Richtig	Falsch		

✗	○	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);`
		`}`
			Code macht effektiv nichts, da (i%2 == 0) immer true ist.

○	✗	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--;`
		`}`
			Nein, da mehrere Threads "gleichzeitig" j dekrementieren. Weiters wird durch die Schleifen-Direktive i nicht "normal" inkrementiert wodurch die reihenfolge des Schleifeninhalts ohne critical-section anders sein kann.

○	✗	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==17) break;`
		`  f(i);`
		`}`
			"break" kann nicht in OpenMP for-Schleifen verwendet werden

●	✗	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);`
		`}`
			Es kann nur 1 variable in einem OpenMP for-Schleifen Kopf verwendet werden. Theoreitsch kann der Code umgeschreiben werden um i&j zu unterstützen, jedoch muss in der loop-condition nur die definierte variable der for-schleife verwendet werden

```
void par() {
#pragma omp parallel default(none)
    {
        int i,j;
        int n = 9;
        int k;
#pragma omp for private(i, j)
        for (k = 0; j + i < n; k++) {
            i = k;
            j = k;
            printf("Thread %d
        }
    }
```

Condition of OpenMP for loop must be a relational comparison ('<', '<=', '>', '>=', or '!=') of loop variable 'k'

int n = 9

Bewerten Sie folgende allgemeine Aussagen zu OpenMP.

Richtig	Falsch		
✗	○	Ein OpenMP-Programm ist eine Folge von parallelisierten Regionen und kann auch sequenzielle Stellen beinhalten.	
○	✗	Die Anzahl der Threads, die in einer parallelen Region arbeiten sollen, wird vom Compiler festgelegt und kann nicht geändert werden.	Die Anzahl der Threads kann dynmaisch mit omp_set_num_threads(num_threads) festgelegt werden
✗	○	Die Summe der Laufzeiten aufeinanderfolgender parallelisierter Regionen bildet eine untere Schranke für die Laufzeit eines parallelen Programmes.	
○	✗	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.	Nicht sicher, testen mit rand()-Werten liefert unkonstante Ergebnisse

Bewerten Sie die folgenden Aussagen.

Richtig	Falsch	
X	○	"Array Compaction" ist eine typische Anwendung von Präfix-Summen zum Zweck der Lastverteilung.
○	X	Für die Berechnung von Präfix-Summen mit Eingaben der Länge n ist Arbeit in $\Omega(n \log n)$ erforderlich.
○	X	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 und in $O(\log n)$ Zeitschritten in die obere Hälfte des zu sortierenden Feldes kopiert werden.
○	X	Präfix-Summen können auf einer EREW PRAM in $O(\log n)$ Zeitschritten berechnet werden.

Remark:
In most reasonable architecture models, Ω(log n) would be a lower bound on the parallel running time for a work-optimal solution

```
Nein, da für den Partitionierungsschritt die prefix-sums berechnet
werden und das benötigt O(n/p + log n) Time Steps
```

Theorem:
The (inclusive/exclusive) prefix-sums problem for an array of n elements with an associative binary operator “+” can be solved on an EREW PRAM with p processors in O(n/p+log p) time steps.

Theorem:
The (inclusive/exclusive) prefix-sums problem for an array of n elements with an associative binary operator “+” can be solved on an EREW PRAM with p processors in O(n/p+log n) time steps.

```
Ich gehe aufgrund der letzten 2 Fragen davon aus, dass die Anzhal der Threads/Prozessesoren p = 8
```

Richtig	Falsch	

X	○	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++;`
		`}`

◉	X	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++;`
		`}`

○	X	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++;`
		`  }`
		`}`

X	○	Die Variable a hat nach Ausführung des Programmstücks immer den Wert 8.
		`int a = 0;`
		`#pragma omp parallel`
		`{`
		`  #pragma omp critical`
		`  a++;`
		`}`

```
#pragma omp master: Nur der Master Thread kann den code block ausführen. Hat KEINE
implizierte barrier.

Durch den fehlenden barrier kann theoretisch ein 2. Thread den unteren Teil ausführen,
während der master Thread den oberen übernimmt.

Dadurch kann es vorkommen, dass beide Threads "gleichezeitig" in a hineinschreiben
```

```
omp_get_thread_num() liefert die Thread-ID des momentan aktiven Threads.

Da a++ nicht innerhalb einer critical section gemacht wird, kann es vorkommen, dass
2 Threads "gleichzeitig" in a hineinschreiben, wodurch inkrementelle erhöhungen
verloren gehen können
```

```
Angenommen p = 8:

a++ wird in der critical section ausgeführt, wodruch es nicht vorkommen kann, dass 2
Threads gleichzeitig den Wert überschreiben
```

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

```
void f()
{
  // something
}

void g()
{
  // something else
}

...

#pragma omp parallel
{
  #pragma omp task
  f();
  #pragma omp task
  g();
}
```

#pragma omp task: gibt die spezifizierten Blöcke in einer art Queue. Tasks werden asynchron von momentan verfügbaren Threads abgearbeitet

Das Programm wird in einer Umgebung mit 2 Threads ausgeführt.

Bewerten Sie die folgenden Aussagen.

Richtig	Falsch	
✗	●	Genau 4 Tasks werden erzeugt.
●	✗	Je nach Programmablauf werden höchstens 2 Tasks erzeugt.
●	✗	Die Funktionen $f()$ und $g()$ dürfen keine weiteren Direktiven der Form `#pragma omp task` enthalten, damit das Programm korrekt kompiliert.
○	✗	Die Anzahl der erzeugten Tasks ist unabhängig von der Anzahl der gestarteten Threads.

Wird in den Folien bei QuickSort rekursiv verwendet

```
void QuickSort(int x[],int n) {
  if (n<=1) return;

  pivot = choosepivot(x,n);
  ix = partition(x,n,pivot);
#pragma omp task shared(x)
  QuickSort(x,ix);
#pragma omp task shared(x)
  QuickSort(x+ix+1,n-1-ix);
//#taskwait
}
```

Mark independent `QuickSort` calls as tasks. Spawned at runtime

Wait for completion of immediately spawned tasks (not needed here)

Bewerten Sie folgende allgemeine Aussagen zu OpenMP.

Richtig	Falsch	
●	✗	OpenMP hat keine und benötigt keine Bibliotheksfunktionen.
✗	○	OpenMP ist ein Programmiermodell das auf Threads basiert.
✗	○	OpenMP kann in C++ Programmen verwendet werden.
●	✗	OpenMP ist eine Erweiterung der Programmiersprache Java.

Es sei folgendes OpenMP-Programmfragment gegeben.

Verbleibende Zeit 0:15:16

```
int f(int n, int m) {
  int i,j,k;
  int r[n][m];

  for(k=0; ; k++) {
    for(i=1; i<n; i++) {

      #pragma omp parallel for
      for(...) { // innermost loop
        ...
      }
    }

    if (condition) break;
  }
  return r[0][0];
}
```

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.

Richtig	Falsch		
○	✗	Wenn die innerste Schleife in der kanonischen Form ist, können alle drei Schleifen mit `#pragma omp parallel for collapse(3)` parallelisiert werden.	Alle Schleifen des collapse directive müssen in kanonischer Form sein. Zusätzlich ist in der äußersten Schleife ein break, was bei parallelisierung auch nicht erlaubt ist
✗	●	Die innerste Schleife sei: `for(j=0; j>-m; j--) { r[i][m+j-1] = i*n-j; }`	Schleife in kanonischer Form. (Trotzdem nicht 100% sicher)
○	✗	Die innerste Schleife sei: `for(j=1; j<m-1; j++) { r[i][j] = r[i][j-1]; }`	Hier können theoretisch 2 Threads gleichzeitig auf des selbe array-element lesend/schreibend zugreifen
✗	○	Die innerste Schleife sei: `for(j=0; j<m; j++) { r[i][j] = r[i-1][j]; }`	Schleife in kanonischer form. "i" verändert sich im laufe der inneren Schleife nicht

Canonical form for parallel loops

`i`: iteration variable
`i0`: lower bound
`n`: upper bound
`inc`: increment

```
for (i = i0; i<n | i<=n | i>=n | i>n ; i++ | ++i | i- | --i | i+=inc | i-=inc | i=i+inc | i=i-inc ) { <body> }
```

- No `break`, `goto` out of/into loop body; `continue` allowed
- Lower, upper, increment expressions must not change during loop iterations, must be known before loop starts, and same for all threads

Canonical form allows static scheduling: Assignment of loop iterations to threads

OpenMP can parallelize nested loops, observing some constraints

```
#pragma omp parallel for collapse(2)
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    x[i][j] = f(i,j); // thread safe
                      // but time depends on i, j
  }
}
```

Any number of loops

Two loops are handled as one with iteration space of n*m iterations, proper schedule perhaps easier to find

Condition: Both loops must be in canonical form, both iteration spaces must be known prior to the outer loop (e.g., m cannot depend on n)

Frage 6

Bisher nicht beantwortet

Erreichbare Punkte: 4,00

Frage markieren

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

Richtig	Falsch		
✗	○	Der Rang eines Elementes x in einem geordneten Feld der Länge n kann mit Binärsuche ermittelt werden.	
✗	○	Wenn der Rang aller Eingabeelemente berechnet werden muss, kann arbeitsoptimal verschmolzen ("merged") werden.	
○	✗	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.	
✗	○	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.	Rang system wird erstellt, damit alle Threads unabhängig von einander den mergeprozess absolvieren können. Heißt wenn n = p dann ist tatsächliche merge-prozess in O(1)

Observation:
For an ordered sequence stored in an array A, rank(x,A) can be computed sequentially by binary search. The number of operations per x (work, time) is O(log n) for input n-element array A

Theorem:
On a shared-memory system, two ordered sequences of size n and m can be merged in time O((n+m)/p+log n)

Frage 7

Bisher nicht beantwortet

Erreichbare Punkte: 4,00

Frage markieren

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.

Richtig	Falsch		

○	✗	Die Schedules
		`#pragma omp parallel for schedule(dynamic,1)`
		und
		`#pragma omp parallel for schedule(guided,1)`
		führen immer zu der gleichen Zuordnung von Iterationen auf Threads.
			`Guided vergiebt die chunks dynmanisch, und bestimmt die chunk-size unter einschränkungen des paramaters <chunksize> selbst`

✗	○	Iteration $i = 16$ wird von Thread 2 ausgeführt.
		`#pragma omp parallel for schedule(static,6)`
		`for (i=0; i<n; i++) {`
		`  ...`
		`}`
			`Thread 0: 0-5`
			`Thread 1: 5-10`
			`Thread 2: 11-16`

○	✗	Iteration $i = 16$ wird immer von Thread 0 ausgeführt.
		`#pragma omp parallel for schedule(dynamic,3)`
		`for (i=0; i<n; i++) {`
		`  ...`
		`}`
			`Kann man bei dynmaic nicht wissen`

✗	○	Iteration $i = 16$ wird von Thread 4 ausgeführt.
		`#pragma omp parallel for schedule(static,1)`
		`for (i=0; i<n; i++) {`
		`  ...`
		`}`
			`Thread 0:0,6,12`
			`Thread 1:1,7,13`
			`Thread 2:2,8,14`
			`Thread 3:3,9,15`
			`Thread 4:4,10,16`
			`Thread 5:5,11,17`

Schedule clause, `chunksize` optional

- `schedule(static[,<chunksize>])`: Iterations divided into chunks of size `chunksize` (default approx. n/p), chunks assigned to threads in a round robin fashion
- `schedule(dynamic[,<chunksize>])`: Chunks are distributed to threads as threads become free and request work (default `chunksize` 1)
- `schedule(guided[,<chunksize>])`: As dynamic, but the actual `chunksize` for each thread is the number of unassigned iterations divided by p, but no smaller than `chunksize` (default 1)

Richtig Falsch

Mit dem folgenden Programmfragment wird in a die inklusive Präfix-Summe der Thread IDs für alle vorhergehenden 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;
}
```

Obwohl a in der critical section erhört wird ist die reihenfolge der inkrementierung unbestimmt. So könnte beispielsweise Thread 2 zuerst a += 2 ausführen, wodurch bei Thread 0 a ≥ 2 gegeben ist.

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
}
```

Firstprivate deklaration, kann nicht gleichzeitig eine Reduktion sein.

+reduction: Jeder Thread hat seine eine kopie von a. Am ender der parallel section werden alle thread-spezifischen a-Werte zusammen addiert (+)

Sharing can be controlled at entry to parallel region

- Clause `shared(<list of vars>)`
- Clause `private(<list of vars>)`
- Clause `firstprivate(<list of vars>)`

- Clauses `default(shared)`, `default(none)`

For variables declared as private, a local copy per thread is created. With `private`: not initialized. With `firstprivate` initialized to value in master thread prior to parallel section

With `default(none)`, sharing mode of all global variables must be declared explicitly

Good practice to avoid sharing errors

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;
```

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...
}
```

#pragma omp atomic capture:

atomic read-modify-write operation auf geteilte Variable

atomic ist nur schneller, wenn es vom prozessor performancetechnisch unterstützt wird. Sonst ist atomic gleich schnell