Der Unterschied im Verhalten der beiden Programme liegt darin, wie die Threads ihre Aufgaben ausführen und wie die Arbeitslast verteilt ist.

**Listing 3.1 ("Hello World")**

*Verhalten:* Die Threads geben ihre Nummern in zufälliger Reihenfolge aus.

*Erklärung:* In diesem Programm führt jeder Thread die printf-Anweisung innerhalb des #pragma omp parallel Blocks unabhängig von den anderen aus. Da es keine Synchronisation gibt und die Operation sehr schnell ist, hängt die Reihenfolge der Ausgaben von der Thread-Planung des Betriebssystems und der OpenMP-Laufzeitumgebung ab. Dies führt zu einer nicht-deterministischen Reihenfolge der Ausgaben.

**Listing 3.2 (Berechnung der Fibonacci-Zahlen)**

*Verhalten:* Die Threads geben ihre Ergebnisse in einer scheinbar geordneten Reihenfolge aus.

*Erklärung:* Obwohl die Threads parallel arbeiten, variiert die Dauer der Berechnungen stark, da die Funktion fib() für größere Werte von n (in diesem Fall n + t mit t ==Thread-Id) exponentiell mehr Zeit benötigt. Threads, die kleinere Fibonacci-Zahlen berechnen (weil ihre Id ja kleiner ist), sind schneller fertig und geben ihre Ergebnisse früher aus. Dies führt dazu, dass die Ausgaben in einer Reihenfolge erscheinen, die den aufsteigenden Thread-IDs entspricht, obwohl keine explizite Synchronisation stattfindet.

**Zusammenfassung**

* *Listing 3.1:* Die zufällige Reihenfolge der Ausgaben resultiert aus der unsynchronisierten und schnellen Ausführung der printf-Anweisungen durch die Threads.
* *Listing 3.2:* Die geordnete Ausgabe entsteht durch die unterschiedliche Arbeitslast der Threads. Die Berechnungszeit von fib() führt dazu, dass Threads mit weniger Arbeit (kleinere t + n Werte) ihre Ergebnisse früher ausgeben, was einen indirekten Synchronisationseffekt erzeugt.

Listing 3.10 verwendet OpenMP-Parallelisierung (#pragma omp parallel for collapse(2)), um die verschachtelten Schleifen, die das size x size-Gitter durchlaufen, parallel auszuführen.

**Effiziente Nutzung der Kerne:**

1. **Parallelisierung mit OpenMP:**
   * Der collapse(2)-Parameter führt dazu, dass die beiden verschachtelten Schleifen in eine einzige Schleife zusammengeführt werden, die dann von OpenMP auf die verfügbaren Kerne verteilt wird. Dies vermeidet das Bottleneck, das entstehen würde, wenn nur die äußere Schleife parallelisiert würde, was zu schlechterer Lastverteilung führen könnte.
2. **Gittergröße und Kernauslastung:**
   * Bei großen Werten für size (z. B. Tausende oder Millionen) ist die Anzahl der Iterationen (size x size) ausreichend hoch, um die verfügbaren Kerne effektiv auszulasten. Jeder Kern erhält eine angemessene Anzahl von Iterationen 🡪 Leerlaufzeiten werden minimiert und die Auslastung maximiert.
   * Bei kleinen Werten für size könnte die Anzahl der Iterationen zu gering sein, um alle Kerne auszulasten, was zu einer ineffizienten Nutzung führt. Somit könnte der Overhead durch Thread-Management den Vorteil der Parallelisierung überwiegen.
3. **Unabhängige Iterationen:**
   * Jede Zelle wird unabhängig von den anderen aktualisiert, da nur die Nachbarn berücksichtigt werden. Dieser sogenannte "[embarrassingly parallel](https://en.wikipedia.org/wiki/Embarrassingly_parallel)"-Charakter ist ideal für Parallelisierung, da keine Synchronisation zwischen den Threads erforderlich ist.

**Potenzielle Ineffizienzen:**

1. **Speicherzugriffsmuster:**
   * Der Zugriff auf plane[i][j] und die Aktualisierung von aux\_plane[i][j] können zu nicht-sequenziellen Speicherzugriffen führen, insbesondere wenn das Gitter als 2D-Array gespeichert wird. Dies kann eine schlechte Cache-Nutzung verursachen, da Kerne mehr Zeit mit dem Warten auf Speicherzugriffe als mit Berechnungen verbringen könnten. 🡪 Ineffizient
   * (Die Funktion neighbors() könnte ebenfalls mehrere benachbarte Zellen lesen. Wenn dieser Zugriff nicht speicherlokal erfolgt, verschlechtert dies die Effizienz weiter.)
2. **Thread-Overhead:**
   * Bei kleinen Gittern kann der Overhead für das Erstellen und Synchronisieren von Threads den Nutzen der Parallelisierung übersteigen, insbesondere wenn die Anzahl der Iterationen kleiner ist als die Anzahl der verfügbaren Kerne. (siehe oben)
3. **Lastverteilung:**
   * Durch das collapse-Attribut wird zwar eine gleichmäßige Verteilung der Iterationen gewährleistet, aber wenn die Berechnung der Nachbarn unterschiedlich aufwändig ist (z. B. durch spezielle Randbedingungen), kann es zu Ungleichgewichten zwischen den Kernen kommen, was die Gesamtleistung reduziert. 🡪 Lösung: vielleicht die Aufgaben auf Threads dynamisch verteilen, abhängig von der Größe des Gitters (siehe unten)

**Optimierungsvorschläge:**

1. **Optimierung der Speicherzugriffe:**
   * Linearisierung des 2D-Arrays kann die räumliche Lokalität verbessern und die Cache-Nutzung optimieren.
2. **Dynamische Lastverteilung:**
   * Durch die Verwendung von OpenMP's schedule(dynamic) kann die Lastverteilung bei unterschiedlichem Rechenaufwand pro Zelle verbessert werden. (aber dann vielleicht mehr Overhead)
3. **Anpassung der Parallelisierung bei kleinen Gittern:**
   * Für sehr kleine Gitter könnte eine dynamische Entscheidung sinnvoll sein, ob überhaupt parallelisiert wird, da eine sequenzielle Berechnung aufgrund des geringen Overheads schneller sein könnte. (laut Google: #pragma omp parallel if (…))

**Fazit:**

Der Code nutzt die verfügbaren Kerne bei großen Gittergrößen effizient, da er OpenMP-Parallelisierung mit collapse(2) verwendet, um die Berechnungen gleichmäßig auf Threads zu verteilen. Bei kleinen Gittern sinkt die Effizienz jedoch aufgrund von Thread-Overhead und möglicherweise ineffizienten Speicherzugriffen. Optimierungen der Speicherstruktur und dynamische Scheduling-Strategien können die Performance weiter verbessern.

**Änderungen am Code**

* + Statt der C++-Bibliothek <random> wird nun die in Example 3.5 genutzte Funktion rnd() verwendet. Diese ist ein einfacher deterministischer Zufallszahlengenerator, der auf einer linearen Kongruenzmethode basiert.
  + Für jeden Thread wird der Seed basierend auf der Thread-Nummer (omp\_get\_thread\_num()) initialisiert. Dadurch hat jeder Thread einen eindeutigen Startwert.
  + Die Schleife für die Monte-Carlo-Simulation wurde so angepasst, dass jede Iteration mit dem neuen rnd() Zufallszahlen für x und y generiert.
  + Die Nutzung von <random> und den zugehörigen Objekten (default\_random\_engine, uniform\_real\_distribution) wurde entfernt.

**Auswirkungen auf die Performance**

* + Die Laufzeit konnte reduziert werden, da die neue Funktion rnd() im Vergleich zur <random>-Bibliothek weniger Rechenaufwand benötigt. Besonders bei einer hohen Anzahl an Zufallszahlen ist der Unterschied da.
  + Durch den Wegfall von komplexen Zufallszahlengeneratoren wird weniger Speicher für Objekte wie default\_random\_engine benötigt.
  + Die Genauigkeit der π-Berechnung bleibt nahezu gleich, da die Monte-Carlo-Methode hauptsächlich von der Anzahl der Zufallszahlen abhängt und nicht von der Komplexität des Zufallszahlengenerators.
  + Mit rnd() sind die Ergebnisse deterministisch und für jede Anzahl an Threads reproduzierbar, da der Zufallszahlengenerator bei gleichem Seed immer dieselben Zahlen liefert.