

Overhead and Scalability in OpenMP

Bachelor Thesis
Jonathan Kock

Thesis written at:
Chair for High Performance Computing, IT Center,
RWTH Aachen, Seffenter Weg 23,
52074 Aachen, Germany
Supervisor: Dr. rer. nat. Christian Terboven

OpenMP is a popular library to parallelize programs. This thesis seeks to evaluate how well OpenMP constructs perform for a variety of parameters like thread affinity and compilers. To do this synthetic benchmarks for a variety of constructs like `for`, `taskloop` or `critical` have been performed on an Intel SkyLake cluster. The results show clear trends like that overhead increases as thread count increases or that `OMP_PLACES=cores` combined `OMP_PROC_BIND=close` experiences higher overhead because all threads are placed on the same socket and become limited by bandwidth.

1 Introduction

The need to speedup a computer program and perform more and more operations at the same time is not new. And while initially all computer programs were executed sequentially in recent years it has become more and more difficult to perform more operations on a single core within the same timeframe and using the same amount of energy. Stemming from this the current trend is to increase the amount of cores working on a problem to speed the computation up. But this poses a new challenges as it is rarely trivial to translate an originally sequential program into a parallel program that utilizes multiple cores.

The problem here is mostly that the higher the amount of cores becomes the less efficient becomes the computation and more and more time is spent dealing with a number of factors like for example contention for locks or resources, inefficient work distribution leading to idle processors or simply a lack off parallelism within the program [3].

There exist a number of standards that attempt to perform the parallelization as efficiently as possible. Most notable here is the Message Passing

Interface (MPI) which is the most used API for clusters and POSIX threads (pthreads) and OpenMP which are the most prevalent for shared memory multiprocessors.

Measuring the before mentioned efficiency is not a new idea and specifically for OpenMP the EPCC benchmark suit already exists [2] [1]. But the latest version of this benchmark exists for OpenMP version 3.0 while the latest version at the point of writing is version 5.1. In the newer versions a number of new interesting features were introduced, most notable here is the `taskloop` construct ¹.

There also exist multiple implementations of OpenMP like GOMP, CLANG or proprietary implementations like the one used by the intel c compiler.

In this thesis I will be presenting my methodology for measuring overhead and point out familiarities and differences between my approach and the EPCC benchmark in section 2. I will then present my results in section section 3 which is separated into four larger parts. In section 3.1 I will be discussing the impact that processor affinities and places can have on performance. I will than proceed to discuss parallel work constructs like `for`, `task` and `taskloop` in section 3.2. Following this I will be presenting results from benchmarks covering synchronization constructs in section 3.3. These include `single`, `critical`, `lock`, `ordered`, `reduction` and `atomic`.

2 Methodology

All benchmarks were performed on the cluster CLAIX 2018 at RWTH Aachen University. The clus-

¹<https://www.openmp.org/spec-html/5.0/openmpsu47.html>

ter consists of Intel SkyLake Xeon Platinum 8160 processors which have 24 cores each and has 2 sockets per node. The cluster utilizes SubNUMA clustering, which divides each processor into 2 NUMA nodes with 12 processing cores each. I performed the benchmarks for 4 different compilers:

- gcc/gomp, version 11.1.0
- clang - llvm project, version 13.0.0
- icc version 19.0.1.144
- icx version 2021.4.0

For each of these compilers I performed a benchmarks for a set of OpenMP constructs which are:

- workload parallelization
 - `for`/worksharing loop
 - `task` construct
 - `taskloop` construct
- synchronization constructs
 - `single`
 - `critical`
 - `lock`
 - `ordered`
 - `reduction`
 - `atomic`
- hints for critical sections: combinations of different hints
 - contended vs noncontended
 - speculative vs nonspeculative

I performed each benchmark across a verity of parameters:

- threads: amount of processing cores utilized, ranging from 2 to 48
- repetitions: amount of times the delay function is executed, ranging from 512 to 32768
- for `for` construct:
 - dynamic vs static schedule
 - chunksize
- for hints benchmarks: array size of the contended array
- differences between processor affinities in OpenMP
 - `OMP_PLACES=cores` vs `OMP_PLACES=sockets`

– `OMP_PROC_BIND=close` vs `OMP_PROC_BIND=spread`

To simulate workload I utilize a delay function which performs a certain amount of floating point operations (delay length). This delay function is executed a certain amount of times (repetitions). These repetitions can be executed in parallel. The delay function for work parallelization constructs has no communication with other instances of the function, while the delay functions for synchronization constructs and hints operate on a shared array to simulate contention. The exact code can be found at: <https://git-ce.rwth-aachen.de/jonathan.kock/overhead-and-scalability-in-openmp.git>

To measure the overhead I calculated the difference between the time a reference function takes to perform a workload in serial fashion vs the time it takes to perform the same workload (for synchronization and hints) or a weakly scaled version of the workload for `for`, `task` and `taskloop` constructs. This means that the runtime of a perfectly parallel version of the benchmark is the same as the runtime of the serial reference function for worksharing constructs while for synchronization and hints the total workload is the same and the runtime should theoretically be faster than reference function.

There are some differences here in comparison to the EPCC [1] benchmark. The first difference is the user input. While the EPCC benchmark expects the user to provide a desired time for how long the benchmark and the delay function should run I expect the user to input the amount of floating point operations the delay function should perform (delay length) and the amount of times the delay function should be executed in each benchmark (repetitions). I chose this because, while it makes it harder for a user to choose appropriate input, it guarantees that the workload performed is guaranteed to be the same across multiple compilers and systems.

Another difference is that I display in most plots the ratio between the mean of the benchmark times and the mean of the reference times (overhead ratio) while the EPCC benchmark calculates the difference between reference and benchmark times. I find this to be a better metric for the input methods I have chosen because if the amount of work to be performed varies and consequently the runtime of the reference function varies it still allows the benchmarks to be compared. All benchmarks were performed 16 times to achieve a certain level of confidence. I also display the maximum and minimum times divided by the average of the reference as error bars.

3 Results

3.1 Thread Affinities

OpenMP allows the user to define where a new thread is to be placed. This is performed via two variables: `OMP_PROC_BIND` and `OMP_PLACES`. `OMP_PLACES` defines whether a thread is pinned to a hardware thread, core or socket while `OMP_PROC_BIND` defines how for each thread the respective place is chosen. **Spread** signifies here that the threads should be spread as evenly as possible across places to maximize space between threads and **close** places threads as close to the master thread as possible.

In fig. 1 displayed are the means of overhead ratios from benchmarks performed across different repetitions for four different benchmarks. The benchmarks have been performed using the gcc compiler and a fix delay length of 16384.

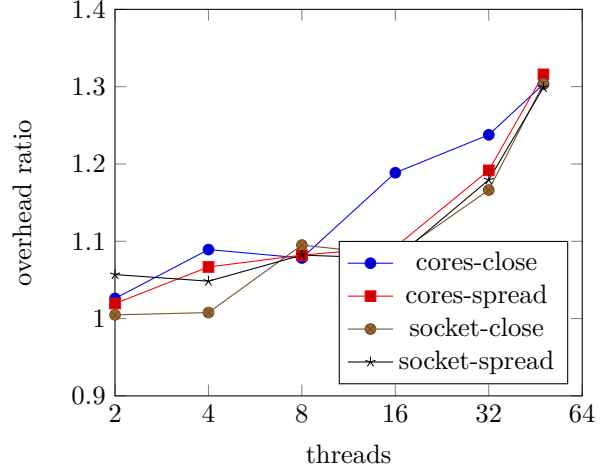
Plots 1 to 3 show the four possible configurations for the `for` construct with a dynamic schedule, the `task` construct and the `taskloop` construct.

Overall there is a clear trend that cost increases as thread count increases. This is to be expected as an increase in core count also increases the amount of communication between the cores. To further illustrate this I display the linear regression corresponding to the values of plots 1 to 3. This again shows a clear trend that overhead increases as thread count increases.

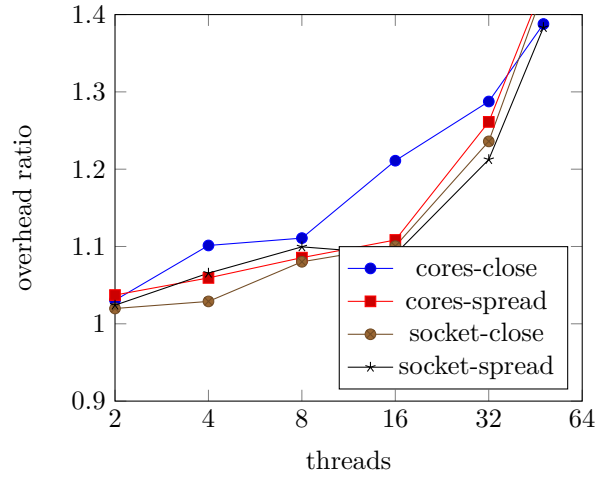
Socket-close shows the best performance for low thread amounts across all three constructs which largely equals out when moving to higher thread counts. This can also be seen in the regression formula which shows the lowest constant but second largest scaling. Clearly seen can be that there is a large increase in overhead when moving from 16 to 32 cores for the combinations cores-spread, socket-close and socket-spread for all three constructs. For the combination cores-close this jump already occurs when increasing the amount of threads from 8 to 16. This is also to be expected as this is probably a limitation of cache bandwidth and, as all threads in cores-close are allocated on the same socket and NUMA-node first this strategy runs into bandwidth contention earlier than other strategies.

Overall when considering the regression it becomes clear that cores-close shows the worst performance of all combinations. Both sockets-close and sockets-spread are very competitive, with sockets close performing better on lower thread counts and sockets spread performing better on higher core counts. I will be using sockets-spread in the following sections unless otherwise mentioned.

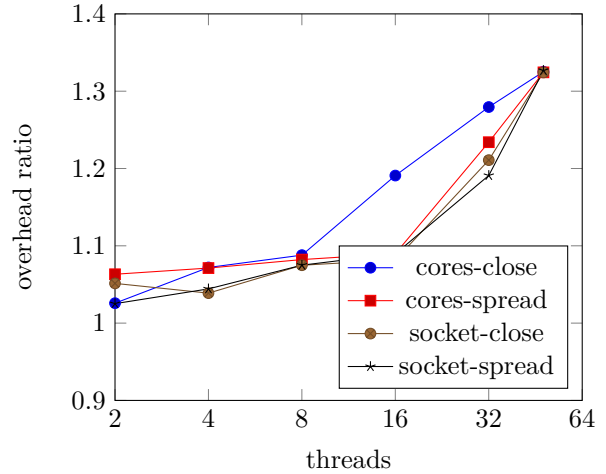
Interesting here is also that the task construct



Plot 1: for construct, dynamic schedule, chunksize = 1

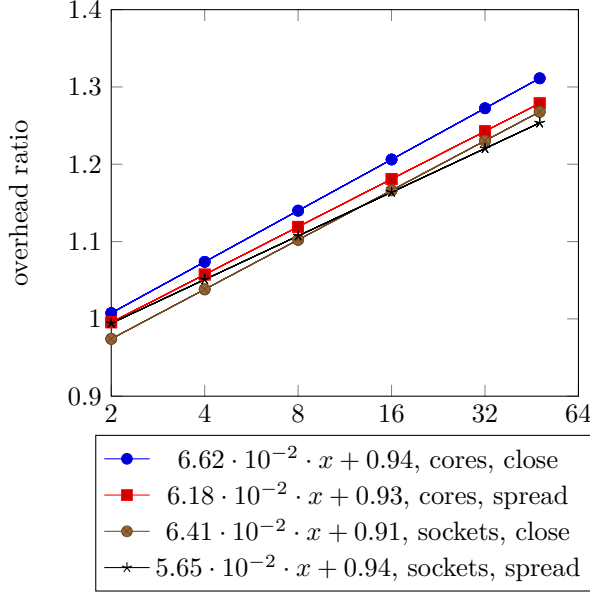


Plot 2: task construct



Plot 3: taskloop construct

Figure 1: close vs spread, socket vs core
gomp, mean across repetitions, delay length = 16384



Plot 4: close vs spread, socket vs core
linear regression corresponding to fig. 1

seems to perform worse than the other constructs which will be further discussed in the following section.

3.2 Parallel work constructs

In this section I will be discussing the advantages and disadvantages of the three different work parallelization constructs `for`, `task` and `taskloop`. In the previous section I mentioned that the `task` construct looks like it performs worse. This is now relativized by plot 9 which displays the mean across a variety of repetitions for the socket-spread affinity combination. The plot shows the `for` construct with both a static and dynamic schedule with chunksize 1, the `task` construct and the `taskloop` construct. In plot 10 I display the linear regressions corresponding to the values displayed in plot 9.

Combining plot 9 and plot 10 it becomes clear that the `task` construct scales the worst but also starts the best. It is also visible that while `for`, dynamic starts off worse than all other constructs it recovers from this as the thread count increases and ends up as the best or second best performer for thread counts from 16 to 48. Looking at the linear regressions and actual values it is hard to determine an overall bestperforming construct because both `for`, static and `taskloop` perform very well for different amounts of threads with `for`, static beating `taskloop` for thread counts ranging from 16 to 48 while losing to it for thread counts ranging 2 to 8. This is also supported by the regressions which display the same scaling for both with a slight

starting advantage for `taskloop`.

Figure 2 displays four 3d plots for the above mentioned four benchmarks now also showing clearly what impact the amount of repetitions can have on performance. This is especially drastic for the `task` construct whose overhead skyrockets when using a low amount of repetitions with 32 or 48 threads.

To further detail this plot 11 displays two plots which have a repetitions number of 512 and 16384 respectively. It is clear that across the board the benchmark with 16384 shows better results, especially for the thread counts 32 and 48. In my opinion this is mainly so because the serial fraction of the program becomes larger when the parallel fraction of the program decreases and so the amount of overhead increases.

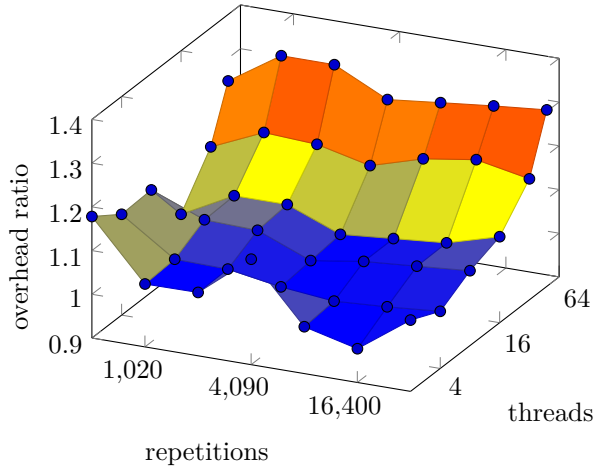
Another interesting observation that can be observed in fig. 2 is that plots 5 and 6 show a considerably lower overhead for 48 cores and 512 repetitions than for 1024 and 48 repetitions. This is also consistent with other compilers as displayed in fig. 3.

Also observable is that `for` and `task` display considerably more volatile behavior than `taskloop`. `Taskloop` does not display the large spikes in overhead that `task` displays at 2048 repetitions and 8 threads and 8192 repetitions and 4 threads in plot 7 and also does not exhibit the unpredictable behavior displayed by `for` for 512 repetitions. This consistency is highly valuable and combined with the good general performance displayed by `taskloop` in plots 9 and 10 shows clear advantages of `taskloop` over `for` and `task`, although these display better performance for some combinations of parameters.

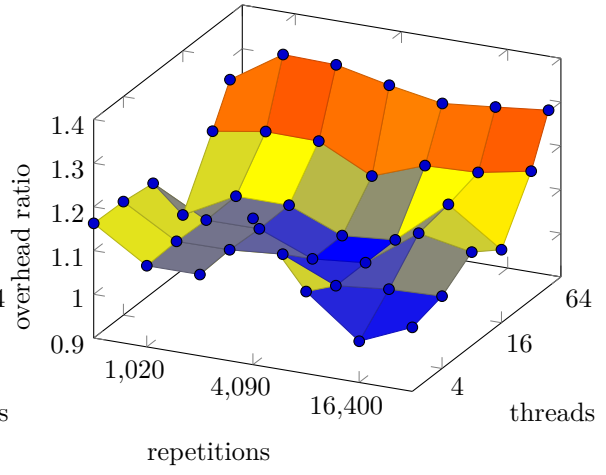
3.2.1 Different compilers

To illustrate differences between the four compilers `gcc/gomp`, `clang`, `icc/intel classic c compiler` and `icx/intel oneAPI DPC compiler`, figs. 3 to 5 display 3d plots showing `for`, `task` and `taskloop` constructs. For all plots a fix delay length of 16384 and the socket-spread affinity combination is used.

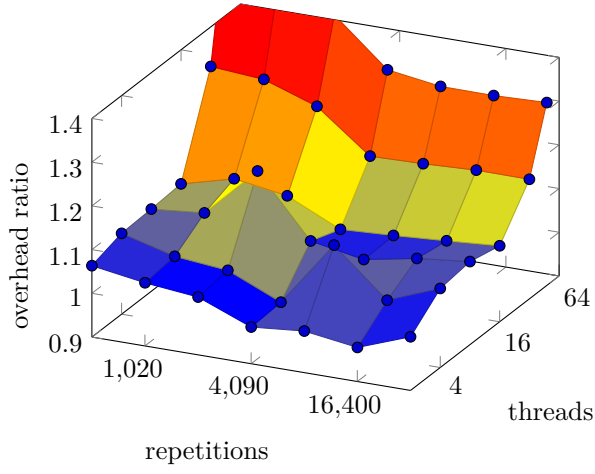
For construct Figure 3 on page 7 displays plots for all four compilers and the `for` construct with a dynamic schedule and chunksize 1. It is clearly visible that all four compilers display the drastically lower overhead for 512 repetitions and 48 threads mentioned in the previous section. `Gomp` and `icc` display a similar increase in overhead around 512 repetitions and 2 threads. `Clang` and `icx` do not display this increase but rather a decrease in overhead. `Clang` and `icx` also display some sudden spikes in overhead which are for `clang` localized at 1024 and 8192 repetitions and for `icx` localized at 4096 and 8192 repetitions. Note that these spikes



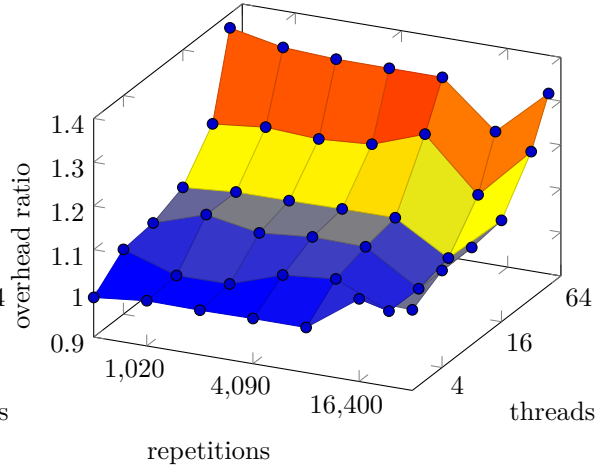
Plot 5: 3d plot of `for`, dynamic



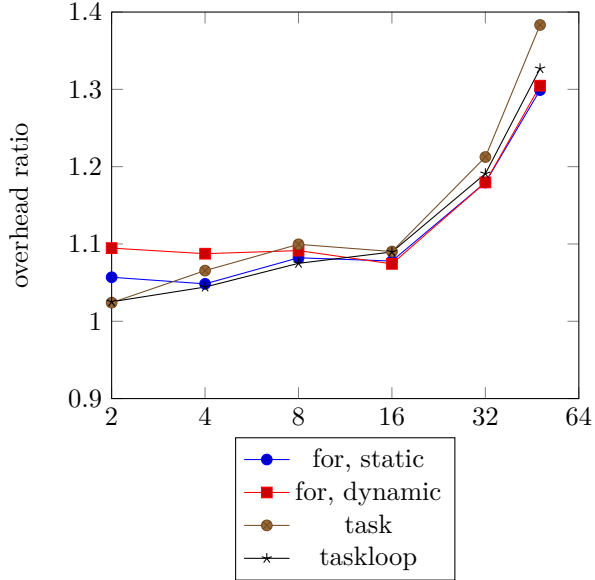
Plot 6: 3d plot of `for`, static schedule



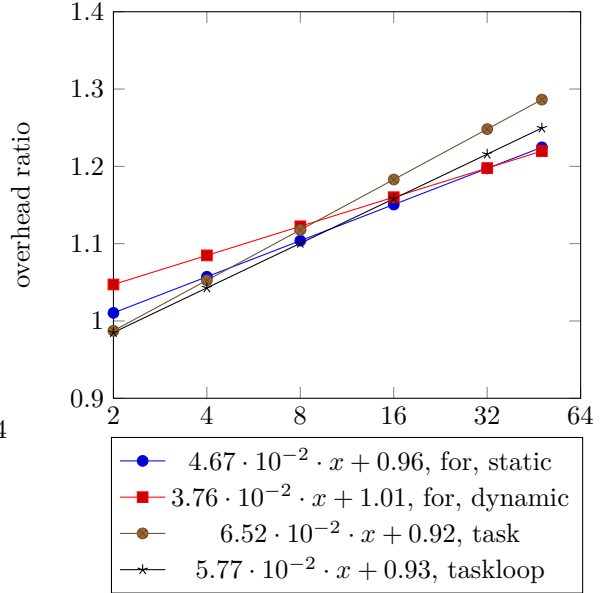
Plot 7: 3d plot of `task`



Plot 8: 3d plot of `taskloop`

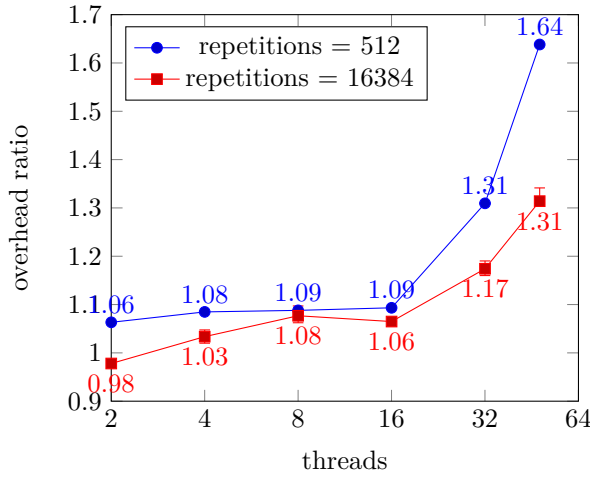


Plot 9: mean across repetitions



Plot 10: linear regressions corresponding to plot 9

Figure 2: 3d plots, `for` static, `for` dynamic, `task`, `taskloop`
gomp, delay length = 16384, affinity = socket-spread, chunksize = 1



Plot 11: **task** construct, 512 vs 16384 repetitions
gomp, delay length = 16384,
affinity = socket-spread

do not occur to the same degree for thread counts where overhead is high for all compilers (48 and to a lesser degree 32 threads).

To give a more general comparison of performance I display the mean across repetitions in plot 16 and the corresponding regression in plot 17. Here it is clearly visible that **clang** performs especially well for lower thread counts while still remaining competitive at higher thread counts. **Icx** displays the overall worst performance while **icc** performs the best at 8 and 16 threads. **Gomp**, **icc** and **icx** display similar scaling in the regression and **clang** displays considerably worse scaling which probably occurs because of the considerably better performance displayed at 2 and 4 threads.

Clang also displays almost linear scaling between 2 and 16 threads while the other compilers behave less consistent, especially from 2 to 4 threads where they all display a decrease in overhead. To contrast this in fig. 3 **gomp** displays the least erratic behavior. This split is definitely interesting as it shows that while **clang** displays better overall performance and a more linear increase in overhead it is also more susceptible to spikes in overhead.

Task construct Figure 4 on page 8 shows **task** benchmarks executed by all four compilers. Here **gomp** displays drastically different behavior than the other three compilers around 512 repetitions and 48 threads. **Gomp** displays here a considerably higher overhead than all three other compilers. This is also consistent with plot 22 and plot 23 which respectively display the means across repetitions and the corresponding linear regression. Here **gomp** also displays considerably worse performance than

the other three compilers, especially at 48 threads.

Also interesting is that **icx** displays no large spikes in plot 21. This is in contrast to all other three compilers which display more or less erratic behavior. **Gomp** displays a large spike at 2048 repetitions and 8 threads and a smaller spike at 8192 repetitions and 4 threads while **clang** shows very erratic behavior around 1028 repetitions and 4 threads. **Icc** behaves similar to **gomp** as that it has some localized spikes but no larger erratic behavior. This is also supported by plot 22 which shows that **icx** displays almost linear growth in overhead until 32 threads while **gomp** displays a large jump in overhead from 4 to 8 threads.

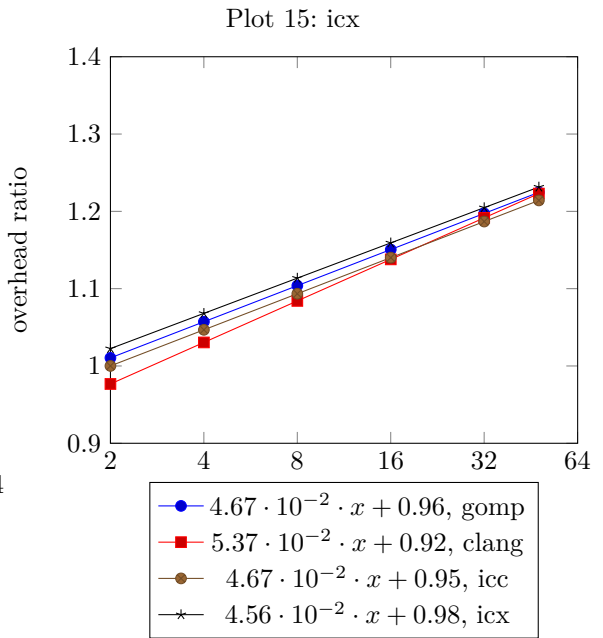
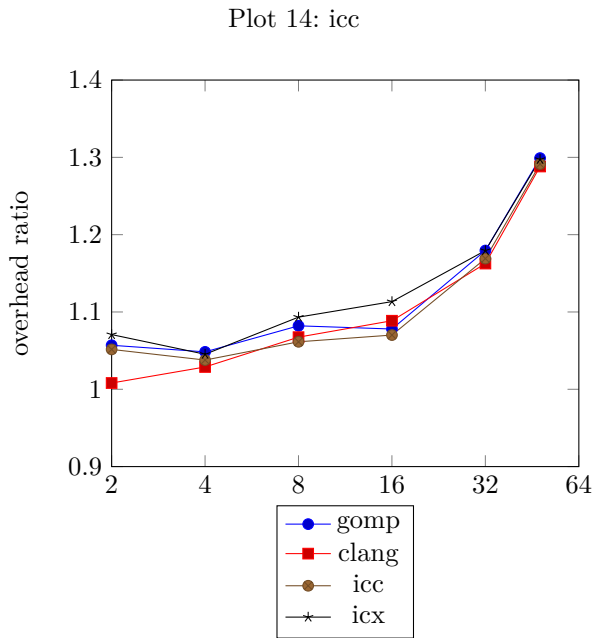
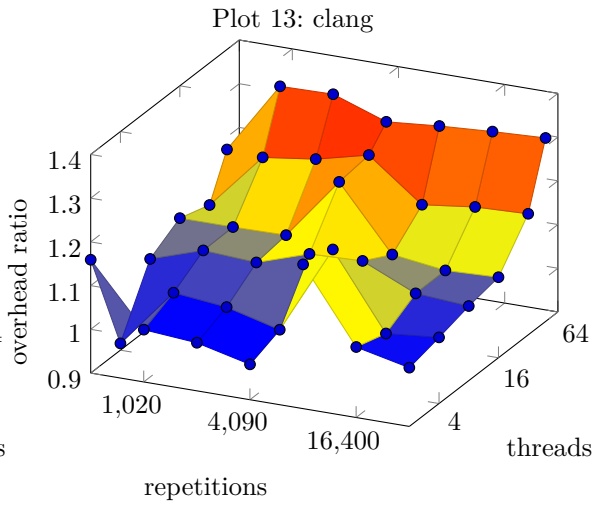
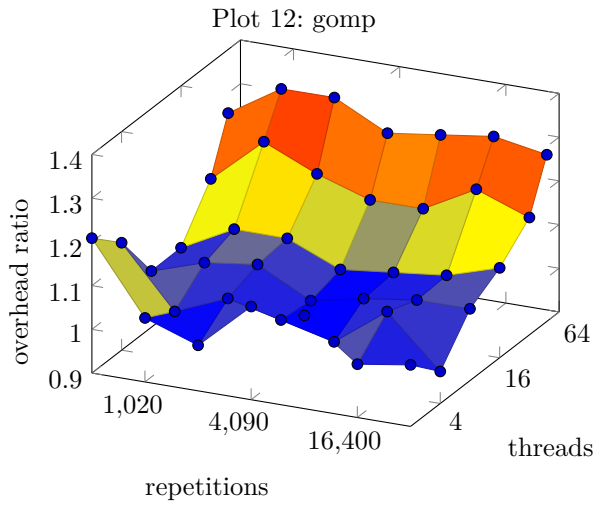
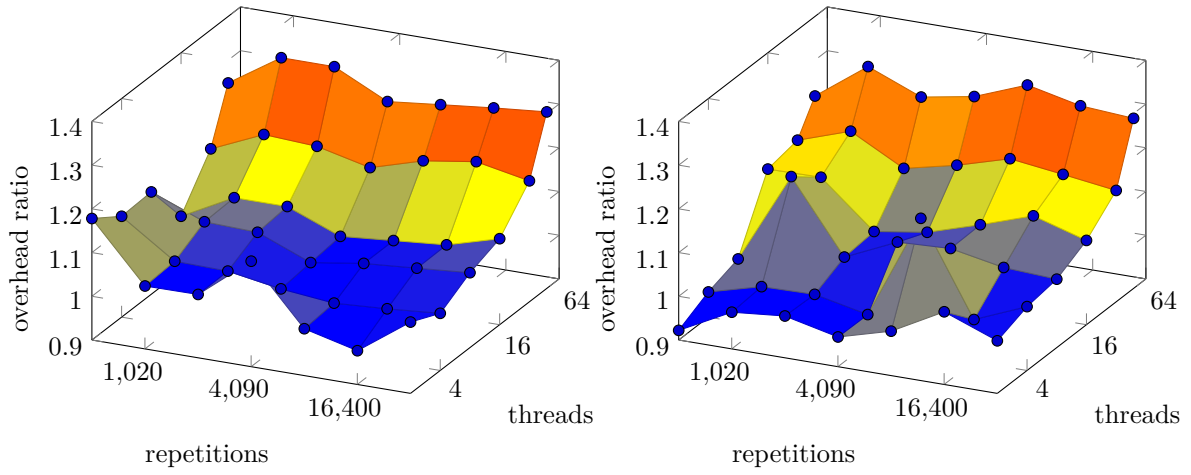
Overall there are no large difference between **clang**, **icc** and **icx** with regards to plots 22 and 23, but **icx** displays a considerable advantage by avoiding localized spikes in overhead like those displayed by **icc** and areas of volatility like the one displayed by **clang** in fig. 4.

Taskloop construct Figure 5 on page 9 displays **taskloop** benchmarks performed by all four compilers. Clearly **clang** displays very erratic behavior in plot 25 that all other compilers do not display. Opposite to this **gomp**, **icc** and **tcx** all display no spikes and almost no variation across repetitions and show an overall very stable performance. Both **gomp** and **icc** display a drop in overhead at 16384 repetitions while **icx** displays a drop in overhead at 512 repetitions.

Interestingly the erratic behavior of **clang** is no longer visible when looking at plots 28 and 29 which display the means across repetitions and the corresponding linear regression respectively. Here the erratic behavior completely evens out and **clang** appears to be performing very similarly to the other three compilers.

Overall the best performing compiler is **icx**, who trumps all others in both consistency and performance regardless of thread count. Both **gomp** and **icc** perform very similar with **icc** performing slightly better than **gomp**. **Clang** performs very well at some points, but also very bad at others and is hard to predict whether it performs good or bad at a certain point.

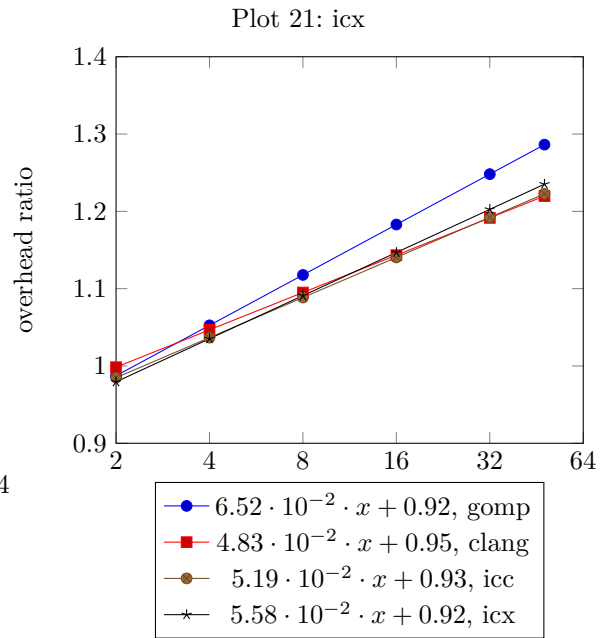
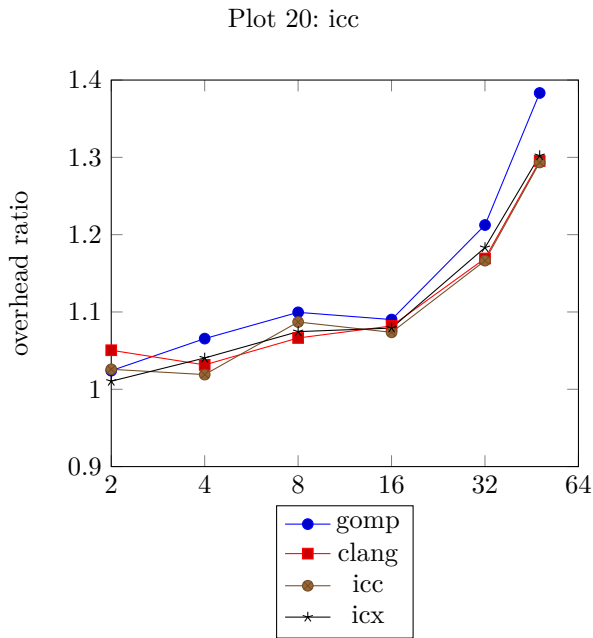
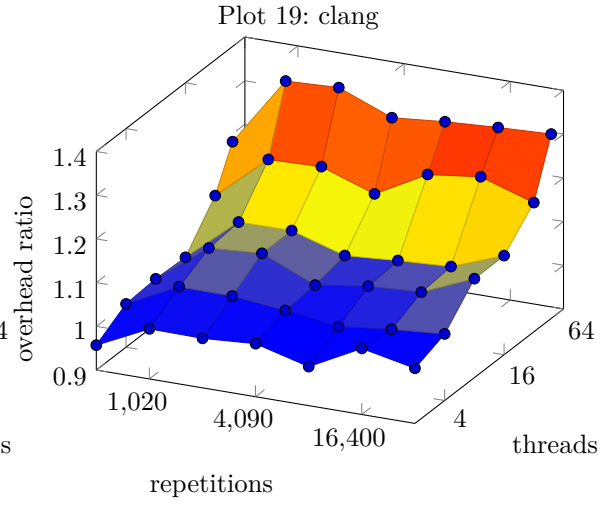
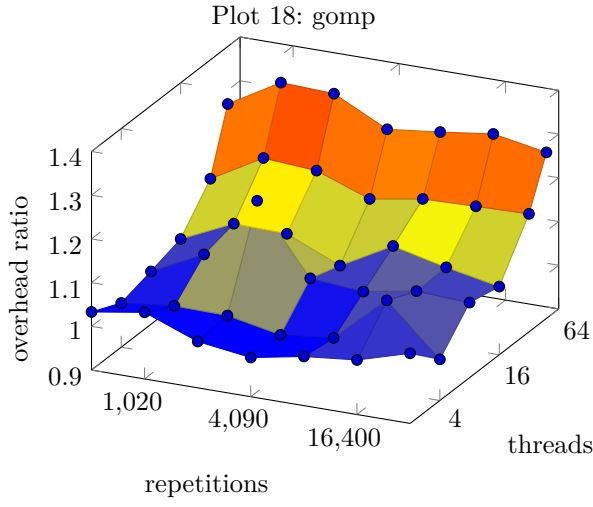
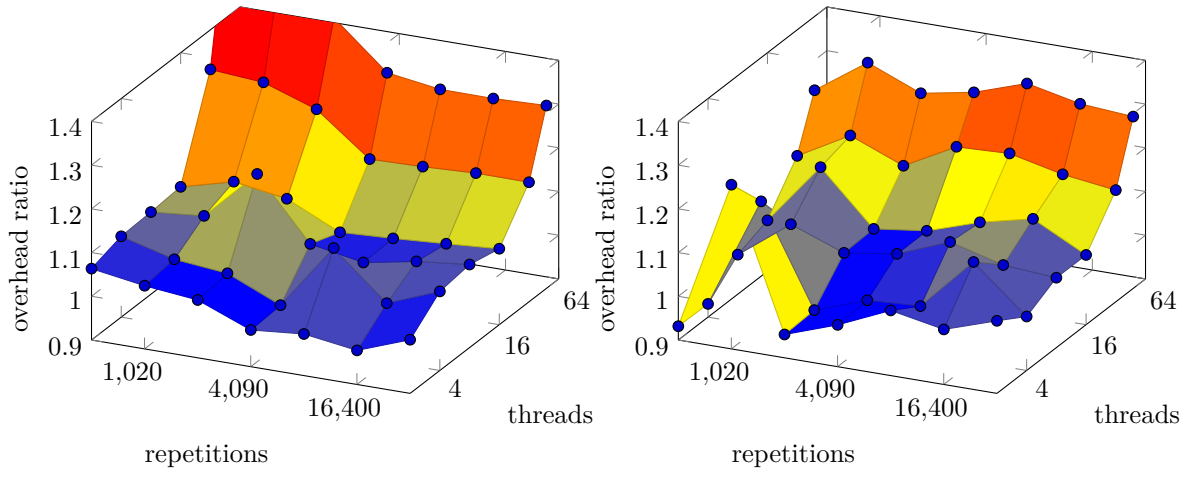
Summary Overall **clang** showed the least consistency across the three constructs, but also displayed excellent overall performance for the **for** construct. The most consistency for the **for** construct displayed **gomp**, but **gomp** showed only middling performance here and for the **taskloop** construct and very bad performance for the **task** construct. **Icc** showed slightly better performance than **gomp** but



Plot 16: means across repetitions

Plot 17: linear regressions corresponding to for, plot 16

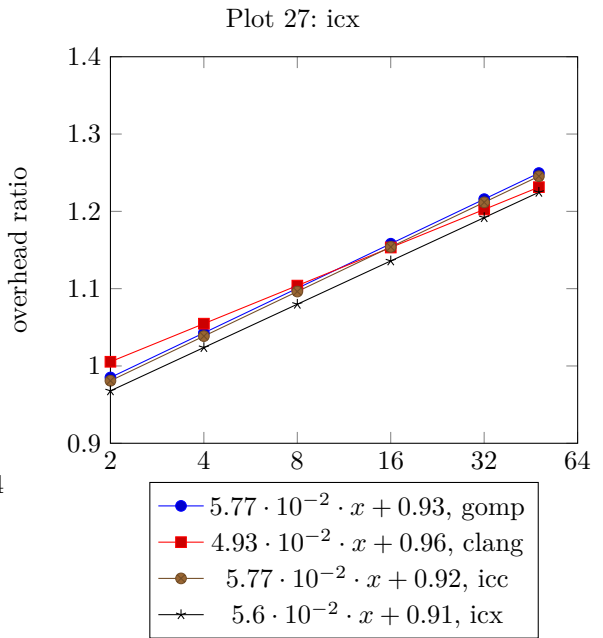
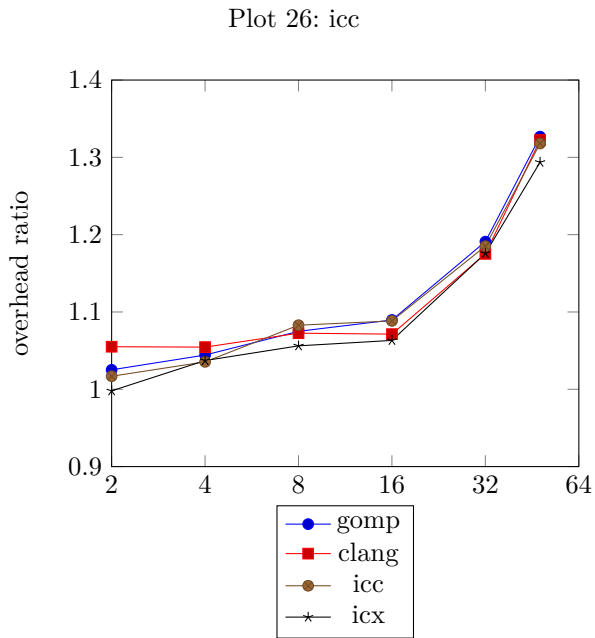
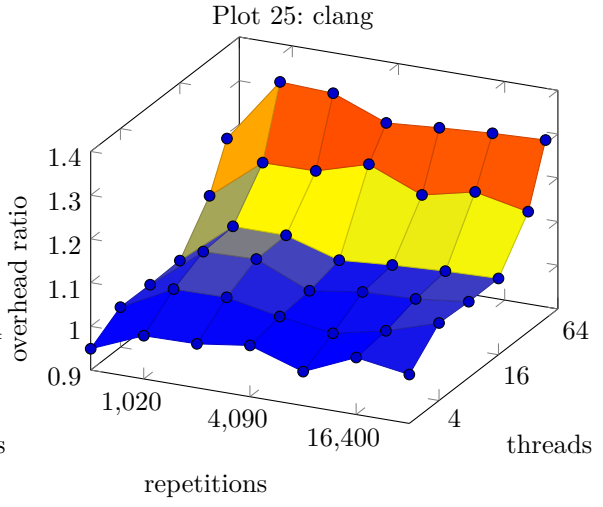
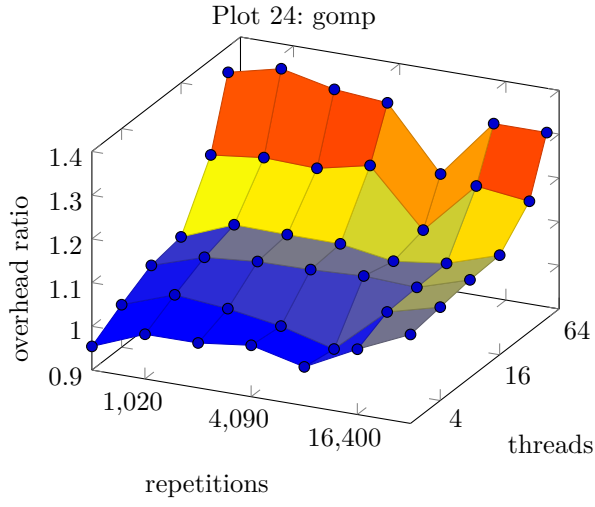
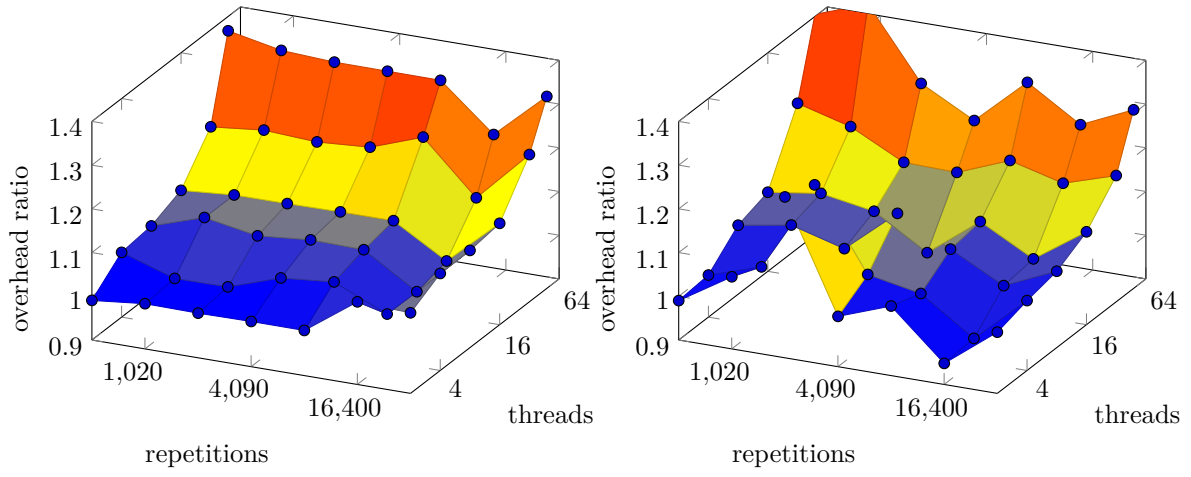
Figure 3: for, schedule = dynamic, chunksize = 1, gomp vs clang vs icc vs icx
delay length = 16384, affinity = socket-spread



Plot 22: means across repetitions

Plot 23: linear regressions corresponding to task, plot 22

Figure 4: **task**, gomp vs clang vs icc vs icx
delay length = 16384, affinity = socket-spread



Plot 28: means across repetitions

Plot 29: linear regressions corresponding to `taskloop`, plot 28

Figure 5: `taskloop`, gomp vs clang vs icc vs icx
delay length = 16384, affinity = socket-spread

remained similar in consistency. `icx` is inconsistent and relatively slow for the `for` construct but outperforms all other compilers in both `task` and `taskloop` benchmarks.

3.2.2 Chunksizes and Static vs Dynamic

There are multiple strategies to distribute iterations of a `for`-loop to threads. I will discuss two strategies today: the static and the dynamic schedule. The static schedule assigns each processor an equal amount of iterations in chunks while the dynamic schedule makes threads request a new chunk once they have completed the previous one. The size of these chunks is called chunksize.

In fig. 6 on page 11 I display two 3d plots of the `for` construct with a static and a dynamic schedule. In the 3d plots displayed are means across repetitions for a delay length of 16384 and the socket-spread affinity combination. It is visible that the dynamic schedule performs largely similar to the static schedule when considering thread counts ranging from 8 to 48 but slightly better when considering thread counts of 2 and 4. Overall chunksize seems to have no consistent impact on performance.

3.3 Synchronization Constructs

In this section I will be presenting my results from benchmarks performed with synchronization constructs. Specifically, I will be discussing `single`, `critical`, `lock`, `ordered`, `reduction` and `atomic`.

In plot 32 I display means across repetitions for 5 constructs. `Atomic` is listed separately in plot 33 as it performs with such a high overhead that it can not be included with the other constructs if the graphic shall still be meaningful.

Considering that the workload performed across the synchronization constructs is the same as performed by the reference task plot 32 shows that OpenMP synchronization constructs come with a sizable overhead that increases as thread count increases. This is not surprising considering that each opening and closing of a lock takes a considerable amount of time and each stop and restart of thread takes a considerable amount of time as well. Because the amount of times this occurs increases with increased thread count it is only to be expected that overhead increases.

3.3.1 Different compilers

Similarly to section 3.2.1 I will be comparing different compilers in this section. I will compare the compilers `gcc/gomp`, `clang`, `icc`/intel classic c compiler and `icx`/intel oneAPI DPC compiler. All plots

use a fix delay length of 16384 except `atomic` which does not utilize a delay function.

Single Figure 7 on page 13 displays benchmarks for all four compilers as well as plots 38 and 39 which display the means across repetitions and the corresponding linear regressions respectively.

Overall all compilers deliver a fairly stable performance and an increase in overhead as the thread count increases. `Clang` displays a spike in overhead at 512 repetitions and 48 threads, `icc` displays a similar spike at 2048 repetitions and 48 threads. `Gomp` shows decreased overhead at 16384 repetitions and `clang` at 8192 repetitions.

Looking at the overall performance of the four compilers in plot 38, `icx` displays comparatively very high overhead at 2 threads which drops drastically to 4 threads. Otherwise all compilers display similar behavior, with `gomp` performing slightly better than other compilers for higher thread counts and `clang` performing better than others for lower thread counts.

Critical In fig. 8 on page 14 I present 3d plots 40 to 43 which display benchmarks of the critical construct in combination with the four compilers. I also present plots 44 and 45 which display the means across repetitions and the corresponding linear regressions respectively.

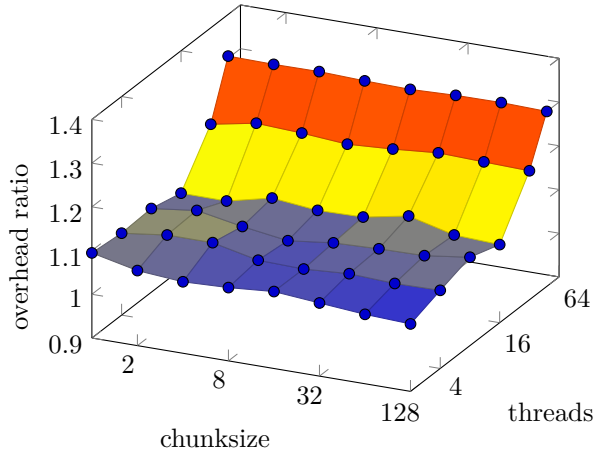
All compilers experience an increase in overhead as thread count increases. `Gomp` provides a very stable performance with a slight overhead decrease at 16384 repetitions. Similarly `clang` performs also very stable with an overhead decrease at 8192 repetitions. `Icc` displays a slight overhead increase at 2048 repetitions and performs otherwise stable, `icx` displays a slight cost decrease at 4096 repetitions.

Looking at the means across repetitions it is visible that `icc` performs clearly worse than all other compilers and `clang` clearly better. `Gomp` and `icx` perform roughly equally well, with `gomp` performing slightly better for lower thread counts.

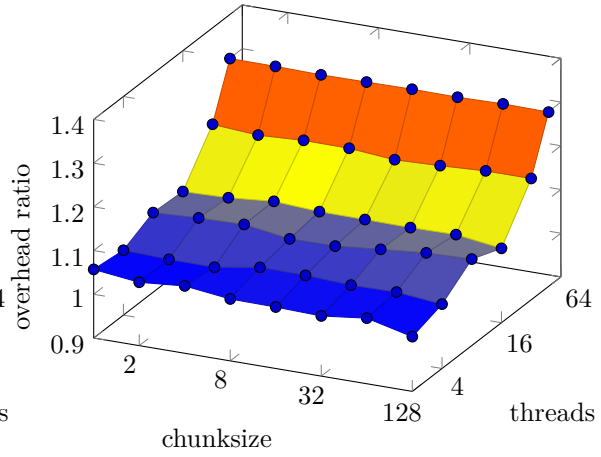
Overall all compilers display an overhead increase as thread count increases. `Clang` performs clearly better than the other compilers and also displays similar consistency. In general all four compilers display good consistency. With regards to performance the four compilers can be clearly ordered from best to worst: `clang`, `gomp`, `icx` and `icc`.

Lock Figure 9 on page 15 presents four 3d plots that display benchmarks of OpenMP locks in combination with the four compilers.

Overall the benchmarks look very similar to the critical construct benchmarks displayed in fig. 8.

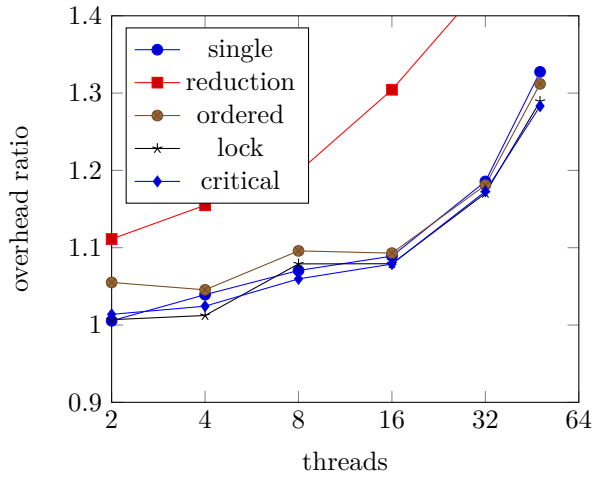


Plot 30: 3d plot of `for`, static schedule

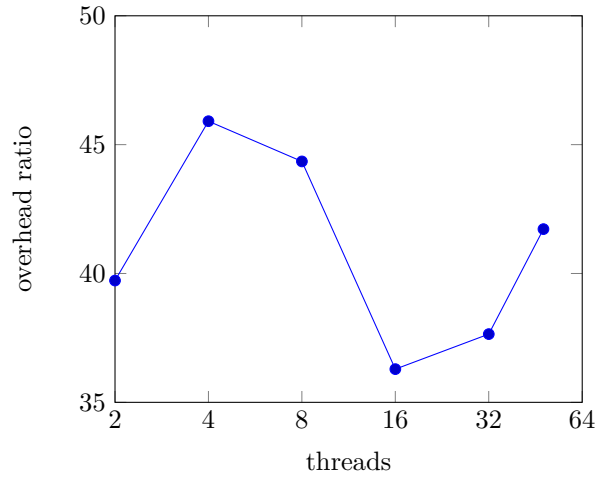


Plot 31: 3d plot of `for`, dynamic schedule

Figure 6: 3d plots, `for` static vs dynamic
means across repetitions, `gomp`, delay length = 16384, affinity = socket-spread



Plot 32: synchronization constructs, means across repetitions
`gomp`, affinity = socket-spread



Plot 33: `atomic` construct, means across repetitions
`gomp`, affinity = socket-spread

This only makes sense as a critical section and a lock are similar. `Gomp` displays the same overhead decrease at 16384 repetitions, `clang` at 8192 repetitions, `icx` at 4096 repetitions and `icc` displays a cost increase at 2048 repetitions. Also the overhead increases for all compilers as thread count increases. There are some new spikes in overhead for `gomp` and `icx` but overall the benchmarks show very similar performance.

This is also visible when looking at plots 50 and 51, which display the means across repetitions and the corresponding linear regression respectively. Here again `clang` clearly displays the best performance, followed by `gomp`, `icx` and finally `icc`. `Clang` also displays the best consistency. Overall

lock performs slightly worse than the critical section and are less consistent.

Ordered In fig. 10 on page 16 displayed are 3d plots for benchmarks of the ordered construct in combination with the four compilers as plots 52 to 55. Here `gomp` displays an overhead increase at 4096 repetitions, `clang` displays an overhead decrease at 8192 repetitions, `icc` two overhead increase at 2048 and 8192 repetitions and `icx` displays a number of overhead spikes.

In plots 56 and 57 `clang` displays the best performance again for all thread counts. `Icc` again displays the worst performance while `gomp` and `icx` perform roughly equally. All compilers experience

an increase in overhead as thread count increases.

Overall **clang** exhibits the best performance and consistency, followed by **gomp** and **icx** and **icc** displays again the worst performance.

Reduction Figure 11 on page 17 shows 3d plots of the reduction construct in combination with the four compilers as plots 58 to 61. Plots 62 and 63 show the means across repetitions and the corresponding linear regression respectively.

All compilers show good consistency with **gomp** and **clang** displaying an overhead increase at 4096 repetitions, **icc** displaying overhead increases at 2048 and 8192 repetitions. Compared to previous synchronization constructs, reduction displays higher overheads which can also be observed in plot 32. Nevertheless for reduction is the trend that overhead increases as thread count increases equally applicable.

Overall all four compilers display very similar levels of performance and consistency.

Atomic In fig. 12 on page 18 displayed is the final synchronization construct we will be discussing. Figure 12 contains 3d plots 64 to 67 which present benchmarks of the atomic construct performed with the four compilers. Atomic differs drastically from previous synchronization constructs.

For one only **icc** displays the previously so prevalent trend of increasing overhead with increasing thread count. Another large difference is the amount of overhead. With **icx** performing the best it is still exhibiting almost 30 times as much overhead as the previous constructs. Both **gomp** and **clang** display their highest overhead between 2 and 8 threads, while **icx** experiences the highest overhead at 48 threads.

Overall **icx** performs clearly the best, with **icc** performing the worst because of the scaling it experiences. The consistency of all four compilers is low especially if you consider the scale of plots.

Summary Overall **clang** shows the best performance for the most constructs. This is further reinforced by **clang** also displaying a very consistent behavior. **icx** shows by far the best performance for atomic, which is not worth a lot because atomic shows overheads that are by a magnitude of almost 30 larger than overheads from all other constructs. **icc** shows the worst performance with regards to critical, lock and ordered. Here both **gomp** and **icx** perform roughly equally. Single performs worse than critical, lock and ordered with all compilers performing roughly equally.

To summarize I would recommend using **clang** when using synchronization constructs and avoiding the intel classic c compiler.

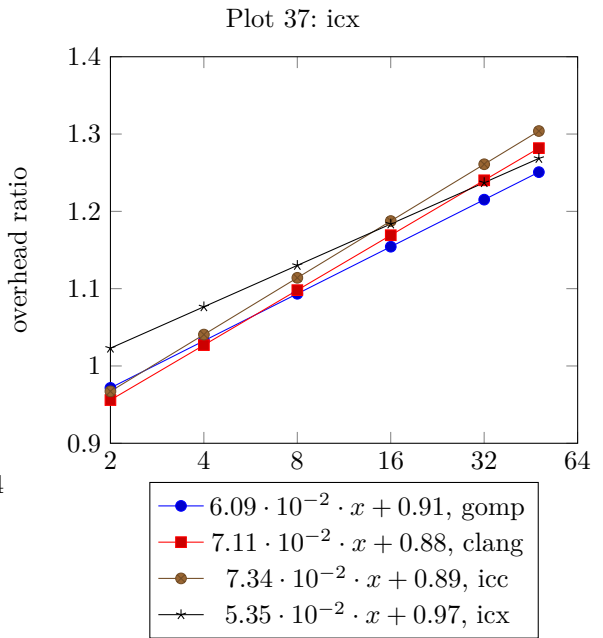
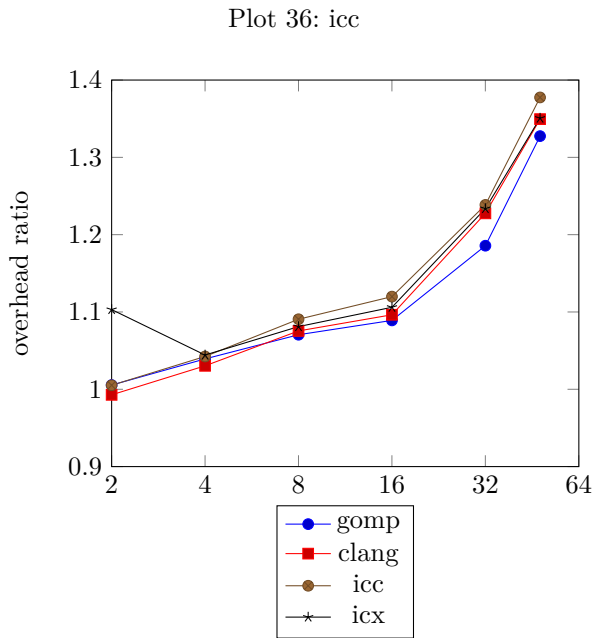
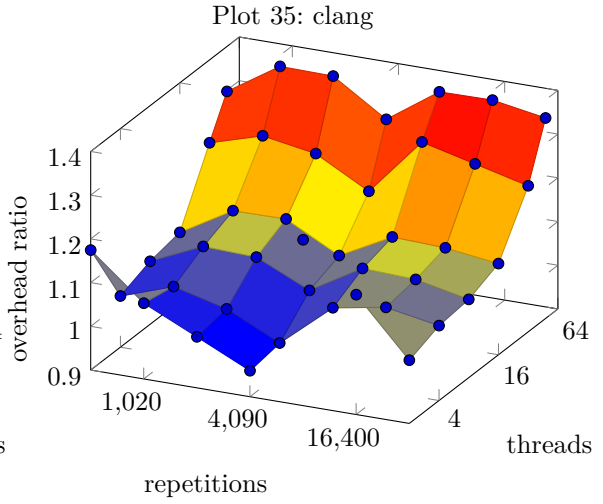
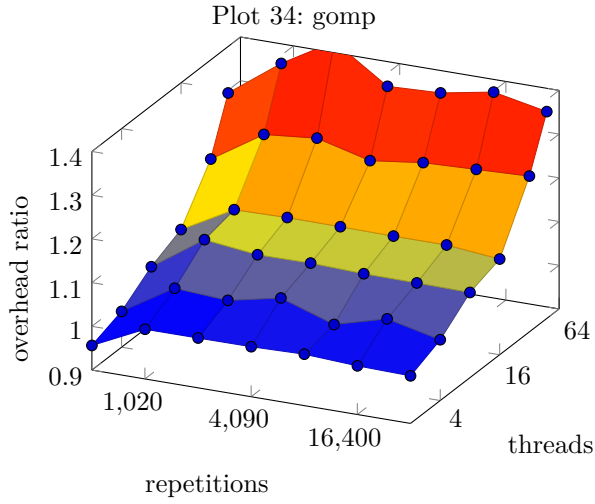
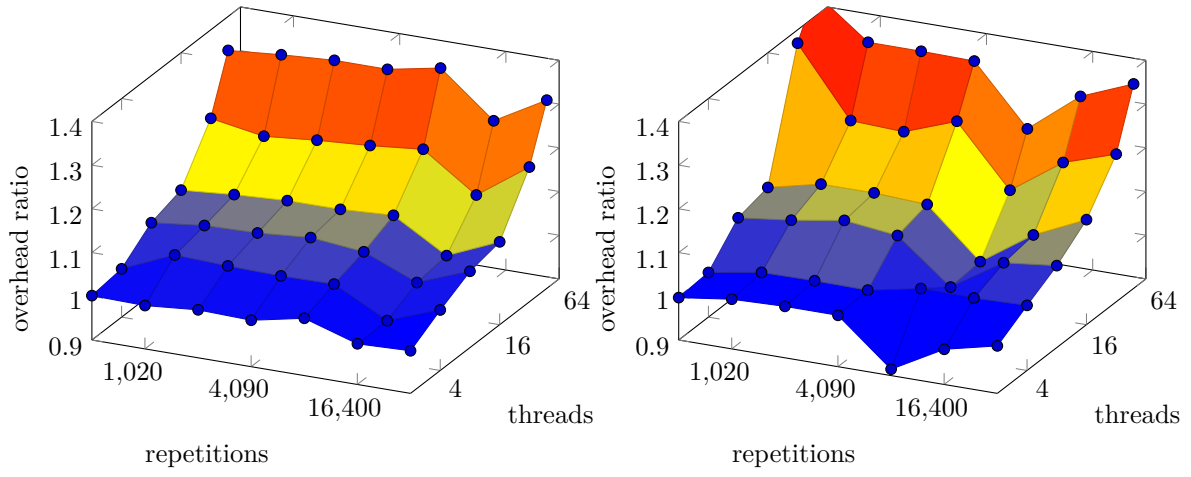
4 Conclusions

In this thesis I presented the results from a large series of benchmarks which aim to point out trends with regards to overhead and threads scaling in OpenMP. The first observation here is that the overhead increases as the thread count increases. For the combination of **OMP_PLACES=cores** and **OMP_PROC_BIND=close** this starts to show earlier than for other combinations when moving from 8 to 16 threads. This occurs probably because of bandwidth limitations as all threads are pinned to the same socket. Another observation is that the combinations **OMP_PLACES=cores**, **OMP_PROC_BIND=spread**; **OMP_PLACES=socket**, **OMP_PROC_BIND=close** and **OMP_PLACES=socket** and **OMP_PROC_BIND=spread** show largely equal performance as displayed in section 3.1 and begin to experience a large increase in overhead when moving from 16 to 32 threads.

In section 3.2 I showed that the **taskloop** construct is a competitive construct to parallelize work, especially if combined with compilers that perform especially well with it like **icx** (see fig. 5). **icx** overall performs very well with tasking considering that in fig. 4 the **icx** compilers shows a considerably more consistent performance when compared with other compilers, but performs badly while executing **for** constructs. **Clang** has very good average performance when executing **for** constructs but shows some spikes in overhead. To avoid those it is preferable to use **gomp** or **icc** which show comparable performance but display no overhead spikes. I also showed that chunksize has little to no impact upon overhead for the **for** construct, regardless of whether a static or a dynamic schedule is used. Overall has the dynamic schedule slightly lower overhead compared with the static schedule.

With regards to synchronization constructs I have found that **clang** displays excellent performance and consistency for all except **atomic**. **Atomic** performs best when compiled with the **icx** compiler. The Intel classic c compiler (**icc**) shows higher overhead than the other compilers and also displays a large amount of erratic behavior for some constructs. **Gomp** and **icx** perform adequately and roughly equally.

As this thesis was performed on an Intel cluster it would be interesting how different vendors or architectures impact these statistics. How does a program compiled with an Intel compiler perform



Plot 38: means across repetitions

Plot 39: linear regressions corresponding to
single, plot 28

Figure 7: **single**, gomp vs clang vs icc vs icx
delay length = 16384, affinity = socket-spread

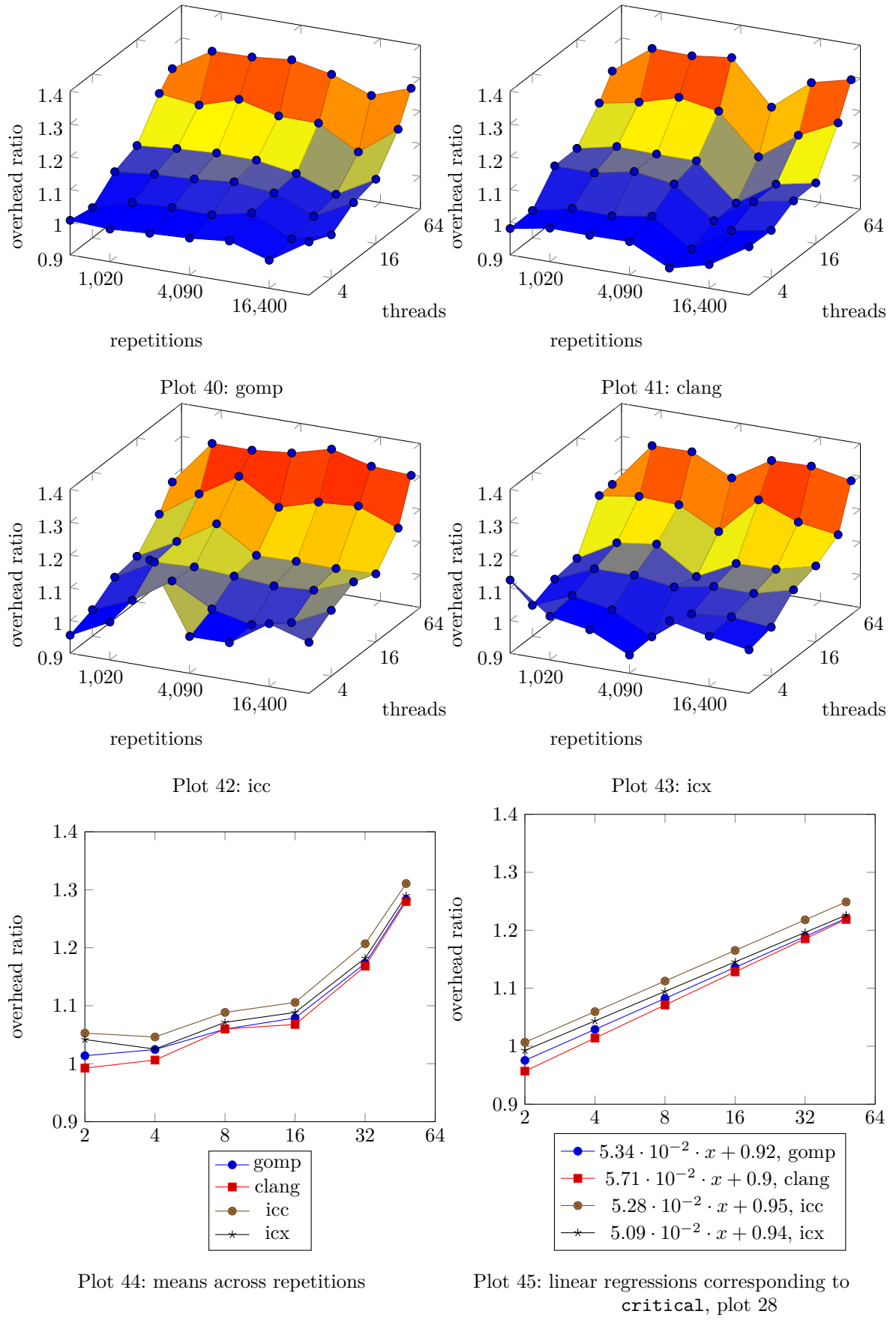
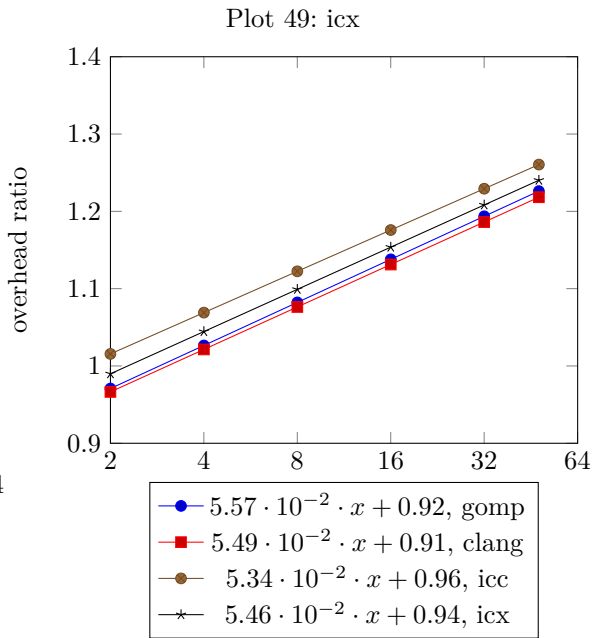
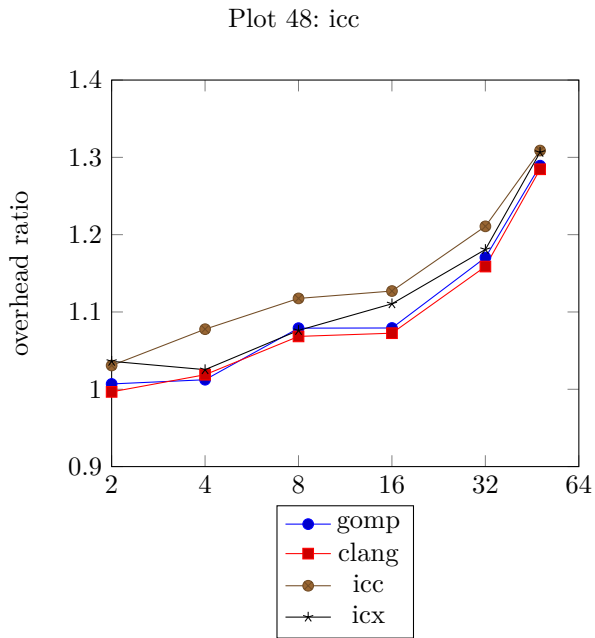
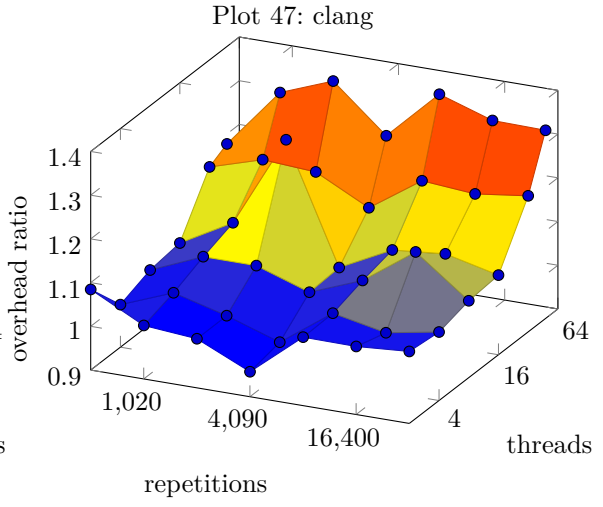
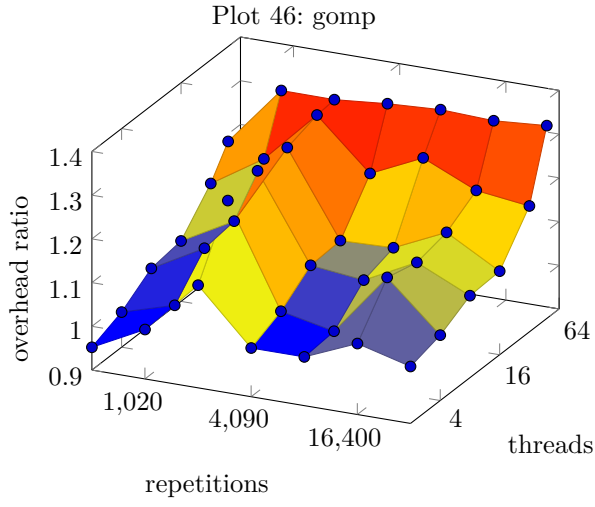
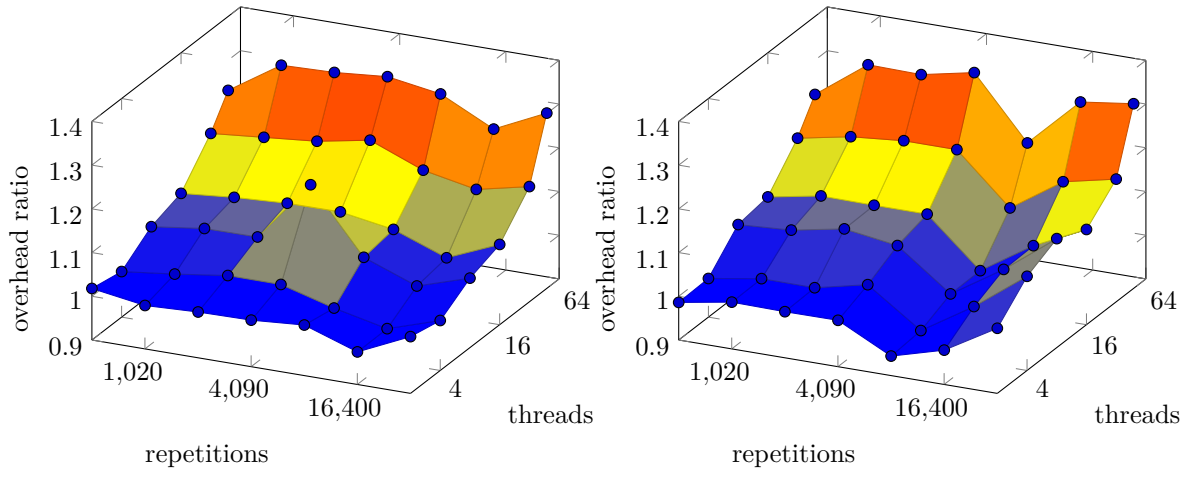


Figure 8: **critical**, gomp vs clang vs icc vs icx
delay length = 16384, affinity = socket-spread



Plot 50: means across repetitions

Plot 51: linear regressions corresponding to lock, plot 28

Figure 9: lock, gomp vs clang vs icc vs icx
delay length = 16384, affinity = socket-spread

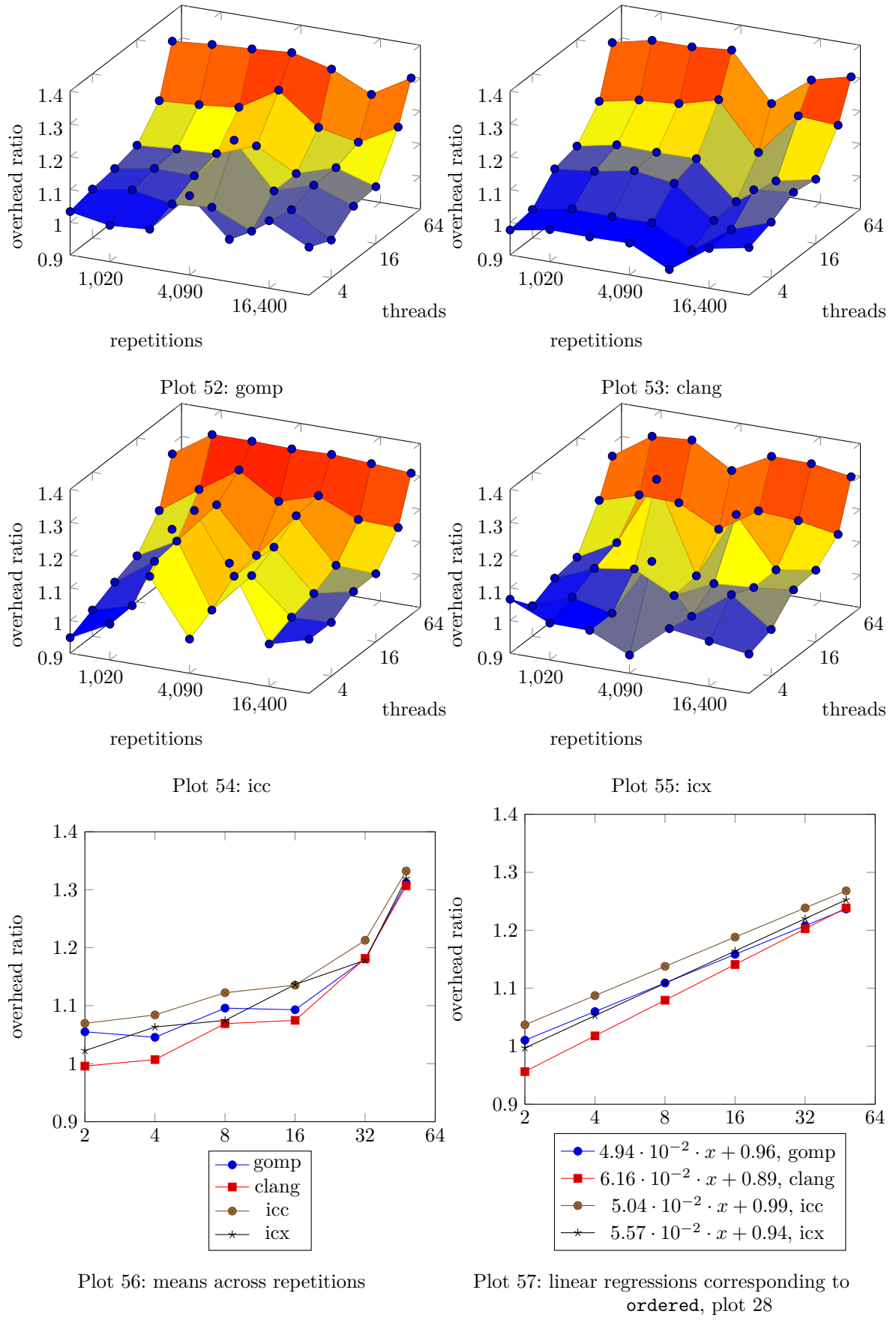
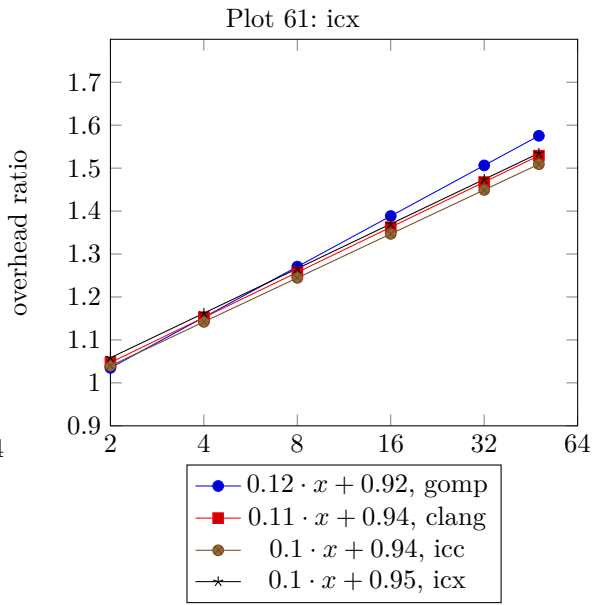
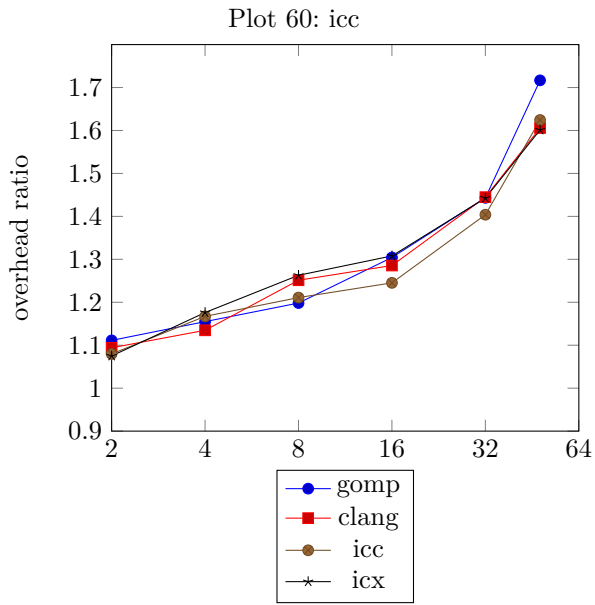
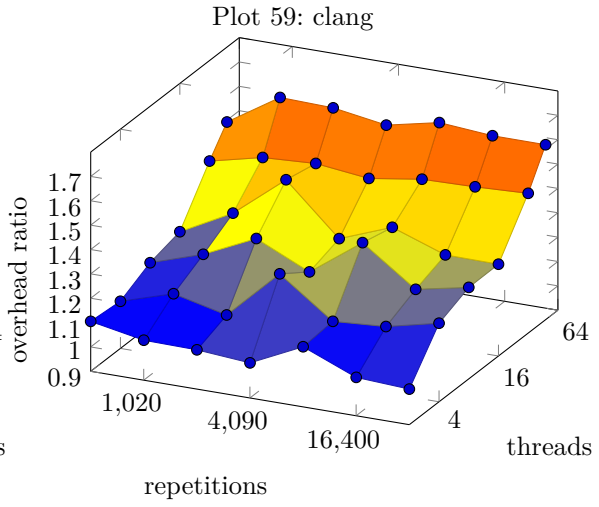
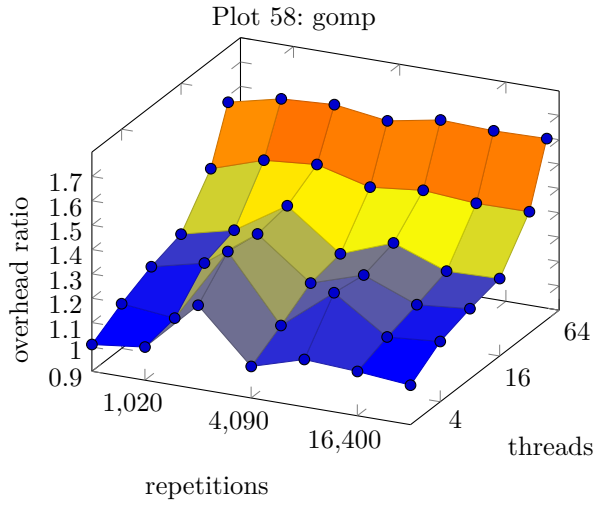
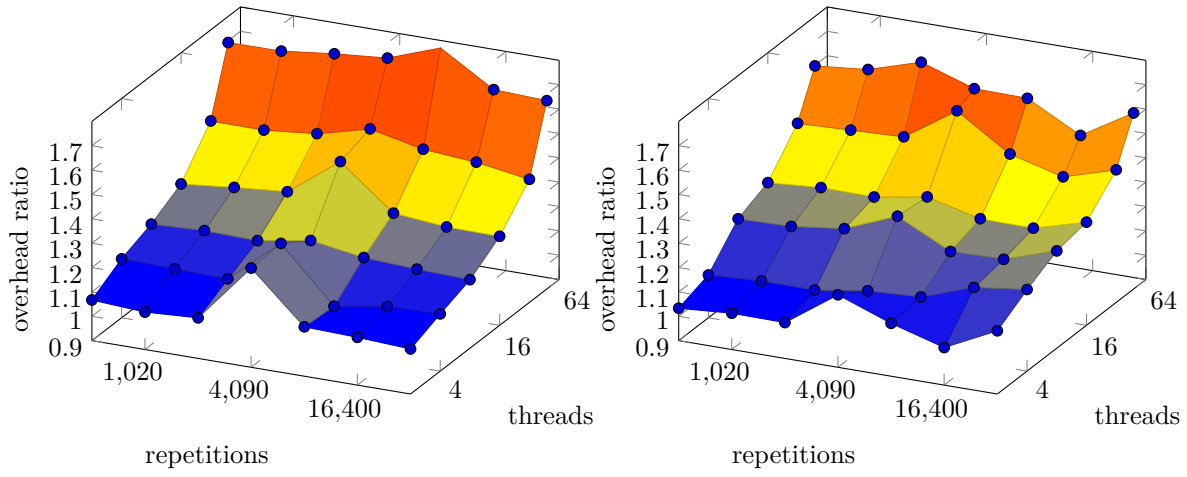


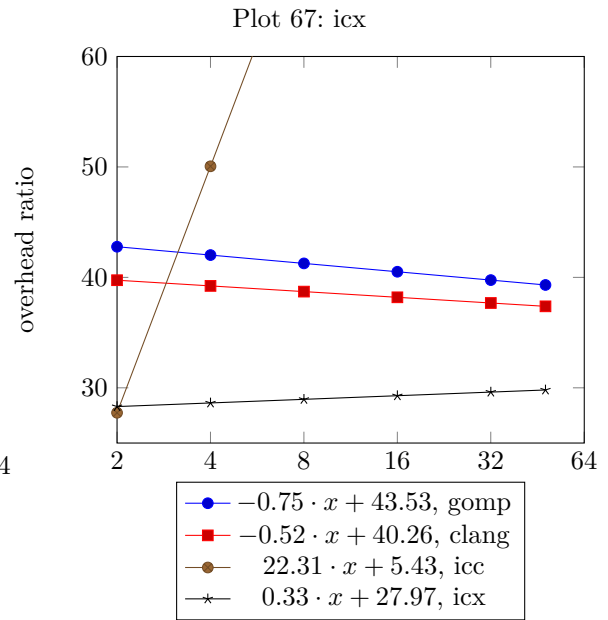
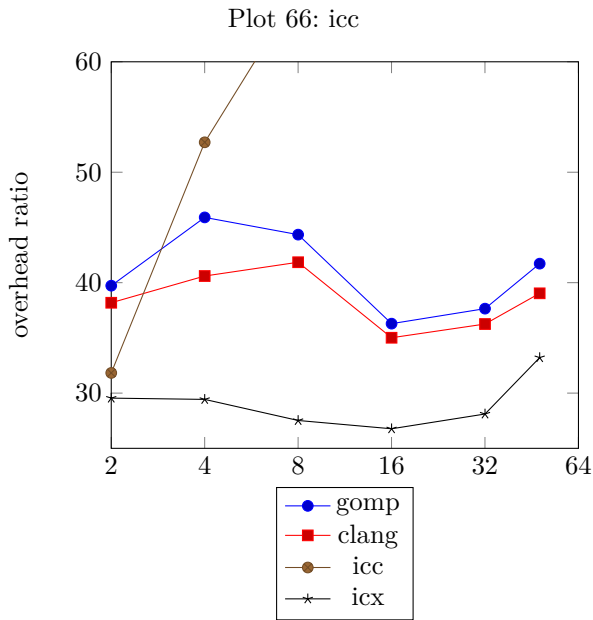
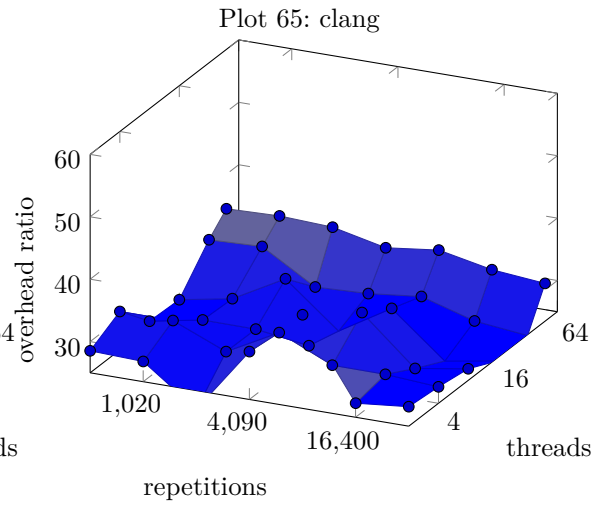
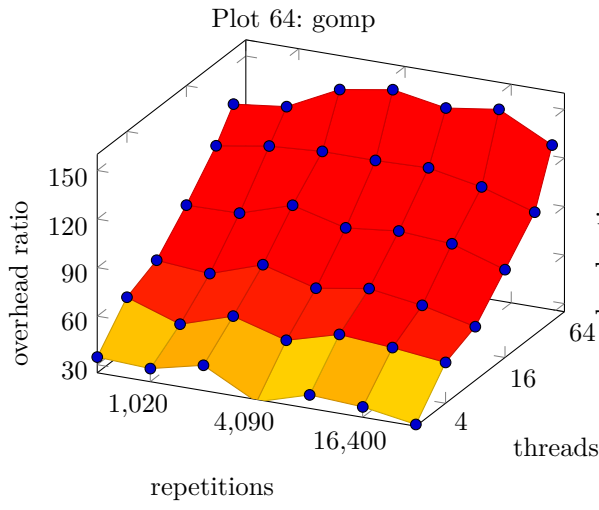
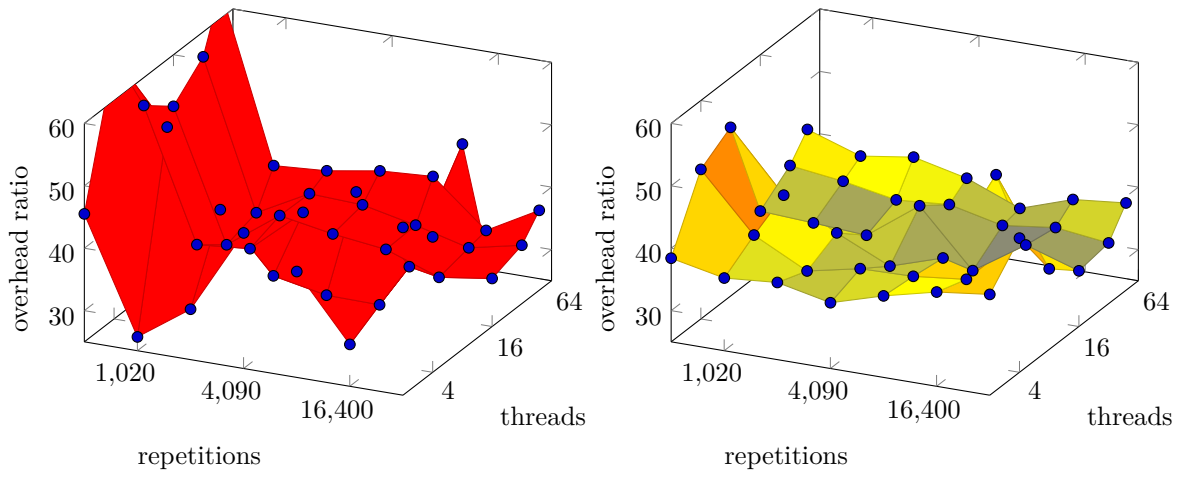
Figure 10: **ordered**, gomp vs clang vs icc vs icx
delay length = 16384, affinity = socket-spread



Plot 62: means across repetitions

Plot 63: linear regressions corresponding to
reduction, plot 28

Figure 11: reduction, gomp vs clang vs icc vs icx
delay length = 16384, affinity = socket-spread



Plot 68: means across repetitions

Plot 69: linear regressions corresponding to **atomic**, plot 28

Figure 12: **atomic**, gomp vs clang vs icc vs icx
delay length = 16384, affinity = socket-spread

on a non Intel machine? How do ARM or AMD perform? This is especially interesting considering AMDs and ARMs more affordable prices in comparison with Intel. OpenMP is also being developed further and in the future there will probably be new features that need to be evaluated as well.

References

- [1] URL: <https://www-staging.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/epcc-openmp-micro-benchmark-suite> (visited on 01/01/2022).
- [2] J Mark Bull. “Measuring synchronisation and scheduling overheads in OpenMP”. In: *Proceedings of First European Workshop on OpenMP*. Vol. 8. Citeseer. 1999, p. 49.
- [3] Mark Roth et al. “Deconstructing the overhead in parallel applications”. In: *2012 IEEE International Symposium on Workload Characterization (IISWC)*. 2012, pp. 59–68. DOI: 10.1109/IISWC.2012.6402901.