# Evaluation of the HPC Applications Dynamic Behavior in Terms of Energy Consumption

O. Vysocky[1], M. Beseda[1], L. Riha[1], J. Zapletal[1], V. Nikl[2], M. Lysaght[3] and V. Kannan [3]

1 IT4Innovations National Supercomputing Center, VSB-TUO, 17. listopadu 15/2172, 708 33, Ostrava, Czech Republic

2 Faculty of Information Technology, Brno University of Technology, Bozetechova 1/2, 612 66, Brno, Czech Republic

3 Irish Centre for High End Computing, Tower Building, Trinity Technology and Enterprise Campus, Grand Canal Quay, Dublin 2, Ireland

## Abstract

This paper introduces the READEX project tuning approach which exploits the dynamic application behavior and its potential for energy savings. The paper is focused on the manual applications evaluation from the energy consumption optimisation point of view. As an examples we have selected one complex application, the ESPRESO library, and two simplified applications from the ProxyApps benchmark tool suite.

ESPRESO contains many types of operations including I/O, communication, sparse BLAS and dense BLAS. The results show that static savings are 5.6–12.3 % and dynamic savings 4.7–9.1 %. The highest total savings for ESPRESO are 21.4 % as a combination of 12.3 % static savings and 9.1 % dynamic savings.

The ProxyApp applications Kripke and Lulesh, were presented for two configurations each. The first configuration of the Kripke saved 29.3 % energy, almost only by static tuning. On the other hand, the second configuration shows us only 18.8 % savings, but over a third of it was saved by dynamic switching CPU core and uncore frequencies. The Lulesh test cases saved 28.9 %, respectively 26.7 %.

**Keywords:** READEX, energy efficient computing, runtime tuning, MERIC, RADAR, HDEEM, RAPL, Haswell processor.

# 1 Introduction

The energy consumption of supercomputers is one of the critical problems for the upcoming exascale supercomputing era. The awareness of power and energy consumption is required on both software and hardware sides.

The Horizon 2020 project READEX (Runtime Exploitation of Application Dynamism for Energy-efficient eXascale computing) [1] faces the problem, and develops a toolsuite that explores a potential of dynamic tuning of the hardware and software parameters to increase the energy efficiency of High Performance Computing (HPC) applications.

READEX has four key applications, the ScoreP tool suite for applications instrumentation and measurements, the DynDetect for dynamism behavior detection, the Periscope Tuning Framework (PTF) for design time analysis and the READEX Runtime Library (RRL) for tuning the environment settings during the runtime [2, 3].

This paper deals with manual evaluation of the applications dynamic behavior and the respective energy savings. The the READEX toolsuite with automatic tuning without any knowledge about the application is expected to achieve approximately 75% of the savings achieved by the manual tuning as presented in this paper. For the manual evaluation other two tools were developed, the MERIC library to measure the applications behavior and RADAR python tool to analyse and report the measurement results.

The paper is divided into following sections. Section 2 introduces the static and dynamic tuning from the READEX point of view. Section 3 describes newly developed tools designed to manually analyse the applications. Section 4 contains a detailed example of the application analysis of the ESPRESO library running in four different configurations. In addition we have analysed two applications from the ProxyApps suite Kripke and Lulesh [4].

# 2 Static and Dynamic Tuning

There is a long list of parameters that can be tuned to reach energy savings. We divide them into three categories: *hardware parameters* (CPU frequency, uncore frequency[1]), *system software parameters* (number of OpenMP threads, thread placement) and *application-level parameters* that depends on the specific application.

We distinguish between static and dynamic tuning. In case of static tuning the optimal parameters for whole application are set at the beginning of the application. This settings brings static energy savings. By dynamic behavior we understand that for different sections, such as functions calls, of the selected application there exists a different optimal configuration. In the READEX terminology these code sections are called significant regions. Basic factors that influence the application are floating

[1]Uncore frequency refers to frequency of subsystems in the physical processor package that are shared by multiple processor cores. E.g., L3 cache or on-chip ring interconnect.

point computations, memory read and write access patterns, I/O operations and inter-process communication. To detect the significant regions manually a profiling tools such as Allinea MAP [5] are used.

The dynamism metrics that are currently measured and used in the READEX project are execution time, energy consumed and computational intensity of each region. The computational intensity, defined by the equation (1), is a metric that is used to model the behavior of an application based on the workload imposed by it on the CPU and the memory. Presently, computational intensity is calculated using the following formula and is analogous to the operational intensity used in the roofline model [6].

$$\textit{Computational Intensity} = \frac{\textit{Total number of instructions executed}}{\textit{Total number of L3 cache misses}} \tag{1}$$

In case of region time and energy consumption it is evident, that we follow the highest values, but it does not stand in case of computation intensity. High computational intensity probably indicate computational intensive region that may benefit when the frequency of CPU cores would increase, however low computational intensity may occur in memory intensive regions with high amount of L3 cache misses. Since this would cause increased traffic between the L3 cache and the main memory, it will be desirable to increase the uncore frequency.

To exploit the dynamic behavior for optimal energy savings one can dynamically tune these parameters during the runtime of the application to achieve lower power consumption. This has to be done for all significant regions independently. Dynamic savings we consider the amount of energy saved when each region tuned when application has been run in static optimal settings.

In the context of the READEX project, an application is termed to exhibit the following two types of dynamism:

- Inter-phase dynamism: This is when each phase of a phase region in the application exhibits different characteristics. This results in different values for the measured objective values and thus may require different configurations to be applied for the tuning parameters.

- Intra-phase dynamism: This is when each run-time situation of the significant regions in a phase region exhibits different characteristics and thus may need different configurations to be applied for the tuning parameters.

Due to the different localities of dynamism in an application, the dynamism metrics are measured and analysed from the respective of the entire run of the application as well as for all phases of the phase region in the application (inter-phase dynamism) and for all run-time situations of the significant regions in the application (intra-phase dynamism).

# 3 Applications Analysis

For the manual evaluation we are developing two tools. The first one is a MERIC, that allows us to manually instrument the code with probes in a way that we put probes at the beginning and at the end of each significant region. Using the probes we measure the application and store the results into output CSV files.

A RADAR report generator is the second tool, written in python, that evaluates the application dynamic behavior to see what is the potential of the dynamic tuning in terms of how much energy can be saved and what is the best parameters settings for each region.

## 3.1 MERIC

The MERIC [7] is a C++ library (with an interface for Fortran applications) that was developed for manual applications evaluation. It is able to measure the resource consumption and make changes to environment and hardware settings through external libraries. Every significant region must be annotated by MERIC_MeasureStart() and MERIC_MeasureStop() functions to enable the dynamic tuning and resources measurements per region. The list of measured resources contains the region runtime, energy consumption and performance counters (measured via perf_event or PAPI library). Counters are mainly used to evaluate the computational intensity of the regions, which is the key metric for dynamism detection as defined in Section 2.

The most important metric is the consumed energy. Contemporary Intel processors has the Running Average Power Limit (RAPL) interface [8] to measure the consumed Joules of each CPU and memory channels per compute node. MERIC uses the RAPL counters as an alternative to the High Definition Energy Efficiency Monitoring (HDEEM) system [9]. This system measures not only the consumption of CPUs and memory subsystem but also the energy consumed by the entire node (also called blade). The node energy baseline (the node energy consumption without the consumption of CPUs and memories) discriminates all cases where the energy significantly drops, but also runtime is notably longer. This extra information is a key HDEEM advantage when compared to the RAPL counters.

MERIC not only evaluate the regions, but it is also able to make changes to the environment settings during the runtime at the beginning of every region. The CPU core (CF) and CPU uncore (UCF) [2] frequencies are set using the cpufreq [10] and the x86_adapt [11] libraries, respectively. The number of active cores is set using the API to the OpenMP runtime by setting the number of threads per parallel region.

The measurement results are stored in CSV files for each region and settings separately and analysed afterwards by the RADAR tool.

[2]Uncore frequency refers to frequency of subsystems in the physical processor package that are shared by multiple processor cores. E.g., L3 cache or on-chip ring interconnect.

### 3.2 RADAR Report Generator

The RADAR report generator [7] is a tool for analysis of measurement results obtained by the MERIC. It is implemented in Python3 and its output is a LaTeX document and configuration file.

Its main goal is not only to present the results themselves, but to evaluate them and compute savings for both static and dynamic tuning. The savings can be computed for an energy consumption and runtime. In MERIC, an energy consumption can be measured via HDEEM or RAPL and an arithmetical intensity can be measured by perf_event or PAPI library. Both quantities can be evaluated with respect to both libraries simultaneously, so the comparison of these libraries can be included in the generated report.

The static tuning means, that we evaluate the energy consumption or the run-time of the whole application for default configuration and then we compare it with all the other configurations, so that we can find the optimal configuration for the entire application (i.e. the configuration with the lowest energy consumption or the shortest run-time). The dynamic tuning means, that we evaluate every nested significant region separately and then we compute additional savings with respect to the static optimum.

Using READEX project terminology, savings are evaluated both as the *intra-phase* and the *inter-phase* dynamism (see Section 2). The *phase region* can be given explicitly or the main region is automatically considered to be the phase region (and the application runs are then considered single phases). If the phase region is not the main region, all nested regions must be nested in the phase region.

The RADAR report itself consists of several elements. The first one is the *Overall application evaluation*, which contains the default configuration of tuning parameters, the optimal configuration for the entire application and both static and dynamic savings. The *Intra Phase Dynamic Tuning Evaluation* contains the optimal configuration and savings for every significant region nested in the main one (except the *phase region*). So, the best configuration for the main region (i.e. the static optimum) is considered the default one here. The *Inter-Phase Dynamic Tuning Evaluation* contains optimal configurations and computes savings for nested regions per phase of the phase region (e.g. per iteration in Conjugate Gradients solver). Beside tables, results are visualized in form of 2D plots and heatmaps.

## 4 Results

In this section we present selected applications from the ProxyApps benchmark suite as well as full fledge complex application, the ESPRESO FEM tool which includes FETI and Hybrid FETI solver. Only one of four experiments is described in more details due to limited length of this paper.

All experiments were measured on Taurus cluster installed at TU Dresden, that has nodes with two Intel(R) Xeon(R) CPU E5-E5-2680 v3 (12 cores) processors,

64 or 128 GB or RAM and the HDEEM system. The core and uncore CPU frequencies ranged between 1.2 – 2.5 GHz and 1.2 – 3.0 GHz, respectively, with the step size of 0.1 or 0.2 GHz. All tests were done on a single node and were ran several times per configuration to reduce measurement oscillations caused mainly due to network traffic. RADAR reports an average values across these measurements.

The test cases were run for all reachable environments settings to find the best static settings, the dynamic savings are enumerated from best static configurations of the regions, but without the switching overhead to demand settings.

## 4.1 ESPRESO Library

The ESPRESO library is a combination of Finite Element (FEM) and Boundary Element (BEM) tools and TFETI/HTFETI solvers. It supports FEM and BEM (uses BEM4I library) discretization for Advection-diffusion equation, Stokes flow and Structural mechanics. Real engineering problems are imported from Ansys Workbench or OpenFOAM. The postprocessing and visualization is based on the VTK library and Paraview including Paraview Catalyst for inSitu visualization.

In 2006 Dostál et al. [12] introduced a new variant of an algorithm called Total FETI (or TFETI) in which Dirichlet boundary condition is enforced by Lagrange multipliers (LM).

The HTFETI method is a variant of hybrid FETI methods introduced by Klawonn and Rheinbach [13] for FETI and FETI-DP. In the original approach a number of subdomains is gathered into clusters. This can be seen as a three-level domain decomposition approach. Each cluster consists of a number of subdomains and for these, a FETI-DP system is set up. The clusters are then solved by a traditional FETI approach using projections to treat the non trivial kernels. In contrast, in HTFETI, a TFETI approach is used for the subdomains in each cluster and the FETI approach with projections is used for clusters. The main advantage of HTFETI is its ability to solve problems decomposed into a very large number of subdomains [14]. We have ran tests with over 21 million subdomains organized into 17,576 clusters.

ESPRESO includes a highly efficient MPI communication layer [15] designed for massively parallel machines with thousands of compute nodes. The parallelization inside a node is done using OpenMP. Three versions of the solver are being developed:

- *ESPRESO CPU* uses sparse matrices and sparse direct solvers to process the system matrices.
- *ESPRESO MIC* is an Intel Xeon Phi accelerated version, which works with both sparse and dense representation of system matrices.
- *ESPRESO GPU* is a GPU accelerated version, which supports dense structures only [16]. Support for sparse structures using cuSolver is under development.

All versions can solve both symmetric (conjugate gradient (CG) solver) and non-symmetric systems (GMRES and BiCGStab).

**Hardware Tuning Parameters:** The dynamism of the ESPRESO library has been evaluated using the following hardware parameters:

- CPU Core frequency
- Number of OpenMP threads
- CPU Uncore frequency

### 4.1.1 Application Tuning Parameters

**Preconditioners:** The ESPRESO solver supports several preconditioners, that can be dynamically switched during the runtime of the iterative solver. The evaluated preconditioners are:

- Lumped preconditioner – uses *sparse* BLAS2 – matrix-vector multiplication,
- Dirichlet preconditioner – uses *dense* BLAS2 – matrix-vector multiplication.

**Stiffness Matrix Processing:** In FETI a stiffness matrix is a sparse matrix which in a general approach is processed by a Sparse Direct Solver (SPDS). In particular each stiffness matrix is factorized once during the preprocessing and then in each iteration a forward and backward substitutions (the solve routine of the SPDS) are called.

ESPRESO contains an alternative method based on the Local Schur Complement method (LSC) for stiffness matrix processing originally developed for GPGPU and Intel Xeon Phi accelerators, see [16]. In this method the preprocessing is more expensive as we have to calculate the LSC for each subdomain using SPDS. However the iterative FETI solver then uses dense matrix-vector multiplication using LSCs instead of more expensive solve routine of the SPDS. So the following methods will be evaluated:

- Sparse Direct Solver (SPDS) – is using the solve routine (in this case the Intel MKL PARDISO solver is used),
- Local Schur Complement (LSC) – is using the dense BLAS2 matrix-vector multiplication.

**FETI Method:** As mentioned above (see sec. 4.1) the ESPRESO solver contains two FETI methods: Total FETI ( better numerical behavioral) and Hybrid Total FETI (better parallel scalability). As of now the dynamic switching between these two methods is not implemented, however with certain effort this can be implemented into ESPRESO. So the dynamism for the following FETI methods can be evaluated:

- Total FETI method (evaluated in this paper)
- Hybrid Total FETI method

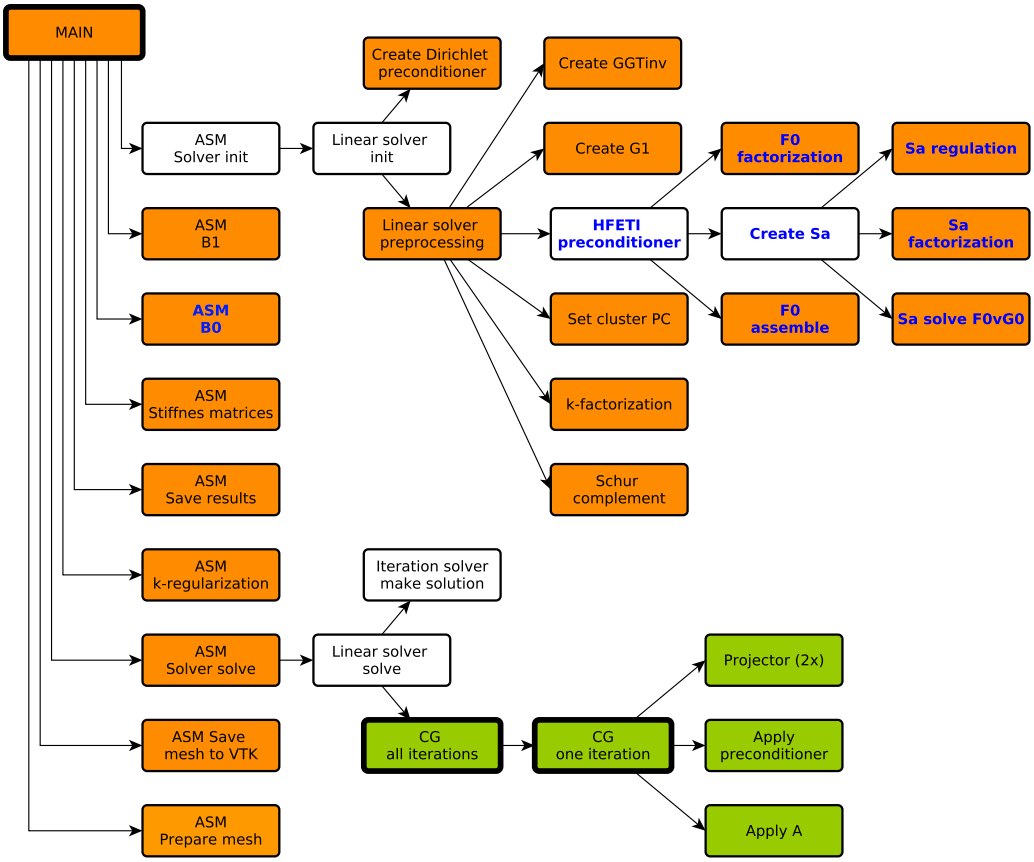


Figure 1: Diagram of the significant regions in the ESPRESO library as used for the dynamic savings evaluation in this section. The orange regions are called just once per iteration and therefore are used only for intra-phase dynamism evaluation. White regions are ignored because there are other significant regions nested in them. The green regions denotes the iterative solver (conjugate gradient (CG)) and provides an opportunity for inter-phase dynamism. The regions with names highlighted in bold are called only if Hybrid Total FETI is used.

### 4.1.2 RADAR Reports for ESPRESO

In this section we present a series of experiments, that have been executed with the ESPRESO library. For all runs the significant regions shown in Figure 1 have been used for measurements.

**Configuration 1: 1 node with 1 MPI process; 2 to 24 OpenMP threads**

- Method: Hybrid Total FETI
- Preconditioner: Dirichlet (dense)
- Stiffness matrix processing: PARDISO Sparse Direct Solver (sparse)
- Decomposition: 1x1x1 cluster; 8x8x8 subdomains per cluster; 11x11x11 elements per subdomain

| | **Default settings** | **Default values** | **Best static configuration** | **Static Savings** | **Dynamic Savings** |
|---|---|---|---|---|---|
| Energy consumption [J] , Blade summary | 24 threads, 3.0 GHz UCF, 2.5 GHz CF | 10678.9 J | 20 threads, 2.0 Ghz, 2.4 Ghz | 597.00 J (5.59%) | 880.75 J (8.74%) |
| Runtime of function [s] | 24 threads, 3.0 GHz UCF, 2.5 GHz CF | 29.73 s | 20 threads, 3.0 Ghz, 2.5 Ghz | 0.00 s (0.00%) | 0.7 s (1.52%) |

Table 1: Overall application evaluation for ESPRESO configuration 1

**Configuration 2: 1 node with 1 MPI process; 2 to 24 OpenMP threads**

- Method: Hybrid Total FETI
- Preconditioner: Dirichlet (dense)
- Stiffness matrix processing: Local Schur Complement method (Dense)
- Decomposition: 1x1x1 cluster; 8x8x8 subdomains per cluster; 11x11x11 elements per subdomain

| | **Default settings** | **Default values** | **Best static configuration** | **Static Savings** | **Dynamic Savings** |
|---|---|---|---|---|---|
| Energy consumption [J], Blade summary | 24 threads, 3.0 GHz UCF, 2.5 GHz CF | 23176.1 J | 24 threads, 1.8 Ghz, 2.0 Ghz | 1815.00 J (7.83%) | 994.91 J (4.66%) |
| Runtime of function [s] | 24 threads, 3.0 GHz UCF, 2.5 GHz CF | 86.38 s | 24 threads, 3.0 Ghz, 2.5 Ghz | 0.00 s (0.00%) | 0.56 s (0.64%) |

Table 2: Overall application evaluation for ESPRESO configuration 2

**Configuration 3: 1 node with 1 MPI process; 2 to 24 OpenMP threads**

- Method: Hybrid Total FETI
- Preconditioner: Lumped (sparse)
- Stiffness matrix processing: Local Schur Complement method (Dense)
- Decomposition: 1x1x1 cluster; 8x8x8 subdomains per cluster; 11x11x11 elements per subdomain

| | **Default settings** | **Default values** | **Best static configuration** | **Static Savings** | **Dynamic Savings** |
|---|---|---|---|---|---|
| Energy consumption [J], Blade summary | 24 threads, 3.0 GHz UCF, 2.5 GHz CF | 20508.9 J | 24 threads, 2.0 Ghz, 2.2 Ghz | 1589.70 J (7.75%) | 1017.92 J (5.38%) |
| Runtime of function [s] | 24 threads, 3.0 GHz UCF, 2.5 GHz CF | 86.38 s | 24 threads, 3.0 Ghz, 2.5 Ghz | 0.00 s (0.00%) | 0.54 s (0.74%) |

Table 3: Overall application evaluation for ESPRESO configuration 3

**Configuration 4: 1 node with 1 MPI process; 2 to 24 OpenMP threads**

- Method: Hybrid Total FETI
- Preconditioner: Lumped (sparse)
- Stiffness matrix processing: PARDISO Sparse Direct Solver (sparse)
- Decomposition: 1x1x1 cluster; 8x8x8 subdomains per cluster; 11x11x11 elements per subdomain

| | **Default settings** | **Default values** | **Best static configuration** | **Static Savings** | **Dynamic Savings** |
|---|---|---|---|---|---|
| Energy consumption [J] , Blade summary | 24 threads, 3.0 GHz UCF, 2.5 GHz CF | 6265.18 J | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 771.63 J (12.32%) | 499.2 J of 5493.6 J (9.09 %) |
| Runtime of function [s], Job info - hdeem | 24 threads, 3.0 GHz UCF, 2.5 GHz CF | 29.55 s | 22 threads, 3.0 GHz UCF, 2.5 GHz CF | 0.01 s (0.04%) | 0.82 s of 29.54 s (2.76 %) |

Table 4: Overall application evaluation for ESPRESO configuration 4

For this experiment we provide more detailed report as it has achieved the most significant static (12.32%) and dynamic savings (9.09%). In the Figure 2 are compared measurements with different core and uncore CPU frequencies, when using 18 OpenMP threads, with a pin at the best configuration from energy savings point of view. The results are also visualized in the heatmap Table 5.

| Uncore freq [GHz] / Core freq [GHz] | 1.2 | 1.4 | 1.6 | 1.8 | 2.0 | 2.2 | 2.4 | 2.6 | 2.8 | 3.0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.2 | 7,774.3 | 7,577.1 | 7,620.9 | 7,712.4 | 7,638.2 | 7,887.5 | 8,017.5 | 8,224.6 | 8,457.6 | 8,713.3 |
| 1.4 | 7,015 | 7,006.6 | 6,951.7 | 6,989.9 | 7,013.9 | 7,100.8 | 7,353.8 | 7,538.7 | 7,540.2 | 7,808.5 |
| 1.6 | 6,657.4 | 6,585.3 | 6,497.8 | 6,405.7 | 6,448.2 | 6,626.3 | 6,742.4 | 6,790.9 | 6,955.3 | 7,114.6 |
| 1.8 | 6,387.4 | 6,286.4 | 6,195.1 | 6,068.2 | 6,093.5 | 6,158.7 | 6,244.5 | 6,354.2 | 6,412.2 | 6,693.6 |
| 2 | 6,303.9 | 6,177.2 | 5,979.1 | 5,892.4 | 5,862.4 | 5,941.4 | 6,094.8 | 6,116.7 | 6,337.8 | 6,405.5 |
| 2.2 | 6,130.9 | 5,908.3 | 5,771.2 | 5,729.3 | 5,696 | 5,732.9 | 5,822.6 | 5,901.7 | 6,020.5 | 6,124.2 |
| 2.4 | 6,219.5 | 5,866.8 | 5,718.8 | 5,548.1 | 5,590.7 | 5,644.1 | 5,679.3 | 5,750.5 | 5,840.8 | 5,940.3 |
| 2.5 | 6,201.3 | 5,871 | 5,678.6 | 5,493.6 | 5,544.8 | 5,507.1 | 5,567.9 | 5,711.9 | 5,834.9 | 5,909.9 |

Table 5: Heatmap of energy consumption [J] for different core and uncore CPU frequencies when using 18 threads to reach the best static energy savings in ESPRESO configuration 4

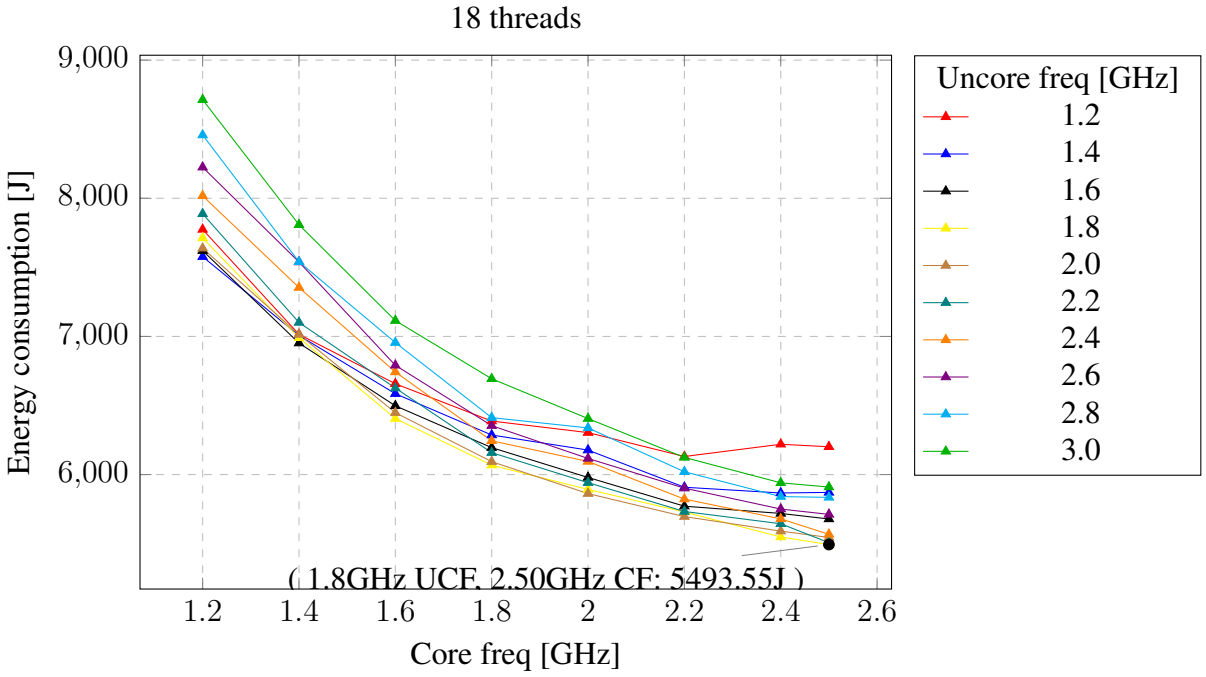


Figure 2: Energy consumtion when using 18 threads to reach the best static energy savings in ESPRESO configuration 4

In this configuration, when the best static settings is applied, the energy savings reach up to 12.32%, however the runtime of the application is extended. In the default settings the runtime takes 29.55 seconds, in the best static settings it takes extra 3.76 seconds, that means 12,72% longer. How the change of both, core and uncore

CPU frequencies, influence the runtime shows the heatmap Table 6, that contains runtime length for each settings when using 22 OpenMP threads (the optimum number of threads to reach the best runtime of this example).

| Uncore freq [GHz] / Core freq [GHz] | 1.2 | 1.4 | 1.6 | 1.8 | 2.0 | 2.2 | 2.4 | 2.6 | 2.8 | 3.0 |
|---|---|---|---|---|---|---|---|---|---|---|
| 1.2 | 60.727 | 58.527 | 56.92 | 55.813 | 55.095 | 54.454 | 53.975 | 53.517 | 53.023 | 52.953 |
| 1.4 | 54.553 | 52.357 | 50.638 | 49.499 | 48.44 | 48.232 | 48.128 | 47.066 | 46.746 | 45.975 |
| 1.6 | 49.634 | 48.031 | 46.226 | 44.984 | 44.163 | 44.134 | 42.875 | 42.999 | 42.153 | 41.551 |
| 1.8 | 46.755 | 44.614 | 42.516 | 40.89 | 40.118 | 40.101 | 39.405 | 38.555 | 38.611 | 38.067 |
| 2 | 44.359 | 41.281 | 39.536 | 38.114 | 37.025 | 36.431 | 36.352 | 35.646 | 35.404 | 35.005 |
| 2.2 | 41.544 | 39.015 | 37.503 | 36.214 | 35.329 | 34.505 | 33.865 | 33.714 | 32.878 | 32.779 |
| 2.4 | 39.773 | 37.102 | 35.515 | 33.968 | 33.053 | 32.318 | 31.931 | 31.36 | 31.106 | 30.634 |
| 2.5 | 38.339 | 35.518 | 34.437 | 33.109 | 32.481 | 31.425 | 30.983 | 30.433 | 29.862 | 29.536 |

Table 6: Time measurement heatmap [s] for static tuning ESPRESO configuration 4 when 22 threads were used to finish in the shortest runtime

Previous tables and figures are focused on static tuning of the ESPRESO library, searching the best settings to apply at the beginning of the runtime. However our work is focused on more detailed investigation, as introduced in the Section 2. In the following Table 7 is a list of significant regions that covers the ESPRESO configuration 4 runtime, as mentioned in the Figure 1.

Every significant region has its best settings, that leads to extra energy savings, if the settings would be changed at the region beginning. If we find out the best regions settings, it is possible to specify it as the MERIC input and it provides the dynamic changes.

| **Region** | **% of 1 phase** | **Best static configuration** | **Value** | **Best dynamic configuration** | **Value and dynamic savings** |
|---|---|---|---|---|---|
| Assembler–AssembleStiffMat | 14.32 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 733.73 J | 20 threads, 2.0 GHz UCF, 2.5 GHz CF | 731.22 J -2.51 J (0.34%) |
| Assembler–Assemble-B1 | 2.23 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 114.30 J | 2 threads, 2.2 GHz UCF, 2.5 GHz CF | 94.15 J -20.15 J (17.63%) |
| Cluster–CreateF0-FactF0 | 0.17 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 8.71 J | 6 threads, 1.6 GHz UCF, 2.5 GHz CF | 6.90 J -1.80 J (20.73%) |
| Assembler–SaveResults | 3.10 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 158.81 J | 2 threads, 1.2 GHz UCF, 2.5 GHz CF | 147.66 J -11.16 J (7.03%) |
| Assembler-K_-Regularization | 5.43 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 278.39 J | 2 threads, 1.8 GHz UCF, 2.5 GHz CF | 231.38 J -47.01 J (16.89%) |

| | | | | | |
|---|---|---|---|---|---|
| Cluster–CreateSa-SolveF0vG0 | 2.22 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 113.87 J | 6 threads, 2.0 GHz UCF, 2.5 GHz CF | 97.46 J -16.41 J (14.41%) |
| Create_GGT_-Inv | 0.28 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 14.23 J | 2 threads, 1.2 GHz UCF, 2.5 GHz CF | 8.92 J -5.31 J (37.34%) |
| Cluster–Kfactorization | 12.84 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 658.07 J | 24 threads, 2.0 GHz UCF, 2.4 GHz CF | 629.62 J -28.45 J (4.32%) |
| Assembler–SaveMeshtoVTK | 6.36 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 325.69 J | 2 threads, 1.2 GHz UCF, 2.5 GHz CF | 296.66 J -29.03 J (8.91%) |
| Cluster–CreateSa-SaFactorization | 1.95 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 99.93 J | 4 threads, 2.2 GHz UCF, 2.5 GHz CF | 80.85 J -19.08 J (19.09%) |
| Cluster–SetClusterPC | 1.46 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 74.70 J | 20 threads, 2.0 GHz UCF, 2.5 GHz CF | 74.54 J -0.16 J (0.22%) |
| Assembler–PrepareMesh | 12.53 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 641.88 J | 22 threads, 1.8 GHz UCF, 2.5 GHz CF | 639.39 J -2.49 J (0.39%) |
| Assembler–SolverSolve | 30.79 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 1578.06 J | 10 threads, 2.2 GHz UCF, 2.5 GHz CF | 1289.85 J -288.21 J (18.26%) |
| Assembler–Assemble-B0 | 0.26 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 13.28 J | 24 threads, 2.0 GHz UCF, 2.5 GHz CF | 12.51 J -0.77 J (5.81%) |
| Cluster–CreateG1-perCluster | 0.47 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 24.20 J | 14 threads, 2.2 GHz UCF, 2.5 GHz CF | 22.32 J -1.88 J (7.76%) |
| Cluster–CreateF0-AssembleF0 | 5.43 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 278.22 J | 24 threads, 2.2 GHz UCF, 2.2 GHz CF | 254.98 J -23.24 J (8.35%) |
| Cluster–CreateSa-SaReg | 0.17 | 18 threads, 1.8 GHz UCF, 2.5 GHz CF | 8.59 J | 8 threads, 2.0 GHz UCF, 2.5 GHz CF | 7.03 J -1.56 J (18.15%) |
| **Total value for static tuning for significant regions** | | 733.73 + 114.30 + 8.71 + 158.81 + 278.39 + 113.87 + 14.23 + 658.07 + 325.69 + 99.93 + 74.70 + 641.88 + 1578.06 + 13.28 + 24.20 + 278.22 + 8.59 = 5124.66 J | | | |

| | |
|---|---|
| **Total savings for dynamic tuning for significant regions** | 2.51 + 20.15 + 1.80 + 11.16 + 47.01 + 16.41 + 5.31 + 28.45 + 29.03 + 19.08 + 0.16 + 2.49 + 288.21 + 0.77 + 1.88 + 23.24 + 1.56 = 499.22 J of 5124.66 J (9.74 %) |
| **Dynamic savings for application runtime** | 499.22 J of 5493.55 J (9.09 %) |
| **Total value after savings** | 4994.33 J (79.72 % of 6265.18 J) |

Table 7: Intra-Phase Dynamism Evaluation, Blade summary energy consumption [J] of each significant region in ESPRESO configuration 4 runtime. For every region the table contains how many percent of energy the region consumes in compare to entire application, regions' the best configuration and the energy savings if the configuration would be applied during the runtime.

## 4.2 ProxyApps benchmark

A part of the project was evaluation of the subset of ProxyApps benchmark suit applications. For this paper we selected Kripke and Lulesh that produced the best total savings from the measured ProxyApps. Detailed reports of each application would be as long as the Section 4.1 about the ESPRESO library, therefore we mention only brief description of these applications, experiment configuration and the result configuration with its savings.

### 4.2.1 Kripke

Kripke is a simple, scalable, 3D Sn deterministic particle transport code (C++, MPI, OpenMP). Its main goal is investigating how different data-layouts affects instruction, thread and task level parallelism, and what the implications are on overall solver performance.

Kripke supports storage of angular fluxes (Psi) using all six striding orders (or nestings) of Directions (D), Groups (G), and Zones (Z), and provides computational kernels specifically written for each of these nestings. An asynchronous MPI-based parallel sweep algorithm is provided, which employs the concepts of Group Sets (GS), Zone Sets (ZS) and Direction Sets (DS). NEST parameter stands for the chosen nesting sequence, whereas the greatest energy savings were achieved for the GZD variant.

The Kripke experiments was measured in two configurations:
1) PX=PY=2, PZ=6, I=1000, NEST=GZD, ZX=ZY=ZZ=4, G=8
2) PX=PY=2, PZ=6, I=30, NEST=GZD, ZX=ZY=ZZ=32, L=8, D=32

The parameters ZX, ZY, ZZ further define the number of zones in each direction. It was found out that biggest static savings can be achieved with a lower number of zones, while more significant dynamic savings are possible with higher values. The numeric value G denotes the number of energy groups. Again, lower value seems to lead to better static savings. For the second instrumented run we dropped the –groups argument and chose additional ones. Firstly, the value L denotes the Legendre

expansion order. Secondly, the parameter D, which has to be a multiple of 8, defines the number of direction sets. Higher values lead to higher dynamic savings. The number I denotes the number of iterations and the parameters PX, PY, PZ denote the number of MPI processes in each direction with the restriction PX×PY×PZ = 24. The results are presented in the Table 8.

| Test case | Default settings | Default values | Best static configuration | Static savings | Dynamic savings | Total savings |
|---|---|---|---|---|---|---|
| Kripke 1 | 24 MPI proc, 1 thread, 3.0 GHz UCF, 2.5 GHz CF | 10608.08 J | 24 MPI proc, 1 thread, 1.2 GHz UCF, 1.6 GHz CF | 2987.48 J (28.16%) | 118.74 J of 7620.60 J (1.56 %) | 3106.22 J (29.28 %) |
| Kripke 2 | 24 MPI proc, 1 thread, 3.0 GHz UCF, 2.5 GHz CF | 14732.36 J | 24 MPI proc, 1 thread, 1.7 GHz UCF, 1.7 GHz CF | 1860.50 J (12.63%) | 906.50 J of 12871.86 J (7.04 %) | 2767.00 J (18.78 %) |

Table 8: Energy consumption of blade summary [J] of the Kripke application in the default settings and in the best static settings, also the best dynamic settings for evaluated significant regions, that in summary gives the total amount of saved Joules.

### 4.2.2 Lulesh

LULESH approximates the hydrodynamics equations discretely by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. It is a highly simplified application (C++, MPI, OpenMP), hard-coded to only solve a simple Sedov blast problem with analytic answers, but represents the numerical algorithms, data motion, and programming style typical in scientific C or C++ based applications.

The Lulesh was measured in these two configurations:
1) MPI_PROCS=1, S=32, I=20, B=20
2) MPI_PROCS=8, S=97, I=20, B=7

Parameters I and S denotes the maximal number of iterations, and the size of the domain, respectively, and B affecting the load balancing. There is the restriction that the total number of MPI processes has to be a cube (i.e. 1, 8, 27, . . . ).

During experiments we have found out that the parameter S has the most significant impact on runtime and power consumption. The best static energy savings were obtained with powers of two, while primes lead to the best dynamic savings. The parameter B has not proven to have a significant impact on the energy consumption, which might be caused by the fact that we only ran the program on a single node and the MPI communication overhead was not significant. The results for both configurations are presented in the Table 9.

| Test case | Default settings | Default values | Best static configuration | Static savings | Dynamic savings | Total savings |
|---|---|---|---|---|---|---|
| Lulesh 1 | 1 MPI proc, 24 threads, 3.0 GHz UCF, 2.5 GHz CF | 13307.00 J | 1 MPI proc, 24 threads, 1.2 GHz UCF, 1.3 GHz CF | 3790.04 J (28.48%) | 52.40 J of 9516.96 J (0.55 %) | 3842.44 J (28.88 %) |
| Lulesh 2 | 8 MPI proc, 3 threads, 3.0 GHz UCF, 2.5 GHz CF | 13636.90 J | 8 MPI proc, 3 threads, 1.2 GHz UCF, 1.6 GHz CF | 3519.32 J (25.81%) | 124.38 J of 10117.58 J (1.23 %) | 3643.70 J (26.72 %) |

Table 9: Energy consumption of blade summary [J] of the Lulesh application in the default settings and in the best static settings, also the best dynamic settings for evaluated significant regions, that in summary gives the total amount of saved Joules.

# 5 Conclusion

In this paper we have introduced the READEX project and its main idea of dynamic application behavior. This paper is mainly focused on the manual applications evaluation from the energy consumption optimisation point of view. This is the key step in exploring the possible gains of the runtime dynamic tuning. The tools for applications measurement and data analysis are also briefly introduced.

Section 4 presents the results of the manual evaluation of the ESPRESO library and two simplified applications from the ProxyApps benchmark tool suite. The ESPRESO library contains both FEM preprocessing tools and sparse iterative solvers based on FETI method. We have annotated more than 20 regions, which includes all types of operations including I/O, communication, sparse BLAS and dense BLAS. The tests also focus on the variation of the arithmetical intensity in form of sparse and dense data structures. Two key kernels of the FETI iterative solver the $F$ operator and the preconditioner can be represented by both dense and sparse matrices providing different type of workload. The results show that static savings are 5.6–12.3 % and dynamic savings 4.7–9.1 %. The highest total savings for ESPRESO are 21.4 % as a combination of 12.3 % static savings and 9.1 % dynamic savings. The ProxyApp applications Kripke and Lulesh, were presented for two configurations each. The first configuration of the Kripke saved 29.3 % energy, almost only by static tuning. On the other hand, the second configuration shows us only 18.8 % savings, but over a third of it was saved by dynamic switching CPU core and uncore frequencies. The Lulesh test cases saved 28.9 %, respectively 26.7 %, but almost none in dynamic tuning.

From the presented results it is possible to see, that one can save over 20 % of the consumed energy. The applications from the ProxyApps tool suite shows, that they are able to deliver significant static savings (over 25 %) over all instrumented regions. On the other hand, since these programs does not contain any extensive I/O regions and they were tested on a single node of the Taurus supercomputer, the further dynamic savings are rather low.

The situation may change when tests will be carried out on a higher number of nodes, where the applications may become communication bounded. However, the exhaustive search algorithm (sweeping over all combinations of tuning parameters) would have to be replaced by a more efficient minimization algorithm. A more detailed output causing some I/O overhead would lead to higher dynamism as well.

More complex applications such as the ESPRESO library has much higher potential for dynamic savings. In presented configurations the dynamic savings were up to 9.1 %.

| Application | Static savings [%] | Dynaminc savings [%] | Total Savings [%] |
|---|---|---|---|
| ProxyApps: Kripke, configuration 1 | 28.16 | 1.56 | 29.28 |
| ProxyApps: Kripke, configuration 2 | 12.63 | 7.04 | 18.78 |
| ProxyApps: LULESH, configuration 1 | 28.58 | 0.55 | 28.88 |
| ProxyApps: LULESH, configuration 2 | 25.81 | 1.23 | 26.72 |
| ESPRESO - configuration 0 | 5.6 | 8.7 | 14.3 |
| ESPRESO - configuration 1 | 12.3 | 9.1 | 21.4 |
| ESPRESO - configuration 2 | 7.8 | 4.7 | 12.5 |
| ESPRESO - configuration 3 | 7.8 | 5.4 | 13.1 |

Table 10: Overview of the static and dynamic energy savings achieved by the applications selected for this paper.

# Acknowledgement

The research leading to these results has received funding from the European Unions Horizon 2020 Programme under grant agreement number 671657.

# References

[1] J. Schuchart, M. Gerndt, P.G. Kjeldsberg, M. Lysaght, D. Horák, L. Říha, A. Gocht, M. Sourouri, M. Kumaraswamy, A. Chowdhury, M. Jahre, K. Diethelm, O. Bouizi, U.S. Mian, J. Kružík, R. Sojka, M. Beseda, V. Kannan, Z. Bendifallah, D. Hackenberg, W.E. Nagel, "The READEX formalism for automatic tuning for energy efficiency", *Computing*, pages 1–19, 2017, ISSN 1436-5057, URL `http://dx.doi.org/10.1007/s00607-016-0532-7`.

[2] Y. Oleynik, M. Gerndt, J. Schuchart, P.G. Kjeldsberg, W.E. Nagel, "Run-Time Exploitation of Application Dynamism for Energy-Efficient Exascale Computing (READEX)", in C. Plessl, D. El Baz, G. Cong, J.M.P. Cardoso, L. Veiga,

T. Rauber (Editors), *Computational Science and Engineering (CSE), 2015 IEEE 18th International Conference on*, pages 347–350. IEEE, Piscataway, Oct 2015.

[3] M. Lysaght, K. Iqbal, J. Schuchart, A. Gocht, M. Gerndt, A. Chowdhury, M. Kumaraswamy, P.G. Kjeldsberg, M. Jahre, M. Sourouri, D. Horak, L. Riha, R. Sojka, J. Kruzik, K. Diethelm, O. Bouizi, "D4.1: Concepts for the READEX Tool Suite", Technical report, ICHEC, TUD, TUM, NTNU, IT4I, Intel, gns, 2016.

[4] "Co-design at Lawrence Livermore National Lab:LLNL ASC Proxy Apps", URL `https://codesign.llnl.gov/proxy-apps.php`.

[5] "Allinea MAP - C/C++ profiler and Fortran profiler for high performance Linux code", URL `https://www.allinea.com/products/map`.

[6] S. Williams, A. Waterman, D. Patterson, "Roofline: An Insightful Visual Performance Model for Multicore Architectures", *Commun. ACM*, 52(4): 65–76, Apr. 2009, ISSN 0001-0782, URL `http://doi.acm.org/10.1145/1498765.1498785`.

[7] K. Venkatesh, R. Lubomir, G. Michael, C. Anamika, V. Ondrej, B. Martin, H. David, S. Radim, K. Jakub, L. Michael, "PRACE WHITEPAPER: Investigating and Exploiting Application Dynamism For Energy-Efficient Exascale Computing", 2017, URL `www.prace-ri.eu`.

[8] M. Hähnel, B. Döbel, M. Völp, H. Härtig, "Measuring Energy Consumption for Short Code Paths Using RAPL", *SIGMETRICS Perform. Eval. Rev.*, 40(3): 13–17, Jan. 2012, ISSN 0163-5999, URL `http://doi.acm.org/10.1145/2425248.2425252`.

[9] D. Hackenberg, T. Ilsche, J. Schuchart, R. Schöne, W. Nagel, M. Simon, Y. Georgiou, "HDEEM: High Definition Energy Efficiency Monitoring", in *Energy Efficient Supercomputing Workshop (E2SC)*, Nov 2014.

[10] D. Brodowski, "Linux CPUFreq", URL `https://www.kernel.org/doc/Documentation/cpu-freq/index.txt`.

[11] R. Schoene, "x86_adapt", URL `https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/X86Adapt`.

[12] Z. Dostal, D. Horak, R. Kucera, "Total FETI-an easier implementable variant of the FETI method for numerical solution of elliptic PDE", *Communications in Numerical Methods in Engineering*, 22(12): 1155–1162, jun 2006, URL `http://dx.doi.org/10.1002/cnm.881`.

[13] A. Klawonn, O. Rheinbach, "Highly scalable parallel domain decomposition methods with an application to biomechanics", *ZAMM*, 90(1): 5–32, jan 2010, URL `http://dx.doi.org/10.1002/zamm.200900329`.

[14] L. Riha, T. Brzobohaty, A. Markopoulos, O. Meca, T. Kozubek, "Massively Parallel Hybrid Total FETI (HTFETI) Solver", in *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '16. ACM, New York, NY, USA, 2016, ISBN 978-1-4503-4126-4.

[15] L. Riha, T. Brzobohaty, A. Markopoulos, M. Jarosova, T. Kozubek, D. Horak, V. Hapla, "Implementation of the efficient communication layer for the highly parallel total FETI and hybrid total FETI solvers", *Parallel Computing*, 2016.

[16] L. Riha, T. Brzobohaty, A. Markopoulos, T. Kozubek, O. Meca, O. Schenk, W. Vanroose, "Efficient implementation of total FETI solver for graphic processing units using Schur complement", *Lecture Notes in Computer Science*, 9611: 85–100, 2016.