Die Methode `fetch_and_add` ist jedoch nur bei ganzzahligen Typen und Zeigertypen verfügbar. Bei Letzteren gelten die Regeln der Zeigerarithmetik. Wenn zum Beispiel ein `int`-Wert vier Bytes benötigt und `x` vom Typ `atomic<int*>` ist, wird `x` bei einem Aufruf von `x.fetch_and_add(2)` um acht Bytes weitergeschaltet. Dem Komfort und der besseren Lesbarkeit zuliebe kann man anstelle von `fetch_and_add` auch die Inkrement- und Dekrementoperatoren sowie die Operatoren `-=` und `+=` verwenden.

Mithilfe des Aufzählungstyps `atomic<T>::memory_semantics` lässt sich die Speicherbarriere (siehe Abschnitt 2.3.1, S. 33) spezifizieren. Es stehen zwei Konstanten zur Verfügung: `acquire` und `release`. In der Voreinstellung kommen für Lesezugriffe ein Acquire Fence und für Schreibzugriffe ein Release Fence zum Einsatz. Für Operationen wie `compare_and_swap`, die einen Wert lesen und verändern, gilt die sequenzielle Konsistenz. Um eine von der Voreinstellung abweichende Speicherbarriere zu verwenden, übergibt man die entsprechende Konstante als Template-Argument, z. B. `x.compare_and_swap<release>(1, 2)`.

10.2 Taskparallelität

Wie wir im ersten Teil des Buches gesehen haben, lassen sich viele Probleme mit Tasks parallelisieren (siehe Abschnitt 4.1). TBB bietet dafür mehrere Möglichkeiten an. Der einfachste Weg ist die parallele Ausführung mehrerer Funktionen. Wem das nicht reicht, der sollte Taskgruppen einsetzen. Für maximale Effizienz lohnt sich ein Blick auf die klassische Schnittstelle des Task-Schedulers, die in [33, 34] ausführlich beschrieben ist.

10.2.1 Parallele Funktionsaufrufe

Für die parallele Ausführung mehrerer Funktionen gibt es das Funktionstemplate `parallel_invoke`, das bis zu zehn Funktoren, Lambda-Funktionen oder Funktionszeiger entgegennimmt. Da `parallel_invoke` erst dann zurückkehrt, wenn alle Tasks fertig sind, muss man sich nicht um die Synchronisation kümmern.

Ein typisches Anwendungsgebiet von `parallel_invoke` sind rekursive Algorithmen nach dem »Teile und herrsche«-Prinzip (siehe Abschnitt 4.1.2). Um das Ergebnis einer Berechnung an die aufrufende Funktion zurückzugeben, muss man den Funktoren eine Referenz oder einen Zeiger auf die Variable übergeben, in der das Ergebnis gespeichert werden soll. Die in Listing 10.6 gezeigte Funktion verdeutlicht dies am Beispiel der Berechnung der Fibonacci-Zahlen.

Voraussetzung für den Einsatz von `parallel_invoke` ist, dass an der Stelle des Aufrufs die Anzahl der dort auszuführenden Tasks zur Übersetzungszeit feststeht. Möchte man eine variable Anzahl von Tasks parallel ausführen oder zur Laufzeit entscheiden, welche Tasks überhaupt ausgeführt werden sollen, muss man Taskgruppen einsetzen oder direkt auf den Task Scheduler zurückgreifen.

```
int fibonacci(int n) {
  if(n < 2)
    return n;
  int x, y;
  parallel_invoke([&x, n] {x = fibonacci(n-1);},
                  [&y, n] {y = fibonacci(n-2);});
  return x+y;
}
```

Listing 10.6 *Parallele Berechnung der Fibonacci-Zahlen mit* `parallel_invoke`

10.2.2 Taskgruppen

Eine Gruppe von Tasks wird durch ein Objekt der Klasse `task_group` repräsentiert. Zum Starten eines Tasks ruft man die Methode `run` auf:

```
template<typename Func> void task_group::run(const Func& f);
```

Die Tasks einer Gruppe können durch einen Aufruf von `cancel` abgebrochen werden. Die Synchronisation erfolgt durch die Methode `wait`, die erst dann zurückkehrt, wenn alle Tasks beendet sind. Sie muss vor dem Destruktor aufgerufen werden, sonst wird eine Ausnahme geworfen. Der Rückgabewert von `wait` ist ein Wert vom Aufzählungstyp `task_group_status`. Es wird entweder `complete` oder `canceled` zurückgegeben. Ein Aufruf von `run` mit einem anschließenden `wait` lässt sich auch »in einem Rutsch« mittels `run_and_wait` erledigen.

Listing 10.7 zeigt den Einsatz von Taskgruppen am Beispiel des Quicksort-Algorithmus (aus Gründen der Einfachheit verzichten wir auf die Darstellung der Funktion `partition`). Um die Teilbereiche parallel sortieren zu können, verpacken wir die beiden Aufrufe von `quicksort` in Lambda-Funktionen und übergeben diese der Taskgruppe `group`. Wie in Abschnitt 4.1.3 angedeutet, besteht keine Notwendigkeit, bei jedem Aufruf eine neue Gruppe anzulegen. Deshalb verwenden wir eine globale Gruppe, die wir `quicksort` per Referenz übergeben. Die Rekursion endet, wenn der durch `from` und `to` gegebene Teilbereich die minimale Blockgröße (`grainsize`) unterschreitet. In diesem Fall sortieren wir den verbleibenden Teilbereich sequenziell durch Aufruf der STL-Funktion `sort`, um den Mehraufwand für die Taskerzeugung und -Verwaltung zu reduzieren.

Um ein Feld zu sortieren, legen wir eine Taskgruppe an, rufen die Funktion `quicksort` auf und warten anschließend, bis alle Tasks beendet sind:

```
vector<int> v;
// ... Initialisierung
task_group group;
quicksort(v.begin(), v.end(), 1000, group);
group.wait();
```

```
template <typename RandomAccessIterator>
void quicksort(const RandomAccessIterator from,
               const RandomAccessIterator to,
               const int grainsize,
               task_group& group) {
  if(size_t(to-from) > grainsize) {
    const RandomAccessIterator pivot(partition(from, to));
    group.run([=, &group] {
      quicksort(from, pivot, grainsize, group);
    });
    group.run([=, &group] {
      quicksort(pivot+1, to, grainsize, group);
    });
  } else
    sort(from, to);
}
```

Listing 10.7 *Paralleler Quicksort-Algorithmus mit Taskgruppen*

10.2.3 Task Scheduler

Seit der TBB-Version 2.2 ist es nicht mehr notwendig, den Task Scheduler explizit zu initialisieren. Dennoch kann dies nützlich sein, z. B. um im Rahmen von Skalierbarkeitsmessungen die Anzahl der Threads festzulegen (standardmäßig wird für jeden Kern ein Thread erzeugt). Die Initialisierung des Task-Schedulers erfolgt über die Klasse `task_scheduler_init`. Die Anzahl der Threads kann dabei sowohl im Konstruktor als auch beim Aufruf der Methode `initialize` angegeben werden. Die Voreinstellung lässt sich durch Aufruf der statischen Methode `default_num_threads` abfragen.

Listing 10.8 zeigt verschiedene Möglichkeiten der Initialisierung des Task-Schedulers auf, wobei die Anzahl der Threads beim Programmaufruf übergeben wird. Die Angabe von `task_scheduler_init::deferred` im Konstruktor von `task_scheduler_init` verhindert, dass der Scheduler mit den Standardwerten initialisiert wird. Wenn der Benutzer beim Programmaufruf den Wert 0 angegeben hat, wird die Anzahl der Threads automatisch bestimmt. Andernfalls wird der Scheduler mit der Anzahl der angegebenen Threads initialisiert. Falls diese größer als die durch `default_num_threads()` gegebene Voreinstellung ist, wird eine Warnung ausgegeben.

10.3 Datenparallelität

TBB stellt eine Reihe von Funktionstemplates für die parallele Ausführung von Schleifen zur Verfügung. Neben »einfachen« Schleifen ohne Datenabhängigkeiten werden auch Reduktionen und Präfixberechnungen unterstützt. Die Teilprobleme

```
int main(int argc, char *argv[]) {
  // ... überprüfe Anzahl der Argumente
  const int threads(atoi(argv[1]));
  task_scheduler_init init(task_scheduler_init::deferred);
  if(threads == 0)
    init.initialize(task_scheduler_init::automatic);
  else {
    if(threads > task_scheduler_init::default_num_threads())
      cout << "Achtung: mehr Threads als Kerne" << endl;
    init.initialize(threads);
  }
  // ... benutze den Task Scheduler
}
```

Listing 10.8 *Initialisierung des Task-Schedulers*

werden dabei auf Tasks abgebildet. Da die Erzeugung und Synchronisation der Tasks automatisch vonstattengeht, muss man sich als Benutzer lediglich um die anwendungsspezifische Funktionalität kümmern.

10.3.1 Schleifen ohne Datenabhängigkeiten

Schleifen ohne Datenabhängigkeiten (siehe Abschnitt 4.2.1) können mit der Funktion `parallel_for`, die wir bereits in der Einleitung zu diesem Kapitel kennengelernt haben, parallel ausgeführt werden. Die Funktion `parallel_for` gibt es in verschiedenen Varianten. Im einfachsten Fall wird neben dem Iterationsbereich `[first, last)` nur der auszuführende Funktor übergeben:

```
template<typename Index, typename Func>
Func parallel_for(Index first, Index last, const Func& f);
```

Optional lässt sich auch eine positive Schrittweite `step` angeben:

```
template<typename Index, typename Func>
Func parallel_for(Index first, Index last, Index step,
                  const Func& f);
```

Bei beiden Fällen erfolgt die Partitionierung zur Laufzeit durch TBB. Als Benutzer hat man somit keinen Einfluss auf die Anzahl und die Größe der Blöcke. Außerdem eignen sich die gezeigten Varianten von `parallel_for` nur für eindimensionale Felder, da es sich bei dem Templateparameter `Index` um einen ganzzahligen Typ handeln muss. Die folgende Variante ist universeller einsetzbar und kann auch mit anderen, z. B. durch Iteratoren definierten Iterationsbereichen umgehen:

```
template<typename Range, typename Body>
void parallel_for(const Range& range, const Body& body
                  [, Partitioner& partitioner]);
```