Barcelona Supercomputing Center
Centro Nacional de Supercomputación

OPTIMIZATION OF AN OCEAN MODEL USING PERFORMANCE TOOLS

BSC-CES-2015-002

NEMO, Computational Performance, Analysis, Optimization

Oriol Tintó Prims, Miguel Castrillo, Kim Serradell, Oriol Mula-Valls, Francisco J. Doblas-Reyes

Earth Sciences Department

Barcelona Supercomputing Center - Centro Nacional de Supercomputación (BSC-CNS)

08 September 2015

TECHNICAL REPORT

Series: Earth Sciences (ES) Technical Report

A full list of ES Publications can be found on our website under:

http://www.bsc.es/projects/earthscience/ES-CFU/doku.php?id=start

Summary

The Barcelona Supercomputing Center-Earth Sciences department uses great amounts of computational resources in simulations involving the NEMO ocean model, in stand-alone mode or as part of a climate model. This technical report explains the work done in analysing and optimizing the NEMO computational performance to make a better usage of computational resources and collaborate with the Nemo System Team to help improving the model by improving the efficiency and the throughput. By optimizing the model bottlenecks, in a low resolution grid we achieved simulations twice faster than before, going **from 11 to 23** simulated years per hour of maximum model throughput being able to use more resources, with an overall increase on the efficiency, **saving up to 40% of resources** in a 128-core run. To do so, we used the state-of-the-art performance tools developed by the BSC Computer Sciences department.

Contents

1.Introduction

In this technical memorandum, we will present **a computational performance analysis and an optimization of bottlenecks of the NEMO ocean model,** to find why the model fails to scale and to improve both model throughput[1] and efficiency.

The **NEMO** [1] (Nucleus for European Modelling of the Ocean) model is a state-of-the-art ocean global circulation model used for oceanographic research, climate studies, seasonal forecasting and also for operational oceanography, where the time-to-solution[2] is crucial due the time limitation to provide forecasts. It has been used in 5 of the 28 Earth system models involved in the CMIP5 project. The total amount of computing resources used in NEMO simulations worldwide can easily exceed a billion of computing hours per year. Moreover, NEMO has limited throughput and its scalability[3] is a well-known issue, which ultimately affects the performance of Earth system models that use it [3]. For these reasons, model developments targeting the computational performance are mandatory and this work does a step in that direction.

The model includes many different modules and options that have an impact on the computational behaviour of the application. Computationally speaking, one of the most determining parameters of the model is the resolution of the grid, as the computational workload and the length of the time-step depend on that. The use of higher resolutions implies more computational workload and also requires shorter time-step for dynamical stability reasons. This means that each time-step needs more wall-clock time to be computed and also more time-steps are needed to simulate a specific period of time. On the other hand, more workload increases the scalability of the model, as a bigger proportion of time is spent in computation against the time spent in communication. In this work, we focused on one of the NEMO reference configurations, that consists in the ocean (OPA) and sea-ice (LIM [2] version 3) modules on a low resolution grid (details can be seen in section 2.2). The advantage of performing an analysis using a low resolution grid is that problems related with small sub-domains that limit the scalability of the model are already visible using a small number of CPUs. Therefore, the issues that will be relevant at higher resolutions using hundreds of thousands of cores can be identified and faced with a lower cost. On the other hand, problems related with higher resolutions such the memory usage or the output size are less determinant and therefore may not be extrapolated.

In this technical report, we illustrate the methodology that we use to identify bottlenecks, develop optimizations and evaluate their impact on the performance with an operational ocean model. To summarize the impact of this work, with the used configuration, the optimized version of NEMO can simulate more than two times faster than before, going **from 11 to 23** simulated years per hour as a maximum model throughput, and improving the efficiency, **saving up to 40% of time in a 128-core run**. This methodology can be extended to any Earth system model running on a high-performance computing (HPC) platform.

[1] Throughput: The amount of simulation years that the model can produce in a specific time.
[2] Time-to-solution: The time required to finish the simulation.
[3] Scalability: The capacity to use more computational resources efficiently as they are increased.

All the simulations were performed in the Marenostrum 3, the HPC hosted by the Barcelona Supercomputing Center (BSC).

2. Experimental setup

2.1. Model Description

NEMO is a state-of-the-art modelling framework for oceanographic research, operational oceanography seasonal forecast and climate studies. It includes:

- five major components:
    - the blue ocean (ocean dynamics, NEMO-OPA);
    - the white ocean (sea-ice, NEMO-LIM);
    - the green ocean (biogeochemistry, NEMO-TOP) ;
    - the adaptive mesh refinement software (AGRIF) ;
    - the assimilation component NEMO_TAM;
- some reference configurations allowing to set-up and validate the applications ;
- A set of scripts and tools (including pre- and post-processing) to use the model.

The model has been written in Fortran90 and uses MPI for the parallelization, so it can run in parallel in HPC clusters. Since version 3.6, it can use an input/output library called XML Input Output Server (XIOS) that allow the use of I/O dedicated servers and minimized the I/O problems that existed at earlier versions of the model.

The evolution and reliability of NEMO are organized and controlled by a European Consortium created in 2008.

2.2. Model Configuration

For our analysis, we used one of the NEMO reference configurations included in the stable release of the model. We have chosen a low-resolution configuration for the analysis. This allows us to study the scalability problems of the model related with small sub-domains by using a relatively low number of CPUs. Therefore, the problems that would arise in higher resolutions with thousands of cores are already visible with a lower cost.

Configuration name	ORCA2/LIM3
Grid	ORCA
Resolution	2 degrees (low resolution
Modules	OPA and LIM3
Time Step	5760 seconds (450 time-steps for one month of simulation)
Compilation keys	key_trabbl key_lim3 key_vvl key_dynspg_ts key_diaeiv key_ldfslp key_traldf_c2d key_traldf_eiv key_dynldf_c3d key_zdftke key_zdfddm key_zdftmx key_iomput key_mpp_mpi key_asminc key_diaobs
Compiler	Intel v13
MPI library	Intel MPI v4.1.3
Compiler Flags	-i4 -r8 -O3 -fp-model precise -xHost

Table 1: Description of the configuration used

2.3. Experiment design

Our analysis consists in two parts that have been done in the original code and with the optimized code:

i) First set of simulations of 1200 time-steps (corresponding to 80 simulated days) using from 1 to 8 nodes (16-cores per node), where we only measure the overall speed of simulation by getting the timestamp of each time-step. This is done by a simple implementation inside the code with no simulation overhead. For this set of simulations we did 10 executions for each case.

ii) Second set of simulations of 40 time-steps (corresponding to 2 days and 16 hours) using 1, 2, 4 and 8 nodes with Extrae instrumentation to collect detailed execution information into traces for a post-mortem analysis. In this second set we selected only a single iteration for each case, so the variability in the time-step time is not reflected.

2.4. Environment

All the simulations were performed in the Marenostrum 3, the HPC hosted by the Barcelona Supercomputing Center (BSC). Marenostrum 3 is a 1.1 PetaFLOPs supercomputer composed by 3056 nodes with 16 cores (2 x SandyBridge-EP 8-core 2.6 GHz processors) with 32 GB of memory per node.

The system uses a LSF queue system and we use a set of scripts in order to automatize the job submission, the trace collection and several other tasks necessaries to carry out the analysis.

3. Analysis

Basic information about model executions with different number of processor cores have been collected to obtain a first guess of the model performance in a strong scaling test. After this first analysis, we collected and analysed model traces using the BSC performance tools to investigate the reasons that explain the behaviour observed in the first tests.

3.1. Simulation speed and efficiency

In order to have a reference of the model throughput and scalability, we measured the time-step duration for different executions done with 1 to 8 computer nodes of 16 processor cores each. From the time-step time we derived the metrics *simulated years per hour* (*syph*) and *normalized efficiency*, where *syph* is the amount of years that the model can simulate per hour of wall-clock time, and the *normalized efficiency* is the relation between the number of years that the model can simulate for a given amount of total CPU-time [4]with a given number of cores and the quantity of years that the model can simulate using the same total CPU-time in the reference simulation (16-cores).

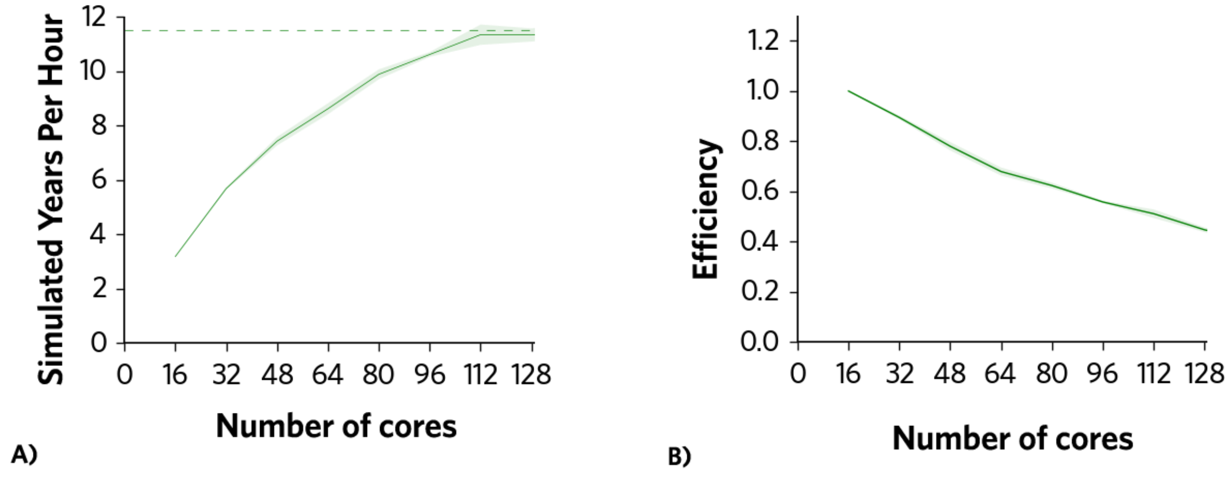

Figure 1 - Throughput and efficiency of simulations done with different number of cores.

We can see that the model reach a maximum throughput of 11.8 *syph* and there is no improvement in speed above 128-cores.

In figure 1B we can see the normalized efficiency. It means the quantity of years that we can

[4] Total CPU-time: the sum of CPU time consumed by all of the CPUs used by the computer program.

simulate with a given number of computational hours in relation to the quantity of years that we can simulate with the reference execution using a single node (16-cores). We can observe an efficiency drop and, when the model reaches the maximum throughput (128-cores), the efficiency is under 50%.

3.2. Detailed analysis

With the outcome of the previous analysis we designed an experiment to analyse in more detail the causes of the loss of efficiency and the bad scalability. As the model reaches a peak when using 128-cores, we decided to analyse different simulations doubling the number of cores from 16-cores (the reference simulation), to 128-cores (the peak).

3.3. Trace collection

As it was explained in section 2.3, we used the Extrae [4] tool to collect information from the execution and store it in traces for a post-mortem analysis.

From each one of the traces we "cut" the smallest repetitive pattern which corresponds to 5 ocean time-steps and one sea-ice time-step. Using this approach we discarded both initialization and finalization of the execution, but we can afford that, given that these parts are negligible in simulations of the typical length (from one to several years).

3.4. Trace analysis

We used several tools to analyse the information contained in the traces. The most important one is Paraver [5], a tool to visualize the information contained in the traces. With this tool is possible to see with high detail what happened during the execution. One can also compute metrics based on hardware counters such as parallel efficiency[5], instruction per cycle (IPC),…

The Dimemas tool is used to simulate the performance of an application in a different machine. Based in a real trace previously collected, it allows simulating distinct scenarios in order to see the impact of different parameters in the overall performance. For this study we used this tool to analyse the model sensitivity to the network latency[6] and bandwidth.

To complete our analyses, we used the tools Clustering, Tracking and Folding to evaluate the computational phases.

3.4.1. Function profile

Looking at a function profile generated by Paraver, we can measure the time spent in each routine. Hence, comparing the function profiles of different configurations, we can observe how they behave and identify if the scalability problems are caused by specific routines or otherwise are a general issue.

[5] Parallel efficiency: In this context, average percentage of the total execution time (computation + communication), from all the CPUs, spent in computation in a parallel run.
[6] Network Latency: the amount of time it takes for a packet to cross the network from a device that created the packet to the destination device.

In the reference case (first row in the figure 2), we can see that the routine **dynspg takes 31% of the execution time.** Apart from this routine, the **profile is very flat** and any other routine reaches 5% of the time at much.

When we observe the function profile for the subsequent cases, we can see that the proportion of time spent by **dynspg** keeps increasing so in the 128-cores case it reaches a **53% of the time**. The main change is that now the profile of the other functions is not flat anymore since now **limhdf takes 15%** and **limrhg 9%** of the time. Both functions belong to the sea-ice module.

The **dynspg**, **limhdf** and **limrhg** routines are considered the main bottlenecks of the model for its bad scalability as their total amount of time increases from **38% of the time** in the 16-cores case to **77% of time** in the 128-cores case.

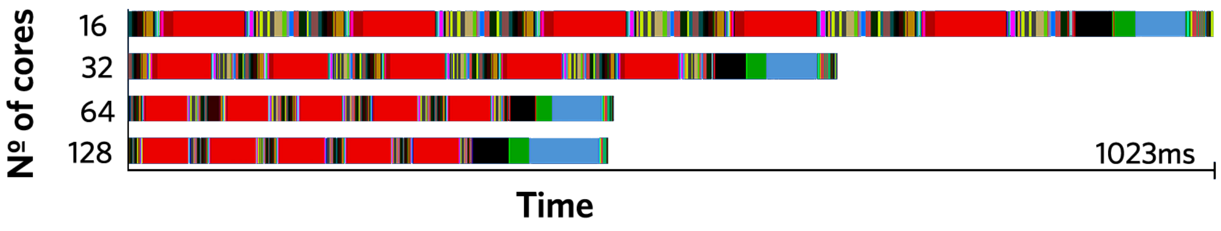

Figure 2- NEMO's function timeline comparison for 16, 32, 64 and 128 cores. Different colours represent different functions. The most time-consuming functions are dynspg in red, limhdf in blue and limrhg in black.

3.4.2. Sensitivity analysis

Using the **Dimemas** simulator we imitate the behaviour of the model in different scenarios, to learn the influence of the different network parameters to the model's performance.

Doing different simulations by varying the **network latency**, we observed that the latency has a huge impact on the performance.

The simulations show a linear influence of the latency, so the model's time-step time follows the subsequent relation:

$$t_{ts} = t_{lat0} + L_n \cdot lat$$

Where $t_{ts}$is the time-step time, $t_{lat0}$is the time-step time in an ideal case with latency 0, $L_n$is what we define as *latency coefficient* and lat is the latency.

This equation is valid for all of the cases, but the coefficient L_n increases with the number of processes.

It is also important to remark that while the value of T_{lat0} decreases when the number of processes increase, the proportion of time related with the latency is bigger and consequently the parallel efficiency is lower.

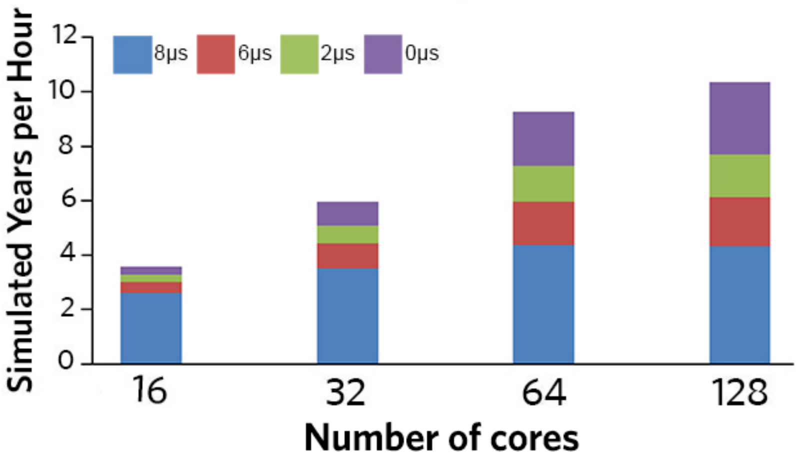

Figure 3- Latency impact on the different runs with 16, 32, 64 and 128 cores.

By plotting the *syph* one can easily observe the impact that the latency has. The more cores we use, the bigger this impact is, and also it determines how much the model can scale up. However, the **network bandwidth** sensitivity analysis showed that the influence of the network bandwidth in the model performance is very small and therefore the transmission time is negligible.

In conclusion, the network sensitive analysis showed a very high impact of the latency and a very low impact of the bandwidth. This is usually related with a **big number of small messages**.

3.4.3. Bottleneck analysis

As it can be seen in the section 2.2.1, the dramatic loss of efficiency is not a general problem for the entire model but instead is localized in few specific routines. Therefore, we focused our detailed analysis on the routines identified as bottlenecks, given that improving these routines will have a higher impact in the performance of the model.

In order to know if the bottleneck problems are related with communication or with computation, we measured the parallel efficiency of the model, since it gives information about the proportion of time spent in useful computation and communication. Analysing the bottlenecks, we can see that even in the 16-cores case the parallel efficiency is bad, especially for the limhdf, and it gets worse when we increase the number of cores.

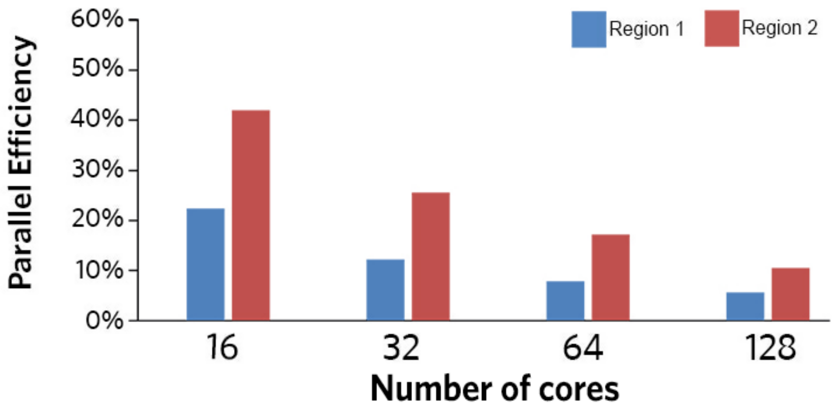

Figure 4: Parallel efficiency of the routines limhdf (blue) and dynspg (red) for executions done with 16, 32, 64 and 128 cores. Region 1 corresponds to the routine limhdf and Region 2 corresponds to the routine dynspg.

The figure above clearly demonstrates that the reason for the bad scalability of the model is the big proportion of the time spent in communication.

If we look in detail what happens inside **dynspg** (the surface pressure gradient routine), we can see that there is a loop consisting in three computation phases separated by three communication phases. The duration of that computation phases is really short (under 100 microseconds in the 16-cores case), and in the communication phases seven border interchanges between neighbours are performed, distributed in two consecutive interchanges in the first and the second communication phases and three interchanges in the third communication phase. In every interchange, each sub-domain interchanges messages with the border neighbours. For the average sub-domain, this interchange supposes sending 4 messages and receiving 4 messages. In addition to that, the sub-domains placed at the north fold have to perform extra communications, and some coastal sub-domains have less communications. On average, one interchange with the neighbour in the 16-cores case lasts 60 μs(4 sends + 4 receives + 4 waits + buffer movements), which means a total of 420 μs for each loop iteration. As a single iteration in the 16-cores case on average lasts 721μs, this means that the parallel efficiency in this routine is a 42%.

With the Clustering and Tracking tools we analysed the computational phases inside *dynspg*. We can observe that the number of instructions of these phases decrease when we increase the number of cores but not at the ideal rate, so the total number of instructions increases. This is caused by two main reasons. First, there is part of the code that cannot be parallelized and so it's replicated in all the processes. Second, the domain decomposition method being used implies an overlapping of the borders between neighbour sub-domains. In addition to that, we can also see that the IPC decreases with the core number increase. These two phenomena imply a loss of efficiency that reaches a 50% loss in the 128-cores case (Figure 3).

The second region targeted as bottleneck is **limhdf**, the sea-ice horizontal diffusion routine.

We can see that there are 41 consecutive calls to this routine in every sea-ice time-step, with high differences in the duration between the different calls. If we look with more detail, we can guess a loop structure, and the differences in time between different calls come from different number of loop iterations. Looking at the loop structure, iterations have two computation phases with very short duration, separated by two communication phases. The first communication phase is a border interchange of values with the neighbours and the second communication phase is a global communication that requires synchronization of all the sub-domains, not only the neighbourhood. In the 16-cores case, the computation phases represent only 22.4% of the iteration time (22/97$\mu s$of the iteration) while the computation regions take 61.8% (60/97μs) and 15.4% (15/97μs) time respectively. This analysis leaves clear that the problem in this routine is the fine granularity of the tasks and the communications.

Studying also the 128-cores case, it is revealed that the time does not improve at all but gets worse, rising to 128 μs (32% more time). If we look for a reason for this increase, we have that the time spent in the neighbour border interchanges remains almost the same (around 60 μs) while the computation time decreases with an acceptable efficiency (Figure 3). However, the time spent in collective communication increases a lot (from 15 to 60μs).

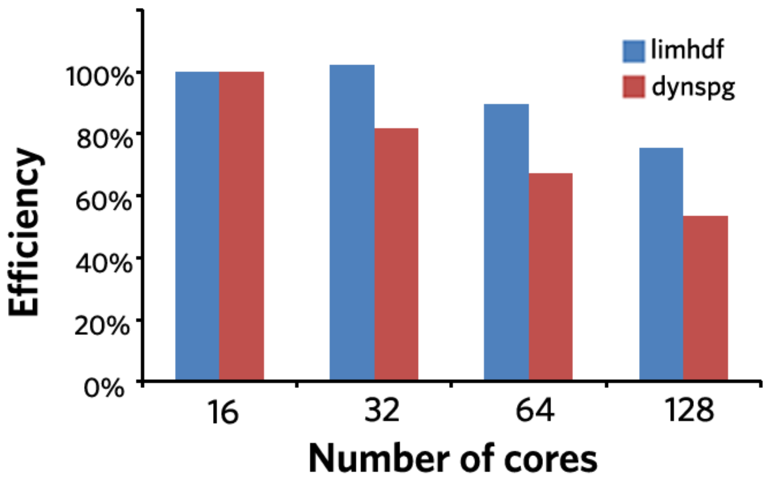

Figure 5: Efficiency of the computational regions in different cases compared with the 16-cores case

Finally, the problems in the **limrhg** routine are very similar to the problems found in **dynspg**.

Finally, from the analysis of the bottlenecks we can conclude that only a very small part of the efficiency loss comes from computation issues (mainly code replication due the overlapping), and the major part comes from the **communication issues**, mainly a lack of parallel efficiency due to low computational work charge with high frequency of communication and synchronization.

4. Optimizations

The diagnostic from the analysis was that the biggest problems constraining the model scalability were located mainly in two regions. Some of them were related with shared problems between them (short computational regions with high frequency of communication) and the others are local issues (collective communications, consecutive interchanges). This diagnostic suggests that the best way to increase the model's performance is **improving the communication**.

As it was shown before, almost all the time spent in communication is related with the **network latency**. To reduce this time, the most easy and effective way is to **reduce the number of messages** interchanged.

We sought a way to reduce the number of messages without changing the model algorithms and found several exploitable spots. Here we present the three optimizations: message packing, reduction of convergence check frequency and reordering of the sea-ice horizontal diffusion routine.

4.1. Message packing

In the routines **dynspg**, **limrhg** and **limadv** there are several **consecutive interchanges** with no calculation between them. In this case, our solution consists in packing the messages that have the same recipient into one single message.

As almost all the communication time is due to the latency, when we pack messages the time per message remains constant, so if we pack ***n*** messages into only one, we reduce the time spent in this communication by a factor of ***n***.

For example, the structure of the **dynspg** inner iteration is composed by three computation phases separated by three communication phases with 7 interchanges in total. By using the message packing optimization, we can pack the consecutive interchanges and therefore reduce the 7 interchanges per iteration to only 3.

This optimization can be implemented in every single part of the model's code where consecutive interchanges are performed. The final improvement is directly related with the number of messages that we can avoid using this.

4.2. Reduction of the convergence check frequency

In the **limhdf** routine there is a convergence check at each one of the inner loop iterations that can go from a few to hundred iterations. As the convergence check requires a collective communication, synchronization between all the model processes is required, with a high negative impact on the performance.

From the physical point of view, the fact that the limhdf routine performs a few more iterations does not degrade the quality of the simulation. Taking this into account, our solution was to reduce the frequency at which the convergence checks are performed in order to reduce the huge payoff of the collective communications. The convergence check frequency can be configured in the model namelist in order to give to the user the possibility to use it or not.

This very simple optimization is expected to have a big impact when using a high number of CPUs

4.3. Sea-ice horizontal diffusion routine reordering

As we could see during the analysis, at every sea-ice time-step there are 41 calls to limhdf, being each one of these calls independent from each other. In order to be able to use the message packing optimization inside this routine our solution consisted in reordering the code.

In the original case, the **limtrp** routine calls limhdf passing different fields one after another, and at each limhdf call it performs several iterations including computation and communication.

Our optimization consists in making the limtrp routine call limhdf for all the variables at the same time. With this approach we compute the inner loop calculations for all the variables before starting the communication. Therefore, in the communication phase now it is possible to interchange all the 41 variables in one single interchange. The convergence check can be also done for all the 41 variables in one single collective communication and in the posterior iterations, we only perform the calculations for the variables that still have not reached the convergence.

This optimization allows us to reduce dramatically the number of messages and achieve coarser granularity of the computation phases.

5. Impact of the optimization

We did similar analyses with the optimized version of the code to evaluate the impact of the optimizations.

5.1. Simulation speed and efficiency

Measuring the simulation speed and efficiency with the optimizations applied, we can see that our improvements have had a massive impact in the performance.

As the figure 4a) shows, the optimized version is faster even in the 16-cores case. While the original version reached a maximum of 11.8 *syph* using 128-cores, the optimized version can keep using more resources and reach 23 *syph*, which is more than double of the original speed. Looking at the efficiency we also have a nice improvement, having in the 128-cores case 60% more simulated years.

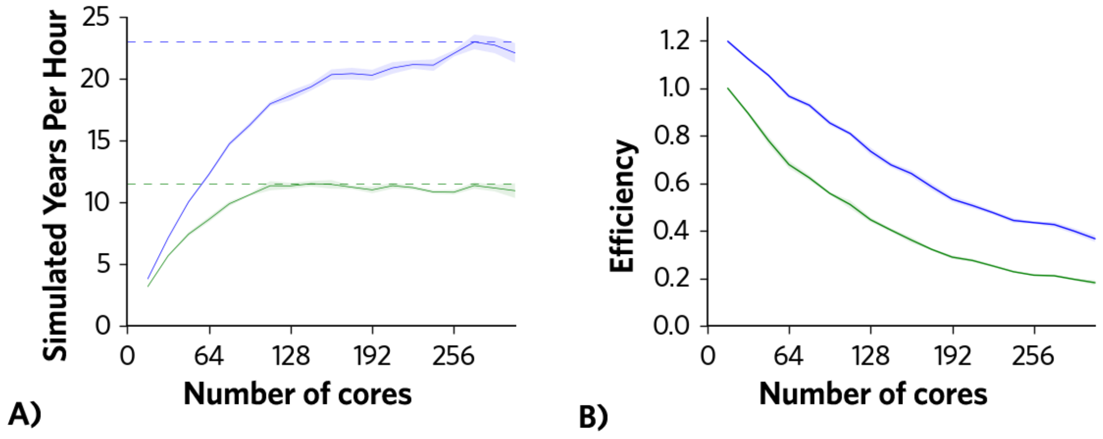

Figure 6: a) Model throughput comparison. b) Normalized efficiency comparison. In green the original model and in blue the optimized one. In green the no-optimized version of the model, in blue the optimized version of the model.

5.2. Trace analysis

5.2.1. Function profile

Comparing the optimized version function profile with the original one, we can see that the total time improvement comes from the time reduction in the bottleneck routines, as we expected.

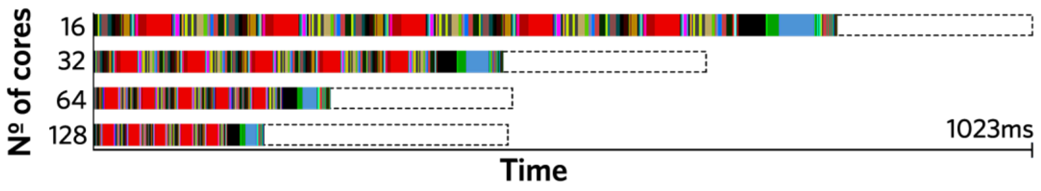

Figure 7 - Function timeline for the optimized version of 5 ocean time-steps and one sea-ice time-step. The dotted line represents the time of the no-optimized version of the model.

5.2.2. Sensitivity analysis

The sensitivity analysis for the optimized version shows a reduction of the variability of the time-step duration dependent on the latency of the network. Even with the 8 us scenario the model keeps scaling.

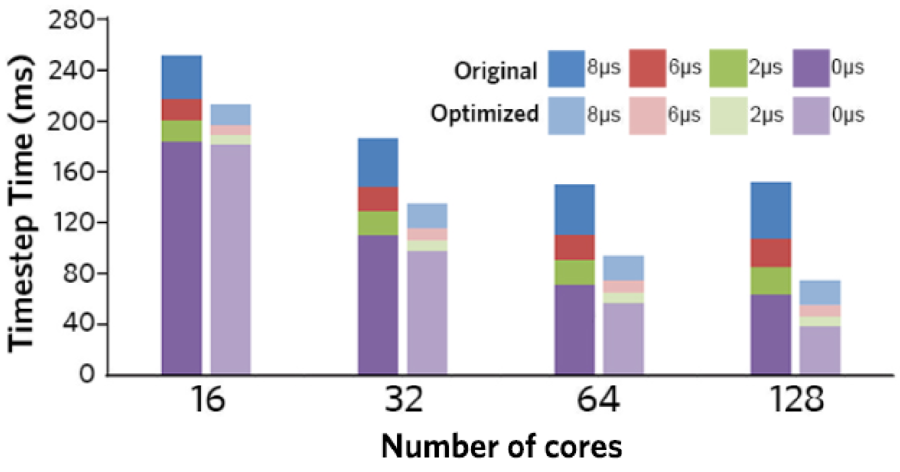

Figure 8- Comparison of the latency impact on the time-step duration of the original and the optimized version.

5.2.3. Bottleneck analysis

Looking again at the parallel efficiency of both regions, we can see that there is a big improvement, especially at the region 2.

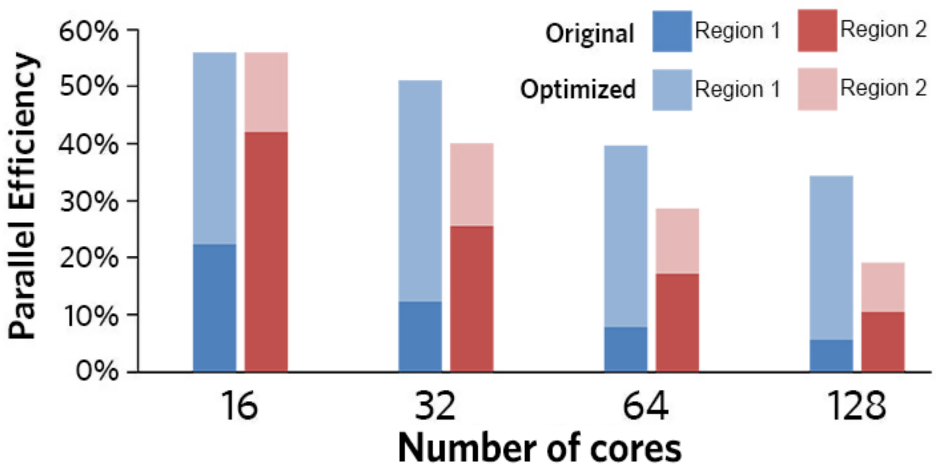

Figure 7: Parallel Efficiency comparison. The region 1 corresponds to the routine limhdf and the region 2 corresponds to the routine dynspg

While both regions experiment an important increment of the parallel efficiency, this is still low for

The efficiency of the computational phases shows the same behaviour than the original case.

6. Conclusions

The amount of computational resources dedicated to simulations involving the NEMO ocean model, as well as the need of faster simulations, constitute the main motivation for analysing the model and investing in its optimization. Our performance analysis on NEMO with the ORCA2/LIM3 configuration showed that the bottlenecks constraining the scalability of the model are related with a lack of parallel efficiency. This lack of parallel efficiency is mostly due to the bad suitability of some of the model algorithms for small sub-domains, where the relation computation/communication is really low. The computation time is reduced as the size of the sub-domains decreases while the communication time does not, and since this size is determined by the resolution of the grid and the number of cores used for the simulation, we have that lower resolutions have worse scalability than higher ones. To increase the parallel efficiency without changing the algorithms used to solve the equations neither increase the resolution of the grid, our solutions aim to increase the computation/communication ratio by reducing the communication overhead. This can be achieved by decreasing the number of messages. With the optimizations proposed in this document the number of messages was reduced without changing the algorithm, preserving, at the same time, the results. The resulting performance shows a great improvement, increasing the maximum simulation speed from 11 to 23 simulated years per hour for this configuration, and increasing considerably the efficiency, saving up to 40% of resources in a 128-core run. We expect these optimizations to save hundreds of millions of computing hours from now on. The optimizations have not been tested in higher resolutions, but are expected to have a positive impact when using an enough number of cores. Taking the present work as a base, besides testing the impact of these optimizations in higher resolutions, our further studies go in the line of analysing and optimizing the particular computational problems that arise when using bigger grids.

7. Acknowledgements

We would like to acknowledge the people from the Computer Science department of the BSC for their help, especially German Llort, Harald Servat and Jesús Labarta. We also want to acknowledge the people from the Nemo System Team for making our work easier and take into account our suggestions.

8. References

[1] Madec G. 2014: "NEMO ocean engine" (Draft edition r5171). Note du Pôle de modélisation, Institut Pierre-Simon Laplace (IPSL), France, No 27 ISSN No 1288-1619.

[2] Vancoppenolle M., Bouillon S., Fichefet T., Goosse H., Lecomte O., Morales Maqueda M.A., and Madec G., 2012 :" LIM The Louvain-la-Neuve sea Ice Model". Note du Pole de modélisation, Institut Pierre-Simon Laplace (IPSL), France, No 31 ISSN No 1288-1619.

[3] Asif, M., A. Cencerrado, O. Mula-Valls, D. Manubens, A. Cortés and F.J. Doblas-Reyes , 2014:" Case study in large scale climate simulations: Optimizing the speedup/efficiency balance in supercomputing environments."14th International Conference on Computational Science and Its Applications, doi:10.1109/ICCSA.2014.57

[4] Barcelona Supercomputing Center, 2015: "Extrae user guide"

[5] Labarta, J., Girona S., Pillet, V., Cortés, T., Gregoris, L., 1996: "DiP: A Parallel Program Development Environment". Euro-Par, Vol. II pages 665-674.