Performance Analysis of the MPAS-Ocean Code using HPCToolkit and MIAMI

Gabriel Marin

February 11, 2014

MPAS-Ocean [4] is a component of the MPAS framework of climate models. MPAS-Ocean is an unstructured-mesh ocean model capable of using enhanced horizontal resolution in selected regions of the ocean domain. The code is publicly available for download [3] and comes with several input problems of different sizes corresponding to different simulation resolutions. In this initial study, we look at the per-core performance of version 2.0 of the MPAS-Ocean code. Our analysis was performed on a single node system with dual Intel Xeon E5-2690 CPUs, based on the Sandy Bridge micro-architecture. Each processor has 8 cores and a shared 20 MB L3 cache. We compiled the code with the Intel Fortran compiler 14.0.0 and optimization flags `-O3 -g`.

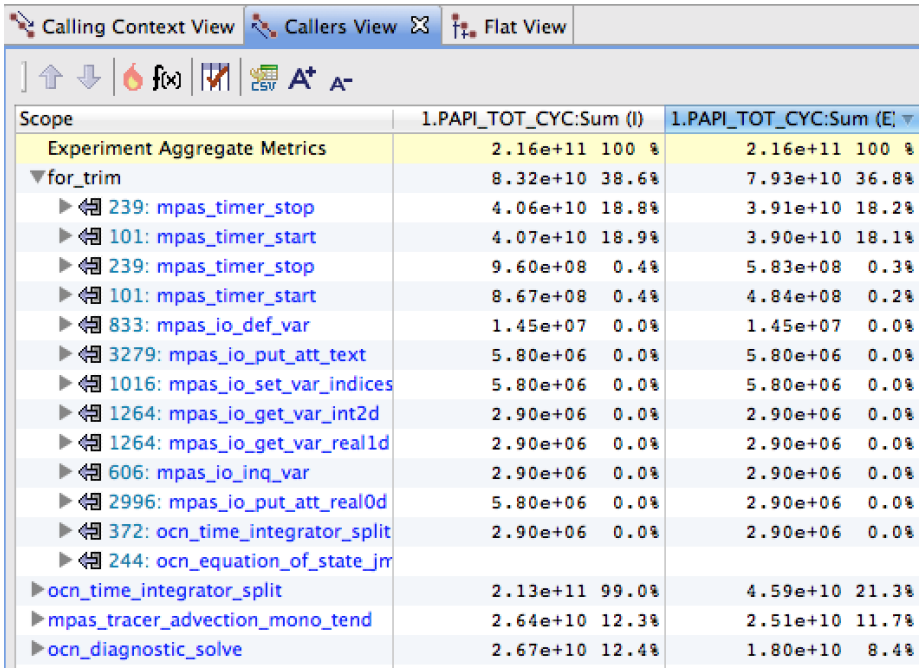


Figure 1: HPCToolkit time profile: the MPAS code spends 38.6% of time in the compiler intrinsic 'trim' invoked primarily from the software timer callipers.

Due to the small system size, we used an input problem of lower resolution (240km), and performed runs with one and four MPI processes. Figure 1 shows an initial HPCToolkit [1] time profile of the application running with a single MPI process. We use HPCToolkit's *callers* view and sort the results

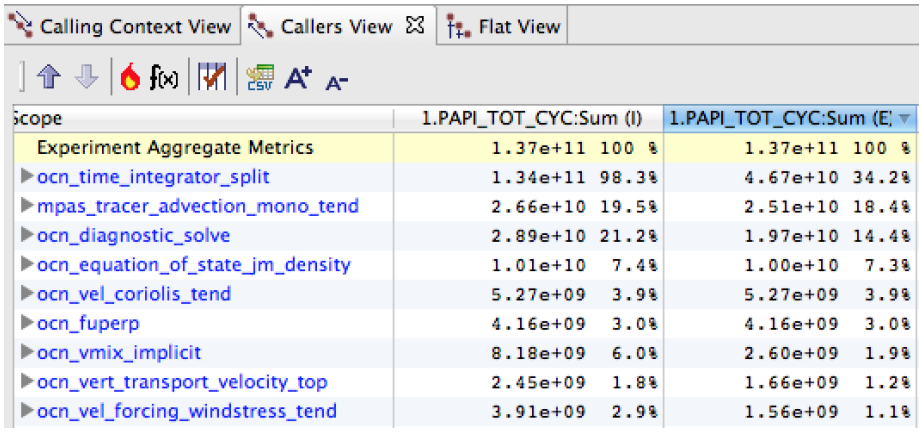


Figure 2: HPCToolkit time profile: the top time consuming routines in MPAS-Ocean after fixing the software timers.

by exclusive time, to identify the routines where the application spends the most time. Surprisingly, the application spends 38.6% of its execution time inside the Fortran compiler intrinsic `trim`. The *callers* view in HPCToolkit shows the places from where a routine is called, as well as a breakdown of the contribution of each call path to every metric. We notice that four call sites in two software timer calipers account for 38.5% of the entire time spent in routine `for_trim`.

Inspecting the timers code, we noticed repeated, unnecessary calls to the `trim` intrinsic for each invocation of the timer calipers. We changed the code to call the `trim` intrinsic once per caliper use, caching the result of its output. Figure 2 presents the updated HPCToolkit time profile after this code fix-up. The total execution time dropped by 36.5% from *2.16e11* to *1.37e11*. Routine `for_trim` accounts for 0.1% of the execution time in the new code version, thus, it does not show up in the list of top time consuming routines. We use the fixed version of the code as the baseline for further analysis and optimization.

Performance analysis using MIAMI

Figure 3 shows a snapshot of several performance metrics computed by MIAMI [2]. MIAMI uses a modeling approach based on first-order principles to estimate the instruction schedule cost of an application on a particular architecture, to understand the data reuse patterns responsible for the highest number of data transfers between the various levels of the memory hierarchy, and to identify the memory accesses that cannot be effectively prefetched by streaming hardware prefetchers.

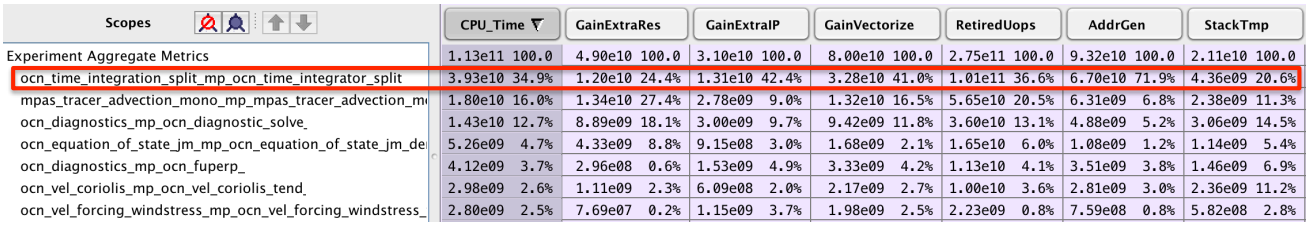


Figure 3: MIAMI performance results: the top routines ranked by instruction schedule cost.

The metrics included in Figure 3 focus on the insight provided by MIAMI into the instruction schedule cost of the MPAS-Ocean code. The metrics included in the figure are, in order: the instruction schedule cost, the potential for improvement from additional machine resources, the potential for improvement from additional instruction level parallelism, the potential for improvement from vectorization, the number of retired MIAMI micro-operations[1], the number of micro-operations used for address arithmetic, and the number of scalar accesses to the program stack.

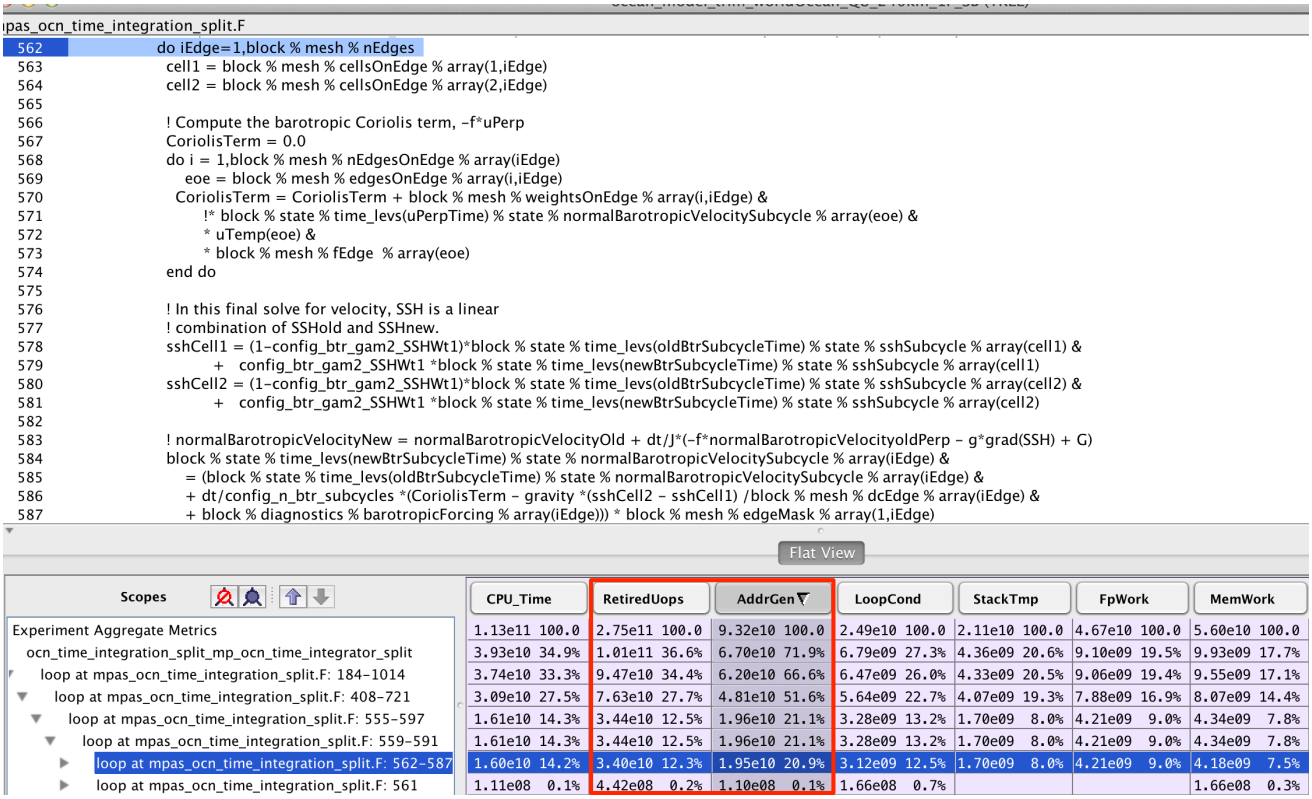
Before delving into the analysis of the results, we present a short guide for the *reading and interpretation* of MIAMI results. MIAMI can output performance results in CSV format, to enable parsing with a post-processing script, and in XML format for top-down analysis using `hpcviewer`, the viewer application distributed with HPCToolkit. The screenshots included in this paper show MIAMI results displayed in `hpcviewer`. The viewer presents data in tabular format. Each metric is shown in a separate column. The rows correspond to program scopes such as loops and routines. We can expand a particular scope to understand the contribution of its inner scopes to the various metrics. The percentages visible on the right side of each cell (see Figure 3) are added by the viewer and are computed column-wise. They represent the contribution of a scope (and its children) to the total value of that metric corresponding to the entire experiment. The percentages are useful to quickly assess that a particular routine is responsible for 25% of the running time of an application, or that it accounts for 40% of the potential for improvement. However, many times we are interested in the relationship between different metrics for a particular scope. In such cases, we have to ignore the pre-computed percentages and compare the absolute values displayed inside the cells on the same row.

The results in Figure 3 are sorted by instruction schedule cost. The execution times shown in the figure do not include a memory penalty component. However, the list of top time consuming routines predicted by MIAMI matches the list and the order of routines ranked by the actual execution time measured with HPCToolkit and shown in Figure 2. Out of the three *potential for improvement* metrics computed by MIAMI, the improvement potential from vectorization is by far the largest at *8.00e10* out of *1.13e11* total instruction schedule cost. These metrics indicate that most of the code could not be auto vectorized by the compiler. While there is a large theoretical potential for improvement from vectorization, transforming the code to provide enough data parallelism in inner loops and adjusting the data layout so that the compiler can automatically vectorize a large fraction of the computation, is not a small task by any measure.

However, one metric that stands out in Figure 3 is the count of micro-operations that perform address arithmetic (metric `AddrGen`). At the entire program level, *9.32e10* out of *2.75e11* total retired micro-operations, or roughly one out of three micro-operations performs address arithmetic. However, when we look at the most time consuming routine, `ocn_time_integrator_split`, we see that this routine is responsible for 36.6% of all retired micro-operations, but it accounts for 72% of all address arithmetic instructions. In fact, just about two out of every three micro-operations executed by this routine perform address arithmetic.

Figure 4 provides a more detailed accounting of executed micro-operations broken down by functionality. We use static opcode decoding and data flow analysis to classify micro-operations. The figure includes the following metrics: the estimated instruction schedule time, the total number of retired micro-operations, the number of micro-operations performing address arithmetic, the number of micro-operations evaluating loop branch conditions, the number of scalar stack accesses, the number of micro-operations performing floating point work, and the number of micro-operations performing memory work. A few other categories are omitted for brevity.

[1]MIAMI micro-operations are very similar to x86 micro-operations, but the mapping is not necessarily 1-to-1



```
ipas_ocn_time_integration_split.F
562                do iEdge=1,block % mesh % nEdges
563                   cell1 = block % mesh % cellsOnEdge % array(1,iEdge)
564                   cell2 = block % mesh % cellsOnEdge % array(2,iEdge)
565
566                   ! Compute the barotropic Coriolis term, -f*uPerp
567                   CoriolisTerm = 0.0
568                   do i = 1,block % mesh % nEdgesOnEdge % array(iEdge)
569                      eoe = block % mesh % edgesOnEdge % array(i,iEdge)
570                      CoriolisTerm = CoriolisTerm + block % mesh % weightsOnEdge % array(i,iEdge) &
571                         !* block % state % time_levs(uPerpTime) % state % normalBarotropicVelocitySubcycle % array(eoe) &
572                         * uTemp(eoe) &
573                         * block % mesh % fEdge  % array(eoe)
574                   end do
575
576                   ! In this final solve for velocity, SSH is a linear
577                   ! combination of SSHold and SSHnew.
578                   sshCell1 = (1-config_btr_gam2_SSHWt1)*block % state % time_levs(oldBtrSubcycleTime) % state % sshSubcycle % array(cell1) &
579                            +  config_btr_gam2_SSHWt1 *block % state % time_levs(newBtrSubcycleTime) % state % sshSubcycle % array(cell1)
580                   sshCell2 = (1-config_btr_gam2_SSHWt1)*block % state % time_levs(oldBtrSubcycleTime) % state % sshSubcycle % array(cell2) &
581                            +  config_btr_gam2_SSHWt1 *block % state % time_levs(newBtrSubcycleTime) % state % sshSubcycle % array(cell2)
582
583                   ! normalBarotropicVelocityNew = normalBarotropicVelocityOld + dt/J*(-f*normalBarotropicVelocityoldPerp - g*grad(SSH) + G)
584                   block % state % time_levs(newBtrSubcycleTime) % state % normalBarotropicVelocitySubcycle % array(iEdge) &
585                      = (block % state % time_levs(oldBtrSubcycleTime) % state % normalBarotropicVelocitySubcycle % array(iEdge) &
586                      + dt/config_n_btr_subcycles *(CoriolisTerm - gravity *(sshCell2 - sshCell1) /block % mesh % dcEdge % array(iEdge) &
587                      + block % diagnostics % barotropicForcing % array(iEdge))) * block % mesh % edgeMask % array(1,iEdge)
```

Flat View

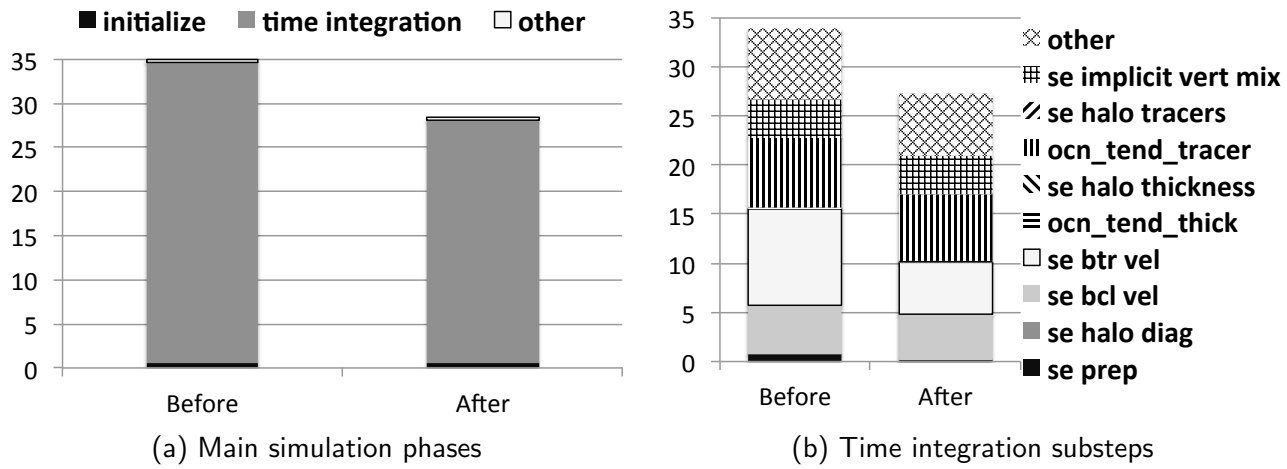
Scopes	CPU_Time	RetiredUops	AddrGen	LoopCond	StackTmp	FpWork	MemWork
Experiment Aggregate Metrics	1.13e11 100.0	2.75e11 100.0	9.32e10 100.0	2.49e10 100.0	2.11e10 100.0	4.67e10 100.0	5.60e10 100.0
ocn_time_integration_split_mp_ocn_time_integrator_split	3.93e10 34.9%	1.01e11 36.6%	6.70e10 71.9%	6.79e09 27.3%	4.36e09 20.6%	9.10e09 19.5%	9.93e09 17.7%
loop at mpas_ocn_time_integration_split.F: 184-1014	3.74e10 33.3%	9.47e10 34.4%	6.20e10 66.6%	6.47e09 26.0%	4.33e09 20.5%	9.06e09 19.4%	9.55e09 17.1%
loop at mpas_ocn_time_integration_split.F: 408-721	3.09e10 27.5%	7.63e10 27.7%	4.81e10 51.6%	5.64e09 22.7%	4.07e09 19.3%	7.88e09 16.9%	8.07e09 14.4%
loop at mpas_ocn_time_integration_split.F: 555-597	1.61e10 14.3%	3.44e10 12.5%	1.96e10 21.1%	3.28e09 13.2%	1.70e09 8.0%	4.21e09 9.0%	4.34e09 7.8%
loop at mpas_ocn_time_integration_split.F: 559-591	1.61e10 14.3%	3.44e10 12.5%	1.96e10 21.1%	3.28e09 13.2%	1.70e09 8.0%	4.21e09 9.0%	4.34e09 7.8%
loop at mpas_ocn_time_integration_split.F: 562-587	1.60e10 14.2%	3.40e10 12.3%	1.95e10 20.9%	3.12e09 12.5%	1.70e09 8.0%	4.21e09 9.0%	4.18e09 7.5%
loop at mpas_ocn_time_integration_split.F: 561	1.11e08 0.1%	4.42e08 0.2%	1.10e08 0.1%	1.66e08 0.7%			1.66e08 0.3%

Figure 4: MIAMI performance results: breakdown of executed micro-operations by functionality.

We sorted the data by the number of address arithmetic micro-operations, and we expanded the scopes to identify the loops that contribute the highest number of address arithmetic operations. Figure 4 shows the source code of one such loop located in file `mpas_ocn_time_integration_split.F` at lines `562 - 587`. There does not seem to be anything out of ordinary with this code, except the extensive use of objects of nested derived types and references to the inner components of such objects. While the compiler should be able to hoist outside the loop most of the loop-invariant address arithmetic code needed to access the inner components of these objects, our analysis shows that 57% of the micro-operations executed by this loop perform address arithmetic. We manually transformed the code to store pointers to the objects' inner components outside the loop. Inside the loop, we are using the pointers to reference the loop-dependent elements of each inner component. We verified that this approach significantly reduced execution time for the transformed loop. Next, we applied the same transformations to all the loops inside routine `ocn_time_integrator_split` that were highlighted by MIAMI as performing a disproportionately large number of address arithmetic operations.

Figure 5 shows side by side execution time profiles of the MPAS-Ocean code before and after our code transformations. The times were collected using the integrated MPAS software timers on our testbed machine. We used the 240km problem size, one MPI process, and 10 days of simulation. Figure 5a shows the costs of the main simulation phases. The time integration step clearly dominates the execution time. The transformed code achieves a 23% speedup in serial mode. Figure 5b shows the contribution of the various time integration substeps. The data preparatory substep (timer `se prep`) and the barotropic velocity prediction substep (timer `se btr vel`) benefit the most from our transformations.

Figure 6 presents a summary of the number and type of micro-operations executed by the MPAS code and by the `ocn_time_integrator_split` routine, before and after our transformations. The number



(a) Main simulation phases (b) Time integration substeps

Figure 5: Execution time breakdown before and after optimizations.

	Scope	RetiredUOps	AddrGen	LoopCond	StackTemp	RegMove	FpWork	IntWork	MemWork
Before	MPAS code	2.75E+11	9.32E+10	2.49E+10	2.11E+10	8.49E+09	4.67E+10	1.67E+10	5.60E+10
	ocn_time_integrator_split	1.01E+11	6.70E+10	6.79E+09	4.36E+09	1.12E+09	9.10E+09	9.03E+08	9.93E+09
After	MPAS code	2.27E+11	4.03E+10	2.48E+10	2.59E+10	8.50E+09	4.66E+10	1.77E+10	5.58E+10
	ocn_time_integrator_split	5.28E+10	1.40E+10	6.68E+09	9.16E+09	1.13E+09	9.08E+09	2.03E+09	9.83E+09

Figure 6: Number of executed micro-operations before and after our transformations.

of address arithmetic operations dropped by almost 80% for the time integrator routine, and by 57% for the entire application. At the same time, we observe a more than doubling in the number of scalar stack references for the transformed routine. Stack references correspond to scalar temporary variables and spill / unspill code inserted by the compiler. By hoisting the bulk of the address arithmetic code outside of loops, we increased the number of concurrent live ranges in the code, which makes the register allocator's job more difficult. However, overall, our transformations reduced the number of micro-operations executed by the `ocn_time_integrator_split` routine by 48%, which yields a 23% speedup as mentioned earlier.

There are other likely opportunities for optimization of the MPAS code's per-core performance. However, clear understanding of these opportunities requires knowledge of the application and a careful review of the performance results produced by MIAMI and other tools.

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. Hpctoolkit: tools for performance analysis of optimized parallel programs http://hpctoolkit.org. *Concurr. Comput. : Pract. Exper.*, 22(6):685–701, April 2010.

[2] Gabriel Marin, Jack Dongarra, and Dan Terpstra. MIAMI: A framework for application performance diagnosis. In *IEEE International Symposium on Performance Analysis of Systems and Software, 2014*. (to appear), March 2014.

[3] MPAS: Model for Prediction Across Scales. `http://mpas-dev.github.io`.

[4] Todd Ringler, Mark Petersen, Robert L. Higdon, Doug Jacobsen, Philip W. Jones, and Mathew Maltrud. A multi-resolution approach to global ocean modeling. *Ocean Modelling*, 69(0):211 – 232, 2013.