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.

A

Mathematical Background

A.1 Mathematical Symbols

$\{e_0, \ldots, e_{n-1}\}$: set containing elements $e_0, \ldots, e_{n-1}$.

$\{e : P(e)\}$: set of all elements that fulfill the predicate P.

$\langle e_0, \ldots, e_{n-1} \rangle$: sequence consisting of elements $e_0, \ldots, e_{n-1}$.

$\langle e \in S : P(e) \rangle$: subsequence of all elements of sequence S that fulfill the predicate P.

$|x|$: the absolute value of x, for a real number x.

$\lfloor x \rfloor$: the largest integer $\leq x$, for a real number x.

$\lceil x \rceil$: the smallest integer $\geq x$, for a real number x.

$[a,b] := \{x \in \mathbb{R} : a \leq x \leq b\}$.

$i..j$: abbreviation for $\{i, i+1, \ldots, j\}$.

A^B: the set of all functions mapping B to A.

$A \times B$: the set of ordered pairs (a,b) with $a \in A$ and $b \in B$.

$\perp$: an undefined value.

$(+/-)\infty$: (plus/minus) infinity.

$\forall x : P(x)$: For *all* values of x, the proposition $P(x)$ is true.

$\exists x : P(x)$: There *exists* a value of x such that the proposition $P(x)$ is true.

$\mathbb{N}$: nonnegative integers; $\mathbb{N} = \{0, 1, 2, \ldots\}$.

$\mathbb{N}_+$: positive integers; $\mathbb{N}_+ = \{1, 2, \ldots\}$.

$\mathbb{Z}$: integers.

$\mathbb{R}$: real numbers.

$\mathbb{R}_{>0}$/$\mathbb{R}_{\geq 0}$: positive/nonnegative real numbers.

$\mathbb{Q}$: rational numbers.

$|$, $\&$, $\ll$, $\gg$, $\oplus$: bitwise OR, bitwise AND, shift left, shift right, and exclusive OR (XOR) respectively.

$\sum_{i=1}^{n} a_i = \sum_{1\leq i\leq n} a_i = \sum_{i\in 1..n} a_i := a_1 + a_2 + \cdots + a_n$.

$\prod_{i=1}^{n} a_i = \prod_{1\leq i\leq n} a_i = \prod_{i\in 1..n} a_i := a_1 \cdot a_2 \cdots a_n$.

$n! := \prod_{i=1}^{n} i$, the *factorial* of n.

$H_n := \sum_{i=1}^{n} 1/i$, the nth *harmonic number* (see (A.13)).

$a \cdot b := \sum_{i=1}^{n} a_i b_i$ is the dot product of vectors $a = (a_1, \ldots, a_n)$ and $b = (b_1, \ldots, b_n)$.

$\log x$: the logarithm to base two of x, $\log_2 x$, for $x > 0$.

$\log^* x$, for $x > 0$: the smallest integer k such that $\underbrace{\log(\log(\ldots \log(x) \ldots))}_{k\text{-fold application}} \leq 1$.

$\ln x$: the (natural) logarithm of x to base $e = 2.71828\ldots$.

$\mu(s,t)$: the shortest-path distance from s to t; $\mu(t) := \mu(s,t)$.

div: integer division; $m \operatorname{div} n := \lfloor m/n \rfloor$.

mod: modular arithmetic; $m \bmod n = m - n(m \operatorname{div} n)$.

$a \equiv b \bmod m$: a and b are congruent modulo m, i.e., $a + im = b$ for some integer i.

$\prec$: some ordering relation. In Sect. 9.3, it denotes the order in which nodes are marked during depth-first search.

`1`, `0`: the Boolean values "true" and "false".

Σ^*: The set $\{\langle a_1, \ldots, a_n\rangle : n \in \mathbb{N}, a_1, \ldots, a_n \in \Sigma\}$ of all *strings* or *words* over Σ. We usually write $a_1 \ldots a_n$ instead of $\langle a_1, \ldots, a_n\rangle$.

$|x|$: The number n of characters in a word $x = a_1 \ldots a_n$ over Σ; also called the length of the word.

A.2 Mathematical Concepts

antisymmetric: A relation R $\subseteq A \times A$ is *antisymmetric* if for all a and b in A, a R b and b R a implies $a = b$.

associative: An operation $\otimes$ with the property that $(x \otimes y) \otimes z = x \otimes (y \otimes z)$ for all x, y, and z.

asymptotic notation:

$$\begin{aligned}
\mathrm{O}(f(n)) &:= \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}. \\
\Omega(f(n)) &:= \{g(n) : \exists c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}. \\
\Theta(f(n)) &:= \mathrm{O}(f(n)) \cap \Omega(f(n)). \\
o(f(n)) &:= \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}. \\
\omega(f(n)) &:= \{g(n) : \forall c > 0 : \exists n_0 \in \mathbb{N}_+ : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}.
\end{aligned}$$

See also Sect. 2.1.

concave: A function f is concave on an interval $[a,b]$ if

$$\forall x, y \in [a,b] : \forall t \in [0,1] : f(tx + (1-t)y) \geq tf(x) + (1-t)f(y),$$

i.e., the function graph is never below the line segment connecting the points $(x, f(x))$ and $(y, f(y))$.

convex: A function f is convex on an interval $[a,b]$ if

$$\forall x, y \in [a,b] : \forall t \in [0,1] : f(tx + (1-t)y) \leq tf(x) + (1-t)f(y),$$

i.e., the function graph is never above the line segment connecting the points $(x, f(x))$ and $(y, f(y))$.

equivalence relation: a transitive, reflexive, and symmetric relation.

field: a set of elements (with distinguished elements 0 and 1) that support addition, subtraction, multiplication, and division by nonzero elements. Addition and multiplication are associative and commutative, and have neutral elements analogous to 0 and 1 for the real numbers. The most important examples are $\mathbb{R}$, the real numbers; $\mathbb{Q}$, the rational numbers; and $\mathbb{Z}_p$, the integers modulo a prime p.

iff: abbreviation for "if and only if".

lexicographic order: the canonical way of extending a total order on a set of elements to tuples, strings, or sequences over that set. We have $\langle a_1, a_2, \ldots, a_k \rangle < \langle b_1, b_2, \ldots, b_\ell \rangle$ if and only if there is an $i \leq \min\{k, \ell\}$ such that $\langle a_1, a_2, \ldots, a_{i-1} \rangle = \langle b_1, b_2, \ldots, b_{i-1} \rangle$ and $a_i < b_i$ or if $k < \ell$ and $\langle a_1, a_2, \ldots, a_k \rangle = \langle b_1, b_2, \ldots, b_k \rangle$. An equivalent recursive definition is as follows: $\langle\rangle < \langle b_1, b_2, \ldots, b_\ell \rangle$ for all $\ell > 0$; for $k > 0$ and $\ell > 0$, $\langle a_1, a_2, \ldots, a_k \rangle < \langle b_1, b_2, \ldots, b_\ell \rangle$ if and only if $a_1 < b_1$ or $a_1 = b_1$ and $\langle a_2, \ldots, a_k \rangle < \langle b_2, \ldots, b_\ell \rangle$.

linear order: (also total order) a reflexive, transitive, antisymmetric, and total relation. Linear orders are usually denoted by the symbol $\leq$. For $a \leq b$, one also writes $b \geq a$. The strict linear order $<$ is defined by $a < b$ if and only if $a \leq b$ and

$a \neq b$. The relation $<$ is transitive, irreflexive ($a < b$ implies $a \neq b$), and total in the sense that for all a and b either $a < b$ or $a = b$ or $a > b$. A typical example is the relation $<$ for real numbers.

linear preorder: (also linear quasi-order) a reflexive, transitive, and total relation. The symbols $\leq$ and $\geq$ are also used for linear preorders. Note that there can be distinct elements a and b with $a \leq b$ and $b \leq a$. The strict variant $<$ is defined as $a < b$ if $a \leq b$ and not $a \geq b$. An example is the relation $\mathrm{R} \subseteq \mathbb{R} \times \mathbb{R}$ defined by $x \mathrm{R} y$ if and only if $|x| \leq |y|$.

median: an element with rank $\lceil n/2 \rceil$ among n elements.

multiplicative inverse: If an object x is multiplied by a *multiplicative inverse* x^{-1} of x, we obtain $x \cdot x^{-1} = 1$ – the neutral element of multiplication. In particular, in a *field*, every element except 0 (the neutral element of addition) has a unique multiplicative inverse.

prime number: An integer n, $n \geq 2$, is a prime if and only if there are no integers $a, b > 1$ such that $n = a \cdot b$.

rank: Let $\leq$ be a linear preorder on a set $S = \{e_1, \ldots, e_n\}$. A one-to-one mapping $r : S \to 1..n$ is a *ranking function* for the elements of S if $r(e_i) < r(e_j)$ whenever $e_i < e_j$. If $\leq$ is a linear order, there is exactly one ranking function.

reflexive: A relation $\mathrm{R} \subseteq A \times A$ is reflexive if $a \mathrm{R} a$ for all $a \in A$.

relation: a set of ordered pairs R over some set A. Often we write relations as infix operators; for example, if $\mathrm{R} \subseteq A \times A$ is a relation, $a \mathrm{R} b$ means $(a, b) \in \mathrm{R}$.

symmetric relation: A relation $R \subseteq A \times A$ is *symmetric* if for all a and b in A, $a \mathrm{R} b$ implies $b \mathrm{R} a$.

total order: a synonym for linear order.

total relation: A relation $\mathrm{R} \subseteq A \times A$ is *total* if for all a and b in A, either $a \mathrm{R} b$ or $b \mathrm{R} a$ or both. If a relation R is total and transitive, then the relation $\sim_{\mathrm{R}}$ defined by $a \sim_{\mathrm{R}} b$ if and only if $a \mathrm{R} b$ and $b \mathrm{R} a$ is an equivalence relation.

transitive: A relation $\mathrm{R} \subseteq A \times A$ is *transitive* if for all a, b, and c in A, $a \mathrm{R} b$ and $b \mathrm{R} c$ imply $a \mathrm{R} c$.

A.3 Basic Probability Theory

Probability theory rests on the concept of a *sample space* $\mathscr{S}$. For example, to describe the rolls of two dice, we would use the 36-element sample space $\{1, \ldots, 6\} \times \{1, \ldots, 6\}$, i.e., the elements of the sample space (also called elementary events or

simply events) are the pairs (x,y) with $1 \le x,y \le 6$ and $x,y \in \mathbb{N}$. Generally, a sample space is any nonempty set. In this book, all sample spaces are finite.[1] In a *random experiment*, any element of $s \in \mathscr{S}$ is chosen with some elementary *probability* p_s, where $\sum_{s\in\mathscr{S}} p_s = 1$. The function that assigns to each event s its probability p_s is called a *distribution*. A sample space together with a probability distribution is called a *probability space*. In this book, we use *uniform distributions* almost exclusively; in this case $p_s = p = 1/|\mathscr{S}|$. Subsets $\mathscr{E}$ of the sample space are called *events*. The probability of an *event* $\mathscr{E} \subseteq \mathscr{S}$ is the sum of the probabilities of its elements, i.e., $\text{prob}(\mathscr{E}) = |\mathscr{E}|/|\mathscr{S}|$ in the uniform case. So the probability of the event $\{(x,y) : x+y = 7\} = \{(1,6),(2,5),\ldots,(6,1)\}$ is equal to $6/36 = 1/6$, and the probability of the event $\{(x,y) : x+y \ge 8\}$ is equal to $15/36 = 5/12$.

A *random variable* is a mapping from the sample space to the real numbers. Random variables are usually denoted by capital letters to distinguish them from plain values. For our example of rolling two dice, the random variable X could give the number shown by the first die, the random variable Y could give the number shown by the second die, and the random variable S could give the sum of the two numbers. Formally, if $(x,y) \in \mathscr{S}$, then $X((x,y)) = x$, $Y((x,y)) = y$, and $S((x,y)) = x+y = X((x,y)) + Y((x,y))$.

We can define new random variables as expressions involving other random variables and ordinary values. For example, if V and W are random variables, then $(V+W)(s) = V(s) + W(s)$, $(V \cdot W)(s) = V(s) \cdot W(s)$, and $(V+3)(s) = V(s) + 3$.

Events are often specified by predicates involving random variables. For example, $X \le 2$ denotes the event $\{(1,y),(2,y) : 1 \le y \le 6\}$, and hence $\text{prob}(X \le 2) = 1/3$. Similarly, $\text{prob}(X+Y = 11) = \text{prob}(\{(5,6),(6,5)\}) = 1/18$.

Indicator random variables are random variables that take only the values 0 and 1. Indicator variables are an extremely useful tool for the probabilistic analysis of algorithms because they allow us to encode the behavior of complex algorithms into simple mathematical objects. We frequently use the letters I and J for indicator variables. Indicator variables and events are in a one-to-one correspondance. If $\mathscr{E}$ is an event, then $I_{\mathscr{E}}$ with $I_{\mathscr{E}}(s) = 1$ if and only if $s \in \mathscr{E}$ is the corresponding indicator variable. If an event is specified by a predicate P, one sometimes writes $[P]$ for the corresponding indicator variable, i.e., $[P](s) = 1$ if $P(s)$ and $[P](s) = 0$ otherwise.

The *expected value* of a random variable $Z : \mathscr{S} \to \mathbb{R}$ is

$$\text{E}[Z] = \sum_{s\in\mathscr{S}} p_s \cdot Z(s) = \sum_{z\in\mathbb{R}} z \cdot \text{prob}(Z = z), \tag{A.1}$$

i.e., every sample s contributes the value of Z at s times its probability. Alternatively, we can group all s with $Z(s) = z$ into the event $Z = z$ and then sum over the $z \in \mathbb{R}$.

In our example, $\text{E}[X] = (1+2+3+4+5+6)/6 = 21/6 = 3.5$, i.e., the expected value of the first die is 3.5. Of course, the expected value of the second die is also 3.5. For an indicator random variable I we have

[1] All statements made in this section also hold for countable infinite sets, essentially with the same proofs. Such sample spaces are, for example, needed to model the experiment "throw a die repeatedly until the value six occurs".

$$\mathrm{E}[I] = 0 \cdot \mathrm{prob}(I = 0) + 1 \cdot \mathrm{prob}(I = 1) = \mathrm{prob}(I = 1).$$

Sometimes we are more interested in a random variable Z and its behavior than in the underlying probability space. In such a situation, it suffices to know the range $Z[\mathscr{S}]$ of Z and the induced probabilities $\mathrm{prob}(Z = z)$, $z \in Z[\mathscr{S}]$. We refer to the function $z \mapsto \mathrm{prob}(Z = z)$ defined on $Z[\mathscr{S}]$ as the *distribution of* Z. Two random variables X and Y with the same distribution are called *identically distributed*.

For a random variable Z that takes only values in the natural numbers, there is a very useful formula for its expected value:

$$\mathrm{E}[Z] = \sum_{k \geq 1} \mathrm{prob}(Z \geq k), \quad \text{if } Z[\mathscr{S}] \subseteq \mathbb{N}. \tag{A.2}$$

This formula is easy to prove. For $k, i \in \mathbb{N}$, let $p_k = \mathrm{prob}(Z \geq k)$ and $q_i = \mathrm{prob}(Z = i)$. Then $p_k = \sum_{i \geq k} q_i$ and hence

$$\mathrm{E}[Z] = \sum_{z \in Z[\mathscr{S}]} z \cdot \mathrm{prob}(Z = z) = \sum_{i \in \mathbb{N}} i \cdot \mathrm{prob}(Z = i) = \sum_{i \in \mathbb{N}} \sum_{1 \leq k \leq i} q_i = \sum_{k \geq 1} \sum_{i \geq k} q_i = \sum_{k \geq 1} p_k.$$

Here, the next to last equality is a change of the order of summation.

Often we are interested in the expectation of a random variable that is defined in terms of other random variables. This is particulary easy for sums of random variables due to the *linearity of expectations* of random variables: For any two random variables V and W,

$$\mathrm{E}[V + W] = \mathrm{E}[V] + \mathrm{E}[W]. \tag{A.3}$$

This equation is easy to prove and extremely useful. Let us prove it. It amounts essentially to an application of the distributive law of arithmetic. We have

$$\begin{aligned} \mathrm{E}[V + W] &= \sum_{s \in \mathscr{S}} p_s \cdot (V(s) + W(s)) \\ &= \sum_{s \in \mathscr{S}} p_s \cdot V(s) + \sum_{s \in \mathscr{S}} p_s \cdot W(s) \\ &= \mathrm{E}[V] + \mathrm{E}[W]. \end{aligned}$$

As our first application, let us compute the expected sum of two dice. We have

$$\mathrm{E}[S] = \mathrm{E}[X + Y] = \mathrm{E}[X] + \mathrm{E}[Y] = 3.5 + 3.5 = 7.$$

Observe that we obtain the result with almost no computation. Without knowing about the linearity of expectations, we would have to go through a tedious calculation:

$$\begin{aligned} \mathrm{E}[S] &= 2 \cdot \tfrac{1}{36} + 3 \cdot \tfrac{2}{36} + 4 \cdot \tfrac{3}{36} + 5 \cdot \tfrac{4}{36} + 6 \cdot \tfrac{5}{36} + 7 \cdot \tfrac{6}{36} + 8 \cdot \tfrac{5}{36} + 9 \cdot \tfrac{4}{36} + \ldots + 12 \cdot \tfrac{1}{36} \\ &= \frac{2 \cdot 1 + 3 \cdot 2 + 4 \cdot 3 + 5 \cdot 4 + 6 \cdot 5 + 7 \cdot 6 + 8 \cdot 5 + \ldots + 12 \cdot 1}{36} = 7. \end{aligned}$$

Exercise A.1. What is the expected sum of three dice?

We shall now give another example with a more complex sample space. We consider the experiment of throwing n balls into m bins. The balls are thrown at random and distinct balls do not influence each other. Formally, our sample space is the set of all functions f from $1..n$ to $1..m$. This sample space has size m^n, and $f(i)$, $1 \le i \le n$, indicates the bin into which the ball i is thrown. All elements of the sample space are equally likely. How many balls should we expect in bin 1? We use W to denote the number of balls in bin 1. To determine $\mathrm{E}[W]$, we introduce indicator variables I_i, $1 \le i \le n$. The variable I_i is 1 if ball i is thrown into bin 1 and is 0 otherwise. Formally, $I_i(f) = 0$ if and only if $f(i) \neq 1$. Then $W = \sum_i I_i$. We have

$$\mathrm{E}[W] = \mathrm{E}\left[\sum_i I_i\right] = \sum_i \mathrm{E}[I_i] = \sum_i \mathrm{prob}(I_i = 1),$$

where the second equality is the linearity of expectations and the third equality follows from the I_i's being indicator variables. It remains to determine the probability that $I_i = 1$. Since the balls are thrown at random, ball i ends up in any bin[2] with the same probability. Thus $\mathrm{prob}(I_i = 1) = 1/m$, and hence

$$\mathrm{E}[I] = \sum_i \mathrm{prob}(I_i = 1) = \sum_i \frac{1}{m} = \frac{n}{m}.$$

Products of random variables behave differently. In general, we have $\mathrm{E}[X \cdot Y] \neq \mathrm{E}[X] \cdot \mathrm{E}[Y]$. There is one important exception: If X and Y are *independent*, equality holds. Random variables $X_1, \ldots, X_k$ are independent if and only if

$$\forall x_1, \ldots, x_k : \mathrm{prob}(X_1 = x_1 \wedge \cdots \wedge X_k = x_k) = \prod_{1 \le i \le k} \mathrm{prob}(X_i = x_i). \tag{A.4}$$

As an example, when we roll two dice, the value of the first die and the value of the second die are independent random variables. However, the value of the first die and the sum of the two dice are not independent random variables.

Exercise A.2. Let I and J be independent indicator variables and let $X = (I + J) \bmod 2$, i.e., X is 1 if and only if I and J are different. Show that I and X are independent, but that I, J, and X are dependent.

We will next show

$$\mathrm{E}[X \cdot Y] = \mathrm{E}[X] \cdot \mathrm{E}[Y] \quad \text{if } X \text{ and } Y \text{ are independent.}$$

We have

[2] Formally, there are exactly m^{n-1} functions f with $f(i) = 1$.

$$\begin{aligned}
\mathrm{E}[X] \cdot \mathrm{E}[Y] &= \left(\sum_x x \cdot \mathrm{prob}(X = x)\right) \cdot \left(\sum_y y \cdot \mathrm{prob}(X = y)\right) \\
&= \sum_{x,y} x \cdot y \cdot \mathrm{prob}(X = x) \cdot \mathrm{prob}(X = y) \\
&= \sum_{x,y} x \cdot y \cdot \mathrm{prob}(X = x \wedge Y = y) \\
&= \sum_z \sum_{x,y \text{ with } z = x \cdot y} z \cdot \mathrm{prob}(X = x \wedge Y = y) \\
&= \sum_z z \cdot \sum_{x,y \text{ with } z = x \cdot y} \mathrm{prob}(X = x \wedge Y = y) \\
&= \sum_z z \cdot \mathrm{prob}(X \cdot Y = z) \\
&= \mathrm{E}[X \cdot Y].
\end{aligned}$$

How likely is it that a random variable will deviate substantially from its expected value? *Markov's inequality* gives a useful bound. Let X be a nonnegative random variable and let c be any constant. Then

$$\mathrm{prob}(X \geq c \cdot \mathrm{E}[X]) \leq \frac{1}{c}. \tag{A.5}$$

The proof is simple. We have

$$\begin{aligned}
\mathrm{E}[X] &= \sum_{z \in \mathbb{R}} z \cdot \mathrm{prob}(X = z) \\
&\geq \sum_{z \geq c \cdot \mathrm{E}[X]} z \cdot \mathrm{prob}(X = z) \\
&\geq c \cdot \mathrm{E}[X] \cdot \mathrm{prob}(X \geq c \cdot \mathrm{E}[X]),
\end{aligned}$$

where the first inequality follows from the fact that we sum over a subset of the possible values and X is nonnegative, and the second inequality follows from the fact that the sum in the second line ranges only over z such that $z \geq c\mathrm{E}[X]$.

Much tighter bounds are possible for some special cases of random variables. The following situation arises several times in the book. We have a sum $X = X_1 + \cdots + X_n$ of n independent indicator random variables $X_1, \ldots, X_n$ and want to bound the probability that X deviates substantially from its expected value. In this situation, the following variant of the *Chernoff bound* is useful. For any $\varepsilon > 0$, we have

$$\mathrm{prob}(X < (1 - \varepsilon)\mathrm{E}[X]) \leq e^{-\varepsilon^2 \mathrm{E}[X]/2}, \tag{A.6}$$

$$\mathrm{prob}(X > (1 + \varepsilon)\mathrm{E}[X]) \leq \left(\frac{e^{\varepsilon}}{(1+\varepsilon)^{(1+\varepsilon)}}\right)^{\mathrm{E}[X]}. \tag{A.7}$$

A bound of the form above is called a *tail bound* because it estimates the "tail" of the probability distribution, i.e., the part for which X deviates considerably from its expected value.

Let us see an example. If we throw n coins and let X_i be the indicator variable for the ith coin coming up heads, $X = X_1 + \cdots + X_n$ is the total number of heads. Clearly, $\mathrm{E}[X] = n/2$. The bound above tells us that $\mathrm{prob}(X \leq (1-\varepsilon)n/2) \leq e^{-\varepsilon^2 n/4}$. In particular, for $\varepsilon = 0.1$, we have $\mathrm{prob}(X \leq 0.9 \cdot n/2) \leq e^{-0.01 \cdot n/4}$. So, for $n = 10\,000$, the expected number of heads is 5000 and the probability that the sum is less than 4500 is smaller than e^{-25}, a very small number.

Exercise A.3. Estimate the probability that X in the above example is larger than 5050.

If the indicator random variables are independent and identically distributed with $\mathrm{prob}(X_i = 1) = p$, X is *binomially distributed*, i.e.,

$$\mathrm{prob}(X = k) = \binom{n}{k} p^k (1-p)^{(n-k)}. \tag{A.8}$$

Exercise A.4 (balls and bins continued). As above, let W denote the number of balls in bin 1. Show that

$$\mathrm{prob}(W = k) = \binom{n}{k} \left(\frac{1}{m}\right)^k \left(1 - \frac{1}{m}\right)^{(n-k)},$$

and then attempt to compute $\mathrm{E}[W]$ as $\sum_k \mathrm{prob}(W = k)k$.

A.4 Useful Formulae

We shall first list some useful formulae and then prove some of them:

- *A simple approximation to the factorial*:

$$\left(\frac{n}{e}\right)^n \leq n! \leq n^n \quad \text{or, more precisely} \quad e\left(\frac{n}{e}\right)^n \leq n! \leq (en)\left(\frac{n}{e}\right)^n. \tag{A.9}$$

- *Stirling's approximation to the factorial*:

$$n! = \left(1 + \mathrm{O}\left(\frac{1}{n}\right)\right) \sqrt{2\pi n} \left(\frac{n}{e}\right)^n. \tag{A.10}$$

- *An approximation to the binomial coefficients*:

$$\binom{n}{k} \leq \left(\frac{n \cdot e}{k}\right)^k. \tag{A.11}$$

- *The sum of the first n integers*:

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}. \tag{A.12}$$

- *The harmonic numbers*:

$$\ln n \le H_n = \sum_{i=1}^{n} \frac{1}{i} \le \ln n + 1. \tag{A.13}$$

- *The geometric series*:

$$\sum_{i=0}^{n-1} q^i = \frac{1-q^n}{1-q} \quad \text{for } q \neq 1 \text{ and} \quad \sum_{i\ge 0} q^i = \frac{1}{1-q} \quad \text{for } |q| < 1. \tag{A.14}$$

$$\sum_{i\ge 0} 2^{-i} = 2 \quad \text{and} \quad \sum_{i\ge 0} i \cdot 2^{-i} = \sum_{i\ge 1} i \cdot 2^{-i} = 2. \tag{A.15}$$

- *Jensen's inequality*:

$$\sum_{i=1}^{n} f(x_i) \le n \cdot f\left(\frac{\sum_{i=1}^{n} x_i}{n}\right) \tag{A.16}$$

for any concave function f. Similarly, for any convex function f,

$$\sum_{i=1}^{n} f(x_i) \ \ge n \cdot f\left(\frac{\sum_{i=1}^{n} x_i}{n}\right). \tag{A.17}$$

- *Approximations to the logarithm following from the Taylor series expansion*:

$$x - \frac{1}{2}x^2 \le \ln(1+x) \le x - \frac{1}{2}x^2 + \frac{1}{3}x^3 \le x. \tag{A.18}$$

A.4.1 Proofs

For (A.9), we first observe that $n! = n(n-1)\cdots 1 \le n^n$. For the lower bound, we recall from calculus that $e^x = \sum_{i\ge 0} x^i/i!$ for all x. In particular, $e^n \ge n^n/n!$ and hence $n! \ge (n/e)^n$.

We now come to the sharper bounds. Also, for all $i \ge 2$, $\ln i \ge \int_{i-1}^{i} \ln x\,dx$, and therefore

$$\ln n! = \sum_{2\le i\le n} \ln i \ge \int_1^n \ln x\,dx = \Big[x(\ln x - 1)\Big]_{x=1}^{x=n} n(\ln n - 1) + 1.$$

Thus

$$n! \ge e^{n(\ln n - 1)+1} = e(e^{\ln n - 1})^n = e(n/e)^n.$$

For the upper bound, we use $\ln(i-1) \le \int_{i-1}^{i} \ln x\,dx$ and hence $(n-1)! \le \int_1^n \ln x\,dx = e(n/e)^n$. Thus $n! \le (en)(n/e)^n$.

Equation (A.11) follows almost immediately from (A.9). We have

$$\binom{n}{k} = \frac{n(n-1)\cdots(n-k+1)}{k!} \le \frac{n^k}{(k/e)^k} = \left(\frac{n\cdot e}{k}\right)^k.$$

Equation (A.12) can be computed by a simple trick:

$$\begin{aligned}1+2+\ldots+n &= \frac{1}{2}\left((1+2+\ldots+n-1+n)+(n+n-1+\ldots+2+1)\right)\\ &= \frac{1}{2}\left((n+1)+(2+n-1)+\ldots+(n-1+2)+(n+1)\right)\\ &= \frac{n(n+1)}{2}.\end{aligned}$$

The sums of higher powers are estimated easily; exact summation formulae are also available. For example, $\int_{i-1}^{i} x^2\,dx \le i^2 \le \int_{i}^{i+1} x^2\,dx$, and hence

$$\sum_{1\le i\le n} i^2 \le \int_1^{n+1} x^2\,dx = \left[\frac{x^3}{3}\right]_{x=1}^{x=n+1} = \frac{(n+1)^3-1}{3}$$

and

$$\sum_{1\le i\le n} i^2 \ge \int_0^{n} x^2\,dx = \left[\frac{x^3}{3}\right]_{x=0}^{x=n} = \frac{n^3}{3}.$$

For (A.13), we also use estimation by integral. We have $\int_i^{i+1}(1/x)\,dx \le 1/i \le \int_{i-1}^{i}(1/x)\,dx$, and hence

$$\ln n = \int_1^n \frac{1}{x}\,dx \le \sum_{1\le i\le n}\frac{1}{i} \le 1+\int_1^n \frac{1}{x}\,dx = 1+\ln n.$$

Equation (A.14) follows from

$$(1-q)\cdot\sum_{0\le i\le n-1} q^i = \sum_{0\le i\le n-1} q^i - \sum_{1\le i\le n} q^i = 1-q^n.$$

If $|q|<1$, we may let n pass to infinity. This yields $\sum_{i\ge 0} q^i = 1/(1-q)$. For $q=1/2$, we obtain $\sum_{i\ge 0} 2^{-i} = 2$. Also,

$$\begin{aligned}\sum_{i\ge 1} i\cdot 2^{-i} &= \sum_{i\ge 1} 2^{-i} + \sum_{i\ge 2} 2^{-i} + \sum_{i\ge 3} 2^{-i} + \ldots\\ &= (1+1/2+1/4+1/8+\ldots)\cdot\sum_{i\ge 1} 2^{-i}\\ &= 2\cdot 1 = 2.\end{aligned}$$

For the first equality, observe that the term 2^{-i} occurs in exactly the first i sums on the right-hand side.

Equation (A.16) can be shown by induction on n. For $n=1$, there is nothing to show. So assume $n\ge 2$. Let $x^* = \sum_{1\le i\le n} x_i/n$ and $\bar{x} = \sum_{1\le i\le n-1} x_i/(n-1)$. Then $x^* = ((n-1)\bar{x}+x_n)/n$, and hence

$$\begin{aligned}\sum_{1\le i\le n} f(x_i) &= f(x_n) + \sum_{1\le i\le n-1} f(x_i)\\ &\le f(x_n) + (n-1)\cdot f(\bar{x}) = n\cdot\left(\frac{1}{n}\cdot f(x_n) + \frac{n-1}{n}\cdot f(\bar{x})\right)\\ &\le n\cdot f(x^*),\end{aligned}$$

where the first inequality uses the induction hypothesis and the second inequality uses the definition of concavity with $x = x_n$, $y = \bar{x}$, and $t = 1/n$. The extension to convex functions is immediate, since convexity of f implies concavity of $-f$.

B

Computer Architecture Aspects

In Sect. 2.2, we introduced several basic models of parallel computing, and then in Sects. 2.4.3 and 2.15 hinted at aspects that make practical parallel computing more complex (see Fig. 2.6 for an overview). However, in order to understand some aspects of our practical examples, one needs a little more background. The purpose of this appendix is to fill this gap in a minimalistic way – learning about these aspects from sources dedicated to them might still be a good idea.

We discuss a concrete example – the machine used to run our examples of shared-memory programs. This example will nevertheless lead to general insights, as the most relevant aspects of its architecture have been stable for more than a decade and essentially also apply to processors from other vendors. For more details of computer architecture in general, see the textbook by Hennessy and Patterson [148]. Details of the x86 architecture can be found in the Architecture Reference Manual [160].

B.1 Cores and Hardware Threads

We programmed our shared-memory examples on a machine with four Intel Xeon E7-8890 v3 processors (previously codenamed Haswell-EX). Each of these processors has 18 *cores*. A processor core has one or several arithmetical and logical units for executing instructions and a *pipeline* for controlling the execution of an instruction stream. The instructions come from up to two *hardware threads*. Each hardware thread has a dedicated set of registers for storing operands. Most other resources of a core are shared by the two threads.[1] Since x86 is a *CISC* (complex instruction set computer) architecture, it has a large range of instructions. Instructions can perform a mix of memory access and arithmetic operations and their numbers of operands and

[1] The hope is that the two hardware threads of a core will substantially increase the performance of the core. Because each hardware thread has its own set of registers, switching between them incurs almost no cost. *Software threads* are managed by the operating system. Activating a software thread requires it to be made into a hardware thread; in particular, the registers must be loaded. The switch therefore incurs substantial cost.

encoding lengths vary. Since CISC instructions are difficult to process in a pipeline, they are first translated to *microinstructions* that are similar to the simple machine instructions used in the machine model in Sect. 2.2 (RISC – reduced instruction set computer).

The pipeline processes micro-instructions in up to 19 *stages*, where each stage is responsible for some small substep of the execution (such as decoding, fetching operands, ...). Thus, each stage becomes very fast, allowing high clock frequencies. Additional *instruction parallelism* is introduced because our machine is *superscalar*, i.e., up to eight instructions can be in the same pipeline stage at the same time. The hardware (assisted by the compiler) automatically extracts instruction parallelism from a sequential program in order to keep this pipeline reasonably well filled. In particular, the instructions might be executed in a different order than that specified by the machine program as long as this does not change the semantics of the (sequential) program. Thus, a core can execute several instructions in every clock cycle.

There are many reasons why the number eight is rarely reached; including data dependencies between instructions, large memory access latencies (cache misses), and conditional branches. Thus, a typical sequential program will most of the time use only a small fraction of the available arithmetical units or slots in the execution pipeline. This waste can be reduced using the two hardware threads sharing these resources.[2] Thus, in some codes with many high-latency operations, the two hardware threads do twice the work that a single thread could do. Usually however, the performance gain is smaller and can even be negative, for example with highly optimized numerical libraries that are carefully tuned to avoid cache misses and to use all the available arithmetical units. Other examples are codes that suffer from software overhead of additional threads (additional work for parallelization, scheduling, and locking). *Thus the PEs we are talking about in this book are either hardware threads or dedicated cores – whatever gives better overall performance.*

B.2 The Memory Hierarchy

Each core of our machine has a 32 KB level-one (L1) data cache; this cache is shared by the hardware threads supported by the core. There is a separate L1 cache of the same size that stores machine instructions. The content of the cache is managed in *blocks* (aka *cache lines*) of 64 bytes whose address starts at a position divisible by the block size – these addresses are said to be *aligned* to the cache line size.

Each core also has a larger (L2) cache of size 256 KB for data and instructions (a *unified cache*). The reason for this division between L1 and L2 is that the L1 cache is made as fast as possible almost without regard for cost per bit of memory, whereas the L2 cache uses a more compact design at the price of higher access latencies and lower throughput. Indeed, even using the same technology, a large cache will inevitably have larger access latency than a small one, ultimately because the speed of light is limited.

[2] Some architectures support four or eight hardware threads per core.

Using an even more compact technology, all cores on a processor chip share a large L3 cache of size 45 MB. Many concurrent memory operations on threads on the same chip can be performed within the L3 cache without having to go through the main memory. This is one reason why many shared-memory programs scale well as long as they run on a single chip but scale worse (or not at all) when running on multiple chips.

Our machine has 128 GB of main memory attached to each processor chip. For uniformity, let us call this level four of the memory hierarchy. The processor chips are connected by a high-speed interconnect interface. On our machine, every chip has a dedicated link to every other chip.[3] Any thread can transparently access the main memory of every chip. However, nonlocal accesses will incur higher latency and yield lower overall bandwidth. This effect is called *Non Uniform Memory Access* (NUMA) and processor chips (or *sockets*) are therefore also called *NUMA nodes*[4].

B.3 Cache Coherence Protocols

We first review how a typical sequential cache replacement strategy works. When a core accesses the data at address i, it looks for the cache line b containing that data in the data caches. Suppose the highest[5] level of the hierarchy containing b is j. Then b is copied to the caches at levels $j-1$ down to 1 and then accessed from level 1. The reason for this strategy is that the main cost is for reading the data from level j and that having copies in the higher levels makes future accesses cheaper. A consequence of moving a cache line into a cache is that another block may have to be evicted to make room. If this block has been modified since the last access, it is written to the next lower layer of the memory hierarchy.

It can happen that the data being accessed straddles two cache lines despite being smaller than the block size. Such *nonaligned* operations cause additional costs by requiring both blocks to be moved. Moreover, certain operations cause additional overheads for nonaligned accesses or do not work at all (e.g., 16-byte CAS). Hence, an important performance optimization involves avoiding unaligned accesses by being careful with the data layout.

Assuming a *write-back cache*, write operations go to the L1 cache. When the cache line being accessed is already present in the L1 cache, this is easy. Otherwise, some other cache line may need to be evicted in order to make room. Moreover, the accessed block first has to be read from the lower levels of the memory hierarchy.

[3] On larger or cheaper machines, a more sparse network might be used, e.g., a ring of four chips or a mesh network.

[4] Identifiying sockets with NUMA nodes is an oversimplification because chips or multichip modules on a single processor socket might also define multiple NUMA nodes.

[5] The L1 cache is the highest level and the main memory is the lowest level. Instead of "highest level" one may also say "closest level", i.e., the level closest to the processing unit.

This is necessary in order to avoid additional bookkeeping about which parts of what cache block contain valid data.[6]

In a shared-memory parallel computer, things get more complicated when several PEs access the same memory block. Suppose block *b* is written by PE *i*. Some other PE *j* may have a copy of *b* in a local cache. This copy is no longer valid, and thus has to be *invalidated*. The inter-PE communication needed for invalidation and rereading invalidated copies causes significant overhead and is one of the main reasons for the limited scalability of shared-memory programs. Note that this overhead is incurred even when PEs *i* and *j* access *different* memory locations in the same cache line. This effect is called *false sharing* and has to be avoided by careful memory layout. For example, we should usually allocate a full cache line for a lock variable or use the remainder of that cache line for data that is accessed only by a thread that owns that lock. The technique of making a data structure larger than necessary is called *padding*. More generally, whenever we have the situation that multiple threads are trying to write to the same cache line *b* at the same time, performance can go down significantly. We call this situation *contention* on *b*.

We can see that accessing memory on a real-world machine is far away from the idealized view of an instantaneous, globally visible effect. The *cache coherence* mechanism of the hardware can only provide an approximation of the idealized view to the application programs. The possible deviations from the idealized view are defined in the *memory consistency model* or *memory model* of the machine. Coping with these deviations is a major challenge for writers of parallel shared-memory programs. Unfortunately, memory models vary between different architectures. Here we describe the memory model of the x86 architecture and hint at differences in other architectures.

The compiler or the hardware can reorder the memory access operations of a thread in order to keep the pipelines filled. Additionally, in order to improve memory performance, some write operations are delayed by buffering the data in additional *memory buffers*. Within a sequential program, this is done in such a way that the outcome of the computation is not changed. However, in a concurrent program this can lead to problems. Thus, the first thing is to instruct the compiler to abstain from undesired reorderings. In C++ the storage class `volatile` ensures that a variable is always loaded from memory before using it. It is also ensured that the compiler does not reorder accesses to volatile variables. In order to also exclude improper reordering of other memory accesses the statement

```
atomic_signal_fence(memory_order_seq_cst);
```

defines a *memory fence* for the compiler – all reads and writes before the fence have to be compiled to happen before all reads and writes after the fence. Once the compiler is tamed, the x86 architecture guarantees that the read operations of a thread appear to all other threads in the right order. The same is true for the write operations. However, the x86 hardware may reorder reads with older writes to different

[6] In order to avoid unnecessary overheads when the user knows that the entire cache line will be written in the near future, one can use *write combining* aided by *nontemporal* write instructions. An example in the case of sorting can be found in [278].

locations. The C++ command `atomic_thread_fence()` can be used to also guarantee the ordering between read and write operations.

Note that it may still happen that the operations of different threads can be mixed arbitrarily and may also appear to different threads in different orders.

B.4 Atomic Operations

We have already explained the compare-and-swap (CAS) operation in Sect. 2.4.1. On the x86 architecture it works exactly as described there and is available for accesses to 32-, 64-, or aligned 128-bit data. Some other architectures have CAS operations up to only 64 bits or with slightly weaker guarantees – they may sometimes fail even though the actual value is equal to the desired value. This sometimes requires additional check loops.

The x86 architecture and others also offer atomic fetch-and-add/subtract/min/-max/and/ or/xor instructions. Although it is easy to implement them using CAS (see also Sect. 2.4.1), using the built-in operation may be faster, in particular when several threads contend for operating on the same variable.

B.5 Hardware Transactional Memory

Beginning with the Haswell microarchitecture, Intel x86 processors have supported an implementation of restricted hardware transactional memory called *Transactional Synchronization Extensions*.[7] The programmer can enclose a critical section using the machine instructions `XBEGIN` and `XEND`. There is an important difference compared to the simple mechanism described in Sect. 2.4.1. A transaction t may fail, i.e., the hardware notices that another thread has accessed a cache line touched by t in such a way that t does not appear to be executed atomically.[8] In that case, the transaction is *rolled back*, i.e., all its effects are undone and an *abort handler* is called. It is possible to retry a transaction. However, it may happen that transactions keep failing without the system making progress. Therefore, the programmer has to provide some ultimate fallback mechanism that is guaranteed to work. For example, using locks or atomic operations. The listing at the end of Sect. 4.6.3 gives an example. Hardware transactions are attractive from the point of view of performance when an abort is unlikely, i.e., when there is little contention on the pieces of memory accessed by the transaction.

B.6 Memory Management

In reality, memory is not the simple one-dimensional array we have seen for the RAM model in Sect. 2.2. First of all, a process sees logical addresses in *virtual*

[7] IBM had already introduced a similar mechanism with the POWER8 architecture.

[8] There are further reasons why a transaction may fail, e.g., that too many cache lines have been touched.

memory instead of physical addresses. Logical addresses are mapped to physical memory by the hardware in cooperation with the operating system. Pages of virtual memory (address ranges beginning at a multiple of the page size which is a power of 2) are mapped to physical memory by changing the most significant digits of the addresses using a combination of hardware buffers (translation lookaside buffers – TLBs) and a translation data structure managed by the operating system.

Besides the overhead in virtual address translation, it matters which socket a piece of virtual memory is mapped to. This has to be taken into account when allocating memory. In LINUX, a useful mechanism is that a thread allocating memory will by default allocate it on its local socket. Other placement policies are possible. For example, sometimes it makes sense to map blocks round-robin, i.e., the ith block of an array is mapped to socket i mod P, where P is the number of socket.

B.7 The Interconnection Network

The processors of a parallel machine are connected through an interconnection network. When the number of processors is small, a complete network with all point-to-point links is feasible. When the number of processors is larger, a sparse network must be used. Figure B.1 shows some typical interconnection networks. Two-dimensional and three-dimensional meshes have the advantage that they can be built arbitrarily large with bounded wire length. This is also possible for *torus* networks, which consist of ring interconnections in every direction; Fig. B.1 shows a physical layout with only short connections. The $\log p$-dimensional mesh is better known under the name "*hypercube*" (see also Fig. 13.2). A good approximation to a complete interconnection network can be achieved using hierarchical networks such as the *fat tree* [199]. In this book, we have consciously avoided topology-dependent algorithms in order to keep things simple and portable. However, we sometimes point out opportunities for adapting algorithms to the network topology. We gave a nontrivial example at the end of Sect. 13.1.4.

In sparse interconnection networks, arbitrary point-to-point communication has to be realized by sending messages along paths through the network. Some of these

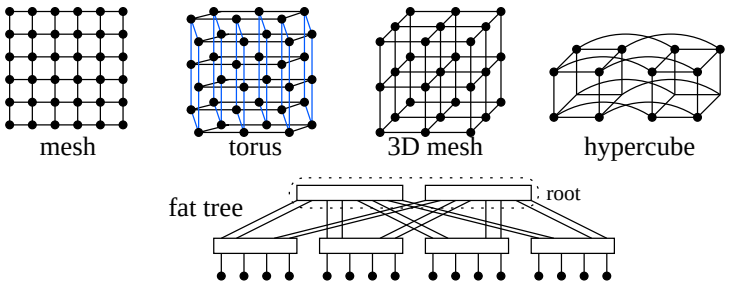

Fig. B.1. Common interconnection networks. The fat tree is an example of a *multistage network*, connecting 16 PEs using two layers of 8-way switches.

paths will have to share the same wire and thus cause *contention*. Carefully designed routing algorithms can keep this contention small. For example, in a hypercube with p nodes, the maximum length of a path between two nodes is $\Theta(\log p)$ and hence, the minimum delivery time is $\Theta(\log p)$. There are routing algorithms for the hypercube that guarantee that p messages with distinct sources and distinct destinations can be delivered in time $O(\log p)$, thus achieving the minimum. We refer our readers to the textbook [197] for a discussion of routing algorithms.

Processors contain dedicated subprocessors for efficient I/O and communication. Using *direct memory access (DMA)*, they move data between memory and interface hardware controlling disk and network access. Therefore, it is possible to overlap communication, I/O, and computation.

B.8 CPU Performance Analysis

Code optimization requires profiling for two reasons. First, algorithm analysis makes asymptotic statements and ignores constant factors. Second, our models of sequential and, even more so, of parallel computers are only approximations of real hardware. Profiling provides detailed information about the execution of a program. Modern processors have dedicated hardware performance-monitoring units (PMUs) to support profiling. PMUs count performance events (CPU cycles, cache misses, branch mis-predictions, etc.) and also map them to particular instructions.

PMUs support hundreds of very detailed performance events[9], which can be programmed using profiling software (CPU profilers). The open-source Linux perf profiler[10] has good support for common CPU architectures. The most advanced profiler for the Intel CPU architecture is the Intel VTune Amplifier.[11] Profilers usually offer a basic analysis of program hotspots (where most of the cycles are spent). Advanced analysis uses fine-grained performance events to identify bottlenecks in the processor which are responsible for incurring many cycles during instruction execution [332].

Reading performance counters introduces a penalty itself, and therefore most profilers use statistical sampling of hardware performance events to minimize the side effects on program execution. If mapping to instructions is not required (for example in an initial phase of performance analysis or when the bottlenecks in the *code* are well understood), then PMUs can be used in counting mode where they are read only when required, for example, before and after program execution to count the total number of events. The Processor Counter Monitor[12] and Linux perf support this kind of lightweight analysis.

For the analysis of performance issues in very short actions (for example for real-time processing in car engines or the response time of graphical user interfaces),

[9] For Intel processors, the events are listed in the software developer manuals; see `www.intel.com/sdm`.

[10] `perf.wiki.kernel.org`

[11] `software.intel.com/en-us/intel-vtune-amplifier-xe`

[12] `github.com/opcm/pcm/`

sampling or counting performance events is not appropriate, because of the high relative overhead. Recently, Intel Processor Trace (PT) has been introduced to address the analysis of such performance issues related to responsiveness. PT allows one to trigger collection of a full instruction execution trace together with timestamps. Performance analysis based on PT is available in Intel VTune Amplifier and Linux perf.

B.9 Compiler

If not stated otherwise we used GNU C++ Compiler (g++) version 4.7.2 to compile our examples of shared-memory programs. The specific compiler options can be found in Makefiles at `github.com` under `basic-toolbox-sample-code/basic-toolbox-sample-code/`.

C

Support for Parallelism in C++

C++11 (a version of the C++ programming language standard) extends C++ by new language constructs for native support of parallelism through multithreading in shared-memory systems. The new constructs hide the implementation details of thread management. They provide basic locks, generic atomic operations, and support for asynchronous tasks. The C++14 version of the standard adds a new lock type for shared access. In our shared-memory implementation examples, we use only constructs available in C++11.

In the following sections we give a short introduction to the most important C++11/14 classes for multithreading.[1]

The parallel-programming support in C++ is not only rather recent but also fairly low-level. Hence, there are also tools outside the C++ standard that can be used for parallel programming with C++. Section C.5 introduces some frequently used ones.

C.1 "Hello World" C++11 Program with Threads

Listing C.1 shows a minimalistic program that demonstrates basic management of threads. It spawns the maximum number of threads supported by the hardware. These threads execute a user-defined worker function. The parameters of the worker function are passed in the constructor of the thread (the current thread count). Each C++ thread is scheduled for execution on one of the hardware threads available in the system. The C++ thread interface does not provide any methods to control the scheduling such that execution of C++ threads might be delayed and/or several C++ threads might need to share the same hardware thread [2]. In our simple example, each worker thread just prints its thread identifier passed as parameter to the C++ standard output (cout). Since the cout object is a shared resource, concurrent access to it might

[1] For an exhaustive overview, see the standard document [161] and the online C++ reference at `en.cppreference.com/w/`. A list of textbooks is available at `isocpp.org/get-started`.

[2] The only available scheduling call is a *yield*(), function giving a hint to deschedule the thread.

Listing C.1. Threading "hello world" example

```
#include <iostream>
#include <vector>
#include <thread>
#include <mutex>

using namespace std;

mutex m;

void worker(int iPE) {
  m.lock();
  cout << "Hello␣from␣thread␣"<< iPE << endl;
  m.unlock();
}

int main() {
  vector<thread> threads(thread::hardware_concurrency());
  int i = 0;
  for(auto & t: threads) t = thread(worker, i++);
  for(auto & t: threads) t.join();
  return 0;
}//SPDX−License−Identifier: BSD−3−Clause; Copyright(c) 2018 Intel Corporation
```

jumble the characters. To avoid this, we protect the access using an exclusive lock (mutex) provided by C++11. The main thread waits for completion of every worker thread by calling their *join* function.

Since spawning and joining threads are expensive operations involving operating system calls, it does not pay off to have a separate C++ thread for each small work item. For robust multithreaded applications, it is advisable to create all required threads at once (a *thread pool*) and to pass the work to them as needed using the load-balancing methods described in Chap. 14.

C.2 Locks

We introduced binary locks in Sect. 2.5. The corresponding C++11 class is *mutex* (mutual exclusion lock). More advanced lock classes provide the ability to allow timeouts (*recursive_timed_mutex*), to acquire a lock multiple times (*recursive_mutex*), or to distinguish between readers and writers (*shared_timed_mutex*). To make the usage of locks less error-prone, C++11 provides several helper classes. The class *lock_guard* acquires a C++11 mutex in its constructor and automatically releases it in its destructor when the execution leaves the scope where the guard was created. The function *lock* is passed multiple mutexes and locks all of them in such a way that no deadlocks can occur.

C.3 Asynchronous Operations

The classes *promise* and *future* provide mechanisms for thread interaction that are slightly higher-level than the basic locking mechanisms we use in this book. Roughly, one thread can produce a value, whereas others can wait for this value to become available. One useful variant of this method is an asynchronous function call that returns a *future* object so that at a later point, a thread can wait for the function call to be completed.

C.4 Atomic Operations

C++11 provides abstractions of atomic operations (Sect B.4) on shared-memory parallel computers. These operations include atomic load and store, exchange, strong and weak compare-and-exchange (equivalent to compare-and-swap (CAS); see Sect B.4), fetch-and-add, fetch-and-substract, fetch-and-or, fetch-and-and, and fetch-and-xor on C++ built-in integers and char types of length 8, 16, 32, and 64 bits. An atomic version of C++ *bool* exists too. *atomic_flag* is a Boolean variable supporting atomic *test_and_set* and *clear* operations that can easily be used to implement a simple spin lock.

C++11 atomic operations have (*_explicit*) versions that allow one to specify the memory-ordering guarantee (Sect. B.3) around an atomic operation. Their range is very rich, such that developers who are new to memory ordering should be very careful when choosing relaxed guarantees. A wrong memory ordering is a latent bug that is very hard to discover. The default and the safest memory-ordering type is *memory_order_seq_cst* (sequential consistency), which is recommended for beginners and is also relatively fast on common x86 architectures. This ordering provides many guarantees: No reads or writes can be reordered, all writes in other threads that access the same atomic variable are visible in the current thread, all writes in the current thread are visible in other threads that access the same atomic variable and writes that carry a dependency into the atomic variable become visible in other threads that access the same atomic variable, all threads observe all modifications in the same order. To enforce a required memory ordering between arbitrary operations (including nonatomic ones), C++ offers the *atomic_thread_fence* function. The weaker *atomic_signal_fence* prevents only reordering of the operations by the compiler but not by the processor. See also Sect. B.3.

The first versions of the C++11 compilers did not always implement the atomic operations listed above using the fastest available CPU instruction. Sometimes the heaviest and slowest CPU instruction was used. Therefore we recommend to use the latest compiler version.

Although the set of atomic operations and types provided in the C++11 standard is very rich, it still represents the lowest common denominator of the vendor-specific processor capabilities. For example, the 128-bit CAS operation which was very useful in Sect 4.6.3 is supported by the x86 architecture but is not part of the C++ standard.

C.5 Useful Tools

OpenMP. This is a compiler extension for shared-memory parallel programming; see also [64] and `openmp.org`. The basic idea behind OpenMP is that one annotates a sequential program with *compiler pragma*s that help the compiler to parallelize it. OpenMP supports SPMD programming, local (*private*) and global (*shared*) variables, and parallel loops. OpenMP supports locking directly, but can also be used together with other libraries. For example, one can use the class *std* :: *mutex* from the C++-standard.

Task-parallel programming. Since version 3.0, OpenMP has supported task-parallel programming. However, in the first implementations, performance was not very good. The Intel tools Cilk Plus[3] and Threading Building Blocks (Intel TBB[4]) give better performance using the work-stealing load balancer described in Sect. 14.5. However, algorithms using task-parallel programming sometimes do not scale beyond one processor chip, since memory access locality is not very good.

Software libraries. Good software libraries considerably ease the life of a software designer. A large variety of libraries is available for C++ and some of them exploit parallelism. Often, using these parallelized libraries can be the key to parallelizing an application. Important examples are libraries for linear algebra and parallel implementations of the the C++ standard template library STL. Note that STL not only supports classical algorithms like sorting, merging, selection, or random permutation, but also a comprehensive set of seemingly simple operations such as *for_each*. If those are also parallelized (perhaps using dynamic load balancing) this blurs the distinction between libraries and parallelization tools like OpenMP or task parallel programming – we can express many parallel algorithms as a set of STL calls. For example this has been done for a minimum spanning tree algorithm [245]. MCSTL [297] is a good parallelization of the STL and is part of the GNU C++ distribution.

C.6 Memory Management

The C++ memory allocation function (*new*) calls the underlying operating system allocator (e.g. *malloc* on Linux). The standard memory allocators are general-purpose and are not optimized for maximum scalability. Typically, small-sized allocations are serviced from a process-local heap protected by a per-process exclusive lock, which leads to scalability bottlenecks. Larger allocations are requested directly from the operating system which also involves locking of operating system memory structures responsible for bookkeeping (i.e., virtual memory page tables). Another issue is that, for security reasons, all allocated memory has to be initialized by the operating system before it can be given to an application. In Sect. 5.13, we saw an

[3] `www.cilkplus.org`
[4] `www.threadingbuildingblocks.org`

example where this initialization turned out to run sequentially – introducing a major scalability roadblock in our sample sort implementation. Thus it can be much faster to reuse memory rather than to allocate and deallocate it over and over again. To facilitate such reuse, Intel Threading Building Blocks and Boost libraries provide user-space memory pools that have standard allocation and deallocation interfaces (see *tbb* :: *memory_pool*[5] and *boost* :: *pool*[6]).

There are also libraries that replace the standard C++ allocators by more scalable implementations. The most known such allocators are Google's Thread Caching Malloc[7] and the Intel TBB scalable memory allocator. They provide per-thread heaps, avoiding global locks, and also automatically cache the memory in user pools for reuse. The TBB allocators additionally provide explicit interfaces for specific data structures that require scalable allocation. They also work with the C++ standard containers (*vector*, *stack*, etc.).

Sometimes there is a requirement to allocate memory on a boundary with a certain alignment. For example, the 16-byte CAS instruction of the x86 architecture requires 16-byte alignment. Also, to prevent false sharing (Sect. B.3), a data structure must begin at a fresh cache line. Common operating systems have custom allocators with support for alignment (*posix_memalign* on Linux and *_aligned_malloc* on Windows). Intel TBB provides a scalable allocator (*tbb* :: *cache_aligned_allocator*) which returns cache-line-aligned pieces of memory that can also be used with C++ containers.

As discussed in Sect. B.6, for performance reasons, the application might want to have control of the memory placement on specific sockets. Most operating systems have libraries and interfaces that support such control: For example, *libnuma* on Linux and *VirtualAllocExNuma* on Windows. See also the next section.

C.7 Thread Scheduling

By default, user threads have no guarantees about when and on what hardware threads or cores they will be executed. The operating system is allowed to deschedule and migrate them arbitrarily following some optimization goal (usually a heuristic). During its execution, a thread can be migrated from one hardware thread to another. In some cases thread migration can be very undesirable from a performance perspective: The migrated thread can no longer use its recently accessed cache lines (its *cache footprint*), it now has higher latency for accessing memory allocated on a different socket, etc. To prevent such migrations, the developer can *pin* a user thread to a set of hardware threads using the *pthread_setaffinity_np* call on Linux and *SetThreadGroupAffinity* on Windows.

On Linux, thread-pinning functions can be also used to control NUMA allocation. If a thread touches a virtual memory block which is not yet assigned to physical

[5] `software.intel.com/en-us/blogs/2011/12/19/scalable-memory-pools-community-preview-feature`

[6] `www.boost.org/doc/libs/1_48_0/libs/pool/doc/html/index.html`

[7] `goog-perftools.sourceforge.net/doc/tcmalloc.html`

memory (lazy memory allocation), the default policy for Linux is to try to allocate the physical memory on the local socket.

D

The Message Passing Interface (MPI)

MPI[1] is a software library for message passing in clusters. It is the de facto standard for high-performance computing. It was initially developed for Fortran and C in 1994. This lineage still shows in the function interfaces – the handling of data types is rather low-level. On the other hand, MPI offers a quite complete set of collective communication operations (see Sect D.3) that are missing from most alternatives. MPI is the result of a standardization process that is a careful compromise between performance, portability, and generality. Most of the functionality needed in this book is already in the MPI-1 standard. Therefore this is also the focus of this appendix. MPI-2, from 1997, adds high-performance I/O and one-sided communication – a way to get some of the functionality one uses in shared-memory programs, albeit with some performance overhead. MPI-3, from 2012, adds nonblocking collective operations.

D.1 "Hello World" and What Is an MPI Program?

An MPI program is an ordinary "sequential" (Fortran), C, or C++ program that includes `mpi.h` and calls the MPI functions declared there. This program is executed in parallel on all PEs of our parallel machine, i.e., MPI programs follow the SPMD approach to parallel programming. Listing D.1 shows a minimalistic example.

The call *MPI_Init* initializes the library. In particular, it initializes a global variable *MPI_COMM_WORLD* that stores a *communicator* object describing the set of PEs available to the program. The procedures *MPI_Comm_size* and *MPI_Comm_rank* extract the total number of PEs p and the PE number (from $0..p-1$). Communicators are also passed to all MPI functions to define the context of the communication. In particular, it is possible to define communicators spanning only a subset of the PEs in *MPI_COMM_WORLD*. In our simple example, each PE outputs p and its processor number i. The overall output of our program is not

[1] `www.mpi-forum.org`

uniquely defined. The characters or lines of output coming from different PEs may be jumbled or the output may be written to one file for each PE.

Listing D.1. MPI hello world example

```
#include <iostream>
#include <mpi.h>

int main(int argc, char** argv)
{ int p, i;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Comm_rank(MPI_COMM_WORLD, &i);
  std::cout << "PE " << i << " out of " << p << std::endl;
  MPI_Finalize();
}
```

In MPI terminology, a PE is called a *process* and, indeed, an MPI process often corresponds to a process in the host operating system. Whether each node of the cluster runs one or several processes depends on how the MPI program is launched. One important option is one process per core (or hardware thread) of the node. This is convenient because then MPI takes care of all the parallelism. At the other extreme, we might have only one process per node. In order to exploit the actual parallelism of the machine, the node should then run a multithreaded shared-memory parallel program. In that case it is advisable that, at any time, only a single thread per node makes MPI calls.[2] If the multithreaded and message-passing parts of the program work well together, this option may lead to better performance at the price of a more complicated program. The middle ground may also make sense. For example, one could run one MPI process on each socket, thus explicitly taking NUMA effects into account.

To actually start our example program, the simplest case is when it runs on the cores of a single machine. For example, under Linux with OpenMPI v3.0.0 (`www.open-mpi.org/`) one would compile the program with the command line

```
mpic++ example.cpp -o example
```

where `mpic++` is a script that calls the GNU compiler with appropriate parameters. To then run the program using four processes one uses another script

```
mpirun -np 4 example
```

On a supercomputer, starting the program is a bit more complicated. One typically writes a configuration file describing which program to call, how many nodes should be used and how many processes run on each node. Then one passes this configaration file to a job scheduler that allocates the appropriate ressources and starts the program on all nodes.

[2] If MPI calls are made concurrently, some MPI implementations do not work at all and others have performance problems.

D.2 Point-to-Point Communication

Supposing message m is an array of k integers, our pseudocode operation $send(i,m)$ can be translated into the MPI call

$$MPI\_Send(\&m,k,MPI\_INT,i,t,MPI\_COMM\_WORLD)$$

where the integer t is a *message tag* that helps the receiver to distinguish different types of messages. The destination PE i is a rank within the global communicator *MPI_COMM_WORLD*. Other communicators can also be used which can encode subsets of PEs – for example the rows and columns in Sect. 5.2. Further data types can be used by replacing the constant *MPI_INT* by another predefined constant such as $MPI\_CHAR/SHORT/LONG/FLOAT/DOUBLE$. User-defined data types are also possible.

There are several variants of send operations – *Send*/*Ssend*/*Isend*/*Issend*/*Bsend*. When the ordinary send returns, the message buffer m can be reused (and overwritten) without affecting the delivery of the message. *Ssend* guarantees in addition that the receiver has begun to actually receive the message. The *nonblocking* or *immediate* operations *Isend* and *Issend* are a little more complicated. They have an additional return parameter that returns a *request object*. Their buffer can only be reused when an additional (blocking) operation waiting for the request to finish has been called. The advantage of nonblocking operations is that they return immediately. Thus, several communications can be initiated together. One can also use this feature to overlap communication and internal work. Function *MPI_Bsend* also returns immediately *and* guarantees that the message buffer can be immediately removed. The disadvantage of this convenience is that the user has to supply additional buffer memory (using the operation *MPI_Buffer_attach*) and that this causes additional copy operations, which incur some overhead.

A receive operation matching the above call is

$$MPI\_Recv(\&m,k,MPI\_INT,j,t,MPI\_COMM\_WORLD,\&status)$$

where j either specifies the PE from which the caller expects to receive a message or is equal to *MPI_ANY_SOURCE*. In the latter case, a message from any sender can be received. Similarly, t specifies the expected tag or *MPI_ANY_TAG*. The parameter k specifies the allocated length of the message buffer. This buffer may be longer then the message actually received. The actual length of the received message can be read from the status variable (which has type *MPI_Status*) using the operation *MPI_Get_count*. A status object has fields *status.MPI_TAG* and *status.MPI_SOURCE* that tell the tag and sender, respectively, of the received message. Sometimes the receiver does not have a useful upper bound on the length of the message to be received. In that case, the operation *MPI_Probe* can be called first which delivers a status that tells the message length (and its tag and source PE). There is also a nonblocking receive operations *MPI_Irecv*.

Finally, there is an operation *MPI_Sendrecv* that corresponds to our pseudocode operations $send(\cdots) \parallel receive(\cdots)$.

To work with the nonblocking operations *Isend* and *Irecv*, one additionally needs operations *MPI_Wait/Waitany/Waitall* and *MPI_Test/Testany/Testall*. These operations are passed the request objects returned by *Isend* and *Irecv*. The wait operations block until the specified requests have finished. The test operations do not block, and thus allow us to perform computations while communication operations are executed in the background.

D.3 Collective Communication

Table D.1. Collective communication operations in MPI.

Our name	MPI name	See also Sect.
broadcast	`MPI_Bcast`	13.1
reduce	`MPI_Reduce`	13.2
all-reduce	`MPI_Allreduce`	13.2
prefix sum	`MPI_Scan`	13.3
barrier	`MPI_Barrier`	13.4.2
gather	`MPI_Gather(v)`	13.5
all-gather	`MPI_Allgather(v)`	13.5
scatter	`MPI_Scatter(v)`	13.5
all-to-all	`MPI_Alltoall(v)`	13.6

MPI supports all the collective communication operations discussed in Chap. 13. Starting with MPI 3.0, this includes the asynchronous ones presented in Sect. 13.7. We view this as a major strength of MPI in particular in comparison with other frameworks for parallel processing. However, one should not assume that all MPI implementations implement all collective operations efficiently. Careful profiling and occasional manual reimplementations of the required operations are therefore important for achieving good performance in practice. Table D.1 summarizes the available collective operations. The collective operations with irregular message size have names ending with `v`. These expect the receiver of a message to specify the length of that message. This often implies that the message lengths have to be transferred in a separate operation.

An example call for a collective operation is

MPI_Reduce(&c, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)

which will perform a sum-reduction of the values of the local variable *c*. The second to last parameter specifies that the overall result will be stored in the variable *sum* at PE 0. MPI supports a number of further predefined reduction operations besides *MPI_SUM*. User-defined operations are also possible.

E

List of Commercial Products, Trademarks and Software Licenses

The following list includes the names of commercial products and trademarks mentioned in the book.

- Microsoft® Windows® (Windows)
- Oracle® Java®
- IBM® RS/6000®
- IBM® Power®
- IBM® POWER8®
- IBM® Blue Gene®/Q
- Intel® Core™
- Intel® Xeon®
- Intel® Pentium®
- Intel® Threading Building Blocks (Intel TBB)
- Intel® Cilk™ Plus
- Intel® VTune™ Amplifier
- Intel® Processor Trace (Intel PT)
- Intel® Transactional Synchronization Extensions (Intel TSX)
- Wikipedia®
- OpenMP®

The listings in the book are distributed under the Open Source BSD-3-Clause license.

E.1 BSD 3-Clause License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.