Intel® Xeon® 6 with P-Cores Configuration and Tuning Guide for HPC Applications

December 2025

Revision 1.2

Document Number: 858491

Contents

Revision History

Revision Number	Description	Date
1.0	Initial document	June 2025
1.1	Updated SPACK install instructions to use oneAPI 2025.3 compilers and SPACK version 1.0.	December 2025
1.2	Updated Financial Services Industry (FSI) application recipes to use SPACK. Added recipe for QuantLib.	December 2025

1 Performance and Optimization Disclaimer

Performance results are based on testing by Intel and may not reflect all publicly available security updates. See configuration disclosure for details. No product or component can be absolutely secure.

Intel does not control or audit third-party benchmark data or the web sites referenced in this document. You should visit the referenced web site and confirm whether referenced data is accurate.

Performance varies by use, configuration, and other factors. Learn more at www.intel.com/PerformanceIndex.

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Refer to the applicable product user and reference guides for more information regarding the specific instruction sets covered by this notice. Notice revision number: 20110804.

Intel® Advanced Vector Extensions (Intel® AVX) provides higher throughput to certain processor operations. Due to varying processor power characteristics, utilizing Intel® AVX instructions may cause a) some parts to operate at less than the rated frequency and b) some parts with Intel® Turbo Boost Technology 2.0 to not achieve any or maximum turbo frequencies. Performance varies depending on hardware, software, and system configuration and you can learn more at http://www.intel.com/go/turbo.

Intel® Hyper-Threading Technology (Intel® HT Technology) is available on select Intel processors. Requires an Intel HT Technology enabled system. Your performance varies depending on the specific hardware and software you use. Learn more by visiting Intel® Hyper-Threading Technology.

2 Preface

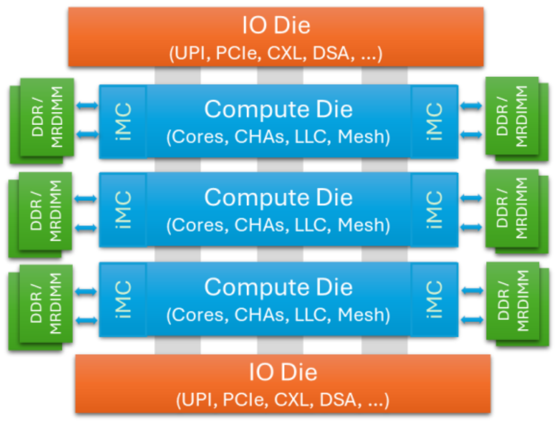
2.1 Objectives

This document contains tuning and optimization guidelines for High Performance Computing (HPC) applications running on Intel® Xeon® 6 with P-cores (codenamed Granite Rapids) processors. It includes general BIOS and Linux configuration guidelines for all HPC applications as well as build and run instructions (recipes) for a selected set of HPC applications and benchmarks.

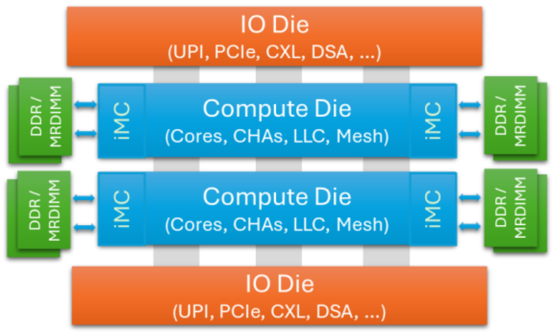
2.2 Architecture Overview

Intel® Xeon® 6 with P-Cores (codenamed Granite Rapids) is the leadership Intel® Xeon® multi-core server processor. Intel® Xeon® 6900-series with P-cores support up to 52 bits of physical address space and 57 bits of virtual address space while the Intel® Xeon® 6700-series with P-cores support up to 52 bits of physical address space and 48 bits of virtual address space and are designed for a platform consisting of at least one processor.

As shown below, the 6900-series processor consists of 3 compute dies and the 6700-series processor consists of 2 compute dies (the 6500-series consists of 1 compute die for low core counts). A single compute die includes compute cores with L1 and L2 caches, a part of L3 cache, and Integrated Memory Controllers (IMC).



6900-series Processor Block Diagram



6700-series Processor Block Diagram

These processors also support Compute Express Link (CXL) 2.0, and an Integrated I/O (IIO). Most processor types support up to 96 lanes of PCI Express* 5.0 capable of 32GT/s, and 8 lanes of DMI while the R1S processor types support up to 136 lanes of PCI Express 5.0. Each IMC supports up to 2 channels of DDR5 DIMMs with up to 2 DIMMs per channel. Intel® Xeon® 6900/6700-series with P-Cores also support MRDIMMs (Multiplexed Rank DIMM) at 1 DIMM per channel.

The processor is comprised of multiple IO interface and CHA/core modules, and the modules are connected to a mesh interconnect which consists of horizontal and vertical interconnects, each interconnect consisting of bi-directional channels. Different SKUs of the processor may have a different combination of modules.

The Intel® Xeon® 6 with P-cores incorporate the Redwood Cove core, building upon the Golden Cove and Raptor Cove core utilized in the 4th and 5th Gen. The core implements a mid-level cache (MLC) of 2MB capacity.

The Input/Output (I/O) architecture has improved since the 5th gen microarchitecture. There are between 88 and 136 total lanes of PCI Express* (PCIe*) Gen5 vs. the 80 lanes of PCIe Gen5 available in the 4th Gen Intel® Xeon® family.

More product information and performance results of Intel® Xeon® 6 with P-cores can be found here.

2.3 Additional Tuning Resources

Intel publishes tuning guides for a variety of server workloads. These guides are designed for people who are already familiar with the workload but are looking for system settings that will improve performance.

Tuning guides for databases, AI frameworks, HPC clusters, and network workloads systems can be found online at this link: https://software.intel.com/content/www/us/en/develop/articles/xeon-performance-tuning-and-solution-guides.html.

Optimized libraries and toolkits, developed by Intel, are available for AI, ML, and DL frameworks and numerous programming environments that can improve performance of various workloads. Here are links to more information:

Optimized Libraries and Tools
Intel Optimized AI Software
Intel Distribution for Python*, including NumPy, SciPy, sckit-learn*
Popular Deep Learning Frameworks including PyTorch, TensorFlow*, MXNet*,v PaddlePaddle*, Caffe*
Tools, libraries, and SDKs including Intel AI Analytics and Intel Distribution of OpenVINO* toolkits oneAPI – a set of compilers, libraries, and porting, analysis and debugger tools. Intel® Resource Director Technology (Intel® RDT) a framework for monitoring and allocation of shared cache and memory resources.

3 Hardware Configuration

3.1 Objectives

This chapter describes hardware selection and configuration options set via BIOS.

3.2 DRAM Selection and Configuration

Intel® Xeon® 6900-series with P-cores Xeon supports 6400 MT/s DDR5 and 8800 MT/s MRDIMMs. Intel® Xeon® 6700-series with P-cores supports 6400 MT/s DDR5 and 8000 MT/s MRDIMMS.

Dual rank memory is recommended for DDR5 to achieve the best possible memory bandwidth.

Note: 32GB DDR5 modules provide slightly higher write bandwidth than larger capacity modules due to the way error correction code (ECC) bits are stored and accessed in DDR5 modules.

Note: MRDIMMs offer the highest possible bandwidth for both 6900 and 6700-series systems. At the same time, MRDIMMs offer lower latency than DDR5 in Granite Rapids systems (regardless of bandwidth utilization).

For best performance, make sure that all memory modules have the same specifications and are from the same manufacturer.

3.2.1 How to Verify

Verify DRAM type and configuration information using `dmidecode` (Section 4.3.6) and/or `lsmem` (Section 4.3.6). For instance, `dmidecode` gives capacity, speed (MT/s), number of ranks, manufacturer and part/serial number of each DRAM module installed in the system.

Verify DDR performance using Intel Memory Latency Checker (Section 5.4), STREAM benchmark (Section 6.4), and HPCG benchmark (Section 6.6).

3.3 BIOS Settings

This section provides BIOS options and settings that are relevant for HPC application performance. Note that BIOS options and default settings provided in a particular BIOS are dependent on the BIOS provider.

3.3.1 Sub-NUMA Clustering (SNC)

The 6900-series processor consists of 3 compute dies and the 6700-series processor consists of 2 compute dies. A single compute die includes compute cores (each with a private L1 and L2 cache), a part of L3 cache, and Integrated Memory Controllers (IMC) supporting 4 memory channels.

It is possible to make each compute die appear as a separate physical NUMA node with Sub-NUMA Clustering (SNC) mode, which can be selected at boot time using the BIOS menu. On Birch Stream platform, SNC can be enabled/disabled via menu:

EDKII -> Socket Configuration -> Uncore Configuration -> Uncore General Configuration -> SNC

Intel® Xeon® 6900-series with P-cores support 3 NUMA nodes per processor and 6700-series with P-cores support 2 NUMA nodes per processor. A typical system with 2 sockets (processors) supports a total of 6 NUMA nodes with 6900-series and 4 NUMA nodes with 6700-series.

Since SNC mode exposes each compute die as a separate NUMA node, it allows software running on the cores of a given NUMA node to access memory attached to the same compute die by default. This reduces memory access latency and improves memory bandwidth that can be consumed by cores.

SNC requires NUMA aware software to take advantage of multiple SNC nodes. Since most HPC software is NUMA aware (e.g., using MPI processes, MPI+OpenMP, or ensembles) HPC applications can readily obtain the benefit of SNC. For instance, an MPI (or MPI+OpenMP) application should use at least one MPI process (rank) on each NUMA node. Similarly, an OpenMP-only application can use first touch policy to make sure that the data used by a given thread is allocated on the same NUMA node (e.g., by making sure that data items accessed by an OpenMP thread are initialized by the same OpenMP thread).

To verify SNC settings, use `numactl` utility (Section 4.3.2) with command '`numactl -H`'. It should show 6 NUMA nodes on a two-socket 6900-series system and 4 NUMA nodes on a two-socket 6700-series system. Further, check amount of memory available on each NUMA node to verify that memory installation is symmetrical.

To verify that an application is NUMA aware, use `numastat` utility (Section 4.3.3). If an application is completely NUMA aware, '`numa miss`' counter should not increase across a run (i.e., compare `numa_miss` values before and after an application execution).

3.3.2 Turbo Settings

Turbo should be enabled in BIOS for best performance. Tubo enables the processor to operate at frequencies higher than the base frequency, up to the TDP (Thermal Design Power) limit.

Use `turbostat` (Section 4.3.4) utility (e.g., `turbostat -qS`) to verify the actual operating frequency and power consumption of the processor.

Linux utility `lscpu` (Section 4.3.1) can show the maximum and minimum Turbo frequencies.

Note: In Birch Stream BIOS, disabling *Energy Efficient Turbo* is recommended for maximum performance.

3.3.3 Fan settings and CPU Temperature

Verify CPU package temperature using `turbostat` (Section 4.3.4) while running a CPU intensive benchmark such as High Performance LINPACK (HPL), which is described in Section 6.5.

Lower CPU temperatures allow CPUs to operate more efficiently since leakage power consumption increases with higher operating temperatures.

Fan setting in BIOS (if available) should be set to high performance with duty cycle (sometimes named as 'PWM Offset') set to a high value (90% or above) to achieve the best air circulation. On Birch Stream platforms, PWM Offset can be set via the BIOS menu:

EDKII -> Platform Configuration -> Miscellaneous Configuration -> Fan PWM offset

Verify using '`dmesg -T`' (ref) to make sure that no prochot events (thermal throttling events) are generated while running a CPU intensive application like HPL (Section 6.5). Further, monitor processor temperature using `turbostat` utility while running HPL.

3.3.4 Latency Optimized Mode

Latency Optimized Mode allows the processor to run its uncore (L3 caches, memory controllers and IO die) at a high frequency even when the uncore is not heavily utilized. This lowers the latency of memory and IO operations (including UPI transfers) for applications that do not heavily utilize the uncore. This can benefit MPI communication and memory latency bound applications. However, this increases power consumption for those underutilized scenarios.

The Latency Optimized Mode can be enabled/disabled using a BIOS option at boot or using the following script after the system is booted up (needs root access): https://github.com/intel/pcm/blob/master/scripts/bhs-power-mode.sh

For Birch Stream BIOS, Latency Optimized Mode can be set with menu:

EDKII -> Socket Config -> Advanced Power Mgmt config -> CPU - Advanced PM tuning -> Latency Optimized mode

Note that setting Latency Optimized Mode via the above script gives more flexibility since it can be enabled/disabled as needed for each application execution separately.

To verify that the Latency Optimized Mode is applied, the uncore frequency of each die (including IO dies) can be checked using the following command with root privileges.

```
cat /sys/devices/system/cpu/intel_uncore_frequency/uncore0*/current_freq_khz
```

When the Latency Optimized Mode is applied, the uncore frequency of all dies should be at maximum even when the system is idle or lightly used. Note that IO dies have different maximum frequencies (e.g., 2500 MHz) compared to the uncore frequencies of compute dies (e.g., 2200 MHz).

3.4 Network Configuration

In a cluster environment, installing a network adaptor per socket allows higher bandwidth and lower latency for network access. Latency Optimized Mode (Section 3.3.4) can lower MPI communication latency of some applications.

Network performance can be verified using Intel® MPI Benchmarks (Section 5.2) and application performance can be further tuned using utilities provided with the MPI Library (Section 5.2).

4 Linux System Configuration

4.1 Objectives

This chapter describes useful Linux utilities and Linux options that can be used to improve performance of HPC applications.

4.2 Linux Distribution and Kernel

The latest stable Linux kernel is recommended for the best performance. At a minimum, a 6.x kernel should be used. The recipes in this guide were tested using kernel version 6.8 or later.

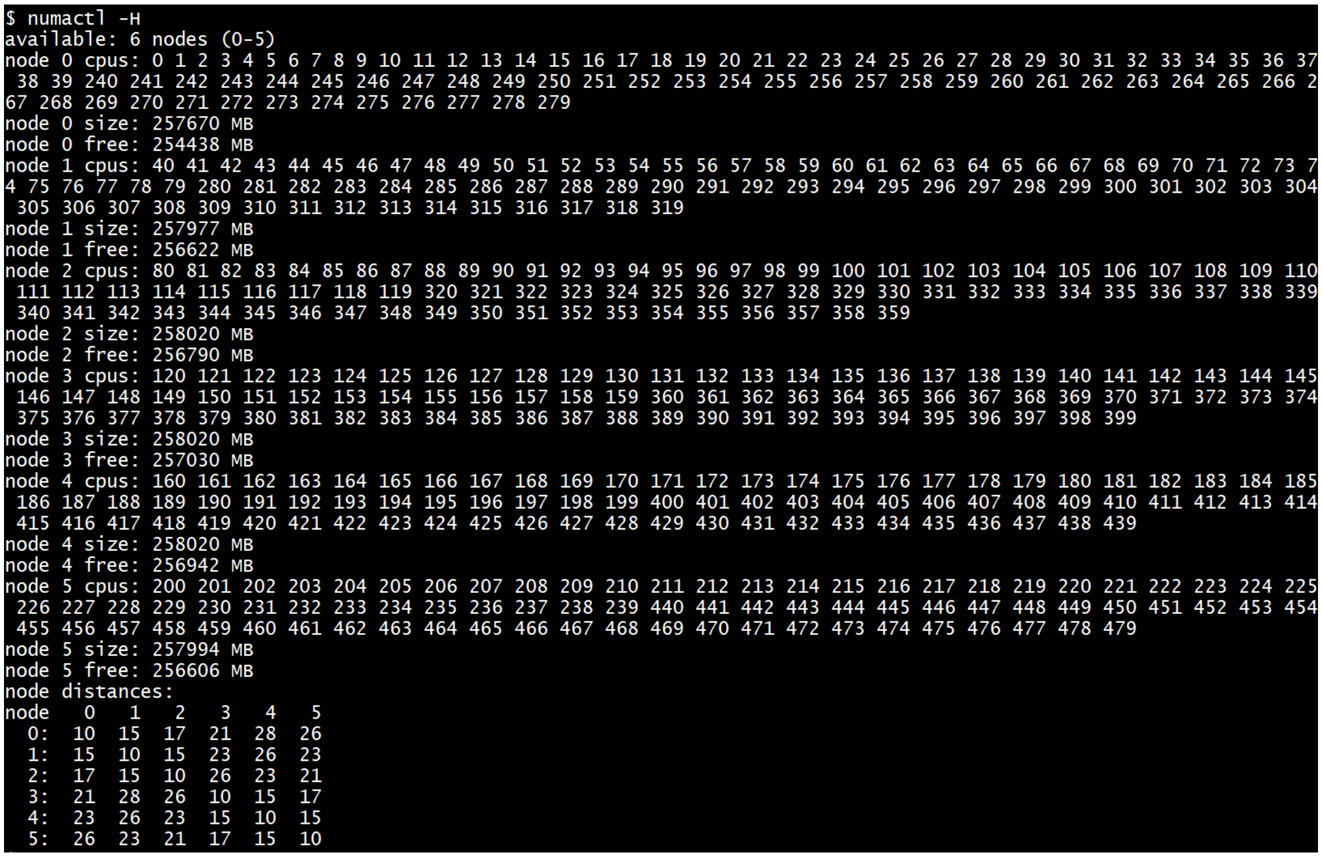
4.3 Useful Linux Utilities

4.3.1 lscpu

This standard Linux utility shows high-level system configuration details, including the NUMA nodes, core counts, base frequency, turbo frequency, cache sizes, and CPU flags (features).

4.3.2 numactl

The Linux utility numactl is often used for both observing the NUMA configuration of the system and executing applications on specific NUMA nodes. For observing the NUMA configuration of a system, use `numactl -H`. The following figure shows the output of `numactl -H` showing the number of NUMA nodes, CPU cores, and memory capacity on each NUMA node, followed by a matrix describing the distance of each node from any other node. See `man numactl` for more information.

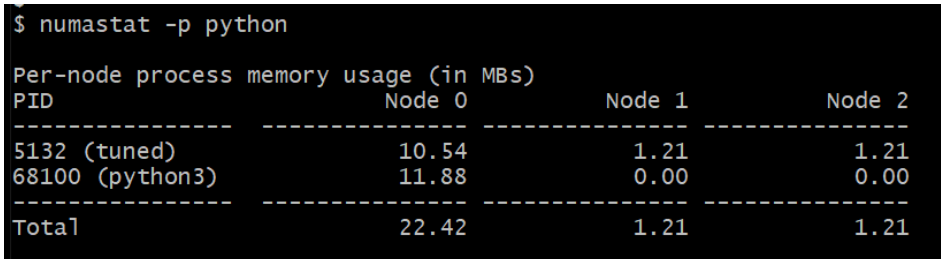


```
$ numactl -H
available: 6 nodes (0-5)
node 0 cpus: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37
 38 39 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 2
67 268 269 270 271 272 273 274 275 276 277 278 279
node 0 size: 257670 MB
node 0 free: 254438 MB
node 1 cpus: 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 7
4 75 76 77 78 79 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304
 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319
node 1 size: 257977 MB
node 1 free: 256622 MB
node 2 cpus: 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110
 111 112 113 114 115 116 117 118 119 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339
 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359
node 2 size: 258020 MB
node 2 free: 256790 MB
node 3 cpus: 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145
 146 147 148 149 150 151 152 153 154 155 156 157 158 159 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374
 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399
node 3 size: 258020 MB
node 3 free: 257030 MB
node 4 cpus: 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185
 186 187 188 189 190 191 192 193 194 195 196 197 198 199 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414
 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439
node 4 size: 258020 MB
node 4 free: 256942 MB
node 5 cpus: 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225
 226 227 228 229 230 231 232 233 234 235 236 237 238 239 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454
 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479
node 5 size: 257994 MB
node 5 free: 256606 MB
node distances:
node   0   1   2   3   4   5
  0:  10  15  17  21  28  26
  1:  15  10  15  23  26  23
  2:  17  15  10  26  23  21
  3:  21  28  26  10  15  17
  4:  23  26  23  15  10  15
  5:  26  23  21  17  15  10
```

4.3.3 numastat

The Linux utility `numastat` provides various statistics about NUMA memory usage. In particular, the following commands are useful (see `man numastat` for more details).

- `numastat -p <process_name>` gives memory usage of a given process as shown below:

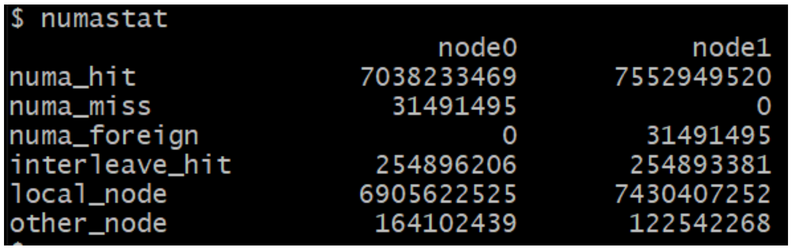


```
$ numastat -p python

Per-node process memory usage (in MBs)
PID                         Node 0          Node 1          Node 2
----------------   ---------------  --------------- ---------------
5132 (tuned)                 10.54            1.21            1.21
68100 (python3)              11.88            0.00            0.00
----------------   ---------------  --------------- ---------------
Total                        22.42            1.21            1.21
```

- `numastat -m` shows the memory usage information of an entire system

- `numastat` (without arguments) shows NUMA hits and misses (cumulative from boot). This is useful in identifying page allocations that lead to NUMA node crossings (`numa_miss`) that could cause unexpected performance degradations. Since these statistics are cumulative from boot, it is necessary to run numastat before and after each application execution to see whether a given execution encountered NUMA misses.

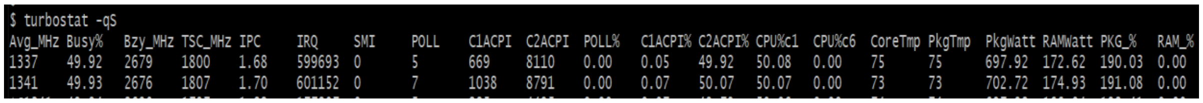


```
$ numastat
                           node0           node1
numa_hit              7038233469      7552949520
numa_miss               31491495               0
numa_foreign                   0        31491495
interleave_hit         254896206       254893381
local_node            6905622525      7430407252
other_node             164102439       122542268
```

4.3.4 turbostat

The `turbostat` utility can be used to examine the power, frequency, and temperature of x86 architecture processors when executed as root (or as a setuid binary). Turbostat can be useful in identifying system cooling issues. For instance, the following shows the system's power, frequency, and temperature:

- `turbostat -qS                     # average for both CPUs`
- `turbostat -qS --cpu package       # separately for each CPU`



```
$ turbostat -qS
Avg_MHz Busy%   Bzy_MHz TSC_MHz IPC     IRQ     SMI     POLL    C1ACPI  C2ACPI  POLL%   C1ACPI% C2ACPI% CPU%c1  CPU%c6  CoreTmp PkgTmp  PkgWatt RAMWatt PKG_%   RAM_%
1337    49.92   2679    1800    1.68    599693  0       5       669     8110    0.00    0.05    49.92   50.08   0.00    75      75      697.92  172.62  190.03  0.00
1341    49.93   2676    1807    1.70    601152  0       7       1038    8791    0.00    0.07    50.07   50.07   0.00    73      73      702.72  174.93  191.08  0.00
```

4.3.5 htop

The `htop` utility is a tool similar to standard Linux `top` utility; however, it shows individual CPU core/thread usage using a visual format. This is helpful in identifying NUMA usage and MPI rank or OpenMP thread placement. In addition, it shows the memory usage of the system.

4.3.6 dmidecode, lshw, and lsmem

These Linux utilities can be used (with root privileges) to inspect hardware components installed, including DDR modules.

4.3.7 lstopo

The `lstopo` utility is part of the `hwloc` library, which can be installed as a package using a standard package manager (e.g., `dnf install hwloc`). The `lstopo` utility (or `lstopo-no-graphics`) shows the hardware topology of a system.

4.4 Linux Configuration Options

This section describes various recommended Linux configuration options.

4.4.1 Transparent Huge Pages (THP)

Most HPC applications benefit from THP. It can be enabled using the following command:

```
echo always >/sys/kernel/mm/transparent_hugepage/enabled
```

4.4.2 Watchdog Timer

Linux kernels older than 6.12 do not contain a patch to fix a false alarm arising from clocksource (TSC) sanity checks on systems that have more than 4 NUMA nodes. This could lead to clocksource falling back to slow HPET timer, which can severely degrade performance of some applications. If a kernel version earlier than 6.12 is used, the following kernel boot option must be used to avoid this unpredictable degradation: `tsc=nowatchdog`

4.4.3 Frequency Governor

If using intel_pstate driver (default setting), set the frequency scaling governor to ‘performance’:

```
/bin/sh -c "for CPU in /sys/devices/system/cpu/cpu[0-9]*; do echo
'performance' > \"$CPU/cpufreq/scaling_governor\"; done"
```

4.4.4 Sub-NUMA Clustering (SNC)

As described in Section (Section 3.3.1), SNC should be enabled in BIOS for NUMA aware applications. No additional steps in Linux are necessary to enable SNC. Verify proper SNC configuration using ‘`numactl -H`’.

4.4.5 Automatic NUMA Balancing

Automatic NUMA balancing is usually enabled on most Linux systems by default, which can lead to automatic migration of pages from one NUMA node to another. Such automatic migration could cause unexpected behavior in NUMA optimized applications. If automatic migration of pages is not desired, it could be disabled manually. The current status can be observed using:

```
cat /proc/sys/kernel/numa_balancing
```

4.4.6 Swap Space

In general, swap space should be disabled for high performance applications since swapping can severely (and unexpectedly) degrade performance. Swap space can be permanently disabled by commenting out the swap entry in `/etc/fstab` and rebooting. Swap can be also disabled using `swapoff` command (but it does not persist across rebooting). See '`man swapoff`' for more details.

4.4.7 GeoPM

GEOPM (Global Extensible Open Power Manager) is a software framework that enables users and administrators to safely and securely modify hardware settings for the duration of a process session. This unique capability allows for dynamic, user-driven power and performance tuning—enabling energy efficiency and workload optimization that is not possible with traditional system tools.

GEOPM is deployed in production on large-scale systems (e.g., Aurora) and is used by HPC centers and cloud providers to help users and administrators meet power, performance, and sustainability goals.

In particular, for memory bound HPC applications, GeoPM can be useful for reducing CPU package power to reduce system power consumption with minimal impact on performance. You can find more information about GeoPM here and a specific use case of power management for memory bound applications here.

5 Compilers, Libraries, Middleware, and Tools

5.1 Intel® OneAPI Toolkits

Intel® compilers and development libraries are included in Intel® oneAPI toolkits available from here. Both the base toolkit and HPC toolkit are recommended.

5.2 Intel® MPI

Intel® MPI Library (Link) is included in the oneAPI HPC toolkit.

Please verify MPI performance of systems using Intel® MPI Benchmarks (Link). Intel® MPI Library also provides several tuning utilities (Link) to tune performance. Detailed MPI communication statistics of an application can be obtained with APS (Section 5.5).

5.3 Intel® oneMKL

Intel® oneAPI Math Kernel Library (Link) is included in the Intel® oneAPI Base toolkit or can be downloaded from here.

5.4 Intel® Memory Latency Checker (MLC)

Intel® Memory Latency Checker (MLC) is a standalone tool used to measure memory latency and bandwidth on various systems. It is available from here.

MLC is an important tool for system validation. Before running HPC applications, MLC should be used to make sure a system provides expected memory bandwidth and latency. Run MLC without any arguments (i.e., `./mlc`) to show a full report. Some commonly used commands include (see `./mlc --help` for available arguments):

```
./mlc --peak_injection_bandwidth
./mlc --bandwidth_matrix
./mlc --latency_matrix
```

5.5 Intel® VTune™ and APS

Intel® VTune™ (Link) and can be used to profile and analyze application performance bottlenecks.

Intel® Application Performance Snapshot (APS, Link) provides a snapshot of application performance and especially useful for analyzing MPI communication inefficiencies.

Specifically, use the following to analyze various MPI statistics:

```
export APS_STAT_LEVEL=5
aps -r <result_dir> <app command>     # run app with aps
aps-report -t <result_dir>            # see report w/ MPI time
aps-report -mDf <result_dir>          # see report w/ message sizes
```

More information and usage instructions are available from the above links.

5.6 Intel® Performance Counter Monitor (PCM)

Intel® Performance Counter Monitor (Intel® PCM) is an application programming interface (API) and a set of tools based on the API to monitor performance and energy metrics of Intel processors. More information is available from here.

5.7 Intel® PerfSpect

Intel® PerfScpect is a command-line tool designed to analyze and optimize Linux servers and software running on them. It is available from here for download. In particular, it can be used to generate a detailed report of the current system configuration using the command:

```
./perfspect report
```

6 Industry Standard Benchmark Recipes

6.1 Objectives

This section presents recipes for basic industry standard benchmarks. These benchmarks should be used to validate the initial performance of a system before running other applications (i.e., as a “smoke test”).

6.2 Common SPACK Environment for Application Building

This section describes how to set up the common Spack environment with oneAPI compilers used in this guide for building applications from source code. If an application is built using a Spack recipe, **this environment needs to be set up** prior to starting any building from source.

Spack "Getting started" page available here.

```
# Clone spack repo (latest release tag) from GitHub
git clone -c feature.manyFiles=true --depth=1 --branch=releases/latest
https://github.com/spack/spack.git

# Source spack environment
export SPACK_ROOT=$HOME/spack       # point to your SPACK installation
source $SPACK_ROOT/share/spack/setup-env.sh

# Check the Spack version
spack --version

# Download and install the latest intel compiler (currently 2025.3.0)
# Latest oneAPI compiler available can be found here.
spack install intel-oneapi-compilers@2025.3.0

# Check if compiler was added to Spack. You should see added compiler.
# i.e., you should see "[+]  intel-oneapi-compilers@2025.3.0" in output
spack compiler list

# If the compiler was not added, please use the following command:
# spack compiler find $(spack location -i intel-oneapi-compilers@2025.3.0)

# (Optionally add system provided packages)
spack external find && spack external find -p /usr

# Done, now you can proceed with installing packages and applications
```

6.3 Common Instructions for MPI x OMP decomposition

The optimal number of MPI ranks and OpenMP threads for a given application/workload can be different on different processor configurations.

Refer to configuration details published with performance results for exact MPI x OMP decomposition used for a given result. When there is an obvious decomposition (e.g., run an MPI rank on each core), this document attempts to specify that.

6.4 STREAM

STREAM is an industry standard benchmark to measure memory bandwidth consumed by compute cores.

We recommend validating STREAM Triad results using MLC (Section 5.4) with command: `./mlc --peak_injection_bandwidth`, which shows bandwidth for several read/write mixes, including a 'Stream-triad-like' mix.

Note: STREAM can be built to use either regular stores (RFO stores) or Non-Temporal (NT) Stores (aka Streaming Stores). RFO stores may provide slightly higher performance on recent Intel® Xeon® processor generations. It is recommended to test both versions to measure memory bandwidth since applications may use both variants of stores. In general, applications use RFO stores, unless the compiler or the user specifically generates NT stores.

Note: STREAM benchmark recommends using an array size of at least 4 times the size of the last level (L3) cache (LLC). The size of the last level cache can be found using `lscpu` utility. However, due to the design of the L3 cache, it is not possible to eliminate all caching. Increasing STREAM array size leads to lower caching in L3.

6.4.1 Download Instructions

Download STREAM source from: https://www.cs.virginia.edu/stream/FTP/Code/

6.4.2 Spack Build Instructions

Two separate environments are recommended for RFO and NT versions. Since only one stream executable will be seen within an environment, conflicts between binaries are not possible.

6.4.2.1 Non-Temporal Store (NT) version

```
spack env create stream-nt
spack env activate stream-nt
spack add stream cflags="-O3 -xCORE-AVX512 -qopt-zmm-usage=high -qopt-
streaming-stores=always -mcmodel=medium" +openmp stream_array_size=600000000
ntimes=2000 stream_type=double offset=0 %oneapi
```

```
spack install
ln -s $(spack location -i stream)/bin/stream_c.exe ./stream_bin

# Use the command-line in Section 6.2.3 to run.
```

6.4.2.2 Regular Store (RFO) version

```
spack env create stream-rfo
spack env activate stream-rfo
spack add stream cflags="-O3 -xCORE-AVX512 -qopt-zmm-usage=high -qopt-
streaming-stores=never -fno-builtin -mcmodel=medium" +openmp
stream_array_size=600000000 ntimes=2000 stream_type=double offset=0 %oneapi
spack install
ln -s $(spack location -i stream)/bin/stream_c.exe ./stream_bin
# Use the command-line in Section 6.4.3 to run.
```

6.4.3 Run Instructions

Use the following command-line to run:

```
OMP_NUM_THREADS=<NCORES> OMP_PROC_BIND=spread
OMP_PLACES=threads OMP_DISPLAY_AFFINITY=true ./stream_bin
```

Where NCORES is the total number of cores on all sockets (processors).

Note: The following alternative command-line can also be used:

```
OMP_NUM_THREADS=<NCORES>
KMP_AFFINITY=verbose,granularity=fine,compact,1,0  ./stream_bin
```

For best results, the above command-line runs one thread per core.

STREAM output shows "Best Rate" for each of its components (Copy, Scale, Add, Triad) and the "Best Rate" serves as the performance metric. Make sure "Avg time" for each component is close to "Min time" ensuring that "Best Rate" is not an outlier.

6.5 HPL (High Performance LINPACK)

The Intel® Distribution for LINPACK* Benchmark is based on modifications and additions to High-Performance LINPACK (HPL). It is available with Intel® oneAPI Math Kernel Library (oneMKL) public library release (or as a part of Intel® oneAPI Base Toolkit (Base Kit)) with instructions.

6.5.1 Download Instructions

HPL is available with Intel® oneAPI Math Kernel Library (oneMKL), which is described in Section 5.3.

6.5.2 Run Instructions

General run instructions are available from here.

As an example, to run on a single 6900-series system with 3 SNC nodes per processor (socket), use the following instructions.

- Source oneAPI environment. E.g.,

```
source /opt/intel/oneapi/setvars.sh
```

- Make a copy of the mp_linpack directory shipped with oneMKL:

```
cp -r <MKL_ROOT_DIR>/mkl/benchmarks/mp_linpack /dev/shm/
```

- Rename .txt files and make scripts executable:

```
cd mp_linpack
mv runme_intel64_prv.txt runme_intel64_prv
mv runme_intel64_dynamic.txt runme_intel64_dynamic
chmod 755 runme_intel64_prv runme_intel64_dynamic
```

- Edit `runme_intel64_dynamic`, change these 3 lines (to run on single node):

```
export MPI_PROC_NUM=6            # use 4 for 6700-series processor
export MPI_PER_NODE=6            # use 4 for 6700-series processor
export NUMA_PER_MPI=1
```

- Execute `runme_intel64_dynamic` with parameters "`-p 3 -q 2 -b 384 -n N`" overriding default settings defined in HPL.dat.
  - p times q must be equal to the number of MPI processes (6 for single 6900-series system, and 4 for single 6700-series system)
  - b sets blocksize `Nb`
  - N should be > 120000, optimally `N` should be a multiple of p*q*Nb
  - Larger `N` leads to higher HPL number but also increases runtime. Ideally, adjust the above parameters for HPL run to consume 70%-80% of the available memory (use `numastat` to observe amount of consumed memory as described in section 4.3.3)

Example: `./runme_intel64_dynamic -p 3 -q 2 -n 221184 -b 384`

Output of HPL shows the final Gflops number, which serves as its performance metric. Make sure that output shows the run as "PASSED".

6.6 HPCG

6.6.1 Download Instructions

Prebuilt executables for the Intel® Optimized HPCG benchmark are supplied with the Intel® oneAPI Math Kernel Library (oneMKL), which is described in Section 5.3.

From oneMKL's top install directory, the HPCG package for CPU can be found under *share/mkl/benchmarks/hpcg/hpcg_cpu*. The package contains a *readme.txt* file and a *QUICKSTART* guide with instructions on how to get started, while the prebuilt executables are located in the *bin/* folder.

6.6.2 Run Instructions

In this section, we present instructions to run the HPCG benchmark on a single node of two Intel® Xeon® 6979P Processors. The instructions below can easily be adapted to run the benchmark on clusters, on other CPU models or with different problem sizes.

As an example, a node of Intel® Xeon® 6979P Processors has 2 sockets, each with 120 physical cores and 3 NUMA domains, corresponding to 40 physical cores per NUMA domain. The node in our example therefore has 240 logical cores if Hyperthreading is turned off (HT OFF) and 480 ones if Hyperthreading is turned on (HT ON). For this example, we aim to run 12 MPI processes on the node, meaning 6 MPI processes per socket, 2 MPI processes per NUMA domain, and 20 (HT OFF) / 40 (HT ON) OpenMP (OMP) threads per MPI process.

To achieve the highest performance possible, we bind each MPI process to a NUMA node. Assuming that the benchmark is run with Intel® MPI, this is achieved by properly setting environment variables `I_MPI_PIN` and `I_MPI_PIN_DOMAIN`. Please refer to this article for more information on MPI process pinning.

The below script takes the configuration described above and illustrates how to run the benchmark with local problem size `n=384` for 100 seconds (note that at least 1800 seconds of runtime is required for official submissions to TOP500), assuming HT is ON. At that given problem size, the total peak memory footprint for the optimized benchmark will stand at around 692GB translating into approximately 2.88GB of memory occupied per physical core.

```
HPCG_BIN=${PATH_TO_HPCG_CPU_BIN} # path to oneMKL's
                                 # share/mkl/benchmarks/hpcg/
                                 # hpcg_cpu/bin folder where the
                                 # executable binary lives

# HPCG GNR Run parameters
nnodes=1             # Number of nodes
n=384                # Problem size for the local HPCG domains
runtime=100          # Recommended to test with ~100s, official run
                     # with 1800s
ppn=12               # MPI processes per node on GNR 6900-series assuming 2
                     # sockets per node, 3 NUMA domains per socket, 2
```

```
                   # MPI processes per NUMA domain

# Number of OMP threads per MPI process
nthr=40            # HT ON
#nthr=20           # HT OFF

# Total number of MPI processes
let nranks=${ppn}*${nnodes}

# Set number of OMP threads subordinate to each MPI rank
export OMP_NUM_THREADS=${nthr}

# Bind MPI processes to NUMA domains
export I_MPI_PIN=yes
export I_MPI_PIN_DOMAIN=numa

# Select MPI fabrics if applicable
export I_MPI_FABRICS=shm:ofi

# Run the AVX-512 version of the prebuilt benchmark
mpiexec.hydra -genvall  -n ${nranks} -ppn ${ppn} ${HPCG_BIN}/xhpcg_avx512 -
n${n} -t${runtime} --run-real-ref=1
```

HPCG run produces an output file with “Final Summary” at the end. It should show the final GFLOP/s rating and the execution time. It also writes the final GFLOP/s rating to stdout.

7 HPC Application Recipes

7.1 Objectives

This section describes recipes for selected HPC applications.

7.2 Common SPACK Environment for Building

If the application needs building from source using Spack, please **set up the common oneAPI Spack environment as described in Section 6.2**.

7.3 Altair RADIOS

7.3.1 Download Instructions

Altair RADIOSS<version> installer can be downloaded from here. The download requires an Altair One account.

After obtaining the installer binary hwSolvers<version>_linux64.bin, install Altair RADIOSS to ALTAIR_PATH. The default location is ALTAIR_PATH=${HOME}/<version>.

```
/path/to/hwSolvers<version>_linux64.bin -i silent -
DUSER_INSTALL_DIR=${ALTAIR_PATH} -DACCEPT_EULA=YES
```

Benchmark model data:

- neon1m: Link
- t10m: Link

For example:

```
wget
https://openradioss.atlassian.net/wiki/download/attachments/47546369/Neon1m11_
2017.zip?api=v2 -O neon1m.zip
mkdir neon1m && pushd neon1m && unzip ../neon1m.zip && popd
```

7.3.2 Run Instructions

7.3.2.1 Environment Setup

```
# Source oneAPI if latest Intel libraries such as Intel MPI are needed
```

```
source /opt/intel/oneapi/setvars.sh
# Altair Radioss exports
export ALTAIR_LICENSE_PATH=<port@license_server>
export PATH=${ALTAIR_PATH}/altair/scripts/:$PATH
export ALTAIR_HOME=${ALTAIR_PATH}/altair
export
LD_LIBRARY_PATH=${ALTAIR_HOME}/hwsolvers/common/bin/linux64:${ALTAIR_HOME}/hws
olvers/radioss/bin/linux64:${ALTAIR_HOME}/hwsolvers/radioss/lib/linux64:${LD_L
IBRARY_PATH}
export RADFLEX_PATH=${ALTAIR_HOME}/hwsolvers/common/bin/linux64
export RAD_CFG_PATH=${ALTAIR_HOME}/hwsolvers/radioss/cfg
export RADIOSS_PATH=${ALTAIR_HOME}/hwsolvers/radioss/bin/linux64
# Example for Radioss version 2024.1 with Intel MPI.
export RADIOSS_STARTER_EXE=${RADIOSS_PATH}/s_2024.1_linux64
export RADIOSS_ENGINE_EXE=${RADIOSS_PATH}/e_2024.1_linux64_impi
export OMP_PROC_BIND=close
export OMP_PLACES=cores
export I_MPI_HYDRA_BOOTSTRAP=ssh
export I_MPI_HYDRA_BOOTSTRAP_EXEC_EXTRA_ARGS="" # avoid Slurm issue with MPI
ssh bootstrap
export I_MPI_PIN_DOMAIN=omp
export I_MPI_PIN_ORDER=bunch
export I_MPI_PIN_CELL=core
export OMP_STACKSIZE=400M
export I_MPI_CBWR=1
export I_MPI_PIN_RESPECT_HCA=0
export I_MPI_HYDRA_TOPOLIB=ipl2
```

7.3.2.2 Model Setup

```
# Model: neon1m
export ALTAIR_STARTER_INPUT=NEON1M11_0000.rad
export ALTAIR_ENGINE_INPUT=NEON1M11_0001.rad
# Model: t10m
export ALTAIR_STARTER_INPUT=TAURUS_A05_FFB50_0000.rad
export ALTAIR_ENGINE_INPUT=TAURUS_A05_FFB50_0001.rad
# Partition the mesh
pushd /path/to/model/
${RADIOSS_STARTER_EXE} -i ${ALTAIR_STARTER_INPUT} -np ${tasks_mpi}
```

7.3.2.3 Model Run

```
mpirun -bootstrap ssh -hosts-group ${SLURM_NODELIST} -np ${tasks_mpi} -ppn
${taskspernode} ${RADIOSS_ENGINE_EXE} -i ${ALTAIR_ENGINE_INPUT}
```

7.4 Ansys CFX

7.4.1 Download Instructions

All Ansys products can be downloaded from https://download.ansys.com. As the download is only open to registered/licensed users, one must first register and verify their account.

The download website provides the current release and many older releases to download. Once you select your desired version and platform, you can either download the full package or selected products. We recommend downloading the "primary packages" and not the "ISO images". With the "primary packages", it could be either a zip file on Windows platform or zipped tar file on Linux. Unpack the file to a directory, go to that directory, and run the INSTALL script.

```
./INSTALL -install_dir /path/to/your/desired/Ansys/location/ -silent
```

For example, with Ansys 2025R1 release, it is installed at `/path/to/your/desired/Ansys/location/v251/CFX`. In the next release 2025R2, the version number will be 252, so we can expect CFX to be installed at: `/path/to/your/desired/Ansys/location/v252/CFX`

All benchmarks need to be properly configured in order to run using a specific CFX installation.

7.4.2 Run Instructions

To run CFX, users must provide license information. You can provide license information during installation. If done during installation, users will not need to provide any extra information while running CFX.

If the installation does NOT contain license information, users must provide the license information through environmental variable `ANSYSLMD_LICENSE_FILE`. Make sure it points to the correct license server in the format: `port@machine_name`

We recommend running an MPI rank per core. To run CFX with a specific workload (for example `perf_LeMansCar_R16.def`) on a single node with 144 ranks use the following command-line:

```
env LM_LICENSE_FILE=<port@machine> ANSYSLMD_LICENSE_FILE=<port@machine>
CFX5RSH=ssh <ANSYS_BASE_DIR>/v251/CFX/bin/cfx5solve -solver
<ANSYS_BASE_DIR>/v251/CFX/bin/linux-amd64/ifort/solver-mpi.exe  -par-local -
part 144 -def <WKLD_DIR>/perf_LeMansCar_R16.def
```

To run CFX with a specific workload on 4 nodes with 144 ranks each, use the following command-line:

```
env LM_LICENSE_FILE=<port@machine> ANSYSLMD_LICENSE_FILE=<port@machine>
CFX5RSH=ssh <ANSYS_BASE_DIR>/v251/CFX/bin/cfx5solve -solver
<ANSYS_BASE_DIR>/v251/CFX/bin/linux-amd64/ifort/solver-mpi.exe -par-dist
```

```
node1*144,node2*144,node3*144,node4*144 -part 576 -def
<WKLD_DIR>/perf_LeMansCar_R16.def
```

For larger workloads (>10M cells), add `-part-large -large` to command-line:

```
env LM_LICENSE_FILE=<port@machine> ANSYSLMD_LICENSE_FILE=<port@machine>
CFX5RSH=ssh <ANSYS_BASE_DIR>/v251/CFX/bin/cfx5solve -solver
<ANSYS_BASE_DIR>/v251/CFX/bin/linux-amd64/ifort/solver-mpi.exe  -par-local  -
part 144 -part-large -large -def <WKLD_DIR>/perf_Airfoil_50M_R16.def
```

To check the performance look for Solver wall time in *.out file (for example `perf_LeMansCar_R16.out`):

```
$ grep “Solver w” *.out
```

7.5 Ansys Fluent

7.5.1 Download Instructions

All Ansys products can be downloaded from https://download.ansys.com. As the download is only open to registered/licensed users, one must first register and verify their account.

The download website provides the current release and many older releases to download. Once you select your desired version and platform, you can either download the full package or selected products. We recommend downloading the "primary packages" and not the "ISO images". With the "primary packages", it could be either a zip file on Windows platform or zipped tar file on Linux. Unpack the file to a directory, go to that directory, and run the INSTALL script.

```
./INSTALL -install_dir /path/to/your/desired/Ansys/location/ -silent
```

In general, it is recommended to install at /opt/ansys, with a product version specific directory:

For example, with Ansys 2025R1 release, you can install it at **/opt/ansys/fluent251**. For the next release 2025R2, the version number will be 252, so we can expect Fluent to be installed at: **/opt/ansys/fluent252**

All benchmarks need to be properly configured in the installation directory in order to run using a specific Fluent installation.

7.5.2 Run Instructions

To run Fluent, users must provide license information. You can provide license information during the installation. If done during installation, users will not need to provide any extra information while running CFX.

If the installation does NOT contain license information, users must provide the license information through environmental variable ANSYSLMD_LICENSE_FILE. Make sure it points to the correct license server in the format: port@machine_name

To run Fluent with customer workloads specified in input.jou, and node information specified in hostfile, with 96 mpi ranks, using intel platform, use:

```
<FluentRoot>/bin/fluent 3d -g -ssh -i input.jou -t96 -mpi=intel
-platform=intel -cflush -cnf=hostfile
```

For best results, it is recommended to use only one MPI rank per physical core, i.e., do not use hyperthreading.

Standard benchmarks can be run using the following command-line, provided that they are properly configured in the installation directory. For instance, to run the standard benchmark sedan_4m, use:

```
<FluentRoot>/bin/fluentbench.pl sedan_4m -ssh -t96 -mpi=intel
-platform=intel -cflush -nosyslog -noloadchk -cnf=hostfile
```

On a successful run, a result file named <case-name>-<number of rank>.out will be generated. Use the following BASH script to obtain the corresponding Solver Rating, which is the main performance metric. E.g., if simpleOutFile=sedan_4m-96.out, use the following commands:

```
resLine=$(grep "Solver rating" "$simpleOutFile")
solverRating=$(echo "$resLine" | awk '{print $4}')
resLine=$(grep "Solver speed" "$simpleOutFile")
solverSpeed=$(echo "$resLine" | awk '{print $4}')
resLine=$(grep "Solver wall time per iteration" "$simpleOutFile")
solverTime=$(echo "$resLine" | awk '{print $7}')
```

7.6 Ansys Mechanical

7.6.1 Download Instructions

All Ansys products can be downloaded from https://download.ansys.com. As the download is only open to registered/licensed users, one must first register and verify their account.

The download website provides the current release and many older releases to download. Once you select your desired version and platform, you can either download the full package or selected products. We recommend downloading the "primary packages" and not the "ISO images". With the "primary packages", it could be either a zip file on Windows platform or zipped tar file on Linux. Unpack the file to a directory, go to that directory, and run the INSTALL script.

```
./INSTALL -install_dir /path/to/your/desired/Ansys/location/ -silent
```

7.6.2 Run Instructions

To run Mechanical, users must provide license information. You can provide license information during the installation. If done during installation, users will not need to provide any extra information while running CFX.

If the installation does NOT contain license information, users must provide the license information through environmental variable `ANSYSLMD_LICENSE_FILE`. Make sure it points to the correct license server in the format: `port@machine_name`

For Mechanical, each version has a unique number associated with a given release number. For example, for the 2025R1 release, it will be 251.

Currently, you have to contact Ansys to obtain the standard benchmarks. As an example, to run either the standard benchmark or any customer provided workload, if the input data is in `v25direct-3.data`, and user allocated 3 nodes for computation, and 32 MPI ranks on each node, we can run Ansys Mechanical using the following command-line:

```
<MechanicalRoot>/bin/ansys251 -b -perf on -dis
-machines nodeA:32:nodeB:32:nodeC:32 -nt 1 -cflush -mopt incore
-i <WKLD_DIR>/v25direct-3.dat -o <OUT_DIR>/my96-3node-
results.out >& <OUT_DIR>/my96-3node-results.log
```

For the machine specification, to reduce the burden of the first MPI rank (rank #0), we could reduce the number of MPI ranks on the first node. For instance, `nodeA:10:nodeB:32:nodeC:32:nodeD:32` uses 106 MPI ranks in total on 4 nodes.

Note: In general, we do not recommend using multiple threads; this is the reason for using the parameter "`-nt 1`" in the command above.

On successful completion of a run, use the following BASH script to compute the Solver Rating, which is the main performance metric:

```
resLine=$( grep "Elapsed time spent computing solution" <OUT_DIR>/my96-3node-
results.out )
solverTime=$( echo "$resLine" | awk '{print $7}' )
solverRating=$( echo "scale=2;  86400 / $solverTime " | bc -l )
```

7.7 Binomial Model to Price European Options

7.7.1 Download Instructions

You can find all Intel FSI (Financial Services Industry) samples at:

https://github.com/intel/Financial-Services-Workload-Samples/

7.7.2 Build Instructions

```
spack env create fsi-env
spack env activate fsi-env

# tested compiler and libraries
spack install --add intel-oneapi-compilers@2025.3.0 intel-oneapi-tbb@2022.3.0
spack load intel-oneapi-compilers intel-oneapi-tbb
git clone https://github.com/intel/Financial-Services-Workload-Samples
cd Financial-Services-Workload-Samples/BinomialOptions

# tested git revision
git checkout 5591fe9
export __INTEL_POST_CFLAGS="-D__INTEL_COMPILER=1 -DVERBOSE=1 \
-qopt-report=max -Wno-vla-cxx-extension -fimf-precision=low \
-vecabi=cmdtarget -O3 -g -fiopenmp -std=c++20 -O3 -xCORE-AVX512 \
-qopt-zmm-usage=high -mrecip=all:0 -Xclang -target-feature \
-Xclang -fast-vector-fsqrt -Xclang -target-feature \
-Xclang -fast-scalar-fsqrt -ffinite-math-only -fno-honor-nans \
-ffp-contract=fast  -ltbb -ltbbmalloc"

icpx binomial_cpu.cpp binomial_main.cpp -o binomial.avx512
```

7.7.3 Run Instructions

To run on a 128-core CPU (single socket):

```
cd MPtest
cp ../binomial.avx512 .;
source ./clean.sh;
PR=128 source ./runbatch.sh binomial.avx512; wait
```

To find the performance metric (elapsed time in seconds) use:

```
PR=128 source ./getresults.sh res
```

7.8 Black-Scholes Model to Price European Options

7.8.1 Download Instructions

You can find all Intel FSI (Financial Services Industry) samples at:

https://github.com/intel/Financial-Services-Workload-Samples/

7.8.2 Build Instructions

```
spack env create fsi-env
spack env activate fsi-env

# tested compiler and libraries
spack install --add intel-oneapi-compilers@2025.3.0 intel-oneapi-tbb@2022.3.0
spack load intel-oneapi-compilers intel-oneapi-tbb
git clone https://github.com/intel/Financial-Services-Workload-Samples
cd Financial-Services-Workload-Samples/BlackScholes

# tested git revision
git checkout 5591fe9

# fix for the compilation error
sed -i 's/#pragma vector/\/\/#pragma vector/g' BlackScholesDP.cpp
export __INTEL_POST_CFLAGS="-D__INTEL_COMPILER=1 -DVERBOSE=1 \
-qopt-report=max -Wno-vla-cxx-extension -fimf-precision=low \
-vecabi=cmdtarget -O3 -g -fiopenmp -std=c++20 -O3 -xCORE-AVX512 \
-qopt-zmm-usage=high -mrecip=all:0 -Xclang -target-feature \
-Xclang -fast-vector-fsqrt -Xclang -target-feature \
-Xclang -fast-scalar-fsqrt -ffinite-math-only -fno-honor-nans \
-ffp-contract=fast  -ltbb -ltbbmalloc"

icpx BlackScholesDP.cpp -o BlackScholesDP.avx512
```

Note: This benchmark was created more than a decade ago for servers with a relatively low number of cores. To run it on machines with hundreds of cores, it is recommended to significantly increase the number of options in `BlackScholes.cpp` from its original value of 65536, for example:

```
      const int OPT_N = 32*65536;
```

With original value of OPT_N = 65536, a significant variation in performance is typically observed on large machines as OpenMP overhead takes more time than useful calculations and thread to core placements become critical.

7.8.3 Run Instructions

For the best performance of the original version, it is recommended to run an instance per socket or NUMA domain (or a single instance on a single socket or NUMA domain for benchmarking purposes). OMP_NUM_THREADS should be set accordingly. We recommend running one thread per core. For example, on a machine with 128 cores per socket, use the following command line:

On one socket (socket 0):

```
env OMP_NUM_THREADS=128 taskset -c 0-127 ./BlackScholesDP.avx512
```

On one NUMA node (node 2) with 42 cores per socket:

```
env OMP_NUM_THREADS=42 numactl --cpunodebind=2 --membind=2
./BlackScholesDP.avx512
```

Multiple commands can be used to run multiple copies simultaneously on multiple sockets (or NUMA nodes).

The output has the following format:

```
Double Precision Black-Scholes Formula, version 1.4
Build Time         = Jun 18 2025 10:10:18
Input Dataset      = 65536
Repetitions        = 32000
Chunk Size         = 256
Worker Threads     = 112

=============================================
Time Elapsed       = 0.3258 sec
Throughput         = 12.8758 GOptions/sec
=============================================
```

The ‘Throughput’ line above reports the performance.

7.9 GROMACS

7.9.1 Download Instructions

Visit https://manual.gromacs.org/, where one can download the version of interest. Generally, one should use the latest patch release of the year of interest, typically the current year. However, because GROMACS has a good Spack recipe, we recommend using the Spack recipe.

7.9.2 Spack Build Instructions

Prerequisite: Setup common SPACK environment (see Section 6.2).

```
spack env create my-gromacs-env
spack env activate my-gromacs-env

spack install gromacs %oneapi ^intel-oneapi-mpi ^intel-oneapi-mkl
```

One can optionally use e.g. @2025.x to get a particular version of GROMACS, compiler, or libraries.

7.9.3 Run Instructions

```
spack load gromacs%oneapi
```

Download an example of real-world usage of GROMACS. See its README for details.

```
wget https://zenodo.org/records/11234002/files/stmv_gmx_v2.tar.gz?download=1 -
O stmv_gmx_v2.tar.gz
tar xfz stmv_gmx_v2.tar.gz stmv_gmx_v2/pme_nvt.tpr
```

Run on a single node as follows:

```
MPI_RANKS=$(nproc)
export OMP_NUM_THREADS=1
mpirun -np $MPI_RANKS gmx_mpi mdrun -notunepme -dlb yes -v -resethway -
noconfout -nsteps 10000 -s stmv_gmx_v2/pme_nvt.tpr
```

NOTE: `-notunepme -dlb -resethway -noconfout -nsteps` are `mdrun` parameters.

For more details on the `mdrun` parameters please refer to:
https://manual.gromacs.org/documentation/current/user-guide/mdrun-performance.html

Usually, the best results can be obtained by running one MPI rank per core with one OpenMP thread (i.e., `export OMP_NUM_THREADS=1`). However, larger workloads prefer using two OpenMP threads per core to utilize both hyperthreads (i.e., `export OMP_NUM_THREADS=2`).

GROMACS prints its final performance metric labeled "Performance" (ns/day) to stderr and its md.log output file.

7.10 LAMMPS

7.10.1 Download Instructions

To obtain the input workloads, download LAMMPS **Last Stable Release (source code package)** from official GitHub or LAMMPS website. This includes all default workloads in `src/INTEL/TEST` directory.

7.10.2 Spack Build Instructions

Prerequisite: Setup common SPACK environment (see Section 6.2).

```
# Create and activate Spack env:

spack env create lammps
spack env activate lammps

# Add application to spec:
```

```
# Using latest available stable version - stable_22Jul2025 (20250722)

spack add lammps@20250722 cflags='-xCORE-AVX512 -O3 -qopt-zmm-usage=high -ffp-
model=fast -ffast-math -freciprocal-math -qopenmp-simd -qopenmp' cxxflags='-
xCORE-AVX512 -O3 -qopt-zmm-usage=high -ffp-model=fast -ffast-math -
freciprocal-math -qopenmp-simd -qopenmp' ldflags='-lmkl_intel_ilp64 -
lmkl_sequential -lmkl_core' +asphere+class2+dpd-
basic+intel+kspace+manybody+misc+molecule+openmp+opt+replica+rigid fft=mkl
fftw_precision=single %oneapi ^intel-oneapi-mkl ^intel-oneapi-mpi ^intel-
oneapi-tbb

# Install all application in current spec (add --deprecated if version is
deprecated)

spack install
```

7.10.3 Run Instructions

In most cases, the best solution is to use an MPI rank on each core (i.e., "pure MPI"). However, for some workloads, hyperthreading can slightly improve performance (MPI on all physical cores x 2 OMP threads); therefore, it is recommended to check both options.

For 1 node runs, please go to `rundir` which can be found in LAMMPS package under `src/INTEL/TEST`. This directory contains all default workloads and `run_benchmarks.sh` script for running all workloads on 1 node.

```
cd <PATH to LAMMPS from github>/src/INTEL/TEST

# do the following once before first run to correct name of binary
sed -i 's/lmp_intel_cpu_intelmpi/lmp_oneapi/g' run_benchmarks.sh

ln -s $(spack location -i lammps)/bin/lmp <PATH to LAMMPS from
github>/src/lmp_oneapi

./run_benchmarks.sh 2>&1 | tee run.lod
```

Note that `run_benchmarks.sh` runs all default workloads. This will run two decompositions for each workload:

1) One MPI rank for each core, without OpenMP threads
2) One MPI rank for each core with 2 OMP threads (using hyperthreads)

In the rundir, it will produce a log file (example: `<NODENAME>_lmp_oneapi_<DATE>`) for each workload and decomposition.

Performance Metric:

In output .log file, you will find 2 lines starting with "Performance:" and total wall time at the end of the file.

```
$ grep Performance airebo.384c1t.log

Performance: 0.063 ns/day, 382.846 hours/ns, 1.451 timesteps/s, 163.692 Matom-
step/s ### ignore this line
Performance: 0.052 ns/day, 461.823 hours/ns, 1.203 timesteps/s, 135.699 Matom-
step/s ### this is the actual performance line
```

Please ignore the first line with "Performance:" since it refers to warm-up steps. Main metric for all workloads is timesteps/s (higher in better).

7.11 Monte Carlo Method to Price European Options

7.11.1 Download Instructions

You can find all Intel FSI samples at:

https://github.com/intel/Financial-Services-Workload-Samples/

7.11.2 Build Instructions

```
spack env create fsi-env
spack env activate fsi-env

# tested compiler and libraries
spack install --add intel-oneapi-compilers@2025.3.0 intel-oneapi-tbb@2022.3.0
spack install --add intel-oneapi-mkl@2024.2.2
spack load intel-oneapi-compilers intel-oneapi-tbb intel-oneapi-mkl
git clone https://github.com/intel/Financial-Services-Workload-Samples
cd Financial-Services-Workload-Samples/MonteCarloEuropeanOptions

# tested git version
git checkout 5591fe9
export __INTEL_POST_CFLAGS="-D__INTEL_COMPILER=1 \
-qopt-report=max -Wno-vla-cxx-extension -fimf-precision=low \
-vecabi=cmdtarget -O3 -g -fiopenmp -std=c++20 -O3 -xCORE-AVX512 \
-qopt-zmm-usage=high -mrecip=all:0 -Xclang -target-feature \
-Xclang -fast-vector-fsqrt -Xclang -target-feature \
-Xclang -fast-scalar-fsqrt -ffinite-math-only -fno-honor-nans \
-ffp-contract=fast  -lmkl_intel_ilp64 -lmkl_sequential -lmkl_core  \
-ltbb -ltbbmalloc"

icpx MonteCarloInsideBlockingDP.cpp -o MonteCarloInsideBlockingDP.avx512
```

7.11.3 Run Instructions

The test is designed to perform as a batch of single-threaded instances. Use the following sequence of commands:

```
cd MPTest                        # this directory contains all .sh files
cp ../MonteCarloInsideBlockingDP.avx512 .
source ./clean.sh

# number of processes (PR) should be equal to the number of cores
PR=256 source ./runbatch.sh MonteCarloInsideBlockingDP.avx512;wait
```

The last command outputs average instance processing time, which is an inverse performance metric.

```
PR=256 source ./getresults.sh res
```

7.12 Monte Carlo Method to Price American Options

7.12.1 Download Instructions

You can find all Intel FSI samples at

https://github.com/intel/Financial-Services-Workload-Samples/

7.12.2 Build Instructions

```
spack env create fsi-env
spack env activate fsi-env

# tested compiler and libraries
spack install --add intel-oneapi-compilers@2025.3.0 intel-oneapi-tbb@2022.3.0
spack install --add intel-oneapi-mkl@2024.2.2
spack load intel-oneapi-compilers intel-oneapi-tbb intel-oneapi-mkl
git clone https://github.com/intel/Financial-Services-Workload-Samples
cd Financial-Services-Workload-Samples/MonteCarloEuropeanOptions

# tested git revision
git checkout 5591fe9
export __INTEL_POST_CFLAGS="-D__INTEL_COMPILER=1 \
-qopt-report=max -Wno-vla-cxx-extension -fimf-precision=low \
-vecabi=cmdtarget -O3 -g -fiopenmp -std=c++20 -O3 -xCORE-AVX512 \
-qopt-zmm-usage=high -mrecip=all:0 -Xclang -target-feature \
-Xclang -fast-vector-fsqrt -Xclang -target-feature \
-Xclang -fast-scalar-fsqrt -ffinite-math-only -fno-honor-nans \
-ffp-contract=fast  -lmkl_intel_ilp64 -lmkl_sequential -lmkl_core  \
-ltbb -ltbbmalloc"

icpx main.cpp utilities.cpp -o amc-avx512
```

7.12.3 Run Instructions

The test was originally designed to perform as a batch of single-threaded instances. Use the following sequence of commands:

```
cd MPTest
cp ../amc-avx512 .
source ./clean.sh

# number of processes (PR) should be equal to the number of cores
PR=256 source ./runbatch.sh amc-avx512; wait
```

The last command outputs average instance processing time, which is an inverse performance metric.

```
PR=256 source ./getresults.sh res
```

7.13 MPAS-A

The Model for Prediction Across Scales – Atmosphere (MPAS-A) is a state-of-the-art, non-hydrostatic global atmospheric model developed collaboratively by the National Center for Atmospheric Research (NCAR) and Los Alamos National Laboratory. MPAS-A is designed for both weather and climate research, offering unique capabilities for high-resolution global and regional simulations.

Official Website: https://www.mmm.ucar.edu/models/mpas
Official GitHub Repository: https://github.com/MPAS-Dev/MPAS-Model

7.13.1 Download Instructions

Please refer to the following link to download MPAS-A version 7 benchmark cases: https://www2.mmm.ucar.edu/projects/mpas/benchmark/v7.0/

For example, to download benchmark_60km dataset, use https://www2.mmm.ucar.edu/projects/mpas/benchmark/v7.0/MPAS-A_benchmark_60km_v7.0.tar.gz

7.13.2 Spack Build Instructions

Prerequisite: Setup common SPACK environment (see Section 6.2).

Check the exact Spack version, which was installed on your system in Section 6.2 by running:

```
spack --version
# Should output: 1.0.0.dev0 (cd3da0e6b7f4bc805777f5fb334e991e45a12262)
```

Install MPAS-A version 7.3 with AVX2 optimization flags and Intel® oneAPI compilers using Spack:

```
spack install --add mpas-model@7.3 \
fflags="-O3 -xCORE-AVX2" \
cflags="-O3 -xCORE-AVX2" \
%oneapi \
^netcdf-c@4.9.2+mpi+parallel-netcdf \
^netcdf-fortran@4.6.1 \
^parallel-netcdf@1.12.3+fortran \
^parallelio@2.6.2+pnetcdf \
^intel-oneapi-mpi
```

For parallel runs with MPI, also install the grid partitioner METIS library:

```
spack install metis%oneapi
```

7.13.3 Run Instructions

Use the following steps to run. This will launch **256** MPI ranks. We recommend running MPAS-A with **one MPI rank per physical core**.

```
#!/bin/bash

# Load MPAS-A and Metis via Spack
spack load mpas-model%oneapi
spack load metis%oneapi

# Navigate to MPAS-A installation directory managed by Spack
spack cd -i mpas-model

# Download MPAS-A 60km benchmark dataset from UCAR
wget https://www2.mmm.ucar.edu/projects/mpas/benchmark/v7.0/MPAS-
A_benchmark_60km_v7.0.tar.gz

 # Extract benchmark dataset
tar -xzvf MPAS-A_benchmark_60km_v7.0.tar.gz

# Enter benchmark directory
cd MPAS-A_benchmark_60km_v7.0

# Create symbolic link to MPAS-A executable in current directory
ln -sf ../bin/atmosphere_model .

# Remove stack size limit to prevent MPI memory errors
ulimit -s unlimited

# Total MPI processes (256 for full-node utilization on 1-node Intel(R)
Xeon(R) 6980P)
export NUM_MPI_RANKS=256

# OpenMP threads per MPI rank (hybrid MPI+OpenMP configuration)
export OMP_NUM_THREADS=1
```

```
# Use shared memory transport (single-node optimization)
# For multi-node InfiniBand: export I_MPI_FABRICS=shm:ofi
export I_MPI_FABRICS=shm

# Automatically configure CPU pinning domains
export I_MPI_PIN_DOMAIN=auto

# Group MPI ranks on adjacent cores for better locality
export I_MPI_PIN_ORDER=bunch

# Bind OpenMP threads close to their parent MPI process
export OMP_PROC_BIND=close

# Place OpenMP threads on physical cores (not SMT threads)
export OMP_PLACES=cores

# Increase OpenMP thread stack size to 512MB
export KMP_STACKSIZE=512M

# Generate domain decomposition using Metis
# Partitions the x1.163842.graph mesh into $NUM_MPI_RANKS subdomains gpmetis
x1.163842.graph.info $NUM_MPI_RANKS

# Launch MPAS-A using Hydra MPI with all configured environment variables
mpiexec.hydra -genvall -n $NUM_MPI_RANKS ./atmosphere_model
```

After completing the simulations, execute the following command to retrieve the average elapsed time per timestep in seconds:

```
string=$(sed -n '/time integration/p' log.atmosphere.0000.out) && read -ra
arr <<< "$string" && echo ${arr[7]}
```

7.14 NAMD

7.14.1 Download Instructions

NAMD source code can be download from here. For the Spack recipe, please choose source code of Version 3.0.1 (2024-10-14) Platforms.

To parse output to obtain performance metric, get this script.

7.14.2 Spack Build Instructions

Prerequisite: Setup common SPACK environment (see Section 6.2).

Use the following Spack recipe (with the diff patch given below applied as needed):

```
spack env create myenv
spack env activate myenv
spack add namd@3.0.1 cflags='-qopt-zmm-usage=high -qopenmp-simd -freciprocal-
math' cxxflags='-qopt-zmm-usage=high -qopenmp-simd -freciprocal-
math' ldflags='-fuse-ld=lld' +avxtiles fftw=mkl interface=tcl %oneapi ^charmpp
~fortran cxxflags='-O3 -xCORE-AVX512 -qopt-zmm-usage=high' ldflags='-fuse-
ld=lld' +omp+smp backend=mpi build-target='charm++' %oneapi ^intel-oneapi-mpi
spack install namd
```

7.14.3 Run Instructions

Use the following script to execute workloads.

```
spack load namd %oneapi

export I_MPI_FABRICS=shm
export I_MPI_PIN=off

DATE_STRING=`date +%s`
LOG_DIR="../RESULTS_"$DATE_STRING
mkdir $LOG_DIR

N_CORES=`grep processor /proc/cpuinfo | wc -l`

BINARY="namd3"

# Change the corresponding workload name here
WORKLOAD="stmv_npt_2fs"

# Number of MPI tasks to use per node
NPPN="32"
# Setup the NAMD affinity based on the number of MPI tasks per node
NAFFIN=`echo $N_CORES $NPPN | awk '{p=($1-$2)/$2; c=$1-1; f=p+1; print
"+ppn",p,"+commap ",0"-"c":"f,"+pemap 1-"c":"f"."p}'`

echo -n "Running $BINARY $WORKLOAD on $H_CORES with PPN=$NPPN..."

cmd="mpirun -ppn $NPPN -np $NPPN $BINARY $NAFFIN
$NAMD_BENCHMARK/$WORKLOAD/$WORKLOAD.namd"

echo -n "The command is $cmd"

echo $cmd >> $LOG_DIR/commands.log
$cmd > $LOG_DIR/$WORKLOAD.log
$PYTHON ./ns_per_day.py $LOG_DIR/$WORKLOAD.log | awk '$1=="Nanoseconds"{print
$4}'

echo "Running $BINARY $WORKLOAD is done"
```

When the run is complete, execute the following script to obtain the performance metric (ns/day): https://www.ks.uiuc.edu/~dhardy/scripts/ns_per_day.py

7.15 NEMO

The NEMO Ocean model, short for Nucleus for European Modelling of the Ocean, is a state-of-the-art, open-source modeling framework designed for simulating and studying the ocean and its interactions with the climate system. Developed by a European consortium, NEMO is widely used for oceanographic research, operational forecasting, seasonal prediction, and climate studies.

Official Website: https://www.nemo-ocean.eu
User Guide: https://sites.nemo-ocean.io/user-guide

7.15.1 Download Instructions

This recipe outlines how to manually build NEMO from a public git repository, since NEMO is not yet available in SPACK. NEMO requires **XIOS, HDF5, NetCDF (C and Fortran)** to be installed on your system.

Below is a step-by-step guide for installing and configuring NEMO version 4.2.2 with Intel® oneAPI compilers.

7.15.2 Build Instructions

First, install HDF5, NetCDF, HDF5, XIOS and other required libraries (e.g., ZLIB) on your system:

```
#!/bin/bash

# Define target path for I/O libraries
export IO_LIBS_PATH=<path on your local system>

# Set Up Environment Variables
export CC=icx
export CXX=icpx
export FC=ifx
export F77=ifx
export F90=ifx
export MPICC=mpiicx
export MPIFC=mpiifx
export CFLAGS="-fPIC -O3 -xHost -ip -fno-alias -align"
export FFLAGS="-fPIC -O3 -xHost -ip -fno-alias -align"
export CXXFLAGS="-fPIC -O3 -xHost -ip -fno-alias -align"
export LDFLAGS="-L/usr/local/lib"
export CPPFLAGS="-I/usr/local/include"
export LD_LIBRARY_PATH=${IO_LIBS_PATH}/lib:$LD_LIBRARY_PATH

# Install ZLIB (if not already present). Adjust IO_LIBS_PATH if you prefer a
# different install prefix
wget https://zlib.net/zlib-1.3.1.tar.gz
tar xf zlib-1.3.1.tar.gz
cd zlib-1.3.1
./configure --prefix=$IO_LIBS_PATH
```

```
make
sudo make install
cd ..

# Install HDF5 library
wget https://support.hdfgroup.org/ftp/HDF5/releases/hdf5-1.14/hdf5-
1.14.6/src/hdf5-1.14.6.tar.gz
tar xf hdf5-1.14.6.tar.gz
cd hdf5-1.14.6
./configure --prefix=$IO_LIBS_PATH --enable-shared --enable-fortran --enable-
parallel --with-zlib=$IO_LIBS_PATH --with-pic
make clean
make -j4 && make install
cd ..

# Install NETCDF-C library
wget https://github.com/Unidata/netcdf-c/archive/refs/tags/v4.9.2.tar.gz
tar xf v4.9.2.tar.gz
cd netcdf-c-4.9.2
export CFLAGS="-O2 -I${IO_LIBS_PATH}/include"
export CPPFLAGS="-O2 -I${IO_LIBS_PATH}/include"
export LDFLAGS="-L${IO_LIBS_PATH}/lib"
./configure --prefix=/usr/local --enable-netcdf-4 --enable-shared --disable-
dap --with-hdf5=$IO_LIBS_PATH
make clean
make -j4 && make install
cd ..

# Install NETCDF-Fortran library
wget https://github.com/Unidata/netcdf-fortran/archive/refs/tags/v4.6.1.tar.gz
tar xf v4.6.1.tar.gz
cd netcdf-fortran-4.6.1
export CPPFLAGS="-I${IO_LIBS_PATH}/include"
export CFLAGS="-I${IO_LIBS_PATH}/include"
export LDFLAGS="-L${IO_LIBS_PATH}/lib"
./configure CC=icx FC=ifx F77=ifx MPICC=mpiicx MPIFC=mpiifx \
  --prefix=$IO_LIBS_PATH \
  --enable-shared \
  LIBS="-lnetcdf -lhdf5_hl -lhdf5 -lz -lsz"
make clean
make -j4 && make install
cd ..
```

Next, install XIOS file I/O library:

```
# Clone XIOS library
svn co http://forge.ipsl.jussieu.fr/ioserver/svn/XIOS/trunk XIOS
cd XIOS

# Add the following makefile configuration files
cat > arch/arch-ifx_local.fcm << EOF
%CCOMPILER           mpiicx
%FCOMPILER           mpiifx
%LINKER              mpiifx
```

```
%BASE_CFLAGS    -ansi -w -D_GLIBCXX_USE_CXX11_ABI=0 -std=c++11 -diag-
disable=10441
%PROD_CFLAGS    -O3 -fp-model source -no-fma -traceback -std=c++11 -
DBOOST_DISABLE_ASSERTS
%DEV_CFLAGS     -g -traceback
%DEBUG_CFLAGS   -DBZ_DEBUG -g -traceback -fno-inline
%BASE_FFLAGS    -D__NONE__
%PROD_FFLAGS    -O3 -r8 -fp-model source -no-fma
%DEV_FFLAGS     -g -O2 -traceback
%DEBUG_FFLAGS   -g -traceback
%BASE_INC       -D__NONE__
%BASE_LD        -lstdc++
%CPP            mpiicx -EP
%FPP            cpp -P
%MAKE           gmake
EOF

# Add the following makefile configuration files
cat > arch/arch-ifx_local.path << EOF
NETCDF_INCDIR="-I<your NETCDF directory>/include"
NETCDF_LIBDIR="-L<your NETCDF directory>/lib"
NETCDF_LIB="-lnetcdf -lnetcdff"
HDF5_INCDIR="-I<your HDF5 directory>/include"
HDF5_LIBDIR="-L<your HDF5 directory>/lib"
HDF5_LIB="-lhdf5_hl -lhdf5 -lhdf5 -lz -lcurl"
EOF

ln -sf arch-ifx_local.path arch.path
ln -sf arch-ifx_local.env arch.env

# Build XIOS
./make_xios --full --prod --arch ifx_local -j8
```

Finally, install NEMO:

```
# Clone NEMO code
git clone --branch 4.2.2 https://forge.nemo-ocean.eu/nemo/nemo.git nemo_4.2.2

cd nemo_4.2.2

# All compiler options in NEMO are controlled using files in
# /arch/arch-'my_arch'.fcm where my_arch is the name you use to refer
# to your computing environment.
# Add the following makefile configuration file:
cat > arch/arch-ifx_local.path << EOF
%NCDF_HOME           <your NETCDF directory>
%HDF5_HOME           <your HDF5 directory>
%XIOS_HOME           <your XIOS directory>
%OASIS_HOME          /not/defiled
%NCDF_INC            -I%NCDF_HOME/include
%NCDF_LIB            -L%NCDF_HOME/lib -lnetcdff -lnetcdf -L%HDF5_HOME/lib -
lhdf5_hl -lhdf5 -lhdf5
%XIOS_INC            -I%XIOS_HOME/inc
```

```
%XIOS_LIB            -L%XIOS_HOME/lib -lxios -lstdc++
%OASIS_INC           -I%OASIS_HOME/build/lib/mct -
I%OASIS_HOME/build/lib/psmile.MPI1
%OASIS_LIB           -L%OASIS_HOME/lib -lpsmile.MPI1 -lmct -lmpeu -lscrip
%CPP                 cpp
%FC                  mpiifx -c -cpp
%FCFLAGS             -i4 -r8 -O2 -fp-model precise -fno-alias -align
array64byte -xCORE-AVX512
%FFLAGS              %FCFLAGS
%LD                  mpif90 -f90=ifx
%LDFLAGS
%FPPFLAGS            -P -traditional -std=c99
%AR                  ar
%ARFLAGS             rs
%MK                  gmake
%USER_INC            %XIOS_INC %NCDF_INC
%USER_LIB            %XIOS_LIB %NCDF_LIB
%CC                   mpiicx
%CFLAGS              -O0
EOF

# Build NEMO GYRE_PISCES test case
./makenemo -m linux_ifx_avx512 -r GYRE_PISCES -n MY_GYRE_PISCES clean
./makenemo -m linux_ifx_avx512 -r GYRE_PISCES -n MY_GYRE_PISCES -j 8


# Build NEMO BENCH test case
./makenemo -m linux_ifx_avx512 -a BENCH -n MY_BENCH clean
./makenemo -m linux_ifx_avx512 -a BENCH -n MY_BENCH
```

For more details about NEMO build process, please check the NEMO official user-guide.

7.15.3 Run Instructions

We recommend running NEMO with **one MPI rank per physical core** without using OpenMP.

To run NEMO BENCH ORCA1 test case, go to NEMO installed directory and execute the following steps:

```
#!/bin/bash

# Navigate to the NEMO benchmark experiment directory
cd nemo-4.2/tests/MY_BENCH/EXP00
# Link the ORCA1-like configuration namelist as the active namelist_cfg
ln -sf namelist_cfg_orca1_like namelist_cfg

# Remove stack size limit to prevent memory allocation errors
ulimit -s unlimited

# Set the total number of MPI ranks
```

```
export NUM_MPI_RANKS=256

# Use shared memory fabric for MPI communication (single-node optimization)
export I_MPI_FABRICS=shm

# Automatically determine optimal CPU pinning domains
export I_MPI_PIN_DOMAIN=auto

# Group MPI ranks together on adjacent cores for better locality
export I_MPI_PIN_ORDER=bunch

# Launch the NEMO simulation with MPI
mpiexec.hydra -genvall -n $NUM_MPI_RANKS ./nemo
```

To run NEMO GYRE_PISCES test case with nn_GYRE=15 (GYRE resolution [1/degrees]), go to NEMO installed directory and execute the following steps:

```
#!/bin/bash

# Navigate to the NEMO benchmark experiment directory
cd nemo-4.2/tests/MY_GYRE_PISCES/EXP00

# Set nn_GYRE=15 (grid resolution) and jpkglo=101 (number of vertical levels)
in namelist_cfg file

# Remove stack size limit to prevent memory allocation errors
ulimit -s unlimited

# Set the total number of MPI ranks
export NUM_MPI_RANKS=256

# Use shared memory fabric for MPI communication (single-node optimization)
export I_MPI_FABRICS=shm

# Automatically determine optimal CPU pinning domains
export I_MPI_PIN_DOMAIN=auto

# Group MPI ranks together on adjacent cores for better locality
export I_MPI_PIN_ORDER=bunch

# Launch the NEMO simulation with MPI
mpiexec.hydra -genvall -n $NUM_MPI_RANKS ./nemo
```

7.15.3.1 Extracting Performance Metric

The benchmark reports timing data in `timing.output` file. The performance metric (the average elapsed time per timestep) can be computed using the following command issued within the run directory after benchmark test is completed:

```
grep ' timing step ' $1 | awk '{print $5}' | awk -f stats.awk
---
    items:      1000
      max:         0.505032
      min:         0.255904
      sum:       263.053349
     mean:         0.263053
 mean/max:         0.520864
```

The `stats.awk` file contains the following:

```
BEGIN{ a = 0.0 ; i = 0 ; max = -999999999  ; min = 9999999999 }
{
    i ++
    a += $1
    if ( $1 > max ) max = $1
    if ( $1 < min ) min = $1
}
END{ printf("---
\n%10s  %8d\n%10s  %15f\n%10s  %15f\n%10s  %15f\n%10s  %15f\n%10s  %15f\n","it
ems:",i,"max:",max,"min:",min,"sum:",a,"mean:",a/(i*1.0),"mean/max:",(a/(i*1.0
))/max) }
```

7.16 NWChem

7.16.1 Download Instructions

git clone https://github.com/nwchemgit/nwchem.git

7.16.2 Spack Build Instructions

Prerequisite: Setup common SPACK environment (see Section 6.2).

Create the following `spack.yaml` file:

```
# Contents of spack.yaml
spack:
  packages:
    all:
      providers:
        blas: [intel-oneapi-mkl]
        lapack: [intel-oneapi-mkl]
        scalapack: [intel-oneapi-mkl]
  specs:
    - nwchem+openmp fflags="-xCORE-AVX512 -qopt-zmm-usage=high" armci=mpi-pr
%oneapi ^intel-oneapi-mkl+cluster+ilp64 mpi_family=mpich ^intel-oneapi-mpi
  view: true
  concretizer:
    unify: true
```

Use the following commands to create environment and install:

```
spack env create nwchem-env
spack env activate nwchem-env
```

```
# Take spack.yaml (from above) and place it in $SPACK_ENV/spack.yaml
cp -pr spack.yaml $SPACK_ENV/spack.yaml
spack install
```

7.16.3 Run Instructions

Use the following script to run:

```
#!/bin/bash

#load spack env
spack env activate nwchem-env

# Loading NWChem build
spack load nwchem armci=mpi-pr +openmp fflags="-xCORE-AVX512 -qopt-zmm-
usage=high" %oneapi ^intel-oneapi-mkl+cluster+ilp64 mpi_family=mpich ^intel-
oneapi-mpi

# Download input file for the workload C240 Buckyball
wget https://nwchemgit.github.io/c240_631gs.nw

# OMP options
export OMP_NUM_THREADS=1

# Run command for NWChem on a node w/ 2 sockets and 96 core per socket
mpirun -n 192 -bootstrap ssh nwchem c240_631gs.nw |& tee run.log

# Print performance metric
cat run.log | grep "Total times"
```

7.17 OpenFOAM

7.17.1 Download Instructions

Download from https://openfoam.com/ (OpenCFD Ltd.).

7.17.2 Spack Build Instructions

Prerequisite: Setup common SPACK environment (see Section 6.2).

Always use the latest git from the [Spack GitHub](). There is a compilation problem with CGAL dependency and it needs to be fixed before "`spack install`" command. Only CGAL 6.x version works without any problem with oneAPI compiler. To install in an environment, use:

```
# Optional: replace "args = ["-silent"]" with "args = []" to have more #
verbose compilation output with "-verbose" option for spack install
# command. To edit, use: spack edit openfoam
```

```
#
spack env create openfoam
spack env activate openfoam
spack install --show-log-on-error --add openfoam%oneapi ^intel-oneapi-mpi
cxxflags="-O3 -mtune=native" ^cgal@6
```

7.17.3 Run Instructions

Here are the run instructions for occDrivAerStaticMesh 65M.

```
git clone https://develop.openfoam.com/committees/hpc.git
cd hpc/incompressible/simpleFoam/occDrivAerStaticMesh
wget https://zenodo.org/records/15012221/files/polyMesh_65M.tar.gz?download=1
-O polyMesh_65M.tar.gz
tar zxvf polyMesh_65M.tar.gz
mv polyMesh constant/

# Change "decompositionMethod hierarchical;" to "decompositionMethod scotch;"
sed -i "/decompositionMethod/c\decompositionMethod scotch;"
system/include/caseDefinition
# Change "nCores 512;" to "nCores N;" where N is digit and the number of MPI
ranks to use
sed -i "/nCores/c\nCores N;" system/include/caseDefinition
# Change "endTime 4000;" to "endTime 200;"
sed -i "/endTime/c\endTime 200;" system/controlDict.noWrite

spack load intel-oneapi-compilers
spack load openfoam

export I_MPI_PIN_PROCESSOR_LIST=allcores:map=spread

./Allrun
```

Performance metric is ClockTime or iterations per minute.

```
# The above will give ClockTime which is usually used for clock time
# comparison. And iterations per minutes also will be calculated.

clocktime=$(grep ClockTime logFiles/50_simpleFoam* | tail -1 | awk '{print
$(NF-1)}')
niter=$(grep ClockTime logFiles/50_simpleFoam* | wc -l)
niterpermin=$(echo "scale=2;${niter}/(${clocktime}/60)" | bc -q)
printf "OpenFOAM ClockTime  = %d seconds\n" $clocktime
printf "OpenFOAM iterations/minutes = %.2f\n" $niterpermin
```

7.18 QuantLib

7.18.1 Download Instructions

You can find QuantLib, a free/open-source library for quantitative finance at:

https://github.com/lballabio/quantlib

7.18.2 Build Instructions

```
spack env create quantlib-env
spack env activate quantlib-env
# tested compiler and libraries
spack install --add intel-oneapi-compilers@2025.3.0
spack install --add boost@1.88
spack load intel-oneapi-compilers boost
git clone https://github.com/lballabio/quantlib
mkdir quantlib/build
cd quantlib/build
cmake .. -G "Unix Makefiles" -D CMAKE_BUILD_TYPE=Release -
DQL_ENABLE_PARALLEL_UNIT_TEST_RUNNER=1 -DCMAKE_CXX_COMPILER=icpx -
DCMAKE_CXX_FLAGS="-O3 -ffast-math -D__FAST_MATH__=1 -fimf-use-
svml=true:exp,pow,expl,erf -fimf-domain-exclusion=31 -fimf-precision=med -fno-
math-errno -fno-finite-math-only -fhonor-nans -Wno-unused-command-line-
argument -fno-associative-math -march=graniterapids"

make -j
```

7.18.3 Run Instructions

The test was originally designed to perform as a batch of single-threaded instances. Use the following sequence of commands:

```
cd test-suite

# select right size with --size and concurrency with --nProc
./quantlib-benchmark --size=1 --nProc=16
```

The last command outputs summary that was run and performance metrics.

```
Benchmark Size         = Custom (1)
Number of processes    = 16
System Throughput      = 11.248 tasks/s
Benchmark Runtime      = 7.73473s
```

7.19 RELION

7.19.1 Download Instructions

Download and build using the git commands given in the next section since a Spack recipe is not currently valid.

The latest version of the Intel® oneAPI compiler, MKL, TBB, IPP, and MPI are required.

RELION requires installation of the following linux packages: CMake, libtiff-dev*, libpng-dev*, libjpeg-dev*, and libxft-dev (or libXft-devel).

7.19.2 Build Instructions

Instructions for building with the oneAPI compilers:

```
git clone https://github.com/3dem/relion.git relion
cd relion; mkdir build; cd build
cmake \
-DCMAKE_C_COMPILER=icx \
-DCMAKE_CXX_COMPILER=icpx \
-DMPI_C_COMPILER=mpiicx \
-DMPI_CXX_COMPILER=mpiicpx \
-DCUDA=OFF \
-DALTCPU=ON \
-DMKLFFT=ON \
-DGUI=OFF \
-DFETCH_WEIGHTS=OFF \
-DCMAKE_BUILD_TYPE=Release \
-DCMAKE_C_FLAGS="-g -O3 -qopenmp-simd -xCORE-AVX512 -qopt-zmm-usage=high" \
-DCMAKE_CXX_FLAGS="-DTBB_SUPPRESS_DEPRECATED_MESSAGES -
DTIFF_DISABLE_DEPRECATED -g -O3 -qopenmp-simd -xCORE-AVX512 -qopt-zmm-
usage=high" \
-DCMAKE_EXE_LINKER_FLAGS="-static-intel -static-libgcc -qopenmp-link=static -
Wno-unused-command-line-argument"  ..
make -j
```

You may also need to specify the location for the TIFF, PNG, and JPEG libraries and include files using (for example), `-DTIFF_LIBRARY=<location>`, `-DTIFF_LIBRARIES=<location>`, `-DTIFF_INCLUDE_DIR=<location>`, and `-DTIFF_INCLUDE_DIRS=<location>`.

7.19.3 Run Instructions

RELION uses a client-server model based on MPI. The primary MPI rank (MPI rank 0) controls and distributes work items to many computing clients. Therefore, **running with an odd number of MPI ranks is required**.

The number of ranks to use, and the number of threads per rank, depends on the number of true cores per socket. With Intel® Hyperthreading Technology enabled (giving about a 20% speedup), we recommend:

- Using 8 MPI ranks ***per socket*** if your system has less than or equal to 64 cores per socket, 12 MPI ranks per socket if around 96 cores per socket, and 16 MPI ranks per socket if around 128 cores.
- Aim for 14-32 threads per MPI rank so the product of the (MPI ranks) and (threads per MPI rank) is slightly less than or equal to the total number of hyperthreads in your system.

For larger problems with very long runtimes or large memory requirements you will likely want to run RELION on multiple nodes (systems), ideally of the same size. In this case you should multiply the number of MPI ranks you calculated above by the number of nodes you will use and then modify the scripts below with the appropriate number of nodes and total number of MPI ranks.

For example, the following script runs **Plasmodium Ribosome 3D classification** for 25 iterations on a single dual-socket server with 86 cores per socket (86*2*2=344 hyperthreads).

```
wget ftp://ftp.mrc-lmb.cam.ac.uk/pub/scheres/relion_benchmark.tar.gz
tar zxf relion_benchmark.tar.gz
cd relion_benchmark
cat << EOF > run_3d.sh
#!/bin/bash

NUM_NODES=1    # single machine
TOTAL_HYPERTHREADS_PER_NODE=344   # For this 86-core part – 86*2*2
NUM_COMPUTE=24   # This is an example of a dual-socket server with 86 cores
per socket, so 12 * 2.
TOTAL_NUM_MPI_RANKS=$((NUM_COMPUTE+1))
NUM_THREADS=$((TOTAL_HYPERTHREADS_PER_NODE/NUM_COMPUTE))
NUM_ITERATIONS=25
POOLSIZE=4
RANKS_PER_HOST=$((TOTAL_NUM_MPI_RANKS/NUM_NODES))
APPEXE=/path/to/relion/build/bin/relion_refine_mpi

export I_MPI_PIN_DOMAIN=numa
export OMP_NUM_THREADS=$NUM_THREADS

STARTTIME=$(/bin/date +%s)
mpirun -n $TOTAL_NUM_MPI_RANKS -perhost $RANKS_PER_HOST $APPEXE --i
Particles/shiny_2sets.star --ref emd_2660.map:mrc --firstiter_cc --ini_high 60
--dont_combine_weights_via_disc --ctf --tau2_fudge 4 --particle_diameter 360 -
-K 6 --flatten_solvent --zero_mask --oversampling 1 --healpix_order 2 --
offset_range 5 --offset_step 2 --sym C1 --norm --scale --pad 2 --random_seed 0
--o ./3D --pool $POOLSIZE --j $NUM_THREADS --iter $NUM_ITERATIONS --cpu
ENDTIME=$(/bin/date +%s)
secs=$((ENDTIME - STARTTIME))

rm -f 3D_it*

printf '3D Classification WALLTIME %d seconds (%d MPI %d iter)\n' $secs
$((TOTAL_NUM_MPI_RANKS-1)) $NUM_ITERATIONS
EOF
chmod 755 run_3d.sh
./run_3d.sh
```

The above script will print wall time in seconds, which is the final performance metric.

Here are the instructions for **Plasmodium Ribosome 2D classification** at 25 iterations.

```
wget ftp://ftp.mrc-lmb.cam.ac.uk/pub/scheres/relion_benchmark.tar.gz
tar zxf relion_benchmark.tar.gz
cd relion_benchmark
## Skip above if you have already downloaded it for 3D classification
cat << EOF > run_2d.sh
#!/bin/bash

NUM_NODES=1    # single machine
TOTAL_HYPERTHREADS_PER_NODE=344   # For this 86-core part – 86*2*2
NUM_COMPUTE=24   # This is an example of a dual-socket server with 86 cores
per socket, so 12 * 2.
TOTAL_NUM_MPI_RANKS=$((NUM_COMPUTE+1))
NUM_THREADS=$((TOTAL_HYPERTHREADS_PER_NODE/NUM_COMPUTE))
NUM_ITERATIONS=25
POOLSIZE=4
RANKS_PER_HOST=$((TOTAL_NUM_MPI_RANKS/NUM_NODES))
APPEXE=/path/to/relion/build/bin/relion_refine_mpi

export I_MPI_PIN_DOMAIN=numa
export OMP_NUM_THREADS=$NUM_THREADS

STARTTIME=$(/bin/date +%s)
mpirun -n $TOTAL_NUM_MPI_RANKS -perhost $RANKS_PER_HOST $APPEXE --i
Particles/shiny_2sets.star --dont_combine_weights_via_disc --ctf --
tau2_fudge 2 --particle_diameter 360 --K 200 --zero_mask --oversampling 1 --
psi_step 6 --offset_range 5 --offset_step 2 --norm --scale --random_seed 0 --
pad 2 --o ./2D --pool $POOLSIZE --j $NUM_THREADS --iter $NUM_ITERATIONS --cpu
ENDTIME=$(/bin/date +%s)
secs=$((ENDTIME - STARTTIME))

rm -f 2D_it*

printf '2D Classification WALLTIME %d seconds (%d MPI %d iter)\n' $secs
$((TOTAL_NUM_MPI_RANKS-1)) $NUM_ITERATIONS
EOF
chmod 755 run_2d.sh
./run_2d.sh
```

The above script will print wall time in seconds, which is the final performance metric.

7.20 VASP

7.20.1 Download Instructions

VASP software is distributed under license. Users should obtain a benchmark license and download one of the latest release versions (e.g. 6.4.3).

Benchmarks can be downloaded from https://github.com/NREL/ESIFHPC3/tree/master/VASP from the subdirectories bench1/input and bench2/input.

The INCAR-* files should be renamed to INCAR for all variants of benchmark (`-gga -gw -hse`). E.g.:

```
cp INCAR-gga INCAR
```

7.20.2 Spack Build Instructions

Prerequisite: Setup common SPACK environment (see Section 6.2).

If the VASP recipe is available on Spack GitHub, use the Spack recipe. Until the Spack recipe becomes available, you can use the "Alternate recipe" described below.

reate file `packages.yaml` in `~/.spack/` containing:

```
packages:
  intel-oneapi-mkl:
    externals:
    - spec: intel-oneapi-mkl@2025.1 +cluster +ilp64 mpi_family=mpich
      prefix: /opt/intel/oneapi/
    buildable: false
  intel-oneapi-mpi:
    externals:
    - spec: intel-oneapi-mpi@2021.15
      prefix: /opt/intel/oneapi/
    buildable: false
```

Now, install VASP with Spack:

```
spack env create vasp
spack env activate vasp

spack add vasp@6.4.2 +openmp  %oneapi ^intel-oneapi-mkl ^intel-oneapi-mpi
spack install
```

Alternate recipe without Spack (Only to be used if Spack recipe is not available)

Source oneAPI environment:

```
source /opt/intel/oneapi/setvars.sh
```

Go to the VASP source directory and copy from the arch/ makefile.include template which needs to be modified:

```
cd vasp-directory/
cp arch/makefile.include.intel_ompi_mkl_omp makefile.include
```

Change the following parameters of `makefile.include`:

```
# compiler and linker flags
FC         = mpiifx -mprefer-vector-width=512  -xCORE-AVX512 -fiopenmp
FCL        = mpiifx -mprefer-vector-width=512  -xCORE-AVX512 -qmkl  -fiopenmp
# optimization
OFLAG      = -O3 -g -traceback
# MKL flags
MKL_PATH   = $(MKLROOT)/lib/intel64
BLAS       =
LAPACK     =
BLACS      = -lmkl_blacs_intelmpi_lp64
SCALAPACK  = -lmkl_scalapack_lp64 $(BLACS)

OBJECTS    = fftmpiw.o fftmpi_map.o fft3dlib.o fftw3d.o
OBJECTS_O1 += fftw3d.o fftmpi.o fftmpiw.o
OBJECTS_O2 += fft3dlib.o

INCS       =-I$(MKLROOT)/include/fftw
LLIBS      = $(SCALAPACK) $(LAPACK) $(BLAS)
OBJECTS_LIB = linpack_double.o

# For the parser library
CXX_PARS    = icpx
LLIBS       = -lstdc++

# comment out these lines
# SCALAPACK_ROOT = ...
```

Build the VASP "std" version

```
make DEPS=1 -j std
```

To clean VASP build, use:

```
make veryclean
```

7.20.3 Run Instructions

Depending on how you build VASP, load OneAPI env or activate Spack environment:

```
spack env activate vasp        # if installed with Spack

spack load vasp %oneapi        # if built without Spack
```

Go to the directory with VASP input files:

```
INCAR
KPOINTS
POSCAR
POTCAR
```

Setup environment variables:

```
export OMP_PROC_BIND=close
export OMP_PLACES=threads
export OMP_PROC_BIND=true
export I_MPI_FABRICS=shm:ofi
export I_MPI_DEBUG=5
export I_MPI_PIN_DOMAIN=omp
export I_MPI_PIN_ORDER=bunch
export I_MPI_PIN_CELL=unit

# Set number of OMP threads and MPI ranks. The following is an example.
# Adjust number of ranks and threads as required.

export OMP_NUM_THREADS=4
mpirun -n 8 /path/to/vasp_std
```

To obtain the output metric use:

```
# get time of DFT loop execution parsing the output file OUTCAR
#
time=$(grep 'LOOP+' OUTCAR | awk '{print $7}')
```

Highest performance is usually achieved with hyperthreading. As an example, use the following MPI x OMP decompositions:

- For 128c 6900-series machines MPIxOMP=24x20
- For 96c 6900-series machines MPIxOMP=24x16

7.21 WRF

The Weather Research and Forecasting (WRF) model is an advanced, open-source numerical weather prediction (NWP) system developed in the United States, primarily by NCAR and NOAA, for both atmospheric research and operational forecasting. WRF is highly flexible and can simulate weather phenomena at scales ranging from meters to thousands of kilometers, making it suitable for a broad range of meteorological applications.

Official Website: https://www.mmm.ucar.edu/models/wrf
GitHub Repository: https://github.com/wrf-model/WRF

7.21.1 Download Instructions

Please refer to the following link to download WRF 4.4 benchmark cases: https://www2.mmm.ucar.edu/wrf/users/benchmark/

For example, to download CONUS-2.5km dataset for WRF v4.4 or later, use https://www2.mmm.ucar.edu/wrf/users/benchmark/v44/v4.4_bench_conus2.5km.tar.gz

7.21.2 Spack Build Instructions

Prerequisite: Setup common SPACK environment (see Section 6.2).

Install WRF version 4.5.2 with AVX-512 optimization flags and Intel® oneAPI compilers using Spack:

```
spack install --add wrf@4.5.2 fflags="-xCORE-AVX512 -qopt-zmm-usage=high"
cflags="-xCORE-AVX512 -qopt-zmm-usage=high" build_type=dm+sm %oneapi ^intel-
oneapi-mpi
```

7.21.3 Run Instructions

```
 #!/bin/bash

# --------------------------------
# Load WRF and Prepare Benchmark
# --------------------------------

# Load the WRF module using Spack package manager
spack load wrf%oneapi

# Move to the WRF installation directory as managed by Spack
spack cd -i wrf

# Download the official CONUS-2.5km benchmark dataset from UCAR
wget
https://www2.mmm.ucar.edu/wrf/users/benchmark/v44/v4.4_bench_conus2.5km.tar.gz

# Extract the downloaded benchmark dataset archive
tar xzvf v4.4_bench_conus2.5km.tar.gz


# Change to the WRF test case directory for real-data simulations
cd test/em_real

# Remove any existing namelist.input to avoid conflicts
rm namelist.input

# Create symbolic links to the benchmark input files in the current directory
ln -sf ../../v4.4_bench_conus2.5km/* .

# Link the restart namelist as the main namelist.input for the benchmark run
ln -sf ../../v4.4_bench_conus2.5km/namelist.input.restart namelist.input

# Link the boundary condition file
ln -sf ../../v4.4_bench_conus2.5km/wrfbdy_d01 .

# Link the WRF restart file, renaming it to match the expected input filename
ln -sf ../../v4.4_bench_conus2.5km/wrfrst_d01_2019-11-26_23:00:00.ifort
wrfrst_d01_2019-11-26_23:00:00
```

```
# --------------------------------
# Environment Setup for WRF Run
# --------------------------------

# Remove stack size limit to prevent memory allocation errors
ulimit -s unlimited

# Restrict communication to shared memory only (optimized for single-node
runs). For multi-node runs over InfiniBand one can use I_MPI_FABRICS=shm:ofi
export I_MPI_FABRICS=shm

# Automatically determine optimal CPU pinning domains
export I_MPI_PIN_DOMAIN=auto

# Group MPI ranks together on adjacent cores for better locality
export I_MPI_PIN_ORDER=bunch

# Total number of MPI processes
export NUM_MPI_RANKS=128

# OpenMP threads per MPI rank (hybrid MPI+OpenMP configuration)
export OMP_NUM_THREADS=2

# Bind OpenMP threads close to their parent MPI process
export OMP_PROC_BIND=close

# Place OpenMP threads on physical cores (not SMT threads)
export OMP_PLACES=cores

# Increase OpenMP thread stack size to 512MB
export KMP_STACKSIZE=512M

# Set WRF's internal tile decomposition
export WRF_NUM_TILES=32

# Launch WRF using Hydra MPI with all environment variables mpiexec.hydra -
genvall -n $NUM_MPI_RANKS ./wrf.exe
```

Optimal MPI x OpenMP decompositions along with tile counts (WRF_NUM_TILES) vary across WRF test cases, CPU architectures, interconnect, and node counts configurations. For example, the CONUS-2.5km dataset achieves peak performance on a single-node Intel® Xeon® 6980P processor (128 cores/socket) using 128 MPI ranks, 2 OpenMP threads, and WRF_NUM_TILES=32. Refer to configuration details published with performance results for specific decomposition details.

7.21.3.1 Extracting Performance Metric

The benchmark reports timing data in rsl.out.0000 or `rsl.error.0000` log files. The performance metric (the average elapsed time per timestep) can be computed using the following command issued within the run directory after test run is completed:

```
grep 'Timing for main' rsl.out.0000 | tail -239 | awk '{print $9}' | awk -f
stats.awk
---
    items:       239
      max:         4.956210
      min:         0.943830
      sum:       247.612110
     mean:         1.036034
 mean/max:         0.209038
```

The above also shows a representative output to measure the mean time (in seconds) for a CONUS 2.5km 1-hour simulation with a time step of 15 seconds. Note that the number of timesteps are computed from the total time of the simulation divided by the timestep.

In this case, the total simulation time is 1 hour and the time step is 15 seconds resulting in 3,600/15 = 240 steps. The command will ignore the first timestep from the mean computation since the first time step typically takes much longer for setup purposes

The stats.awk file contains the following:

```
BEGIN{ a = 0.0 ; i = 0 ; max = -999999999  ; min = 9999999999 }
{
    i ++
    a += $1
    if ( $1 > max ) max = $1
    if ( $1 < min ) min = $1
}
END{ printf("---
\n%10s  %8d\n%10s  %15f\n%10s  %15f\n%10s  %15f\n%10s  %15f\n%10s  %15f\n","it
ems:",i,"max:",max,"min:",min,"sum:",a,"mean:",a/(i*1.0),"mean/max:",(a/(i*1.0
))/max) }
```