Using Parallel-For Loops in Practice

1 What happens when we make a loop parallel?

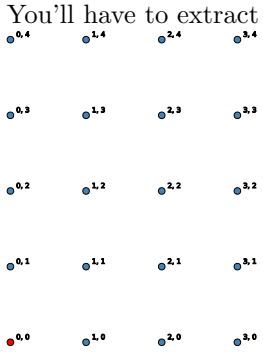
1.1 Basic case

We have a parallel-for loop construct. Can you use it all the time? No, not all the time. That will depend on the structure of the underlying graph of dependencies.

Let's assume we have a program with two nested for-loops as an example, but let's leave the content of the loops unspecified.

```
for (int i=0; i<n; ++i)
  for (int j=0; j<m; ++j)
    ...
```

You'll have to extract the graph of dependency. For now, let's assume the tasks are all indepedent:

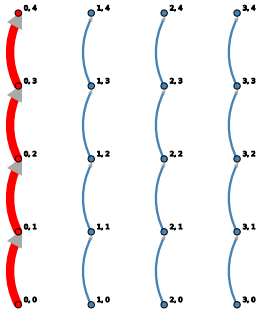


1.2 Parallelizing the `i`-loop

What happens if we make the `i`-loop parallel:

```
parfor (int i=0; i<n; ++i)
  for (int j=0; j<m; ++j)
    ...
```

(Obviously that syntax does not work, we are looking at it conceptually.) Each iteration of the `i`-loop will execute independently, but the iterations of the `j`-loop will be one after the other (for each `i`). So the execution will look like:



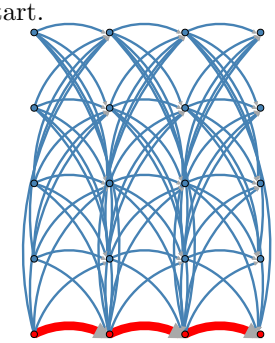
Now, is that OK? Yes, it is OK! because the dependencies that are implied by the code are compatible with the actual dependencies of the problem. You get more dependencies in the execution than in the problem. So you may not get ALL the parallelism, but you will get SOME parallelism and the execution will be correct.

1.3 Parallelizing the `j`-loop

What if we were making the `j`-loop parallel like that:

```
for (int i=0; i<n; ++i)
  parfor (int j=0; j<m; ++j)
    ...
```

Then each iteration of the `j` loop will be independent. But the implicit synchronization at the end of the parallel loop still means that all the `j` task for a particular `i` needs to be complete before the next `i` can start.



Now; is that OK? Yes it is. Once again, the code will imply SOME dependencies in the execution that are not necessary. But there are no missing dependencies. So you will get some parallelism, but not all the parallelism available.

2 Rewriting loops

2.1 Fully parallel

Can you make both loops parallel? Maybe like that:

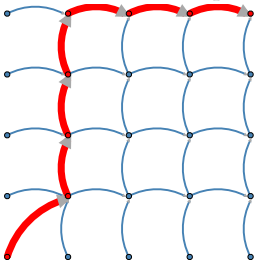
```
parfor (int i=0; i<n; ++i)
  parfor (int j=0; j<m; ++j)
    ...
```

Fundamentally, you should be able to do that. But many systems in practice may not support that. And that create "nested threads" which may not have the behavior that you want. You may prefer re-writing the loops in a single loop and map the `i` and `j` values:

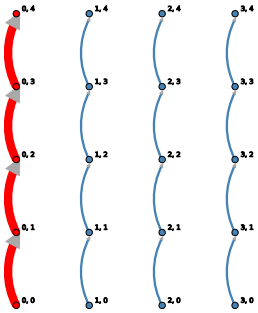
```
parfor (int x=0; x<n*m; ++x) {
  int i = x/n;
  int j = x%n;
  ...
}
```

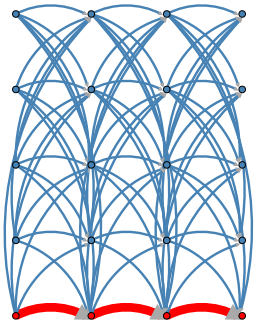
2.2 A misaligned example

Now rewritting loops can be useful in cases where the loops do not align with the parallelism. Consider a classic two nested loop that exhibit this parallelism:



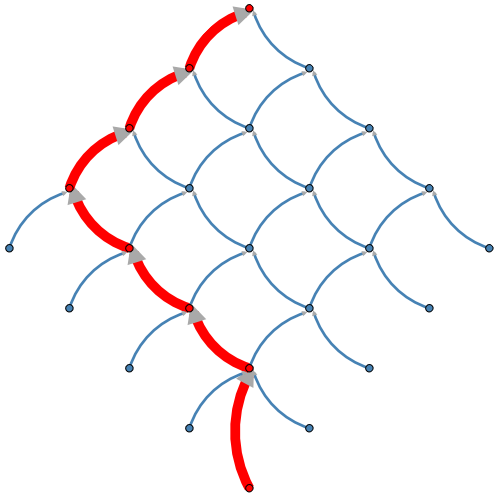
If you parallelise the `i`-loop or the `j`-loop you get either structures:





Neither structure is good. One of them breaks the vertical depednencies, the other breaks the vertical dependencies. So you can not make the `i`-loop parallel. And you can not make the `j`-loop parallel.

So does that mean, we can't use parallel loops here? We can, but we need to think a little bit more. See what happens when we turn the graph of dependencies by 45 degrees:



We do see there are independent tasks, they just don't align with the `i`-loop and the `j`-loop.

So if we can re-write the code to be one diagonal at a time, like this:

```
for (int diagonal=0; diagonal<nbdiagonal; ++diagonal) {
  for (int taskindiagonal=0; taskindiagonal<nbtaskindiagonal ++taskindiagonal)
  int i = ...;
  int j = ...;
  ...
}
```

Then we can make the taskindiagonal loop a parallel-for loop.

Now is it always 45 degree diagonal? No, but the point is to identify a pattern of tasks that are always independent and reorder the loops based on that.

3 Conclusion

You'll understand now why knowing how to extract a dependency graphs is important. You need to know which tasks are independent to see which loop can be made a parallel-for loop. Maybe the code as written has no loop which can be made parallel, but maybe we can rewrite the loop to expose parallel loops.

If there is no parallelism in the code, maybe we need to redesign algorithm to have parallel tasks. And maybe we will need to reindex the loop to be able to use parallel-for loops.

This document discussed the case of two nested loops, but a code structured as three nested loop analyzes similarly. A single loop analyze similarly. Recursive codes analyzes similarly.