Project of Distributed System: Paradigms and Models

Matteo Mazza: 463856

Exam date: 12/02/2015

Project: Parallel Prefix

Prefix sum (the sequential version):

Given an array A of n elements and a commutative and associative function +, the prefix sum problem consist of generate an output Array B such that B[*i*] = A[*0*] + A[*1*] + … A[*i*].

Thus, in the prefix sum problem, each output element B[*i*] depends on all the elements A[*j*] with *j <= i*.

The trivial sequential program is:

```
for i = 1 to n
        A[i] = A[i]+A[i-1]
```

Parallel prefix project:

It does not exist a trivial solution at the parallel prefix problem. We can adopt different algorithms to parallelize it, so we must compare them to choose the better one for each situation.

In this short report we will have a look at some algorithms and we will compare them adopting an abstract cost model. Then we will look at a real implementation and compare the expected behavior with the real measurements.

In the project folder there are the source files, the test scripts, the Makefile for compilation.
There is also a folder with the plots generated from the test executions and the abstract cost models written in this report.

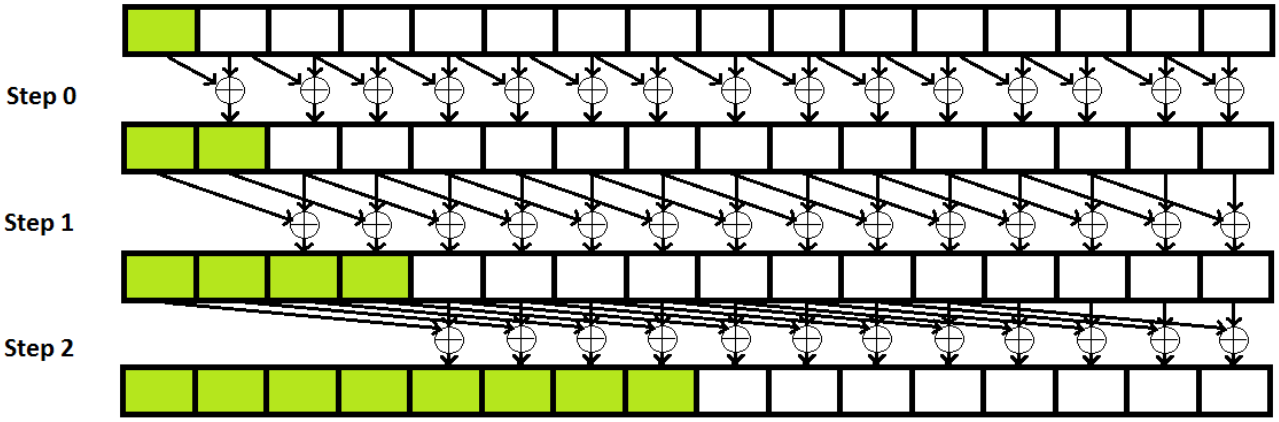
At the end of this report some plots and the instructions to compile and execute are given.

Algorithm 1

Considering the case in which we have unlimited computational resources, we can write an algorithm that at the generic step *k* compute the 2^{k+1} output elements. (call n the length of the input sequence)

Virtual Processor i:
for k = 0 to log_2 n
A[i] = A[i] + A[i-2^k]

In fact before doing any step (before step *0*) the first element is already correct. After step *0*, the first 2 elements are correct. After step *1*, the first 4 elements are correct and so on.

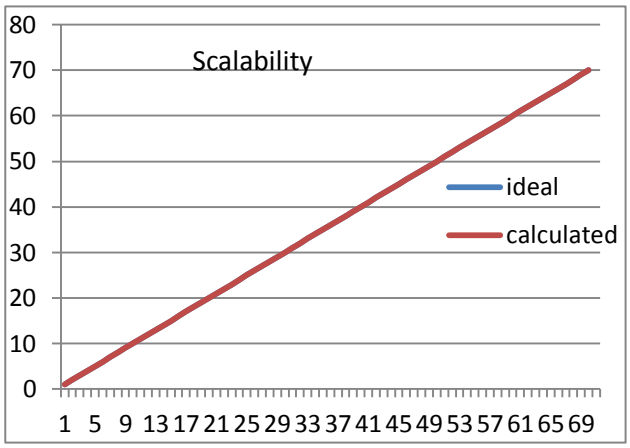


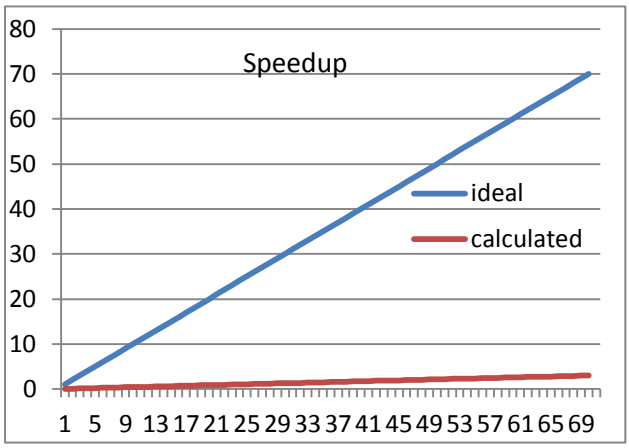
Since we have only a limited set of resources, each processor must compute more virtual steps. Consider the case we have an input array of length *n* and *m* processors, at each step each processor must compute $\frac{n}{m}$ virtual steps.
Since we don't have synchronous processors, we also need a memory barrier after each step to synchronize processors between steps.

Abstract cost model:
Call *Tseq* the time in which the sequential algorithm compute the output array.
The number of steps is $\log_2 n$. with *n* the length of the input sequence.
The time spent in a step is $\frac{Tseq}{m}$.
From the above reasoning we can obtain an approximation of the performance of this algorithm:

$$Tpar(m) = \#steps \cdot Tstep = \log_2 n \cdot \frac{Tseq}{m}$$

This formula give us an optimal scalability but a very bad speedup.





There is a possible improvement in the algorithm.

In the previous version at each step, each processor compute on an interval of length $\frac{n}{m}$. Since after each step *k*, 2^{k+1} elements are already computed correctly, at each step *k* each processor can compute on a smaller interval of $\frac{n}{m} - \frac{2^{k+1}}{m}$ elements.

However this new algorithm is not enough to see a great improvement in the speedup graph.

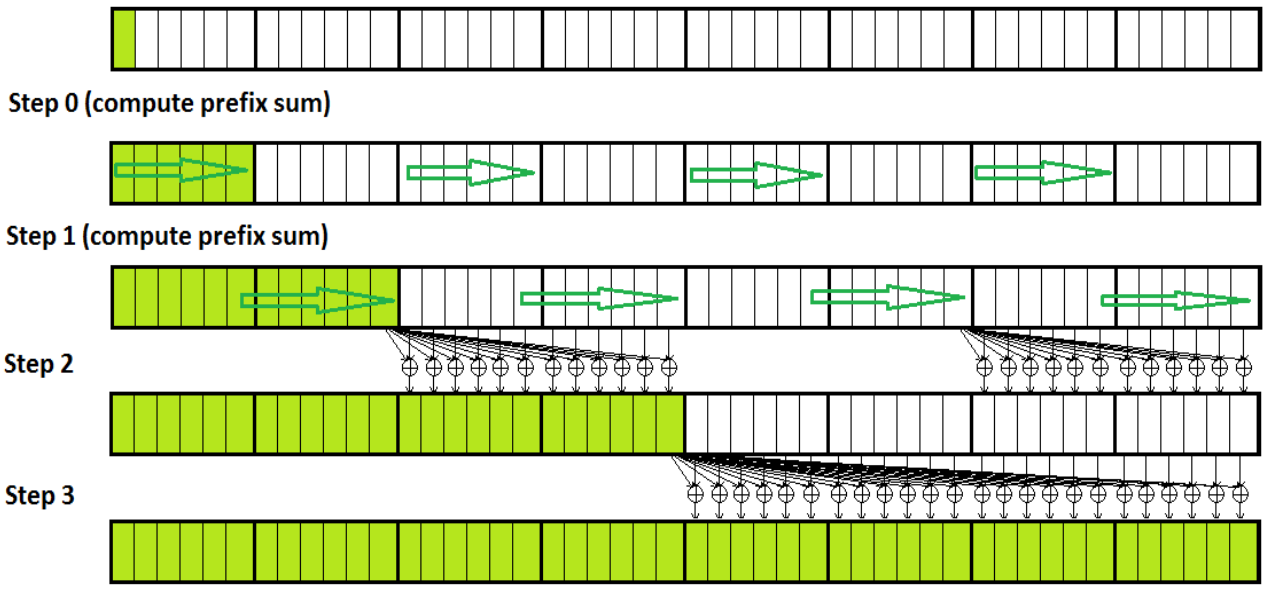
We need 24 processors to compute the prefix sum in less time respect to the sequential case.

Algorithm 2

Starting from the same idea of algorithm 1,

Virtual Processor i:
for k = 0 to log_2 n
A[i] = A[i] + A[i-2^k]

we can change the real implementation on a real machine with a limited set of resources.
At each step, each processor, instead of computing the virtual step on a subset of the input array, solves the prefix sum problem on a subset of the input array.
Call *n* the length of the input; *m* the number of processors.
We subdivide the array in *2m* subarrays and we compute the prefix sum on all this subarrays in 2 steps.
Then we start $\log_2 m$ steps in which we compute the commutative and associative function between all the elements in one array and the last element of one subarray following this schema:



Abstract Cost model:

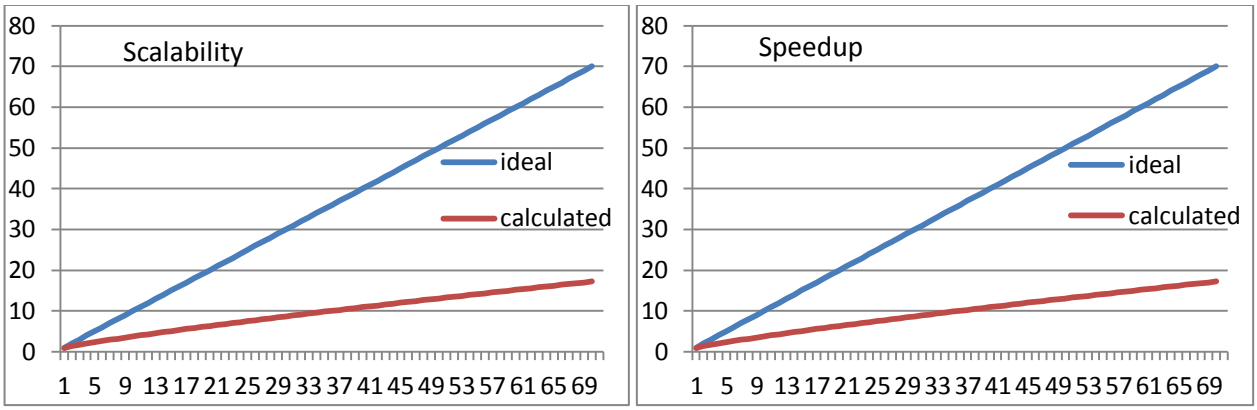
At each step, each processor compute $\frac{n}{2m}$ functions in $\frac{Tseq}{2m}$ time. The structure is like the first algorithm but the number of steps is changed in $2 + \log_2 m$ and the time is divided by 2:

$$Tpar(m) = \#steps \cdot Tstep = (2 + \log_2 m) \cdot \frac{Tseq}{2m}$$

This is better than the first algorithm if

$$\frac{(2+\log_2 m)}{2} < \log_2 n$$

since the length of the input, in this kind of problem, is higher than the parallelism degree, we can imagine that this algorithm performs better in every runs.



We notice a high deterioration in the scalability, however we gain a lot in the speedup. With parallelism degree equal to 1 we obtain (more or less) the same performance of the sequential program, instead of the 24 processors needed by the first algorithm. In fact, if we have only 1 worker, the algorithm compute only the first 2 stages that are exactly the prefix_sum algorithm.

Furthermore this algorithm makes a better use of the cache respect to the algorithm 1, in fact each processor does not need to retrieve 2 elements to apply each function because 1 element is already in cache. In the first 2 steps the element already in cache is the last calculated, in the remaining $\log_2 m$ steps the element is loaded in cache only 1 time for every calculations during the step.

Algorithm 3

Instead of computing all in parallel, it is possible to spend a little time to compute something in a sequential way.

The algorithm 3 is subdivided in 3 stages and one of them is executed sequentially.

As always, call *n* the length of the input array and *m* the number of processors.

Stage 0:

Subdivide the input array in *m+1* subarrays. Compute in parallel the prefix sum in the first *m* subarrays.

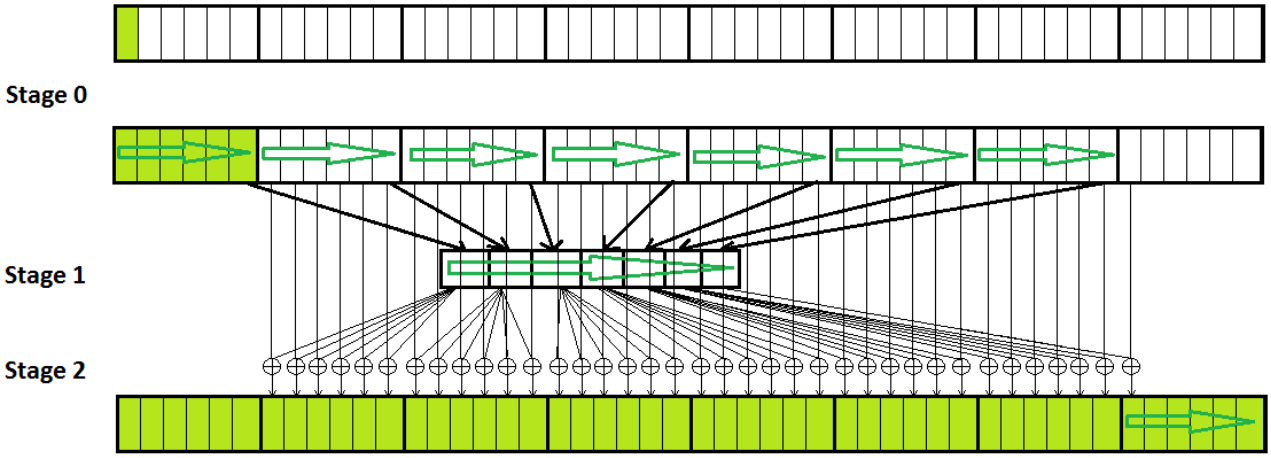
Stage 1 (sequential):

Create a temporary array T of *m* positions such that T[*i*] is equal to the last elements of subarray *i*.

Compute the prefix sum sequentially on this short subarray.

Stage 2:

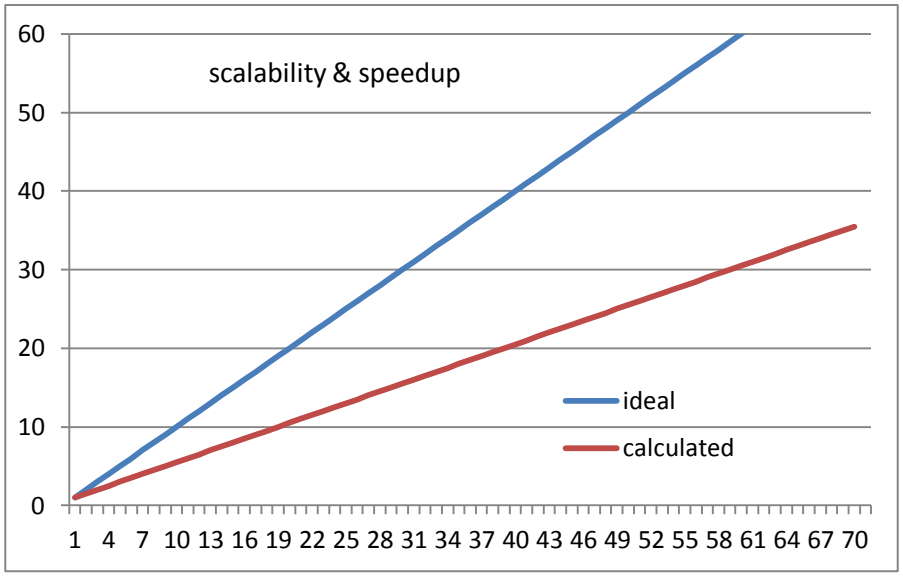
For each subarray S_i, apply the commutative and associative function with the elements of S_i and T[*i-1*]. Except for the last subarray in which we compute the prefix_sum taking into account the last value of the array T



Abstract cost model:

Both stage 0 and stage 2 are the executions of $\frac{n}{m+1}$ functions per processor. In the case in which *n* is various order of magnitude greater than *m*, the time spent in stage 1 (the sequential one) is negligible.

Tpar(m) = $\frac{2\,Tseq}{m+1}$



Under the previous hypothesis *n >> m*, we have an increase in scalability and speedup. Furthermore the parallel program equals the sequential one at parallelism degree 1: *Tpar(1) = Tseq*.

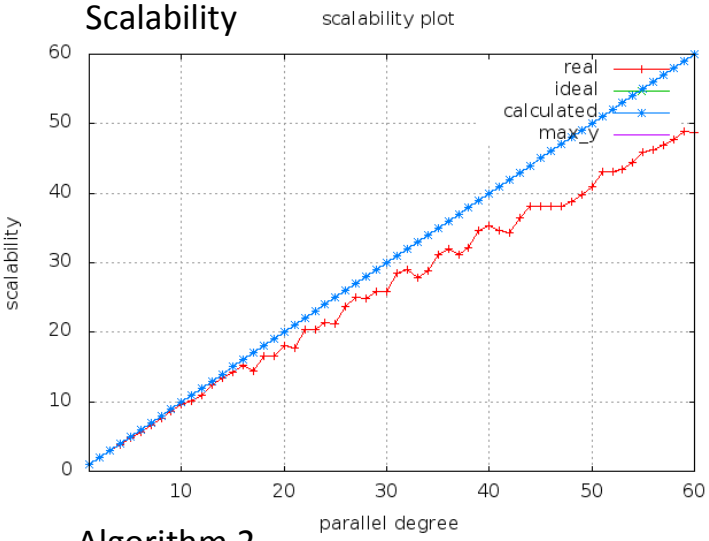
Implementation:
Having a look at these 3 algorithms and their abstract cost models, we can see that the third algorithm is better, so we can choose it to be implemented.
However in this project I've implemented all the algorithms to have a look at how the real measures are close to the abstract calculation done before.

Each algorithm is implemented in C with posix_thread to be executed in a shared memory environment. The tests were done on my laptop, on the machine r720-phi.itc.unipi.it and on the xeon phi co-processor. All the performance plots are obtained by appropriate averaging the execution times on the xeon phi.
Each algorithm is tested on different commutative and associative functions. From the less computational expensive to the most one, call them *f1*, *f2* and *f3*.
Each implementation have the same structure: one thread is the coordinator and others *m* identical threads are the workers. The coordinator is only in charge to synchronize the workers.

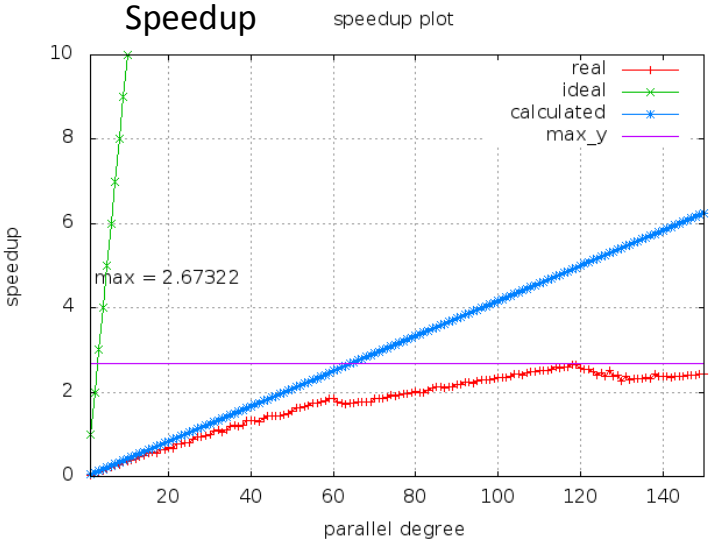
Graphs:

Algorithm 1

Scalability

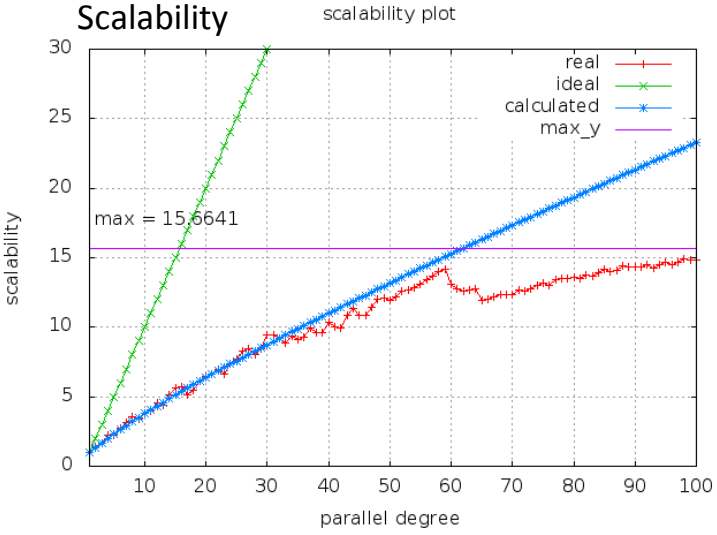


Speedup

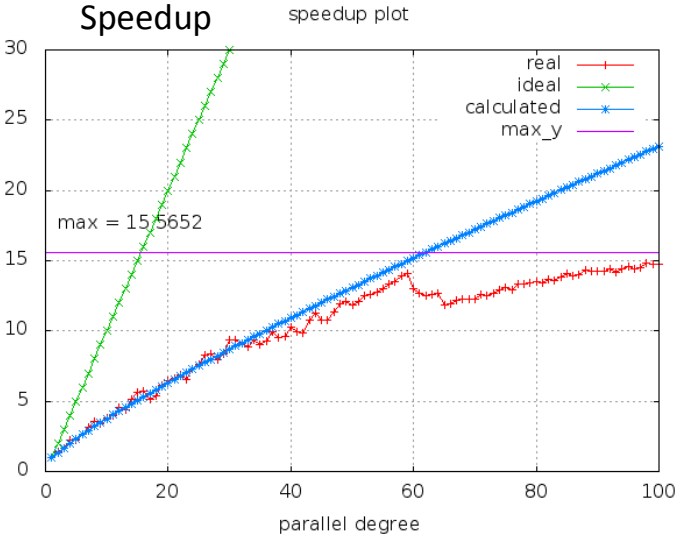


Algorithm 2

Scalability

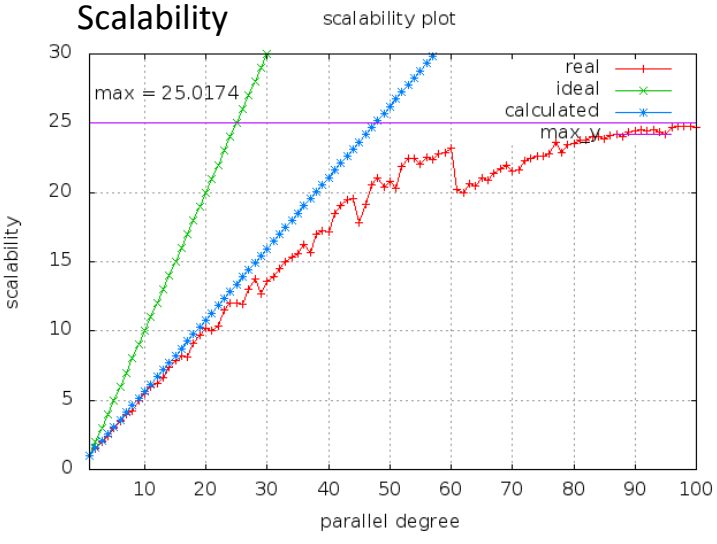


Speedup

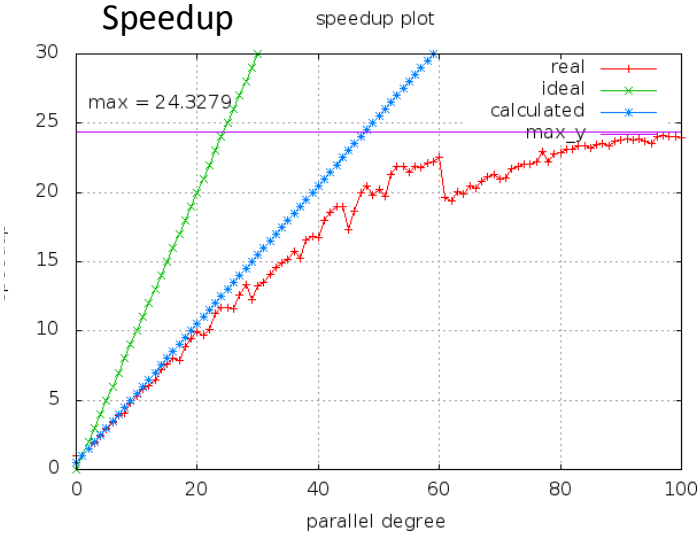


Algorithm 3

Scalability



Speedup



User Manual:

The project files are organized in this way:

- parallel_prefix
  - algorithm1
    - source
    - source with optimization
  - algorithm2
    - source
    - source with optimization
  - algorithm3
    - source
  - graphs

In each source directory it is present a Makefile.

To compile the program:

$ cd source_directory
$ make

To compile for the mic (on the host machine):

$ make mic

To perform the tests and to generate the graphs (from the local machine):

$ cd source_directory
$ make test1_mic0
$ make test2_mic1
$ make test3_mic1

- *test1* does the computation with f1 (the less expensive function), *test2* with f2 and *test3* with f3 (the more expensive function). *_mic0* and *_mic1* select the co-processor to be used
- to generate the graphs (using makefile) there are two dependencies: *gnuplot* and *php5*
- to avoid a lot of ssh and scp password request, it is useful to configure ssh pulic key

Of course it is possible to run "by hand" the compiled programs.

To run the sequential one: **./prefix_sum_seq input_size func [test]** where func represent the function to be computed: *0* -> f1 ; *1* -> f2 ; *2* ->f3

For example **./prefix_sum_seq 1000000 0** compute the less expensive function on 1 million items.

The parallel version has one parameter more: the parallel_degree:

./prefix_sum input_size parallel_degree func [test].

The test parameter is used to make plots and to check the correctness. It can be omitted.

The binary compiled for the mic are renamed in **prefix_sum_mic** and **prefix_sum_mic_seq**.

The "*make test*" command copies the necessary files on the host machine, compiles them, copies the binaries on the mic and starts the test computations with different parallelism degrees (with the selected function). Than it analyzes the outputs (time statistics) deleting the smallest and greatest time value and obtaining a mean value from the remaining statistics. This is done for each parallel degree. Than a graph with the ideal, calculated and real performances is created locally in the source_directory.