Començat el	dimecres, 24 de novembre 2021, 17:25
Estat	Acabat
Completat el	dimecres, 24 de novembre 2021, 17:27
Temps emprat	1 minut 56 segons
Qualificació	**6,00** sobre 6,00 (**100**%)

Pregunta **1**

Correcte

Puntuació 1,00 sobre 1,00

What kind of OpenMP implementation will you use to parallelize a countable loop like the one shown below? (assuming that you cannot modify the sequential version of the code)

```
for (int i = 0; i < n; i++)
    C[i] = A[i] + B[i];
```

Trieu-ne una o més:

- [ ] We can ONLY use the implicit tasks generated in «#pragma omp parallel», assigning to each of them a subset of the iterations with the «omp_get_thread_num» and «omp_get_num_threads» intrinsic functions.
- [x] We can use explicit tasks by inserting «#pragma omp task» in the loop body, one task per iteration of the loop; however it may suffer of too small granularity. The code should be executed within the scope of «#pragma omp parallel» and «#pragma omp single». ✔ Right!
- [x] We can use explicit tasks using «#pragma omp taskloop» with an appropriate task granularity (default or set by using the grainsize or num_tasks clauses). The code should be executed within the scope of «#pragma omp parallel» and «#pragma omp single» ✔ Right!

La resposta és correcta.

Les respostes correctes són:
We can use explicit tasks by inserting «#pragma omp task» in the loop body, one task per iteration of the loop; however it may suffer of too small granularity. The code should be executed within the scope of «#pragma omp parallel» and «#pragma omp single».,

We can use explicit tasks using «#pragma omp taskloop» with an appropriate task granularity (default or set by using the grainsize or num_tasks clauses). The code should be executed within the scope of «#pragma omp parallel» and «#pragma omp single»

Pregunta **2**

Correcte

Puntuació 1,00 sobre 1,00

Which OpenMP constructs will you use to implement an iterative task decomposition for the following code?

```
void traverse_list(List l) {
   Element e;
   for (e = l->first; e; e = e->next)
       process(e);
}
```

```
void main() {
   traverse_list(ll);
}
```

Trieu-ne una o més:

- [ ] Just adding «#pragma omp parallel» in the loop body, enabling in this way the execution of multiple invocations of function process as implicit tasks.
- [ ] We need to add «#pragma omp parallel» and «#pragma omp single» before calling function traverse_list from main and then «#pragma omp taskloop» just in front of the for loop.
- [x] We need to add «#pragma omp parallel» and «#pragma omp single» before calling function traverse_list from main and then «#pragma omp task» just in front of the process function invocation. ✔ Right! You got it!
- [ ] Independently of the option chosen above, we also need to introduce some kind of synchronization to protect the update of `e = e->next`.

La teva resposta és correcta.

La resposta correcta és: We need to add «#pragma omp parallel» and «#pragma omp single» before calling function traverse_list from main and then «#pragma omp task» just in front of the process function invocation.

Pregunta **3**

Correcte

Puntuació 1,00 sobre 1,00

Assume the following four versions of a function (in the scope of a parallel region with single) counting the number of positive values (including zero) in a vector (with a sufficiently large number of elements n), all versions using a linear iterative task decomposition:

```
Code 1)
int it_hist_pos (int *vector, int n) {
    int pos = 0;
    #pragma omp taskloop shared(pos)
    for (int i = 0; i < n; i++)
        if (vector[i] >= 0) {
            #pragma omp critical
            pos++;
        }
    return pos:
}

Code 2)
int it_hist_pos (int *vector, int n) {
    int pos = 0;
    #pragma omp taskloop shared(pos)
    for (int i = 0; i < n; i++)
        if (vector[i] >= 0) {
            #pragma omp critical(positives)
            pos++;
        }
    return pos:
}

Code 3)
int it_hist_pos (int *vector, int n) {
    int pos = 0;
    #pragma omp taskloop shared(pos)
    for (int i = 0; i < n; i++)
        if (vector[i] >= 0) {
            #pragma omp atomic
            pos++;
        }
    return pos:
}

Code 4)
int it_hist_pos (int *vector, int n) {
    int pos = 0;
    #pragma omp taskloop reduction(+: pos)
    for (int i = 0; i < n; i++)
        if (vector[i] >= 0)
```

```
        if (vector[i] >= 0)
            pos++;
    return pos;
}
```

Which of the following statements establishes the appropriate order of the four codes above in terms of performance, from higher to lower:

Trieu-ne una:

- Code 4 > Code 3 > Code 2 >= Code 1, taking into account both the number and kind of synchronisations performed in each case. Between Code 2 and 1 it depends on the existence of other critical regions in the program. ✔ Correct! Reduction only needs to protect the update of the shared variable with the contributions of task local variables at the end of the loop. Atomic makes use of hardware support to protect the update of individual memory positions. The use of named critical could be better than unnamed critical if there are other tasks in the program running in parallel with this function that make use of unnamed critical regions.
- Code 3 > Code 4 > Code 2 > Code 1 since atomic incurs a negligible overhead thanks to hardware support.
- Code 2 > Code 3 > Code 4 > Code 1 since named critical regions allow a better isolation of data races, even better than atomic that only protect the read-update-write of a shared variable.
- Code 1 > Code 2 > Code 3 > Code 4 since unnamed critical has less overhead than named critical. Both kind of critical regions have less restrictions than atomic, which is also used in the implementation of the reduction clause.

La teva resposta és correcta.

La resposta correcta és: Code 4 > Code 3 > Code 2 >= Code 1, taking into account both the number and kind of synchronisations performed in each case. Between Code 2 and 1 it depends on the existence of other critical regions in the program.

Pregunta **4**

Correcte

Puntuació 1,00 sobre 1,00

Assume the following code that inserts the elements of vector ToInsert in a hash table (defined as a vector of linked lists) named HashTable making use of locks to protect the access to it:

```
#define SIZE_TABLE ...
#define MAX_ELEM ...
int ToInsert[MAX_ELEM];

#define SIZE_TABLE 32384
typedef struct {
   int data;
   element * next;
} element;
```

```
element * HashTable[SIZE_TABLE];

omp_lock_t lock_vector[<<size>>];
```

```
int main() {
   ...
   #pragma omp parallel
   #pragma omp single
   for (long i = 0; i < MAX_ELEM; i++) {
      #pragma omp task <<task-clause>>
      {
          int index = hash_function(ToInsert[i], SIZE_TABLE); /* not intensive */
          <<statement-1>>
          insert_elem (ToInsert[i], index, HashTable);  /* Computationally intensive task */
          <<statement-2>>
      }
   }
   ...
}
```

The implementation of function insert_elem does not allow multiple instances of it running in parallel for the same value of index. Which of the following statements are true?

Trieu-ne una o més:

- [x] «size» could be equal to 1. In this case «statement-1» and «statement-2» should simply set and unset, respectively, the lock variable lock_vector[0]. However, this strategy is not exploiting the whole parallelism in the parallel region. ✔ True! This solution does not allow parallel insertions in the hash table.
- [ ] «size» equal to MAX_ELEM. In this case «statement-1» and «statement-2» should simply set and unset, respectively, the lock variable lock_vector[i]; in this way each iteration of the loop can do the insertion of ToInsert[i] without any chance for data races.
- [x] «size» equal to SIZE_TABLE. In this case «statement-1» and «statement-2» should simply set and unset, respectively, the lock variable lock_vector[index]; in this way multiple iterations of the loop will compete to do insertions in the same entry of the hash table. ✔ Right! You got it!
- [x] «task-clause» can be left empty. In this code, by default the compiler will capture the value for all the variables that are needed to execute each iteration of the loop as a task. ✔ Right! variable i has to be captured, which in this case is the default.
- [ ] «task-clause» should be shared(i). By default the compiler will make variable i firstprivate; this is not the appropriate behaviour since each task needs to access to the shared variable ToInsert[i].

La teva resposta és correcta.

Les respostes correctes són: «size» could be equal to 1. In this case «statement-1» and «statement-2» should simply set and unset, respectively, the lock variable lock_vector[0]. However, this strategy is not exploiting the whole parallelism in the parallel region., «size» equal to SIZE_TABLE. In this case «statement-1» and «statement-2» should simply set and unset, respectively, the lock variable lock_vector[index]; in this way multiple iterations of the loop will compete to do insertions in the same entry of the hash table., «task-clause» can be left empty. In this code, by default the compiler will capture the value for all the variables that are needed to execute each iteration of the loop as a task.

Pregunta **5**

Correcte

Puntuació 1,00 sobre 1,00

Assume the following code that inserts the elements of vector ToInsert in a hash table (defined as a vector of linked lists) named HashTable making use of task dependences the access to it:

```
#define SIZE_TABLE ...
#define MAX_ELEM ...
int ToInsert[MAX_ELEM];

#define SIZE_TABLE 32384
typedef struct {
   int data;
   element * next;
} element;
element * HashTable[SIZE_TABLE];
```

```
int main() {
   ...
   #pragma omp parallel
   #pragma omp single
   for (long i = 0; i < MAX_ELEM; i++) {
      int index = hash_function(ToInsert[i], SIZE_TABLE); /* not intensive */
      #pragma omp task <<data-clause>> <<depend-clause>>
      insert_elem (ToInsert[i], index, HashTable);  /* Computationally intensive task */
      }
   }
   ...
}
```

Which of the following statements is true?

Trieu-ne una:

- ( ) The task definition in this code is not correct. A task should include the two statements in the loop body, i.e. the invocations of hash_function and insert_element. With that changed, «depend-clause» should be «depend(inout:index)». «data-clause» could be left empty since the defaults appropriately handle all variables used in the task.
- ( ) The task definition in this code is correct. In order to exploit maximum parallelism, «depend-clause» should be «depend(inout:index)» and «data-clause» could be left empty since the defaults appropriately handle all variables used in the task.

- The task definition in this code is correct. In order to exploit maximum parallelism, «depend-clause» should be «depend(inout:HashTable[index])» and «data-clause» could be left empty since the defaults appropriately handle all variables used in the task. ✔ Correct! You got it!

La teva resposta és correcta.

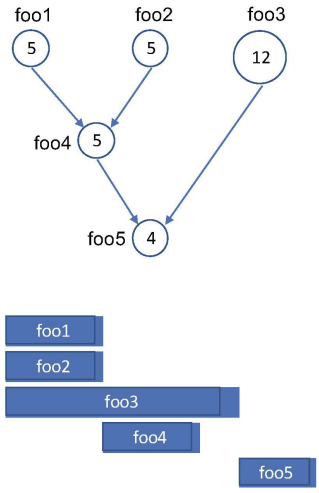
La resposta correcta és: The task definition in this code is correct. In order to exploit maximum parallelism, «depend-clause» should be «depend(inout:HashTable[index])» and «data-clause» could be left empty since the defaults appropriately handle all variables used in the task.

Pregunta **6**

Correcte

Puntuació 1,00 sobre 1,00

Consider the TDG (expressing dependencies between tasks) and execution timeline for the tasks on the left of the following figure:



Code 1

```
#pragma omp task
foo1()
#pragma omp task
foo2()
#pragma omp taskwait
#pragma omp task
foo3()
#pragma omp task
foo4()
#pragma omp taskwait
#pragma omp task
foo5()
```

Code 2

```
#pragma omp task
foo1()
#pragma omp task
foo2()
#pragma omp task
foo3()
#pragma omp taskwait
#pragma omp task
foo4()
#pragma omp taskwait
#pragma omp task
foo5()
```

Code 3

```
#pragma omp task
  {
  #pragma omp task
  foo1()
  #pragma omp task
  foo2()
  #pragma omp taskwait
  #pragma omp task
  foo4()
  #pragma omp taskwait
  }
#pragma omp task
foo3()
#pragma omp taskwait
#pragma omp task
foo5()
```

Code 4

```
#pragma omp task
foo3()
#pragma omp taskgroup
  {
  #pragma omp task
  foo1()
  #pragma omp task
  foo2()
  }
#pragma omp task
foo4()
#pragma omp taskwait
#pragma omp task
foo5()
```

Which of the four code above would achieve the parallelism that is available in the TDG?

Trieu-ne una o més:

- [ ] None of the codes above achieves the potential parallelism that ia available in the TDG.
- [ ] Code 1
- [ ] Code 2
- [x] Code 3 ✔ Correct!
- [x] Code 4 ✔ Correct!

La teva resposta és correcta.

Les respostes correctes són: Code 3, Code 4

◄ Slides for Unit 4: Task decomposition

Salta a...