On process pinning options: MARCONI-FUSION

Tiago Tamissa Ribeiro[1, *]

[1] *Max-Planck-Institut für Plasmaphysik, EURATOM Association,*
Boltzmannstrae 2, 85748 Garching, Germany

(Dated: November 16, 2017)

CONTENTS

I. INTRODUCTION: PROCESS BINDING ON MARCONI

The MARCONI supercomputer uses the popular cache-coherent Non-Uniform-Memory-Access (ccNUMA) architecture in its nodes. This means in particular that not all the cores are equivalent in terms of how fast they can access data from the node's main memory. Specifically, a node on MARCONI has two sockets with 18 cores each, yielding 36 cores sharing a total of 128 GB of main memory. However, this memory is made out of two banks (of 64 GB), each directly connected to one of the two sockets, as schematized in Fig. 6. Using NUMA terminology, we say that the cores on each socket together with the attached memory bank form a *NUMA node*. Thus, a compute node on MARCONI is comprised of two *NUMA nodes*. For the sake of simplicity, we shall denominate *NUMA*

[*] email: tiago.ribeiro@ipp.mpg.de; URL: http://www.rzg.mpg.de/~ttr/

node by *socket* in the remainder text. Therefore, even though any core on the compute node can access the whole main memory of the node, the access is faster within the same socket than across the sockets. As a result, it is generally very important performance-wise to specify appropriate binding affinities of the computational processes to the hardware cores within a node at run-time, and to ensure that the processes remain pinned that way throughout the simulation. This holds for both OpenMP threads, as well as MPI tasks, including obviously their combination in hybrid codes.

In the remaining text we present examples of MARCONI PBS batch submission scripts that allow to obtain several relevant process binding affinities, when invoked together with the Intel MPI library. Indeed, OpenMP thread and MPI task pinning are discussed in Secs. I A and I B, respectively, while Sec. I C describes the hybrid case of using both. The first two provide the simplest batch submission script examples for codes which are either purely OpenMP or purely MPI. The latter provides a more general case, which although slightly more involved, works also for pure OpenMP and MPI cases, so it can be regarded as an interesting alternative if one does not want to have different scripts for different parallelization paradigms. Finally, Sec. II provides some intra-node strong scaling results of a production code (REFMULXp [1]), which serve to emphasize the practical relevance of the process binding concepts explained before.

The material presented here is not intended to be, in any fashion, exhaustive. It merely attempts to provide a sub-set of rules that are of practical use for the typical cases encountered within codes in the framework of the EUROfusion consortium. For more detailed information we suggest consulting the Intel manual pages on *Process Pinning* [3] and respective subsections, *Environment Variables for Process Pinning* [4] and *Interoperability with OpenMP API* [5].

A. OpenMP thread affinity

Let's begin with the pure OpenMP threaded case, for which the script provided in Tab. I is our baseline example. The lines that control the number of threads to be spawned and their binding affinity to the hardware cores are highlighted in blue. The first assumption that we make here, and that will be clear later on, is that one needs to ensure that pinning of the threads throughout the run. This is clearly reasonable, even before looking at any results, since we know already that the main memory bandwidth is not uniform within the node. This can be achieved via the `KMP_AFFINITY` environment variable. Therefore, it is recommended to always set it explicitly in the batch script.

The example script sets `KMP_AFFINITY=compact`, whereby the processes are mapped to neighboring cores as closely as possible, filling first the first socket, and only if there are more threads than cores per socket, do the threads begin to fill the second socket. This also means that there is no need to turn on the option `OMP_PROC_BIND=true`, which is superseded by the former. We further added the option `verbose` so that the explicit binding of the threads to the cores is written to the standard error file, which makes it easy to confirm the actual affinity used in the corresponding simulation afterwards.

```
#!/bin/bash
#$ -cwd
#PBS -N omp_12
#PBS -l select=1:ncpus=36:mpiprocs=1:mem=118GB
#PBS -l walltime=01:00:00
#PBS -q xfuaprod
#PBS -A <project>
#PBS -j eo
#PBS -m abe
#PBS -M <email.address>@ipp.mpg.de

module load <needed modules>
export OMP_NUM_THREADS=12
export KMP_AFFINITY=compact,verbose
BIN=a.out
CMDLOPT=''
./$BIN $CMDLOPT
```

TABLE I. Basic script used to run `a.out` with 12 threads distributed according to the default affinity on a MARCONI node, which at the time of writing of this document was *compact*.

The *compact* affinity of the 12 threads (processes) yielded by this script is depicted on the left side of Fig. 1. However, depending on the algorithm at hand, this might not necessarily be the best choice of binding pattern, as we shall see later, in Sec. II. Changing it is quite simple, namely, if we replace the blue line setting the `KMP_AFFINITY` in Tab. I with the following one

```
export KMP_AFFINITY=scatter,verbose
```

a *scatter* affinity is obtained, whereby the processes are mapped as remotely as possible so as not to share common resources, in this case, the L3 caches and the memory banks in each socket (see Fig. 6). The particular example yielded by the modified script is also illustrated in Fig. 1, but now on its right side.

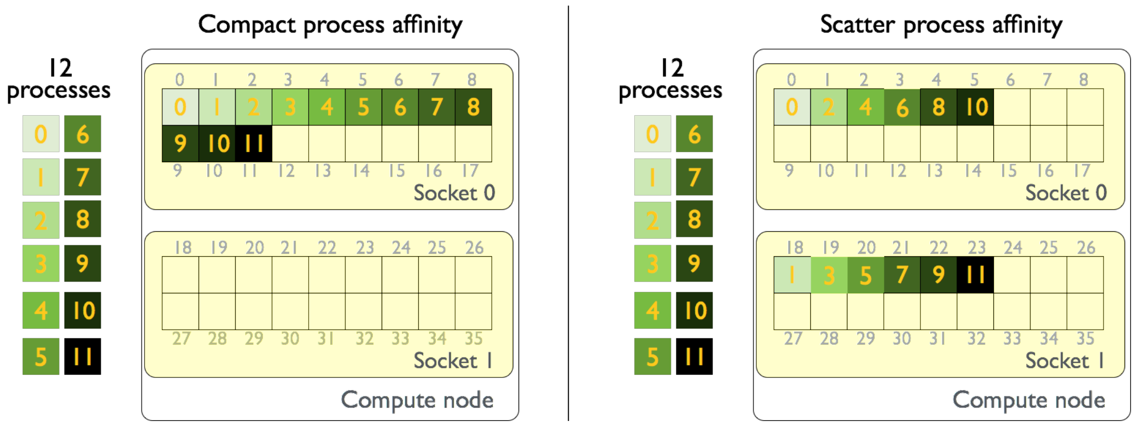

FIG. 1. Schematics of 12 processes (threads or tasks) distributed with *compact* (left) and *scatter* (right) affinity on a MARCONI node. Note how consecutive processes alternate between sockets on the latter.

If no value is explicitly set for the number of OpenMP threads, then the PBS system will fix this automatically to `OMP_NUM_THREADS=ncpus`, where `ncpus` is the number of cores requested in the select PBS directive, in our example 36. Note that, alternatively to using the variable `OMP_NUM_THREADS`, the number of threads can also be set by adding `ompthreads` to the select PBS directive above. For instance, for 12 threads, this would be

```
#PBS -l select=1:ncpus=36:mpiprocs=1:ompthreads=12:mem=118GB
```

It is also noteworthy to comment on the value used for the PBS batch resource list option `ncpus=36`. This tells the batch scheduler to reserve the full node, even if less resources are requested, like in the example provided (12 threads, instead of 36). On one hand, for the *EUROfusion* dedicated part of MARCONI (MARCONI-FUSION), this makes no difference in terms of budget accounting because we can only ask for full nodes, unlike the remaining part of the machine where nodes can be shared by different jobs. On the other hand, `ncpus=36` is really needed for pinning affinities other than *compact*, because resources (cores) not included in the PBS select directive will not be available at run-time. For example, if we were to set it instead to the number of threads in the script above, namely `ncpus=12`, then the setting `KMP_AFFINITY=scatter` would, in practice, yield a *compact* affinity. An even more extreme situation would be to set `ncpus=1`, in which case all the 12 threads would be pinned to core 0 of the node, regardless of the pinning affinity specified.

B. MPI task affinity

For a domain decomposed code, much of the same considerations about process pinning to the hardware resources (cores) apply. In this case, the processes are MPI tasks, so a different set of environment variables is responsible for specifying the affinities. For the simplest case of a pure MPI code, our baseline script is listed in Tab. II. As before, the lines highlighted in blue are the ones used to tune binding affinities, which now refer to the MPI tasks. The first sets `I_MPI_DEBUG=5` and its purpose is similar to the `verbose` keyword used before for the `KMP_AFFINITY` variable, namely, to write the explicit binding of the processes (MPI task ranks) to the cores to the standard error. The second line turns on the MPI task pinning `I_MPI_PIN=1`, so that the chosen affinity is enforced throughout the simulation, preventing the task to move between cores. Note that the process pinning features presented in the following are only valid if this option is turned on, which according to Intel reference pages [4] is the default value.

```
#!/bin/bash
#$ -cwd
#PBS -N mpi_12
#PBS -l select=1:ncpus=36:mpiprocs=12:mem=118GB
#PBS -l walltime=01:00:00
#PBS -q xfuaprod
#PBS -A <project>
#PBS -j eo
#PBS -m abe
#PBS -M <email.address>@ipp.mpg.de

module load <needed modules>
export I_MPI_DEBUG=5
export I_MPI_PIN=1
export I_MPI_PIN_PROCESSOR_LIST=0-35
BIN=a.out
CMDLOPT=''
mpirun -prepend-rank ./$BIN $CMDLOPT
```

TABLE II. Basic script used to run `a.out` with 12 MPI tasks distributed *compactly* on a MARCONI node. The `-prepend-rank` option simply adds the MPI rank number in front of each output line, which can be a useful information.

The last line in blue is the one that actually sets the affinity, and it does so by explicitly providing a <*proclist*>, which in the case of MARCONI, where hyper-threading is disabled, corresponds to a list of cores, specified in terms of their logical numbering. Note that it is typical for Intel NUMA nodes to have different logical and physical numberings for the processors, as can be seen from the schematics of a MARCONI production node in Fig. 6. By specifying the <*proclist*> to cover all cores within the node in their increasing sequential order, this tells the Intel MPI library to place the MPI tasks such that their rank matches the logical core number. In other words, place the MPI tasks requested using a compact affinity, as depicted on the left side of Fig. 1, except that now the 12 processes are 12 MPI tasks.

To obtain the scatter affinity shown on the right side of the same picture, all that is needed is to change the <*proclist*> accordingly. Namely, by replacing the corresponding line in the script Tab. II with the following one.

```
export I_MPI_PIN_PROCESSOR_LIST=0,18,1,19,2,20,3,21,4,22,5,23
```

This ensures that both sockets of the node are used if more than one MPI task is requested, whereas in the compact affinity case, this only happens if more that 18 MPI tasks are used. An alternative way to enforce the property of using both sockets, while still keeping consecutive task ranks as close to each other as possible, is to use instead the following.

```
export I_MPI_PIN_PROCESSOR_LIST=0-5,18-23
```

We shall call this *bunch* affinity, following the nomenclature of the Intel MPI library, whereby the processes (*domains*) are mapped consecutively as closely as possible but at the same time distributed over both sockets in a balanced way. This kind of affinity is illustrated in Fig. 2. However, it must be pointed out that this particular configuration only works as intended for 12 MPI tasks. With less than 12 tasks, more of them will be placed on the first socket than on the second, creating an asymmetric distribution; with more than 12, the extra tasks will start to be placed in the already occupied cores following the same pattern specified in the <*proclist*=0-5,18-23> in a round-robin fashion, using the resources in a sub-optimal manner, and further potentially creating load-balance issues. Therefore, for the *bunch* affinity to work appropriately with different numbers of tasks, the <*proclist*> must be adapted accordingly.

Finally, it is worth mentioning that there is an alternative way to easily obtain the *scatter* and *bunch* affinities invoking the same Intel MPI library environmental variable,

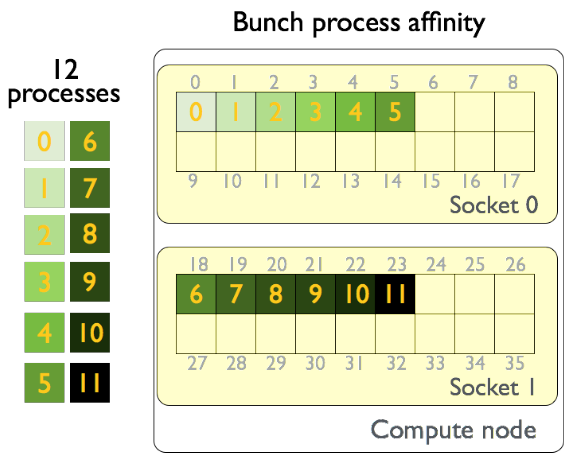

FIG. 2. Schematics of 12 processes distributed with bunch affinity on a MARCONI node. Note that now consecutive processes stay close to each other within each socket.

without having to specify explicitly the <*proclist*>, as was done before. Instead, a predefined <*procset*>:<*map*> pair can be invoked. So, for a scatter affinity one can use

```
export I_MPI_PIN_PROCESSOR_LIST=allcores:map=scatter
```

whereas for a bunch affinity one would use

```
export I_MPI_PIN_PROCESSOR_LIST=allcores:map=bunch
```

corresponding to Figs. 1(right) and 2, respectively, when 12 MPI tasks are requested.

It should be noted that there is no `I_MPI_PIN_PROCESSOR_LIST` configuration implicitly exported by the PBS scheduler, nor is `I_MPI_PIN` set, so, to be on the safe side, the recommended practice is to set them explicitly in the batch script, as done in the examples provided.

1. Reference

The available options for `I_MPI_PIN_PROCESSOR_LIST=` <*procset*>:<*map*> are

- <*procset*>=`all, allcores, allsocks`
- <*map*>=`bunch, scatter, spread`.

For more details consult the reference guide [4].

C. Hybrid task/thread affinity

So far we have been dealing with pure OpenMP or pure MPI codes, whose basic pinning strategies were discussed in Secs. I A and I B, respectively. Now we move to the hybrid case of mixing both parallelization paradigms. The Intel MPI library provides alternative environment variables that allow the interoperability with the OpenMP API. Namely, `I_MPI_PIN_DOMAIN` defines the concept of *domains*, which are non-overlapping subsets of logical processors within a node, with each one mapped to a single MPI task that can have OpenMP threads below. Such domains can then be pinned with a given affinity. The threads spawned inside them can also be pinned using `KMP_AFFINITY`. It is noteworthy that, if the `I_MPI_PIN_DOMAIN` environment variable is defined, then the `I_MPI_PIN_PROCESSOR_LIST` setting is ignored. In other words, the Intel MPI process affinity environment variables that are interoperable with OpenMP supersede the ones described in the previous section. Actually, they can even be used as an alternative for pure MPI code, provided that, either the domain environment variable is set to `I_MPI_PIN_DOMAIN=core`, or that it is set to `I_MPI_PIN_DOMAIN=omp` with the number of OpenMP threads set to `OMP_NUM_THREADS=1`, even if the code being ran does not use OpenMP threads whatsoever. Both alternatives essentially say that each MPI *domain* maps to one core only. The reason we dedicated the previous section to the case of pure MPI codes is that the concepts and subtleties just described can simply be ignored by their users. However, for a user/developer of a hybrid code, even if running it just in pure MPI or OpenMP mode, it makes sense to have a single script that works for all situations (pure OpenMP, pure MPI or hybrid), rather than having three different scripts to contemplate each separate case. The purpose of this section is to lay down the basic rules to devise such a baseline "hybrid" script.

Let's start with the case of a pure MPI code and mimic the last two affinity configurations of Sec. I B. By replacing the line `I_MPI_PIN_PROCESSOR_LIST=` <*procset*>:<*map*>, in the script Tab. II with

```
export I_MPI_PIN_DOMAIN=omp          |    export I_MPI_PIN_DOMAIN=core
export I_MPI_PIN_ORDER=scatter       |    export I_MPI_PIN_ORDER=scatter
export OMP_NUM_THREADS=1             |    <leave OMP_NUM_THREADS unset>
```

or with

```
export I_MPI_PIN_DOMAIN=omp        |    export I_MPI_PIN_DOMAIN=core
export I_MPI_PIN_ORDER=bunch       |    export I_MPI_PIN_ORDER=bunch
export OMP_NUM_THREADS=1           |    <leave OMP_NUM_THREADS unset>
```

we obtain the *scatter* affinity of Fig. 1(right), or the *bunch* affinity of Fig. 2, respectively. The default value for `I_MPI_PIN_ORDER` is *compact*. Note that in the above script lines on the right side, not explicitly setting the number of threads is equivalent to setting them to `OMP_NUM_THREADS=ncpus`, which is the default value. However, this does not affect the size of the domain, which in this case is set independently of the number of threads. Note also that besides the *bunch* ordering, there is also the *spread* ordering available, but on MARCONI's dual-socket node architecture (see Fig. 6), they are in practice almost equivalent.

To generalize this to an hybrid OpenMP/MPI code is pretty straight forward and the easiest way to show this is to provide an example of the corresponding MARCONI batch script, just like we did in previous sections. Let's assume that we want to run four MPI tasks, each spawning three OpenMP threads, for a total of 12 processes. The script in Tab. III does so by creating four *domains*, each with the size of the three OpenMP threads, and then pinning them with a *compact* (or *bunch*) affinity. Inside each *domain*, the threads are pinned with *compact* affinity, for the resulting hybrid process pinning affinity illustrated on the left (or right) side of Fig. 3.

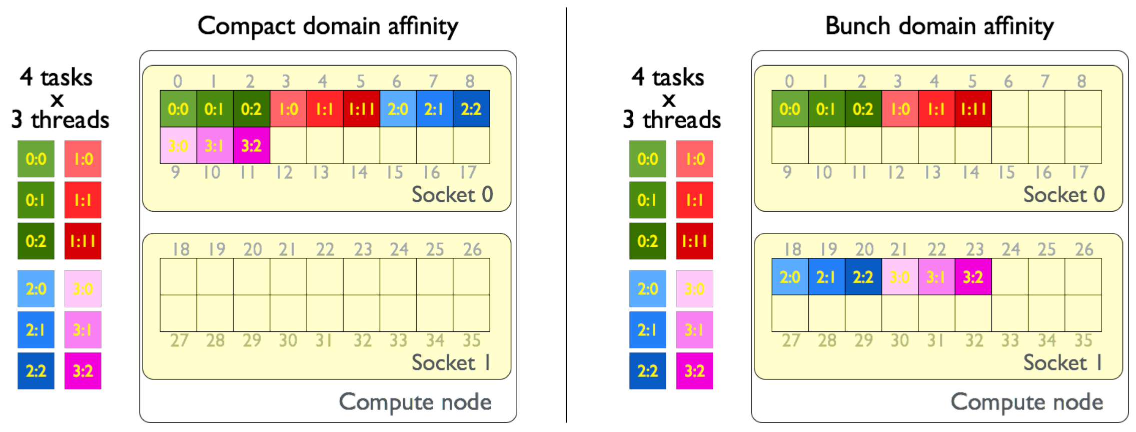

FIG. 3. Schematics of four domains (MPI tasks) each with three OpenMP threads, for a total of 12 processes, distributed with *compact* (left) and *bunch* (right) affinities on a MARCONI node. The numbers inside the coloured squares represent <*task rank*>:<*thread index*>

It is worth noting the reason why we always used `KMP_AFFINITY=compact` with the hybrid script examples. On MARCONI, since the first level of non-shared resources is the L3 cache, which is local to each socket, the only way to obtain a scattered affinity for the

```
#!/bin/bash
#$ -cwd
#PBS -N hyb_4_3
#PBS -l select=1:ncpus=36:mpiprocs=4:mem=118GB
#PBS -l walltime=01:00:00
#PBS -q xfuaprod
#PBS -A <project>
#PBS -j eo
#PBS -m abe
#PBS -M <email.address>@ipp.mpg.de

module load <needed modules>
export I_MPI_DEBUG=5
export I_MPI_PIN_DOMAIN=omp
export I_MPI_PIN_ORDER=compact    <OR>    export I_MPI_PIN_ORDER=bunch
export OMP_NUM_THREADS=3
export KMP_AFFINITY=verbose,compact
BIN=a.out
CMDLOPT=''
mpirun -prepend-rank ./$BIN $CMDLOPT
```

TABLE III. Basic script used to run the hybrid code `a.out` with four domains (MPI tasks) each with three OpenMP threads, for a total of 12 processes, distributed with `compact` <OR> `bunch` affinities on a MARCONI node.

threads is if only a single *domain* with more than 18 threads is used. So, in practice, as soon as two or more *domains* (MPI tasks) per node are requested, `KMP_AFFINITY=scatter` yields in practice a *compact* affinity, because each *domain* fits inside a single socket, i.e. there are no non-shared resource at the *domain* level.

To finalize this section, we refer to the most flexible way to specify the affinity of the MPI domains. Rather than relying on the `I_MPI_PIN_ORDER` presets used before, one can directly specify a mask domain, pretty much like we did before in Sec. I B for the pure MPI code using the environment variable `I_MPI_PIN_PROCESSOR_LIST`. In the hybrid case this is however slightly more complicated because, instead of providing a list of cores, we need to provide instead a hexadecimal mask describing each domain requested. So if, like before, four MPI tasks are used each with three OpenMP threads, this means we need four *domains* and so, we need to provide four masks, with

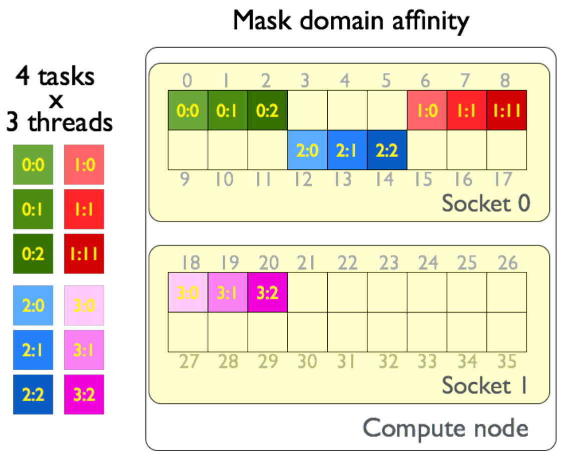

FIG. 4. Schematics of four domains (MPI tasks) each with three OpenMP threads, for a total of 12 processes, distributed with a tailored affinity on a MARCONI node.

the syntax `I_MPI_PIN_DOMAIN=[mask`$_0$`,mask`$_1$`,mask`$_2$`,mask`$_3$`]`. In binary format, each core in a MARCONI node is represented by a sequence of 35 zeros and a one in the position corresponding to that core's logical number. For instance, core 0 has the binary mask "`000000000000000000000000000000000001`", core 1 has the binary mask "`000000000000000000000000000000000010`", all the way to core 35, whose binary mask is "`100000000000000000000000000000000000`". If we want to specify the four domains represented in Fig. 4 in terms of their binary masks, these would be:

- domain rank 0 (green): cores {0,1,2}

  binary mask: "`000000000000000000000000000000000111`"

- domain rank 1 (red): cores {6,7,8}

  binary mask: "`000000000000000000000000000111000000`"

- domain rank 1 (blue): cores {12,13,14}

  binary mask: "`000000000000000000000111000000000000`"

- domain rank 1 (magenta): cores {18,19,20}

  binary mask: "`000000000000000111000000000000000000`".

These four masks in binary basis are quite long. It is much more convenient to represent them in a hexadecimal basis, in which case they take the more condensed representation `[7,1C0,7000,1C000]`. This is precisely the format required in the script. Hence the batch script corresponding to the process affinity of Fig. 4 is obtained by replacing the blue coloured lines in Tab. III with the following ones

```
export I_MPI_DEBUG=5
export I_MPI_PIN_DOMAIN=[7,1c0,7000,1c0000]
export OMP_NUM_THREADS=3
export KMP_AFFINITY=verbose,compact
```

1. Reference

The available options for `I_MPI_DOMAIN=`<*size*> are

- <*size*>=`omp, auto, <n>`

and for `I_MPI_PIN_ORDER=`<*order*> are

- <*order*>=`compact, bunch, scatter, spread, range`.

For more details consult the reference guide [5].

II. SINGLE-NODE SCALING RESULTS IN MARCONI

We already noted in the introduction that, under NUMA architectures, a processor (core) can access its own local memory (within the socket) faster than non-local memory (across sockets). Therefore, as also emphasized there, pinning affinity strategies can be of importance performance-wise. However, the choice of the best process affinity obviously depends on the code being used. If an algorithm implies strong local data dependencies and high arithmetic intensities (CPU-bound), then using a compact affinity to have the neighbouring processes mapped as closely as possible in terms of the hardware cores is probably advantageous. On the other hand, if access to main memory is the bottleneck of the algorithm (memory-bound), then having the processes distributed over both sockets with a *scatter* or *bunch* affinity is most probably a better solution, since within one socket, only half the memory bandwidth of the full node can be achieved.

Here we shall test these concepts in practice by using the hybrid MPI/OpenMP REFMULXP code [1], whose underlying memory-bound algorithm makes it well suited to illustrate the effect of the different process affinities on the overall performance. At the same time, it provides a real-world example with an actual finite-differences-time domain (FDTD) wave-propagation production code, which is used for physics studies. Examples of similar analysis using the standard STREAM benchmark [2] can be found elsewhere.

Being that REFMULXp can be compiled as a pure threaded code using OpenMP, as a pure MPI code or using both paradigms for a hybrid parallel configuration, it allows us

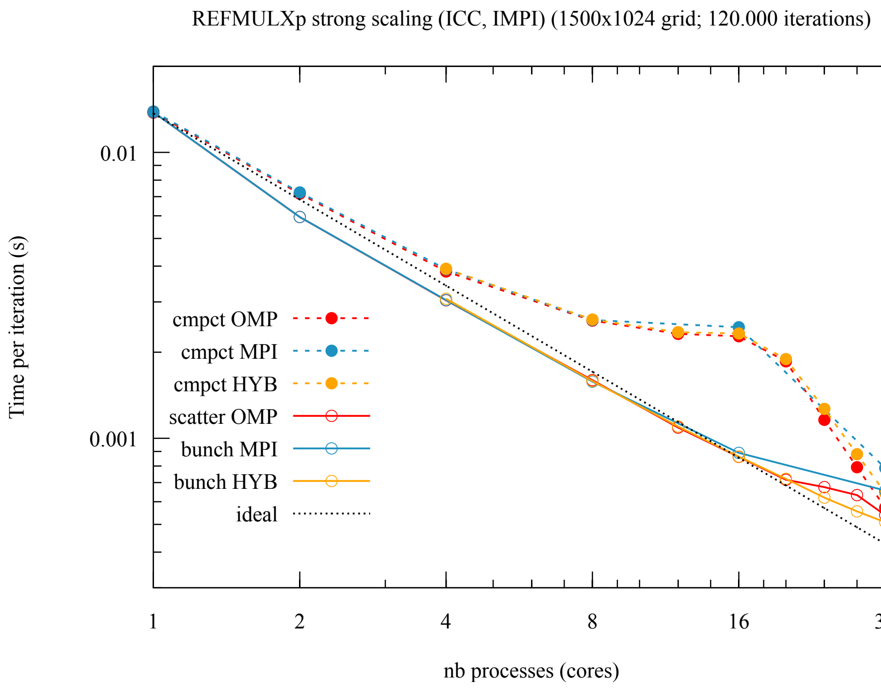

FIG. 5. MARCONI intra-node strong scaling results of REFMULXp using different parallelization paradigms (pure OpenMP, pure MPI and hybrid MPI/OpenMP) with several process binding affinities. The hybrid configuration has four MPI tasks and varies the number of threads, which corresponds to the examples given in Figs. 3–4 when three threads are used.

to test all three cases separately and then compare them. This is what is shown in Fig. 5. A detailed discussion of the results, for which better statistics than the “single run per data point” used here would be required, goes beyond the scope of this writeup. Instead, what is intended is to highlight the influence of the pinning strategies on the performance of the code. It is clear from the plot that, as expected, when the processes are distributed compactly on the hardware cores, the time to run an REFMULXp simulation is higher than when they are evenly distributed across sockets. The two main reasons which justify this difference are that, to access the full main memory bandwidth both sockets need to be used, and that the main memory bandwidth increases nonlinearly with the number of cores used, with roughly 8 cores per socket being enough to saturate it within each socket for the algorithm at hand. In light of this, it is easy to understand that a scatter or bunch process affinity, which always distributes the work load evenly among both sockets, provides a higher bandwidth per core value than a compact affinity, for which the first socket necessarily has a heavier load, except when the full node is used. The maximum difference is therefore yielded when exactly half a node is used (18 cores).

Another concept whose effect, although less relevant, still seems to be measurable on this algorithm, is the data locality. Comparing the compact and scatter affinities for the pure threaded case when the whole node is used (rightmost point in the red curves), it seems to be advantageous to have consecutive threads placed in consecutive cores, rather than scattering them across sockets (recall the difference between Figs. 1(right) and 2).

III. SUMMARY AND OUTLOOK

This writeup was made to provide practical guidelines on how to use the process pinning options within the Intel MPI library together with the PBS batch queueing system available on the MARCONI supercomputer. Examples of scripts for pure OpenMP, pure MPI and hybrid OpenMP/MPI cases were given for process affinities which are typically relevant for the codes used within the community. Their impact on code performance was then illustrated with strong scaling results from an FDTD production code on MARCONI. For more detailed information on the Intel MPI library and OpenMP environment variables, the reader is referred to the corresponding technical reference guides listed in the bibliography.

IV. ACKNOWLEDGEMENTS

This work has been carried out within the framework of the EUROfusion Consortium and has received funding from the Euratom research and training programme 2014-2018 under grant agreement No 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

[1] F. da Silva, M. Campos Pinto, B. Desprs and S. Heuraux, *J. Comp. Physics*, **295**: 24-45 (2015).

[2] John D. McCalpin "STREAM: Sustainable Memory Bandwidth in High Performance Computers" *University of Virginia* (1991-2007)

[3] `https://software.intel.com/en-us/node/528816`

[4] `https://software.intel.com/en-us/node/528818`

[5] `https://software.intel.com/en-us/node/528819`

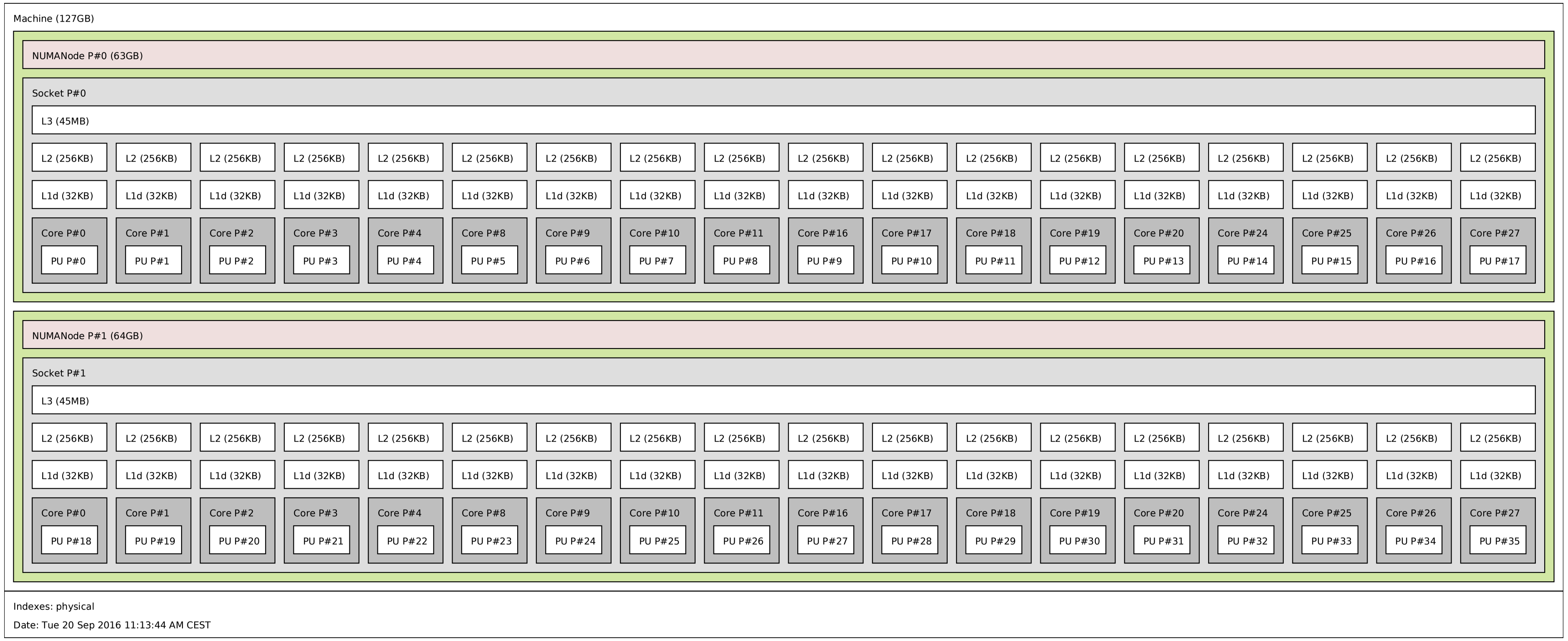

FIG. 6. Schematics of a MARCONI node obtained using the hwloc software package, by invoking the command:
"`lstopo --no-io --no-icaches lstopo_marconi`".
Note the difference between the (somewhat odd) physical numbering of the cores (`Core P#`), compared to the logical numbering of the processing units (`PU P#`) below.