SPM Report

Massimo Equi
AY 2017/2018

1 Introduction

The chosen project is the one consisting in using a genetic algorithm to estimate an unknown function f given a set of pairs $(x, f(x))$ as input.

2 User Manual

In order to compile and run the program follow these instructions:

- move to the `SPM_FinalProject` directory;
- (if a previous pair of `main.out` and `parallel_main.out` files is present) run the command `make clean`;
- compile the program with the command `make -f [makefile]`
- run the sequential program with `./main.out [parameters] < [input_file]`
- run the parallel program with `./parallel_main.out [parameters] < [input_file]`
  Here it is an example of a typical session:

```
$ make clean
$ make -f Makefile.icc
$ make -f Makefile.parallel.icc
$ ./main.out 12000 5 4000 10 20 0.5 no < input_cos\(x\)-pow\(x,3\)10-3.txt
$ ./parallel_main.out 12000 5 4000 10 20 0.5 8 no < input_cos\(x\)-pow\(x,3\)10-3.txt
```

Makefile There are several Makefiles to compile the program with different options and compilers. For compiling the sequential program, `[makefile]` can be one of the following:

  – `make -f Makefile` (or simply `make`) compiles the program with the `g++` compiler;
  – `make -f Makefile.icc` compiles the program with the `icc` compiler;

– `make -f Makefile.icc.mic` compiles the program with the `icc` compiler using the flag `-mmic`;

All of the above commands output a `main.out` file.
For compiling the parallel program instead use the following in place of `[makefile]`:

– `make -f Makefile.parallel` compiles the program with the `g++` compiler;
– `make -f Makefile.parallel.icc` compiles the program with the `icc` compiler;
– `make -f Makefile.parallel.icc.mic` compiles the program with the `icc` compiler using the flag `-mmic`;

All of the above commands output a `parallel_main.out` file.

Run the program The program runs with several parameters and has to be called with and input file. Calling `./main.out` or `./parallel_main.out` with no arguments will show a quick guide. `[parameters]` consists in the following:

– `tree_no` is the number of trees that will be created in the random pool generated at the beginning of every execution;
– `depthmax` is the maximum possible depth a tree in the pool can have;
– `threshold` is the number of trees the algorithm will select to generate the next population (has to be lesser than `tree_no`);
– `randmax` is the largest value a leaf of a tree holding a constant can have;
– `gen_no` is the number of generations such that, once reached, the algorithm will terminate;
– `err` is the threshold for the fitness. If the fitness of the best tree is lesser that `err` the program will terminate; if `err` is a negative number the algorithm will stop only after reaching `gen_no` generations;
– `nw` is the number of parallel workers (skip this argument if the program is sequential);
– `debug` can be either `yes` or `no`; if it is `yes` it runs the program with detailed information at each generation.

The program requires an input file to run. Any file in this project root directory whose name begins with the string `input_` is a legal input file. In all the executions mentioned in this report we used the input file `input_cos(x)-pow(x,3)10-3.txt` (`10-3` in the name of the file means there are 10^3 input pairs $(x, f(x))$ in that file).
After having compiled the main program, the command `make [makefile] test` can be used to generate the `test_FitnessTime.out`. This little test program can be executed specifying an input file (and no arguments) in order to get some estimations about the time needed to evaluate a tree and compute the fitness function.

3 Structure of the code

The code has two main directories: `src/` is where the `.cpp` source files are located while `include/` is where the `.h` header files can be found. Both `src/` and `include/` are structured in three subdirectories: `grammar/`, `genetics/` and `main/`.

grammar In `grammar/` there are all the classes needed to implement the grammar provided in the project assignment. Every class is implementing a nonterminal symbol of the grammar with methods that allows it to be expanded using one of its production. The nonterminal `<unop>` and `<binop>` are directly encapsulated in the class `Node` which of course implement the nonterminal `<node>`. Each of these classes gives a concrete implementation of the abstract class `INode`, which is meant to represent a generic node in the syntax tree. An object of class `Node` can be randomly expanded to a given depth using the method `Node::expandRandom()`. Finally, the parameter `randmax` gives and upper bound to the highest number that the class `Const` can store.

genetics In `genetics/` the class `Tree` is implementing the behavior of a syntax tree meant to represent a function. Basically, `Tree` provides a nice interface to store and manage the root node of this syntax tree. Indeed, such syntax tree will be generated calling the `Node::expandRandom()` method of that root node. The class `Forest` is the core of the algorithm since it manages a pool of `Tree`s providing methods to perform mutation and crossover over them. The `ParallelForest` class is a subclass of `Forest` and it is used to override the part of the code that was chosen to be turned from sequential to parallel.

main In `main/` the file `evolution_cyle.hpp` uses a `Forest` to implement the genetic algorithm. `main.cpp` and `parallel_main.cpp` call `evolution_cycle.hpp` passing it respectively either a `Forest` or a `ParallelForest`.

4 Parallelization Choices

4.1 Decomposition Strategy

The main operations that our algorithm is performing are: *selection*, *mutation* and *crossover*, *replication*. Running the sequential version of the program we see that the completion time of *selection* is usually two orders of magnitude larger than the one needed for *replication*, *mutation* and *crossover*. Thus, we focus only on optimizing *selection* since, for this reason, that is the crucial issue.
Analyzing the structure of the problem we observe that it is embarrassingly parallel at two different level of grain. At each iteration our program has to compute the fitness function of all the trees in our pool, hence for every of those trees it is required to evaluate the function that tree is representing over all input data points. This means we could parallelize:

- at a **fine grain** level, evaluating in parallel the function represented by a single tree over different data points;
- at a **coarse grain** level, computing in parallel the fitness function for different trees.

Therefore, the first thing we have to discuss if it is worth it to parallelize at any of this two levels of grain.

Single tree parallelization In a typical execution of the code we have to handle trees whose depth is set to be not greater that 8. This is because most of the times the best trees tent to be the shallower ones hence it is useless to set a very high depth since the deeper trees most probably will be discarded soon. Moreover, the average depth is usually smaller than 3; this means that on the average we perform $2^{3+1} - 1 = 15$ floating point operations per tree, assuming a floating point operation per node. This means that the amount of time we spend evaluating a tree has roughly at most $10^{-1}\mu s$ order of magnitude. If we compare this with the time spent to create a thread (approximately $10\mu s$) we realize that even if every worker of our hypothetical parallel computation processed 100 input points this would still be comparable with the effort needed to set up the thread for that worker, hence at least 1000 input points per thread are needed to have a reasonable advantage. Therefore, if for instance we supposed to work with 10 workers, we would need more that 10^4 input points to get any advantage from the parallelization, and this would rarely be the case. This is clearly a scenario where we would have a very poor scalability. Thus, we conclude claiming that it is not worth it to parallelize the program relaying on such a fine grain.

Pool parallelization Considering instead the whole pool of trees the workload that would be assigned to a worker in a parallel computation is reasonably large and, more importantly, increases easily with the input size. The fitness function for a tree is computed in order of $10^2\mu s$, hence having only 10 trees per worker is already enough to get a workload of $10^3\mu s$. Typically we work with a pool consisting of thousands of trees and this usually means that every worker will have order of 10^2 trees to handle, even in situations with hundreds of parallel workers. Hence we conclude that is worth it to parallelize at this level of grain.

4.2 Implementation of the Parallel Part of the Code

The difference between the sequential and the parallel version of the code resides in the class `ParallelForest`. This class has two additional instance variables: `nw` specifies the number of workers to be used for the parallel computation; `pf` is an unique pointer to a `FastFlow ParallelFor` object. `ParallelForest` overrides the `Forest::fitness()` method, replacing the `C++` sequential `for` loop with the `FastFlow pf->parallel_for()` function, called with the `ParallelForest::nw` parameter. We could use a parallel for due to the fact that no synchronization is needed among the various iterations of the loop since computing the fitness function resembles the *map* pattern.

5 Expected Results

We want to be able to express our results in terms of *speedup*, *scalability* and *efficiency*, hence the first thing we need to know is the running time of the sequential program. We ran the sequential program on the Xeon machine 12 times with these parameters:

```
tree_no = 12000; depthmax = 5; threshold = 4000;
randmax = 10; gen_no = 20; err = -1.
```

Removing the most outstanding outliers and taking the average we got a sequential completion time of $T_{seqXeon} = 28.62564s$. For the Xeon PHI we followed the same procedure for performing the measurements but we decided to use a smaller set of trees since the clock speed of this machine is significantly slower. We therefore used:

```
tree_no = 2400; depthmax = 5; threshold = 800;
randmax = 10; gen_no = 20; err = -1;
```

and we got this sequential completion time: $T_{seqPHI} = 60.10967$.
In our scenario we have parallelized our code using a farm pattern whose overheads resides mostly on splitting the initial task in several subtasks and assigning them to different parallel executors. This splitting-and-assigning phase has to be performed at the beginning of each iteration of the evolution cycle, hence could potentially lead to a substantial overhead
What we expect to get is a speedup that grows steadily with the growing of the parallel executors. In particular, we would like to at least halve the completion time with already 2 to 6 workers and to almost achieve the best completion time with 10 to 12 workers. As a consequence, we expect to have a speed up closer to the ideal one using up to 6 workers and then to have it grow slower. Additionally, an ideal speedup would imply the following ideal completion times:

$$T_{par(2)} = \frac{T_{seqXeon}}{2} = 14.31282s \tag{5.1}$$

$$T_{par(4)} = \frac{T_{seqXeon}}{4} = 7.15641s \tag{5.2}$$

$$T_{par(6)} = \frac{T_{seqXeon}}{6} = 4.77094s \tag{5.3}$$

Assuming that roughly $frac110$ of the sequential completion time is related to the part of the code which remains serial also in the parallel version (*mutation* and *crossover* time plus *replication* time) we will effectively parallelize the $\frac{9}{10}$ of that time. Considering that we will also pay for some overhead, realistically the completion times could be the followings:

$$T_{par(2)} = \frac{\frac{9}{10}T_{seqXeon}}{2} + \frac{1}{10}T_{seqXeon} + overhead = 15.744102s + overhead \tag{5.4}$$

$$T_{par(4)} = \frac{\frac{9}{10}T_{seqXeon}}{4} + \frac{1}{10}T_{seqXeon} + overhead = 9.30333s + overhead \tag{5.5}$$

$$T_{par(6)} = \frac{\frac{9}{10}T_{seqXeon}}{6} + \frac{1}{10}T_{seqXeon} + overhead = 7.15641s + overhead \tag{5.6}$$

For the Xeon PHI machine we expect indeed to have a good scalability since the clock speed is lower and even a small amount of trees per worker, say 100, could be a reasonable workload.

6 Achieved Results

Here we report the actual completion times we measured on the Xeon CPU E-2650 and the Xeon PHI machines for the parallel executions of the program, alongside with the graphs for the *speedup*, *scalability* and *efficiency*, comparing these results with the sequential program's ones. The `err` parameter has been always set to `-1` in order to perform exactly `gen_no` iterations; we would not have been able to have comparable results otherwise.

Format of the results The program uses `std::chrono` statements in order to outputs the time spent to perform the computation. We stored all the results in two files: the file `tests` for the Xeon machine and the file `tests.phi` for the Xeon PHI machine. Both files has a line storing the output of the `uptime` command in order to record the state of the machine at the execution time. Each one of the lines storing the results of the tests starts with the string `results`, hence you can get the list of all the outputs with the command `grep results tests[.phi]`. `tests` has been produced recording the output on the shell via a `script` command while for `tests.phi` we just used output redirection (since there is no command `script` on the Xeon PHI). The scripts `test.sh` and `test.phi.sh` run all the tests from scratch (without recording them by their own), but this requires a huge amount of time and we think it will not be needed.

Gathering the results Certainly, in order to have comparable measures and to compute *speedup*, *scalability* and *efficiency*, we used the same parameter setting we had for the sequential execution and we repeat it here for the seek of readability. Thus, for the **Xeon** we had:

```
tree_no = 12000; depthmax = 5; threshold = 4000;
randmax = 10; gen_no = 20; err = -1;
nw = [1, 2, 4, 6, 8, 10, 12, 14, 16];
```

while for the **Xeon PHI** we had:

```
tree_no = 2400; depthmax = 5; threshold = 800;
randmax = 10; gen_no = 20; err = -1;
nw = [1, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120].
```

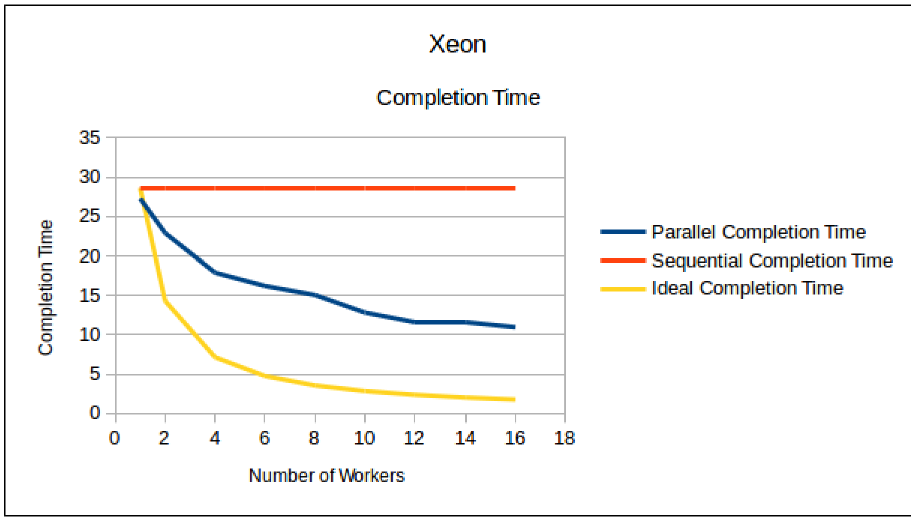
We passed `input_cos(x)-pow(x,3)10-3.txt` as input file, hence we had 1000 input pairs $(x, f(x))$. As previously stated, the smaller number of trees on the Xeon PHI is due to the lower clock speed. Since our program is very flops-intensive and the Xeon PHI has 120 ALU, we did not test with more than 120 workers for avoiding not to have a true parallelism. Both in the sequential and parallel case we performed 12 runs of the program, we took out the two most outstanding completion time outliers and then we computed the average of the remaining.

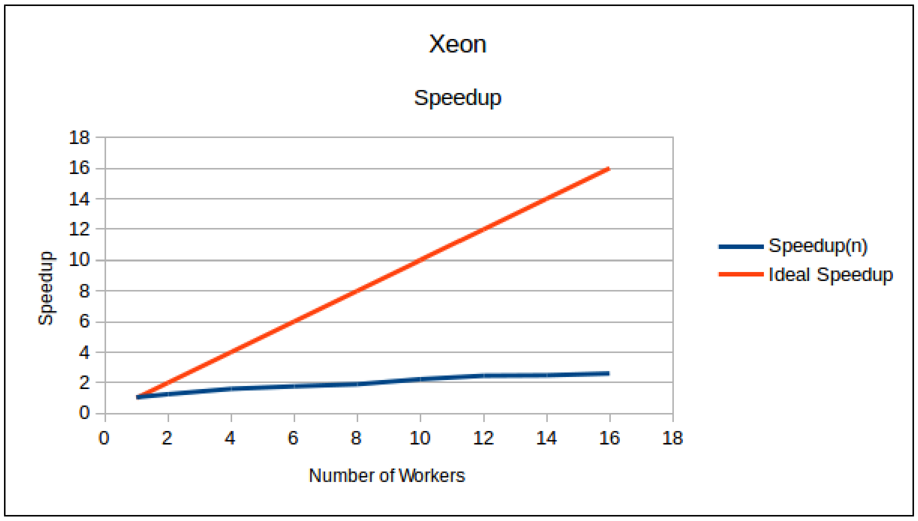
6.1 Xeon

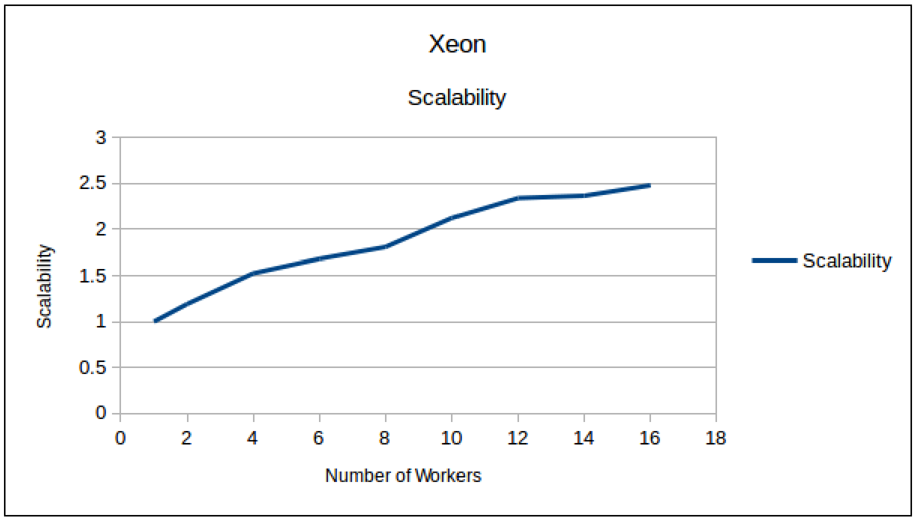
Has we can see from the graphs, the completion time is not as good as it was supposed to be. Indeed, in the formula we gave for a realistic parallel completion time we left the variable "*overhead*" to capture the fact that we had still to exactly quantify how much time it is required to dispatch the workload to the various workers. Considering for instance $T_{par(2)}$ we see that now we get a real value of $22.89348s$, implying that we should have $overhead = 22.89348s - 15.744102s = 7.149378s$. This seems to suggest that the overhead is still to high and would explain why we got such a poor speedup (as showed in the graphs). Another possible explanation could be that the compiler is able to perform some optimizations for the sequential program that are not possible with the parallel one, although we do not think this is the case since the completion time for one worker $T_{par(1)}$ is more or less the same of the sequential one.

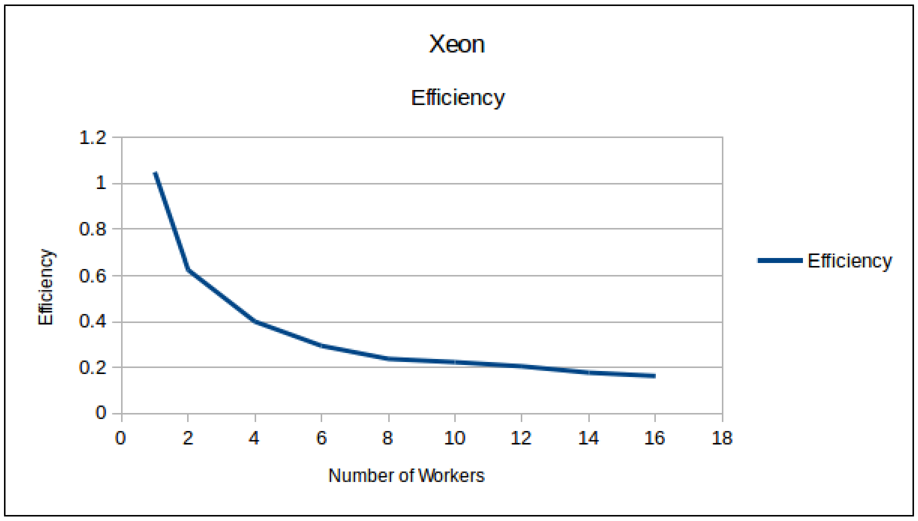
6.2 Xeon PHI

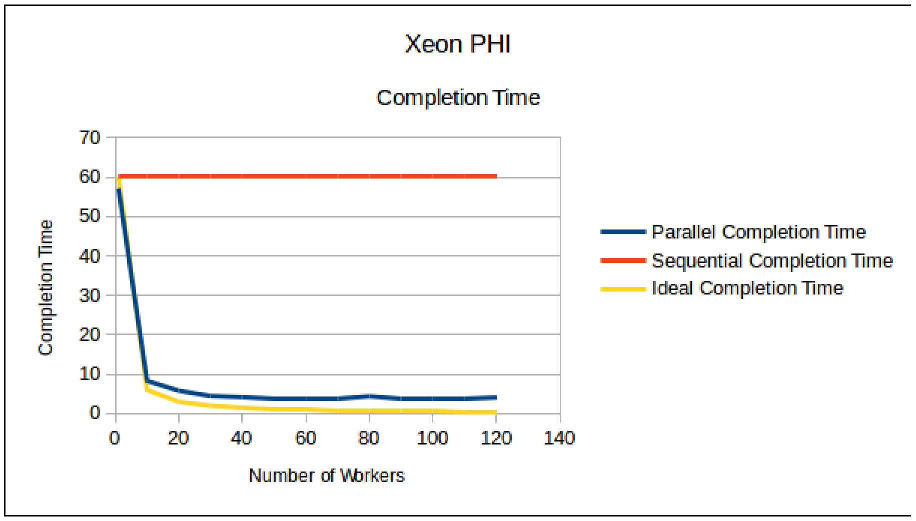
Even in this case we experience the same issue as for the Xeon. Here we got better results in terms of speedup, passing from a completion time of $\ 58s$ using 1 worker to $\ 4s$ with 10 workers. The problem is that the poor scalability: the performances go worse with the growing of the number of workers we use. Due to this behavior (high boost with fewer workers, little improvements with more of them) we believe that the input size has really been an issue and it would be interesting to run the tests again with a larger pool of trees. We chose such a pool size for our measurements because a larger one would require much more time to complete all the tests.

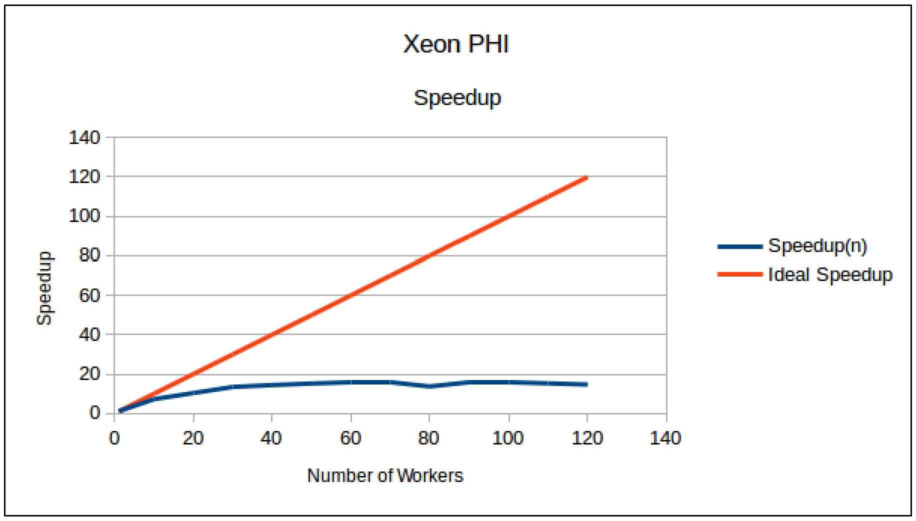

Xeon
Completion Time
Completion Time
Number of Workers
Parallel Completion Time
Sequential Completion Time
Ideal Completion Time

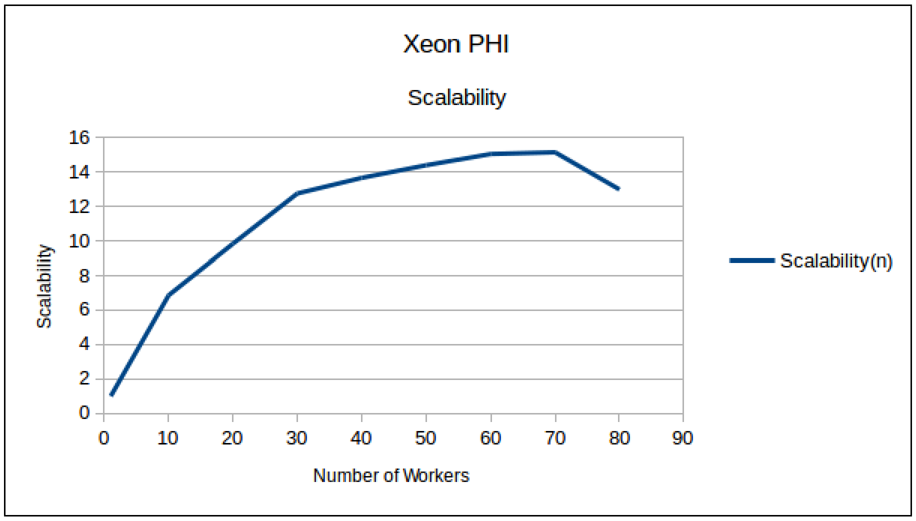

Xeon
Speedup
Speedup
Number of Workers
Speedup(n)
Ideal Speedup

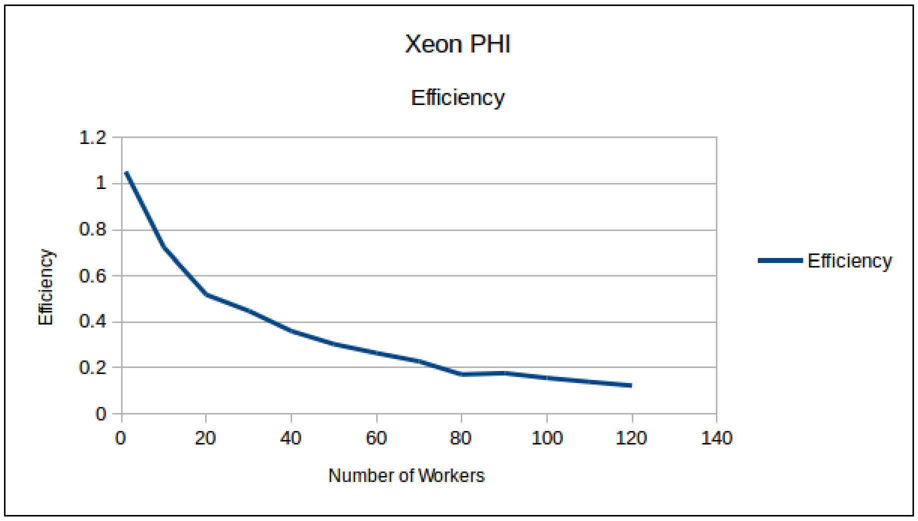

Xeon
Scalability
Scalability
Number of Workers
Scalability


Xeon
Efficiency
Efficiency
Number of Workers
Efficiency
1.2
1
0.8
0.6
0.4
0.2
0
0
2
4
6
8
10
12
14
16
18


Xeon PHI
Completion Time
Completion Time
Number of Workers
Parallel Completion Time
Sequential Completion Time
Ideal Completion Time
70
60
50
40
30
20
10
0
0
20
40
60
80
100
120
140


Xeon PHI
Speedup
Speedup
Number of Workers
Speedup(n)
Ideal Speedup
140
120
100
80
60
40
20
0
0
20
40
60
80
100
120
140


Xeon PHI
Scalability
Scalability
16
14
12
10
8
6
4
2
0
0
10
20
30
40
50
60
70
80
90
Number of Workers
Scalability(n)


Xeon PHI
Efficiency
Efficiency
1.2
1
0.8
0.6
0.4
0.2
0
0
20
40
60
80
100
120
140
Number of Workers
Efficiency