{\ell} = \max_i \ell_i$ and $\hat{t} = \max_j t_j$. Show the following three facts and use them to prove that the prefix-sum-based assignment is within a factor two of optimal:

(a) $\hat{\ell} \leq s + \hat{t}$ for the prefix-sum-based algorithm.
(b) $\hat{\ell} \geq \hat{t}$ for any schedule of atomic tasks.
(c) $\hat{\ell} \geq s$ for any schedule.
(d) If $\hat{t} \leq \varepsilon s$, the prefix-sum-based assignment is within a factor of $1 + \varepsilon$ of optimal.

14.3 The Master–Worker Scheme

One of the most widely used dynamic load-balancing algorithms uses a dedicated *master* PE to deal out tasks to *worker* PEs. Initially, the master has all tasks, and all workers are idle. An idle worker sends a work request to the master. The master waits for requests and sends one task to the requesting PE. When all tasks have been sent, the master informs requesting PEs about this. Figure 14.4 illustrates this scheme. The master–worker scheme is easy to implement even with a rudimentary communication infrastructure. This scheme may also be a good choice if the tasks are generated by a sequential (legacy) algorithm or are read from a centralized I/O device. Another advantage is that even *without* knowing the task sizes, good load balance is achieved – disregarding the scheduling overhead, one will get the same balancing quality as with the prefix-sum-based algorithm presented in the previous section.

Load balancing can be further improved if at least an estimate of the task sizes is available. It is then a good idea to deal out large tasks first. Exercise 12.8 mentions an approximation ratio of $4/3$ in a similar situation.

Unsurprisingly, the main disadvantage of the master–worker scheme is that it does not scale to large p. Unless the average task size is much larger than αp, the startup overheads alone will already limit performance. If there is a large number

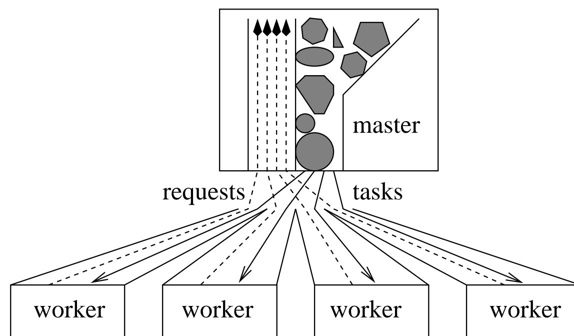


Fig. 14.4. The master–worker load-balancing scheme

of small tasks, one can mitigate this effect by dealing out *task bundles* – multiple tasks at a time. Unfortunately, this introduces a trade-off between load imbalance and scheduling overhead. Let us analyze this for a fixed bundle size b and small task descriptions. The master has to spend time $\geq \alpha m/b$ on sending out task bundles. The last bundle sent out might incur work $b\hat{t}$. Hence the parallel execution time is at least $\alpha m/b + b\hat{t}$. Using calculus, we can determine the optimal bundle size as $b = \sqrt{\alpha m/\hat{t}}$. Note that this consideration is only useful if we have a reasonable estimate of $\alpha/\hat{t}$. A somewhat better but more complicated strategy is to change the bundle size dynamically – beginning with a large bundle size and decreasing it as the computation progresses. It can be argued that a near-optimal strategy is to try constructing a bundle involving total work $W/(pC) + W_{\min}$ where W is an estimate of the total work not yet handed out and where C and $W_{\min}$ are tuning parameters that depend on the accuracy of the work estimation [29].

****Exercise 14.4.** Investigate whether using this strategy can improve the scalability of the master–worker scheme in shown in Table 14.1 so that the factor $1/\varepsilon^2$ is replaced by $\frac{1}{\varepsilon}\log\frac{1}{\varepsilon}$.

A more sophisticated approach to making the master–worker scheme more scalable is to make it hierarchical – an overall master deals out task bundles to submasters, which deal out smaller bundles to subsubmasters, etc. Unfortunately, this destroys the main advantage of the basic scheme – its simplicity. In particular, we get tuning parameters that are harder to control. When you are in the situation that a basic scheme does not work well. Thinking about clean new strategies is often better than making the basic scheme more and more complicated. For example, one could consider the relaxed FIFO queue described in Sect. 3.7.2 as a truly scalable means of producing and consuming tasks.

Shared-Memory Master–Worker. If the tasks are stored in an array $t[1..m]$, a simple version of the master–worker scheme can be implemented without an explicit PE for the master. We simply use a shared counter to indicate the next task to be delegated. This also has limited scalability for large p, but the constant factors are much better than in the distributed-memory case.

The Mandelbrot Example. Dealing out individual tasks is not very scalable here, in particular, if most of the area under consideration is not in the Mandelbrot set M. In that case, most of the iterations may terminate after a small number of iterations. Bundling multiple tasks is easy in the sense that we can describe them succinctly as a range of task numbers, which we can easily convert into the starting number z.

Exercise 14.5. Develop a formula $c(i)$ that computes the parameter c of the sequence z_c from a task number i.

Furthermore, we can derive an upper bound on the execution time of a task from the maximum iteration count $\hat{t}$. However, estimating the size of k tasks as taking $k\hat{t}$ iterations may grossly overestimate the actual execution time, whereas using more aggressive bounds may be wrong when all tasks in a bundle correspond to members of M. We conclude that using the master–worker scheme with many PEs can cause major headaches even for trivial applications.

14.4 (Randomized) Static Load Balancing

A very simple and seemingly bad way to do load balancing is to simply assign a similar number of jobs to each PE without even considering their size. Figure 14.5 illustrates three different such static assignment strategies, which we shall now discuss for the Mandelbrot example. The most naive way to implement this idea is to map n/p consecutively numbered tasks to each PE or, even simpler, to assign a fixed number of rows to each PE. This makes displaying the results easy but looking at Fig. 14.2, one can see that this is a bad idea – some rows have many more elements of M than others, leading to bad load balance. A slightly more sophisticated approach is *round-robin* scheduling – task j is mapped to PE $j \bmod p$. For the Mandelbrot example, this is likely to work well. However, we get no performance guarantee in general. Indeed, for every fixed way to map tasks to PEs, there will be distributions of task sizes that lead to bad load balance.

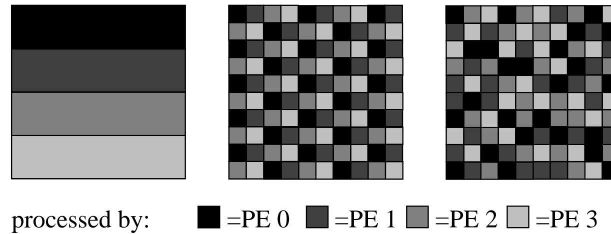
Once more, randomization allows us to solve this problem. One can show that, for arbitrary distributions of task sizes, a random assignment of tasks to PEs leads to good load balance with high probability if certain conditions are met.

Theorem 14.1 ([267, 268]). *Suppose m independent tasks of size $t_1, \ldots, t_m$ are mapped to random PEs. Let $\hat{t} := \max_j t_j$ and $T := \sum_j t_j$. Then the expected maximum load $\mathrm{E}[\hat{\ell}]$ of a PE is no worse than for the case where we randomly map $\lceil T/\hat{t}\rceil$ tasks of size $\hat{t}$. In particular, for any constant $\varepsilon > 0$,*

$$\mathrm{E}[\hat{\ell}] \leq (1+\varepsilon)\frac{T}{p} \quad \textit{if} \quad \frac{T}{\hat{t}} \geq \frac{c}{\varepsilon^2} p \log p \quad \textit{for some appropriate constant } c.$$

This is a very good bound if $\hat{t}$ is not too much larger than the average task size and if we can live with a large value of the imbalance ε. Decreasing the imbalance requires increasing the input size as $1/\varepsilon^2$.

One strength of randomized static load balancing is that a random distribution of the input is often enough to enforce a random distribution of the work *in every phase of a computation*. For example, consider the distributed-memory BFS discussed in Sect. 9.2.3. By assigning the nodes to random PEs, we ensure good expected load balance in every layer of the BFS algorithm which considers a sufficiently large number of edges, namely $\Omega(\Delta p \log p)$, where Δ is the maximum degree of the graph.



processed by: ■ =PE 0 ■ =PE 1 ■ =PE 2 □ =PE 3

Fig. 14.5. Block decomposition (*left*), round-robin (*middle*), and random decomposition (*right*) of a two-dimensional range

14.4.1 *Using Pseudorandom Permutations to Avoid Communication

In some situations, randomized static load balancing can be implemented essentially without communication (beyond broadcasting a description of the input). Consider the Mandelbrot example. Initially, we broadcast the parameters n, z_0, and a_0 describing the input. From these parameters and a task index $j \in 1..n$ we can reconstruct the tasks without communication. Hence, each PE needs only to know the indices of the tasks mapped to it. At first glance, this requires computing a (pseudo)random mapping $f : 1..n \to 1..p$ and communicating every i to PE $f(i)$. We can avoid the communication by using a (pseudo)random *permutation* $\pi : 1..n \to 1..n$ rather than a random function and making PE i responsible for tasks $\pi(j)$ for $j \in (i-1)\lceil n/p \rceil + 1..i\lceil n/p \rceil$. Effectively, this means that tasks are mapped using the inverse permutation π^{-1} of π. Since the inverse of a random permutation is also random, this means that the jobs are mapped using a random permutation rather than a random mapping. This is not the same. For example, a random permutation maps at most $\lceil n/p \rceil$ tasks to each PE, whereas a random mapping will, in general, allocate tasks less evenly. It can be shown that such a random-permutation mapping is at least as good as a plain random mapping [169, 267].

Although random permutations can be generated in parallel [269], this also requires communication. However, there are simple algorithms for computing *pseudorandom* permutations directly. *Feistel* permutations [236] are one possible approach. Assume for simplicity that $\sqrt{n}$ is an integer[1] and represent j as $j = j_a + j_b\sqrt{n}$. Now consider the mapping

$$\pi_k((j_a, j_b)) = (j_b, j_a + f_k(j_b) \bmod \sqrt{n}),$$

where $f_k : 1..\sqrt{n} \to 1..\sqrt{n}$ is a hash function.

Exercise 14.6. Prove that π_k is a permutation.

It is known that a permutation $\pi(x) = \pi_1(\pi_2(\pi_3(\pi_4(x))))$ built by chaining four Feistel permutations is "pseudorandom" even in a cryptographic sense if the f_k's are sufficiently random. The same holds if the innermost and outermost permutations are replaced by even simpler permutations [236]. This approach allows us to derive a pseudorandom permutation of $1..n$ from a small number of hash functions $f_k : 1..\sqrt{n} \to 1..\sqrt{n}$. When $\sqrt{n} = \mathrm{O}(n/p)$, it is even feasible to use tables of random numbers to implement f_k.

14.5 Work Stealing

The load-balancing algorithms we have seen so far are simple but leave a lot to be desired. prefix-sum-based scheduling works only when we know the task sizes. The

[1] For general n, we can round n up to the next square number and generate corresponding empty jobs.

master–worker scheme suffers from its centralized design and the fact that task bundles may commit too much work to a single PE. Randomized static load balancing is not dynamic at all.

We now present a simple, fully distributed dynamic load-balancing algorithm that avoids these shortcomings and, in consequence, guarantees much better performance. This approach can even handle dependent jobs and fluctuating resource availability [20]. However, to keep things simple and understandable, we describe the approach for independent computations and refer the reader to the literature and Sect. 14.6 for more.

14.5.1 Tree-Shaped Computations

We use a simple but rather abstract model of the application – *tree-shaped computations* [272]. Initially, all the work is subsumed in a *root* task J_{root}. We can perform two basic operations on a task – *working* on it sequentially, and *splitting* it. When sequential processing finishes a task, we are left with an *empty* task $J_\emptyset$. Splitting replaces a task by two tasks representing the work of the parent task. Very little is assumed about the outcome of a task split, in particular, the two subproblems may have different sizes. One subproblem may even be empty. In that case, the parent task is *atomic*, i.e., unsplittable. Initially, the root task resides on PE 1. The computation terminates when all remaining tasks are empty. Figure 14.6 illustrates the concept. A tree-shaped computation can be represented by a binary tree. Each node of this tree consists of some (possibly zero) sequential computation, followed by a split (for internal nodes) or by a transition to an empty task (for leaves).

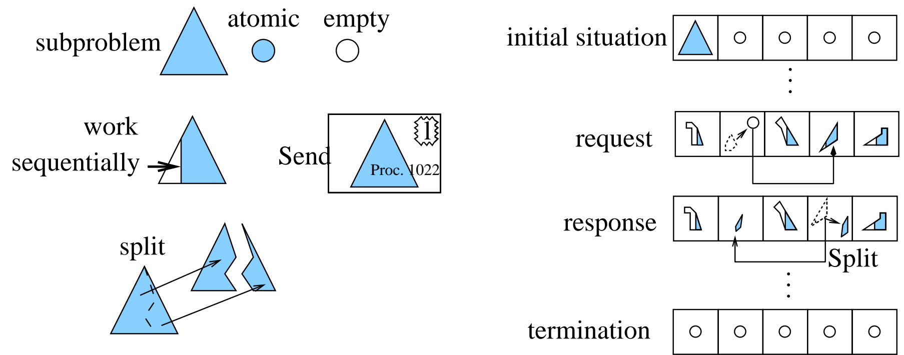


Fig. 14.6. Tree-shaped computations

Tree-shaped computations subsume many applications. For example, when applied to the independent job model, we obtain a fully distributed multilevel generalization of the master–worker scheme where every PE is both a master and a worker. We can also handle applications such as the tree exploration described in Sect. 12.4.2 for the knapsack problem, where we do not have a fixed set of jobs up front.

In order to analyze the performance of a load-balancing algorithm for tree-shaped computations, we need to model the cost of splitting and moving tasks. We also need some way to measure the performance of the splitting operation. This performance is expressed by a bound t_{atomic} on the size of an atomic task and by the *splitting depth* h, which is the maximum number of subsequent splits needed to arrive at an atomic task. In other words, h is a bound on the height of the task tree, disregarding splits of atomic tasks.

Example: Independent tasks. These can be translated into tree-shaped computations as follows: Let the range $a..b$ represent the task of processing the (atomic) tasks numbered a to b. Then the root problem is $J_{\text{root}} = 1..m$. Splitting $a..b$ yields $a..\lfloor(a+b)/2\rfloor$ and $\lfloor(a+b)/2\rfloor + 1..b$. Atomic tasks have size at most $t_{\text{atomic}} = \hat{t}$, and the splitting depth is $h = \lceil \log m \rceil$.

14.5.2 The Randomized Work-Stealing Algorithm

We now describe a simple and efficient load-balancing algorithm for tree-shaped computations. This algorithm maintains the invariant that each PE works on exactly one task. Initially, this means that PE 1 works on the root task J_{root} while all other PEs have an empty task $J_{\emptyset}$.

Exercise 14.7. Work out the details of an optional fast initialization for the work-stealing algorithm that first broadcasts J_{root} and then locally splits it $\lfloor \log p \rfloor$ or $\lceil \log p \rceil$ times, each time throwing away either the first or the second subproblem. Afterwards, the locally present tasks should still represent the entire computation.

After initialization, a busy PE (one working on a nonempty task) works on it sequentially. An idle PE i, i.e., one with an empty task, sends a work request to another, random PE and waits for a reply. This is repeated until PE i receives a nonempty task. A PE receiving a work request splits its local task, keeps one part, and sends the other part to the requesting PE. Idle PEs or PEs working on an atomic task may not actually call the general splitting routine and also need not send a full-blown task description. Rather, they can simply respond with a rejection. However, we prefer to interpret this rejection as a compact representation of an empty task in order to underline the simplicity of the algorithm.

We still need to discuss one issue – how can we make a PE working sequentially respond to requests? On a distributed-memory machine, one solution is to periodically check for incoming messages during sequential work. On a shared-memory machine, the task description could be a concurrent data structure that allows concurrent splitting and sequential work. For example, our “packaging” of independent tasks into tree-shaped computations could implement the required operations by simple atomic updates of the range $a..b$ of unprocessed tasks. Working sequentially means, most of the time, working on task $a-1$. When this task is finished, a is atomically incremented. Work stealing atomically updates a to $\lfloor(a+b)/2\rfloor + 1$. The same approach can also be used on a distributed-memory machine if, on each node, we keep a separate thread for handling incoming messages.

We shall not give a complete analysis of the asynchronous work-stealing algorithm described above, but instead analyze a related synchronous algorithm. Nevertheless, this analysis still illuminates the essential ideas behind the general analysis. We consider a *random shift* algorithm. This algorithm works in synchronized *rounds* of duration Δ, large enough to allow sending a request, splitting a subproblem, and sending one piece back to the requestor. A PE i that is idle at the beginning of a round sends a request to PE $(i+k \bmod p)+1$, where $k \in 1..p-1$ is a uniformly distributed random value that is the *same* on all PEs. The receiving PE answers this request by splitting its local subproblem and sending one of the pieces to the requesting PE. Note that each PE receives at most one request per round, since all PEs use the same k. We make the further simplifying assumption that busy PEs that do *not* receive a request in a round are in no way delayed by the load-balancing process. In other words, in a round an unfinished nonatomic task is either split or completed, or its unfinished work is reduced by Δ. If the task is split, the unfinished work is distributed over the two subtasks. An atomic task is either finished in a round or its unfinished work is reduced by Δ. The task tree has depth h and hence at most 2^h leaves. Each leaf stands for an atomic task, and atomic tasks require at most work t_{atomic}.

Theorem 14.2. *For*[2] $h \geq \log p$ *and any constant* $d > 0$*, the random shift algorithm has an execution time bounded by*

$$\frac{T_{\text{seq}}}{p} + (3ch+3)\Delta + t_{\text{atomic}} = \frac{T_{\text{seq}}}{p} + \mathrm{O}(t_{\text{atomic}} + h\Delta)$$

with probability at least $1 - p^{-d}$*. Here,* $c = 2 + 2(d+1)\ln 2$.

Proof. Let

$$k = \left\lceil \frac{T_{\text{seq}}}{p\Delta} + 3ch \right\rceil + 1. \tag{14.1}$$

We shall show that, with probability at least $1 - p^{-d}$, all unfinished tasks are atomic after k rounds. Assume this is the case. Then the parallel execution time is at most $k\Delta + \lceil t_{\text{atomic}}/\Delta \rceil \Delta \leq (k+1)\Delta + t_{\text{atomic}}$.

Suppose there are m_i idle PEs at the beginning of round i. Each idle PE issues a split request, and each such request may hinder a PE that is working on a task. Moreover, up to m_{i+1} PEs may become idle during that period and hence not do the full amount Δ of work. We conclude that in round i, at least $p - 2m_i - m_{i+1}$ PEs do Δ units of useful sequential work each. Let $M := \sum_{1 \leq i \leq k} m_i$. Since the useful sequential work accounted for in this way cannot be more than the sequential execution time, we obtain

$$T_{\text{seq}} \geq \sum_{i=1}^{k-1} (p - 2m_i - m_{i+1})\Delta \geq (k-1)p\Delta - 3\Delta M.$$

Thus

[2] Note that the case $h < \log p$ is not interesting, since the number of nontrivial tasks generated is at most 2^h and hence some PEs will always stay idle.

$$k \leq 1 + \frac{T_{\text{seq}} + 3\Delta M}{p\Delta}. \tag{14.2}$$

In combination with (14.1), we obtain $M \geq chp$.

Consider any nonempty task t after round k. Let the random variable $X_i \in \{0,1\}$ express the event that the ancestor of t was split in round i. Then t is the result of $X := \sum_i X_i$ splits. If $X \geq h$, t is atomic. We have $\text{prob}(X_i = 1) = m_i/(p-1)$ since in round i, m_i split requests are issued and for each split request there is one choice of k that will lead to a split of the ancestor of t. Thus $\text{E}[X] = \sum_i m_i/(p-1) \geq M/p$.

We want to use Chernoff bounds (see Sect. A.3) to bound the probability that X is less than h. For this, we require that the X_i's are independent. They are not, because m_i and hence the probability that X_i is equal to 1 depend on the preceding rounds. However, for any fixed choice of the m_i's, the X_i's are independent (they depend only on the random shift values). Hence, we may apply Chernoff bounds if we afterwards take the worst case over the choices of m_i's. We shall see that only the sum of the m_i's is relevant and hence we are fine.

Let $\varepsilon = 1 - 1/c$. Then $(1-\varepsilon)\text{E}[X] \geq 1/c \cdot M/p \geq h$, and hence the probability that X is fewer than h is at most $\gamma := e^{-(1-1/c)^2 ch/2}$. The probability that *any* task is split less than h times is then bounded by $2^h\gamma$. Recall that there are at most 2^h leaf tasks. We show that $2^h\gamma \leq p^{-d}$ for our choice of c. For this it suffices to have (take natural logarithms on both sides)

$$-h\ln 2 + \frac{1}{2}\left(1 - \frac{1}{c}\right)^2 ch \geq d\ln p.$$

Since $h \geq \log p = (\ln p)/\ln 2$, this holds true for the choice $c = 2 + 2(d+1)\ln 2$. Namely,

$$\frac{1}{2}\left(1 - \frac{1}{c}\right)^2 ch \geq \frac{1}{2}(c-2)h \geq (d+1)h\ln 2 \geq d\ln p + h\ln 2. \qquad \square$$

Let us apply the result to allocating m independent tasks in order to establish the entry in Table 14.1. We have $h = \lceil \log m \rceil$, $t_{\text{atomic}} = \hat{t}$ and $\Delta = \text{O}(\alpha)$. Thus, Theorem 14.2 yields a parallel execution time $T/p + \text{O}(\hat{t} + \alpha \log m)$. If $m = \Omega(p^2)$, the term $T/p = \bar{t}m/p$ asymptotically dominates the term $\alpha \log m$. Then

$$\frac{T}{p} + \text{O}(\hat{t}) \leq (1+\varepsilon)\frac{T}{p} \quad \Leftrightarrow \quad \text{O}(\hat{t}) \leq \varepsilon\frac{T}{p} \quad \Leftrightarrow \quad T = \Omega(p\hat{t}/\varepsilon).$$

If $m = \text{O}(p^2)$, we have $\log m = \text{O}(\log p)$ and hence

$$\frac{T}{p} + \text{O}(\hat{t} + \alpha\log p) \leq (1+\varepsilon)\frac{T}{p} \quad \Leftrightarrow \quad T = \Omega\left(\frac{p}{\varepsilon}(\hat{t} + \alpha\log p)\right).$$

14.5.3 Termination Detection

The randomized work-stealing algorithm as described above does not detect when all PEs have become idle – they will simply keep sending work requests in vain.

One solution is that a PE i receiving a task t is responsible for reporting back to the PE from which t was received when all the work in t has been performed. In order to do this, PE i has to keep track of how many subtasks have been split away from t. Only when termination responses for all these subtasks have been received and the remaining work has been performed sequentially, can PE i send the termination response for t. Ultimately, PE 1 will receive termination responses for the subtasks split away from J_{root}. At this point it can broadcast a termination signal to all PEs. The overhead for this protocol does not affect the asymptotic cost of load balancing – the termination responses mirror the pieces of work sent. The final broadcast adds a latency $O(\alpha \log p)$. This termination detection protocol can also be used to perform final result calculations.

**Double Counting Termination Detection.* Even faster termination detection is possible if we adapt a general purpose termination detection protocol suitable for asynchronous message exchange. Mattern [208] describes such a method. We count the number of received and completed tasks locally. On PE 1, we initially set the receive counter to 1 for the root problem. All other counters are initially 0. We repeatedly run an asynchronous reduction algorithm to determine the global sums of these two counters (see also Sect. 13.7). The corresponding messages are sent when a PE is idle and has received the messages from its children in the reduction tree. If the global sums of sent and completed subproblems are equal *and* the next cycle yields exactly the same global sums, then the system has globally terminated. If we replace the linear-latency summation algorithms given in [208] with our logarithmic algorithms, we get overall latency $O(\log p)$ for the termination detection.

14.5.4 Further Examples: BFS and Branch-and-Bound

We discuss two further examples here.

Breadth-First Search. On a shared-memory machine, we can use work stealing as a load balancer in each iteration of BFS.[3] The simple variant in Sect. 9.2.1 that parallelizes over the nodes in the queue could view each node in the queue as an independent job. We get $h = \lceil \log n \rceil$ and t_{atomic} proportional to the maximum node degree.

The algorithm for handling high-degree nodes in Sect. 9.2.4 could be adapted for work stealing by making tasks splittable even when they only work on a single node. In principle, an atomic task could simply represent the operations performed on a single edge. This might be too fine-grained, though, since there is some overhead for reserving an edge for sequential processing. We can amortize this overhead by always assigning a bundle of k edges to a PE. Theorem 14.2 tells us that this has little effect on the load-balancing quality as long as $k = O(\log n)$.

[3] On distributed memory, global work stealing would require more communication than we can afford. We could use randomized static load balancing to assign graph nodes to compute nodes. Cores within a compute node could use work stealing as described here.

Branch-and-Bound for the Knapsack Problem. Let n be the number of items. The algorithm backtracks through a binary tree of depth n that enumerates all solutions that cannot be excluded using upper and lower bounds (see Sect. 12.4.2). A task is a subtree of this tree. By always splitting off the highest unexplored subtree of a task, we obtain a splitting depth of n. For describing a split off at level k of the tree, it suffices to transfer a bit vector of length k indicating which decisions have been made up to level k.

14.6 Handling Dependencies

A quite general way to specify a parallel computation is to view it as a directed acyclic graph (DAG) $G = (V, E)$. Tasks are nodes of this graph and edges express *dependencies*, i.e., an edge (u, v) indicates that v can only be executed after u has been completed. As before, let t_v denote the execution time of task v. This is a natural generalization of our model of independent tasks considered in Sect. 14.1.1 and also in Sect. 12.2. We have already seen task DAGs for integer multiplication (Fig. 1.5) and for tiled computation on arrays (Fig. 3.2). Even disregarding data flow between the tasks, it is already an interesting question how one should assign the tasks to the PEs. It is **NP**-hard to find a schedule that minimizes the parallel execution time (the *makespan*). On the other hand, any "reasonable" scheduling algorithm achieves a two-approximation.

In order to explain this statement in more detail, we need to introduce some terminology. A schedule x assigns each task j to a PE $p_j \in 1..p$ and specifies a starting time $s_j \geq 0$. A schedule is feasible if the task execution times on any particular processor do not overlap (i.e., $\forall i, j : p_i = p_j \Rightarrow s_j \notin [s_i, s_i + t_i)$) and no task starts before it is ready for execution (i.e., $\forall j : \forall (i, j) \in E : s_j \geq s_i + t_i$). The makespan is the last finishing time, $\max_j(s_j + t_j)$, of a job. We are looking for a schedule with minimum

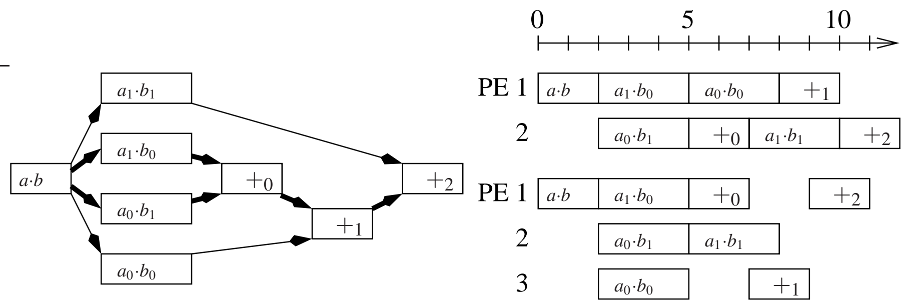


Fig. 14.7. Two optimal schedules for the recursive multiplication algorithm of Sect. 1.4 for two-digit numbers. One schedule is for two PEs and the other for three PEs. It is assumed that a recursive call and an addition/shift cost 2 units of time while a multiplication costs 3 units. The thick edges indicate the critical paths (of length 11). A schedule drawn as above with one line per PE is also known as a *Gantt chart*.

makespan. A trivial lower bound for the makespan is the average work $\sum_j t_j/p$. Another lower bound is the length $\sum_{j \in P} t_j$ of any path P in G. The *critical path length* is the maximum such length over all paths in G. Figure 14.7 gives an example.

Theorem 14.3. *Consider any schedule that never leaves a PE idle when a task is ready for execution. Then its makespan is at most the average work plus the critical path length. This is a two-approximation of the optimal schedule.*

Proof. Let $G = (V, E)$ be the scheduling problem and let T denote the makespan of the schedule. Partition $[0, T]$ into (at most $2|V|$) intervals $I_1, \ldots, I_k$ such that jobs start or finish only at the beginning or end of an interval. Call an interval *busy* if all p processors are active during that interval and call it *idle* otherwise. Then T is the total length of the busy intervals plus the total length of the idle intervals. The total length of the busy intervals is at most $\sum_j t_j$. Now consider any path P through G and any idle interval. Since the schedule leaves no ready job idle, some job from P must be executing during the interval or all jobs on P must have finished before the interval. Thus the length of P is bounded by the total length of the idle intervals.

Since both the average work and the critical path length are lower bounds for the makespan, their sum must be a two-approximation of the makespan. □

A more careful analysis yields an approximation ratio of $2 - 1/p$. Improving upon this seems difficult. The only known better bounds increase the constant factor in the $1/p$ term [120]. We view this as a good reason to stick to the simple schedules characterized above. In particular, Theorem 14.3 applies even when the execution times are unknown and G unfolds with the computation. We only have to make sure that idle PEs find ready jobs efficiently. All the load-balancing algorithms described in this section can be adapted for this purpose.

Master–Worker. The master is informed about finished tasks. When a task becomes ready, it is inserted into the queue of tasks that can be handed out to idle PEs.

Randomized static. Each PE executes the ready jobs assigned to it.

Work stealing. Multithreaded computations [20] define a computation DAG implicitly by spawning tasks and waiting for them to finish. It can be shown that randomized work stealing leads to asymptotically optimal execution time. Compared with our result for tree-shaped computations, this is a more general result but also a constant factor worse with respect to the T/p term. Also, in practice, we might observe this constant factor because a multithreaded computation needs to generate the entire computation graph whereas tree-shaped computations only split work when this is actually needed.