SUMMARY The process for developing a climate model often involves a wide community of developers. All of the code releases can be classified into two groups: (i) improvements and updates related to modeling aspects (new parameterizations, new and more detailed equations, remove of model approximations, and so on); (ii) improvement related to the computational aspects (performance enhancement, porting on new computing architectures, fixing of known bugs, and so on). The developing process involves both programmers, scientific experts, and rarely also computer scientists. The new improvements and developments are mainly focused on the scientific aspects and, in second stage, on the computing performance. The developments to improve the physic model often does not care about its impacts on the computational performances. This poses some issue in the developing process; after a new implementation, the code must be revised after new implementation to face out with the performance issues. In this work we analyze 5 different releases starting from the NEMO v3.2 (to be considered as our reference) and evaluate how new developments impact on the computational performances.
INTRODUCTION

The evolution and reliability of NEMO [1] are organized and controlled by a European Consortium created in 2008 between CNRS (France), Mercator-Ocean (France), NERC (UK), UKMO (UK) and, since 2011, CMCC (Italy) and INGV (Italy). The purpose of the Consortium is to set up appropriate arrangements for the successful and sustainable development of the NEMO System as a well-organized, state-of-the-art ocean model code system suitable for both research and operational work. In order to achieve this goal the Consortium Members have agreed on:

- the resources they will commit each year to the NEMO System Team which will maintain the code;
- the arrangements for managing and coordinating the work of the NEMO System Team and for setting its priorities;
- the arrangements for the Intellectual Property rights over the Software to make the code freely available under an appropriate version of the CeCILL Free Software License with the aim of attracting a critical mass of scientists to use the software and contribute developments to be incorporated into it.

The NEMO System Team is in charge of the maintaining of the reference NEMO code and its distribution through the NEMO System web server. The NEMO System Team shall carry out the Work Plan proposed by the Developers Committee and approved by the Steering Committee. This Work Plan will include:

- incorporation into NEMO System of new developments (scientific or technical);
- reorganization of the code to improve its readability, orthogonality or structure;
- optimization of NEMO System on the architectures available in the Consortium;
- maintenance of the scientific paper and on-line documentation;
- configuration control of the available versions of NEMO System;
- testing and release of new versions (typically once or twice a year);
- making NEMO System readily available to the scientific community and to the Consortium Members;
- providing assistance to new users;
- practical support for user meetings (held typically once a year);
- assistance in scientific development in an area of high priority.

The Work Plan, describing the work which each of the Consortium Members will contribute to the NEMO development, is updated annually according to the following schedule:

31 March: Report of work by the NEMO System Team delivered by the NEMO Project Manager to Steering Committee;
Not later than one month before the Steering Committee: draft of Work-Plan for following year prepared by NEMO Project Manager and approved by NEMO Scientific Leader, following consultation of Developers Committee and discussion with all Consortium Members;
Not later than the 30th of November: Work-Plan for following year agreed by Steering Committee;
Before the end of the year: All schedules to Agreement updated for following year.
The developing process involves both programmers, modeling experts, and rarely also computer scientists. The improvements and new developments are mainly focused on modeling and, in a second stage, on computing performance. This often poses some issue in the developing process where the code must be revised after new implementations to face out with the performance issues. In this work we take 5 different releases starting from the NEMO v3.2 (to be considered as our reference) and evaluate how new modeling developments impact on the computational performances. The NEMO releases are shortly described below:

**nemo_v3.2.2** as reference

**nemo_v3.3.0** new developments include: (i) merge between TRA and TRC; (ii) introduction of a modified leapfrog-Asselin filter time stepping scheme; (iii) additional scheme for iso-neutral mixing; (iv) addition of a Generic Length Scale vertical mixing scheme; (v) addition of the atmospheric pressure as an external forcing on both ocean and sea-ice dynamics; (vi) addition of a diurnal cycle on solar radiation; (vii) river runoffs added through a non-zero depth, and having its own temperature and salinity; (viii) CORE II normal year forcing set as the default forcing of ORCA2-LIM configuration; (ix) generalization of the use of fldread for all input fields; (x) optional application of an assimilation increment.

**nemo_v3.3.1** this version matches with 3.3.0 code with dynamic allocation added. The main implications are: (i) the number of processors are now part of the namelist nam_mpp; (ii) the definition of the configuration parameters in par*_h90 have to be slightly changed; (iii) the coding rules have been updated.

**nemo_v3.4a** this is a major release and relevant improvements have been added. From scientific point of view: (i) new pressure gradient suitable for s-coordinate; (ii) completion of Griffies iso-neutral diffusion; (iii) Pacanowski-Philander scheme for computation of Ekman depth added; (iv) a new bulk formulae added; (v) a drag coefficient compute by wave model introduced; (vi) tidal potential forcing added; (vii) Netpune effect parametrization; (viii) point to point MPI communication for north fold; (ix) sub time stepping for biogeochemistry models when using non-linear free surface allowed; (x) improvement in PISCES.

From coding point of view some simplification have been made: (i) simplification of dynamic allocation; (ii) completion of merging between TRA and TRC; (iii) more flexible definition of BDY input data; (iv) simplification of interfaces toward biogeochemical models; (v) interface with CICE in coupled mode; (vi) use of fldread to read/interpolate data for passive tracers and dynamical input data for OFFLINE configurations.

**nemo_v3.4b** this version corresponds to 3.4a code with some performance improvements on dynamical allocation using the stack: no explicit coding nor POINTER.

**DEFINITION OF TESTS**

Tests have been done using the GYRE configuration as equivalent to 1/4° (cfg=48) and 1/12° (cfg=144) resolutions. In our case the GYRE configuration makes no I/O (nor write nor restart), and runs 720 time steps. The CPU time sums CPU time in seconds for time steps between 2 and 719. In the namelist, nn_bench=1, nn_rdt = 180. Cpp keys: key_gyre key_dynspg flt key_ldfslp key_zdfike key_mpp_mpi.
## COMPUTING ENVIRONMENT

Tests have been performed on the IBM Power6 cluster at CMCC. The compiler name, flags and environment variables, chosen to improve the performance, are reported in the following.

Computer used:

<table>
<thead>
<tr>
<th>Name</th>
<th>Calypso (calypso.cmcc.it)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>IBM P575 node (32 cores Power6 4.7GHz per node)</td>
</tr>
<tr>
<td>Compiler</td>
<td>mpixlf90_r</td>
</tr>
<tr>
<td>Compiler flags</td>
<td>-qstrict -qfree=f90 -O3 -qrealsize=8 -qextname -qsource -q64 -qlargepage -qmaxmem=-1 -qarch=pwr6 -qtune=pwr6 -q64</td>
</tr>
<tr>
<td>Environment variables</td>
<td>export MP_INSTANCES=4 export MP_WAIT_MODE=poll export MP_POLLING_INTERVAL=3000000 export MP_SHARED_MEMORY=yes export MP_EUILIB=us export MP_EUIDEVICE=sn_all export LDR_CNTRL=TEXTPSIZE=64K@STACKPSIZE=64K@DATAPSIZE=64K export MP_TASK_AFFINITY=core</td>
</tr>
</tbody>
</table>
RESULTS

The wall clock time increases at each new release except for the last one, as shown in figure 1. The biggest increment appears between release v3.3.1 and v3.4.a. The wall clock time for the release v3.4.a is almost double with respect to the wall clock time of the v3.2.2. To evaluate the use of SMT, a decomposition 8x8 has been used. Enabling SMT, all of the 64 processes have been mapped on one node. Disabling SMT, 2 nodes have been allocated. Figure 2 shows that using half number of processors, the CPU time is less than double.

In the previous test the number of nodes has been fixed to 1 and the decomposition has been changed. The results show that the a computing node is better exploited when SMT is active. The difference of each new release with respect to the previous one is reported in figure 3.

The most significant increment occurs during the transition from v3.3.1 to v3.4.a. The v3.4b reduced the time with respect to the v3.4a.

The wall clock time behaviour for cfg=144 confirms the increment inserted at each new release. In the test reported in figure 4, the SMT is disabled.

The behaviour observed for the configuration with cfg=48 persists when cfg=144, as shown
in figure 5. A more detailed analysis has been made. We used the configuration cfg=48, SMT OFF and a domain decomposition of 8x8. We used gprof to trace each routine called by the stp main routine along the different releases. Table 1 gives the details at low level.

Considering only those routines that have a variation greater than 5% during the transition from one release to the next one, we can focus our attention only on the last 9 routines and namely: tra_zdf - tra_unswap - tra_ldf - tra_adv - zdf_tke - ldf_slp - dyn_zdf - dyn_vor - dyn_spg.

Figure 6 shows the wall clock time spent on those routines and highlights how the time changes during the evolution of the code.

Figure 7 better empathizes which routine is responsible for the increment of the wall clock time during the transition from a release to the next one. Only the difference in time is shown.

The increment of time during the transition from v3.2.2 to v3.3.0 is due to the tra_ldf and zdf_tke routines. The routines responsible for the increment of time during the transition from v3.3.0 to v3.31 are tra_ldf and tra_adv. Finally the routine dyn_zdf is the solely routine responsible for the increment of time in the release v3.4a.

MFS CONFIGURATION

The performance comparison between NEMO releases has been applied also to the MFS16 configuration [2]. This is a production configu-
ration with a scientific interest for CMCC. The tests have been carried out with the following data:

- Domain decomposition: 8x8
- SMT: disabled (32 core per node)
- Total number of nodes: 2
- Number of time steps: 144
- I/O: creation of output and restart file only at the end of the simulation

The wall clock time between v3.3.1 and v3.4a releases differs only for 5.7%. Having a look at each routine we can note that the increment of the execution time of the dyn_zdf routine is partially balanced by a reduction of the execution time of the tran_adv and other routines. Figure 8 reports the timing only for those routines with a difference greater then 5% of the execution time measured in release v3.3.1

Also for this configuration the inefficiencies previously described for dyn_zdf have a deep impact on the performance. The examined configuration does not activate any other routine with relevant performance inefficiency. The detailed values for each routine are reported in the table 3.

<table>
<thead>
<tr>
<th>Routine</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>dyn_zdf</td>
<td>6.61</td>
</tr>
<tr>
<td>dynldf</td>
<td>1.28</td>
</tr>
<tr>
<td>obc_rsd</td>
<td>0.53</td>
</tr>
<tr>
<td>zdf_trf</td>
<td>0.45</td>
</tr>
<tr>
<td>tra_adv</td>
<td>-0.29</td>
</tr>
<tr>
<td>sbc</td>
<td>-0.40</td>
</tr>
<tr>
<td>dyn_vor</td>
<td>-0.94</td>
</tr>
<tr>
<td>tra_nxt</td>
<td>-1.78</td>
</tr>
</tbody>
</table>

**ANALYSIS**

To identify exactly which is the reason of this increment of time in dyn_zdf routine, we focused the attention only on the releases v3.3.1 and v3.4.a and only on the dyn_zdf routine. Actually the dyn_zdf calls the dyn_zdf_imp. We have instrumented a region within the dyn_zdf_imp for both releases. The instrumented region excludes the calls to the wrk_array routines (that has been introduced in v3.4a).
Figure 9: Floating point operations (a and b) and L1 cache misses (c and d): MFS configuration
The configuration used for the instrumented executable is characterized by config=48, a domain decomposition 8x8, SMT OFF and 72 total timesteps. We first checked if the increment of time was due to new operations introduced in the new release, we checked the total number of floating point operations (Flop counter) using the HPM (High Performance Monitoring) and we got the result in figure 9(a) and 9(b).

The total execution time in release v3.4a is about 23 seconds greater than v3.3.1 and the time spent within the instrumented section in release v3.4a reached 21.3 seconds against 3.9 seconds on the same region in release v3.3.1. This confirms what has been observed with gprof. The result says also that the management of the work arrays does not have a deep impact on the computing performance. The most part of the time increment in the v3.4a is due to dyn_zdf_imp routine. Having a look at the total number of floating point operations, the v3.4a executes less operations than v3.3.1 (3648.5Mflop against 3810Mflop). Hence the new release reduces the number of flops and in some way tries to optimize the execution. But the execution rate of the floating point operations is drastically low with respect to the v3.3.1 (171.3Mflop/s against 978.5 Mflop/s). This is typically due to a bad use of the memory hierarchy.

We used the hardware counters to check the L1 data cache misses (see figure 9(c) and 9(d)). The results highlight that in the new release an inefficiency due to the bad use of L1 and L2 caches has been introduced. The number of L1 misses of the v3.4a is 2 order of magnitude greater than the v3.3.1 (317,388,531 against 5,007,385); also the L2 data cache accesses is doubled in the v3.4a (461M against 290M).

Having a look at the code, the lose of performance due to the bad cache usage is quite evident. The implementation of the dyn_zdf_imp in the release v3.3.1 accesses the 3D array sweeping the data following contiguous memory locations: as example see the snapshot in figure 10 where the elements of the arrays avmu and umask are accessed following the column wise order, indeed the first index (ji) iterates before the second index (jj) that iterates before the third index (jk).

In release v3.4a the loops have been restructured and the ordered access to the memory has been lost (see figure 11). In this case, the innermost loop, iterates on jk that is not the first index of the arrays.

Regarding the transition from v3.3.0 to v3.3.1, that corresponds to the introduction of the dynamical memory allocation, the major impact on performance can be seen on routines tra_df and tra_adv. But, contrary to what happened in the previous case where the code has been restructured. In this case both routines are the
same in v3.3.0 and v3.3.1 (apart for the calls to \texttt{wrk\_in\_use} and \texttt{wrk\_not\_released} functions in v3.3.1). We focused on the \texttt{tra\_ldf} that actually has only one call to the \texttt{tra\_ldf\_iso} routine (this is a leaf routine). We instrumented a region that excludes the calls to the work arrays functions. Figure 12(a) and 12(b) reports the values of the FLOP counters. The total number of floating point operations is exactly the same for both releases. This implies that the compiler performs the same optimizations regarding to the floating point operations. However the number of completed instructions in the v3.3.1 is almost twice the number of instructions of the v3.3.0. This implies that the compiler introduces more instructions (such as integer or load/store operation).

Taking into account the counters related to the cache misses, we can notice that there is not an evident difference. The L1D cache misses are of the same order of magnitude (see figure 12(c) and 12(d)). Once again we notice that the number of instructions in v3.3.1 is almost double even if the instruction rate (MIPS) for the v3.3.1 is greater than v3.3.0.

Analyzing the list file produced by the compiler, the transformations performed by the compiler and the assembler code, we can notice that, the access to an allocatable array (that is stored in the heap memory instead of the stack memory) produces a lot of register spilling that implies an increase number of load and store instructions. To confirm this observation we can see the counters related to the total number operations of the LSU (Load Store Unit): in v3.3.1 we have 11398M operations against 7651M (see figure 12(e) and 12(f)).

To conclude, the main reason for the performance decrement during the transition from v3.3.0 to the v3.3.1 release is due to the compiler that is not able to make the same optimizations when the code uses allocatable arrays and when the number of iterations of the loops are not known at compile time. The lack of performance observed during the transition to v3.4a release is mainly due to a worst use of the cache. The management of the work arrays, introduced in v3.4a, has not an evident impact on the performance.
### Common Pitfalls Coding a Parallel Model

#### v3.3.0

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Utilization rate</th>
<th>MIPS</th>
<th>Instructions per cycle</th>
<th>L2 traffic</th>
<th>L2 access from L2 per cycle</th>
<th>L2 bandwidth per processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOPS (a)</td>
<td>99.356 %</td>
<td>1231.56 MIPS</td>
<td>8.467</td>
<td>4.229 MBytes</td>
<td>1.564 MBytes</td>
<td>293.619 MBytes/seconds</td>
</tr>
<tr>
<td>FLOPS (b)</td>
<td>99.356 %</td>
<td>1231.56 MIPS</td>
<td>8.467</td>
<td>4.229 MBytes</td>
<td>1.564 MBytes</td>
<td>293.619 MBytes/seconds</td>
</tr>
<tr>
<td>FLOPS (c)</td>
<td>99.356 %</td>
<td>1231.56 MIPS</td>
<td>8.467</td>
<td>4.229 MBytes</td>
<td>1.564 MBytes</td>
<td>293.619 MBytes/seconds</td>
</tr>
<tr>
<td>FLOPS (d)</td>
<td>99.356 %</td>
<td>1231.56 MIPS</td>
<td>8.467</td>
<td>4.229 MBytes</td>
<td>1.564 MBytes</td>
<td>293.619 MBytes/seconds</td>
</tr>
<tr>
<td>FLOPS (e)</td>
<td>99.356 %</td>
<td>1231.56 MIPS</td>
<td>8.467</td>
<td>4.229 MBytes</td>
<td>1.564 MBytes</td>
<td>293.619 MBytes/seconds</td>
</tr>
<tr>
<td>FLOPS (f)</td>
<td>99.356 %</td>
<td>1231.56 MIPS</td>
<td>8.467</td>
<td>4.229 MBytes</td>
<td>1.564 MBytes</td>
<td>293.619 MBytes/seconds</td>
</tr>
</tbody>
</table>

### v3.3.1

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Utilization rate</th>
<th>MIPS</th>
<th>Instructions per cycle</th>
<th>L2 traffic</th>
<th>L2 access from L2 per cycle</th>
<th>L2 bandwidth per processor</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOPS (a)</td>
<td>99.356 %</td>
<td>1231.56 MIPS</td>
<td>8.467</td>
<td>4.229 MBytes</td>
<td>1.564 MBytes</td>
<td>293.619 MBytes/seconds</td>
</tr>
<tr>
<td>FLOPS (b)</td>
<td>99.356 %</td>
<td>1231.56 MIPS</td>
<td>8.467</td>
<td>4.229 MBytes</td>
<td>1.564 MBytes</td>
<td>293.619 MBytes/seconds</td>
</tr>
<tr>
<td>FLOPS (c)</td>
<td>99.356 %</td>
<td>1231.56 MIPS</td>
<td>8.467</td>
<td>4.229 MBytes</td>
<td>1.564 MBytes</td>
<td>293.619 MBytes/seconds</td>
</tr>
<tr>
<td>FLOPS (d)</td>
<td>99.356 %</td>
<td>1231.56 MIPS</td>
<td>8.467</td>
<td>4.229 MBytes</td>
<td>1.564 MBytes</td>
<td>293.619 MBytes/seconds</td>
</tr>
<tr>
<td>FLOPS (e)</td>
<td>99.356 %</td>
<td>1231.56 MIPS</td>
<td>8.467</td>
<td>4.229 MBytes</td>
<td>1.564 MBytes</td>
<td>293.619 MBytes/seconds</td>
</tr>
<tr>
<td>FLOPS (f)</td>
<td>99.356 %</td>
<td>1231.56 MIPS</td>
<td>8.467</td>
<td>4.229 MBytes</td>
<td>1.564 MBytes</td>
<td>293.619 MBytes/seconds</td>
</tr>
</tbody>
</table>

---

### Figure 12: Floating point operations (a and b), L1 cache misses (c and d) and LSU operations (e and f): MFS configuration
Bibliography
