Development of a shared-memory parallel solver for contact mechanics

STUDENT PROJECT

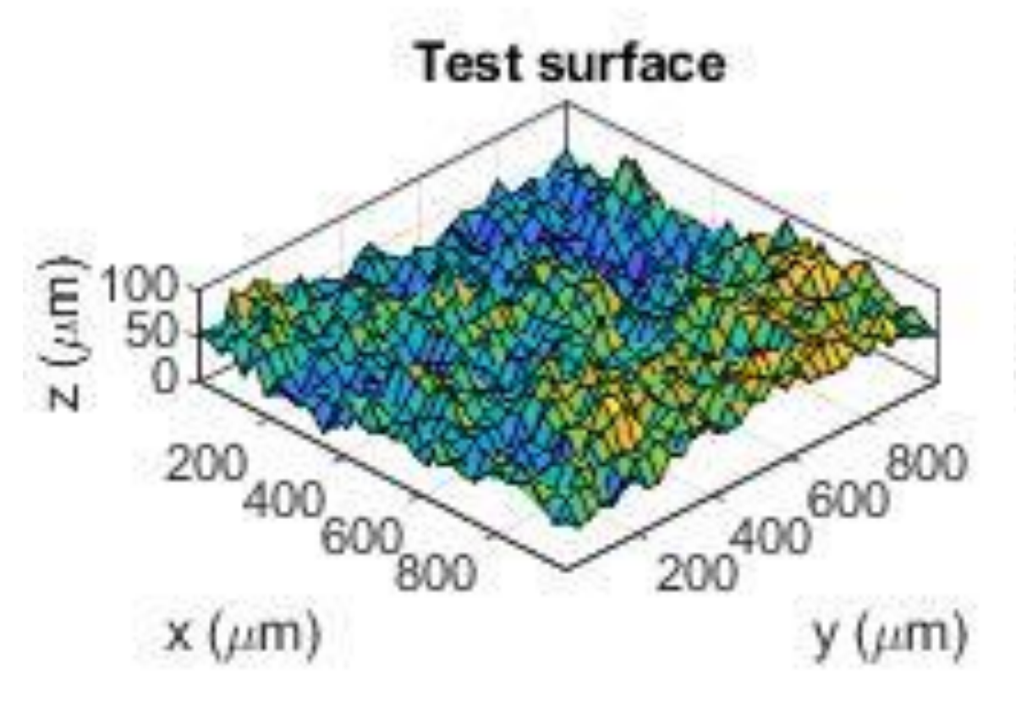
BAU

Henrik Bartsch, Nora Hagmeyer[1] and Alexander Popp[1]
[1]Institute for Mathematics and Computer-Based Simulation, University of the Bundeswehr Munich, Germany

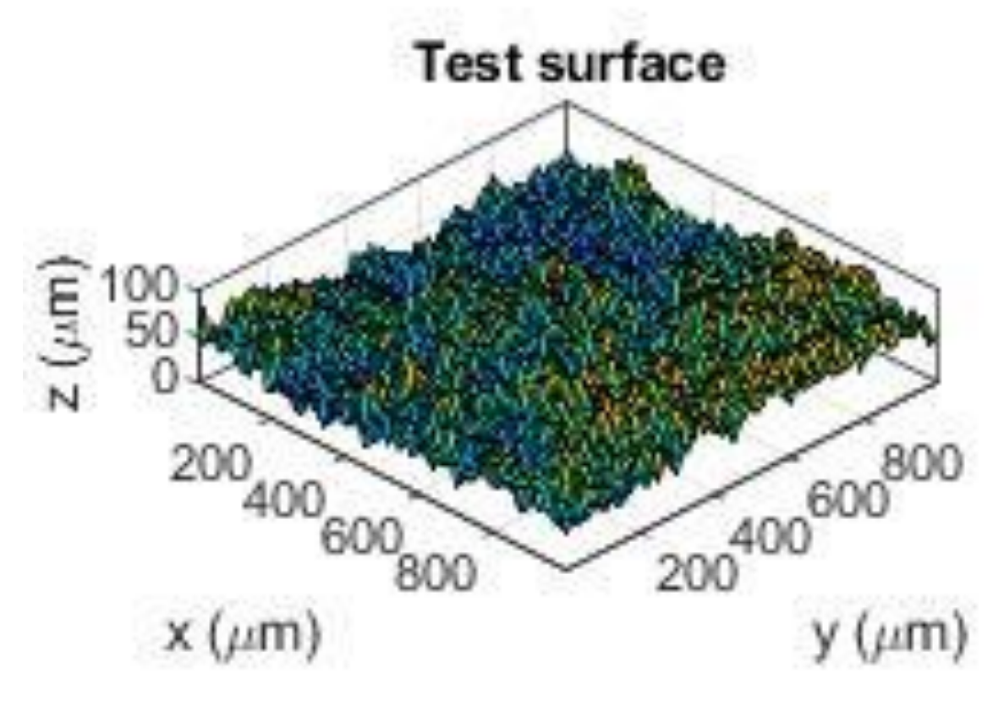
Universität der Bundeswehr München

Motivation

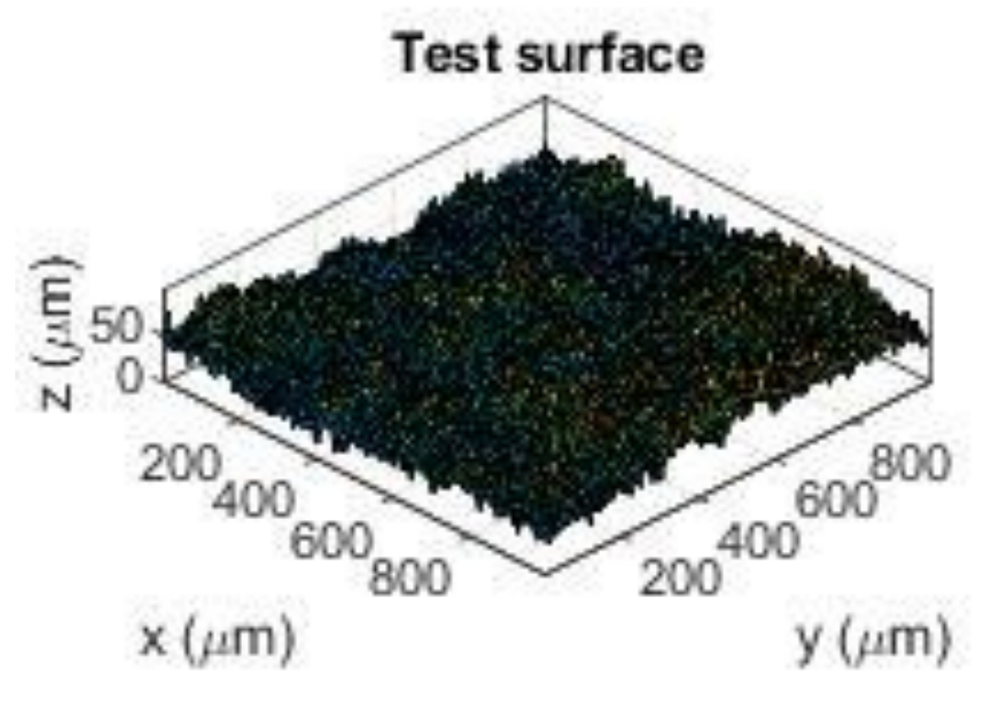
Contact mechanics between rough surfaces represents an increasingly important field or research. The boundary element method (BEM) is applied to solve the frictionless contact problem between two linear elastic rough surfaces. For sufficient accuracy this already complex problem requires finely resolved underlying surface mashes. Parallel programming can aid in achieving higher accuracy and decreasing execution time. On one hand, parallel programming increases hardware usage and code efficiency, but on the other hand requires more know-how and time to test.



Surface 5: 1024 nodes (sup5)



Surface 6: 4096 nodes (sup6)



Surface 7: 16000 nodes (sup7)

Parameters of parallel programming

Isolating specific code parts for examination of parallelization parameters

To analyze how OpenMP – a parallel programming library - and its parameters effect the code efficiency, repetitive tasks have been isolated. The isolated tasks are:

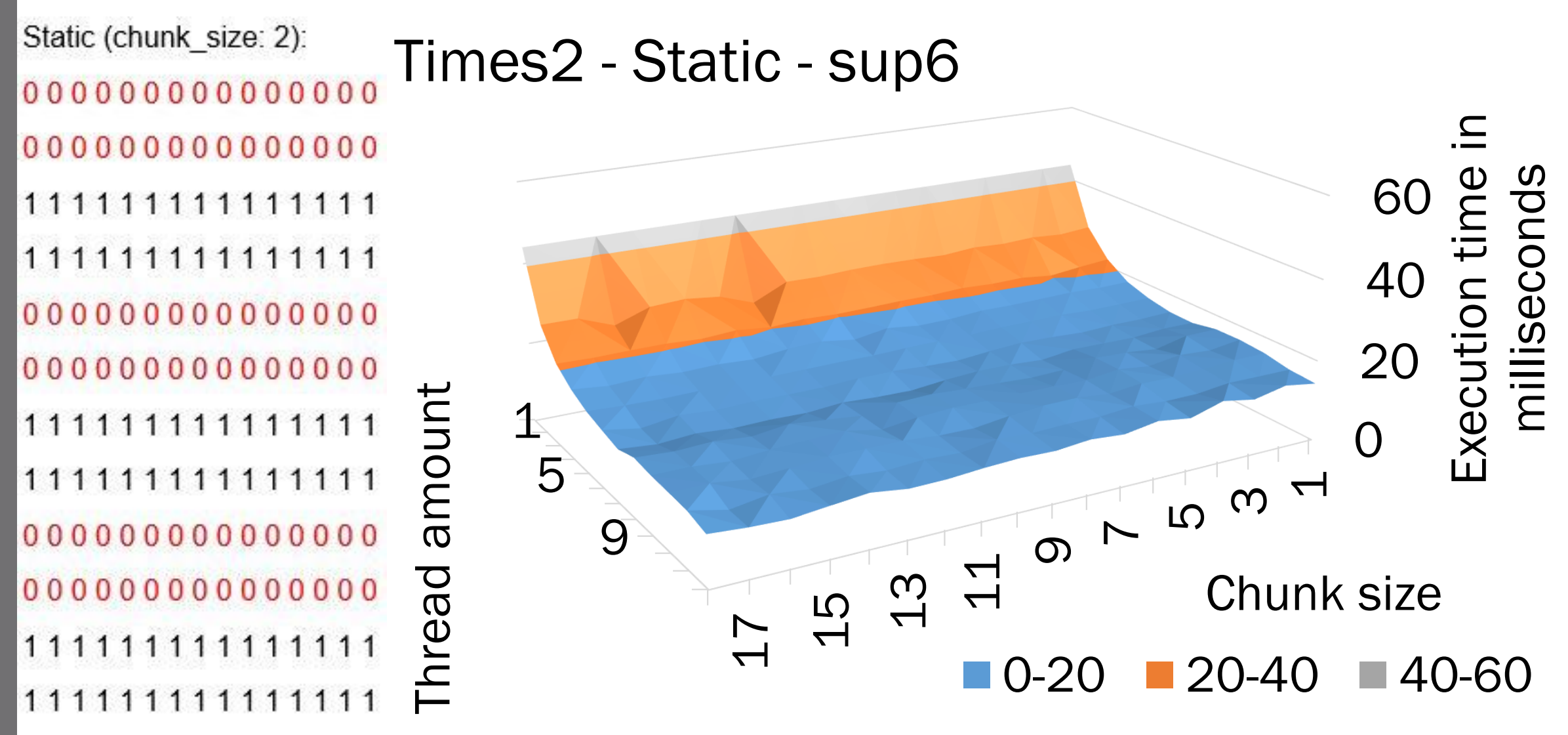
1. "Times1": Filling elements of a matrix with values
2. "Times2": Solving matrix-vector-product
3. "Times3": Solving matrix-vector-product with indirect addressing

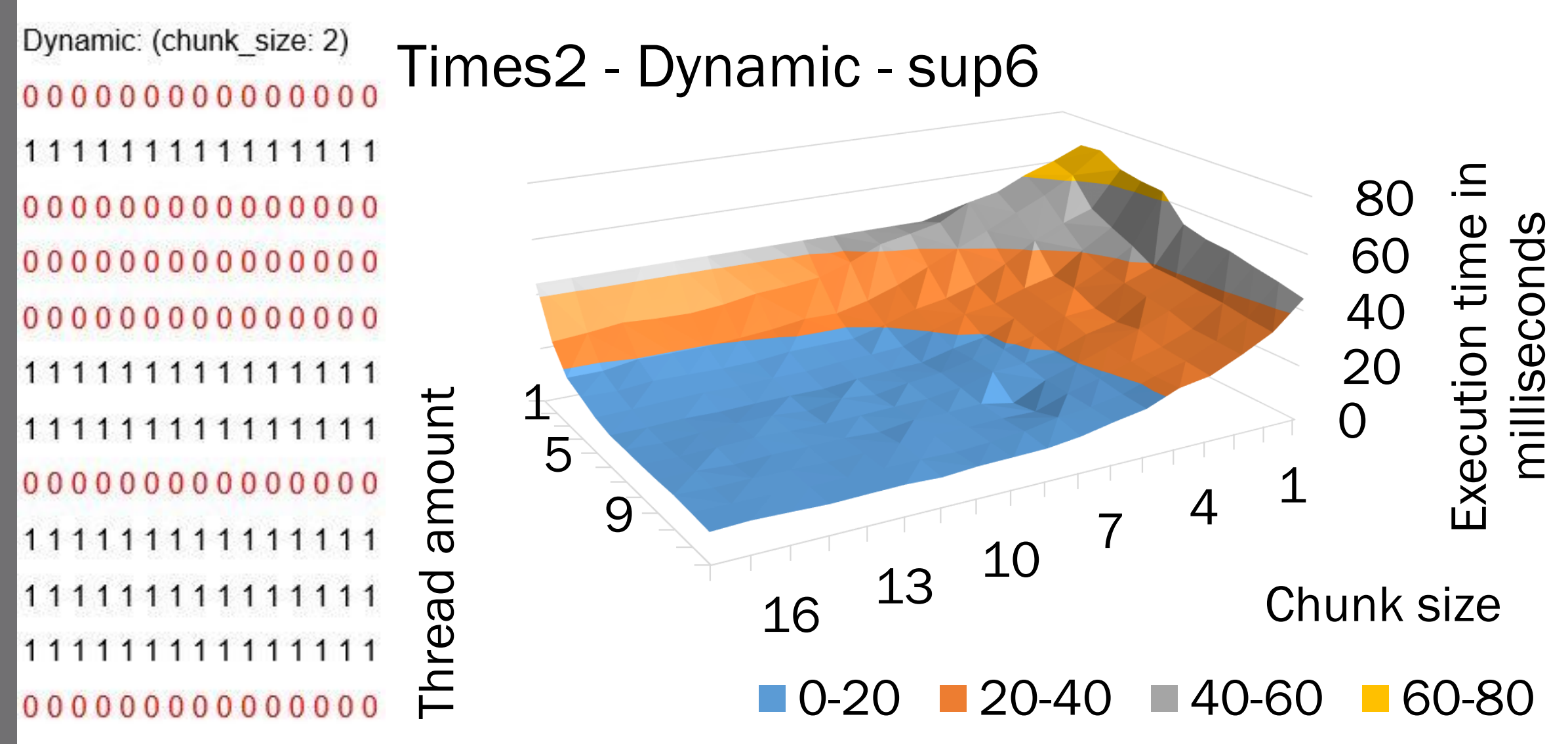
Parameters of parallel programming

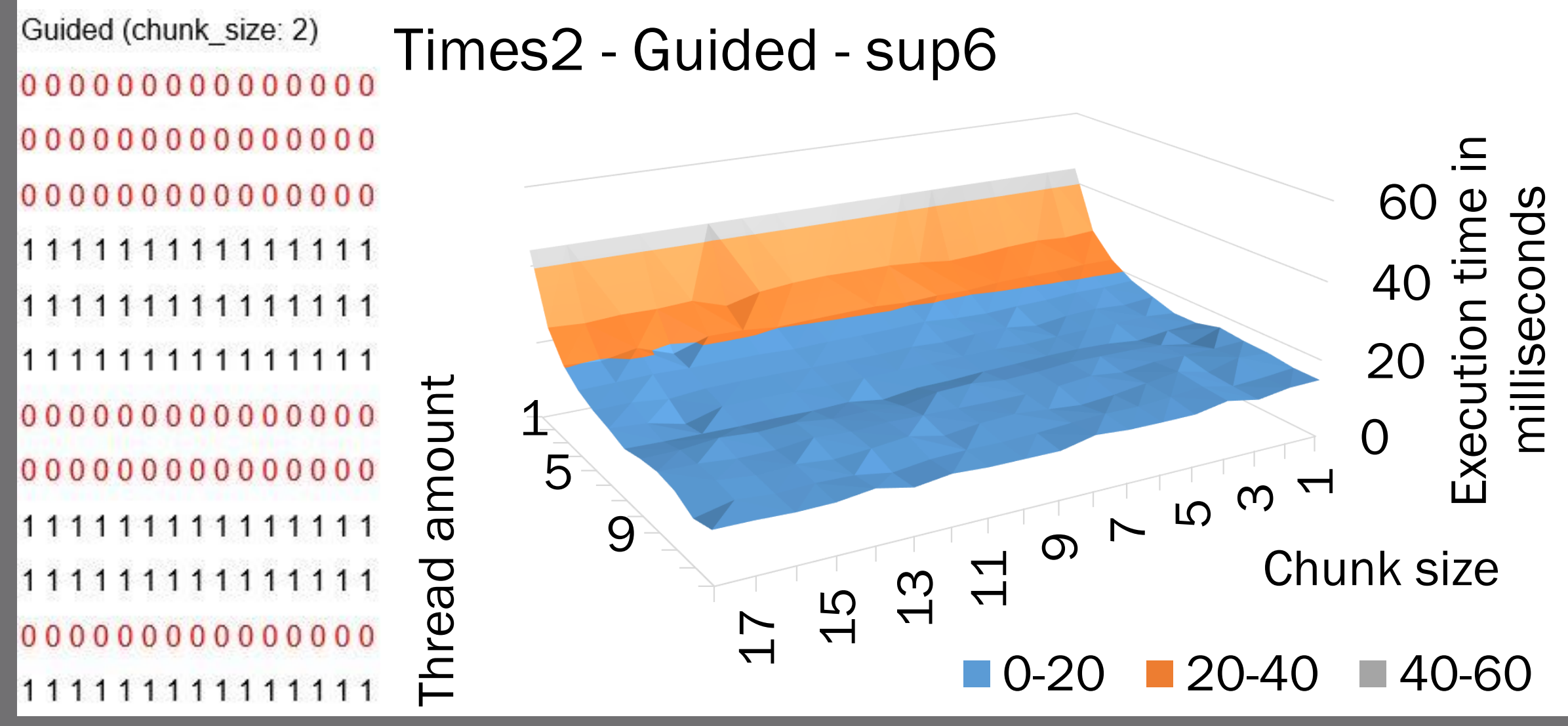
There are three distinguishable parameters within parallel programming to keep in mind:

The first parameter is the thread amount. The thread amount determines, how many processes are allocated to perform a task.

The second parameter is the chunk size, which is considered to be the amount of iterations requested for each thread per cycle.





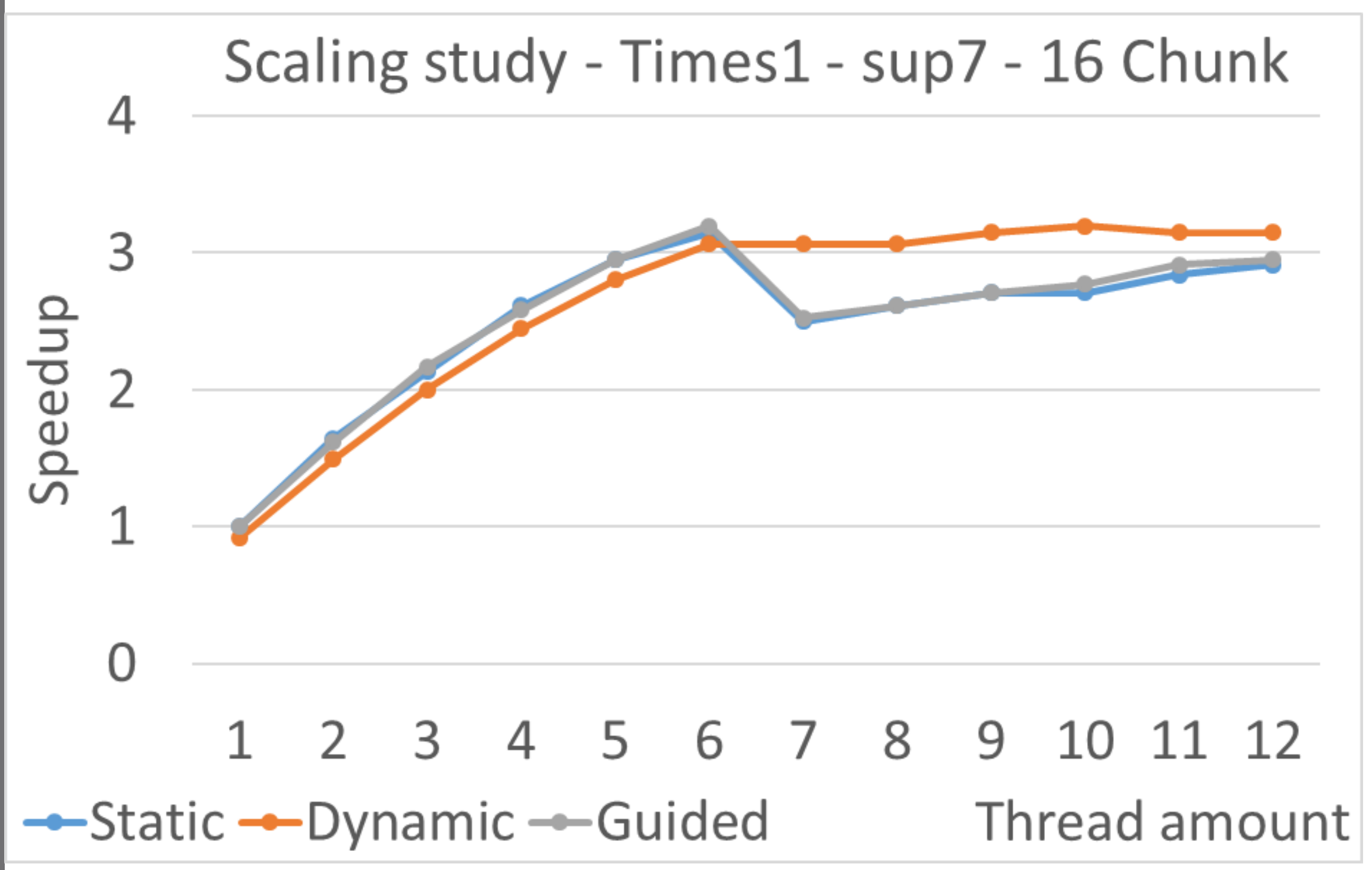


The third parameter is the scheduling type. Shown here are the three scheduling types for parallel loops and their respective effects on execution time and how the distribution of the workload across threads works. On the left side, the visualizations show how the workload is distributed across threads 0 and 1 for a chunk size of 2. On the right side, images present a visualization on how chunk sizes and thread amount affect execution time for the different scheduling types.

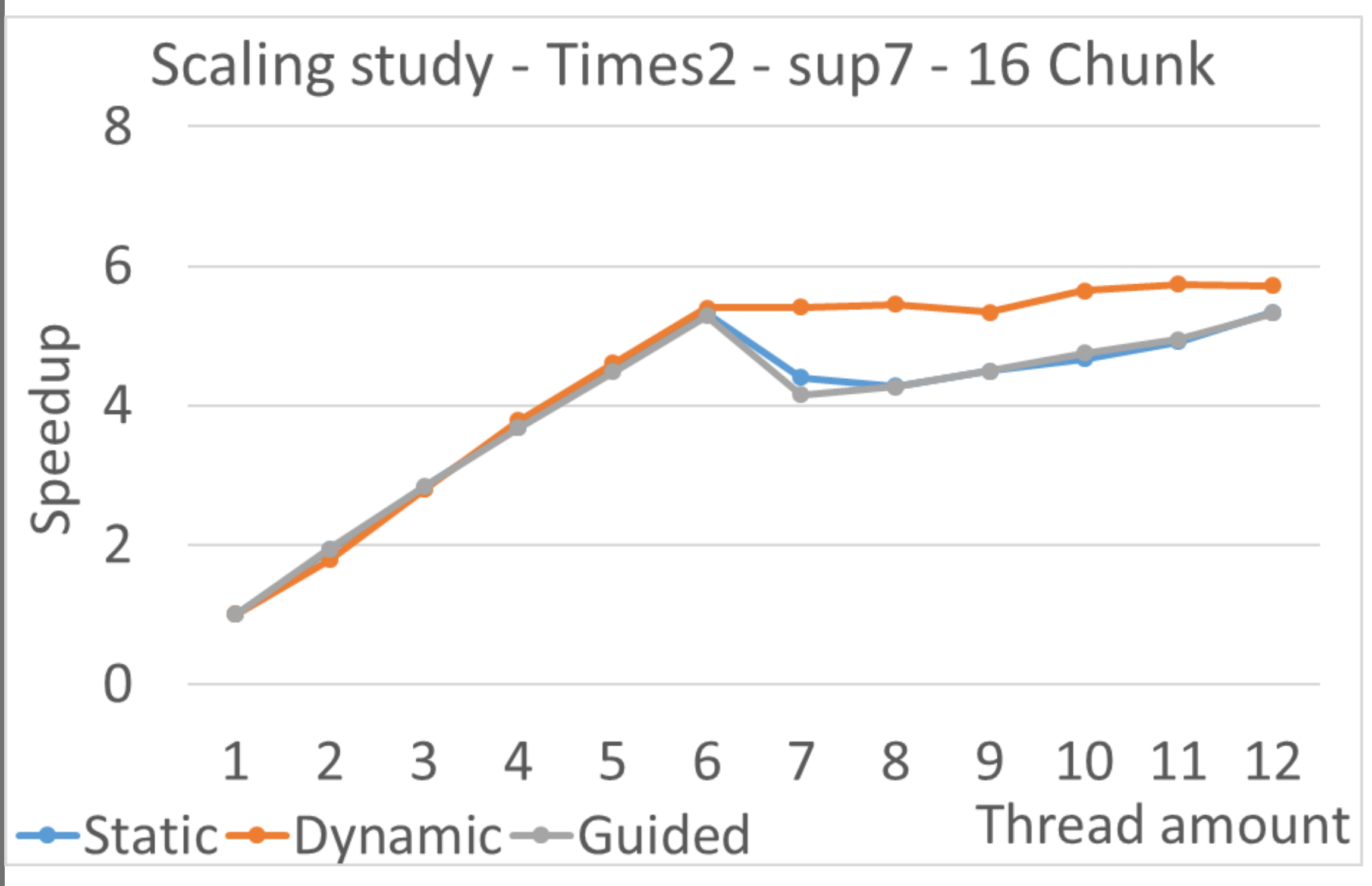
Strong scaling results for the model problems

Conditions of the strong scaling study

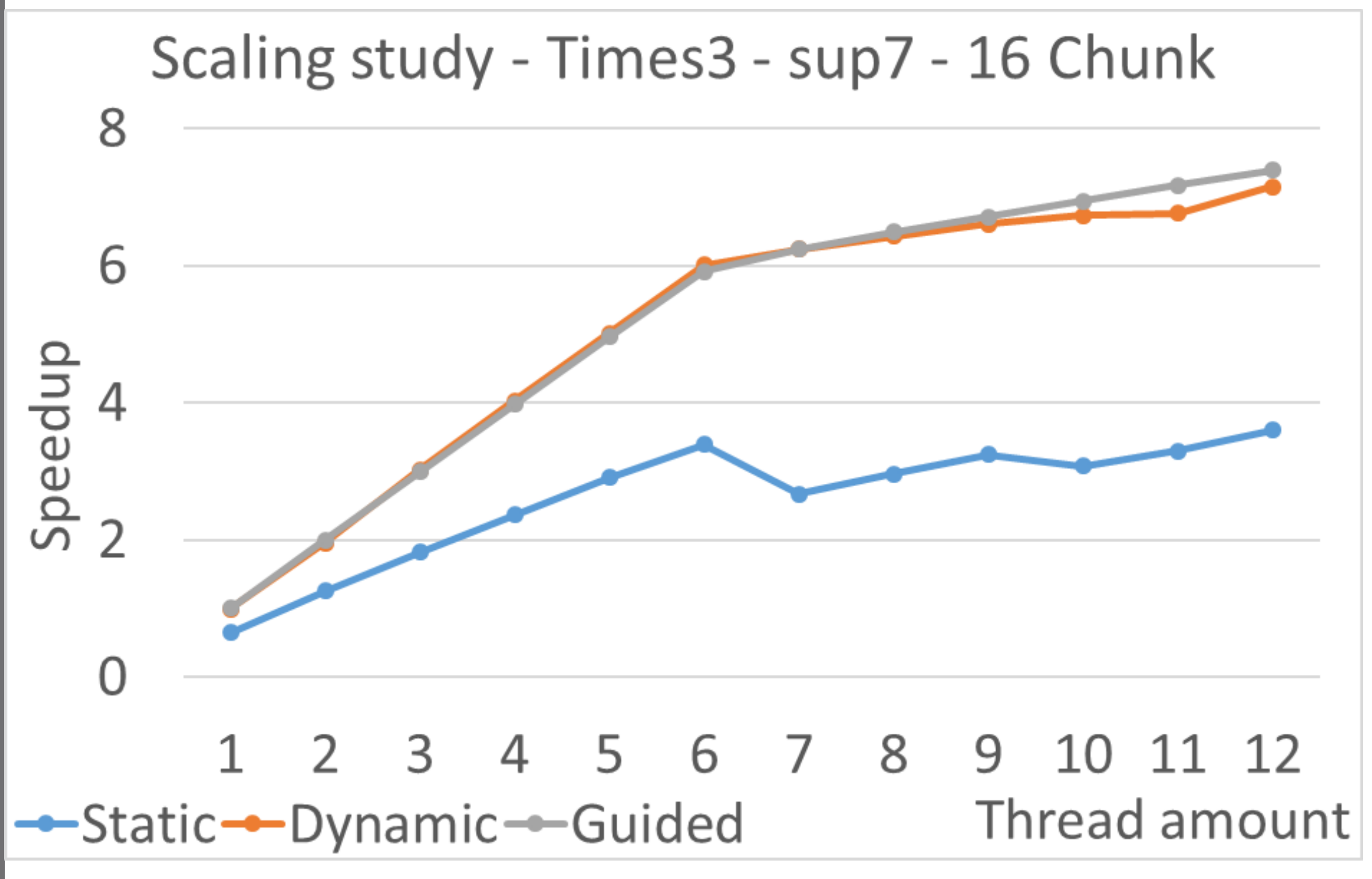
These scaling studies have been generated on a computer with an Intel Xeon CPU E5-1650. This processor provides 6 cores capable of utilizing hyper-threading technology, essentially doubling the available hardware thread amount, raising it to 12 threads in total.



- Does not scale, overhead seems to be large in comparison to execution time.
- Kink after 6 threads suggests that hyper-threading cannot be efficiently utilized using scheduling types static and guided. "Dynamic" counteracts this negative effect caused by hyper-threading and increases speedup slightly.

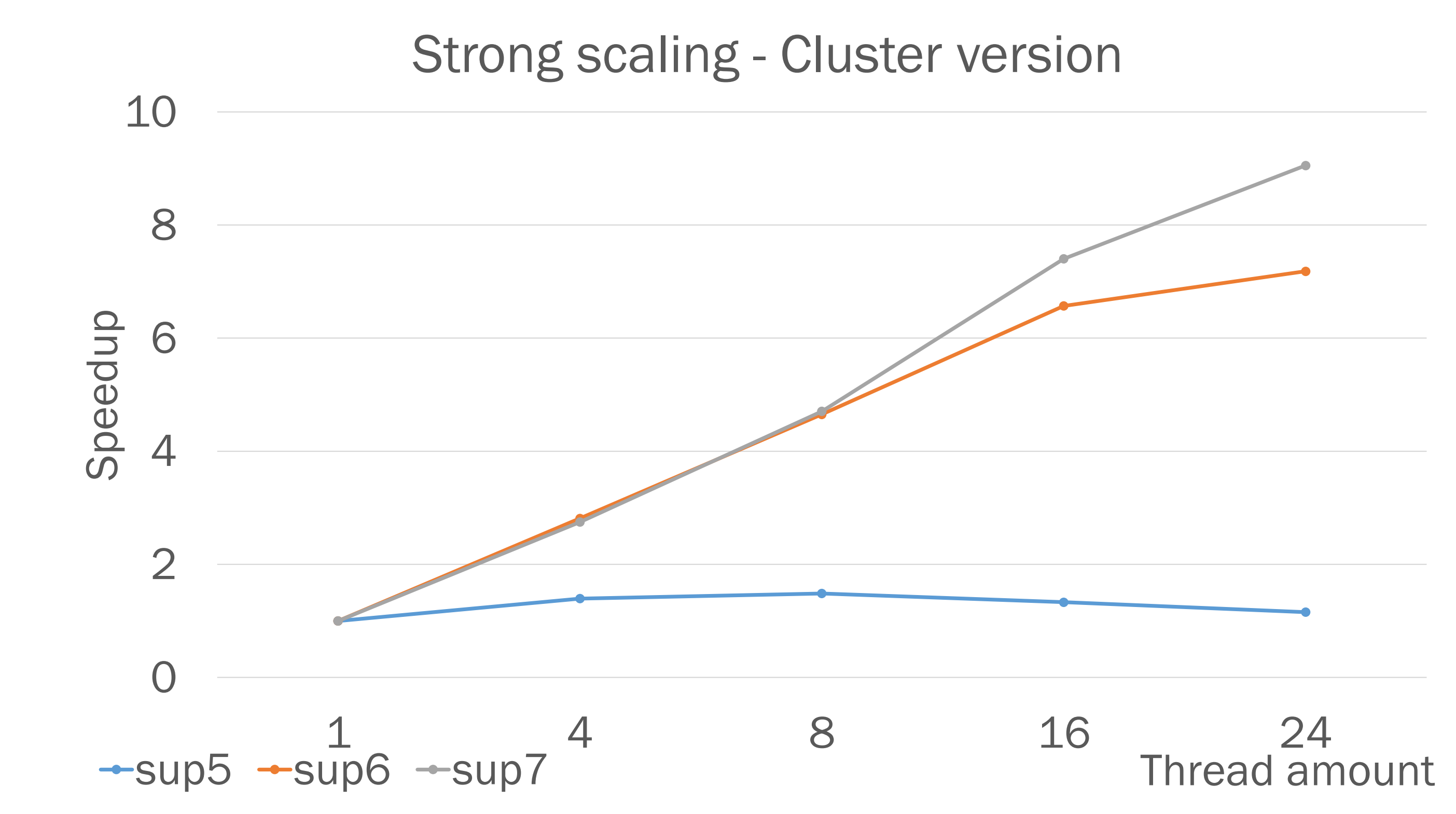


- Almost perfect scaling for all schedules up to 6 threads.
- Same falloff in speedup as before regarding „static" and „guided" for more than 6 threads.



- Scales perfectly for scheduling types "dynamic" and "guided" up to 6 threads.
- Indirect addressing leads to differences in terms of execution time of the iterations. Cause of this are cache and memory latencies. Data suggests that "static" is not optimal for those circumstances.

Strong scaling results for the BEM code



Cluster version: (generated on compute node with 2 Intel Xeon Gold 5118, each processor is capable of hyper-threading on up to 12 cores)

- sup5 scales much worse in comparison to the other two problem sizes, since sup5 is too small to make efficient use of parallel programming in this instance.
- Sup6 and sup7 share nearly the same speedup until they reach 8 threads.
- Sup7 scales better than sup6 does after 8 threads. This suggests that the drop-off is due to the problem size.