PC - Processus légers et boucle parallèle
- English notes

June 2013
for (int i = 0; i < n; i++)
    task(i);
A set of tasks are independent iff for $i \neq j$
there is not $task(i)$ that writes to an identical
memory address as $task(j)$
Q2

• How can we execute in parallel the following problems?
  – adding two vectors
  – scalar product of two vectors
  – vector matrix multiplication
  – Polynomial product (assume the polynomials of the coefficients co-efficients)
  – evaluating a polynomial based on the formula
    \[ a_0 + x(a_1 + x(a_2 + (\cdots (a_{n-1} + x_a) \cdots)). \]
Vector sum

var vector1
var vector2
var sum;
//consider the following method
//that ads vectors from index start to index end
addV(v1,v2,from,end)
//lets assume that from and end are within the
//two vectors limits and that the
//two vectors have equal length ..its easy to
generalize
//for different lengths
for(i=from;i<end;i++)
    sum[i]=v1[i]+v2[i]
end

//main program
//assume two threads (or as many as you
want just have to split the vector accordingly)
//could be done in recursion as well
main
    thread(addV(v1,v2,0,v1.length/2));
    thread(addV(v1,v2,v1.length/2,v1.length));
end
Vector scalar product

for the scalar product of two vectors we have
\[ v_1 \cdot v_2 = \sum (v_1(i) \cdot v_2(i)) \]

so we have:

```javascript
var vector1;
var vector2;
var sumV;
//consider the following method that multiplies
//vectors from index start to to index end
prod(v1,v2,from,end)
//lets assume that from and end are within the two
vectors limits and that the
//two vectors have equal length ..
    for(i=from;i<end;i++)
        sumV[i]=v1[i]*v2[i]
end
end
```

//main program
//assume two threads (or as many as you want just have to split the vector accordingly)
//could be done in recursion as well
main
    thread1(prod(v1,v2,0,v1.length/2));
    thread(prod(v1,v2,v1.length/2,v1.length));
return sum(sumV)
end
matrix * vector multiplication

The multiplication of a matrix by a vector:

\[
\begin{pmatrix}
A & B & c \\
D & E & F \\
G & H & I \\
\end{pmatrix}
\begin{pmatrix}
P \\
Q \\
R \\
\end{pmatrix}
= \begin{pmatrix}
AP+BQ+CR \\
DP+EQ+FR \\
GP+HQ+IR \\
\end{pmatrix}
\]

//assume the dimensions match

// it is a scalar product of each line so now we can
// either do a thread for each product or
// we can serialize the process for each line and use
// the previous function

// OR we can do both:

```javascript
var matrix, vector, result;

//consider the following method
//that multiplies vectors from index start to to index end

prod(v1, v2, from, end) //line * vector

var sumV[end-from]; //vector

//lets assume that from and end are within the two
vectors limits and that the
//two vectors have equal length ..its easy to
generalize for diferent lengths

for (i = from; i < end; i++)
    sumV[i-from] = v1[i] * v2[i]

end

return sumV
end
```
matrix * vector multiplication

//this is now a method of scalar product
//assume two threads (or as many as you want just have to split the vector accordingly)

scalar(v1,v2)
var sumV1 // half size vector to store the intermediate result
var sumV2
thread(sumV1=prod(v1,v2,0,v1.length/2));
thread(sumV2=prod(v1,v2,v1.length/2,v1.length));
// wait until all threads have finished!!!
return (sum(sumV1) + sum(sumv2));
end

main program
for(i=0;i<matrix.rows;i++)
    thread(result(i)=scalar(matrix.row(i),vector));
end
end
Thread control

spawn \{ I \}: launches a new thread to execute instruction I;

The thread terminates after evaluating its expressions and then returns the thread id that was created.

– The join predicate id indicates the end of the thread execution
- Operations for creating, getting and restoring a lock:

\begin{verbatim}
l = new lock(); l.lock(); l.unlock();
\end{verbatim}

The acquisition of a new lock is a blocking operation as the thread attempting to acquire a lock is blocked as long as the lock is held by another thread. If two threads simultaneously attempt to acquire a lock, only one succeeds

Example:

\begin{verbatim}
TwoTasksInParallel { 
id1 = spawn \{ firsttask \}
id2 = spawn \{ secondtask \}
join id1
join id2
}
\end{verbatim}
Q3: write pseudo code for executing n tasks $task(i)$ with threads

- Since we want to perform tasks independently, the simplest implementation is to launch $n$ independent threads, each responsible for carrying $task.run(i)$ for different values of $i$.
- First loop is used to launch these $n$ threads then wait for a second loop they have all finished (with join).

```
ParallelFor {
    for i from 0 to n-1 do id[i] = spawn { task(i) }
    for i from 0 to n-1 do join id[i]
}
```

- Issues
Starting a thread is expensive (time and space wise), and run more threads than the existing processors is also expensive (context switching) so the naive solution is likely to be slow, especially if the calculation made by task.run (i) is relatively inexpensive.
Question 4

• Suppose we have k processors and tasks that operate on n data require roughly the same processing time.
• How can we speed up the processing? Write the pseudo code.

Solution.
We split the n values into k-chuncks of size ~n/k and assign them to the k processors (threads)

```
ParallelFor {
  CHUNK=floor(n/k); j=0;
  for i from 0 to n step by CHUNK do
    id[j++] = spawn { for j from i to min(i+CHUNK,n)-1 do task(j) }
    for i from 0 to j-1 do join id[i]
}
```
Question 5

- **Question 5** What risk do we have in the case that tasks require very different execution times? Propose a solution to improve the parallelism when you know in advance processing time for different tasks, by reducing it to an optimization problem (We do not ask here pseudo-code.) Reflect of the complexity of the problem?

**Solution.**
- In both versions above, we can speak of "static scheduling" is decided from the beginning which thread will handle what task indices.
- This technique is simple but has a flaw: if the cost (in terms of computation time) of the task task (i) varies greatly depending on the value of i, we run the may assign all "expensive" i values in the same thread and the "easy" i values to other threads, so that (after some time) we will end up with a single thread that works. It is not effective.

- If we know the processing time, the idea is to optimize the assignment of tasks to processors:
  - k processors for n independent tasks whose duration is d_i (i = 1...n), we can is to find a partition (I_j) 1<= j <= k of the tasks that minimizes
  \[
  \max\left(\sum_{i \in I_j} d_i \mid j = 1, \ldots, k\right)
  \]
  - This problem is difficult (more precisely, it is NP-complete). The study of approximate solutions or heuristics practices this type of issue scheduling is a branch full of research operational.
Question 6:

Suppose now we do not know in advance the processing time of tasks. Propose a solution using a queue and write the pseudo code.

Solution.
As the processing time is unknown, it is better to make "dynamic scheduling“. We construct a queue in which one inserts indices (or intervals of indices) to treat, and our k threads operate on the queue. It is then necessary that the file/queue is protected by a lock. If a thread finds that the queue is empty, it dies.

```
DynamicParallelFor{
F = new File;
L = new lock();
for i from 1 to n do F.put(i) // fill in the queue with the data
for i from 0 to k-1 do // for each of the threads
    id[i] = spawn { //start thread
        while (true) {
            L.lock() // thread takes control
            if F.isempty() then { L.unlock(); exit() }
            j = F.take() // get data from the queue
            L.unlock()
            task(j) // compute task(j)
        }
    }
for i from 0 to k-1 do join id[i] // sync the tasks
}
```
Assume that the method returns an integer and we want to sum up all these integers:

```c
int sum = 0;
for (int i = 0 i < n i += )
sum += task (i);
```

Solution.
This situation is common and "easy" because the operation + is associative / commutative. $k$ threads launched, each will complete (in arbitrary order) providing a partial sum, we have only to add these partial sums (in the order they come to us) in a global sum protected by a `lock`.

```c
DynamicParallelSum (n, task){
S = 0
LS = new lock () // declare the lock
ParallelFor (n,
    proc (i) {
        s = task(i);
        LS.lock() // lock access to S
        S += s
        LS.unlock()
    }
)
return S
}```
Q8 - parallelization of Merge sort:

- Propose an algorithm to parallelize the merging of two sub-arrays of index table T respective \([l_1, h_1]\) and \([l_2, h_2]\). We can assume that the procedure ParallelMerge does merge sort in those two parts and the results is stored in the array Tmp in the range \([l_3, h_3]\)

- We use divide an conquer in recursion.
  - T1 and T2 are the same array but referred here as two different ones for easier understanding of the splitting.
  - T3 is the final table for the results.
**ParallelMerge** (tableau T, int l1, int h1, int l2, int h2, tableau Tmp, int l3, int h3)
{
    if (h1-l1 < h2-l2)
        ParallelMerge (T,l2,h2, l1,h1, Tmp,l3,h3) // first range should be longer
    else if (h2-l2 == 1){  //if right range is 1, only one element
        if (h2-h1 == 1) { // if neighboring elements
            Tmp[l3]=min(T[l1],T[l2])   //assign first element in T3
            Tmp[l3+1]=max(T[l1],T[l2])} // //assign second element in T3
        else
    }
    else { // if bothe ranges >1
        int m1=l1+(h1-l1)/2; // find middle element of range [h1-l1]
        int m2=BinarySearch(T[m1],T,l2,h2); // divide the second array range with
            // T[m1]
        Tmp[l3+(m1-l1)+(m2-l2)]=T[m1] // place T[m1] in Tmp
        // create recursive processes for the left and right parts of the two ranges
        int id1 = spawn{ParallelMerge(T,l1,m1,l2,m2,Tmp,l3,l3+(m1-l1)+(m2-l2))}
        int id2 = spawn{ParallelMerge(T,m1+1,h1,m2,h2,Tmp,l3+(m1-l1)+(m2-l2)+1,h3)}
        join id1; join id2; //wait until they complete.
    }
}
Q 8 .. analysis

- if \((h1-l1 < h2-l2)\) switch \([T1,l1,h1],[T2,l2,h2]\) :
  Select the longest of the two arrays.
- \(\text{int } m1=l1+(h1-l1)/2\): the middle of \(T1\)
- \(\text{int } m2=\text{BinarySearch}(T1[m1],T2,l2,h2)\): we split \(T2\) by the first element of the second half of \(T1\)
- \(T3[m1+m2]=T1[m1]\): we have spliced both Arrays by the same value. The new position of this value is the number of elements that are smaller than in it in \(T1\) plus the number of elements that are smaller than in it in \(T2\).
- \(\text{int } id1 = \text{spawn}\{\text{ParallelMerge}(T1,l1,m1-1,T2,l2,m2,T3,l3,l3+m1-l1+m2-l2-1)\}\)
- \(\text{int } id2 = \text{spawn}\{\text{ParallelMerge}(T1,m1,h1,T2,m2,h2,T3,l3+m1-l1+m2-l2,h3)\}\)

Call recursively in an distributed manner the Merge procedure for the corresponding “halves” of the \(T1, T2\) spliced arrays.
Question 9 – complexity analysis

• The depth of the computation is defined by the binary search \( O(\log n) \) and by the recursion: \( P(n) \leq P(3n/4) + O(\log n) \)

• Therefore bind by \( O(\log^2(n)) \)
Question 10

- Articulate the parallel version of the mergesort:

```c
ParallelMergeSort(tableau T, int l, int h, tableau R){
    if (h-1>1{
        int m=l+(h-l)/2;
        int id1 = spawn {ParallelMergeSort(T,l,m,R,)}
        int id2 = spawn {ParallelMergeSort(T,m,h,R,)}
        join id1; join id2;
        ParallelMerge(T,l,m,T,m,h,R,l,h)
        Copy(Tmp,1,h,T)
    }
}
```
Q11, Q12

11: Suppose we have \( k \) processors. How should we allocate the tasks?

Solution.

- The tasks are predictable cost. It is therefore possible to set a depth maximum beyond which the stops parallel calls.

- A classical optimization in recursive sorting algorithms is to stop the recursion when tables become too small to handle (typically a few tens of elements, when a sorting non-recursive type "insertion sort" becomes faster).
Q12 change your pseudo-code to introduce this optimization?

Task(T) {
    if size(T) < s then iterative_algorithm(T)
    else
        divide(T, T1, T2)
        Task(T1)
        Task(T2)
        merge(T1, T2)
}
Conclusion

It makes sense to use parallelism for a task $T$

- if its complexity is considerable and exceeds the complexity requirements incurred from parallelism.

- If there are available CPUs sufficient for the tasks we want to create.