# Status on KNOSOS analysis and performance improvements

Ricard Zarco Badia
Barcelona Supercomputing Center

23/11/2022

## 1 Introduction

Since we got the KNOSOS code, we have been doing some profiling and tracing of the application using inputs provided by the developers. One of the main issues detected was the existence of load imbalance between MPI ranks, which was discussed with one of the developers (José Luís Velasco). Such load imbalance seemed to be the product of the execution of different amounts of iterations in the `CALC_LOW_COLLISIONALITY` subroutine between MPI ranks, which could oscillate between 1 and 2800 iterations.

After this discovery, we decided to focus our efforts on analyzing and improving a sequential execution of the KNOSOS code. Since the MPI implementation of this code relies on launching "independent" calculations, improving a sequential version would translate to an overall improvement in the MPI version. We will briefly detail our advancements in the following sections. Please take into account that this is a brief report and some details and concrete results will be omitted.

## 2 Sequential profiling

In order to identify the most consuming parts of the code at this stage, we had to adapt the input files received (which consisted of 22 "independent" calculations) to a new input that only contained a single calculation (concretely, the first one). As soon as we got a working version with a single MPI rank and input, we tried to get some basic profiling using Valgrind.

After some compatibility issues with Valgrind on MN4 that we couldn't manage to solve, we decided to port everything to Nord3. Nord3 had a working Valgrind and an architecture similar to MN4, with the difference of having 16 cores per node instead of 48. We were able to obtain a callgrind output for a sequential execution there, which showed the most critical sections of the execution. Using Kcachegrind, we got a call graph of the most intensive functions of the code.

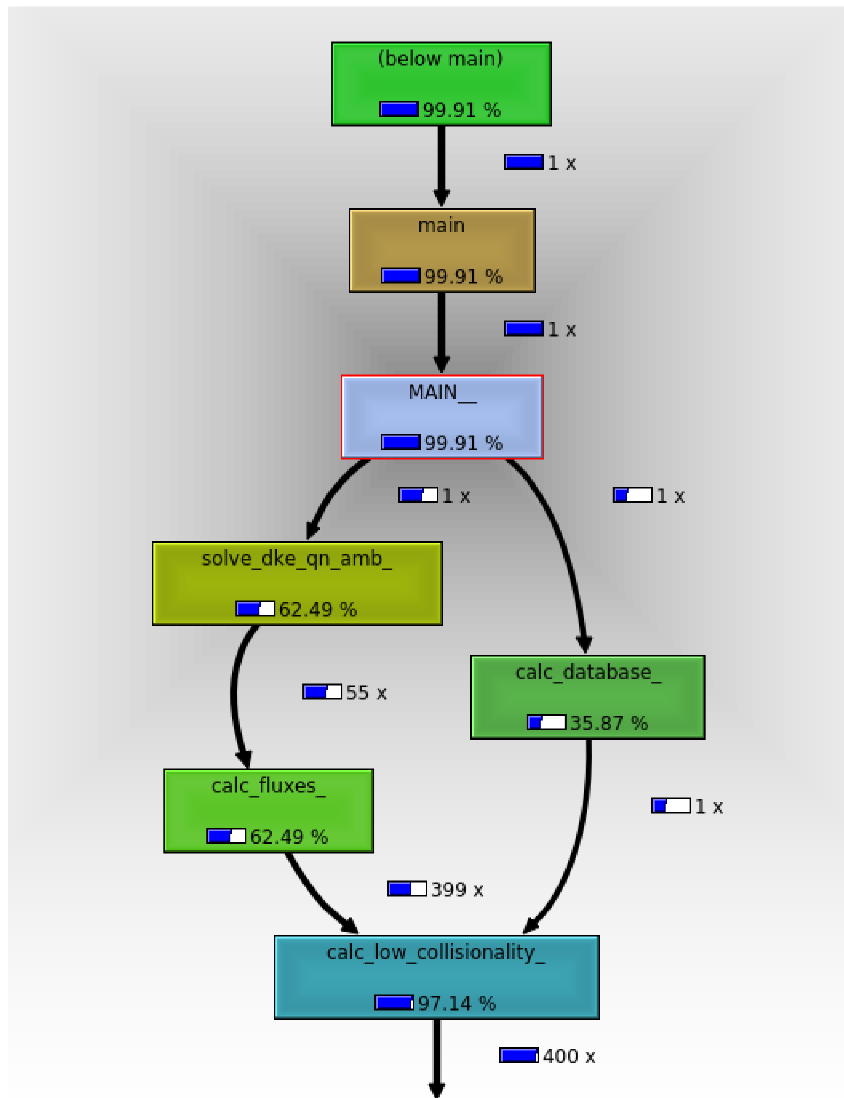


Figure 1: First half of the graph of most intensive calls.

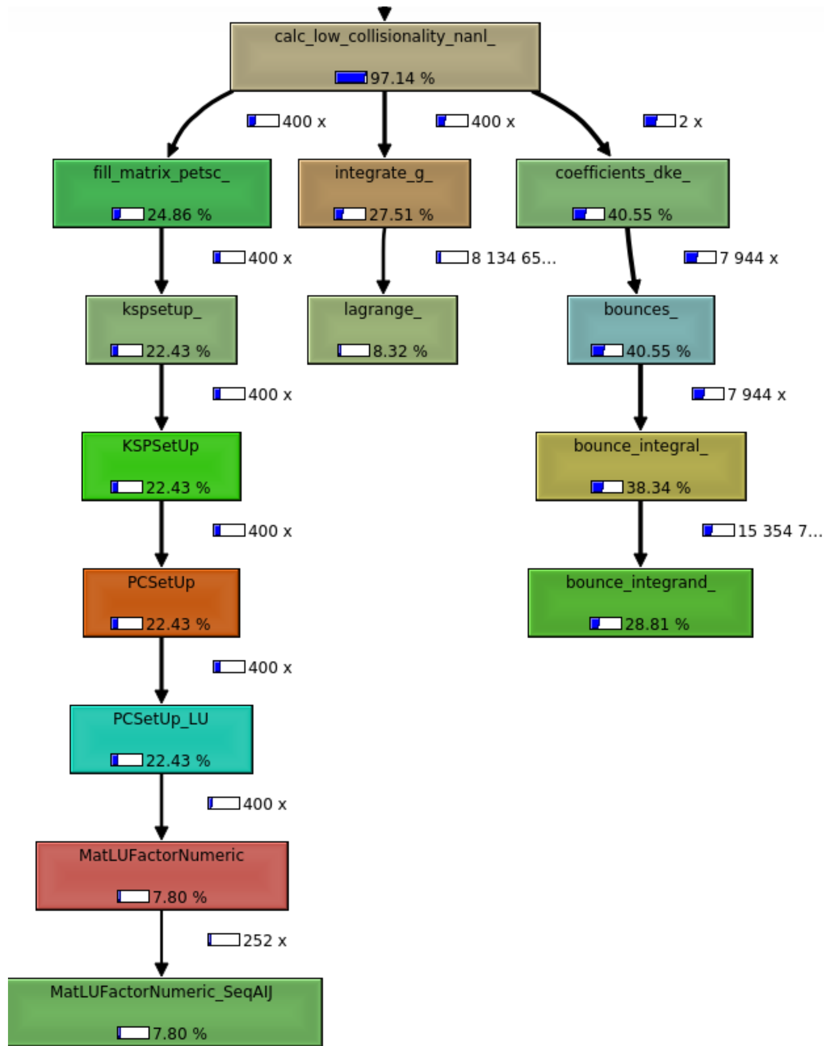


Figure 2: Second half of the graph of most intensive calls.

The percentage shown in the graph nodes determines the amount of time spent inside that function compared to the total execution time. The values alongside the edges of the graph represent the number of calls to that node (function). On figure 1 we can see a representation of the top-most functions of the execution, which indicates that `CALC_LOW_COLLISIONALITY` takes up almost the totality of the execution time for this case in particular. We can't say for sure if this is true for all inputs, it might change if the amount of times `CALC_LOW_COLLISIONALITY` is needed decreases. This should be tested with other inputs.

On figure 2 we see the functions that stem from `CALC_LOW_COLLISIONALITY`. We see that the main contributors to the execution time are the following subroutines:

- `FILL_MATRIX_PETSC (40.55%)`
- `INTEGRATE_G (27.51%)`
- `COEFFICIENTS_DKE (24.86%)`

By pure application of Amdahl's Law, we decided to focus on those functions (and their direct callers/callees). Although we could study how to optimize a strictly sequential version of this code, we decided to try to parallelize the most intensive parts using OpenMP first.

# 3 Parallellization using OpenMP

Our first focus was the relationship between the subroutines `COEFFICIENTS_DKE` and `BOUNCES`, since `BOUNCES` was inheriting practically all of the execution time of its caller and it takes up about 40% of the whole execution. We identified that different instances of the `BOUNCES` subroutine could be performed in parallel without producing data conflicts between OpenMP threads, so we applied the following naive implementation of OpenMP parallelization in `COEFFICIENTS_DKE` inside the file `low_collisionality.f90`:

```
!$OMP PARALLEL DO FIRSTPRIVATE(Q)
  DO ipoint=1,npoint
     iw =i_w(ipoint)
     ila=i_l(ipoint)
     CALL BOUNCES(iw,z1(iw),t1(iw),B1(iw),hBpp1(iw),vd1(:,iw), &
          &            zb(iw),tb(iw),Bb(iw),hBppb(iw),vdb(:,iw), &
          &            z2(iw),t2(iw),B2(iw),hBpp2(iw),vd2(:,iw), &
          & 1./lambda(ila),ipoint.EQ.1,nq,Q,&
          & zlw(ipoint),tlw(ipoint),zrw(ipoint),trw(ipoint))
     BI1(ipoint)=Q(1)
     BI2(ipoint)=Q(2)
     BI3(ipoint)=Q(3)
     BI4(ipoint)=Q(4)
     BI5(ipoint)=Q(5)
     BI6(ipoint)=Q(6)
     BI7(ipoint)=Q(7)
     IF(SOLVE_QN) BI8(ipoint,1:nmodes)=Q(8:nq)
     IF(DEBUG) WRITE(3100+myrank,'(3I6,8(1pe13.5),I5,4(1pe13.5))') &
          &  ipoint,ila,iw,BI1(ipoint),BI2(ipoint),BI3(ipoint),&
          &  BI4(ipoint),BI5(ipoint),BI6(ipoint),BI7(ipoint),BI8(ipoint,1),&
          & npoint,zlw(ipoint),tlw(ipoint),zrw(ipoint),trw(ipoint)
  END DO
!$OMP END PARALLEL DO
```

For reference, this change was able to reduce the execution time from 15 seconds (sequential) to 10 seconds (using 4 OpenMP threads or more). Seeing that we could achieve some benefits, we went to try to do the same with the `INTEGRATE_G` and `LAGRANGE` subroutines. We also identified a similar situation where we could run multiple instances of `INTEGRATE_G` in parallel. Here are the modifications we applied to `INTEGRATE_G` in `low_collisionality.f90`:

```
!$OMP PARALLEL DO FIRSTPRIVATE (thetape, D11_ale, dn1_ale)
DO ia=1,nalphab
   D11_zt(ia,il)=0
   CALL LAGRANGE(thetape(1:tnalpha,il),D11_ale(1:tnalpha),tnalpha,&
        & theta(ia),D11_zt(ia,il),0)
   IF(QN) THEN!.OR.NTV)THE
      dn1_zt(ia,il)=0
      CALL LAGRANGE(thetape(1:tnalpha,il),dn1_ale(1:tnalpha),tnalpha,&
         & theta(ia),dn1_zt(ia,il),0)
   END IF
   IF(DEBUG.AND.ipoint.EQ.1) WRITE(3800+myrank,'(3(1pe13.5),2I6)') &
        & zetap(il),theta(ia),D11_zt(ia,il),il,ia
END DO
!$OMP END PARALLEL DO
```

However, this time we didn't see much improvement using this approach. Since `LAGRANGE` doesn't inherit all the burden of `INTEGRATE_G` and its contribution to the total execution time is just about 8%, the OpenMP overhead makes this improvement negligible. For this specific case, we should check how to directly improve the sequential performance of the code inside `INTEGRATE_G`.

Right now we are working on it.

# 4 MPI load imbalance

For our original input, we detected that the total execution time of an MPI run is largely affected by load imbalance: some MPI ranks take a lot longer to finish their executions than others. Thanks to some insight provided by José Luís Velasco, we determined that the main culprit for the extreme cases is the difference in the amount of computed iterations of `CALC_LOW_COLLISIONALITY`.

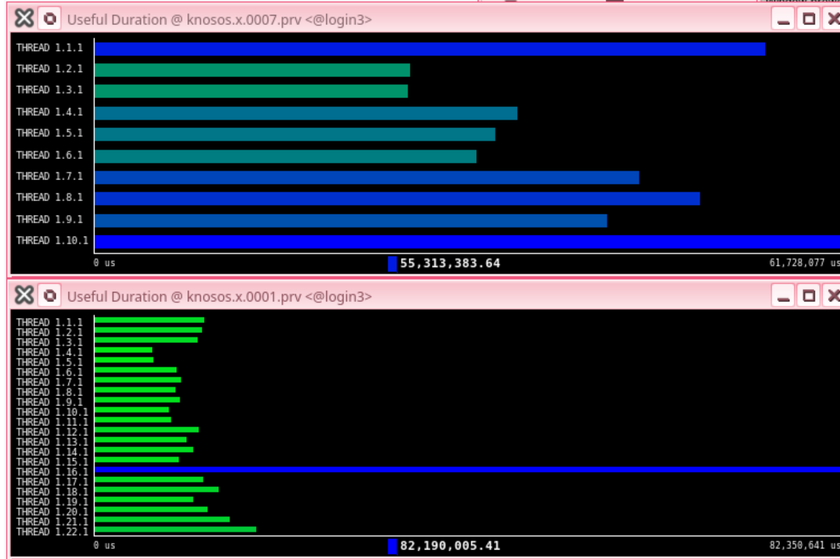


Figure 3: Paraver traces of KNOSOS where load imbalance becomes aparent

On 3 we can see an example of that case. In the first trace, we can already see load imbalance. The second trace shows an extreme case where the unconstrained execution time of a single rank negatively affects the whole execution. José Luís provided some profiling information about the number of iterations and time spent on `CALC_LOW_COLLISIONALITY`:

```
Time in CALC_LOW_COLLISIONALITY  Total: 10.982  Average: 0.019  Iterations:  402
Time in CALC_LOW_COLLISIONALITY  Total: 10.924  Average: 0.018  Iterations:  418
Time in CALC_LOW_COLLISIONALITY  Total: 11.272  Average: 0.019  Iterations:  413
Time in CALC_LOW_COLLISIONALITY  Total: 11.366  Average: 0.021  Iterations:  366
Time in CALC_LOW_COLLISIONALITY  Total: 11.135  Average: 0.024  Iterations:  309
Time in CALC_LOW_COLLISIONALITY  Total: 16.662  Average: 0.051  Iterations:  218
Time in CALC_LOW_COLLISIONALITY  Total:  5.979  Average:   NaN  Iterations:    1
Time in CALC_LOW_COLLISIONALITY  Total:  5.562  Average:   NaN  Iterations:    1
Time in CALC_LOW_COLLISIONALITY  Total:  5.839  Average:   NaN  Iterations:    1
Time in CALC_LOW_COLLISIONALITY  Total: 11.062  Average: 0.108  Iterations:   58
Time in CALC_LOW_COLLISIONALITY  Total:  5.111  Average:   NaN  Iterations:    1
Time in CALC_LOW_COLLISIONALITY  Total:  7.498  Average:   NaN  Iterations:    1
Time in CALC_LOW_COLLISIONALITY  Total:  6.430  Average:   NaN  Iterations:    1
Time in CALC_LOW_COLLISIONALITY  Total:  7.025  Average:   NaN  Iterations:    1
Time in CALC_LOW_COLLISIONALITY  Total:  5.809  Average:   NaN  Iterations:    1
Time in CALC_LOW_COLLISIONALITY  Total: 79.407  Average: 0.026  Iterations: 2801
Time in CALC_LOW_COLLISIONALITY  Total:  7.648  Average:   NaN  Iterations:    1
Time in CALC_LOW_COLLISIONALITY  Total:  9.160  Average:   NaN  Iterations:    1
Time in CALC_LOW_COLLISIONALITY  Total:  7.033  Average:   NaN  Iterations:    1
Time in CALC_LOW_COLLISIONALITY  Total:  8.334  Average:   NaN  Iterations:    1
Time in CALC_LOW_COLLISIONALITY  Total: 10.281  Average:   NaN  Iterations:    1
Time in CALC_LOW_COLLISIONALITY  Total: 10.096  Average:   NaN  Iterations:    1
```

Here we can see that, in general, the computations that require a higher amount of `CALC_LOW_COLLISIONALITY` iterations need more time than the ones with substantially lower iteration requirements. If that number of

required iterations scales too much on a single rank compared to its peers, the whole MPI execution needs to wait for just an outlier. We still haven't studied what causes those outliers, but our initial recommendation would be to implement a maximum iteration limit to avoid corner cases where one rank executes too many iterations compared with the rest of MPI ranks. We don't know if such a limitation would affect the overall quality of the computation for that rank, we would need some input from the developers.

Even if we managed to eliminate those potential outliers, we would still see variability in the execution times of each rank. For example, in the profiling information we can also see that some ranks that only do one iteration still need more time than others that do more than 200. It would be helpful to know if the expected computation time/complexity could be somewhat predicted based on the inputs used. If this is possible, a potential option would be to just launch MPI ranks of similar expected computation times.

# 5 Points of discussion for a future meeting with the developers

As a final summary of what has been described in this report, we would like to make a list of topics to discuss in a future meeting. The main points would be the following:

- **OpenMP pros and cons.** We have already seen some benefits from OpenMP when trying to reduce the execution time of a single surface. However, those benefits are not for free. Using the same amount of resources, we should decide what is more desirable: using more threads for each surface (decreasing the total compute time but also decreasing the number of surfaces being computed) or just sticking with an MPI version where each rank computes a surface using a single thread (maximizing the number of surfaces being computed, but slower).

- **Current MPI approach.** We understand that this MPI implementation is based on the idea of running multiple similar computations in a Monte Carlo approach. It would be interesting to discuss if this is the approach that you want/need in the future. It would help in order to see what options are worth exploring or not.

- **Load imbalance and viability of potential solutions.** Load imbalance is one of the main issues detected in the current MPI approach. In some cases, it is produced by an abnormally high amount of executed iterations of some functions on some MPI ranks (as described in section 4). We have proposed to implement some kind of iteration limit for those cases, but we don't know if it would make sense for your specific needs. Another proposed option would be to determine the expected time or complexity of the computation of each surface before actually starting computing on it. We would like to know if that would be possible.

- **Doubts about code.** While reading sections of the code, we have seen that there are some subroutines that have variants with the suffix "NEW". For example, `CALC_LOW_COLLISIONALITY_NANL` and `CALC_LOW_COLLISIONALITY_NANL_NEW`. The "NEW" versions don't show up on our valgrind traces, so they don't seem to be used on our executions. Are these subroutines improved versions of their normal counterparts? Should we use them? If that is the case, how?

This is just a brief list of topics proposed from our side, any other topics are welcome. If you want to propose other discussion points, please let us know.