Blatt 05 Antworten

Aufgabe 1.

1. Seitengröße N in Abhängigkeit vom interlines Parameter i: $N(i) = 8 * i + 9$.

   Sei $k(i)$ die Anzahl der zu bearbeiten Aufgaben. Z.b. $k(i) = N(i) - 2$ wenn wir eine Zeilen oder Spaltenweise Aufteilung machen, und $k(i) = (N(i) - 2)^2$ wenn wir eine Elementweise Aufteilung machen. Die matrix hat eine Größe von $N \times N$, jedoch wird die erste und letzte spalte und Zeile nicht berechnet.

   Nun wollen wir $c(k, t, j)$ definieren so dass die Anzahl der Aufgaben k gleichmäßig Aufgeteilt werden auf die threads mit thread id $j, 0 \leq j < t$.

   $c(k, t, j) = \lfloor \frac{k}{t} \rfloor + (j < (k \mod t))$

   Wir können auch eine explicit formel $n_j(k, t)$ für die grenzen zwischen threads aufschreiben (der grenzwert gehört zum thread mit der niedrigen id)

   $n_j(k, t) = j * \lfloor \frac{k}{t} \rfloor + \min(j, k \mod t)$

   Wenn wir nun $k = N(i) = 8 * i + 9$ einsetzen, bekommen wir:

   $c(i, t, j) = \lfloor \frac{8*i+7}{t} \rfloor + (j < ((8 * i + 7) \mod t))$

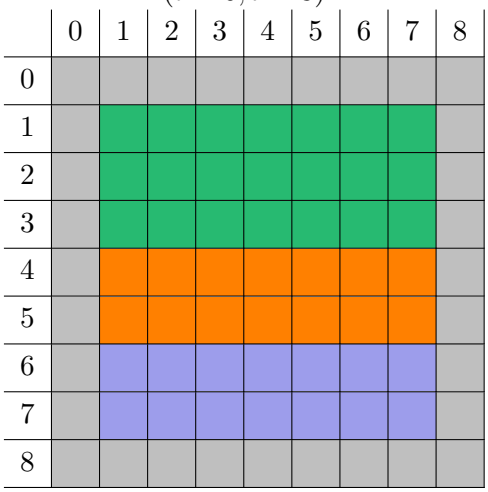
   $n_j(i, t) = j * \lfloor \frac{8*i+7}{t} \rfloor + \min(j, (8 * i + 7) \mod t)$

2.

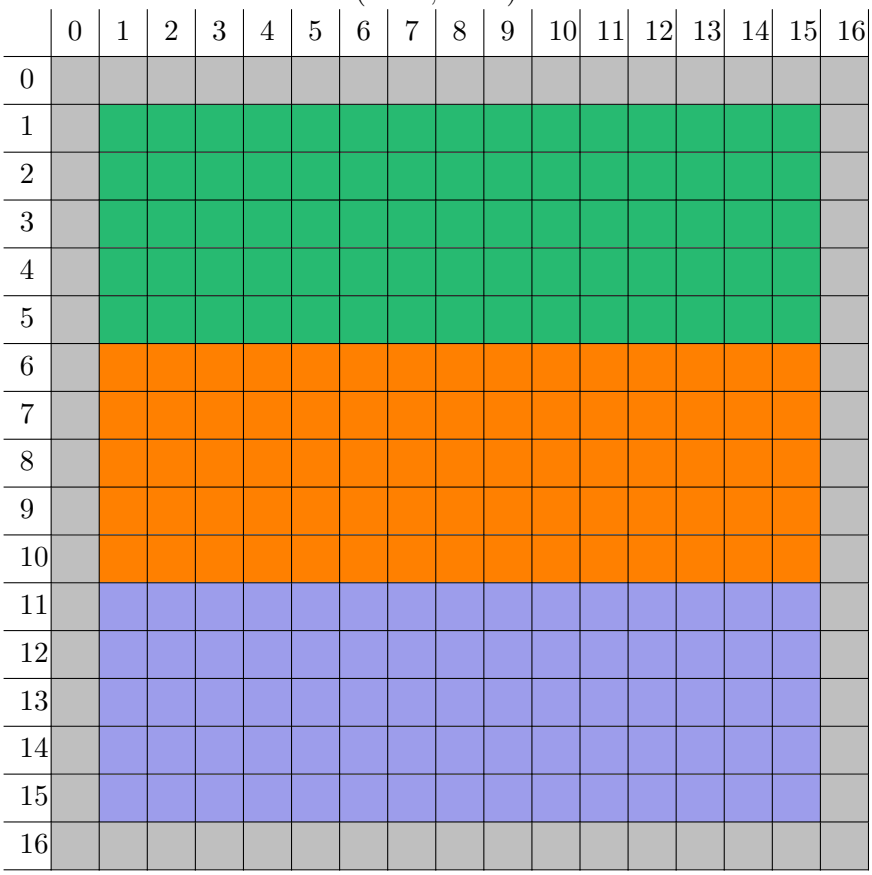
(i, t)	k	$\lfloor \frac{k}{t} \rfloor$	$k \mod t$	j	$c(k, t, j)$	$n_j(k, t)$ - $n_{j+1}(k, t)$
$(0, 1)$	7	7	0	0	7	0 - 7
$(0, 3)$	7	2	1	0	3	0 - 3
				1	2	3 - 5
				2	2	5 - 7
$(0, 12)$	7	0	7	0	1	0 - 1
				1	1	1 - 2
				2	1	2 - 3
				3	1	3 - 4
				4	1	4 - 5
				5	1	5 - 6
				6	1	6 - 7
				7	1	7 - 7
				8	1	7 - 7
				9	1	7 - 7
				10	1	7 - 7
				11	1	7 - 7
$(1, 3)$	15	5	0	0	5	0 - 5
				1	5	5 - 10
				2	5	10 - 15

3. Grün is thread 1, Orange ist thread 2, Blau is thread 3. Hier wird reihenweises Scheldulingbeispielsweise gezeigt.

$(i = 0, t = 3)$



$(i = 1, t = 3)$



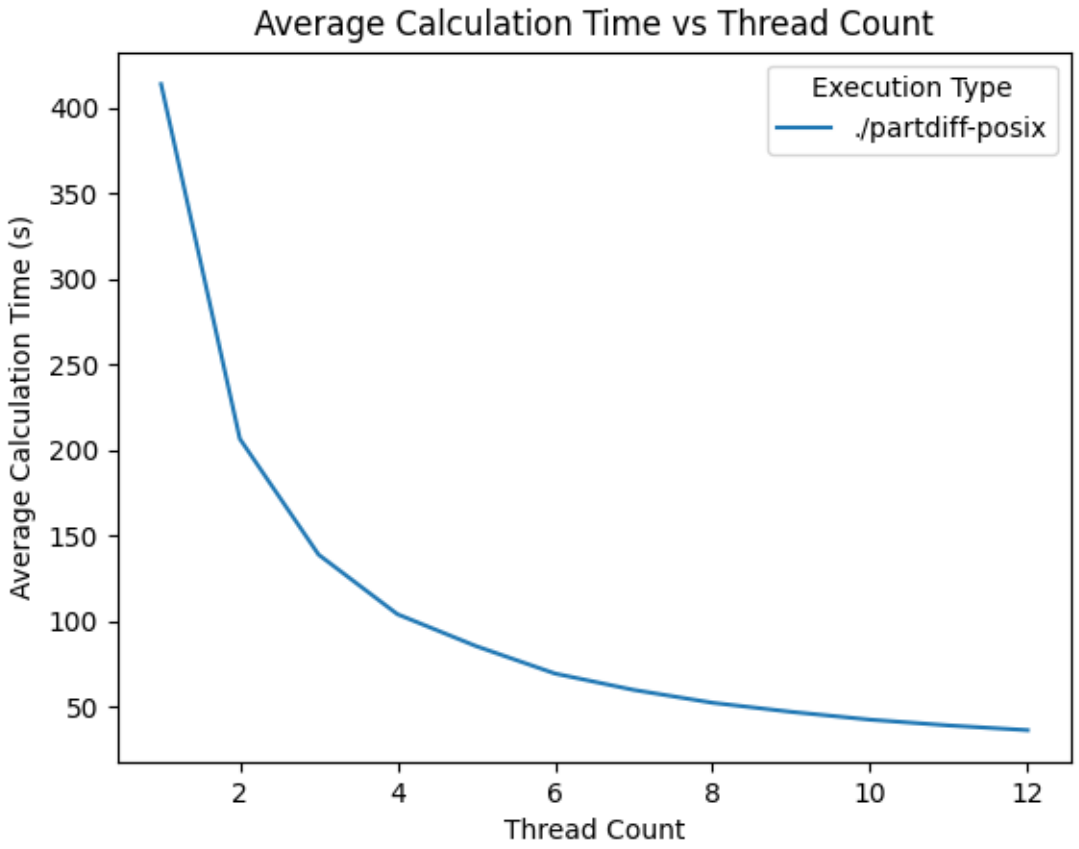
4.
```
interlines = <interlines parameter>
thread_count = <thread count parameter>
N = 8 * interlines + 9 # doesn't match usage in C code,
                       # need to be careful when implementing this
workload = N - 2 # total workload
running = True

def n(k, t, j):
    return j * (h // t) + min(j, k % t)

def thread_run(thread_id: int, arguments, options, shared):
    w_start = n(workload, thread_count, thread_id)
    w_end = n(workload, thread_count, thread_id)
    row_start = w_start + 1 # We skip the first row/column
    row_end = w_end + 1 # exclusive
    column_start = 1
    column_end = N - 1 # exclusive
    while running:
        maxLocalResidiuum = 0.0
        if thread_id == 0:
            # setup shared.m1, shared.m2, shared.maxResidium
        barrier()
        Matrix_Out = arguments.matrix[shared.m1]
        Matrix_In = arguments.matrix[shared.m2]
        for i in range(row_start, row_end):
            # calculate fpisin_i as normal
            for j in range(column_start, column_end):
                star = 0.25 * (Matrix_In[i-1][j] + Matrix_In[i]
                               [j-1] + Matrix_In[i][j+1] + Matrix_In[i+1][j])
                if options.inf_func == FUNC_FPISIN:
                    star += fpisin_i * sin(pih * j)
                Matrix_Out[i][j] = star
                # Calculate residium
                maxLocalResidium = max(maxLocalResidium, residium)
        with critical():
            maxResidium = max(maxLocalResidium, maxResidium)
        if thread_id == 0:
            # check termination conditions
        barrier()


# Do normal setup as required for JACOBI
for thread_id in range(1, thread_count):
    start_thread(thread_run, thread_id, arguments, options, shared)
thread_run(thread_id)
```

Aufgabe 2. Alle Messungen wurden auf Knoten west9 mit interlines=512, `METH_JACOBI`, `FUNC_FPISIN`, `TERM_ITER` und iterations=100 durchgeführt.

Aufruf	Messwerte	Durchschnit
`partdiff-seq`	41.202, 41.208, 41.191	41.200
`partdiff-posix 1`	41.292, 41.256, 41.412	41.320
`partdiff-posix 12`	5.377, 4.967, 4.293	4.866

Der Speedup liegt somit bei ungefähr 8.5. Dies ist ein bisschen schlechter als was bei OpenMP erreicht wurde, vermutlich aufgrund von anderem Scheduling. Wenn man `partdiff-seq` und `partdiff-posix 1` vergleicht sieht man das die rein sequentielle version ein kleines bisschen Schneller ist, was daran liegt das die multi-threading Variante einen Overhead hat selbst wenn keine weiteren threads erzeugt werden. (Das Programm ist so geschrieben das der main thread normal teilnimmt an den Berechnungen)

Aufgabe 3. Die Messungen entsprechen den Erwartungen das die Laufzeit mit steigender Thread Count abnimmt, ungefähr dem $1/x$ Prinzip folgend, sodass z.B. TC=2 nur halb solang braucht wie TC=1. Im vergleich mit dem idealisierten werden erreicht durch $412.976/TC$ sieht man aber einen Overhead von 1-2 Sekunden bei den höheren Thread Counts. Für einen Vergleich zum sequentiellen original Program siehe Aufgabe 2. Wenn man dies mit den OpenMP Werten vergleicht ist die OpenMP Implementierung grundsätzlich ein bisschen schneller. Da OpenMP auch einfacher zu verwenden ist als posix threads wurden wir grundsätzlich OpenMP bevorzugen.



Thread Count	Messwerte	Durchschnitt
1	413.492, 411.834, 416.600	413.976
2	206.480, 206.959, 206.488	206.642
3	139.243, 138.766, 138.771	138.927
4	104.200, 104.120, 104.116	104.145
5	83.419, 89.909, 83.074	85.467
6	69.426, 69.455, 69.487	69.456
7	60.196, 59.695, 59.764	59.885
8	52.280, 52.399, 52.454	52.378
9	46.851, 47.220, 47.111	47.061
10	42.564, 42.188, 42.758	42.503
11	39.256, 39.258, 38.845	39.120
12	36.213, 36.949, 35.964	36.375

Tabelle 1: Messwerte und Durchschnitt nach Thread Count. Alle Messungen wurden auf Knoten west9 mit interlines=512, `METH_JACOBI`, `FUNC_FPISIN`, `TERM_ITER` und iterations=1000 durchgeführt