Vrije Universiteit Amsterdam Name: Vasileios Christou
Course: Parallel Programming Practical Student ID: vch202

1. Introduction

In the current study, the challenges of the parallelization of a Rubik's cube solver using the Ibis library in Java and the results of the parallel program are presented, in terms of measured times, speedups and efficiencies, compared to the sequential algorithm. In section 2, we discuss the changes that have to be made to the sequential algorithm in order to make it parallel, including the implementation of a master – workers scheme and the communication between the different processes using Ibis SendPorts and ReceivePorts in order to pass requests, jobs and solutions from the master to the workers and vice versa. In section 3, we present the results of the parallel algorithm for two different initial cubes, one of size 3 and 11 twists with a 7-bound solution and one of size 3 and 13 twists with a 9-bound solution, to show the dependence of performance on the problem size. In section 4, we describe a bonus implementation, which includes work stealing between workers, and we compare it to the basic implementation. Finally, in section 5, we summarize the conclusions that we have been led to by this study.

2. Implementation

The first change to the sequential algorithm, in order to parallelize it using Ibis, is to create the Ibises. This happens inside the *joinAndElectMaster* method, where the program waits until all the processes (Ibises) have joined the pool with the *WaitUntilPoolClosed* method and then utilizes Ibis' election mechanism to elect a master, in order to implement the master-workers scheme. Then, after initializing the input cube and the cube cache, the master and the workers start their corresponding procedures.

Regarding the master process, it creates an Ibis *ReceivePort* for incoming job requests and solutions from workers, and a separate *SendPort* for each worker, to send out jobs and orders messages. It then starts searching for a solution using the sequential IDA* algorithm until a specified bound. It is important that this bound is set to a value that is neither too high, letting worker's waiting for a long time, nor too low, introducing unnecessary communication for jobs that the master could handle alone faster. After testing different values, 4 proved to be the appropriate bound for the master. When this bound is reached and searched, if no solution has been found, the bound is increased for the next iteration and the master starts building a queue of cubes – jobs to share with the workers. This queue is populated by the "children" of all cubes which have been twisted to the bound that the master can search. At the same time, the master checks for job requests using the *sendJob* method. Inside this method, the master checks for any messages with the *poll* method of its *ReceivePort*. It should be noted that *RECEIVE_POLL* has been enabled for this port in the *PortType* definition. If there is a request message, it identifies the sender, removes a chunk of cubes from the queue and sends the job to the requestor. Different job sizes were tested, and 9 had the best performance, so an array of 9 cubes is sent in each job message. When the queue is complete, the master starts searching for a solution too, by taking cubes from the queue one by one and checking for job requests at the same time, until the queue is empty. To make this possible, the *sendJob* method is also used inside the recursive *solutions* method, but only by the master ibis. A smaller job size of 3 cubes is also set, in case there are not enough cubes in the queue, so that the master can share the last chunk of work with some

workers. When the queue is emptied, the master calls the *sendOrders* method, where it waits for the next request from each worker and answers with a "report" string, informing the workers that they must report their number of solutions for the current bound. The master waits until all workers have sent their last request before sending the orders and then sends out all the messages, so that none of the workers moves on to send its solutions and the messages get mixed. Then the master receives the solutions from all workers, sums them up and calls the *sendSecondOrders* method to send out the second type of orders, which is either a "stop" string if a solution has been found or a "continue" string if there was no solution found for the current bound. Finally, the master either prints the results or moves to the next iteration, incrementing the current bound.

As far as the worker process is concerned, a *SendPort* is created and connected to the master *ReceivePort*, so that requests can be sent out, and a *ReceivePort* is created so that the master can connect his corresponding *SendPort* and reply to the requests with jobs. Then the main loop is entered, with the worker requesting a job. When a message arrives, it is handled as an array of cubes, by class casting, and the worker starts searching the included cubes up to the current bound, before requesting again. In case the class cast throws an exception, it gets caught with a try – catch block, and is handled as a string, since it is the "report" orders, and means that the queue is empty, and the worker should report its solutions. The *checkOrders* method is called and the worker sends its solutions and receive the next orders mentioned above. If it is a "continue", the loop is repeated, else the procedure ends.

The program ends with all ibises calling the *end* method to leave the pool, so that it can be properly closed.

3. Performance Analysis

The results of the master – workers implementation described in the previous section are presented in terms of measured times, speedups, and efficiencies, compared to the sequential program, for different setups and problem sizes. Speedups and efficiencies were computed using the following formulas:

$$speedup = \frac{measured\ time\ of\ the\ sequential\ program}{measured\ time\ of\ the\ parallel\ program\ with\ p\ processes}$$

$$efficiency = \frac{speedup}{p}\ 100\%$$

where p is the number of processes (nodes x cores).

The parallel algorithm was tested on all the combinations of 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node for 2 different initial cubes. The first cube is one of size 3 and a 7-bound solution. This cube results from the default argument values. The second one is also a cube of size 3, but with a 9-bound solution, which resulted from using the value of 3 for the seed argument (--seed 3). Regarding the measurements, the total time from the moment the master starts searching the first bound until a solution has been found is measured. All simulations were run 3 times for

each setup and the average measured time was considered. The results are given in three graph categories for each cube case, one for each type of metric, with the number of processes on the X axis and the measurements on the Y axis.

Regarding the first cube, the metrics are presented in **Figures 1-3**, and they are not satisfactory. To begin with, the changes to the algorithm in order to parallelize it, such as the introduction of ports, leads to the single process case of the parallel program resulting in slightly higher measured time than the sequential one. As the number of processes increases, there is no significant decrease in measured time, with this decrease diminishing further at 16 total cores and on, resulting in sublinear speedups and low efficiencies. The small problem size includes a low number of computations, resulting in a low computation to communication ratio. Especially in the cases of 16 cores per node, where the amount of communication is much higher, the efficiency drops significantly, down to only 4.2% for 128 processes. A notable fact is that the 16 cores per node setups perform worse than the setups with the same processes, but 4 cores per node. This could be due to the high amount of data on a single node, slowing it down, even though the communication between local processes should be faster. At 128 cores, there is a slight rise of the speedup curve, but there is a chance that this is only due to randomness, since the measured times of the small problem size presented significant variation between runs. In general, it is evident that the small problem size does not favor scaling, putting a limit to the number of processes that can be used to efficiently parallelize the solver. The numbers for the metrics of the current case are given in **Table 1**.

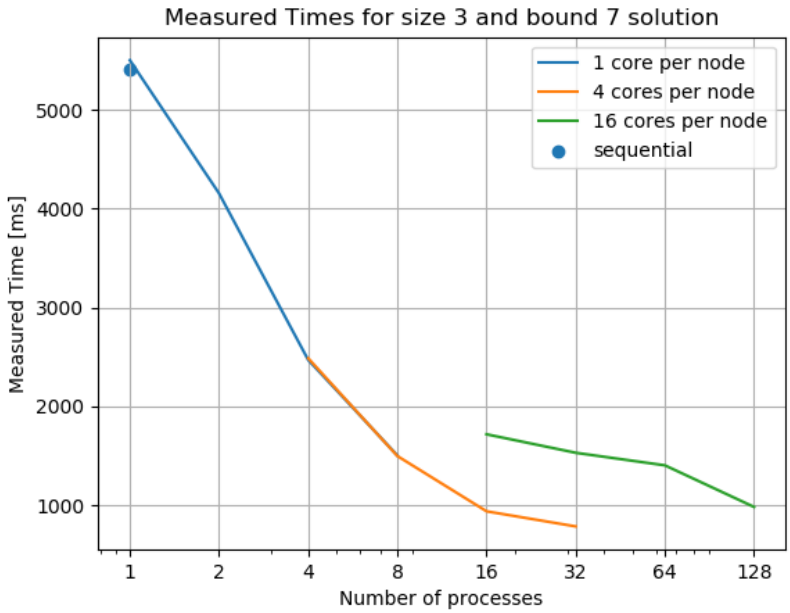

Figure 1. Measured Times for a cube of size 3 with a 7-bound solution, for all setup combinations of 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

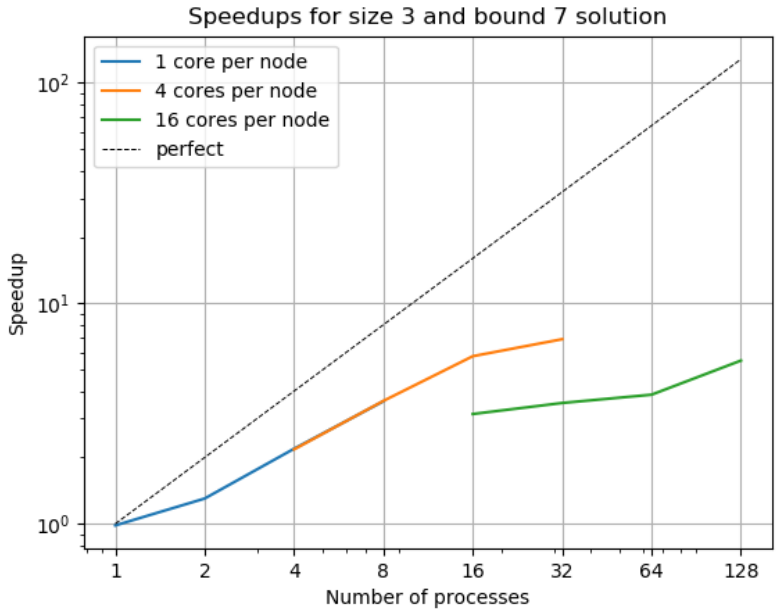

Figure 2. Speedups, compared to the sequential algorithm, for a cube of size 3 with a 7-bound solution, for all setup combinations of 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

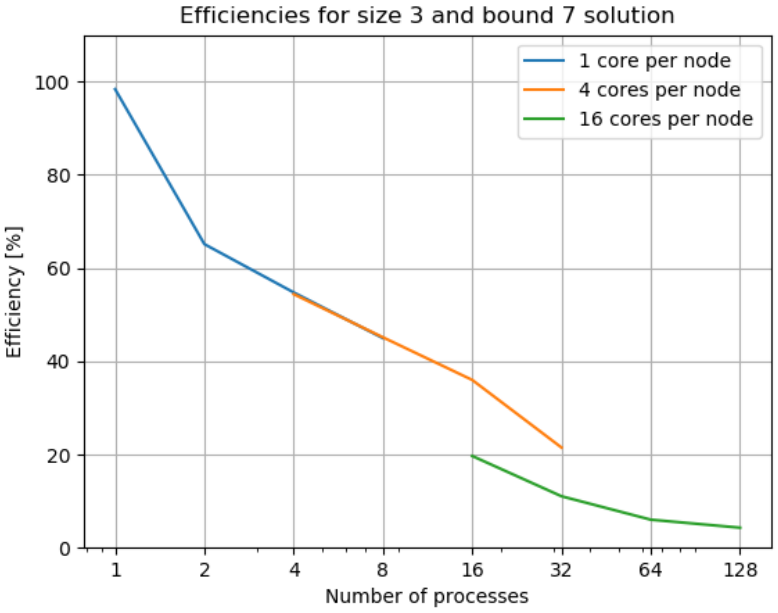

Figure 3. Efficiencies, compared to the sequential algorithm, for a cube of size 3 with a 7-bound solution, for all setup combinations of 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

Table 1. Metrics for a cube of size 3 with a 7-bound solution, for all setup combinations of 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

Processes (nodes x cores)	Measured Time (ms)	Speedup	Efficiency (%)
Sequential	5413	1	100
1 (1x1)	5503	0.98	98.3
2 (2x1)	4156	1.3	65.1
4 (4x1)	2471	2.19	54.7
8 (8x1)	1505	3.59	44.9
4 (1x4)	2488	2.17	54.3
8 (2x4)	1499	3.61	45.1
16 (4x4)	940	5.75	35.9
32 (8x4)	787	6.87	21.4
16 (1x16)	1719	3.14	19.6
32 (2x16)	1532	3.53	11
64 (4x16)	1405	3.85	6
128 (8x16)	984	5.5	4.2

Moving on to the second input cube, the higher bound translates to deeper searching down the tree, thus a larger problem size. Looking on **Figures 4-6**, one can see that the dependence of the performance of the parallel algorithm on the problem size is high. This time, the decrease in measured times is significant, resulting to linear speedups, which are close to perfect for the 1 and 4 cores per node setups. This is due to the increase in the number of computations required by each process. However, there is the same amount of communication with the previous case, since the cube size is the same, resulting in the same number of cubes in the master's queue.

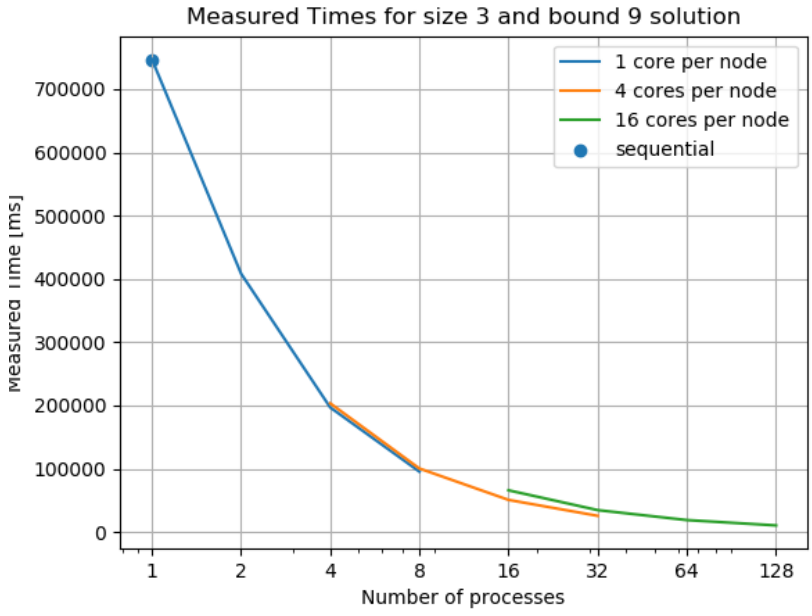

Figure 4. Measured Times for a cube of size 3 with a 9-bound solution, for all setup combinations of 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

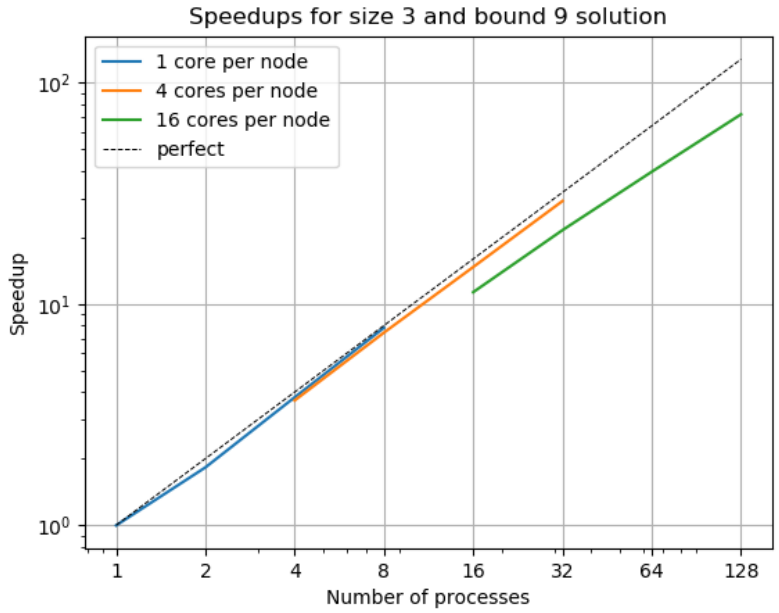

Figure 5. Speedups, compared to the sequential algorithm, for a cube of size 3 with a 9-bound solution, for all setup combinations of 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

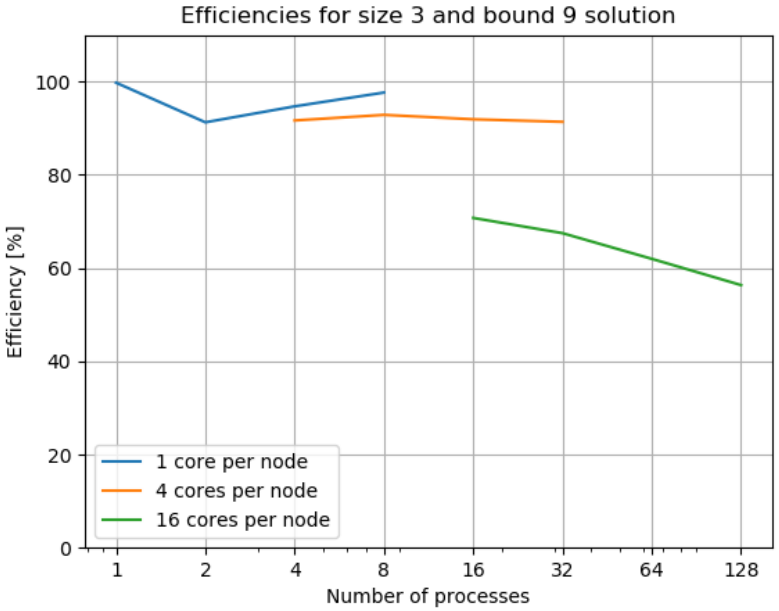

Figure 6. Efficiencies, compared to the sequential algorithm, for a cube of size 3 with a 9-bound solution, for all setup combinations of 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

Therefore, the computation to communication ratio is higher, favoring parallelization. Performance presents a step decrease at the 16 cores per node setups, again probably due to the high amount of data in the same processor, but is still in much higher levels compared to the previous case, over 50% for 128 total cores. In general, the algorithm seems to perform better in setups with less cores per node, compared to the corresponding ones with more cores per node, a fact that is surprising, since communication inside a node is usually faster than travelling through the network. Another noteworthy fact is that the cases of 4 and 8 nodes with 1 core per node present a drastic increase in performance, almost touching the perfect curve, but still not reaching 100% efficiency. This could be due to the perfect balancing of work and communication for the current problem, in this specific number of processes, since after this point the increased communication leads to the performance dropping. It would be interesting to see the results in setups with more than 8 nodes and one core per node. On the other hand, the mediocre performance of the 2 processes case is due to the master being one of the workers. Since the master needs to poll its *ReceivePort* in each recursive call of the *solutions* method, it is working slower than a normal worker. Overall, it is clear that as the problem size increases, the computation to communication ratio rises, and the scalability of the parallel program increases too. However, if more processes were to be added, the metrics would probably drop to unacceptable levels, requiring an even larger problem size. The numbers for the metrics of this case are given in **Table 2**.

Table 2. Metrics for a cube of size 3 with a 9-bound solution, for all setup combinations of 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

Processes (nodes x cores)	Measured Time (ms)	Speedup	Efficiency (%)
Sequential	745341	1	100
1 (1x1)	747237	0.99	99.7
2 (2x1)	408384	1.82	91.2
4 (4x1)	196810	3.78	94.6
8 (8x1)	95422	7.81	97.6
4 (1x4)	203311	3.66	91.6
8 (2x4)	100342	7.42	92.8
16 (4x4)	50688	14.7	91.9
32 (8x4)	25495	29.23	91.3
16 (1x16)	65845	11.31	70.7
32 (2x16)	34518	21.59	67.4
64 (4x16)	18793	39.66	61.9
128 (8x16)	10336	72.11	56.3

4. Work Stealing (Bonus)

Another implementation has been also tested, where work stealing is used as a load balancing mechanism. In this approach, once for each bound, when a worker finishes its tasks and there are no more cubes in the master's queue, before reporting its solutions, the worker calls the *stealWork* method and tries to steal work from another random worker. There was a number of

challenges implementing work stealing, that required specific changes to the basic master – workers algorithm. First of all, since the memory is distributed in different DAS5 nodes, the thief has no access to the victim's tasks, as in shared memory work stealing. Therefore, extra communication is required, with each worker maintaining a new *ReceivePort*, and several new *SendPorts*, one for each other worker. The thief sends a steal request to the victim, asking for job, just like sending a job request to the master. In order for this to work, all workers need to maintain their personal queue of cubes, in which they add any cubes received by the master. They must also poll their new *ReceivePorts* for requests, using the *sendJob* method. Different polling frequencies were tested, and it was decided that each worker should check for steal requests each time it finishes searching a whole cube in its queue. Polling between cube children, inside the *solutions* method, was also tried, but its performance was worse, probably due to the increased unnecessary polling.

If the victim receives a steal request when it is still solving its cubes, it sends half of the remaining cubes in its queue to the requestor. When the victim's reply is received by the thief, it is checked. In case it is a list of cubes, the victim had available cubes to share, and these cubes are handled as a normal job, balancing the workload between the two Ibises. However, there is a chance that the victim has no cubes in its queue, or that it is even sending a steal request back to the thief due to simultaneous stealing. Therefore, if the message is empty, or a string, the stealing operation is aborted, and the thief moves on to check the master's orders.

Additionally, a change to the mechanism of sending orders and receiving solutions was necessary, since the previous one was blocking the workers after sending their last request, until all of them have communicated with the master. In the basic implementation, there was no problem with this blocking since the worker had nothing to do between sending their last request and then their solutions. In work stealing, however, workers must continue polling their *ReceivePorts*, so that other, slower workers, do not get blocked when they send their steal requests. Therefore, the receiving of solutions has been integrated to the sending of "report" orders in the master Ibis. In the new version of the *sendOrders* method, the master keeps receiving messages and sending out "report" orders and checks if the received message is an integer or a string, in order to handle it as a solutions message or a request message accordingly. If it is a solutions message, the received integer is added to the total number of solutions for the current bound. There is also an *Arraylist* to keep track of the workers that have sent their solutions, so when an integer is received, the sender is added to the list. When every worker has reported their solutions, all workers are informed about stopping or continuing working as in the basic master – workers algorithm. As for the workers, in the *checkOrders* method, after receiving the "report" orders and sending their solutions to the master, they keep polling their *ReceivePorts* to reply to steal requests, until they receive the second type of orders, and then either stop or continue to the next bound.

The work stealing implementation was compared to the basic master – workers one for different problem sizes and the results are presented below. The same setups and approaches regarding measurements as in the previous section were chosen. The first case concerns the size 3 cube with the 9-bound solution of the previous experiment and the second one, a cube of size 5 and a 7-bound solution. The comparison of the two algorithms for the first cube is presented in the left-side graphs of **Figure 7** and the metrics for the work stealing approach are given in **Table 3**. It is evident that the performance of the work stealing approach is, in general, equal to this of the

basic algorithm. This is probably due to the workload being already sufficiently balanced, since workers receive jobs of the same size. Additionally, the algorithm does not store or share between processes any previously checked cube states, in order to use them for pruning. Therefore, all branches are of the same height, and only the difference in job receiving times and core speeds may lead to a process finishing its tasks earlier and manage to steal cubes from another one. Even in this case, some testing showed that stealing happened mainly at bounds 8 and 9, and the size of the stolen jobs was small – mostly 1 or 2 cubes – which at best compensates for the additional communication required for distributed memory work stealing. A notable fact is that in the last 2 setups, the work stealing efficiency is slightly higher than the basic one. This shows that work stealing is more efficient in cases of many workers, probably due to more successful steals happening, since there is a higher chance that a victim with available jobs is found, and also that thieves pick different victims. It is expected that in cubes of larger size, where the lower bounds also require significant searching time and thus can lead to greater differences between processes, work stealing will take place more often, increasing the performance of this approach. Additionally, in higher bound searching, where even 1 or 2 cubes will need a significant amount of time to be searched, work stealing will be more efficient than the basic master – workers algorithm.

Table 3. Metrics of the work stealing algorithm for a cube of size 3 with a 9-bound solution, for all setup combinations of 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

Processes (nodes x cores)	Measured Time (ms)	Speedup	Efficiency (%)
Sequential	745341	1	100
1 (1x1)	748376	0.995945	99.59446
2 (2x1)	409423	1.820467	91.02334
4 (4x1)	194668	3.82878	95.71951
8 (8x1)	96547	7.719981	96.49976
4 (1x4)	207317	3.595176	89.87939
8 (2x4)	102253	7.289185	91.11481
16 (4x4)	51147	14.57253	91.07829
32 (8x4)	26061	28.59986	89.37457
16 (1x16)	67538	11.03588	68.97423
32 (2x16)	34967	21.31555	66.61111
64 (4x16)	17605	42.33689	66.1514
128 (8x16)	9789	76.14067	59.4849

To verify the hypothesis regarding the higher cube sizes, we have also tested both approaches on a cube of size 5. However, this cube has a 7-bound solution, since higher bounds could not be searched within the time limit in low number of processes setups. Looking at the right-side graphs of **Figure 7**, it is clear that our expectations were correct, since the superiority of the work stealing algorithm is notable. Work stealing shines mostly in cases of many workers, due to the large pool of possible victims, as mentioned earlier. Even though the difference in measured times is not clearly visible due to large numbers, in 16 cores per node setups, the efficiency difference between the two approaches is clear. In larger cubes, the divergence between the moments each

process finishes working is higher, a fact that increases the load imbalance and results in a higher frequency of successful stealing. However, since it is a 7-bound solution, both metrics are worse, compared to the previous case. In a size 5 cube with a 9-bound solution, efficiencies would rise and the difference between the two algorithms would probably be even greater. The numbers for the metrics of the two approaches are given in **Tables 4** and **5**.

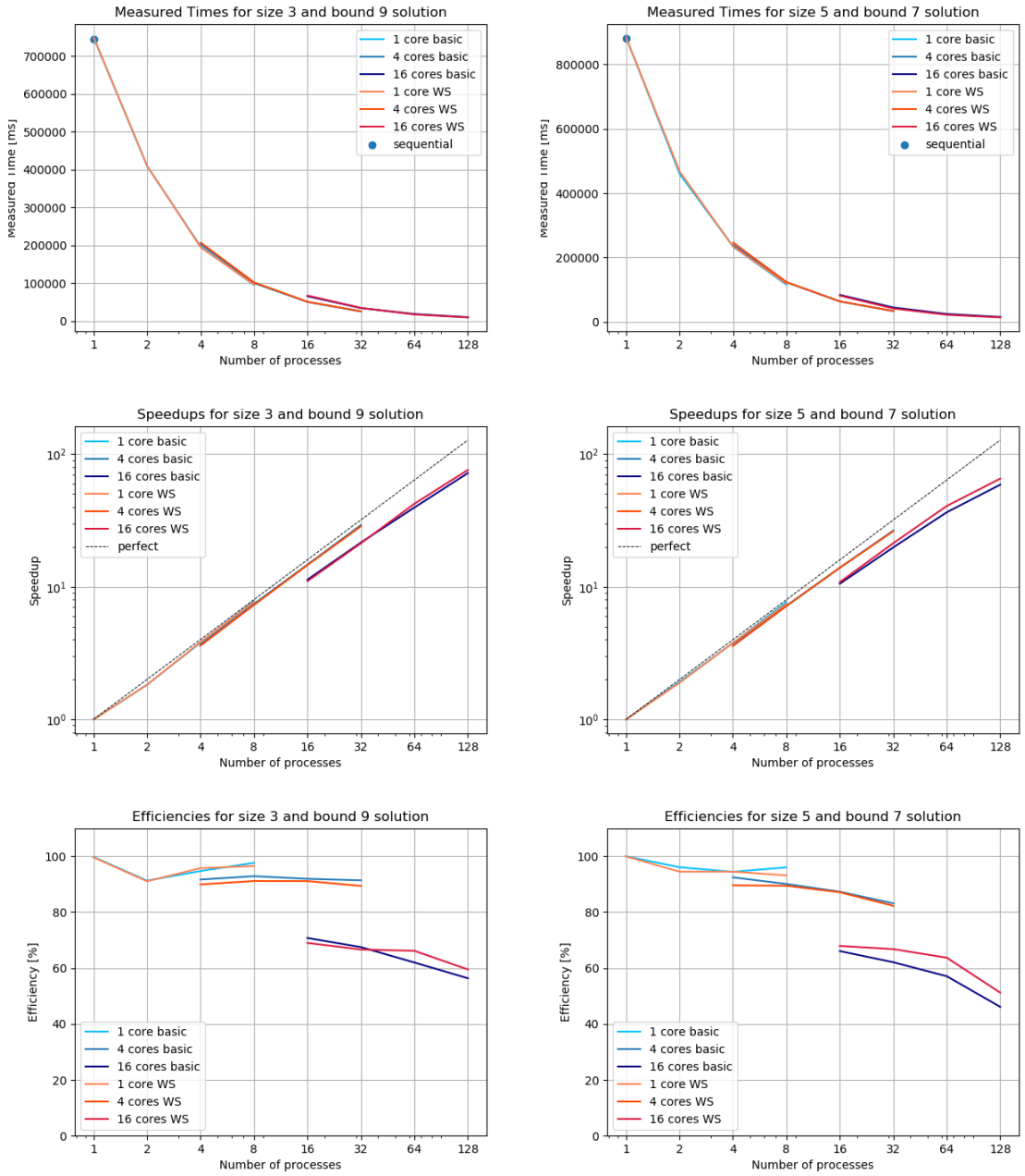

Figure 7. Comparison of the basic master - workers algorithm and the work stealing (WS) algorithm, for all setup combinations of 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node. On the left side, the metrics for the cube of size 3 and a 9-bound solution are given. On the right side, the metrics for the cube of size 5 and a 7-bound solution are given. The top graphs depict measured times, the middle graphs depict speedups, compared to the sequential algorithm, and the bottom graphs depict efficiencies.

Table 4. Metrics of the basic master – workers algorithm for a cube of size 5 with a 7-bound solution, for all setup combinations of 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

Processes (nodes x cores)	Measured Time (ms)	Speedup	Efficiency (%)
Sequential	882569	1	100
1 (1x1)	882737	0.99981	99.98097
2 (2x1)	459369	1.921264	96.06319
4 (4x1)	233732	3.775987	94.39968
8 (8x1)	114896	7.68146	96.01825
4 (1x4)	238716	3.697151	92.42876
8 (2x4)	122487	7.20541	90.06762
16 (4x4)	63188	13.96735	87.29595
32 (8x4)	33177	26.60183	83.13073
16 (1x16)	83505	10.56906	66.0566
32 (2x16)	44434	19.86247	62.07022
64 (4x16)	24166	36.5211	57.06423
128 (8x16)	14955	59.01498	46.10545

Table 5. Metrics of the work stealing algorithm for a cube of size 5 with a 7-bound solution, for all setup combinations of 1, 2, 4, 8 DAS5 nodes and 1, 4, 16 cores per node.

Processes (nodes x cores)	Measured Time (ms)	Speedup	Efficiency (%)
Sequential	882569	1	100
1 (1x1)	883049	0.999456	99.94564
2 (2x1)	467130	1.889343	94.46717
4 (4x1)	233565	3.778687	94.46717
8 (8x1)	118439	7.451676	93.14594
4 (1x4)	246331	3.582858	89.57145
8 (2x4)	123364	7.154186	89.42732
16 (4x4)	63334	13.93515	87.09471
32 (8x4)	33536	26.31706	82.24082
16 (1x16)	81242	10.86346	67.89661
32 (2x16)	41314	21.36247	66.75771
64 (4x16)	21651	40.76343	63.69286
128 (8x16)	13469	65.52595	51.19215

5. Conclusions

The parallelization of the Rubik's cube solver using the master – workers approach leads to a number of conclusions, regarding the efficiency of the different approaches, on different problem sizes and core setups. To begin with, it is evident that a fixed problem size can only be efficiently parallelized to a specific degree, a fact that comes in agreement with Amdahl's Law, which states that for a fixed problem size, increasing the degree of parallelization will never reduce the measured time to less than the part of the algorithm that cannot be parallelized. A high computation to communication ratio is a requirement for efficient parallelization. After a number of processes, communication increases for the same number of computations and performance starts diminishing. Therefore, we need to consider Gustafson's Law and move to bigger problem sizes in order to obtain higher performance as the degree of parallelization increases. Additionally, it seems that the algorithm does not favor cases with 16 cores per node. In these cases, a step decrease in efficiency was observed, which could be due to the large amount of data (e.g., ports) within a single processor, slowing down the processor.

As far as work stealing is concerned, the approach proved to be more efficient as the cube size grows. In larger cubes, all bounds for which workers are responsible require a significant amount of searching. Therefore, the difference between end times of faster and slower processes is greater, which results in larger job steals. Additionally, successful stealing is more frequent, compared to the case of a smaller cube, and happens in all bounds that workers work on. In summary, workload in larger cubes is more imbalanced and work stealing provides the required load balancing. However, it should be noted that there was notable variation between runs of the same setup for the case of the basic master – workers program, with "good" runs presenting a slightly better performance than the work stealing approach, but "bad" runs being significantly slower. On the other hand, the work stealing algorithm presented significantly lower variation between same setup runs, which is probably due to balancing the workload in the unbalanced "bad" runs. Finally, work stealing's performance is mostly notable in setups with a lot of processes. This fact is due to the increased chance that a thief finds a unique random victim, that has available cubes to share, and thus steals successfully. Unfortunately, it was not possible to test a size 5, 9-bound solution cube withing the time limit. It is expected that the results in such a case would be even more satisfactory, since even more stealing will take place in higher bounds.