Multilevel Scheduling Prototype plus LB4OMP

Master's Project

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
High Performance Computing

Advisor: Prof. Dr. Florina Ciorba
Supervisor: Dr. Ahmed Eleliemy
Supervisor: Jonas H. Müller Korndörfer

Gian-Andrea Wetten
gian-andrea.wetten@stud.unibas.ch

Abstract

The massive horizontal and vertical scaling of supercomputers in recent times gives rise to applications that induce load imbalance on multiple layers. Multilevel scheduling has become an increasingly compelling method of addressing this issue. In this work we asses the implications on performance of combining thread-level scheduling with scheduling at the batch and application level using a two-phase approach of exploration and experimentation. The results show that introducing a second layer of scheduling with the MLS prototype leads to very limited improvements in overall performance at a small scale. We show that this was mainly due to the low degree of process-level load imbalance which reduces the effectiveness of the MLS prototype.

Contents

1 Introduction

When wanting to optimize the performance of parallel and distributed programs achieving an even workload distribution across all processing units (PU) is very important. This is why scheduling has been an popular research topic in high performance computing (HPC) over the last 30 years. Coping with irregularities in the workload of a computational domain, however, is a very complex task. It is often crucial to find the appropriate trade-off between the performance gain of an optimal distribution and its induced overhead.

Parallelism in HPC has been growing at a substantial rate across multiple levels. As a consequence it becomes increasingly crucial to analyze the performance impact of reducing load imbalance by using scheduling procedures on each individual level. Ali et. al. [13] have shown that combining multiple levels of scheduling can lead to an even more significant reduction of load imbalance.

The goal of this work is to analyze the performance impact of combining two different kinds scheduling levels. It is yet unclear how thread-level scheduling affects the application performance when we use the MLS prototype[7]. We want to examine this relation between thread-level and batch-level scheduling and measure its overall implications on performance.

In shared-memory environments OpenMP is widely considered the standard method of parallelism. The standard implementation offers us 3 methods of scheduling, namely *static*, *dynamic* and *guided* scheduling. However it has been shown[4] that these limited options are often not producing optimal results. Korndörfer et al.[11] showed that we can reach better performance gains using different kinds of scheduling algorithms. In their work they presented a library called LB4OMP which implements several known dynamic dynamic load balancing algorithms. In addition to that a further extension of the library, called Auto4OMP, provides the user with options for automated scheduling algorithm. This automated selection procedure has shown promising results and is built on the principle of not requiring any user input or profiling ahead of the execution of the loops. For this reason we are including all 4 automated scheduling methods in our experiments as well as 6 additional baseline algorithms implemented in LB4OMP.

The MLS prototype[7] has been implemented with the idea of applications sharing their idle computing resource once they reach an idle state. This is done by releasing resources used by MPI ranks as soon as the they are done with their calculations, i.e. they reach the *MPI_Finalize* function. Therefore we can minimize idle time of nodes, leading to a reduction in the overall execution time of a batch.

Section two introduces the tools and methods used in this work as well as the experimental setup. In section three we present the evaluation procedures and the results, followed by a discussion of them in section four. We close the report with a brief conclusion and opportunities for future work.

2 Related Work

LB4OMP[11] was introduced to address a lack of scheduling options in the OpenMP standard. Korndörfer et al. implemented 14 dynamic scheduling algorithms in their work and performed an analysis of the performance gain possible with each option. They showed that the presented techniques outperform the ones from the OpenMP standard on multiple application-systems pairs. This work is using their research infrastructure LB4OMP to perform thread-level scheduling.

The MLS prototype[7] is used for application-level and batch-level scheduling. Ahmed Eleliemy demonstrated that scheduling on an pplication level never completely eliminates load imbalance and therefore it is still possible to improve performance using batch-level scheduling. Thus he illustrates that a coordination between those two levels leads to reduced idle time.

Ali et al. combined thread-level and process-level scheduling. They used six scheduling techniques at the thread-level using an extended version of the GNU OpenMP runtime library called eLaPeSD[4] and 11 techniques at the process level with DLS4LB[3]. Their results showed an impressive improvement in application performance of up to 21%.

3 Methods

To appropriately evaluate the performance impact of multi-level scheduling we decided on using a two-phase approach. The first step is comprised of an exploration of hybrid OpenMP and MPI applications. Our main goal of this exploration is to find applications which are suitable candidate for our experiments. This is done using the measuring capabilities built in the LB4OMP library. The second phase of our evaluation consists of running experiments with several different scheduling techniques for OpenMP loop. These experiments are run once with the MLS prototype and once without. We then compare the total execution time as well as the idle time on nodes of these runs for each OpenMP scheduling technique that we used.

3.1 LB4OMP

First we need to set the OpenMP schedule clause for the loops that we would like measure to *runtime*. As with the standard OpenMP implementation we choose a schedule for those loops by exporting the *OMP_SCHEDULE* environment variable. The library then provides us with the loop execution time of each thread per time-step upon setting an environment variable called *KMP_TIME_LOOPS* to the desired location of the output file. It also records the number of iterations and the location of the loop. With this information we can calculate several metrics for load imbalance which then in turn can be used to make an informed selection on the applications that we want to include in our experiments.

The 10 scheduling techniques used in the experiments have been chosen from all three domains. We use straightforward scheduling (*static*), three dynamic non-adaptive algorithms, where two are from the OpenMP standard (*guided,dynamic*) and one is a practical variant of factoring implemented in LB4OMP (*fac2a*). As a dynamic and adaptive technique we chose adaptive factoring(*af_a*). The automated methods are comprised of RandomSel,ExhaustiveSel,BinarySel and Expertsel (*auto,2-auto,5*).

3.2 MLS Prototype

The MLS prototype is based on a custom version of the Slurm workload manager[18]. For our experiments we installed this custom version on five nodes which have been segregated from the miniHPC[5] cluster. One node acts as a login-node and is also the host for the Slurm controller service. The other four nodes are used for computations and run the compute node daemon *slurmd*. Applications are linked against a library which intercepts MPI function calls. This library helps us to keep track of exactly how long a program is executed in each MPI rank by using a timer between the MPI function calls. It is possible to output these rank times to a file using the *MPI_TIMES_FILE* environment variable. In addition to that we notify the Slurm controller when *MPI_FINALIZE* is called on a rank. Since we are using only one rank per node this means that the controller can free up the node for other jobs upon receiving this notification.

At the application level we use static scheduling and at the the batch level we employ a first come first serve (FCFS) strategy.

3.3 Exploration

The first step is to find a hybrid application that uses both OpenMP and MPI. Afterwards we go through the procedure of determining if the application is a suitable candidate for our experiments or not, which is shown in Figure 1.

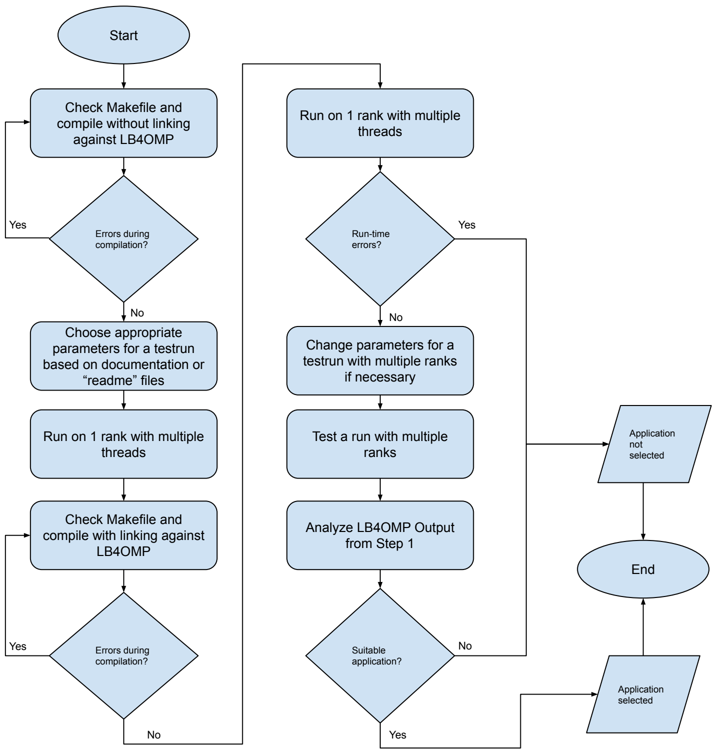
Start
Check Makefile and compile without linking against LB4OMP
Errors during compilation?
Yes
No
Choose appropriate parameters for a testrun based on documentation or "readme" files
Run on 1 rank with multiple threads
Check Makefile and compile with linking against LB4OMP
Errors during compilation?
Yes
No
Run on 1 rank with multiple threads
Run-time errors?
Yes
No
Change parameters for a testrun with multiple ranks if necessary
Test a run with multiple ranks
Analyze LB4OMP Output from Step 1
Suitable application?
No
Yes
Application not selected
Application selected
End

Figure 1: Procedure of selecting an application

Table 1 contains a listing of all the applications that have been considered for our experiments. We summed up, where possible, the metrics provided by LB4OMP. Furthermore we added two widely used metrics when dealing with load imbalances, the percent imbalance(p.i.)[6], as well as the coefficient of variation (c.o.v.)[10]. Based on these results we chose the following applications:

- **SPHYNX Evrard**[2] simulates an Evrard collapse. It has been shown by Korndörfer et. al. [11] that this application is a suitable candidate for thread-level scheduling.
- **Mandelbrot**[12] is an application to compute the Mandelbrot set. This application was also selected by them and has been proven to be interesting when dealing with load balances at the thread- and at the process-level. Because of that fact and because the application parameters can be easily tuned we chose to run this application in all workload sizes.
- **CoMD**[15] implements algorithms and workloads from the domain of molecular dynamics. It exhibits a similar behaviour to the SPEC-352.nab application used in [11] in terms of loop execution time and iterations.
- **miniVite**[8] uses the Louvain method for graph community detection. Loop execution times are moderately high and it is easy to adjust iterations/time-steps via input files or parameters. We are running this application once in the small workload (S) with a graph that is generated by the application itself. Additionally we also use it in the medium workloads ($M1, M2$) with a graph provided by the SuiteSparse Matrix Collection[16].
- **Cloverleaf**[9] is a benchmark from the world of hydrodynamics and it provides several input configurations that can be used. Furthermore it also has a good mean loop execution time to iterations ratio ($19\mu s$/iter) compared to the applications we did not select.

Table 1: List of applications that have been considered for our experiments

Application | Suite	**Description**	**Selected?**	**Reason**	**Execution command**	**Function**	μ	**T**	**Iterations**	**P.I.**	**C.O.V.**
CloverLeaf_ref-1.3 | Mantevo	A Lagrangian-Eulerian hydrodynamics benchmark	Yes		srun clover_leaf	viscosity_kernel_:52:93	0.018238	2955	961	51.12	0.57
					calc_dt_kernel_:91:132	0.018260	2955	961	50.57	0.56
					advec_cell_kernel_:105:157	0.018548	2955	961	50.56	0.56
					advec_mom_kernel_:143:173	0.018241	5910	962	51.14	0.58
					advec_cell_kernel_:194:245	0.018651	2955	963	50.05	0.55
					advec_mom_kernel_:203:234	0.018239	5910	962	51.08	0.57
TeaLeaf_ref-1.3 | Mantevo	A mini-app that solves the linear heat conduction equation on a spatially decomposed regularly grid using a 5 point stencil with implicit solvers.	No	Only a single loop worth scheduling. Loop has rather low execution time	srun tea_leaf	kernel_ppcg_inner_:104:117	0.017589	856	501	52.71	0.61
hpcg-3.1 | HPCG	An effort to create a new metric for ranking HPC systems	No	High number of iterations with low mean loop execution time	srun hpcx	ComputeSPMV_ref:59:70	0.017514	264	17576	53.04	0.61
					ComputeRestriction_ref:49:53	0.017408	264	2197	52.72	0.61
					ComputeProlongation_ref:46:51	0.017935	264	140608	52.23	0.60
					ComputeWAXPBY_ref:54:57	0.017637	830	1124864	52.54	0.60
					ComputeDotProduct_ref:56:59	0.017422	285	1124864	52.53	0.61
					ComputeDotProduct_ref:61:64	0.017962	552	1124864	52.19	0.59
miniTri | Mantevo	A triangle based data analytics miniapp	No	No hybrid implementation (MPI & OpenMP separate)							
miniMD | Mantevo	A simple proxy for the force computations in a typical molecular dynamics applications	No	High number of iterations with low mean loop execution time	srun miniMD_intel -t 20 -s 40 -n 10000	build:126:191	0.040339	501	256000	23.35	0.17
					halfneigh_threaded:296:349	0.033328	101	256000	28.06	0.22
					pbc:108:121	0.017829	500	256000	52.72	0.60
					sort:390:405	0.017882	500	59319	52.91	0.61
					borders:764:768	0.017791	3000	20	52.92	0.61
					run:201:201	0.017722	10000	256000	52.68	0.61
					run:98:98	0.017736	10000	256000	52.73	0.61
miniFE-2.2.0 | Mantevo	An approximation to an unstructured implicit finite	No	Too many iterations (time/iter)	srun miniFE.x -nx 400 -ny 400 -nz 400	cg_solve:173:173	0.028865	200	64481201	32.63	0.28
					cg_solve:158:158	0.025006	199	64481201	37.97	0.35
CoMD	A reference implementation of typical classical molecular dynamics algorithms and workloads	Yes		srun CoMD-openmp-mpi -e -i 1 -j 1 -k 1 -x 80 -y 40 -z 40	redistributeAtoms:152:155	0.020253	101	57660	46.47	0.48
					eamForce:238:245	0.021178	101	3690240	43.61	0.43
					eamForce:304:318	0.018822	101	48778	49.00	0.53
					advanceVelocity:71:80	0.018702	200	48778	50.01	0.55
					advancePosition:85:96	0.018629	100	48778	49.42	0.54
miniVite	MiniVite is a proxy app that implements a single phase of Louvain method in distributed memory for graph community detection	Yes		srun miniVite -f rgg.bin -t 1.0E-07	distLouvainMethod:477:485	0.025894	587	16777216	36.37	0.32
					distLouvainMethod:1378:1386	1.359486	587	16777216	30.51	0.19
					distLouvainMethod:463:470	0.027255	587	16777216	35.65	0.31
					distLouvainMethod:424:433	0.022742	587	16777216	41.50	0.40
					distLouvainMethod:1409:1422	0.025064	586	16777216	37.56	0.34
SPHYNX-MPI Evrard | SPHYNX	A simulation of a star colapse	Yes		srun evrard_NODLB						
Mandelbrot	A computation of the Mandelbrot set	Yes		srun mandel.o 2000000 1024 0 0 0.5						
wrf2 | SPECmpi2007	Weather prediction	No	MPI run-time errors							
pop2 | SPECmpi2007	Ocean modelling	No								
RAxML | SPECmpi2007	DNA matching	No								
GAPgeofem | SPECmpi2007	Heat Transfer using Finite Element Methods (FEM)	No								
l2wrf2 | SPECmpi2007	Weather prediction	No								
miniSMAC2D | Mantevo	Solves the finite-differenced 2D incompressible Navier-Stokes equations with Spalart-Allmaras one-equation turbulence model on a structured body conforming grid.	No	Compilation failed							
miniAero | Mantevo	MiniAero is an explicit (using RK4) unstructured finite volume code that solves the compressible Navier-Stokes equations.	No	Compilation failed							
miniAMR | Mantevo	3D stencil calculation with Adaptive Mesh Refinement (AMR).	No	MPI run-time errors							

3.4 Experimental Setup

The experiments are prepared and run in 10 batches. Each batch is associated with one of the scheduling technique mentioned in Table 3 and is comprised of 46 jobs. The jobs are divided into 4 different size categories which correspond to a fixed number of MPI ranks and determine the input parameters for the applications. These categories will from now on be noted as S, M_s, M_l, L where S is the set of jobs belonging to the small workload category and accordingly for the other categories. Furthermore $J = S \cup M1 \cup M2 \cup L$ denotes the set of all jobs. To get a representative workload we decided to divide the 46 jobs according to the information provided by the same table. This partitioning represents a down-scaled version of the ESP system benchmark[17]. Since we run each batch once with the MLS prototype and once without we get a total number of $46*10*2 = 920$ jobs. As can be imagined the order of execution actually plays an important role when we would like to involve the MLS prototype. Let us assume for example that we have 2 jobs $l1, l2 \in L$ which use all our available nodes and 2 jobs $s1, s2 \in S$, which run only on 1 node each. If we submit the jobs in the sequence $< l1, l2, s1, s2 >$ we would need to wait for $l1$ to end before we can proceed with the other jobs. With a reordering to $< l1, s1, s2, l2 >$ the Slurm controller might potentially already assign nodes to $s1$ and $s2$ while $l1$ is still processing. Because we usually do not have an optimal order of job execution in terms of workload size in the real world we chose to go with the following order: $S'_1, M1'_1, M2'_1, L'_1, S'_2, M1'_2, M2'_2, L'_2, S'_3, M1'_3, M2'_3$, where $S'_i \subset S$, and accordingly for the other categories. The detailed distribution of applications on to workload categories can be seen in Table 2.

Table 2: Number of jobs per subset of workload category

	S'_1	S'_2	S'_3	S	$M1'_1$	$M1'_2$	$M1'_3$	$M1$	$M2'_1$	$M2'_2$	$M2'_3$	$M2$	L'_1	L'_2	L
Cloverleaf	3	2	2	7											
CoMD					2	2	1	5	1	1	1	3			
miniVite	3	3	3	9	1	1	1	3	2	1	1	4			
Mandelbrot	2	2	2	6	1	1	1	3	1	2	1	4	1		1
Sphynx														1	1
Total	8	7	7	22	4	4	3	11	4	4	3	11	1	1	2

Following the evaluation of our exploration procedure we designed an experiment table as described in Table 3

Table 3: Design of the experiments

Factors	**Values**	**Properties**
Applications	Cloverleaf	N = 961 | T = 2955 | Total loops = 176 | Modified loops = 6 | Workload = S
	CoMD	N = 48,778 | T = 101 | Total loops = 15 | Modified loops = 5 | Workload = M1,M2
	miniViteGen	N = 100,000 | T = 17 |Total loops = 39 | Modified loops = 7 | Workload = S
	miniViteRgg	N = 16,777,216 | T = 518 | Total loops = 39 | Modified loops = 7 | Workload = M1,M2
	Mandelbrot	N = 262,144 | T = 200 | Total loops = 1 | Modified loops = 1 | Workload = S,M1,M2,L
	Sphynx	N = 1,000,000 | T = 200 | Total loops = 41 | Modified loops = 2 | Workload = L
Scheduling techniques	static	Straightforward parallelization
	static_steal	Extension of static scheduling
	guided	Dynamic and non-adaptive self-scheduling techniques
	dynamic	
	fac2a	
	af_a	Dynamic and adaptive self-scheduling techniques
	RandomSel	Automated DLS algorithm selection
	ExhaustiveSel	
	BinarySel	
	ExpertSel	
Workload size	S	MPI ranks = 1 | Jobs = 22
	M1	MPI ranks = 2 | Jobs = 11
	M2	MPI ranks = 3 | Jobs = 11
	L	MPI ranks = 4 | Jobs = 2
Computing nodes	miniHPC-KNL	Intel(R) Xeon Phi(TM) CPU 7210 (1 socket, 64 cores) P=64 without hyperthreading, Pinning: OMP_PLACES=cores OMP_PROC_BIND=close

4 Performance Evaluation and Results

4.1 Job execution times

Figure 2 shows us the summed up elapsed time of all jobs per LB4OMP scheduling technique. What this graph represents is the time that we would need to execute all jobs in a batch sequentially, while maintaining node-level parallelism. We can clearly see that the choice of scheduling algorithm has quite a big influence on this execution time. As expected the batch with the *static* schedule performed overall the worst. The techniques from the dynamic, non-adaptive domain all led to good results. When looking at the automated algorithm selection we see that both *RandomSel* and *BinarySel* performed quite well, while the other two methods were slightly worse. Perhaps the biggest surprise was the results that we got using *static_ steal*, which were actually the best in terms of execution time. In Figure 3 we can observe the same graph but with runs were we turned off

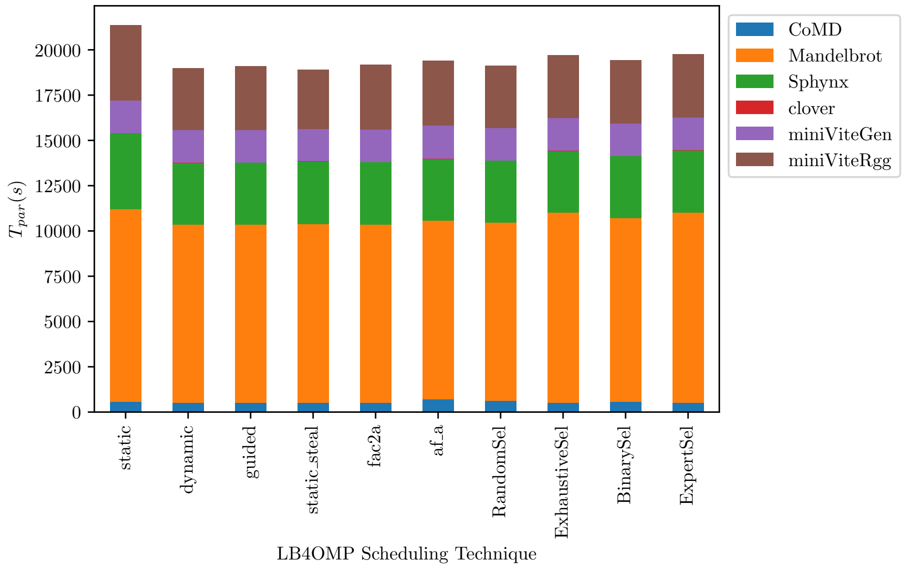

Figure 2: Total elapsed time in seconds for each application per batch with the MLS prototype

the MLS prototype. These numbers should be roughly the same as in the previous graph since we look at sequential job execution. In fact the only difference between the two should be the induced overhead from the MLS prototype. As expected we get very small deviations when comparing both graphs, with the biggest being roughly 18 seconds in the overall execution time. This tells us that the overhead is quite small.

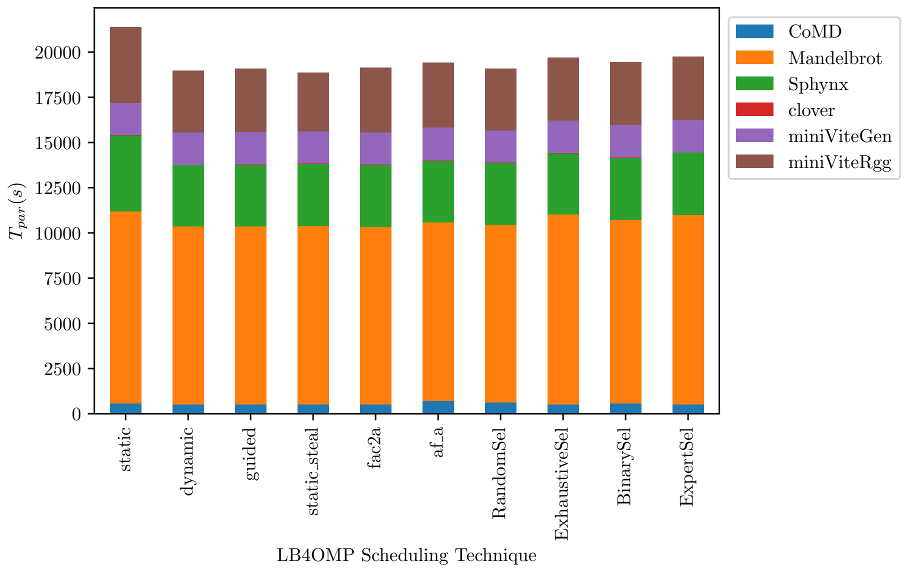

Figure 3: Total elapsed time in seconds for each application per batch without the MLS prototype

4.2 Batch execution times

The next evaluation we did was the comparison of real elapsed time per batch. For this we used the difference between the end time of the last job and the start time of the first job in each batch. This has again been calculated for our batch runs with and without the MLS prototype. We can observe the results in Figure 4. The main thing that stands out is that we see very little difference between the two runs. Looking at Figure 5 we see the reason for that. The ranks are being freed up almost at the exact same time in most applications. The only small exception is Mandelbrot where some ranks finish up to 40 second before the others. We can in fact calculate the maximal time-save possible t^* in our experiments with the following formula:

$$t^* = \sum_j \sum_i r_j^{'} - r_{ij} \tag{1}$$

Where:

- $j \in J$
- $r_j^{'}$: is the largest rank time of the job j
- R_j : is the set of rank times for job j
- $r_{ij} \in R_j$

The results for our batches can be observed in Table 4.2.

As we can see t^* is quite small when comparing to the overall execution time from Figure 4.

Table 4: Maximal time-save possible when using the MLS prototype

	static	**dynamic**	**guided**	**static_steal**	**fac2a**	**af_a**	**RandomSel**	**ExhaustiveSel**	**BinarySel**	**ExpertSel**	**Total**
t^* **(s)**	34.91	34.83	34.69	34.67	34.86	34.93	34.79	34.81	80.50	34.84	393.83

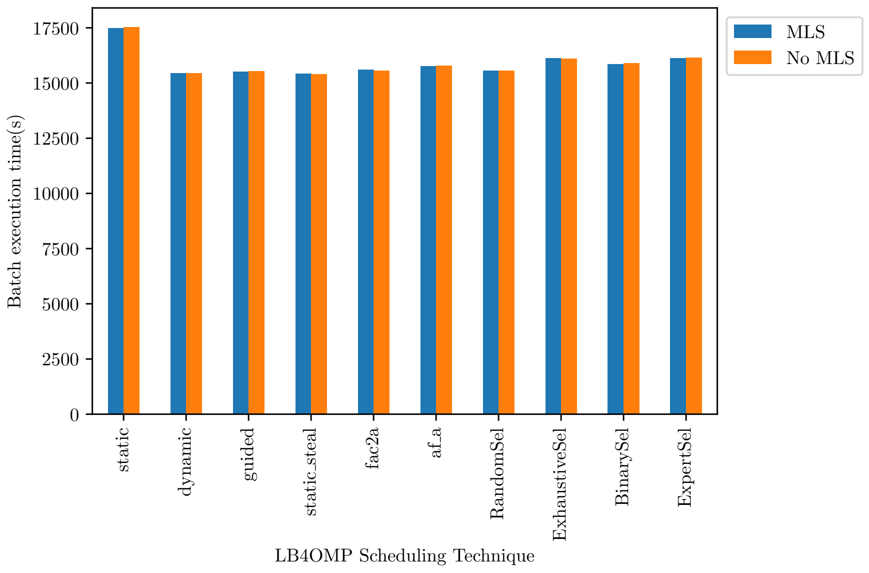

Figure 4: Execution time of a batch in seconds with and without the MLS prototype

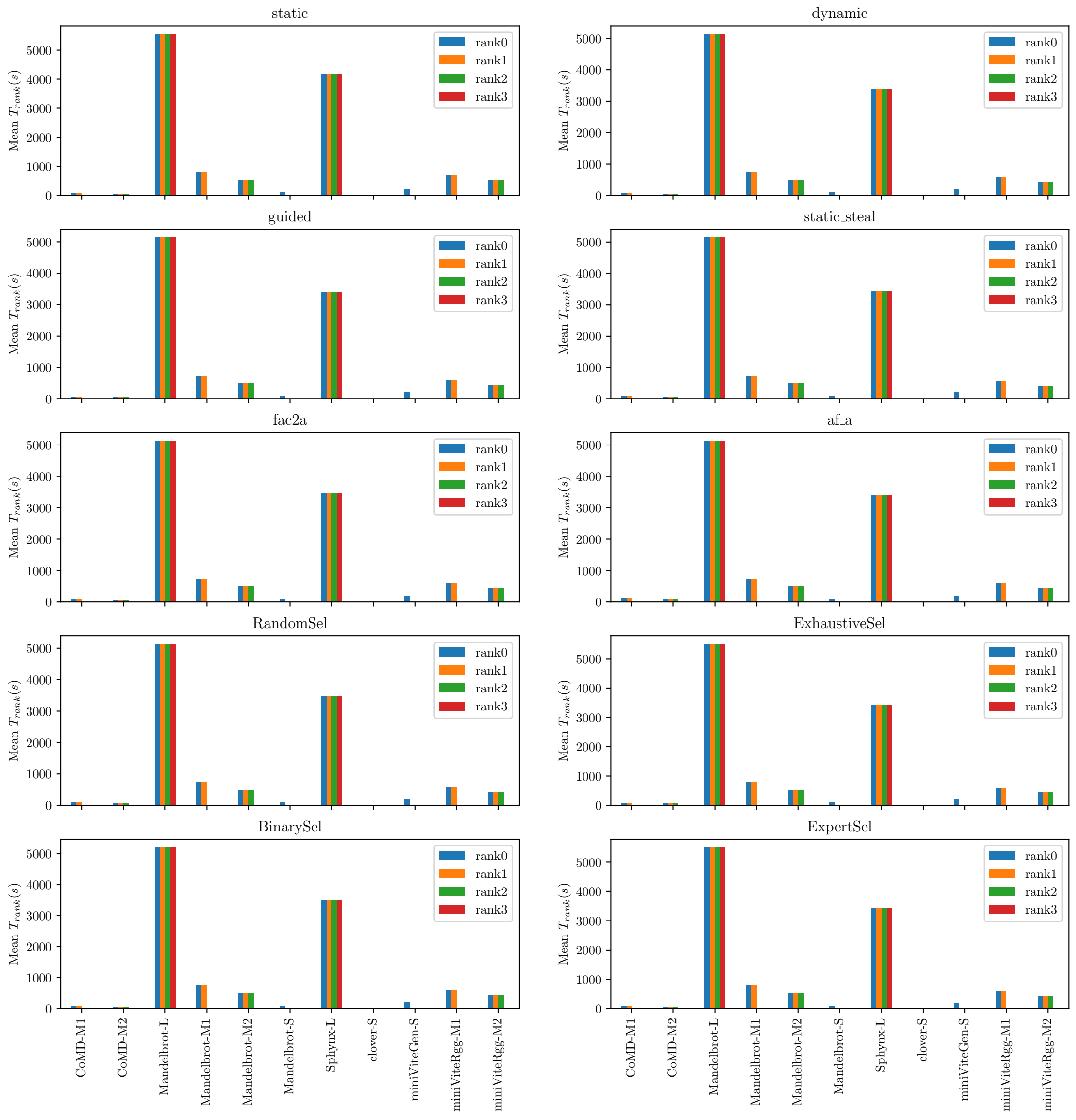

Figure 5: Total elapsed time in ranks for each application per batch

5 Discussion

When looking at the results presented in the previous section we can make the following two statements:

1. The reduction of load imbalance at thread-level has significantly increased the performance of our jobs.
2. Combining the thread-level scheduling with MLS prototype led only to a very small reduction in execution time.

The second statement might be explained in several ways. First of all we only used 4 nodes in our computation cluster for our experiments which severely reduces the usefulness of the MLS prototype. A second point to be made is that the order of job submission also plays an important role. Finally the low level of process-level load imbalance could have multiple reasons. It has been shown[13][1] that node-level scheduling may also implicitly reduce cross-node load imbalance in a significant manner. Additionally the applications from the large workload category use a collective MPI function at the end of their execution. This function call has an implicit barrier at the end which means that the ranks are going to call the *MPI_FINALIZE* function at virtually the same time. These are two possible reasons for the results we got from the t^* calculation, explaining the small amount of load imbalance at the process-level. This, in turn, has the consequence that nodes are released very close to each other in the timeline of the application execution. Therefore we did not have a high potential for performance gain from using the MLS prototype.

6 Conclusion and Future Work

In this work we evaluated the performance implications of combining the MLS prototype with thread-level scheduling using LB4OMP. In summary, we can say that we saw a minimal performance gain when combining thread-level scheduling and the MLS prototype, compared to pure thread-level scheduling. We also showed that the maximal possible time-save when adding the MLS prototype was quite small in contrast to the overall execution time of the batches. This was mainly due to ranks finishing their computations at very similar times and thus the applications having a low process-level load imbalance. Furthermore we observed a low induced overhead for the MLS prototype. Thus it might still be worth using both scheduling on both levels in conjunction, especially when using more nodes and a higher number of jobs.

Further work could include scaling up the amount of nodes to increase the effectiveness of the MLS prototype. Another interesting areas of exploration would be the evaluation of the significance of the job submission order and the addition of non-static scheduling at the application level using LB4MPI[14]. Lastly it might be a good idea to focus on process-level load imbalance in the exploration step in addition to the thread-level load imbalance.

References

[1] David Böhme, Markus Geimer, Lukas Arnold, Felix Voigtlaender, and Felix Wolf. Identifying the root causes of wait states in large-scale parallel applications. *ACM Transactions on Parallel Computing (TOPC)*, 3(2):1–24, 2016.

[2] Rubén M Cabezón, Domingo Garcia-Senz, and Joana Figueira. Sphynx: an accurate density-based sph method for astrophysical applications. *Astronomy & Astrophysics*, 606:A78, 2017.

[3] Ricolindo L Carino, Ali Mohammed, and Florina M Ciorba. Dynamic loop self-scheduling for load balancing (dls4lb), 2020. *URL: https://github.com/unibas-dmi-hpc/DLS4LB*. Accessed:2021-07-17.

[4] Florina M Ciorba, Christian Iwainsky, and Patrick Buder. Openmp loop scheduling revisited: making a case for more schedules. In *International Workshop on OpenMP*, pages 21–36. Springer, 2018.

[5] Ciorba, Florina M. minihpc: Small but modern hpc. `https://hpc.dmi.unibas.ch/en/research/minihpc/`. Accessed:2021-07-15.

[6] Luiz DeRose, Bill Homer, and Dean Johnson. Detecting application load imbalance on high end massively parallel systems. In *European Conference on Parallel Processing*, pages 150–159. Springer, 2007.

[7] Ahmed Hamdy Mohamed Eleliemy. *Multilevel Scheduling of Computations on Parallel Large-scale Systems*. PhD thesis, University_of_Basel, 2021.

[8] Sayan Ghosh, Mahantesh Halappanavar, Antonino Tumeo, Ananth Kalyanaraman, and Assefaw H. Gebremedhin. Minivite: A graph analytics benchmarking tool for massively parallel systems. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 51–56, 2018.

[9] JA Herdman, WP Gaudin, Simon McIntosh-Smith, Michael Boulton, David A Beckingsale, AC Mallinson, and Stephen A Jarvis. Accelerating hydrocodes with openacc, opencl and cuda. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 465–471. IEEE, 2012.

[10] Susan Flynn Hummel, Edith Schonberg, and Lawrence E Flynn. Factoring: A method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, 1992.

[11] Jonas H Müller Korndörfer, Ahmed Eleliemy, Ali Mohammed, and Florina M Ciorba. Lb4omp: A dynamic load balancing library for multithreaded applications. *arXiv preprint arXiv:2106.05108*, 2021.

[12] Benoit B Mandelbrot. Fractal aspects of the iteration of z$\rightarrow$ λz (1-z) for complex λ and z. *Annals of the New York Academy of Sciences*, 357(1):249–259, 1980.

[13] Ali Mohammed, Aurélien Cavelan, Florina M Ciorba, Rubén M Cabezón, and Ioana Banicescu. Two-level dynamic load balancing for high performance scientific applications. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*, pages 69–80. SIAM, 2020.

[14] Ali Mohammed, Ahmed Eleliemy, Florina M Ciorba, Franziska Kasielke, and Ioana Banicescu. An approach for realistically simulating the performance of scientific applications on high performance computing systems. *Future Generation Computer Systems*, 111:617–633, 2020.

[15] Ghosh Sayan and Halappanavar Mahantesh. Comd. `https://github.com/ECP-copa/CoMD`, 2019. Accessed:2021-07-16.

[16] Texas A&M University. Suitesparse matrix collection. `https://sparse.tamu.edu/DIMACS10/rgg_n_2_24_s0`. Accessed:2021-05-07.

[17] Adrian T Wong, Leonid Oliker, William TC Kramer, Teresa L Kaltz, and David H Bailey. Esp: A system utilization benchmark. In *SC'00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, pages 15–15. IEEE, 2000.

[18] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on job scheduling strategies for parallel processing*, pages 44–60. Springer, 2003.