ParCFD2024
35th International Conference on Parallel Computational Fluid Dynamics
Sep. 02-04, 2024, Bonn, Germany

Correctness and Performance Analysis of an Open-Source CFD Application

Fabian Orland[a,*], Joachim Jenke[a] and Radita Liem[a]

[a]RWTH Aachen University, Chair for High-Performance Computing, Seffenter Weg 23, 52074, Aachen, Germany

ARTICLE INFO[†]

Keywords:
Computational Fluid Dynamics;
Finite Volume Method;
OpenMP;
Data Race;
Performance Analysis;
Input/Output

ABSTRACT

This contribution highlights correctness and performance analysis results for an open-source CFD application. During the performance analysis of the code, we observed unexpected non-deterministic behavior in the application. Data race analysis with ThreadSanitizer, available in GNU and LLVM compilers, in combination with our extension for OpenMP-aware data race analysis, identified the root cause of the non-deterministic behavior. We will look into the analysis setup and the reports from the tool. Furthermore, we will discuss insights about the performance and I/O behavior of the code collected with the Scalasca tool-suite & Darshan and highlight performance optimizations.

1. Introduction

The open-source CFD application under investigation in this work is CalculiX[1]. CalculiX was mainly developed to solve problems in structural mechanics using the finite-element method. However, CalculiX also offers the solution to computational fluid dynamics problems using the finite-volume method described by Moukalled et al. [1]. The use case investigated in this work simulates the laminar airflow through a 90° bent pipe with a square cross-section. The code is written in C and Fortran and parallelized using pthreads.

1.1. Pthreads parallelization

To achieve worksharing via multithreading using pthreads, the pattern shown in Fig. 1a is repeatedly found in the CalculiX code. The code snippets show how the residual of solving the momentum equations is parallelized. The application's main thread creates `num_cpus` new worker threads using `pthread_create` (l.48). The function pointer passed to the creation call, i.e. `calcresvfluid1mt`, is a high-level wrapper function that takes care of the work distribution for each spawned thread (Fig. 1c, l.146) and calls into a Fortran subroutine that implements the actual work (l.148). In this case, it is `calcresvfluid1`, which calculates the residual using a sparse matrix-vector product (Fig. 1d, l.30). After all threads have been spawned, the main thread waits until all spawned worker threads have finished their work by calling `pthread_join` (Fig. 1a, l.51).

This code pattern leads to an excessive total number of threads created during execution because each time the application performs multithreading, *num_cpus* threads are created, but they are not reused. For example, executing the simulation for 5 timesteps with $num\_cpus = 8$ threads creates 3809 threads in total. This pattern prohibits tracing longer simulation runs, because performance analysis tools like Extrae, Score-P and Vampir cannot handle those huge number of threads. For example, Extrae was not able to handle more than 7800 threads.

1.2. OpenMP refactoring

The problematic pattern shown in Fig. 1a can be refactored into equivalent OpenMP code, as shown in Fig. 1b. This enables the application to benefit from the thread pool internally managed by the OpenMP runtime and avoids excessively creating new threads without any reuse. It also highlights the importance of application developers relying on best practices provided by open standards like OpenMP.

2. Correctness analysis

At the beginning of our analysis, the application showed unexpected non-deterministic behavior when executed by more than a single thread. In repeated executions using the same input file and the same number of threads, the internal, parallel GMRES solver required a different number of iterations until convergence. A data race analysis using ThreadSanitizer[2] (TSan) revealed the root cause of this non-determinism. As shown in Fig. 2, TSan detected a data race caused by two concurrent writes by thread T1 and thread T2 to the same memory location at address `0x7b1000000000` in

[†]This paper is part of the ParCFD 2024 Proceedings. A recording of the presentation is available on YouTube. The DOI of this document is 10.34734/FZJ-2025-02484 and of the Proceedings 10.34734/FZJ-2025-02175.

*Corresponding author

orland@itc.rwth-aachen.de (F. Orland); jenke@itc.rwth-aachen.de (J. Jenke); liem@itc.rwth-aachen.de (R. Liem)

ORCID(s): 0000-0002-8681-2661 (F. Orland); 0000-0003-0640-8966 (J. Jenke); 0000-0002-2506-1841 (R. Liem)

[1]https://www.calculix.de/

[2]https://github.com/google/sanitizers

```c
43  void calcresvfluidmain(ITG *num_cpus, /* ... */){
44      NNEW(ithread,ITG,*num_cpus); pthread_t tid[*num_cpus];
45      NNEW(res1,double,*num_cpus);
46      for(i=0; i<*num_cpus; i++)  {
47        ithread[i]=i;
48        pthread_create(&tid[i], NULL,
49          (void*)calcresvfluid1mt, (void*)&ithread[i]);
50      }
51      for(i=0; i<*num_cpus; i++)  pthread_join(tid[i], NULL);
52      /* ... */
53  }
```

(a) `calcresvfluidmain.c` (pthreads)

```c
43  void calcresvfluidmain(ITG *num_cpus, /* ... */){
44      NNEW(ithread,ITG,*num_cpus); pthread_t tid[*num_cpus];
45      NNEW(res1,double,*num_cpus);
46      #pragma omp parallel num_threads(*num_cpus)
47      {
48        int omp_get_thread_num(void);
49        ITG thread_num = omp_get_thread_num();
50        calcresvfluid1mt(&thread_num);
51      }
52      /* ... */
53  }
```

(b) `calcresvfluidmain.c` (OpenMP)

```c
143 /* subroutine for multithreading of calcresvfluid1 */
144 void *calcresvfluid1mt(ITG *i){
145     /* number of equations for this thread */
146     ITG n=nestart1[*i+1]-nestart1[*i];
147
148     FORTRAN(calcresvfluid1,(&n,a1,&b1[nestart1[*i]],
149         &au1[nestart1[*i]],ia1,&ja1[nestart1[*i]],
150         &x1[nestart1[*i]],res1));
151     return NULL;
152 }
```

(c) `calcresvfluidmain.c`

```
25  subroutine calcresvfluid1(a,b,x,res)
26  implicit none
27  real*8 a(*),b(*),x(*),res
28
29  res=0.d0
30  ! calculation of res = (A*x)-b
31
32  return
33  end
```

(d) `calcresvfluid1.f`

Figure 1: Multithreading implementation in CalculiX by example of residual calculation. Subcaptions and line numbers refer to CalculiX source code files.

line 29 of `calcresvfluid1.f` (Fig. 1d). This memory location corresponds to the variable `res`. TSan also shows that the main thread allocated this memory location in line 45 of `calcresvfluidmain.c` (Fig. 1a). The `NNEW` macro expands to a `u_calloc` call that allocates `res1` as an array of double-precision values of size `*num_cpus`. In our case `num_cpus` = 8, which matches the size of 64 bytes reported by TSan for the memory block. The developer's intention behind this allocation is that thread i should store its calculated residual at index i of the array `res1`. However, in lines 148-150 of the file `calcresvfluidmain.c`, a pointer to the first element of array `res1` is passed to the subroutine `calcresfluid1` instead of a pointer to the i-th element. We modified the call to the subroutine `calcresvfluid1` and passed `&res1[i]` instead of `res1`, and no data race was detected anymore. After this change, the internal GMRES solver's convergence also behaves deterministically.

```
==================
    WARNING: ThreadSanitizer: data race (pid=115530)
      Write of size 8 at 0x7b1000000000 by thread T2:
        #0 calcresvfluid1_ CalculiX/calcresvfluid1.f:29
        #1 calcresvfluid1mt CalculiX/calcresvfluidmain.c:148

      Previous write of size 8 at 0x7b1000000000 by thread T1:
        #0 calcresvfluid1_ CalculiX/calcresvfluid1.f:29
        #1 calcresvfluid1mt CalculiX/calcresvfluidmain.c:148

    Location is heap block of size 64 at 0x7b1000000000
        allocated by main thread:
        #0 calloc libsanitizer/tsan/tsan_interceptors.cc:964
        #1 u_calloc CalculiX/u_calloc.c:41
        #2 calcresvfluidmain CalculiX/calcresvfluidmain.c:45
        #3 compfluid CalculiX/compfluid.c:856
        #4 nonlingeo CalculiX/nonlingeo.c:1307
        #5 main CalculiX/CalculiX.c:1150

    SUMMARY: ThreadSanitizer: data race CalculiX/calcresvfluid1.f:29
    ==================
```

Figure 2: TSan's analysis output when executing the application with eight threads.

We were able to reproduce and detect the same data race with the refactored OpenMP version of the code (Fig. 1b) using our extensions for OpenMP-aware data race analysis implemented on top of TSan in the Archer tool [2] and distributed as part of LLVM. To enable Archer's analysis capabilities, the application code must be instrumented by adding the flag `-fsanitize=threads` to the compiler. LLVM's OpenMP runtime automatically loads Archer when appropriate. Fortran code can be compiled with `gfortran` if `clang` is finally used to link the code with its OpenMP runtime.

3. Performance analysis

We conducted a performance analysis with the data-race free application and used the Scalasca tool suite, including Score-P, Cube, and Vampir, to collect and investigate

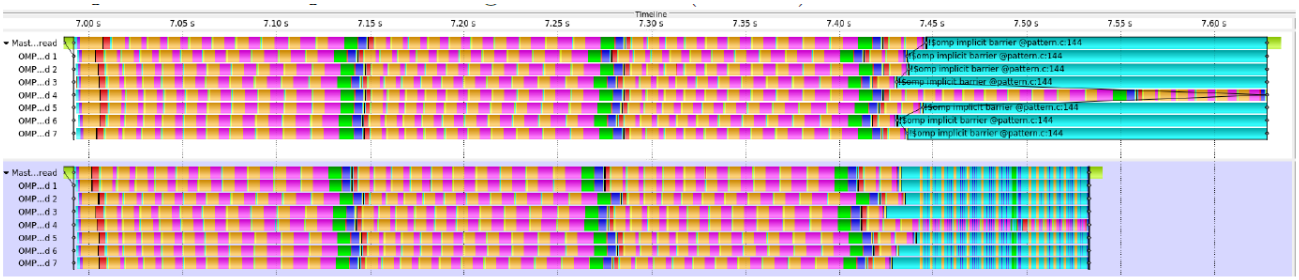

Figure 3: Trace comparison of reference version with nested tasks.

profiles and traces. The profile revealed a function called `dgmres1mt` as the hotspot, accounting for roughly 70% of the application's runtime, depending on the number of threads. This function is the main driver of the internal, parallel GMRES solver, which is used to solve large, sparse, non-linear systems arising from the discretization of the momentum equations. GMRES is parallelized by splitting the whole system matrix into independent subsystems so that each thread can solve one of these subsystems.

To further assess the parallel efficiency of the code, we calculated fundamental model factors proposed in [3] using the Cube Advisor Plugin, which indicated a loss of parallel efficiency caused by a load imbalance inside the GMRES solver. Further investigation of the trace with Vampir revealed a repeating pattern in which the same thread consistently requires more iterations to solve its subsystem compared to the others, as illustrated by the top trace in Fig. 3.

To improve the GMRES solver's load balance, we implemented a nested taskloop inside suitable parallelization subroutines, which are color-coded in the trace. This includes the matrix-vector product (orange), application of the preconditioner (yellow), a `daxpy` operation from LAPACK (blue), and computation of the residual (green) to check the convergence of GMRES. The orthogonalization (pink) of computed Krylov subspace vectors could not be parallelized. The nested taskloop is triggered to split the work between the straggling and idling threads if a threshold of more than 75% idling threads is detected using a global atomic counter variable. This optimization improved the load balance by 15% on average. It resulted in a runtime speedup of around 1.08x, which can also be visually observed using Vampir's trace comparison shown in Fig. 3.

Finally, we also investigated the I/O performance of the application using Darshan [4]. Darshan's analysis revealed that more than 20% of the runtime is spent in write operations. Moreover, the data is written in tiny chunks of 3-6 double-precision values. We enabled the usage of buffered I/O operations by adding the `buffered="yes"` keyword to all `open` calls in the code. This resulted in a speedup of 1.43x for the whole application.

References

[1] F. Moukalled, L. Mangani, M. Darwish, The Finite Volume Method in Computational Fluid Dynamics: An Advanced Introduction with OpenFOAM and Matlab, 1st Edition, Springer Publishing Company, Incorporated, 2015. doi:10.1007/978-3-319-16874-6_5.

[2] J. Protze, J. Hahnfeld, D. H. Ahn, M. Schulz, M. S. Müller, Openmp tools interface: Synchronization information for data race detection, in: IWOMP, Vol. 10468 of Lecture Notes in Computer Science, Springer, 2017, pp. 249–265. doi:10.1007/978-3-319-65578-9_17.

[3] C. Rosas, J. Giménez, J. Labarta, Scalability prediction for fundamental performance factors, Supercomput. Front. Innov. 1 (2) (2014). doi:10.14529/jsfi140201.

[4] S. Snyder, P. Carns, K. Harms, R. Ross, G. K. Lockwood, N. J. Wright, Modular hpc i/o characterization with darshan, in: 2016 5th Workshop on Extreme-Scale Programming Tools (ESPT), 2016, pp. 9–17. doi:10.1109/ESPT.2016.006.