Abgabe zur Übung 3

Felix Wolff - 765508
Johannes Radmer - 790147

Aufgabe 3.1: Parallele Berechnung von PI mit OpenMP

3.1.1:

Siehe Code.

3.1.2:

Die OpenMP Variante ist deutlich kürzer, einfacher zu implementieren und zu verstehen. Es fällt Code zum Erstellen und Verwalten von Threads weg und es wird sogar parallel zusammengezählt.

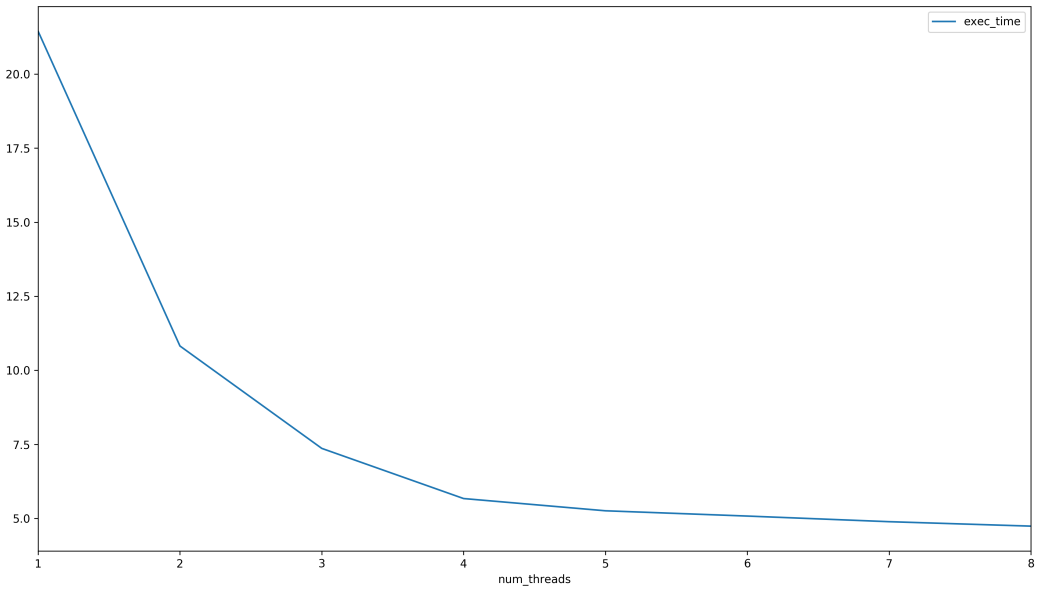
3.1.3:

(Siehe Code)
Eine Möglichkeit den Zufallszahlengenerator thread-sicher zu machen ist, jedem Thread seinen eigenen, lokalen Zufallszahlengenerator-Zustand zu geben. So kommen sich Threads nicht beim Verwenden in die Quere, indem der Zustand überlappend geändert wird. Wie von uns implementiert werden keine weiteren Synchronisations-Direktive benötigt, es muss nur jeder Thread seine eigne *state* Variable bekommen. Dies ist genau so auch mit pthreads (oder anderen Threading Bibliotheken/Modellen) möglich.

```
Relative Error Statistics for 100 runs.
```

	num_samples	mean	std	median
0	10000	0.007514	0.0	0.007514
1	100000	0.001492	0.0	0.001492
2	1000000	0.000365	0.0	0.000365
3	10000000	-0.000008	0.0	-0.000008
4	100000000	-0.000047	0.0	-0.000047
5	1000000000	0.000003	0.0	0.000003

Wie die Tabelle zeigt ist die Standartabweichung des relativen Fehlers bei der Implementierung mit lokalem Zufallszahlengenerators 0, da das Programm nun deterministisch arbeitet.
Die Skalierbarkeit der OpenMP-Implementierung ist auch weiterhin sehr gut:



3.1.4:

Siehe elektronishe Abgabe.

Aufgabe 3.2: Kollektive MPI-Operationen

3.2.1:

MPI_Barrier erhällt ein MPI communicator Objekt als Parameter. Der Aufruf blockiert, bis alle Prozesse die dem communicator angehören MPI_Barrier aufgerufen haben.

3.2.2:

Pseudocode für MPI_Barrier Implementierung:

```
int MPI_Barrier(MPI_Comm  comm) {
  int comm_size;
  MPI_Comm_size(comm,  &comm_size);
  if (rank > 0) {
    err = MPI_Send(NULL,  0, MPI_BYTE,  0, MPI_ANY_TAG,  comm);
    if (err != MPI_SUCCESS)  {
      return err;
    }
    err = MPI_Recv(NULL,  0, MPI_BYTE,  0, MPI_TAG_ANY,  comm);
    if (err != MPI_SUCCESS)  {
      return err;
    }
  }
  else {
    for (int i = 0; i < comm_size;  i++) {
      err = MPI_Recv(NULL,  0, MPI_BYTE, MPI_SOURCE_ANY,
MPI_TAG_ANY,  comm);
      if (err != MPI_SUCCESS)  {
        return err;
      }
    }
    for (int i = 0; i < comm_size;  i++) {
      err = MPI_Send(NULL,  0, MPI_BYTE,  i, MPI_TAG_ANY,  comm);
      if (err != MPI_SUCCESS)  {
        return err;
      }
    }
  }
}
```

Aufgabe 3.3: Parallelisierung des Zellularautomaten

3.3.1:

(Siehe Code, elektronische Abgabe)

Die OpenMP-Implementierung ist mehr als 5 mal schneller auf einem Core i7 (4 Kerne, 8 Threads) und liefert den selben Hash-Wert:

```
time ./omp_caseq 10000 1000
hash: 512ED532A6868DDEC4DE63544E0FC167
real 0m6.242s
user 0m41.740s
sys 0m0.231s
time ./caseq 10000 1000
hash: 512ED532A6868DDEC4DE63544E0FC167
real 0m32.733s
user 0m32.687s
sys 0m0.034s
```