Començat el	diumenge, 28 de novembre 2021, 11:11
Estat	Acabat
Completat el	diumenge, 28 de novembre 2021, 11:16
Temps emprat	4 minuts 52 segons
Qualificació	**5,00** sobre 5,00 (**100**%)

Pregunta **1**

Correcte

Puntuació 1,00 sobre 1,00

Assume the following code, using a tree recursive task decomposition to compute variable result from positive values in the vector (including zero):

```
int result=0;
void rec_func (int * vector, int n) {
      if (n == 0) return;
      if (n == 1) {
          if (vector[0] >= 0) result+=compute(vector[0]);
      }  else {
          int n2 = n/2;
          #pragma omp task
          rec_func (vector, n2);
          #pragma omp task
          rec_func (vector + n2, n - n2);
      }
}

void main () {
       ....
       #pragma omp parallel
       #pragma omp single
       hist_pos (vector, n);
}
```

Is there any concurrency problem in this parallel program?

Trieu-ne una:

- ○ No, there is no concurrency problem in this program; however, the programmer should have used a cut-off mechanism to control task granularities and avoid excessively small tasks.
- [✓ Yes, there may be a data race condition in the update of variable result by different threads executing tasks in parallel. This can be solved by using «#pragma omp atomic» just before «result+=compute(vector[0]);», which has less overhead than «#pragma omp critical».] ✓ Correct!
- ○ Yes, there may be a data race condition in the update of variable result by different threads executing tasks in parallel. This race condition can be avoided by adding «depend(inout: result)» in each task construct (remember that task dependences apply between sibling tasks, i.e. between tasks with the same parent task) with a minor impact in performance.

La teva resposta és correcta.

La resposta correcta és: Yes, there may be a data race condition in the update of variable result by different threads executing tasks in parallel. This can be solved by using «#pragma omp atomic» just before «result+=compute(vector[0]);», which has less overhead than «#pragma omp critical».

Pregunta **2**

Correcte

Puntuació 1,00 sobre 1,00

Assume the following code counting the number of positive values in a vector, using a tree recursive task decomposition:

```
int hist_pos (int * vector, int n) {
    int n2 = n/2;
    if (n < 1)
        return (vector[0] >= 0);
    else {
        #pragma omp task «clause-task-1»
        count1 = hist_pos (vector, n2);
        #pragma omp task «clause-task-2»
        count2 = hist_pos (vector + n2, n - n2);
        «statements»
    }
}

int pos = 0;
void main () {
    ...
    #pragma omp parallel
    #pragma omp single
    int result = hist_pos (vector, n);
    ...
}
```

Complete the previous code indicating which of the following statements are true:

Trieu-ne una o més:

- [ ] «clause-task-1» and «clause-task-2» should specify that variables count1 and count2, respectively, are private; otherwise, the parent task and each child task may have a race condition when accessing those variables.
- [x] «clause-task-1» and «clause-task-2» should specify that variables count1 and count2 are shared, so that the parent task can see the values of count1 and count2 generated by each child task. ✓ Correct!
- [x] «statements» should include both «#pragma omp taskwait» (to wait for the two children tasks to finish and have variables count1 and count2 updated), and then «return (count1+count2)». ✓ Correct!
- [ ] «statements» only needs to include «return (count1+count2)» since tasks are executed immediately after creating them.

La teva resposta és correcta.

Les respostes correctes són: «clause-task-1» and «clause-task-2» should specify that variables count1 and count2 are shared, so that the parent task can see the values of count1 and count2 generated by each child task., «statements» should include both «#pragma omp taskwait» (to wait for the two children tasks to finish and have variables count1 and count2 updated), and then «return (count1+count2)».

Pregunta **3**

Correcte

Puntuació 1,00 sobre 1,00

Cut-off is a mechanism to control the number of tasks that are generated, and therefore their granularity. Assume the following incomplete parallel code implementing a leaf recursive task decomposition, introducing a cut-off based on the size of the vector to process.

```
#define MIN_SIZE 32
#define CUTOFF_SIZE 1024
void function_increment(int * vector, int n) {
    if (n<=MIN_SIZE)                        // base case
       for(int i=0; i<n; i++) vector[i]++;
    else {
       int n2=n/2;
       if (<<expression1>>) {
          function_increment(vector, n2);    // n is a power of 2, so ...
          function_increment(vector+n2, n2); // ... both invocations of same size
       } else {
          #pragma omp task
          {
             function_increment(vector, n2);
             function_increment(vector+n2, n2);
```

```
        }
      }
    }
}
```

```
void main() {
    ...
    #pragma omp parallel
    #pragma omp single
    function_increment(vector, n); // n always power of 2
    }
    ...
}
```

Trieu-ne una o més:

- [✓] «MIN_SIZE» is used to control the minimum size of vector for which it is worth to continue with recursion. ✓ Right, this is an algorithmic parameter independent of task generation control.
- [ ] «CUTOFF_SIZE» should be properly used to control the task granularity (size of vector) for which we want to generate tasks. In this case, to implement a leaf recursive task decomposition «expression1» should be «n<CUTOFF_SIZE».
- [✓] «CUTOFF_SIZE» should be properly used to control the task granularity (size of vector) for which we want to generate tasks. In this case, to implement a leaf recursive task decomposition «expression1» should be «n!=CUTOFF_SIZE». ✓ Right, tasks are only generated at one recursion level.
- [ ] «CUTOFF_SIZE» should be properly used to control the task granularity (size of vector) for which we want to generate tasks. In this case, to implement a leaf recursive task decomposition «expression1» should be «n>CUTOFF_SIZE».
- [✓] The base case should also include a conditional statement to check if tasks have already been created: ✓ Correct, if tasks have not yet been generated this is the last point in the recursion tree where you can do it!

```
if (n>=CUTOFF_SIZE)
    #prama omp task
    for(int i=0; i<n; i++) vector[i]++;
else
    for(int i=0; i<n; i++) vector[i]++;
```

For example this would be needed if MIN_SIZE is larger than or equal to CUTOFF_SIZE.

La teva resposta és correcta.

Les respostes correctes són: «MIN_SIZE» is used to control the minimum size of vector for which it is worth to continue with recursion., «CUTOFF_SIZE» should be properly used to control the task granularity (size of vector) for which we want to generate tasks. In this case, to implement a leaf recursive task decomposition «expression1» should be «n!=CUTOFF_SIZE».,
The base case should also include a conditional statement to check if tasks have already been created:

```
if (n>=CUTOFF_SIZE)
    #prama omp task
    for(int i=0; i<n; i++) vector[i]++;
else
    for(int i=0; i<n; i++) vector[i]++;
```

For example this would be needed if MIN_SIZE is larger than or equal to CUTOFF_SIZE.

Pregunta **4**

Correcte

Puntuació 1,00 sobre 1,00

And now assume the following incomplete parallel code implementing a tree recursive task decomposition:, again implementing cut-off based on the size of the vector:

```
#define CUTOFF_SIZE 1024
void function_increment(int * vector, int n) {
    if (n==0) return;
    if (n==1) vector[0]++;
    else {
        int n4=n/4; // asymmetric divide-and-conquer
        if (<<expression1>>)
            #pragma omp task
            function_increment(vector, n4);
        else
            function_increment(vector, n4);
        if (<<expression2>>)
            #pragma omp task
            function_increment(vector+n2, n-n4);
        else
            function_increment(vector+n2, n-n4);
        }
    }
}
void main() {
    ...
    #pragma omp parallel
    #pragma omp single
    function_increment(vector, n);
    }
    ...
}
```

Trieu-ne una o més:

- ☐ Both «expression1» and «expression2» should be «n4>=CUTOFF_SIZE».
- ☐ «expression1» and «expression2» should be «n4>=CUTOFF_SIZE» and «n4<CUTOFF_SIZE», respectively.
- ☐ «expression1» and «expression2» should be «n4>=CUTOFF_SIZE» and «(n-n4)<CUTOFF_SIZE», respectively.
- ☑ «expression1» and «expression2» should be «n4>=CUTOFF_SIZE» and «(n-n4)>=CUTOFF_SIZE», respectively. ✓ Right! Each expression verifies that the size of the vector to process in its recursive invocation is appropriate.

La teva resposta és correcta.

La resposta correcta és:
«expression1» and «expression2» should be «n4>=CUTOFF_SIZE» and «(n-n4)>=CUTOFF_SIZE», respectively.

Pregunta **5**

Correcte

Puntuació 1,00 sobre 1,00

Finally assume the following incomplete parallel code implementing a tree recursive task decomposition making use of the final clause and the omp_in_final intrinsic function:

```
int avg(int * vector, int n);
int max(int a, int b);
int zoo(int * vector, int n);
void foo(int * vector, int n) {
    if (n==2) {
        int tmp = vector[0];
        vector[0]=vector[1];
        vector[1]=tmp;
    }
    else {
        n2 = n/2;
        if(!omp_in_final()) {
            int avg1 = avg(vector, n2);
            int avg2 = avg(vector+n2, n-n2);
            #pragma omp task final(avg1>avg2)
            foo(vector, n2);
            #pragma omp task final(avg1<=avg2)
            foo(vector+n2, n-n2);
        } else {
            zoo(vector, n);
        }
    }
}

void main() {
    #pragma omp parallel
    #pragma omp single
    foo(v, N);
}
```

For N=64 how many times the omp_in_final() intrinsic function will be invoked and will return true and false?

Trieu-ne una:

- ○ Intrinsic omp_in_final() will be invoked 31 times, in 16 of them returning false.
- [✓ Intrinsic omp_in_final() will be invoked 9 times, in 5 of them returning false.] ✓ Great! You got it.
- ○ Intrinsic omp_in_final() will be invoked 8 times, in half of them returning true.
- ○ Intrinsic omp_in_final() will be invoked 30 times, in half of them returning true.

La teva resposta és correcta.

La resposta correcta és: Intrinsic omp_in_final() will be invoked 9 times, in 5 of them returning false.

◄ Questions Unit 4: Iterative decompositions and data sharing/task ordering constraints

Salta a...

Lab1 laboratory assignment ►