

Leider ist es schwer Testfragen von Parallel Computing zu bekommen. Hier ist eine Zusammenfassung von Fragen die ich auf Discord, Studo, Vowi etc. gefunden habe. Die Fragen sind zum Üben absichtlich freigelassen worden. Bitte füllt sie aus und postet die Erklärungen zu den ausgefüllten Fragen auf Vowi. Falls Träff versucht die Fragen vom Vowi runter zu bekommen, speichert diese auf euren PCs und verbreitet sie für künftige Generationen, auf verschiedenen Discord-Servern oder anderen Plattformen, weiter. Vergisst nicht richtig anzukreuzen, sonst gibt es Minuspunkte ;)

Question 1: Die Funktion `reduce` des folgenden OpenMP-Programms berechnet die Summe der Elemente eines Feldes. Die Parallelisierung erfolgt mittels OpenMP "tasks".

```

1 double reduce(double a[], int n) {
2     int i;
3     double r1, r2;
4
5     if(n <= 1) return a[0];
6
7     #pragma omp task shared(r1)
8     r1 = reduce(a, n/2);
9
10    #pragma omp task shared(r2)
11    r2 = reduce(a+n/2, n-n/2);
12
13    #pragma omp taskwait
14
15    return r1+r2;
16 }

```

Die `reduce`-Funktion wird in einer `parallel`-Region wie folgt aufgerufen:

```

1 int main() {
2     int n = ... // given
3     double a[n];
4     double result;
5
6     // ... Initialisierung des Feldes
7
8     #pragma omp parallel
9     {
10        // ...
11        #pragma omp single
12        result = reduce(a, n);
13    }
14 }

```

Bitte bewerten Sie folgende Aussagen.

- (a) TRUE FALSE : Die Anzahl der insgesamt erzeugten "tasks" ist abhängig von der Anzahl von Threads.
- (b) TRUE FALSE : Die Anzahl der insgesamt erzeugten "tasks" ist in $O(n)$.
- (c) TRUE FALSE : Der berechnete Wert in Zeile 15 ändert sich nicht, wenn `#pragma omp taskwait` in Zeile 13 entfernt wird (die Zeilenangaben beziehen sich jeweils auf die `reduce`-Funktion).
- (d) TRUE FALSE : Unter der Annahme, dass n Prozessorkerne unbegrenzt vorhanden sind, ist die bestmögliche Ausführungszeit in $O(\log n)$.

Question 2: Bitte Sie folgende Aussagen.

- (a) TRUE FALSE : Die folgende Schleife kann mit einer OpenMP-Schleifen-Direktive parallelisiert werden.

```

int i;
int n = ...; //fixed after initialization

for (i=0; i<n; i+=2){
    if (i%2==0) continue;
    g(i);
}

```

- (b) TRUE FALSE : Die folgende Schleife kann mit einer OpenMP-Schleifen-Direktive parallelisiert werden.

```

int i, j;
int n = ...; //fixed after initialization

j = n;
for (i=0; i<j; i++){
    if (i%3 == 0) j--;
}

```

- (c) TRUE FALSE : Die folgende Schleife kann mit einer OpenMP-Schleifen-Direktive parallelisiert werden.

```

int i;
int n = ...; //fixed after initialization

for (i=0; i<n; i+=2){
    if (i%2==0) break;
    f(i);
}

```

- (d) TRUE FALSE : Die folgende Schleife kann mit einer OpenMP-Schleifen-Direktive parallelisiert werden.

```

int i, j;
int n = ...; //fixed after initialization

for (i=0, j=0; i+j<n; i++,j++){
    h(i,j);
}

```

Question 3: Bitte Sie folgende Aussagen zu OpenMP.

- (a) TRUE FALSE : Ein OpenMP-Programm ist eine Folge von parallelisierten Regionen und kann auch sequenzielle Stellen beinhalten.
- (b) TRUE FALSE : Die Anzahl der Threads, die in einer parallelen Region arbeiten sollen, wird vom Compiler festgelegt und kann nicht geändert werden.
- (c) TRUE FALSE : Die Summe der Laufzeiten aufeinanderfolgender parallelisierter Regionen bildet eine untere Schranke für die Laufzeit eines parallelen Programmes.
- (d) TRUE FALSE : Die Anzahl der aufeinanderfolgenden parallelen Regionen (`#pragma omp parallel`) eines OpenMP-Programmes muss unabhängig vom Programmablauf sein (statisch festgelegt), damit der Compiler die Arbeit auf die Threads verteilen kann.
- (e) TRUE FALSE : OpenMP hat keine und benötigt keine Bibliotheksfunktionen.
- (f) TRUE FALSE : OpenMP ist ein Programmiermodell das auf Threads basiert.
- (g) TRUE FALSE : OpenMP kann in C++ Programmen verwendet werden.
- (h) TRUE FALSE : OpenMP ist eine Erweiterung der Programmiersprache Java.

Question 4: Bewerten Sie die folgenden Aussagen.

- (a) TRUE FALSE : "Array Compaction" ist eine typische Anwendung von Präfix-Summen zum Zweck der Lastverteilung.
- (b) TRUE FALSE : Für die Berechnung von Präfix-Summen mit Eingaben der Länge n ist die Arbeit in $\Omega(n \log n)$ erforderlich.
- (c) TRUE FALSE : Im Partitionierungsschritt des Quicksort Algorithmus können alle Elemente aus dem zu sortierenden Feld, die größer als das ausgewählte Pivot-Element sind, in $O(n)$ Operationen in $O(\log n)$ Zeitschritten in die obere Hälfte des zu sortierenden Feldes kopiert werden.
- (d) TRUE FALSE : Präfix-Summen können auf einer EREW PRAM in $O(\log n)$ Zeitschritten berechnet werden.

Question 5: Bewerten Sie folgende Aussagen zu den folgenden Programmteilen. Das Programm wird in einer Umgebung mit 8 Threads ausgeführt.

- (a) TRUE FALSE : Die Variable a hat nach Ausführung des Programmstücks immer den Wert 2.

```
int a=0;
#pragma omp parallel
{
    #pragma omp single
    a++;
    #pragma omp single
    a++;
}
```

- (b) TRUE FALSE : Die Variable a hat nach Ausführung des Programmstücks immer den Wert 2.

```
int a=0;
#pragma omp parallel
{
    #pragma omp master
    a++;
    #pragma omp single
    a++;
}
```

- (c) TRUE FALSE : Die Variable a hat nach Ausführung des Programmstücks immer den Wert 2.

```
int a=0;
#pragma omp parallel
{
    int t = omp_get_thread_num();

    if (t >= 6) {
        a++;
    }
}
```

- (d) TRUE FALSE : Die Variable a hat nach Ausführung des Programmstücks immer den Wert 8.

```
int a=0;
#pragma omp parallel
{
    #pragma omp critical
    a++;
}
```

Question 6: Ein OpenMP-Programmfragment erzeugt Aufgaben ("Tasks") wie folgt.

```

1 void f()
2 {
3     // something
4 }
5
6 void g()
7 {
8     // something else
9 }
10
11 ...
12
13 #pragma omp parallel
14 {
15     #pragma omp task
16     f();
17     #pragma omp task
18     g();
19 }

```

Das Programm wird in einer Umgebung mit 8 Threads ausgeführt.
Bewerten Sie die folgenden Aussagen.

- (a) TRUE FALSE : Genau 4 Tasks werden erzeugt.
 (b) TRUE FALSE : Je nach Programmablauf werden höchstens 2 Tasks erzeugt.
 (c) TRUE FALSE : Die Funktionen $f()$ und $g()$ dürfen keine weiteren Direktiven der Form

`#pragma omp task`

enthalten, damit das Programm korrekt kompiliert.

- (d) TRUE FALSE : Die Anzahl der erzeugten Tasks ist unabhängig von der Anzahl der gestarteten Threads.

Question 7: Es sei folgendes OpenMP-Program gegeben:

```

1 int i;
2 int n = 20;
3 int a[n];
4
5 #pragma omp parallel for schedule(runtime)
6 for (i=0; i<n; i++) {
7     a[i] = omp_get_thread_num();
8 }
9
10 for(i=0; i<n; i++) {
11     printf("%d ", a[i]);
12 }
13 printf("\n");

```

Bewerten Sie die folgenden Aussagen:

- (a) TRUE FALSE : Setzt man `OMP_SCHEDULE=dynamic,2` und `OMP_NUM_THREADS=4` kann das Programm folgendes herausgeben:

2 2 2 2 2 0 0 0 0 3 3 3 1 1 0 0 3 3 2 2

- (b) TRUE FALSE : Setzt man `OMP_SCHEDULE=dynamic,4` und `OMP_NUM_THREADS=5` kann das Programm folgendes herausgeben:

0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3 2 2 2 2

- (c) TRUE FALSE : Setzt man `OMP_SCHEDULE=static,1` und `OMP_NUM_THREADS=2` kann das Programm folgendes herausgeben:

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

- (d) TRUE FALSE : Setzt man `OMP_SCHEDULE=static,2` und `OMP_NUM_THREADS=2` kann das Programm folgendes herausgeben:

0 0 1 1 2 2 0 0 1 1 2 2 0 0 1 1 2 2 0 0

- (e) TRUE FALSE : Setzt man `OMP_SCHEDULE=dynamic,1` und `OMP_NUM_THREADS=6` kann das Programm folgendes herausgeben:

0 1 2 3 4 5 0 1 2 3 4 5 0 1 2 3 4 5 0 1

- (f) TRUE FALSE : Setzt man `OMP_SCHEDULE=dynamic,2` und `OMP_NUM_THREADS=4` kann das Programm folgendes herausgeben:

0 0 1 1 2 2 0 0 1 1 2 2 0 0 1 1 2 2 0 0

Question 8: Das folgende OpenMP-Programm soll die Summe zweier Vektoren a und b mit je n Elementen und deren Skalarprodukt berechnen.

```

1 int n = ...; // gegeben
2 int a[n], b[n], c[n];
3 double a;
4 int i;
5
6 // ... initialisierung des Programms
7
8 d = 5;
9 #pragma omp parallel for reduction(+:d)
10 for(i=0; i<n; i++){
11     d += a[i] + b[i];
12     c[i] = a[i] + b[i];
13 }
```

Bewerten Sie die Gültigkeit der folgenden Aussagen:

- (a) TRUE FALSE : Um die Korrektheit der Berechnungen zu gewährleisten, muss die Addition der Elemente von a und b (Zeile 12) in einer eigenen Schleife erfolgen (wegen der `reduction(+:d)`-Klausel).
- (b) TRUE FALSE : Für die korrekte Berechnung von d muss zusätzlich `reduction(*:d)` zur Zeile 9 hinzugefügt werden.
- (c) TRUE FALSE : Mit einer zusätzlichen `schedule(static,1)`-Klausel in Zeile 9 kann unter Umständen "false sharing" auftreten.
- (d) TRUE FALSE : Das Hinzufügen von `schedule(dynamic,2)` zur Zeile 9 kann zu inkorrekten Resultaten führen.

Question 9: Das folgende OpenMP-Programm soll den maximalen Wert im Feld a (der Länge n) und die Anzahl (count) der Vorkommen dieses Maximums ermitteln.

```

1 int n = ... // gegeben
2 int a[n];
3
4 // ... Initialisierung des Feldes
5 int max = a[0];
6 int count = 0;
7 int i;
8
9 #pragma omp parallel private(count)
10 {
11     #pragma omp for
12     for(i=1; i<n; i++) {
13         if(a[i] > max) {
14             #pragma omp critical
15                 max = a[i];
16         }
17     }
18
19     #pragma omp for
20     for(i=0; i<n; i++) {
21         if(a[i] == max) {
22             count++;
23         }
24     }
25 }
26 printf("max=%d, count=%d\n", max, count);

```

Bewerten Sie folgende Aussagen.

- (a) TRUE FALSE : Wegen der Verwendung von `critical` in Zeile 14 ist die Laufzeit des Programms immer in $\Omega(n)$.
- (b) TRUE FALSE : Das gegebene Programm enthält einen Fehler bei der Aktualisierung von `count`.
- (c) TRUE FALSE : `count` wird korrekt berechnet.
- (d) TRUE FALSE : `max` wird korrekt berechnet.

Question 10: Das folgende PRAM-Programm arbeitet mit einer vorgegebenen Anzahl Prozessoren p auf drei Vektoren a , b und c mit jeweils n Elementen.

```

1 int n = ...; // gegeben
2 int a[n], b[n], c[n];
3 // a, b, c werden initialisiert
4 par (0<=i<p) {
5     int k;
6     for (k=1; k<n; k+=p) a[k] = b[k]+c[k];
7 }

```

Bewerten Sie folgende Aussagen.

- (a) TRUE FALSE : Der relative Speed-up des Programms ist mit $p \leq n$ in $O(p)$.
- (b) TRUE FALSE : Die Laufzeit des Programms mit p Prozessoren ist $O(p)$.
- (c) TRUE FALSE : Die Arbeit des Programmes ist $O(n)$.
- (d) TRUE FALSE : Das Programm kann auf einer EREW PRAM Maschine ausgeführt werden ($p \geq 1, n \geq 1$).

Question 11: Bitte bewerten Sie die Richtigkeit folgender Aussagen zu OpenMP.

- (a) TRUE FALSE : Beim Start eines OpenMP-Programmes werden so viele Threads gestartet, wie der Compiler im Übersetzungsschritt festgelegt hat.
- (b) TRUE FALSE : Ein OpenMP-Programm kann mehr Threads starten als Prozessorkerne ("processor-cores") verfügbar sind.
- (c) TRUE FALSE : Der OpenMP-Standard ermöglicht die explizite Anwendung von Locks.
- (d) TRUE FALSE : OpenMP bietet Funktionalitäten, um "Mutual Exclusion" (wechselseitigen Ausschluss) zu gewährleisten.
- (e) TRUE FALSE : "Tasks" in OpenMP werden immer von dem Thread ausgeführt, der sie erzeugt hat.
- (f) TRUE FALSE : OpenMP enthält u.a. Compiler-Direktiven (Pragmas) zur Parallelisierung von Schleifen.
- (g) TRUE FALSE : OpenMP enthält u.a. spezielle Bibliotheken, um über das Netzwerk ("communication network") zu kommunizieren.
- (h) TRUE FALSE : OpenMP stellt keine Bibliotheksfunktionen zur Verfügung, d.h. die Funktionalität wird alleine von den Compiler-Direktiven bestimmt.

Question 12: Es sei folgendes OpenMP-Programm gegeben.

```

1 int n = 5;
2 int b[n], x, i;
3
4 x = 0;
5 #pragma omp parallel for reduction(incscan,+:x)
6 for (i=0; i<n; i++) {
7     x += i;
8     #pragma omp scan inclusive(x)
9     b[i] = x;
10 }
11 printf("x=%d\n",x);
12 for (i=0; i<n; i++) printf("b[%d]=%d\n",i,b[i]);
13 printf("\n");

```

Bewerten Sie folgende Aussagen.

- (a) TRUE FALSE : Das Programm gibt (neben x) folgende folgende Werte für das Array b aus:

```

b[0]=0
b[1]=1
b[2]=3
b[3]=6
b[4]=10

```

- (b) TRUE FALSE : Der am Ende des Programms ausgegebene Wert von x ist 10.
- (c) TRUE FALSE : Die inklusive Präfixsumme ("prefix sum") für ein Eingabefeld der Länge n kann auf einer PRAM in $O(n/p + \log p)$ Zeit mit p Prozessoren berechnet werden.

Question 13: Dass folgende OpenMP-Programm soll das Produkt einer Matrix a der Größe $m \times n$ und eines Vektors b der Länge n berechnen.

```

1 int m = ... ; // given
2 int n = ... ; // given
3 double a[m][n];
4 double b[n];
5 double c[m];

```

```

6
7 int i, j;
8
9 #pragma omp parallel for private(j)
10 for (i=0; i<m; i++) {
11     double r = 0;
12     for (j=0; j<n; j++) {
13         r += a[i][j]*b[j];
14     }
15     c[i] = r;
16 }

```

Bewerten Sie die Gültigkeit der folgenden Aussagen:

- (a) TRUE FALSE : Fügt man in Zeile 9 collapse(2) hinzu, verbessert sich die Komplexität von $O(\frac{mn}{p})$ auf $O(\frac{m}{p})$.
- (b) TRUE FALSE : Anstatt die äußere Schleife zu parallelisieren, kann man auch nur die innere Schleife (inner loop, Zeile 12) mit einer #pragma omp parallel for-Direktive versehen, was immer (ohne weitere Direktive) zu einem korrekten Ergebnis führt.
- (c) TRUE FALSE : Die äußere Schleife (outer loop, Zeile 9) kann mit collapse(2) annotiert werden, ohne die Korrektheit zu beeinflussen.
- (d) TRUE FALSE : Um eine "Race Condition" in Zeile 13 zu vermeiden, muss die Aktualisierung von r geschützt werden, z.B. durch #pragma omp critical.

Question 14: Es sei folgende OpenMP-Funktion gegeben:

```

1 int compute_result(int a, int b) {
2     #pragma omp parallel
3     {
4         #pragma omp single
5         a++;
6
7         #pragma omp master
8         b++;
9
10        #pragma omp single
11        if(a > b) {
12            a++;
13        }
14    }
15    return a+b;
16 }

```

Bewerten Sie folgende Aussagen.

- (a) TRUE FALSE : Der Codeblock

```

omp_set_num_threads(4);
printf("%d\n", compute_result(0,0));

```

kann die folgende Ausgabe erzeugen:

```
9
```

- (b) TRUE FALSE : Der Codeblock

```

omp_set_num_threads(3);
printf("%d\n", compute_result(0,0));

```


kann die folgende Ausgabe erzeugen:

2

(c) TRUE FALSE : Der Codeblock

```
omp_set_num_threads(3);  
printf("%d\n", compute_result(0,0));
```

kann die folgende Ausgabe erzeugen:

3

(d) TRUE FALSE : Der Codeblock

```
omp_set_num_threads(3);  
printf("%d\n", compute_result(0,0));
```

kann die folgende Ausgabe erzeugen:

7

Question 15: Es sei folgendes OpenMP-Programm gegeben.

```
1 int f(int n, int m) {  
2   int i, j, k;  
3   int r[n][m];  
4  
5   for(k=0; ; k++) {  
6     for(i=1; i<n; i++) {  
7  
8       #pragma omp parallel for  
9       for(...) { // innermost loop  
10        ...  
11      }  
12    }  
13  
14    if (condition) break;  
15  }  
16  return r[0][0];  
17 }
```

Die innerste Schleife nimmt in den folgenden Fragen verschiedene Formen an. Die Semantik der Funktion ändert sich in jedem Beispiel und ist für die Lösung irrelevant. Bewerten Sie, welche dieser Formen korrekt parallelisierbar sind.

(a) TRUE FALSE : Wenn die innerste Schleife in der kanonischen Form ist, können alle drei Schleifen mit

```
#pragma omp parallel for collapse(3)
```

parallelisiert werden.

(b) TRUE FALSE : Die innerste Schleife sei:

```
for(j=0; j>-m; j--) {  
  r[i][m+j-1] = i*n-j;  
}
```

(c) TRUE FALSE : Die innerste Schleife sei:

```
for(j=1; j<m-1; j++) {
    r[i][j] = r[i][j-1];
}
```

- (d) TRUE FALSE : Die innerste Schleife sei:

```
for(j=0; j<m; j++) {
    r[i][j] = r[i-1][j];
}
```

Question 16: Bewerten Sie folgende Aussagen zu parallelen Verfahren für das Verschmelzen ("merging") von zwei geordneten Feldern ("arrays").

- (a) TRUE FALSE : Der Rang eines Elementes x in einem geordneten Feld der Länge n kann mit Binärsuche ermittelt werden.
- (b) TRUE FALSE : Wenn der Rang aller Eingabeelemente berechnet werden muss, kann arbeitsoptimal verschmolzen ("merged") werden.
- (c) TRUE FALSE : Der sequenzielle Ansatz zum Verschmelzen ("merging") beinhaltet Schleifen mit ausreichend viel Arbeit, sodass für Eingaben der Größe $2n$ eine parallele Laufzeit von $O(n/p + \sqrt{n})$ mit p Prozessoren erreicht werden kann.
- (d) TRUE FALSE : Wenn der Rang eines jeden Elementes in den Eingabefeldern der Größe n berechnet worden ist, kann ein korrekt verschmolzenes ("merged") Ausgabefeld in $O(n/p)$ Zeitschritten erstellt werden, wenn p Prozessoren vorhanden sind.

Question 18: Die Zuordnung von Schleifen-Iterationen zu den einzelnen Threads wird in OpenMP durch "Schedules" bestimmt. Beantworten Sie für die folgenden Schleifen, welcher Thread was getan hat.

Nehmen Sie an, dass insgesamt 6 Threads aktiv sind und dies nicht geändert wird.

Die Variable n ist auf 17 gesetzt und bleibt unverändert.

- (a) TRUE FALSE : Die Schedules

```
#pragma omp for schedule(dynamic,1)
```

und

```
#pragma omp for schedule(guided,1)
```

führen immer zu der gleichen Zuordnung von Iterationen auf Threads.

- (b) TRUE FALSE : Iteration $i = 16$ wird von Thread 2 ausgeführt.

```
#pragma omp for schedule(static,6)
for (i=0; i<n; i++) {
    ...
}
```

- (c) TRUE FALSE : Iteration $i = 16$ wird von Thread 0 ausgeführt.

```
#pragma omp for schedule(dynamic,3)
for (i=0; i<n; i++) {
    ...
}
```

- (d) TRUE FALSE : Iteration $i = 16$ wird von Thread 4 ausgeführt.

```
#pragma omp for schedule(static,1)
for (i=0; i<n; i++) {
    ...
}
```

Question 19: Bewerten Sie folgende Aussagen.

- (a) TRUE FALSE : Mit dem folgenden Programmfragment wird in a die inklusive Präfix-Summe der Thread IDs für alle vorhandenen Threads korrekt berechnet. Wenn das Ergebnis korrekt berechnet wird, ist der Wert von a pro Thread ID wie folgt:

Thread ID	0	1	2	3	...
a	0	1	3	6	...

```
int a = 0;
int n = omp_get_max_threads();

#pragma omp parallel for schedule(static, 1)
for (i=0; i<n; i++) {

    #pragma omp critical
    a += i;
}
```

- (b) TRUE FALSE : Mit einer Reduktion in einer parallelen Region kann für jeden Thread die exklusive Präfix-Summe der Thread IDs der vorhergehenden Threads berechnet werden, vorausgesetzt die Reduktionsvariable ist als "firstprivate" deklariert. Wenn das Ergebnis korrekt berechnet wird, ist der Wert von a pro Thread ID wie folgt:

Thread ID	0	1	2	3	...
a	0	0	1	3	...

```
int a = 0;

#pragma omp parallel firstprivate(a), reduction(+:a)
{
    a += omp_get_thread_num();
    #pragma omp barrier
}
```

- (c) TRUE FALSE : Die folgende Schleife mit einer OpenMP-Reduktion berechnet für $n = 10$ in a den Wert 55.

```
int a = 0;

#pragma omp parallel for reduction(+:a)
for (i=0; i<n; i++) a = a+1+i;
```

- (d) TRUE FALSE : Gegeben sind zwei parallelisierte Schleifen, die das gleiche Ergebnis berechnen. Schleife 1 wird immer schneller ausgeführt als Schleife 2 (auf dem gleichen System, unter gleichen Bedingungen).
Schleife 1:

```
#pragma omp parallel
for (i=0; i<n; i++) {
    int b;
    #pragma omp atomic capture
    b = a++;
    // something with b ...
}
```

Schleife 2:

```
#pragma omp parallel
for (i=0; i<n; i++) {
    int b;
    #pragma omp critical
    b = a++;
    // something with b ...
}
```