OpenMP

Ich habe überlegt, über OpenMP zu recherchieren und mir eine Zusammenfassung zu erstellen, die mir in der Zukunft helfen wird.

OpenMP (Open Multi-Processing) = API, die paralleles Programmieren auf Mehrkernprozessoren ermöglicht. Für **Shared-Memory-Systeme** (bei denen alle Prozessoren oder Threads denselben physischen Speicher (RAM) nutzen)

- **Sprachen: C**, **C++**, **Fortran**.
- **Komponenten:** Besteht aus "compiler directives" - Direktiven (`#pragma`), "library routines" - Laufzeitfunktionen (`omp_*`) und "environment variables" - Umgebungsvariablen (`OMP_*`).

Must-Do's um mit OpenMP zu arbeiten:

1. **OpenMP-Header einfügen:** stellt OpenMP-Funktionen bereit, definiert Makros

```
    #include <omp.h>
```

2. **Compiler-Flag aktivieren:** je nach Compiler...
   - GCC: `-fopenmp`
   - MSVC: `/openmp`

OpenMP-Direktiven

- "Direktiven" hier = Anweisungen mit `#pragma omp ...`, die dem Compiler sagen, WIE der Code PARALLEL ausgeführt werden soll

`#pragma omp parallel` - parallele Region. Master-Thread erstellt mehrere Threads und jeder Thread führt denselben Codeblock aus!

```
#pragma omp parallel
{
    printf("Thread Nr. %d\n", omp_get_thread_num());
}
```

`#pragma omp critical` - nur EIN Thread darf (zur gleichen Zeit) auf den Code im critical-Block zugreifen und den ausführen - alle anderen Threads müssen warten, bis der aktuelle Thread fertig ist (globaler Lock)

```
#pragma omp critical
{
    shared_variable++;
}
```

`#pragma omp atomic` - "automare Operation" = einfache Speicheroperation (Addition, Subtraktion, Multiplikation) wird als UNTEILBAR angesehen - kein anderer Thread darf den Wert verändern (funktioniert ohne locks)

```
#pragma omp atomic
counter++;
```

`#pragma omp for` - die Schleife wird automatisch auf Threads verteilt

..Und wie wird es entschieden, wie groß die Regionen bei `#pragma omp for` sind?

Anzahl der Iterationen, die ein Thread verarbeitet hängt vom **schedule**-Mechanismus ab. Es gibt 3 wichtige Schedulings:

1. **static [,chunksize pro thread]:** [DEFAULT] statische gleichmäßige Aufteilung - zur Kompilierzeit steht schon fest, welcher Thread und was er machen wird (wenn wir die Anzahl von Threads kennen) **Beispiel:** Bei 4 Threads und 16 Iterationen macht jeder Thread 4 Iterationen.

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < 16; i++) {
    printf("Thread␣Nr␣%d␣bearbeitet␣i␣=␣%d\n", omp_get_thread_num(), i);
}
```

   Ausgabe:

```
Thread Nr 0 bearbeitet i = 0, 1, 2, 3
Thread Nr 1 bearbeitet i = 4, 5, 6, 7
Thread Nr 2 bearbeitet i = 8, 9, 10, 11
Thread Nr 3 bearbeitet i = 12, 13, 14, 15
```

2. **dynamic [,chunksize pro thread]** wenn ein Thread mit seiner aktuellen Aufgabe fertig ist, werden ihm Arbeitspakete dynamisch (zur Laufzeit) vergeben - so, wie es am effizientesten ist!

```
#pragma omp parallel for schedule(dynamic)
```

3. **guided:** wie `dynamic`, aber die Größe der Aufgaben verringert sich mit der Zeit! Das bedeutet: zu Beginn werden große Chunks von Iterationen an Threads vergeben. Wenn die Threads ihre Aufgaben fertig machen, wird die Größe der neuen Chunks immer kleiner. **Vorteil:** Threads, die schneller fertig sind, können schneller kleinere Chunks aufnehmen

Wichtige Funktionen aus `<omp.h>`

- `omp_get_thread_num()`: gibt ID des aktuellen Threads zurück
- `omp_get_num_threads()`: gibt die Anzahl der Threads in der parallelen Region zurück
- `omp_set_num_threads(n)`: setzt die Anzahl der Threads
- `omp_get_max_threads()`: gibt die max Anzahl der verfügbaren Threads zurück