Available online at www.prace-ri.eu

Partnership for Advanced Computing in Europe

Performance Analysis of Alya on a Tier-0 Machine using Extrae

Jorge Rodríguez [a]*

[a]*BSC-CNS: Barcelona Supercomputing Center, Torre Girona, C/Jordi Girona, 31, 08034 Barcelona, Spain*

Abstract

Alya [5] is a computational mechanics code capable of solving different physics. It has been extensively used in MareNostrum III (BSC's Tier-0 machine), and it has been also used as a benchmarking code in PRACE Unified European Applications Benchmark Suite. In this document, Extrae will be used to collect and analyze performance data during an Alya simulation in a petaflop environment.

As a result of the performance analysis using Extrae [2] [3], some potential improvements in Alya have shown up, and if considered, exascale scalability could be achieved.

Application Code: Alya

1. Introduction

Alya is a computational mechanics code developed at BSC, which solves partial differential equations (PDEs) in non-structured meshes, using finite element methods. Among the problems it solves are: convection-diffusion reactions, incompressible flows, compressible flows, turbulence, bi-phasic flows and free surface, excitable media, acoustics, thermal flow, quantum mechanics (DFT) and solid mechanics (large strain).

The source code is written in Fortran 90/95 and parallelized using MPI and OpenMP. Being a large scale scientific code, Alya demands substantial I/O processing, which may consume considerable time and can therefore potentially reduce speed-up at exascale.

Alya is organized in modules that solve different physical problems. These modules are now being used in production, and scalability has been proven on 20,000 cores using meshes with billions of elements. The code has been tested and proven to run efficiently on many Tier-0 machines such as JUGENE/JUQUEEN, FERMI (BlueGene/Q), CURIE (Bull Cluster) and other Intel-based clusters around the world.

2. Code porting and tracing

In order to compile a stable release of Alya, firstly we have to adapt a configure file to tune compiler flags and to use the available software components in MareNostrum III (Intel MPI, OpenMP, HDF5, METIS etc.). For the current compilation, the following components will be used:

-Intel MPI version 4.1.1.036
-Third party library: Metis 4.0.1

* Corresponding author. *E-mail address*: Jorge.rodriguez@bsc.es

```
f90==     mpif90 -module $O -c -O2
fpp90==   mpif90 -fpp -module $O -c -O2
cpp==     mpicc -c -O2
link==    mpif90 -L../../Thirdparties/metis-4.0/ -lmetis
```

Note that `mpif90` and `mpicc` are wrapper commands to `ifort` and `icc` Intel compilers:

```
icc -I/gpfs/apps/MN3/INTEL/impi/4.1.1.036/intel64/include
-L/gpfs/apps/MN3/INTEL/impi/4.1.1.036/intel64/lib -Xlinker --enable-new-dtags -Xlinker -rpath
-Xlinker /gpfs/apps/MN3/INTEL/impi/4.1.1.036/intel64/lib -Xlinker -rpath -Xlinker
/opt/intel/mpi-rt/4.1 -lmpigf -lmpi -lmpigi -ldl -lrt -lpthread

ifort -I/gpfs/apps/MN3/INTEL/impi/4.1.1.036/intel64/include/gfortran
-I/gpfs/apps/MN3/INTEL/impi/4.1.1.036/intel64/include
-L/gpfs/apps/MN3/INTEL/impi/4.1.1.036/intel64/lib -Xlinker --enable-new-dtags -Xlinker -rpath
-Xlinker /gpfs/apps/MN3/INTEL/impi/4.1.1.036/intel64/lib -Xlinker -rpath -Xlinker
/opt/intel/mpi-rt/4.1 -lmpigf -lmpi -lmpigi -ldl -lrt -lpthread
```

For the purpose of this study, we will use 2 of the modules provided by Alya: '`nastin`' and '`parall`'. These modules have to be configured and compiled following these steps:

```
configure -g -f=configure-linux-mn.txt nastin parall
make
```

After a successful compilation, the executable file `Alya.g` will be generated. The next step is to create a trace file of the execution for the subsequent analysis. In order to achieve it, Extrae will be used to instrument the code and to collect the performance data:

1. Selection of the PAPI hardware counters to be collected at runtime: Total instructions, Total cycles and Level 1 data cache misses. An XML file with the following format should be edited:

```
<?xml version='1.0'?>
<trace enabled="yes"
 home="/apps/CEPBATOOLS/extrae/latest/impi/64"
 initial-mode="detail"
 type="paraver"
  xml-parser-id="Id: xml-parse.c 2327 2013-11-22 11:47:07Z harald $">
  <mpi enabled="yes">    <counters enabled="yes" />  </mpi>
  <openmp enabled="no"><locks enabled="no" /><counters enabled="yes" /></openmp>
  <pthread enabled="no"><locks enabled="no" /><counters enabled="yes" /></pthread>
  <callers enabled="yes"><mpi enabled="yes">1-3</mpi><sampling enabled="no">1-5</sampling>
  </callers>
  <user-functions enabled="no"><counters enabled="yes" /></user-functions>
  <counters enabled="yes">
    <cpu enabled="yes" starting-set-distribution="1">
      <set enabled="yes" domain="all" changeat-globalops="5">
        PAPI_TOT_INS,PAPI_TOT_CYC,PAPI_L1_DCM
        <sampling enabled="no" frequency="100000000">PAPI_TOT_CYC</sampling>
      </set>
      <set enabled="yes" domain="user" changeat-globalops="5">
        PAPI_TOT_INS,PAPI_FP_INS,PAPI_TOT_CYC
      </set>
    </cpu>
    <network enabled="no" />
    <resource-usage enabled="no" />
    <memory-usage enabled="no" />
  </counters>
  <storage enabled="no">
    <trace-prefix enabled="yes">TRACE</trace-prefix>
    <size enabled="no">50</size>
    <temporal-directory enabled="no"></temporal-directory>
    <final-directory enabled="no"></final-directory>
    <gather-mpits enabled="no" />
  </storage>
  <buffer enabled="yes"><size enabled="yes">500000</size><circular enabled="no" />
  </buffer>
  <trace-control enabled="no">
    <file enabled="no" frequency="5M"></file>
    <global-ops enabled="no"></global-ops>
    <remote-control enabled="no">
      <signal enabled="no" which="USR1"/>
    </remote-control>
  </trace-control>
  <others enabled="no"><minimum-time enabled="no">10M</minimum-time></others>
  <bursts enabled="no">
    <threshold enabled="yes">500u</threshold>
    <mpi-statistics enabled="yes" />
  </bursts>
  <cell enabled="no">
    <spu-file-size enabled="yes">5</spu-file-size>
    <spu-buffer-size enabled="yes">64</spu-buffer-size>
    <spu-dma-channel enabled="no">2</spu-dma-channel>
  </cell>
  <sampling enabled="no" type="default" period="50m" />
  <merge enabled="no" synchronization="default" binary="Alya.g" tree-fan-out="16"
    max-memory="512" joint-states="yes" keep-mpits="yes" sort-addresses="yes"
    remove-files="no"
  >
</merge>
</trace>
```

2. Adapt a wrapper script that will be executed before the executable file. This wrapper depends on the used MPI implementation (Intel MPI, OpenMPI), and on the programming language of the source code (C, Fortran).

```
#!/bin/bash

export EXTRAE_HOME=/apps/CEPBATOOLS/extrae/latest/impi/64
export EXTRAE_CONFIG_FILE=extrae.xml
# For C and Fortran apps
export LD_PRELOAD=${EXTRAE_HOME}/lib/libmpitrace.so:${EXTRAE_HOME}/lib/libmpitracef.so

## Run the desired program
$*
```

3. Submit a job with the previous trace wrapper script in the following way:

```
mpirun ./trace.sh ./Alya.g input
```

4. Once the job has been successfully executed, the performance data per core must be collected and merged into the single trace file (with the extension `.prv`). In order to do it, we must create a parallel job in MareNostrum such as:

```
#!/bin/bash
#BSUB -n 256
#BSUB -o %J.out
#BSUB -e %J.err
#BSUB -R"span[ptile=16]"
#BSUB -W 1:00

mpirun /apps/CEPBATOOLS/extrae/latest/impi/64/bin/mpimpi2prv -e ./Alya.g -syn -f
./TRACE.mpits -o alya_4096.prv
```

5. The generated trace file can now be opened with Paraver [2][3] for performance analysis by executing:

```
/apps/CEPBATOOLS/wxparaver/latest/bin/wxparaver alya_256.prv
```

3. Performance analysis

The code of Alya [1] is modular and it is developed using finite element methods, which makes it particularly adaptable in terms of scalability.
For this study, we will focus on the MPI version of Alya; however, OpenMP support is also available to allow shared memory parallelism within a compute node.
Basically, Alya uses mesh partitioning which is automatically done by METIS [4]. As a result, METIS creates a set of subdomains that communicate via MPI with a typical Master-Slave strategy.
Within this strategy, the master process reads the mesh and performs the partition of the mesh into subdomains, which will be managed by the rest of the processes (slaves). These processes will be in charge of solving the resulting system solution in parallel.

In the assembling tasks, no communication is needed between the slaves, and the scalability depends only on the load balancing.
In the iterative solvers, the scalability depends on the size of the interfaces and on the communication scheduling.
During the execution of the iterative solvers, two main types of communications are required:

- Global communications via `MPI_AllReduce`, which are used to compute residual norms and scalar products
- Point-to-point communications via `MPI_SendRecv`, which are used when sparse matrix-vector products are calculated.

All solvers need both types of communications

In the current study, we will use Alya executions with Test Case B as input (a 27 million mesh representing the respiratory system), which can be obtained from PRACE UEABS [6].

As we can see in a general Paraver window, the colour meanings are:

- Blue for computation
- Orange for collective calls
- Yellow for source and destination of MPI calls

In the following figure (Fig. 1), we have an Alya trace execution using 2048 processors in MareNostrum III. Figure 1 represents the initialization steps of Alya, and we can see that there is some load imbalance between tasks, as they have different duration depending on the workload.

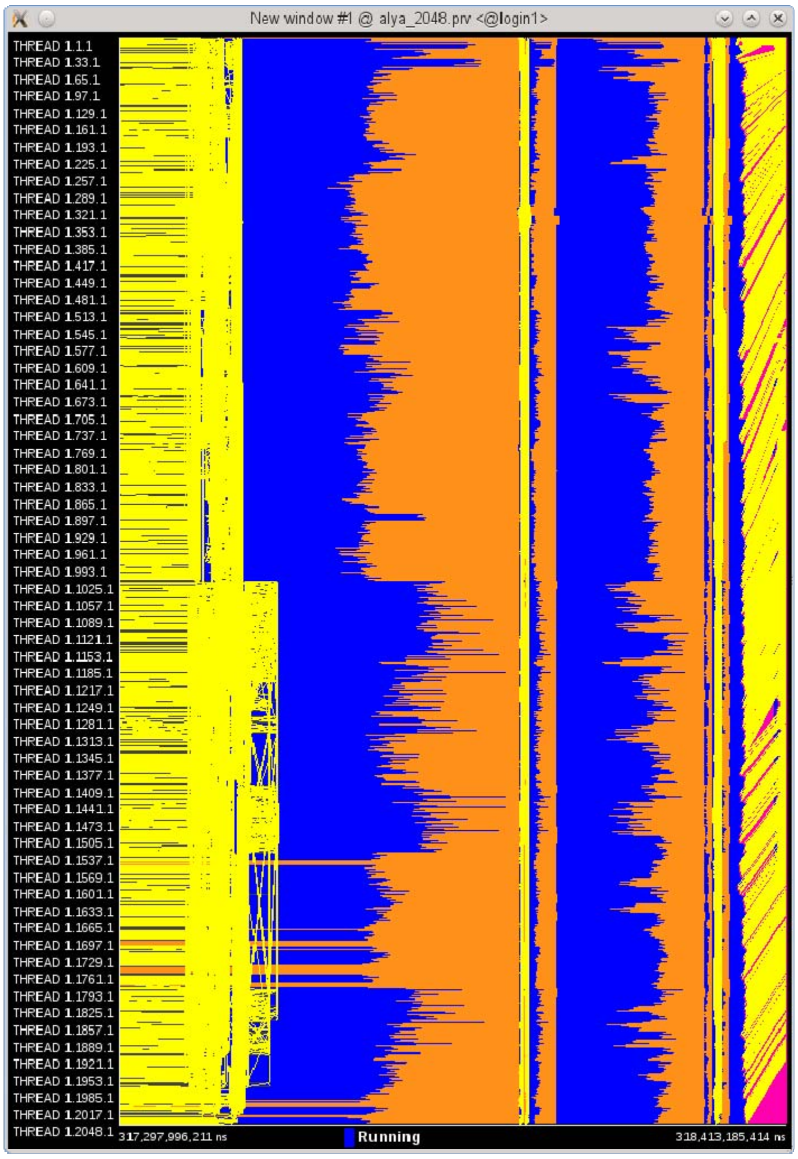
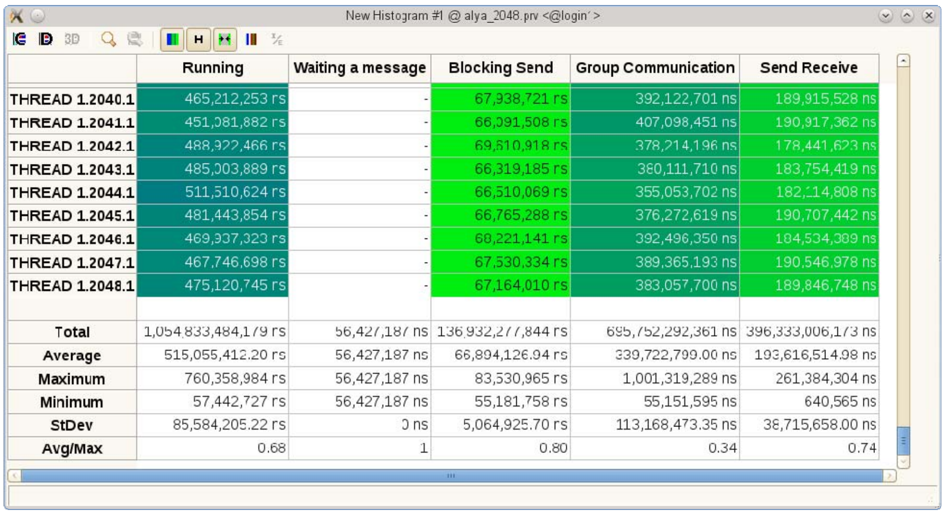


Figure 1: Alya trace file with 2048 processes

Paraver is also able to show a timing table, called Histogram, in which the different timings for a certain operation can be checked. In this case (Fig. 2), we can see a variation in compute time between threads in the range of 57 ms – 760 ms. However, this specific part of the code is not iterative, as it is only executed at start-up, hence optimization of this part of the code would not make a significant impact on the overall performance of the code.


New Histogram #1 @ alya_2048.prv <@login´>

	Running	Waiting a message	Blocking Send	Group Communication	Send Receive
THREAD 1.2040.1	465,212,253 rs	-	67,938,721 rs	392,122,701 ns	189,915,528 ns
THREAD 1.2041.1	451,081,882 rs	-	66,091,508 rs	407,098,451 ns	190,917,362 ns
THREAD 1.2042.1	488,922,466 rs	-	69,610,918 rs	378,214,196 ns	178,441,623 ns
THREAD 1.2043.1	485,003,889 rs	-	66,319,185 rs	380,111,710 ns	183,754,419 ns
THREAD 1.2044.1	511,510,624 rs	-	66,510,069 rs	355,053,702 ns	182,114,808 ns
THREAD 1.2045.1	481,443,854 rs	-	66,765,288 rs	376,272,619 ns	190,707,442 ns
THREAD 1.2046.1	469,937,323 rs	-	68,221,141 rs	392,496,350 ns	184,534,389 ns
THREAD 1.2047.1	467,746,698 rs	-	67,530,334 rs	389,365,193 ns	190,546,978 ns
THREAD 1.2048.1	475,120,745 rs	-	67,164,010 rs	383,057,700 ns	189,846,748 ns
Total	1,054,833,484,179 rs	56,427,187 ns	136,932,277,844 rs	695,752,292,361 ns	396,333,006,173 ns
Average	515,055,412.20 rs	56,427,187 ns	66,894,126.94 rs	339,722,799.00 ns	193,616,514.98 ns
Maximum	760,358,984 rs	56,427,187 ns	83,530,965 rs	1,001,319,289 ns	261,384,304 ns
Minimum	57,442,727 rs	56,427,187 ns	55,181,758 rs	55,151,595 ns	640,565 ns
StDev	85,584,205.22 rs	0 ns	5,064,925.70 rs	113,168,473.35 ns	38,715,658.00 ns
Avg/Max	0.68	1	0.80	0.34	0.74

Figure 2: Histogram of an Alya execution with 2048 processes

Trace files obtained with Extrae collect a large amount of data (hardware counters, states, bursts, events), which may lead to a trace file of several gigabytes in size (e.g. the 2048-core execution generated a 66 GB trace file). As it is not possible to manage such an amount of data on an average computer, there are some options to reduce the trace file size to a few gigabytes or even some megabytes in size. Among these options, a trace file can be cut just for a specific time interval, or the trace file can be filtered to show some of the events in the trace.
However, in the next part of the study we will focus on the same Alya execution using 256 cores, as the performance bottlenecks detected will also affect larger executions.

In Fig.3 we have a general Alya execution with 256 processes, and we will focus on the largest communication part (in yellow).

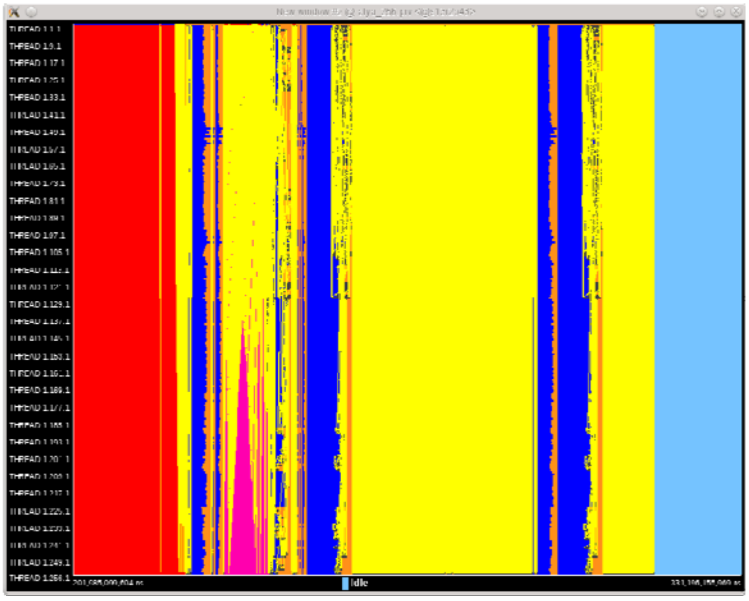


Figure 3: Alya trace file with 256 processes

Consequently, if we zoom in the largest yellow part, we will see an iterative part of the code, show in Fig.4

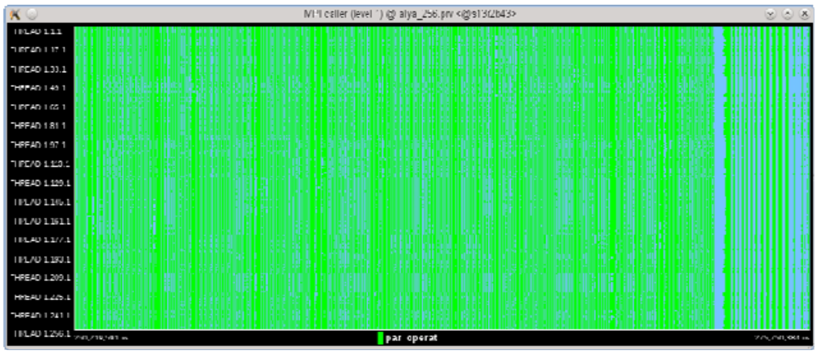


Figure 4: Iterative part of Alya with 256 processes (zoom out)

After a close look, we can see some load imbalance in the `par_operat()` Fortran routine between the different processes (Fig.5 in green). According to the histogram (Fig.6) at this part of the execution, the timings in `par_operat()` calls vary from 106 ms to 292 ms.

If we look at the source code for subroutine `par_operat()` we can check that depending on the process rank there are different loops and different `MPI_AllReduce()` calls to be executed, which will explain the mentioned behaviour.

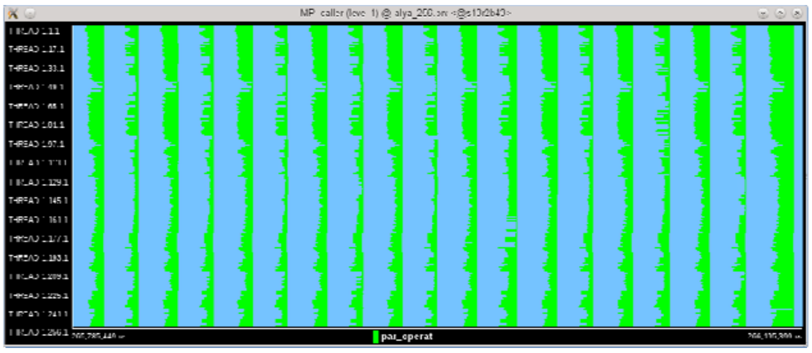
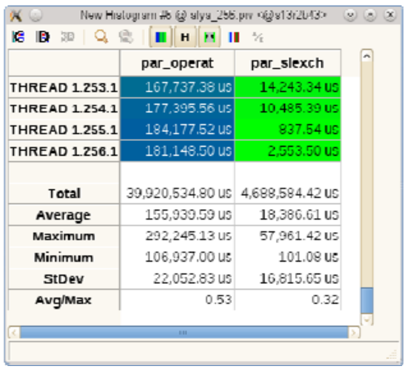


Figure 5: Iterative part of Alya with 256 processes (zoom in)


New Histogram #6 @ alya_256.prv <@s13r2b43>

	par_operat	par_slexch
THREAD 1.253.1	167,737.38 us	14,243.34 us
THREAD 1.254.1	177,395.56 us	10,485.39 us
THREAD 1.255.1	184,177.52 us	837.54 us
THREAD 1.256.1	181,148.50 us	2,553.50 us
Total	39,920,534.80 us	4,688,584.42 us
Average	155,939.59 us	18,386.61 us
Maximum	292,245.13 us	57,961.42 us
Minimum	106,937.00 us	101.08 us
StDev	22,052.83 us	16,815.65 us
Avg/Max	0.53	0.32

Figure 6: Histogram of an Alya execution with 256 processes

4. Conclusion

In this study, we have analysed the performance of a Tier-0 code (Alya) using Extrae and Paraver for collecting and viewing the performance data.
Moreover, we have verified the powerful options provided by Extrae and Paraver in terms of detail, insight and parallel behaviour.
Regarding the use of Extrae and Paraver, we confirm that they are very valuable tools when getting an in-depth analysis of a code's behaviour. We also conclude that it is a key toolset in the exascale path because of its low level detail of hardware analysis. However, when generating trace files for a very large amount of cores (10,000 cores or above), we may have some intricacy in merging the performance data into one single trace file, or even managing a trace file of several gigabytes.

References

[1] G. Houzeaux, R. de la Cruz, M. Vázquez, Parallel Uniform Mesh Subdivision in Alya, PRACE deliverable
[2] BSC Performance Tools , Paraver internals and details, http://www.bsc.es/ssl/apps/performanceTools/files/docs/W2_Paraver_details.pdf
[3] BSC Performance Tools, Tools Scalability, http://www.bsc.es/ssl/apps/performanceTools/files/docs/T1_Scalability.pdf
[4] METIS, Family of Multilevel Partitioning Algorithms, http://glaros.dtc.umn.edu/gkhome/views/metis
[5] Alya System, http://www.bsc.es/computer-applications/alya-system
[6] PRACE UEABS, Unified European Applications Benchmark Suite, http://www.prace-ri.eu/ueabs?lang=en

Acknowledgements

This work was financially supported by the PRACE project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-312763.
The work is achieved using the PRACE Research Infrastructure resources MareNostrum III (BSC-CNS), Barcelona.