The 9th European Congress on Computational Methods in Applied Sciences and Engineering
ECCOMAS Congress 2024
3–7 June 2024, Lisboa, Portugal

NODE-LEVEL PERFORMANCE ANALYSIS OF THE STRUCTURAL MECHANICS SOLVER B2000++PRO

N. Ebrahimi Pour[1] and H. Klimach[1]

[1] German Aerospace Center (DLR)
Institute of Software Methods for Product Virtualization
Zwickauerstraße 46, 01069 Dresden, Germany
e-mail: (neda.ebrahimipour, harald.klimach)@dlr.de, https://www.dlr.de/de/sp

Key words: Finite element method, Structural mechanics, High-performance computing, Node-level performance

Summary. Fluid-structure interactions are routinely simulated in the field of aerospace engineering. These simulations are complex, but essential in the design process of devices such as aircrafts or turbomachinary. They involve multiple scales and multiple physics, and typically require massively parallel systems for their realization. However, small surfaces can become a bottleneck for the data exchange. The structural part often covers only a small portion of the computational effort, but can induce idle times in the overall computation. To use available computational resources efficiently, scalable methods and highly parallel software packages are required. This study examines the node-level performance of the structural mechanics solver `b2000++pro` on three distinct supercomputing architectures, with the objective of identifying an optimal configuration for the solver on these systems.

1 INTRODUCTION

Finite element methods constitute an indispensable foundation in contemporary engineering applications. Nevertheless, the simulation of comprehensive models with a multitude of degrees of freedom is a computationally onerous process, which constrains the overall productivity of the aforementioned applications. The complexity and difficulty of realising simulations is further compounded when considering fluid-structure interactions. In order to employ highly resolved simulations and meet the time-to-solution expectations, it is possible to utilise high-performance computing systems. This study examines the node-level performance of the structural solver `b2000++pro` [1, 2], designed to solve problems related to structural mechanics in a wide range of application domains. The objective of this study is to examine the performance of the solver at the node level on various high-performance computing architectures. The investigation uncovers potential avenues for enhancing the performance of the highly flexible C++ code, which employs Intel's Threading Building Blocks (Intel-TBB) [3] for shared memory parallelization and the OpenMP [4] parallel MUMPS direct solver [6, 7] to address the arising linear equation systems. In the context of the considered applications, a run comprises the following steps: reading the mesh, generation of a model representation, assembly of the system matrices and solution of the corresponding linear equation systems. The assembly of the system matrices and the solution of the linear equation systems are typically the most time-consuming steps in this procedure. This contribution further discusses potential avenues for narrowing the performance gap and

accelerating the time to solution of b2000++pro on the aforementioned HPC systems. The following structure is adopted for this paper: first, the structural solver (b2000++pro) and the existing parallelism are introduced; then, the node-level performance on the various systems is presented and discussed; thereafter, a summary is given and a conclusion drawn.

2 STRUCTURAL SOLVER b2000++pro

The solver is a general-purpose structural mechanics solver, dedicated to solving various finite element (FE) problems, with a particular emphasis on shell and composite structures in lightweight construction, as well as buckling and post-buckling. b2000++pro is capable of solving a range of structural mechanics problems, including both linear and nonlinear problems, as well as conducting eigenvalue and damage analyses on laminates. It is written in modern modular C++ and features a plugin infrastructure with exchangeable parts for user-written code, such as user-defined elements, materials, or solvers for different problems. Its flexibility and wide application range makes it suitable for addressing a variety of structural mechanics problems [1].

Exsisting parallelism of b2000++pro

Figure 1 depicts the existing parallelism of the solver. The solver is predominantly parallelized using Intel Threading Building Blocks (Intel-TBB) on a shared memory platform. Distributed memory parallelism is only employed by the linear algebra package MUMPS, which is hybrid parallel (OpenMP & MPI [5]). The overall concept is based on a main/worker paradigm, wherein a single main MPI process is responsible for managing the overall problem, while the MPI workers actively contribute to the computation by solving the linear equations. The two regions with the Intel-TBB parallelism and the Hybrid MPI, OpenMP solving of linear equation systems are not overlapping. That is the case during the initialisation and matrix assembly, as well as during the post-processing and output of the results, where only the main process is active. Conversely, during the solution of the linear algebra, all MPI processes (1 ... N), including the main MPI process, are involved.

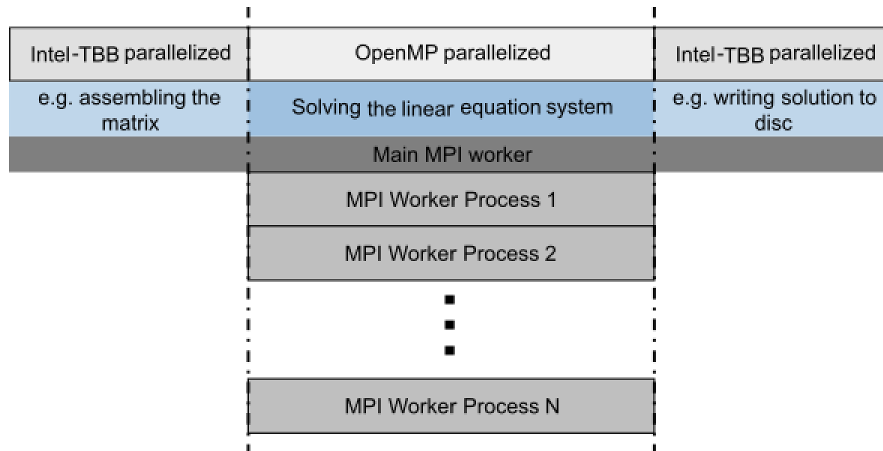

Figure 1: Existing parallelism in b2000++pro

3 INVESTIGATION OF THE NODE-LEVEL PARALLELIZATION

In this section the results of the node-level performance for the structural mechanics solver `b2000++pro` will be examined. To conduct our investigation we utilize three supercomputing systems, with a single full node allocated for each of our runs. The analysis is performed taking into account MPI and shared memory parallelism. As previously stated, the structural mechanics solver is parallelized using Intel-TBB, while the underlying linear algebra solver MUMPS incorporates OpenMP and MPI for computation. To comprehensively explore the available parallelism, a single node with diverse configurations is considered, and its behavior on different computing systems is investigated.

Computing sytems - Machine configuration

Our studies are carried out on two supercomputers of the German Aerospace Center (DLR), namely CARA and CARO, and on one of the German national supercomputers called HAWK, located at the High Performance Computing Center Stuttgart (HLRS). The specifications of the system are as follows:

- CARA (DLR)
  - 2,168 CPU-nodes with 2 AMD EPYC 7601 (2×32 cores)
  - 664 CPU-nodes with 2 AMD EPYC 7702 (2×64 cores) (Extension in 2023)
  - 17 PB lustre file system
  - Operational since 2020
- CARO (DLR)
  - 1,364 CPU-nodes with 2 AMD EPYC 7702 (2×64 cores)
  - 8.4 PB Lustre file system
  - Operational since 2022
- HAWK (HLRS)
  - 5,632 CPU-nodes with 2 AMD EPYC 7742 (2×64 cores)
  - $2 \times$ Lustre file systems available (22 PB and 15 PB)
  - Operational since 2020

For our investigation we consider a full node (2 x 64 cores) using the AMD EPYC 7702 on CARA and CARO and the AMD EPYC 7742 CPUs on HAWK. The main difference between the two CPU types is the base clock, which is 2.0 GHz and 2.25 GHz, respectively. The source code has been compiled using the GCC version 10.4.0 compiler, while MUMPS, the linear equation solver, has been compiled with OpenMPI version 4.1.5 and MPI.

Structural mechanics - Test case description

For our study we consider a simple three-dimensional unit cube with $50 \times 50 \times 50$ elements in each spatial direction. The cube is fixed at the bottom, resulting in a displacement of 0.0 in this spatial direction. An external force F acts uniformly on the top of the cube. The Young's modulus is set to 207 914.0 MPa (steel) and the Poisson's ratio is 0.28342.

Results - CARA computing system

The objective of this investigation is to analyse the node-level performance, whereby a constant number of threads is initially employed, while the number of processes is varied. Figure 2a illustrates the measurement on CARA. Each curve represents a measurement with a specific thread count, with the number of processes doubled from point to point in the data series. The x-axis provides the total amount of utilized cores given by the product of number of processes and number of OpenMP threads. Note, that for the region where only the main process is active, only a number of cores equal to the number of threads is actually used by the program. The optimal time to solution for this particular test case is achieved when 8 threads and 16 processes are used, as indicated by the last measurement point of the red curve. This is followed by the green and purple curves, representing 4 threads and 32 processes, and 16 threads and 8 processes, respectively. Thus, it is beneficial to make use of all cores in the node. Figure 2b

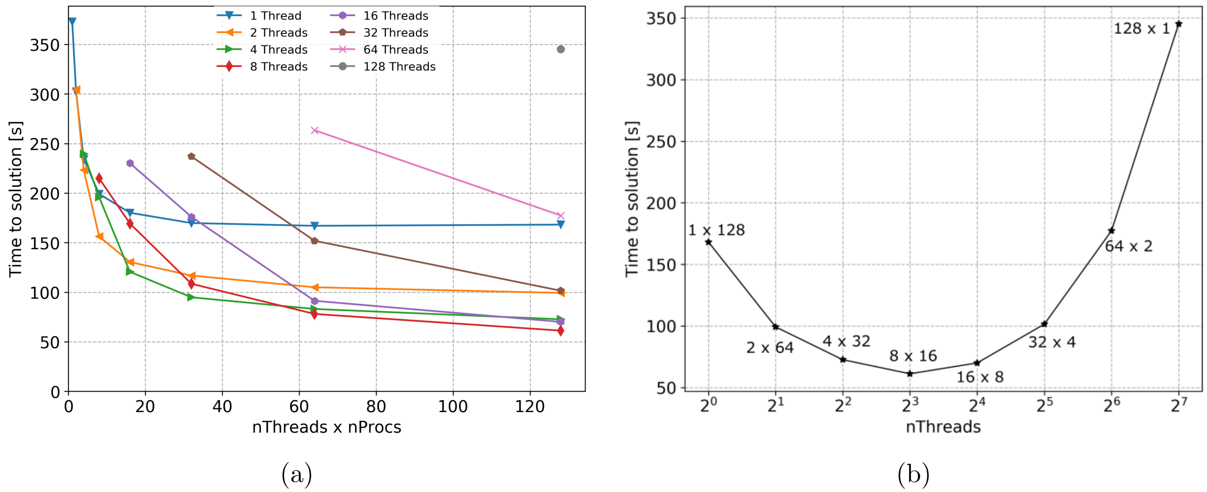

Figure 2: Node-level performance on CARA: (a) Fix number of threads, represented by each curve, the number of processes are doubled for each subsequent point. (b)Full node configuration using nThreads × nProcs (last data point of each curve in Figure (a)) for each data point.

presents the computational time for a full node, which is comparable to the last data point of each curve in Figure 2a. Here the x-axis indicates the number of threads in the computation, and the number of processes is implied accordingly, falling from 128 processes on the left to a single process on the right end. It is evident that the time required to reach the solution decreases with an increase in the number of threads up to 8, but then begins to increase again with an increase in the number of threads. This continues until the time required to reach a solution is at its greatest with 128 threads and a single process. On this system 4 cores share a single L3 cache, so we may expect the optimum for the shared memory parallelism to end up at 4 threads. However, as one part of the computation only benefits from the shared memory parallelism, the optimum gets slightly shifted to 8 threads instead as this allows for a better utilization of the node when considering the overall execution. But what we can also discern is that the benefit of using more threads for the shared memory parallel part is limited and already exhausted when increasing the thread count beyond 8. This is because the solving of

linear equations in MUMPS seems to benefit more from MPI parallelism rather than more than 4 threads in the shared memory parallelization. The speed up for the combinations of 4×32, 8×16 and 16×8 (threads $\times$ processes) are 5.1, 6.1 and 5.3, respectively. It is calculated by comparing the time to solution of the respective parallel run with the sequential run.

Results - CARO computing system

Similar to the analysis on CARA, Figure 3a shows the time to solution over the utilized cores, given by Thread count $\times$ Process count, for the CARO system. Again each line represents a fixed number of threads, while the number of processes is doubled with each subsequent point. For this system we now find the combination of 32 processes with 4 threads to provide the fastest computation. This configuration is followed by the combination of 8 threads and 16 processes, which is only slightly slower than the optimum. Also 16 threads and 8 processes still yields a comparable time to solution. The change of the system here resulted in a slightly different optimal configuration despite the same processor architecture, tending towards larger benefits from the MPI parallelism and hence using more MPI processes rather than threads as compared to CARA. The time to solution for a full node is depicted in Figure 3b. A comparison of the

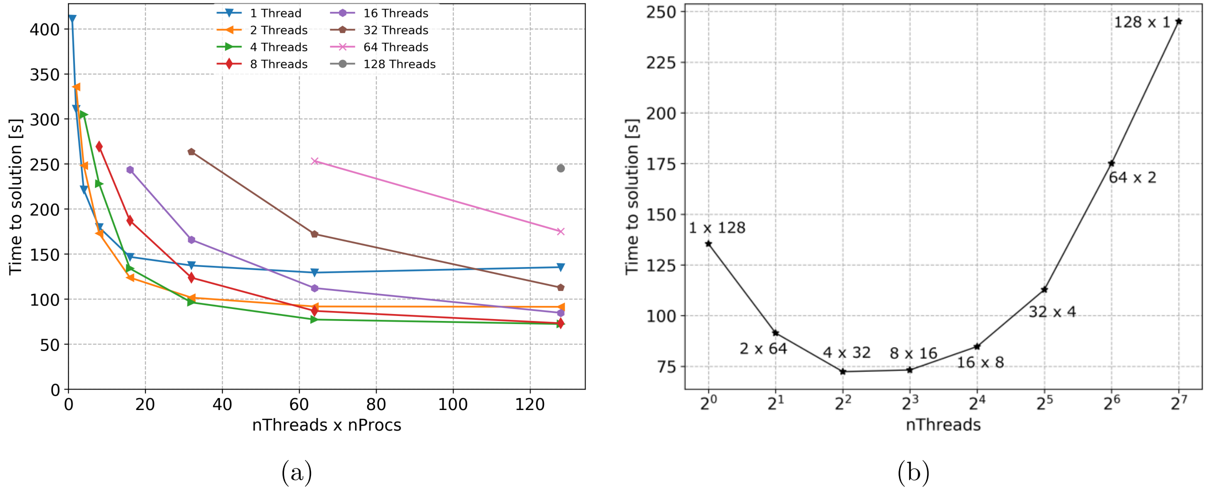

Figure 3: Node-level performance on CARO: (a) Fix number of threads, represented by each curve, the number of processes are doubled for each subsequent point. (b) Full node configuration using nThreads $\times$ nProcs (last data point of each curve in Figure (a)) for each data point.

time to solution for 4 threads $\times$ 32 processes with the sequential run, in which a single thread and process were utilised, reveals a speed up of 5.7 for this configuration. In the case of 8×16 and 16×8 (threads $\times$ processes), a speed up of 5.6 and 4.8, respectively, is observed.

Results - HAWK computing system

The same measurement previously demonstrated for the computing systems CARA and CARO is now applied to the computing system HAWK. The respective results are provided in Figure 4a. The most rapid computation on a full node is achieved through the utilisation of

8 threads and 16 processes, a result that is comparable to that observed in the CARA system. Subsequently, the combination of 16×8 and 4×32 (threads $\times$ processes) is considered. The speed up for the fastest computation is 2.7 in comparison to the run with 1 thread and 1 process. A comparative analysis of the first two curves, representing the behaviour of the 1 thread and 2 threads scenarios, respectively, reveals a divergence from the observed behaviour in the runs on the CARA and CARO computing systems. The time to solution decreases with an increasing number of processes for each curve on the aforementioned systems. In contrast, on HAWK, the curves representing 1 and 2 threads increase in computational time after a certain amount of processes and reach a steady behaviour. This unexpected behavior is attributed to the energy policy of the system, which aims to reduce energy consumption by decreasing the base clock when increasing the workload on the node. Consequently, the time to solution increases for the `b2000++pro` execution, where some parts are only computed by the main process with its threads, when these few cores get slowed down due to the full utilization in the solving of the linear systems, an overall slowdown occurs. Nevertheless, as shown in Figure 4b with the full node configurations, the general behavior and optimal distribution between threads and processes is similar to the observation on the other AMD systems, with the lowest execution time achieved by 16 processes with 8 threads each. Certainly, the combination of 4×32, 8×16 and

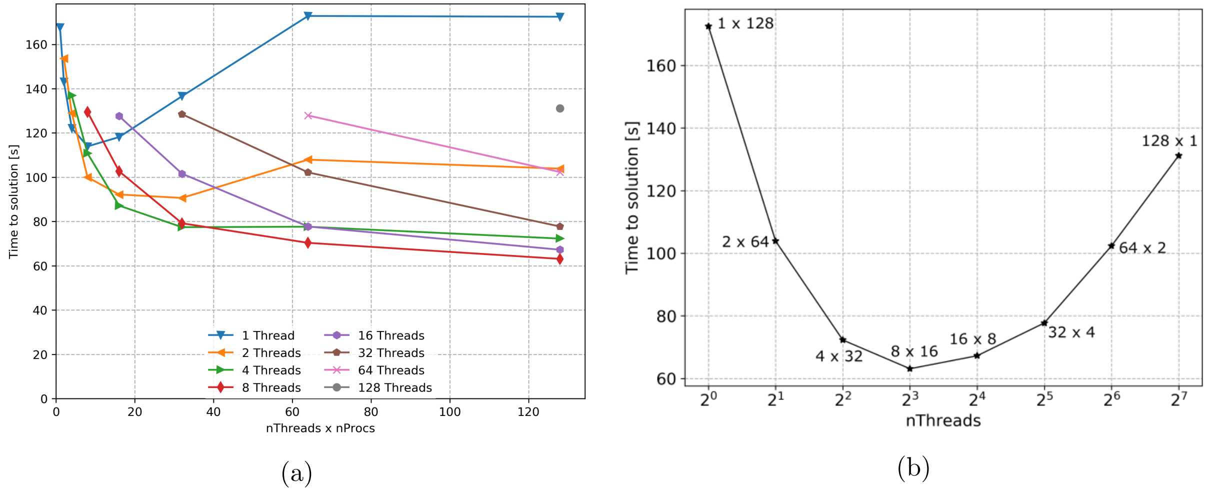

Figure 4: Node-level performance on HAWK: (a) Fix number of threads, represented by each curve, the number of processes are doubled for each subsequent point. (b) Full node configuration using nThreads $\times$ nProcs (last data point of each curve in Figure (a)) for each data point.

16×8 (threads $\times$ processes) provide the fastest computation for this setup, comparable with the CARA and CARO system.

Results - Comparing results of the different computing system

A comparison of the three systems reveals that the optimal configuration for this test case can be found in the range of combinations with 4×32, 8×16 and 16×8 (threads $\times$ processes). In the case of CARA and HAWK, the fastest computation is achieved with 8 threads and 16 processes. While on the CARO system, the combination of 4 threads and 32 processes allows

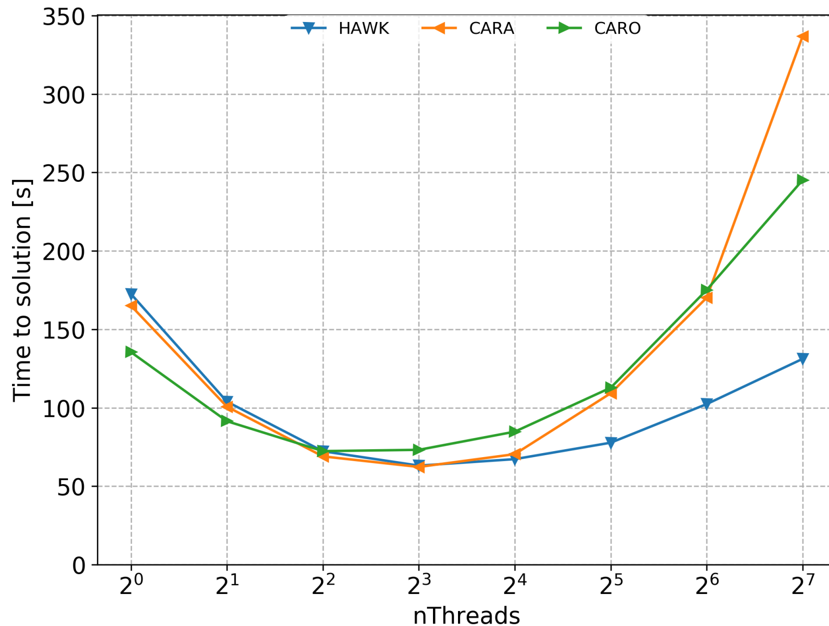

Figure 5: Comparison of the time to solution for all three systems (CARA, CARO and HAWK) on a full node.

for a slightly faster time to solution. Nevertheless, all three systems exhibit a strikingly similar behaviour, as evidenced by the observation that the aforementioned three combinations, namely 4×32, 8×16 and 16×8 (threads $\times$ processes), consistently emerge as the best performing ones. Figure 5 illustrates the full node configuration on the various systems, with the orange and blue curves representing the CARA and HAWK systems, respectively. It can be observed that the two systems exhibit a high degree of similarity, particularly in their optimal configuration of 8 threads and 16 processes, where their respective curves overlap.

4 Conclusion

We examined the node-level performance of the general-purpose structural solver `b2000++pro`. The investigation included three supercomputing systems with similar hardware configurations. Different parallelization configurations were examined to identify the optimal run time parameters for the examined test case. This analysis revealed that the optimal configuration for the CARA and HAWK systems is a combination of 8 threads on 16 processes, while the optimal configuration for the CARO system is found with 4 threads and 32 processes. It is notable that there is such a difference between CARA and CARO despite the same processors in these systems. Further it is notable that the usage of 8 threads can yield an advantage, though just 4 cores share a single L3 cache in the processors. As described, this can be attributed to the different regions in the parallelization concept of `b2000++pro`, with only one part benefiting from MPI parallelism. It would, therefore, also be interesting to consider different numbers of threads to be used in the two regions of `b2000++pro`, as the intel-TBB parallel part could potentially benefit more from more cores, while the OpenMP parallelism in MUMPS seems to be sensitive to the use of more than 4 threads. However, what this investigation has shown is

the relatively strong limitation of the shared memory parallelism in the current implementation of b2000++pro, and thus the need to change the parallelization concept and allow for the exploitation of more distributed memory parallelism.

Acknowledgement

The authors gratefully acknowledge the scientific support and HPC resources provided by the German Aerospace Center (DLR). The HPC system CARA is partially funded by "Saxon State Ministry for Economic Affairs, Labour and Transport" and "Federal Ministry for Economic Affairs and Climate Action". The HPC system CARO is partially funded by "Ministry of Science and Culture of Lower Saxony" and "Federal Ministry for Economic Affairs and Climate Action". The authors also gratefully acknowledge the Gauss Centre for Supercomputing e.V. for providing computing time on the GCS Supercomputer HAWK at Höchstleistungsrechenzentrum Stuttgart.

REFERENCES

[1] SMR Documentation. https://www.smr.ch/newdoc/, [Online; accessed January 12, 2024].

[2] M. Petsch, D. Kohlgrüber, C. Leon Munoz and T. Rothermel: Integration of the structural solver b2000++ in a multi-disciplinary process chain for aircraft design. In: DLRK 2020 (September 2020), https://elib.dlr.de/139438/.

[3] Getting Started with Intel®Threading Building Blocks (Intel®TBB). https://www.intel.com/content/www/us/en/developer/articles/guide/get-started-with-tbb.html, [Online; accessed January 12, 2024].

[4] L. Dagum and M. Ramesh: OpenMP: an industry standard API for shared-memory programming. Computational Science & Engineering, IEEE 5.1 (1998): 46-55.

[5] Message Passing Interface Forum: MPI: A Message-Passing Interface Standard Version 3.0. https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, [Online; accessed August 13, 2024].

[6] MUMPS : a parallel sparse direct solver. https://mumps-solver.org/index.php?page=home, [Online; accessed August 13, 2024].

[7] P.R. Amestoy, A. Buttari, J.-Y. L'Excellent and T. Mary: Performance and Scalability of the Block Low-Rank Multifrontal Factorization on Multicore Architectures. In: ACM Transactions on Mathematical Software, 2019.