H2020 FETHPC-1-2014

Enabling Exascale Fluid Dynamics Simulations
Project Number 671571

D2.2 – Initial report on the ExaFLOW algorithms, energy efficiency & IO strategies.

WP2: Efficiency improvements towards exascale.

Document Information

Deliverable Number	D2.1
Deliverable Name	Initial report on the ExaFLOW algorithms, energy efficiency & IO strategies
Due Date	31/03/2017 (PM18)
Deliverable Lead	UEDIN
Authors	Nick Johnson, UEDIN Michael Bareford, UEDIN Mirren White, UEDIN Niclas Jansson, KTH Adam Peplinski, KTH Jing Gong, KTH Nicolas Offermans, KTH Björn Dick, USTUTT Jing Zhang, USTUTT Patrick Vogler, USTUTT Satya P. Jammy, SOTON Christian T. Jacobs, SOTON Martin Vyzamal, IMPERIAL
Responsible Author	Nick Johnson, UEDIN, n.johnson@epcc.ed.ac.uk
Keywords	Efficiency; Efficient Implementation; Exascale; Energy Efficiency; Data compression
WP	WP2
Nature	R
Dissemination Level	PU
Final Version Date	31/03/2017
Reviewed by	Erwin Laure, KTH David Moxey, IMPERIAL Christian T. Jacobs, SOTON
MGT Board Approval	31/03/2017

Document History

Partner	Date	Comments	Version
UEDIN	17/01/17	Skeleton outlining contributions expected from partners.	0.1
UEDIN	27/02/17	Merge contributions from partners.	0.2
UEDIN	28/02/17	Implement consistent references; correct figures and placing; add chapter introductions.	0.3
UEDIN	28/02/17	Write conclusions, introductions; update author list.	0.4
UEDIN	25/03/17	Incorporated reviewer feedback	0.5
KTH	31/03/17	Final version for submission	1.0

Executive Summary

This deliverable reports on the work carried out in the first half of the ExaFLOW project as part of WorkPackage 2. Specifically, it reports on actions taken to improve the efficiency of the implementations of the algorithms used for Computation Fluid Dynamics (CFD) as developed in WorkPackage 1. The aims of this deliverable are to:

- Report on the progress of efforts undertaken to formulate efficient implementations of algorithms, primarily stemming from WorkPackage 1, and,
- Look ahead to work which will be undertaken in the second half of the project to ensure completion of the objections as set out in the project proposal and description of work.

In this deliverable, efficiency improvements are any changes which improve the performance in terms of time to solution and energy to solution. In the former case, this includes methods of code generation where time to solution can be reduced by having the exact code to resolve the problem generated by machine rather than coded by hand.

The work is structured into four primary strands, aligned with the four tasks and a summary of the work in each strand is follows.

Implementation of algorithms in co-design applications.
Three different activities have been undertaken in this area:

- Use of code generation software, where code is generated from a description of the input problem and target architecture where it is to be run and then executed. This method has two principal advantages over traditional methods; being able to transform the input problem and the solution method with knowledge of the execution target which may provide an opportunity for cross-layer optimisation and, reducing the time required to code the problem and solution method and reducing the requirement for coding on the end user. The OpenSBLI software package is an implementation of this method and results show good agreement of solutions to standard problems obtained by this software with those obtained other, traditionally coded software.
- Implementation of algorithms for Adaptive Mesh Refinement in Nek5000 and extension of those developed in earlier projects. In this case, algorithms developed by partners in previous projects are refined and re-implemented to provide further performance and functionality. In some cases, external library software is used in place of in-built routines to provide stability, speed and functionality, reducing the time to solution.
- Implementation of an alternative communications library for Nek5000 and Nektar++. Both Nek5000 and Nektar++ can use the GS library for performing MPI-based communications. Whilst robust, this library does not take advantage of newer MPI features such as single-sided communications. The ExaGS library is offered as a replacement. It is

implemented in UPC and therefore able to make efficient use of RDMA mechanisms on machines which support it.

Energy Efficiency

- Three separate studies of the energy efficiency of applications were performed. The first two used the built-in power monitoring features of the Cray system available to the project in UEDIN and USTUTT. Whilst studying separates code the two studies showed that changing the operating frequency of nodes can result in a significant savings in energy to solution without excessively increase time to solution. In one case, changing CPU frequency to 1.7GHz is optimal when measured using the Energy Delay Product (EDP). Doing so reduces the energy demand by 19.2% compared to the default setting of 2.5GHz while increasing the runtime by 12.6%. If compared to turbo mode (3.4 GHz), energy savings of 34% are possible while increasing the runtime by 20%.
- The third study used custom hardware to look at individual hardware components such as Disk, DRAM and CPU to investigate if there was potential for optimisation to further reduce energy consumption.

I/O

A small part of this workpackage is tied to the algorithms for data reduction in workpackage 1, focussing on the measurement and implementation of algorithms for lossless and lossy compression of CFD specific data.

Technology Watch

A final task in this workpackage is to keep abreast of emerging technologies which will likely have a positive effect on the performance of CFD codes. An outline of the various improvements which would likely have positive effects were discussed in deliverable D2.1. Specifically, it is likely that Xeon Phi processors and memory with large bandwidth and more complex structures will be available to test with the co-design applications in the second half of this project.

1 Contents

2 Table of Figures

3 Table of Acronyms

Acronym	Definition
AMG	Algebraic Multi-grid
AMR	Adaptive Mesh Refinement
ATX	Advanced Technology eXtended
BL	Baseline
CFD	Computational Fluid Dynamics
CPU	Central Processor Unit
D	Deliverable
DNS	Direct Numerical Simulation
DRAM	Dynamic Random Access Memory
DVFS	Dynamic Voltage Frequency Scaling
EDP	Energy-Delay Product
GS	Gather-Scatter
HDF5	Hierarchical Data Format version 5
I/O (IO)	Input/Output
MMS	Method of Manufactured Solutions
MPI	Message Passing Interface
NetCDF	Net Common Data Format
SEM	Spectral Element Method
SSD	Solid State Disk
UPC	Unified Parallel C
WP	Work-package
XML	Extensible Mark-up Language

4 Introduction

This deliverable reports on work performed in work-package 2 in the first half of the ExaFLOW project. The focus of this work-package (WP) is different to that of WP1. Whereas WP1 focuses on the derivation of new algorithms which can scale to the size expected of future exascale machines, the motivation for the work in this WP is to make the implementations of those algorithms efficient in order to give the most efficient use of new machines.

As part of the overall efficiency package, we considered code implementations such as: changing programming model; using libraries rather than hand-crafted code and generated code; system aspects such as CPU frequency control and choice of algorithm to minimise energy and runtime; and reducing data volumes to ease IO bottlenecks.

The remainder of this deliverable is as follows:

- In Section 5 we report on efficient implementations of algorithms in the co-design applications and support libraries, including code-generation as an alternative to traditionally written code.
- In Section 5.4 we report on work to measure, analyse and understand the energy consumption of the co-design applications, with a view to reducing the consumption both via code optimisation and adjustment of system features.
- In Section 7 we report on implementation of the algorithms researched in WP1 relating to data compression and I/O reduction improvements.
- Finally, in Section 8 we conclude with pointers to future work to be carried out in the second half of the project.

5 Implementation of algorithms in co-design applications

In this section we examine different implementations of CFD algorithms, which may offer solutions to the problem of scaling current codes and problems to a size which would make efficient use of an exascale machine.

In Section 5.1 we examine the code generation aspects of OpenSBLI as a solution to writing code that will make efficient use of a particular architecture. In Section 5.2 we examine work done to implement h-type Adaptive Mesh Refinement (AMR) in Nek5000, including the required pre-conditioners to support this. In Section 5.2.3 we examine work on Algebraic Multi-grid methods and finally, in Section 5.3 we report on initial work to improve the underlying communication library used by both Nektar++ and Nek5000 to allow for improved scaling.

5.1 Code generation for CFD

Version 1.0.0 of the OpenSBLI code has been released on GitHub under the GNU General Public License **[6].** This comprises code generation techniques that allow users to write the equations they wish to solve as high-level expressions - the model code that performs the finite difference approximations is then generated automatically. This helps to future-proof models as new exascale-capable architectures become available. It also introduces a separation of concerns between domain specialists, numerical modellers, and HPC experts, allowing better maintainability and extensibility of the codebase.

Verification and validation of OpenSBLI has been accomplished through a suite of test cases, such as the Taylor-Green vortex problem. In this problem, vortex stretching and turbulence are simulated by solving the 3D compressible Navier-Stokes equations on approximately 16 million grid points. A visualization of the results is shown in Figure 1.

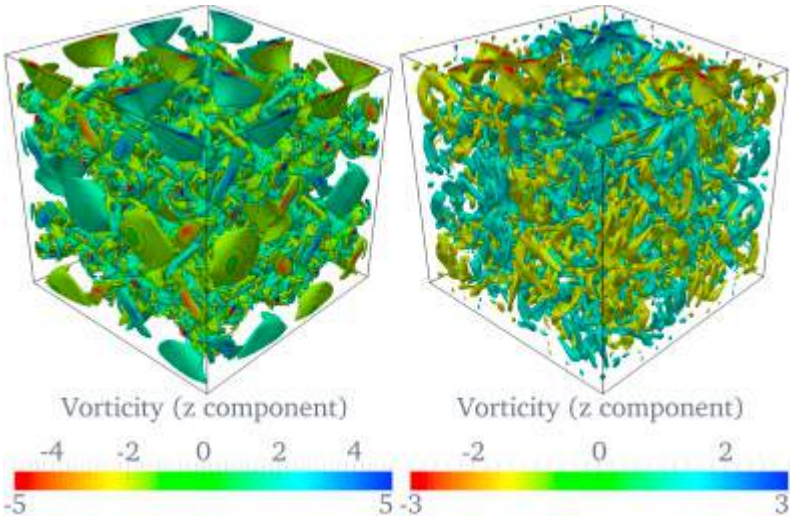

Figure 1: Visualisation of the solution to the Taylor-Green vortex problem in 3D.

The numerical results from OpenSBLI with respect to enstrophy and kinetic energy agree very well with existing DNS data, as shown in Figure 2.

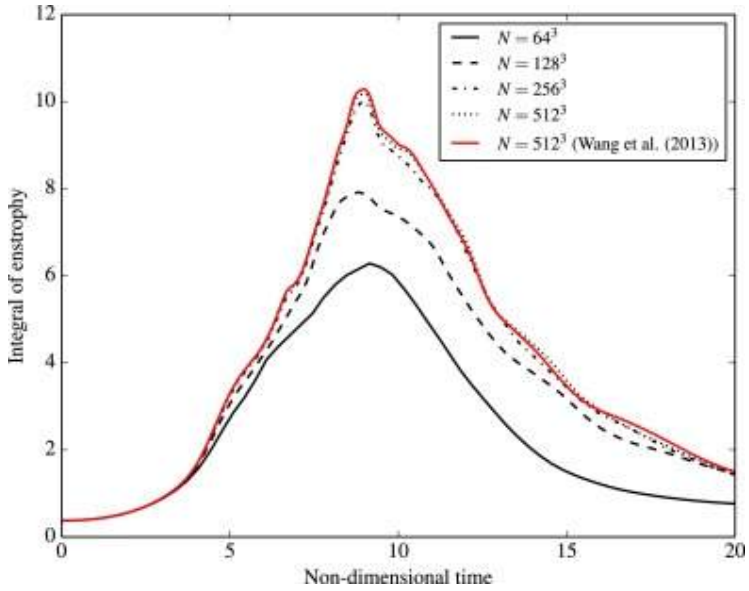

Figure 2: Comparison of enstrophy and kinetic energy results from OpenSBLI to other DNS data.

The method of manufactured solutions (MMS) has also been used to highlight the flexibility of OpenSBLI and its code generation techniques, by running a convergence analysis with finite difference schemes of various orders simply by changing a single parameter in the problem specification/setup file. The generation of the code that implements the finite differencing scheme of arbitrary order *n* is generated automatically. The results in Figure 3 show convergence of the solution field 'phi' for schemes of order 2 up to 12, thereby verifying the correctness of the code:

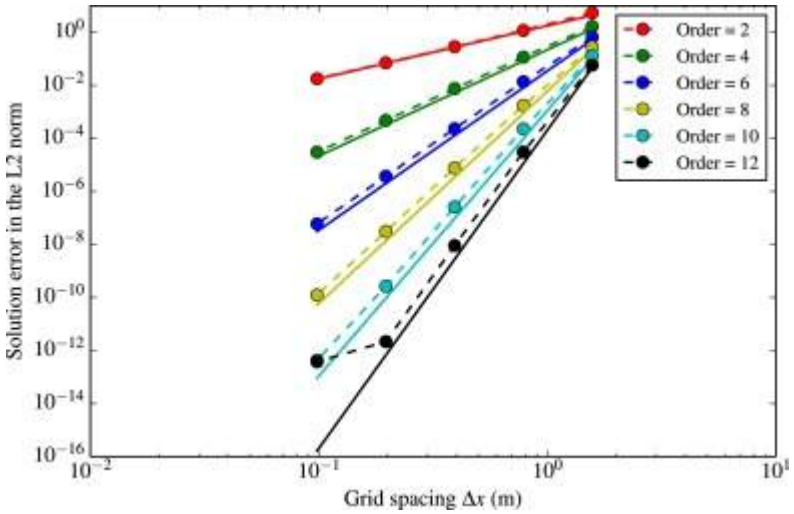

Figure 3: Convergence of the solution field 'phi' for schemes of order 2 up to 12.

Further details on OpenSBLI can be found in a journal article recently published by Jacobs *et* al **[7]**.

OpenSBLI and its dependencies have been successfully installed on both the ARCHER (UEDIN) and Hazel Hen (USTUTT) supercomputing facilities in preparation for performance analysis and large-scale Use Case runs for WP3. OpenSBLI scales well both strongly and weakly up to tens of thousands of cores on ARCHER, as reported in another recently-accepted journal article **[8].**

The development and evaluation of five different finite difference algorithms has also taken place in OpenSBLI. These are characterized by varying degrees of computational and memory intensiveness brought about by storing derivatives in memory vs. re-computing them on the device. Recent work has focused on their evaluation on multicore CPUs, and it has been shown that by storing the first derivatives of the velocity components as thread/process-local variables (rather than as global work arrays over the whole grid) a speed-up of ~2 relative to traditional CFD algorithms (in which all field values are stored in global work arrays) can be attained, as shown in Figure 4.

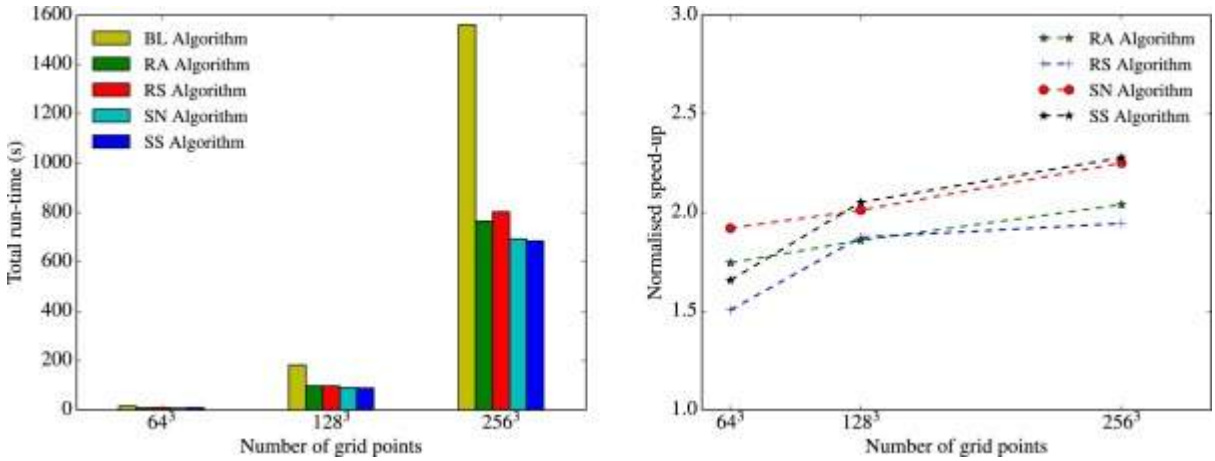

Figure 4: The total run-time (left) and normalised speed-up of 5 different finite difference algorithms in the context of a Taylor-Green vortex simulation. The Baseline (BL) algorithm which stores all derivatives in global work arrays is the worst algorithm in t terms of run-time, while the Store Some (SS) algorithm which stores the first derivatives of velocity components as thread/process-local variables is the fastest algorithm.

Progress has also been made on integrating the error indicators for finite difference methods into the OpenSBLI code, and we are aiming to present this work at the ParCFD 2017 conference.

5.2 Adaptive Mesh Refinement

Here we describe the code developments closely related to the WP1 task (1.1) concerning grid adaptivity. The goal is to introduce the h-type Adaptive Mesh Refinement (AMR) framework in CFD solvers based on the Spectral Element Method (SEM). The presented work was done with the Nek5000 code and consists of the development of different AMR framework components:

- reimplementation of the tools for grid management;
- implementation of the pressure pre-conditioner for nonconforming meshes;
- algebraic multi-grid solvers.

5.2.1 Grid management

The most fundamental operations of AMR on parallel computers are grid modification and partitioning. In context of SEM, in which discretisation is based on a decomposition of a computational domain into a number of non-overlapping, high-order sub-domains called elements, these operations mean changing the polynomial order in particular element (p-refinement) or splitting the element into the smaller one (h-refinement), and deriving a proper element-to-process mapping.

The work on h-type AMR framework for Nek5000 code was started within EU project CRESTA, where these basic tasks were implemented using existing external libraries. As h-refinement modifies element connectivity, a special grid manager is required to perform local refinement/coarsening and to build globally consistent mesh. For this task the p4est library **[1]** has been chosen, as it is designed to manipulate domains composed of the multiple, non-overlapping logical cubic subdomains, which can be represented by a recursive tree structure. For grid partitioning ParMETIS **[2]** was used. It adapts a variety of algorithms for partitioning and repartitioning of unstructured graphs, however within CRESTA only the repartitioning from scratch strategy was extensively studied. It guarantees the best mesh decomposition, but is the most computationally intensive. The more detailed description of implementation and scaling tests can be found in **[3]** and in this document we will focus only on modifications done as part of ExaFLOW.

The first important change is the upgrade of p4est from version 0.3.4 to 1.1, as the new version offers a significantly improved interface. However, this forced us to change a number of key routines developed within CRESTA, so we finally decided to re-implement the whole code. One of the crucial modifications concerns the global numbering of degrees of freedom. The old implementation relied in this case on p4est numbering, which was inconsistent with Nek5000 standard and made it difficult to combine h- and p-type refinement. In the new implementation we perform global ordering of vertices, edges, faces and elements that are later used to evaluate global node numbering. This method is much more flexible and closer to Nek5000 standard. Other important improvements are cleaner algorithms; giving a framework to develop p-refinement techniques for future developments in WP1; allowing more control over the coarsening process; and additional tests performed after p4est refinement phase. These tests avoid potentially costly solver restarts in situations where they are not needed. In the case of ParMetis, we focused on this library's native adaptive repartitioning strategy combining diffusive and remapping schemes. Unlike the partitioning from scratch strategy, this is a trade-off between the quality of mesh decomposition and the computational time, and has to be taken into account, as the partitioning from scratch was found to be the most important bottleneck in the context of the code strong scaling.

We have also upgraded our nonconforming version of Nek5000 solver, moving from old SVN revision 1050 and adapting to the new code development workflow on GitHub.

5.2.2 Implementation of the pressure pre-conditioner for nonconforming meshes

The AMR framework implemented into Nek5000 during CRESTA supported only the advection-diffusion equation. One of the main goals of ExaFLOW is to extend this implementation to the full nonlinear Navier-Stokes equations for incompressible flows. In this case, one has to perform costly pressure calculations which require efficient pre-conditioners for nonconforming meshes to limit the number of iterations. To support all Nek5000 features, we upgraded both steady and time dependent solvers.

First we implemented the nonconforming version of the pressure pre-conditioner for steady state calculations. It uses Uzawa decoupling with the full inverse of the Helmholtz operator evaluated by nested velocity iterations. Following previous development (Paul Fischer; private communication) we modified routines responsible for action of the inverse mass matrix operator and those enforcing the solution continuity across element boundaries. As most calculation is performed on velocity mesh, no modification to the gather-scatter operator was necessary. This implementation was tested with the Kovasznay flow **[4]** which is the two-dimensional flow-field behind a periodic array of cylinders. It is a perfect test case as it provides an analytical solution that allows for calculation of the exact error of the numerical solver. The error of the stream-wise velocity component for the unrefined and refined simulations is presented in Figure 5 and Figure 6 respectively showing the error reduction by two orders of magnitude for the refined grid.

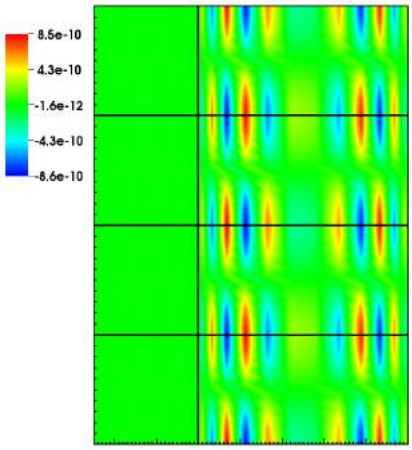

Figure 5: The error of the stream-wise velocity component for the conforming mesh. Black lines show element boundaries.

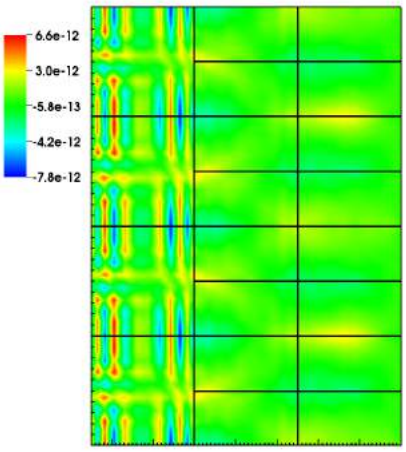

Figure 6: The error of the stream-wise velocity component for the nonconforming mesh. Black lines show element boundaries.

Next we upgraded the pressure pre-conditioner for time dependent flows based on the additive overlapping Schwarz method (further discussed in Section 5.2.3.4). This pre-conditioner together with necessary algorithm modifications and performed test cases are extensively discussed in the WP1 deliverable 1.1, so we will focus here on the major code changes only. The most significant modifications are related to redefinition of both the restriction/prolongation operators for the local problem and the coarse base functions used for assembly of the coarse grid operator.

The coarse base functions have to be adjusted to remove the hanging nodes from consideration, which makes functions shape dependent on the configuration of the refined region. As not all of the possible patterns can be represented as a simple tensor-product, we define and store all necessary components of the base functions (5 in 2D and 15 in 3D) at the initialisation step, and after each mesh refinement step we perform proper assembly of the functions taking into account relative position of children faces with respect to their parents and other elements in the neighbourhood. Finally, the coarse grid operator is formed and the coarse grid solver is restarted. An example shape of the coarse base function is presented in Figure 7.

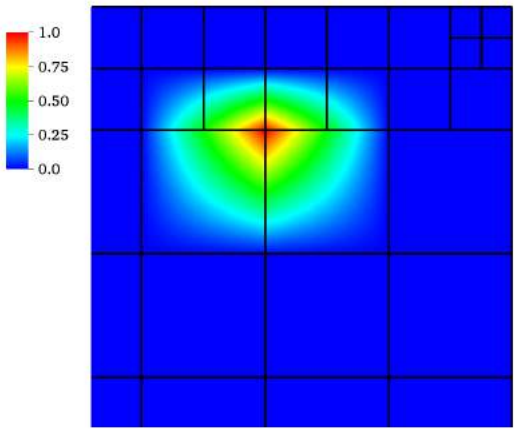

Figure 7: Exemplary shape of the coarse base functions for the nonconforming mesh. Element boundaries are marked by black lines.

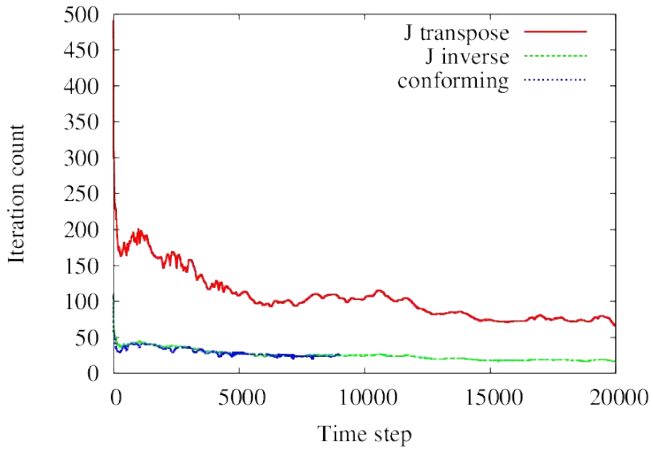

Figure 8: Iteration count of the pressure solver as a function of time step for conforming and nonconforming setups of 3D lid driven cavity. The nonconforming setup uses two different definitions of the local child-to-parent mapping operator: J^{-1} and J^T.

For simplicity, the restriction/prolongation operators for local problems are built using the gather-scatter communicator for the velocity mesh with the proper node mapping, exchanged interpolation operator and without edge interpolation performed. The analogy with the velocity mesh operator was used as well to construct local interpolation operators, so the child-to-parent mapping was performed by a transpose of the parent-to-child mapping matrix $\boldsymbol{J}$. This implementation was tested with a lid-driven cavity setup, showing similar convergence rate of the conforming and nonconforming solvers for 2D simulation and significant increase of pressure iterations for the nonconforming solver in 3D case. Further investigation showed the definition of the child-to-parent mapping to be responsible for the reduced convergence rate, and a new operator closer to $\boldsymbol{J^{-1}}$ rather than $\boldsymbol{J^T}$ was proposed and implemented. The new operator was found to be superior to $\boldsymbol{J^T}$, reducing the number of pressure iterations for the nonconforming solver to the level of conforming solver, as can be seen in Figure 8.

Combining all implemented tools, we were able to perform the full AMR simulation of the 2D backward-facing step with the Reynolds number $Re = 450$. For this simulation we used spectral error indicator developed by C. Mavriplis **[5]** with the refinement and de-refinement criteria set to 10^{-4} and 10^{-6} respectively. The simulation started with the smallest possible resolution mesh and zero initial condition, and was allowed to evolve freely for 20000 steps with the mesh refinement performed every 100 steps. The restriction on the CFL condition was set to 0.3 and the simulation was run on 4 processors. The initial (before the first refinement took place at 100 steps) and final mesh structures together with the error indicator and the stream-wise velocity component are presented in Figure 9, Figure 10, Figure 11 and Figure 12. In all plots the element boundaries are marked by black lines.

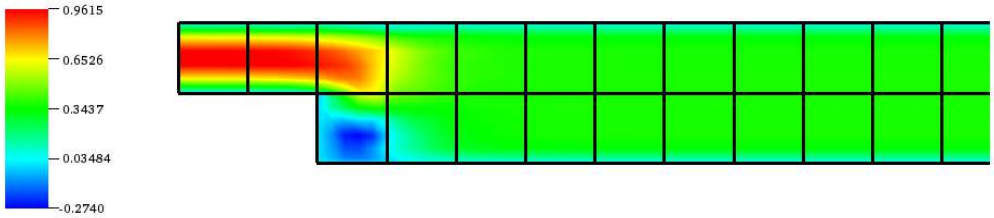

Figure 9: Initial stream-wise velocity component for the backward-facing step simulation.

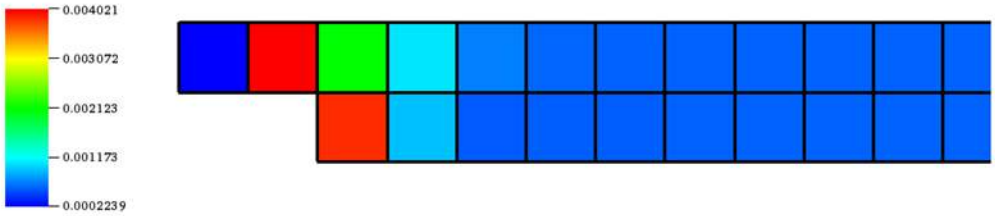

Figure 10: Initial error estimator for the backward-facing step simulation.

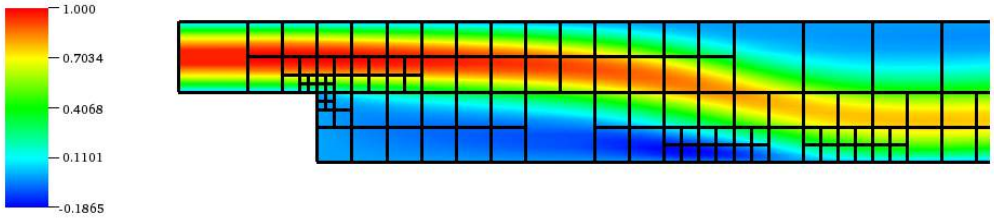

Figure 11: Final stream-wise velocity component for the backward-facing step simulation.

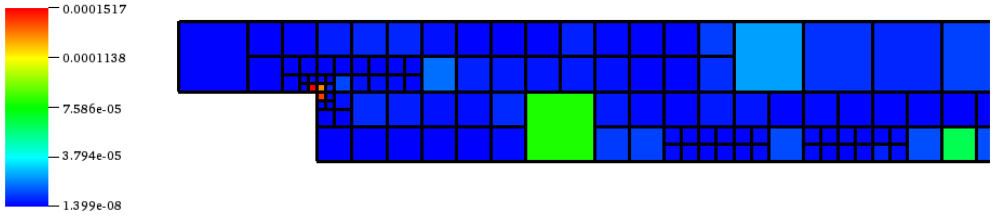

Figure 12: Final error estimator for the backward-facing step simulation.

As can be seen, there were three refinement levels added during the simulation, reducing the maximum value of error indicator 25 times and confining the error to the proximity of the sharp step edge. At the same time the total number of

elements raised from 72 to 246. The overall cost of mesh adaptivity with 199 calls to error indicator and the mesh being regenerated 135 times is negligible (1.12 sec) compared to the total simulation runtime (186.5 sec) proving AMR to be very efficient for small test cases run on small (4) number of cores. However, for the big number of cores the communication intensive restart of the coarse grid solver can set a constraint on Nek5000 scalability. This can be an additional limitation as results of CRESTA showed the grid partitioning performed by ParMETIS to be a major bottleneck at the strong scaling limit (1 element per core). That is why we currently consider different strategies for both grid partitioning and coarse grid solver setup. Some of these issues are addressed in the next paragraph.

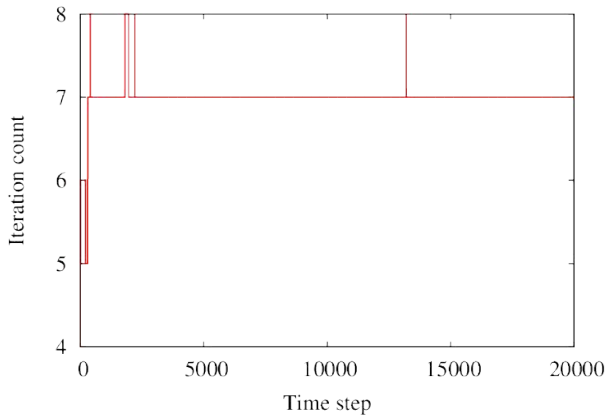

Figure 13: Number of velocity iteration as a function of time step for the backward-facing step simulation.

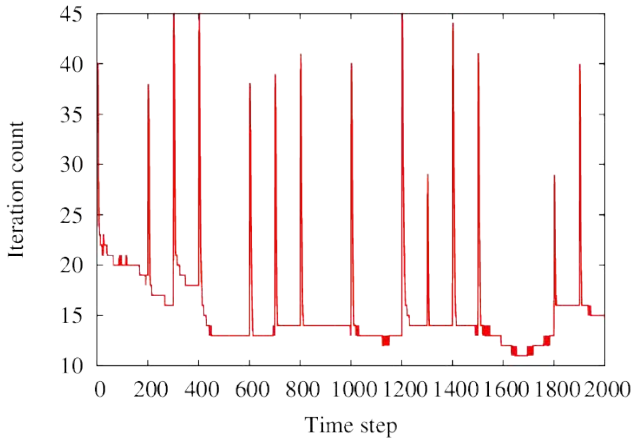

Figure 14: Number of pressure iterations as a function of time step for the backward-facing step simulation. To make plot readable we presented first 2000 time steps only.

The last aspect to be investigated was the influence of mesh refinement on the solver iteration count, as this should be considered as an additional cost of AMR that is not included in timing of the mesh regeneration routines. Figure 14 presents the number of iterations for the velocity solver as a function of time step, and shows almost no influence of the mesh refinement on solver performance. On the other hand, the pressure solver shows strong variations in the number of iterations every time the mesh is regenerated, as can be seen in Figure 13. The iteration count rises rapidly from about 15 to 45 and then drops back within a few

time steps. This behaviour is identical to the initialisation phase of the conforming solver, and its source is not fully clear yet.

5.2.3 AMG Pre-conditioners

The major source of stiffness when solving the Navier-Stokes equations comes from the pressure equation, which requires an efficient preconditioning strategy. The method chosen for Nek5000 is called additive Schwarz **[12]** and the pre-conditioner can be expressed as:

$$M_0^{-1} = R_0^T A_0^{-1} R_0 + \sum_{k=1}^{K} R_k^T A_k^{-1} R_k$$

where K is the number of spectral elements and R_k and R_0 are restrictions operators. This pre-conditioner can be seen as the sum of the global coarse grid operator (subscript 0) and local subdomain operators (subscript k). The present report focuses on the solution of the coarse grid operator, A_0, which is the finite element Laplace operator with linear base functions, built on the elements' vertices (independently on the polynomial order and inner points within each element). Two methods are available in Nek5000 to solve this problem. The first one is a sparse basis projection method, called XXT **[14]**. The second method uses an algebraic multigrid method (AMG), which is more efficient for massively parallel (P > 1e4) large simulations (K > 1e5) **[11]**.

The convergence rate of iterative solvers usually stalls after a certain number of iterations because of the slow decay of low frequency errors. Multigrid solvers tackle this issue by transferring the problem to a coarser grid, where low frequency errors will be damped more rapidly. This step is applied recursively until the problem is sufficiently small to be solved directly. Two main multigrid methods exist: the geometric multigrid and the algebraic multigrid. The first method builds a geometrically explicit coarser grid, while the latter builds a coarser algebraic operator. The AMG implemented in Nek5000 is divided into two parts: a setup and a solver. In the following section, we describe the algorithms for the AMG setup and solver. We also present the implementation of an alternative and faster way to perform the setup, which does not modify the solver part. In the deliverables for WP3, a comparison of the results with both methods will be presented.

The current AMG setup is performed by an external, serial MATLAB code and can be decomposed into three main steps: coarsening of the operator A_0, computation of the interpolation operators between the various levels, and computation of a smoother on each level. The operator A_0 should be first written out to binary files by running Nek5000 with proper pre-processing flags. The setup needs to be performed once per computational grid and does not depend on the polynomial order of the spectral elements.

5.2.3.1 AMG Coarsening

Given an initial operator A, of dimension n x n, the coarsening operation defines a hierarchy of coarse operators. One requirement for the present AMG setup is to have a 1 x 1 (i.e. a scalar) operator at the coarsest level. At each level, the coarse

grid simply refers to a subset of the original unknowns, called *C-variables*. The remaining unknowns are termed *F-variables*. In the case of the AMG setup, a variable is associated to a vertex of the domain and to a given line of the operator. The partitioning between the C- and F-variables is done using the Ostrowski coarsening with norm bound. The algorithm is explained in details in **[13]** and presented in algorithm 1 shown in Figure 15 (also taken from **[13]**). **1** is a vector of all ones and e_i is the i-th coordinate vector.

Algorithm 1 Ostrowski coarsening with norm bound.

procedure $C \leftarrow \text{COARSEN}(A, \rho)$
$C \leftarrow \{\}, F \leftarrow \{1, ..., n\}$
$A_f \leftarrow A$
while true do
$D_f \leftarrow \text{diag}(A_f)$
$X \leftarrow I - D_f^{-1/2} A_f D_f^{-1/2}$
$S_{ij} \leftarrow |X_{ij}|$
$\mathbf{v}^{(1)} \leftarrow S^* S\mathbf{1}$, $\mathbf{v}^{(2)} \leftarrow S^* S\mathbf{v}^{(1)}$
$\mathbf{w}^{(1)} \leftarrow SS^*\mathbf{1}$, $\mathbf{w}^{(2)} \leftarrow SS^*\mathbf{w}^{(1)}$
$I \leftarrow \{i \in F \mid v_i^{(1)} \neq 0, v_i^{(2)}/v_i^{(1)} \geq \rho^2\} \cup \{i \in F \mid w_i^{(1)} \neq 0, w_i^{(2)}/w_i^{(1)} \geq \rho^2\}$
if $I = \{\}$ **then stop**
$g_i \leftarrow (\mathbf{e}_i^* S\mathbf{1})^{1/2} (\mathbf{1}^* S\mathbf{e}_i)^{1/2}$ for each $i \in F$
$C \leftarrow C \cup \{i \in I \mid g_i$ locally maximal among $I\}$
$F \leftarrow \{1, ..., n\} \setminus C$
$A_f \leftarrow A(F, F)$

Figure 15: Ostrowski coarsening with norm bound.

At the end of the coarsening, the initial operator A is divided into 4 operators such that:

$$A = \begin{bmatrix} A_{ff} & A_{fc} \\ A_{cf} & A_{cc} \end{bmatrix}$$

where A_{ff} = A(F, F), A_{fc} = A(F, C), A_{cf} = A(C, F), A_{cc} = A(C, C), and where we have assumed that the coarse variables are ordered last. The coarsening algorithm ensures that:

$$\left\| I - D_f^{-1/2} A_{ff} D_f^{-1/2} \right\|_2 > \rho$$

where D_f = diag(A_{ff}) and ρ is an imposed tolerance. Algorithm 1 is applied recursively until only one variable is left in C.

5.2.3.2 Interpolation

The computation of the interpolation operator is rather complex and details can be found in **[5]**. Let us just note that the computation of the interpolation operator can be divided into two main parts:

- Compute interpolation weights,
- Compute interpolation support.

The interpolation produces, for each level, an operator W such that the interpolation matrix:

$$P = \begin{bmatrix} W \\ I \end{bmatrix}$$

allows to restrict an array x to its coarse and fine subparts x_f = x(F) and x_c = x(C) as:

$$x = \begin{bmatrix} x_f \\ x_c \end{bmatrix} \approx Px_c = \begin{bmatrix} Wx_c \\ Ix_c \end{bmatrix}$$

5.2.3.3 Smoother

The smoother chosen is a parameter-free diagonal sparse approximate inverse (SPAI-0) **[13]**. The smoothing operator D_{ff} (denoted D because diagonal) if given by:

$$[D_{ff}]_{ii} = [A_{ff}]_{ii} / \|A_{ff} e_{ii}\|_2^2$$

where $\|A_{ff} e_{ii}\|_2^2$ is simply the sum of the squares of the elements of the i-th column of A_{ff}. This smoother is optimal in a certain Frobenius norm. The smoothing strategy in the solver is a Chebyshev iteration method. The associated number of iterations, denoted m, as well as the associated contraction factor, denoted ρ_{chebs}, are also computed during the setup phase.

5.2.3.4 AMG Solver

Given the data produced by the setup phase, the coarse grid problem is solved, in parallel, in Nek5000 at each time iteration. The AMG solver performs a single V -cycle. The corresponding algorithm is given in algorithm 2 shown in Figure 16.

Algorithm 2 AMG solver.

```
procedure x ← AMG_SOLVE(x, b)
    for l = 1 : (nlvls − 1) do
        b_{l+1} = W_l^T b_l
    for l = (nlvls − 1) : 1 do
        x_l = W_l x_{l+1}
        b_l = b_l − ([A_ff]_l W_l + [A_fc]_l) x_{l+1}
        c_1 = [D_ff]_l b_l
        if m_l > 1 then
            α = [ρ_cheb]_l/2, α = α^2
            γ = 2α/(1 − 2α), β = 1 + γ
            r_1 = b_l − [A_ff]_l c_1
            c_2 = (1 + γ)(c_1 − [D_ff]_l r_1)
        for i = 3 : m_l do
            γ = αβ, γ = γ/(1 − γ), β = 1 + γ
            r_{i−1} = b_l − [A_ff]_l c_{i−1}
            c_i = (1 + γ)(c_{i−1} − [D_ff]_l r_{i−1}) − γ c_{i−2}
        x_l = x_l + c_{m_l}
```

Figure 16: AMG solver.

The inputs to the function are b, the right hand side, and x, an initial guess to the solution. The different matrices and the parameters m and ρ_{chebs} appearing in the algorithm are the ones computed during the setup phase and are identified at each level by the subscript *l*.

5.2.3.5 Hypre library as an alternative to MATLAB

Using MATLAB for performing the setup procedure has several disadvantages:

- A MATLAB licence is required,
- There is no way to integrate the setup inside Nek5000,
- The code can be fairly slow (up to several hours/day for largest cases),
- It cannot be parallelized.

In order to tackle these issues, we propose to use Hypre **[10]**, a library for linear algebra, as an alternative to perform the main part of the setup. The chosen approach is to use Hypre for performing the coarsening and interpolation operations instead of MATLAB (meaning that algorithms will differ) but to keep the same smoother. This way, Hypre takes care of the time consuming coarsening and interpolation parts and we do not need to modify the AMG solver in Nek5000. Hypre offers several choices for the coarsening method **[9]**:

- CLJP,
- Ruge-Stueben,
- Falgout,
- PMIS,
- HMIS,
- CGC.

In practice, the MATLAB code is replaced by a C code calling the Hypre routine for the AMG setup. The information about the coarsening and interpolation is then used to compute the exact same smoothers at each level as with MATLAB. Then, setup data is written out in binary files using the same formalism. The use of Hypre has been tested in serial only and significant improvement in setup time has been achieved while similar performance for the AMG solver was kept, as we will show in WP3. The setup in Hypre can also be parallelized but this requires a proper matrix partition among processes and it has not been tested yet.

5.3 ExaGS library

Gather-scatter operations are one of the most important communication kernels in Nek5000 for fetching data dependencies (gather), and spreading back results (scatter). The current implementation in Nek5000 is based on the Gather-Scatter library, GS, which utilizes three different strategies: pairwise nearest neighbour communication; the Crystal router, which aggregates smaller messages and route them around the network in a hypercube like pattern; and finally, collective all-to-all communication. Implemented using non-blocking two-sided message passing, the library has proven to scale well to hundreds of thousands of cores.

However, the necessity to match sending and receiving messages in the two-sided communication abstraction can quickly increase latency and synchronisation costs for very fine grained parallelism, in particular, for the unstructured communication patterns created by unstructured problems.

We have therefore started to develop ExaGS, a reimplementation of the Gather-Scatter library, with the intent to use the best suitable programming model for a given architecture. Following the promising results from the EPiGRAM project **[15]**, we have chosen to write a first implementation of ExaGS using the one-sided programming model provided by the Partitioned Global Address Space (PGAS) abstraction, using Unified Parallel C (UPC).

There are two major issues for an efficient UPC implementation. First an efficient shared data structure is needed, which allows for different sized memory blocks on each thread, with easy access from any thread. Traditionally, distributed shared memory models have allocated shared memory across threads in an equal, fixed block size. To solve this problem, we have used the directory of objects approach, where each thread defines its own shared memory region, which can grow and shrink independently of each other. Each region can be accessed by going through a directory, consisting of shared pointers to each region. The downside of this approach is that all library functionalities provided by UPC, e.g. collective operations, have to be re-implemented to suit our data structure.

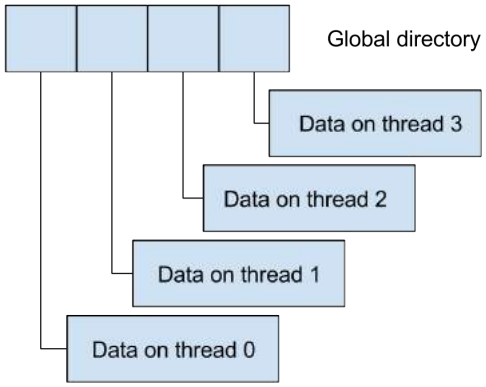

Figure 17: An illustration of the directory of object data structure on four threads.

The second issue is to derive efficient point-to-point synchronization primitives in UPC, necessary for nearest-neighbour communication. For this we have implemented two different strategies. First, by protecting each transfer with a locking mechanism, and secondly by a shared variable, using a strict access policy, acting as a semaphore. Both approaches have their pros and cons.

A lock is standard and provided natively by UPC, but requires at least two locks to protect both send and receive buffers, thus the communication becomes very similar to the two-sided model, with an explicit synchronization point for both send and receive operations. With a shared semaphore, a less restrictive communication abstraction can be chosen, where the semaphores implicitly enforce synchronization by protecting the shared memory buffers. This way, the communication pattern becomes very similar to multithreaded programming. However, care should be taken to avoid deadlocks, race conditions and network contention when polling the remote memory locations.

By utilizing both the shared data structure and both of our point-to-point synchronization schemes, we have successfully rewritten both the collective communication parts of the Gather-Scatter library as well as all three (two for the semaphore) of the gather-scatter routines communication strategies.

To test our new communication kernel, we have developed Nekbarebone, a new version of Nek5000's co-design application Nekbone, for which different communication back-ends can be used. In order to assess the feasibility of our approach, we ran a weak scalability test of the crystal router's ping-pong test in Nekbarebone, using both the original MPI and our new lock-free UPC algorithms in ExaGS. The results in Figure 18, are promising, in particular at scale, and clearly demonstrates the feasibility of our approach.

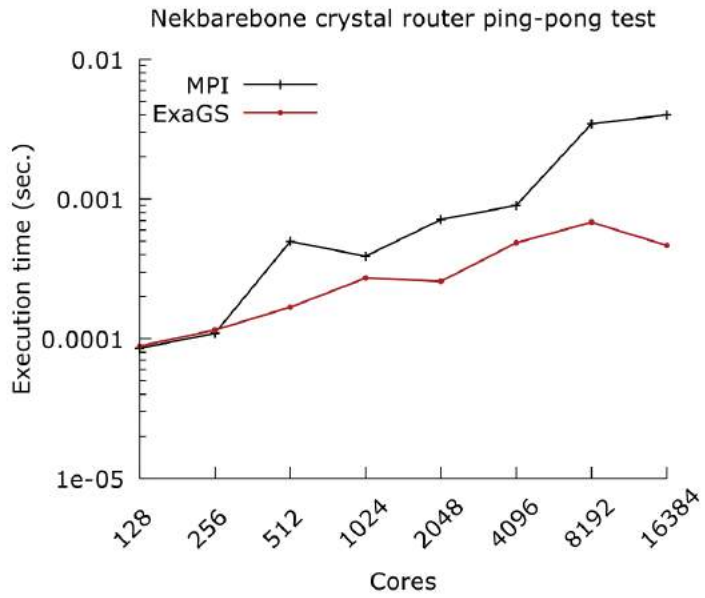

Figure 18: Weak scalability test of the crystal router gather-scatter operation inside Nekbarebone, using 128 elements per core, with a polynomial order of ten, running on the Cray XC40 Beskow.

5.4 Hybridizable DG

The motivation for the CG-DG solver was to bring together the advantages of discontinuous (DG) and continuous Galerkin (CG) methods in one hybrid scheme. The approach consisted of replacing each element in HDG (hybridizable discontinuous Galerkin) method by a group of elements discretized by CG (which would correspond to one mesh partition in parallel setting). We expected such a solver to be efficient in terms of:

- communication: HDG only requires pairwise communication between partitions regardless of mesh topology. The amount of data transferred between cluster nodes would therefore be lower than in classical DG and collective broadcast and reduce operations would be limited.
- computation/work: HDG is more efficient than classical DG because the global system involves only hybrid unknowns located on element traces (between partitions in our case). The remaining (element-interior)

degrees of freedom are then reconstructed from hybrid variable and this step is fully parallel.

The HDG algorithm consists of two major steps:

1. Assembly and solution of global system for hybrid variable. The rank of this system is comparable to statically condensed CG system.
2. Reconstruction of remaining unknowns from given values defined on element traces.

When replacing each finite element in HDG by a group of CG elements, we were aware that the second step (partition-wise reconstruction of unknown degrees of freedom) would involve a solution of a dense system with rank proportional to the number of unknowns in each partition. We hoped to exploit the known block structure of this matrix to keep the cost in step 2) comparable to classical CG solve in one partition. The reduced communication would then still ensure favorable scaling and wall clock times.

When doing the cost analysis of the CG-DG algorithm, however, we found out that the assembly and solution of the global system for hybrid variable (i.e. step 1) incurs significantly larger asymptotic cost when each partition is regarded as a macro-element instead of a series of classical finite elements. The technical details regarding this have been reported in deliverable 1.2.

This additional cost means that the CG-DG solver has limited potential to be competitive with CG or DG in terms of efficiency, even though its formulation is mathematically sound and implementation possible. We have not attempted a concrete quantitative comparison with CG and DG which would consider other factors beyond the scope of the performance model such as memory bandwidth and latency, but based on former experience with the implementation and timings of both CG and 'classical' HDG in Nektar++, we conclude that the CG-DG solver would not yield smaller run times.

The application of the CG-DG solver on a mesh with a single partition amounts to the solution of a CG system with Dirichlet boundary conditions prescribed by the hybrid variable (weakly). This part of the algorithm was implemented and we plan to use it in challenging test cases with complex flow physics. On the other hand, we do not intend to implement the remaining step in the CG-DG solver and would rather invest our resources in other areas that fall in the ExaFlow remit and which we currently regard as more promising than the CG-DG algorithm.

6 Energy efficiency

In this section, we examine different methods for investigating and understanding energy consumption of our target co-design applications with the eventual goal of reducing energy to solution without harming temporal performance.

As there are no exascale machines we can study, there is some necessary speculation about architectures and how they might relate to current architectures which can be studied. In Section 6.1, a comparison is made of energy to solution and power consumption for a node based on a current Intel Xeon processor and one based on an Intel Xeon Phi processor. In Section 6.2, a study is performed examining the effects of varying clock frequency on each phase of execution of a CFD code, using one of the test-cases from WP3. Finally, in Section 6.3, the effect of changing pre-conditioner on both energy to solution and power consumption are examined using a high-accuracy high-speed measurement system able to gather data on system components other than CPUs.

6.1 Energy efficiency of generated codes

In addition to the runtime analysis performed on the five different finite difference algorithms, SOTON has evaluated the five algorithms in terms of energy efficiency on multicore CPUs and Intel Xeon Phi KNL processors. The Taylor-Green vortex problem presented earlier was used for this purpose, and was run in parallel for 500 time-steps on (a) 1 ARCHER CPU node (24 CPU cores) using MPI and (b) 64 cores on an Intel Xeon Phi KNL processor, again using MPI. The PAT MPI library **[16]** reads the hardware counters to obtain power and energy consumption data. Some preliminary results from a single simulation are shown in Figure 19 & Figure 20.

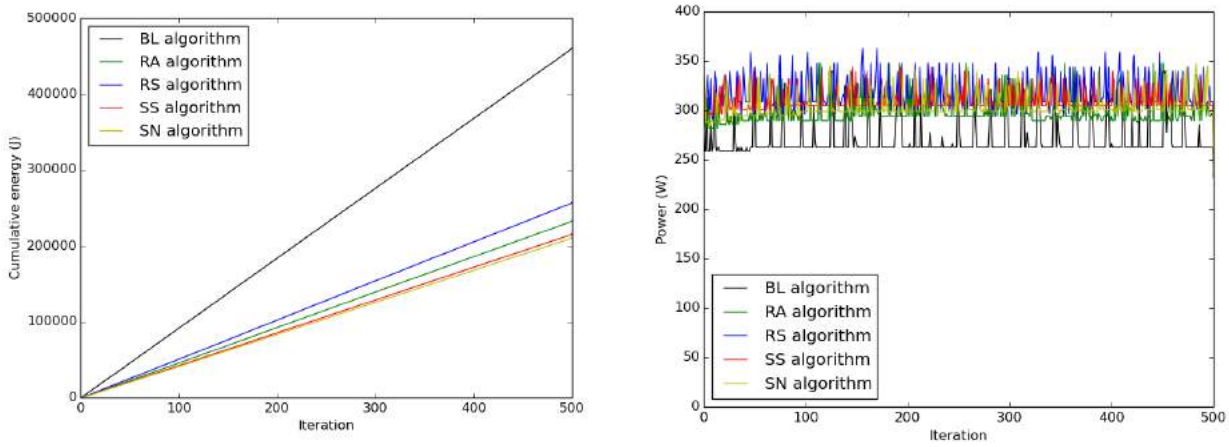

Figure 19: Cumulative energy (left) and power consumption (right) over 500 iterations of a Taylor-Green vortex simulation when run in parallel over 24 MPI processes on a single 24-core ARCHER node.

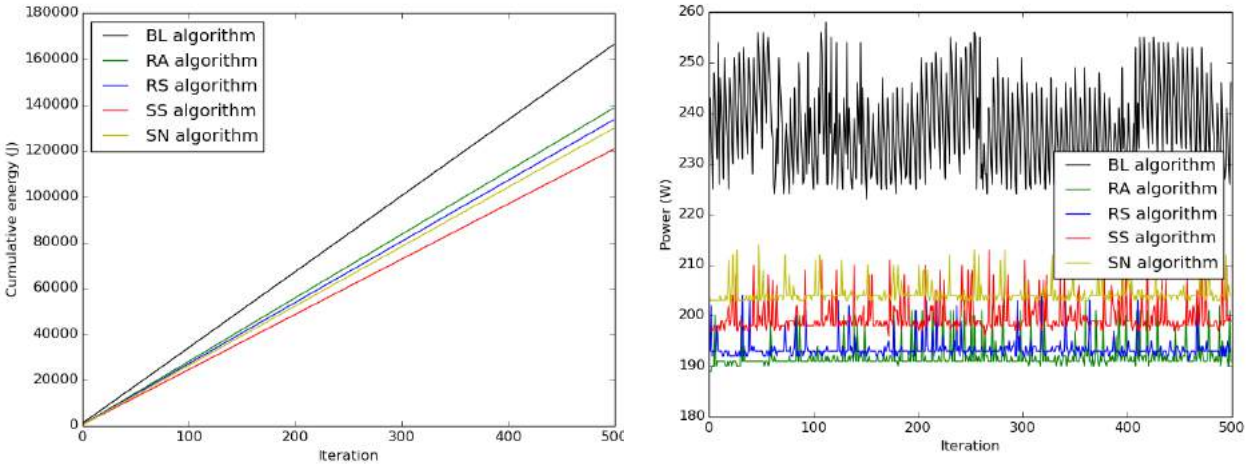

Figure 20: Cumulative energy (left) and power consumption (right) over 500 iterations of a Taylor-Green vortex simulation when run in parallel over 64 MPI processes on a single Intel Xeon Phi KNL processor.

Interestingly, the Baseline (BL) algorithm, which stores finite difference derivatives in global work arrays over the whole grid, is once again the worst-performing algorithm both in terms of runtime and energy usage. This suggests that a large amount of time and energy are required to deal with the relatively high memory-intensity of the BL algorithm. In contrast, the Store None (SN) and Store Some (SS) algorithms (which are more computationally-intensive yet less memory-intensive) are the fastest and most energy efficient across the two architectures. Furthermore, the new KNL processor uses less than half as much energy across all the algorithms as a result of the faster runtimes and lower power consumption. Preparing codes for such novel architectures and continuing to evaluate different algorithms with energy in mind will therefore be advantageous, especially where energy budgets are a factor when submitting jobs to a supercomputing service. This work has been submitted as another ParCFD 2017 conference abstract**[17]**.

6.2 Clock frequency adaption

Since numerical applications typically comprise different phases in each time-stepping iteration (e.g. the actual computation, I/O, etc.), the CPU might not be the bottleneck in all of these phases. The CPU's clock frequency might hence be adapted in order to reduce the energy demand if the respective hardware allows for this. Furthermore, one may ask "what is the optimal degree of parallelism if assessing this question from the energy point of view?" Moreover, different I/O formats and strategies (e.g. large number of rather small XML files vs. a few rather large HDF5 files) may have an impact on an applications energy footprint. We hence decided to address these questions in an appropriate measurement framework.

Since we are interested in the energy consumption of real world problems, ExaFLOW's industrial automotive use case has been used for this purpose with Nektar++. The questions mentioned above have been addressed by varying:

- CPU clock frequency (entire available range as well as (non-trivial) DVFS governors), *fixed* frequency used within entire run
- number of used nodes in a strong scaling fashion (3 nodes to 48 nodes, minimum determined by memory demand of the use case, maximum determined by Nyquist–Shannon theorem)
- I/O strategy and format (XML vs. HDF5, cf. above)

By deploying Cray's power monitoring features, we can query the aggregated energy consumption of entire nodes at a frequency of 10Hz from within the time-stepping loop in Nektar++ for the three most dominant phases (actual computation, halo exchange and I/O) of an iteration, which allows distinguishing between different phases and time-stepping iterations. After every odd iteration, a checkpoint is written to achieve a significant size of the sample set.

It is easiest to consider the performance by splitting the results into three distinct phases:

6.2.1 Actual computation

Figure 21 shows the energy demand of this phase for different amounts of parallelism, this is the total energy consumed by the nodes but does not include energy used by any interconnect, for which data is unavailable on this platform. The depicted values are the sum over 95 iterations (the first five iterations have been dropped since they do further resource-intensive setup tasks). Obviously, deploying as much parallelism (in terms of node count) as possible is optimal from the energy point of view (at least within the examined range). Figure 22 depicts the energy as well as runtime demand of this setting in detail. Fortunately, the energy demand drops faster when compared to the increase of runtime. The energy-delay-product (EDP), which is a widely-used measure to assess energy vs. runtime trade-offs, is shown in Figure 23. EDP is simply the product of energy and runtime an optimal can be found by minimising this metric. From this, it seems running at 1.7GHz is optimal per EDP. Doing so reduces the energy demand by 19.2% compared to the default setting of 2.5GHz while increasing the runtime by 12.6%. If compared to turbo mode (depicted as 3300000 kHz), energy savings of 34% are possible while increasing the runtime by 20%.

6.2.2 Halo exchange

Figure 24 shows the energy demand of this phase for different amounts of parallelism. Again, the depicted values are the sum over 95 iterations (cf. above). Using a maximal amount of parallelism is shown to be the best choice here. Hence, Figure 25 depicts the energy as well as runtime demand of this setting. Unfortunately, the decreasing energy demand is contrasted now by a more severe increase in runtime, so that the EDP minimum is achieved by the governor "conservative" **[18]** per Figure 26, whose value is close to those of the CPU's base clock frequency of 2.5GHz. We even suppose that the delta is within the margin of error. The CPU governor is a mechanism which determines how aggressively the CPU changes frequency based on computational demand As a comparison, all

figures also include results from letting the governor dictate CPU frequency rather than fixing it for each simulation.

The relative amount of time and energy spent in the actual computation and halo exchange phases points to a substantial load imbalance. Furthermore, the observed behaviour of the halo exchange does not match with the conjectures one may have, i.e. during MPI communication, the energy efficiency of CPUs should benefit from a reduced clock frequency as well as lower node count. It is thus necessary to further investigate what is going on in this phase. Another measurement approach may be required here since thwarting a *few* high loaded ranks in the actual computation phase by decreased clock frequency will increase the energy consumption of *many* waiting ranks afterwards.

6.2.3 I/O

Since we are investigating XML as well as HDF5 output, we will analyse each separately:

In contrast to the actual computation and halo exchange phases, I/O deploying the *XML* strategy (cf. above) exhibits the best energy efficiency with quite a low number of nodes (3 or 6) per Figure 27. This encourages an escrow I/O strategy with idling ranks set to an energy saving state. It is however contrasted by an increase in runtime if decreasing the amount of parallelism, as shown in Figure 28, which is due to a significantly reduced power demand (not shown here). While one would expect a reduced power and energy demand in the I/O phase - because the I/O subsystem, not the compute node, is the bottleneck here – the considerable increase in runtime does not entirely fit with expectations. We will thus have to further investigate what is happening here. Despite these considerations, it's not reasonable to run on a rather small number of nodes, since the computation and halo exchange phases are dominant in actual production runs. We hence use the 48-node case for our considerations regarding optimal clock frequencies. As seen in Figure 29, the energy demand drops slightly with reduced clock frequency, while the runtime rises significantly, resulting in an optimal EDP value when utilizing the governor "ondemand" (cf. Figure 30).

If using the I/O strategy based on *HDF5* (cf. above), the results remain almost the same (cf. Figure 31 to Figure 34). The optimal amount of parallelism with respect to runtime, however, seems to be 12 nodes here. In contrast to the XML strategy, the optimal clock frequency pertaining to EDP turned out to be 1.7GHz. It's worth highlighting that the absolute level of energy as well as runtime is higher by a factor of about 4 in this case compared to XML output! We attribute this to an – up to now - suboptimal implementation of the HDF5 module. This is because it triggers 15 *independent* reads (which hence needs to be serialized by the MPI-IO library) per rank and checkpoint which might significantly degrade the overall I/O performance. This issue requires further investigation in the next months. We expect an optimized HDF5 output to perform much better compared to a large number of XML files.

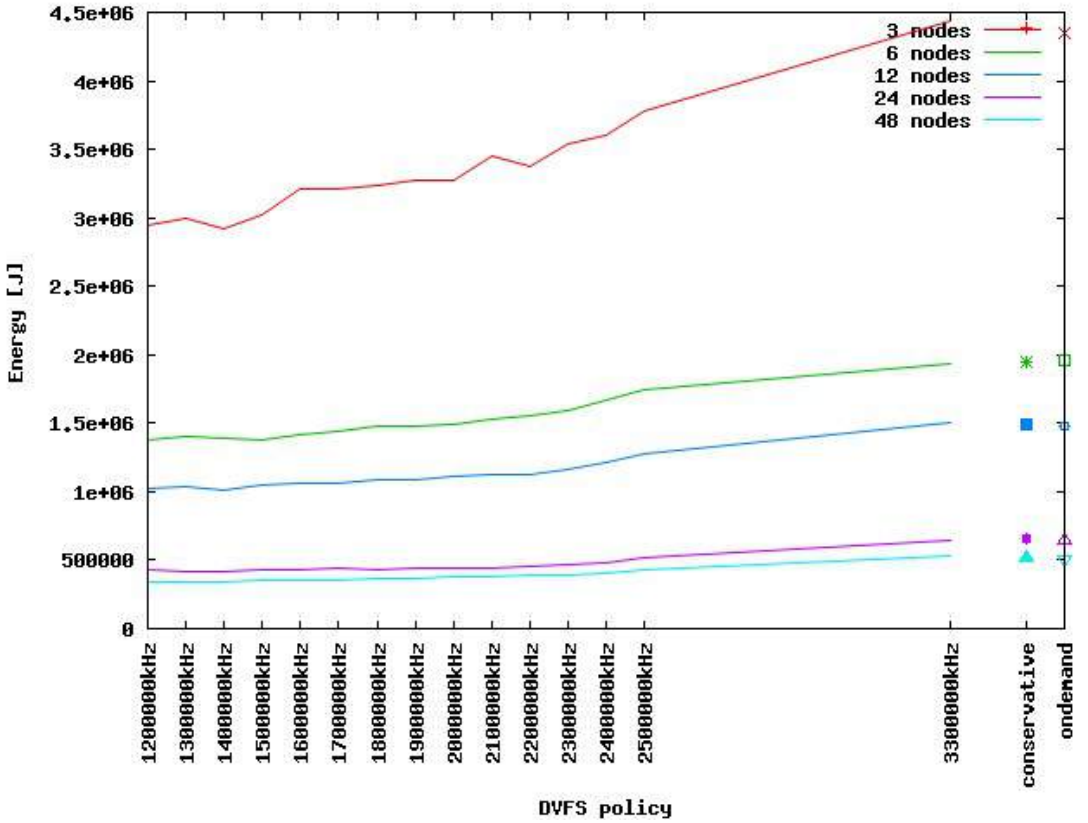

Figure 21: Energy demand of actual computation

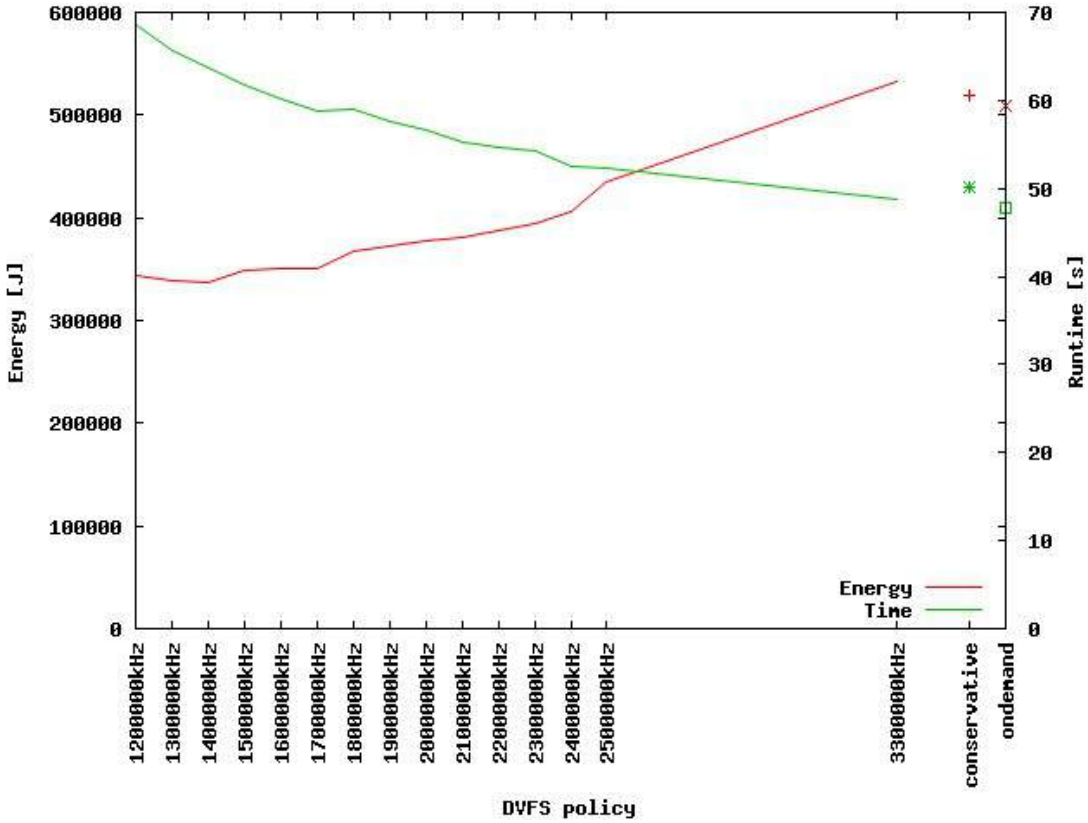

Figure 22: Energy and runtime demand of actual computation on 48 nodes

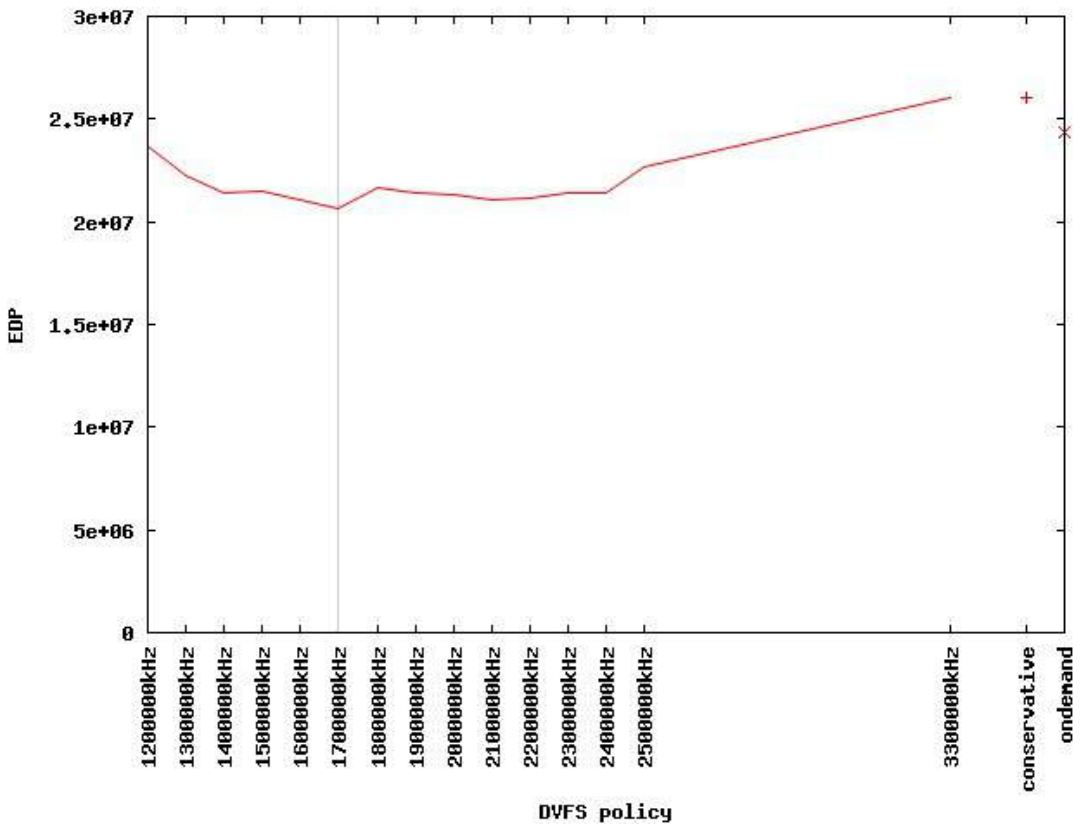

Figure 23: EDP of actual computation on 48 nodes

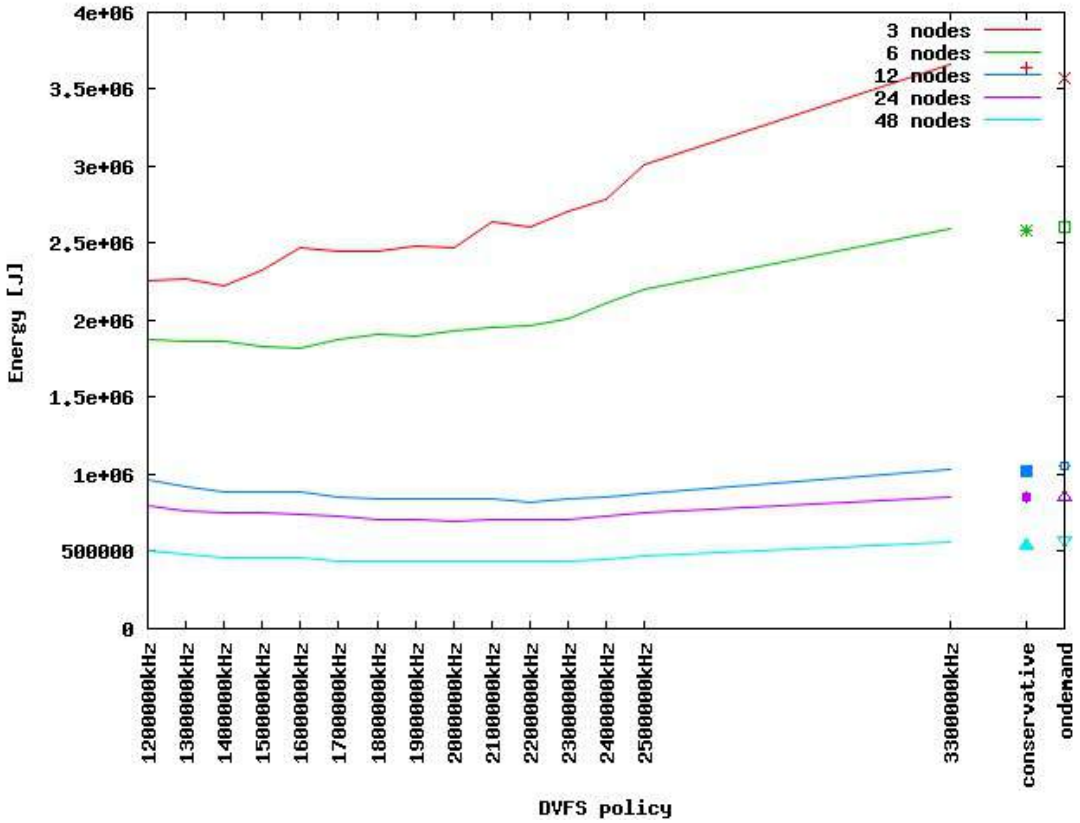

Figure 24: Energy demand of halo exchange

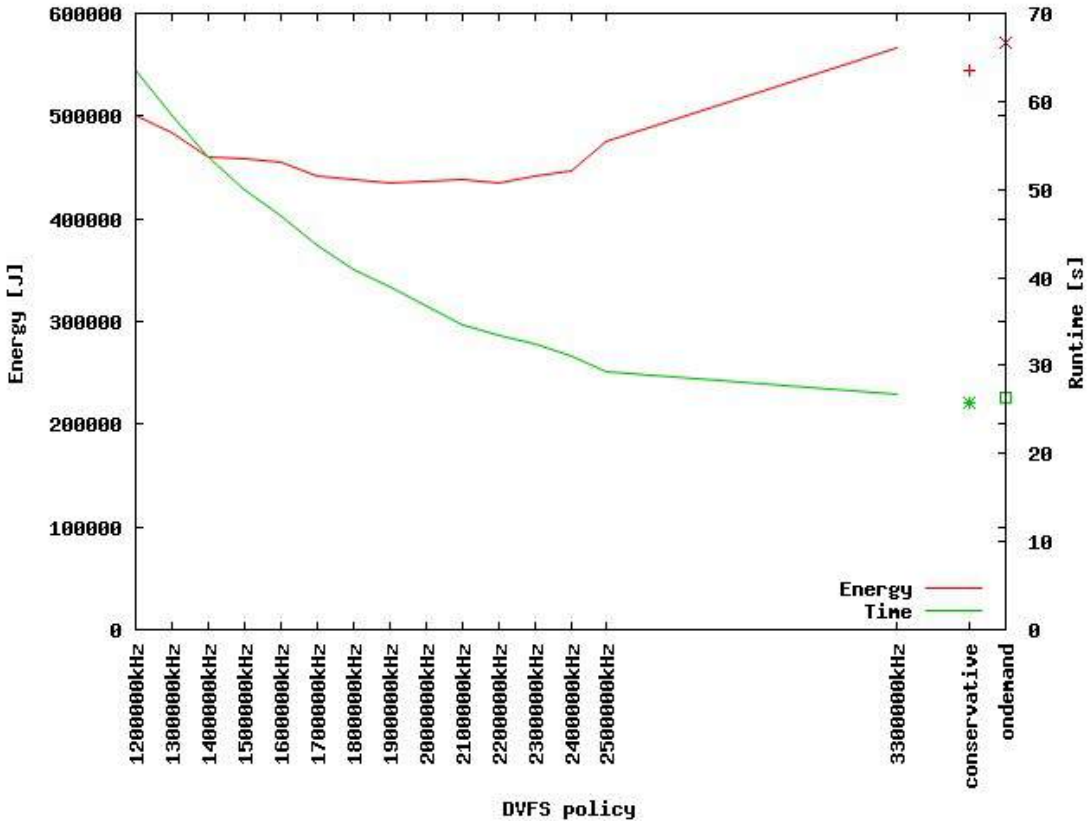

Figure 25: Energy and runtime demand of halo exchange on 48 nodes

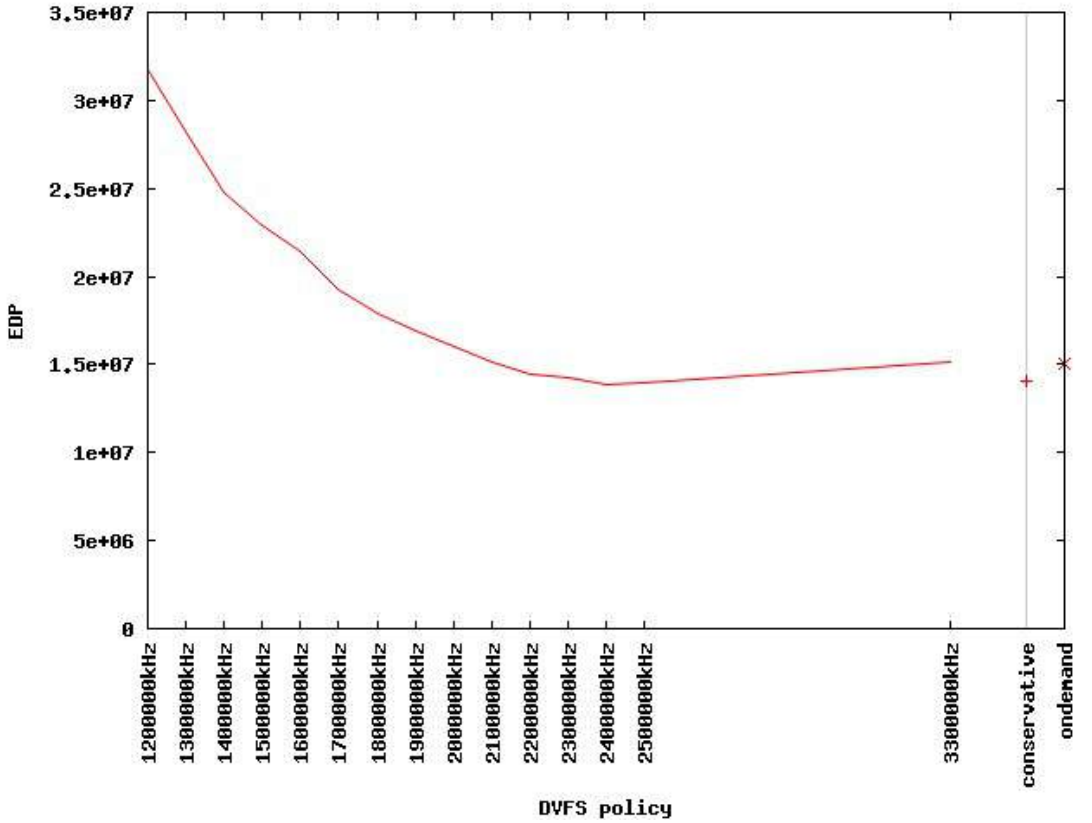

Figure 26: EDP of halo exchange on 48 nodes

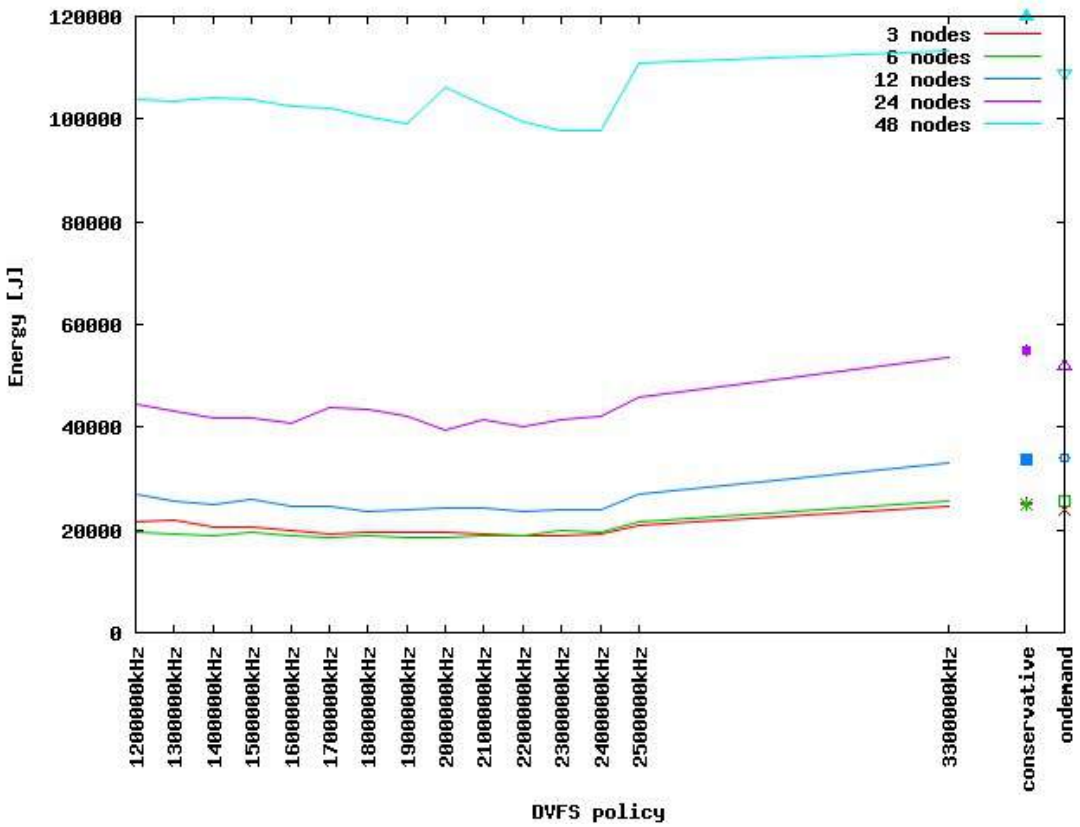

Figure 27: Energy demand of I/O with XML strategy

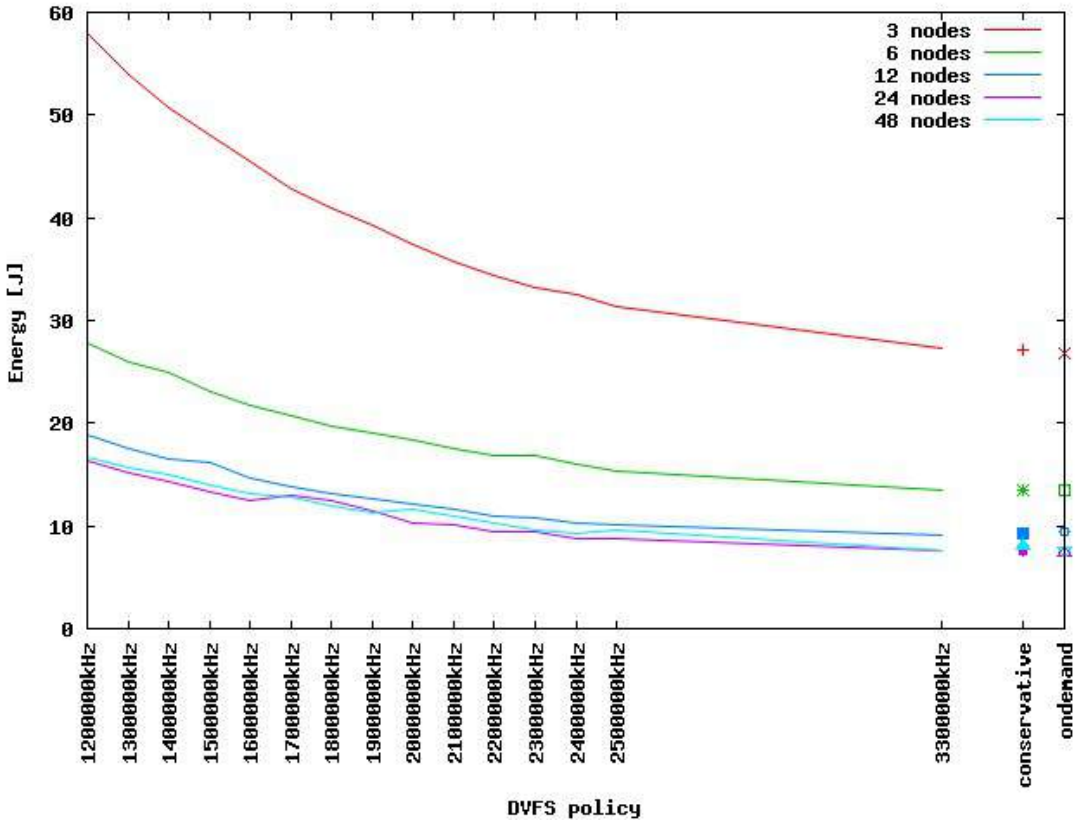

Figure 28: Runtime demand of I/O with XML strategy

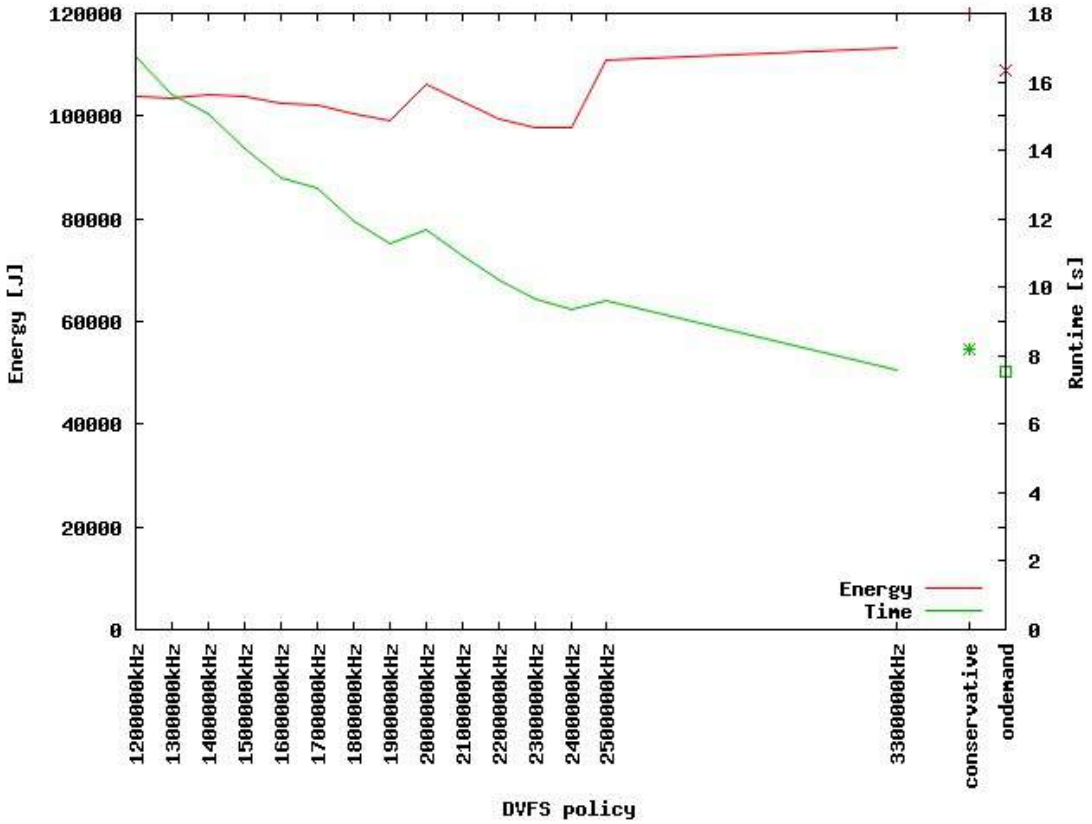

Figure 29: Energy and runtime demand of I/O with XML strategy on 48 nodes

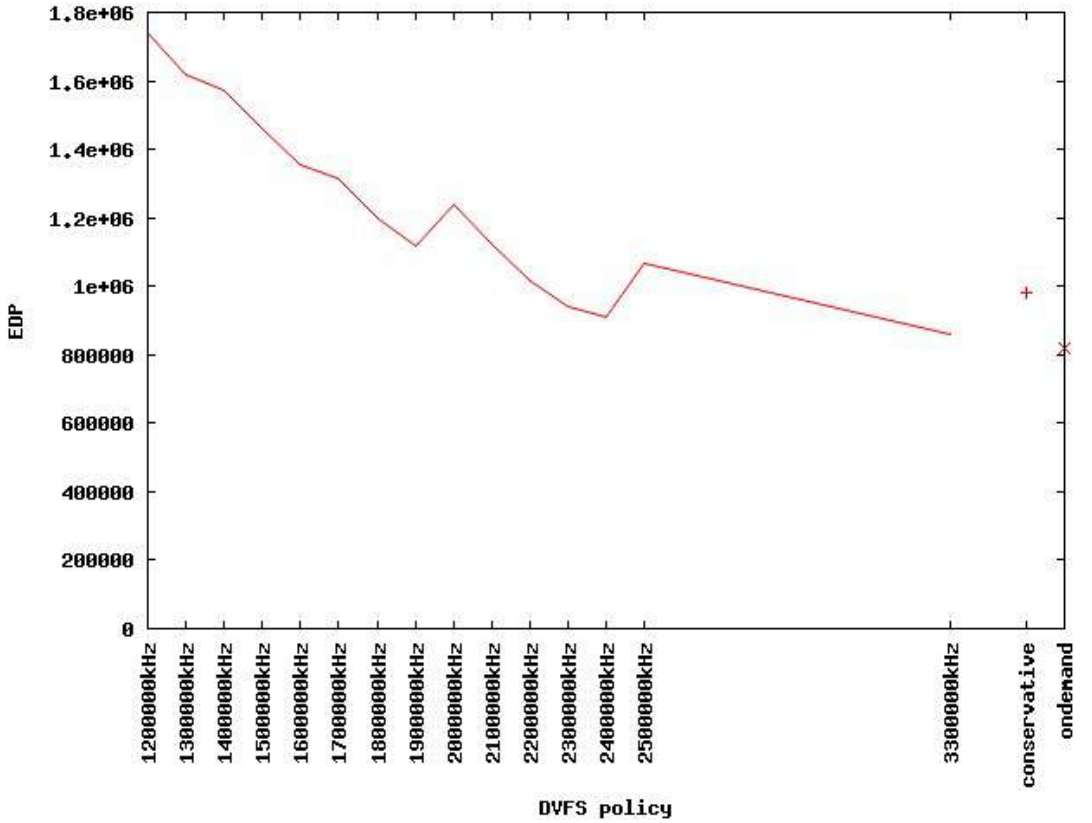

Figure 30: EDP of I/O with XML strategy on 48 nodes

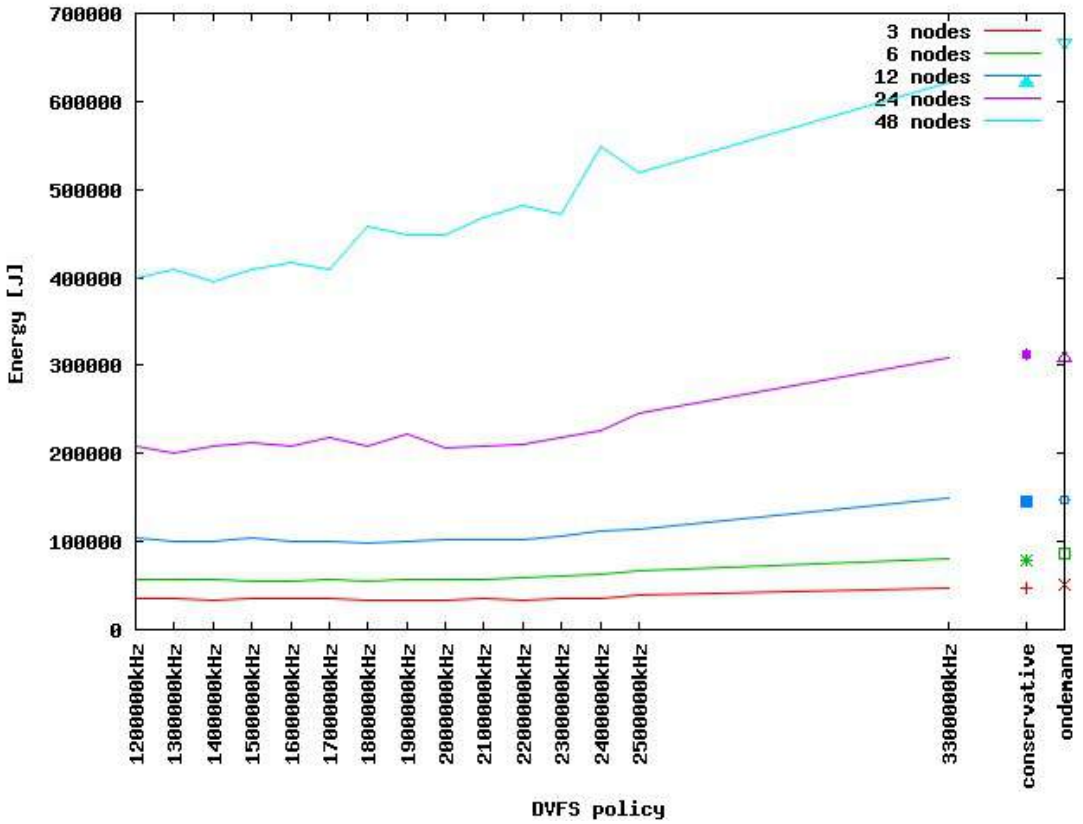

Figure 31: Energy demand of I/O with HDF5 strategy

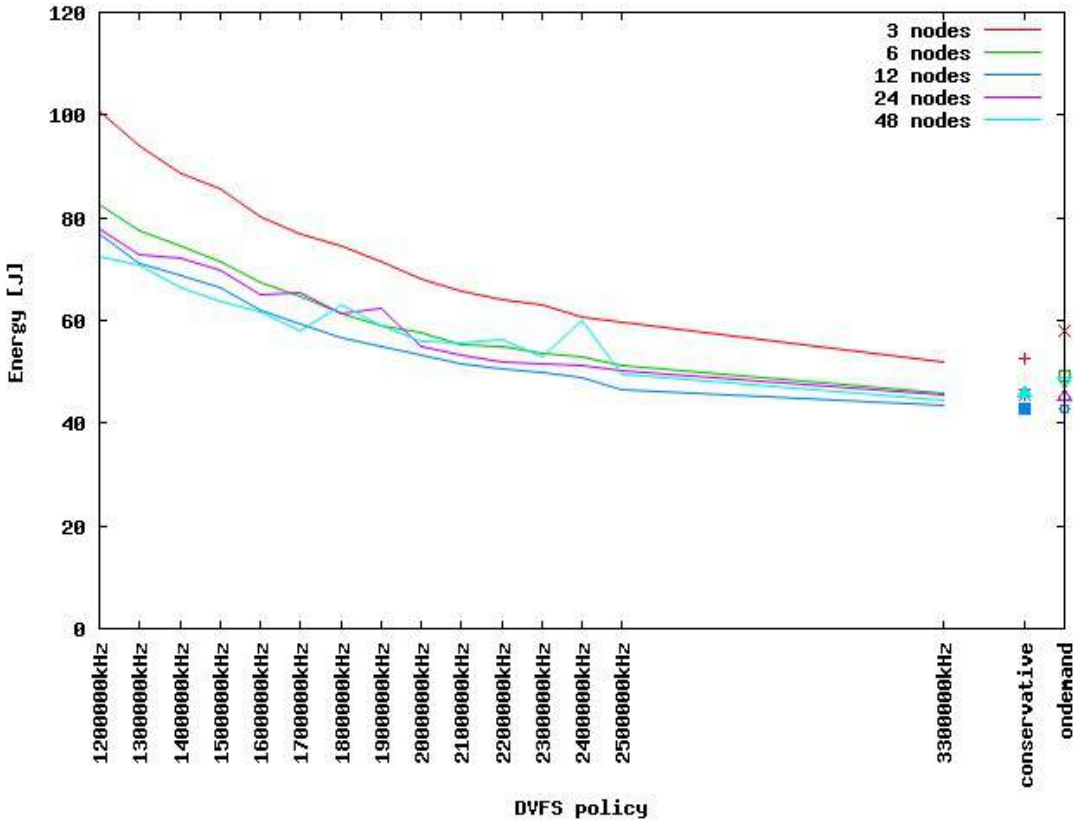

Figure 32: Runtime demand of I/O with HDF5 strategy

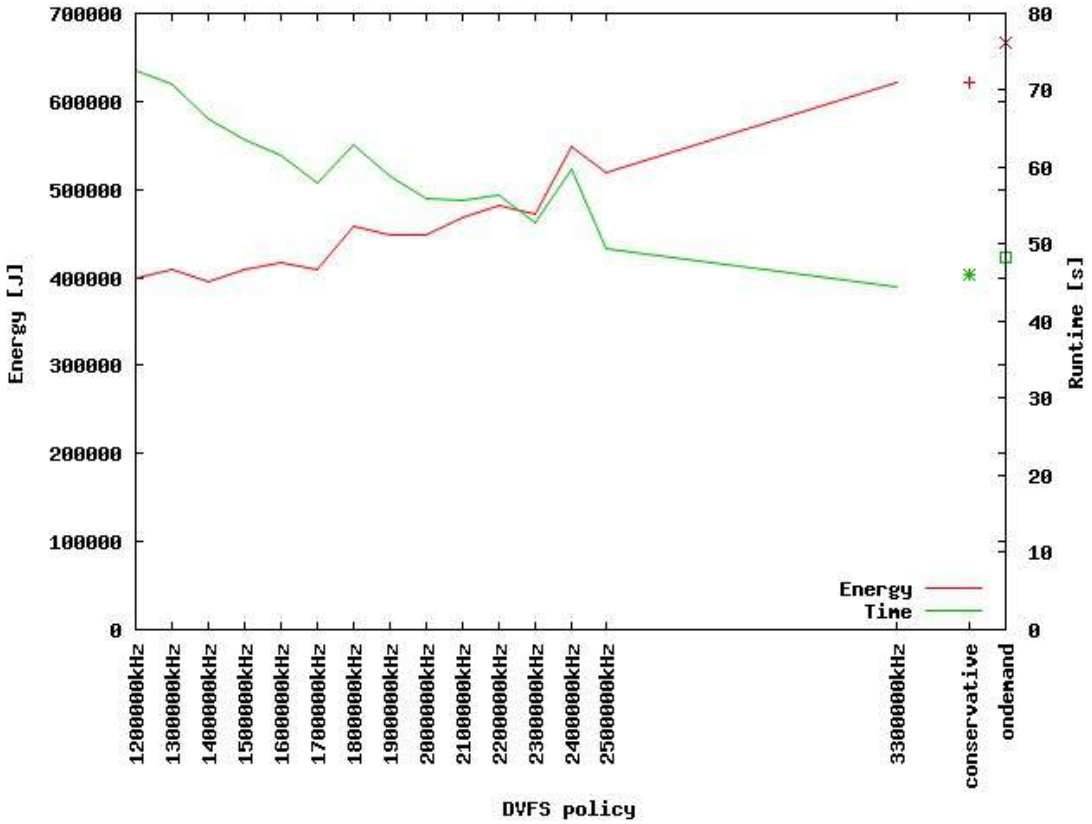

Figure 33: Energy and runtime demand of I/O with HDF5 strategy on 48 nodes

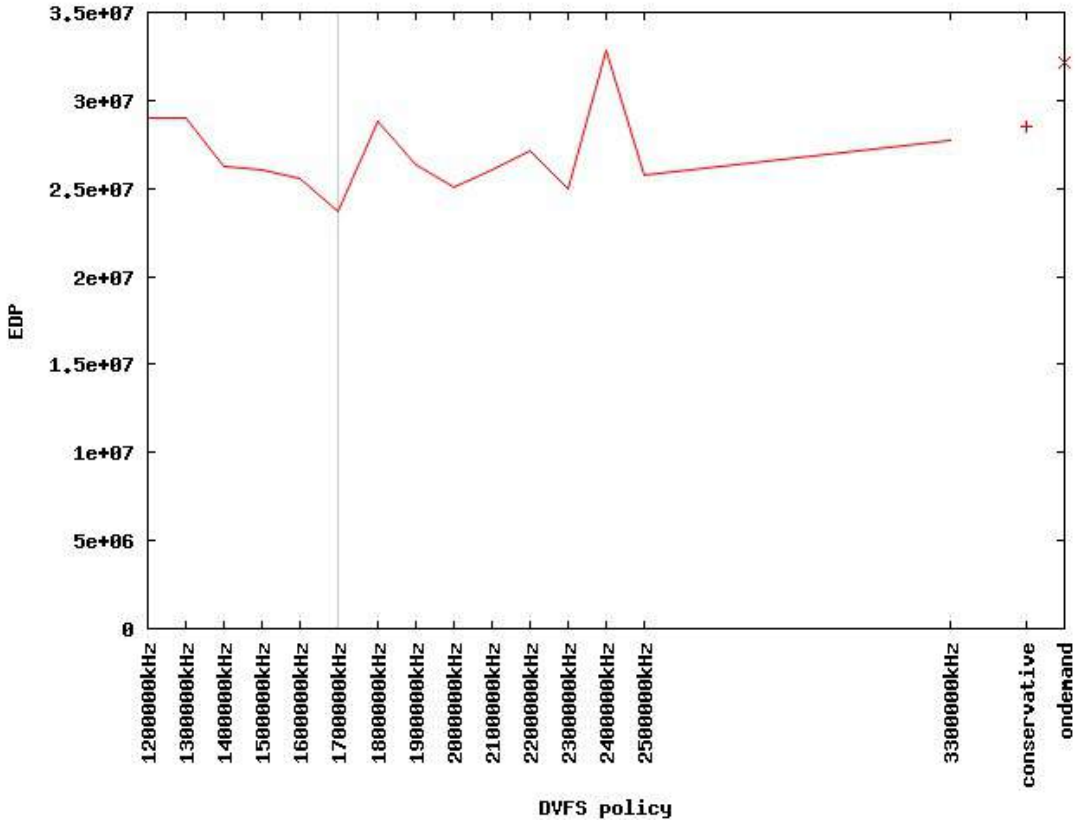

Figure 34: EDP of I/O with HDF5 strategy on 48 nodes

Overall, using an appropriate amount of parallelism (and hence doing the calculations *as fast as possible*) seems to be much more crucial with respect to energy efficiency than using a proper clock frequency.

Although the presented results are neither surprising nor immediately allow for substantial energy savings, we have a framework now to further investigate interesting questions related to DVFS based energy efficiency of Nektar++ and address the issues identified above.

If, after doing so, the halo exchange and I/O phases are going to exhibit optimal clock frequencies which significantly differ from that of the actual computation phase, we will try to implement a mechanism to dynamically adapt the clock frequency in order to lower the application's energy footprint while (almost) preserving runtimes. Furthermore, if an improved load balancing is not appropriate due to heavy data transmission demands, we will consider "load balancing" by DVFS (i.e. reducing the clock frequency of waiting ranks).

6.3 Further Power Analysis

In this section, we look at the effect of changing pre-conditioner on both the energy to solution and the power profile for a single test-case, that of blood flow in a rabbit aorta modelled using Nektar++. This test-case is chosen as it runs well on a single-node system in a reasonable time-frame. In this section, the machine used is an instrumented dual-CPU Intel Xeon system based at UEDIN. Each power line is sampled at a rate of 500 kHz using custom hardware developed by the Adept Project to allow for a high fidelity of measurement and greater accuracy than available on the supercomputers used in other sections.

6.3.1 Energy to Solution

Figure 35, Figure 36, Figure 37 & Figure 38 show the energy consumption by selected system components over time for the test-case when different pre-conditioners are used. They are, respectively: diagonal, full-linear, low-energy and a combination of full-linear and low-energy. In each figure, the dashed line shows the sum of CPU, DRAM and SSD energy.

Choice of pre-conditioner has marked effect on runtime. Low-energy results in the quickest solution at 58s, followed by full-linear and low-energy at 230s, followed by diagonal at 375s and finally full-linear which did not complete after 1000s and was aborted.

Two trends are evident in the graphs:

- DRAM and SSD energy is dwarfed by CPU energy and consequently these are poor targets for initial energy reduction efforts, and;
- ATX energy, the energy drawn by components other than the CPU, DRAM and SSD is high for this system. Reducing this energy by switching computation to another system with a lower power draw for inactive components would result in a fast energy saving.

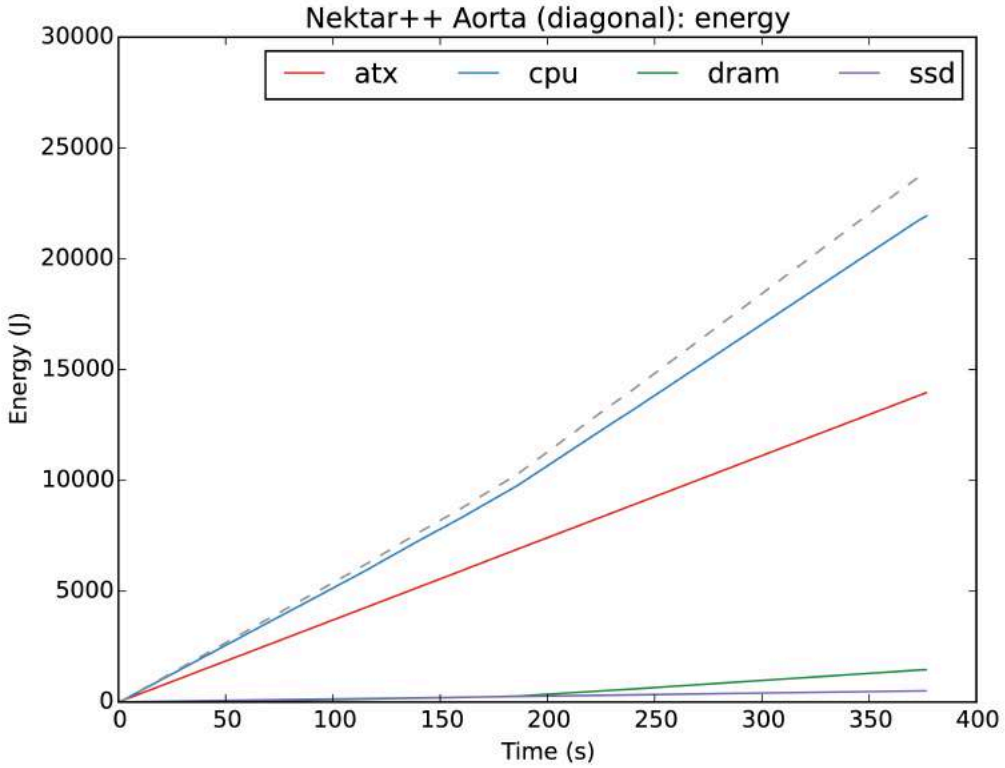

Figure 35: Energy usage over time for the Aorta test case using the diagonal pre-conditioner.

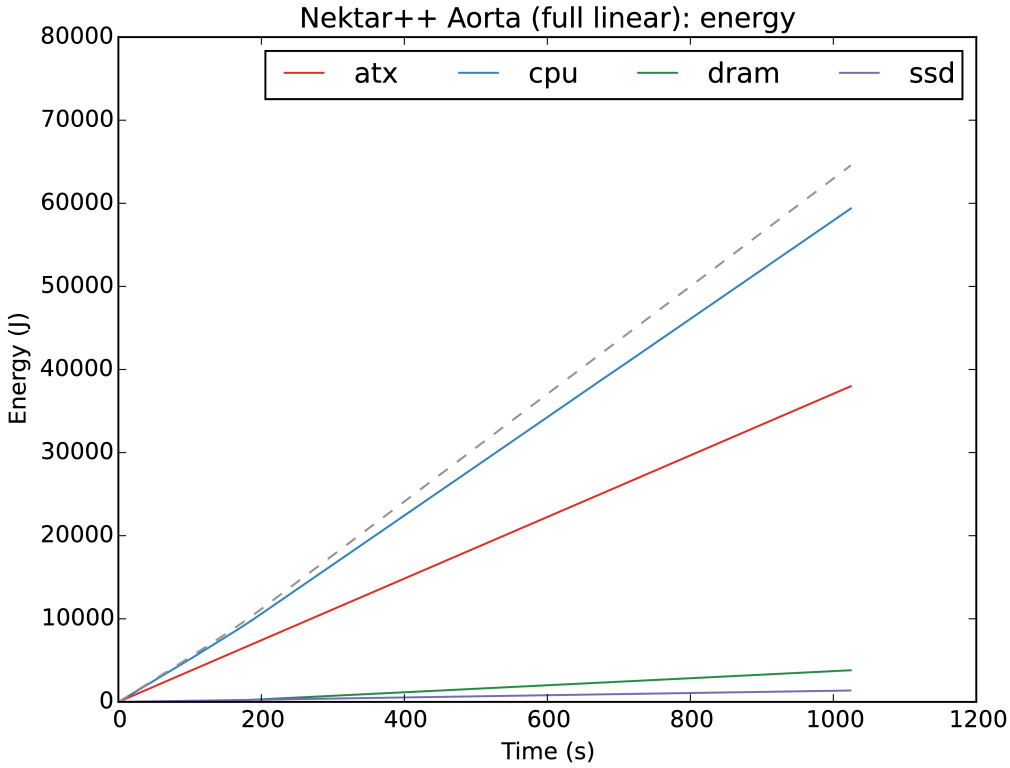

Figure 36: Energy usage over time for the Aorta test case using the full-linear pre-conditioner.

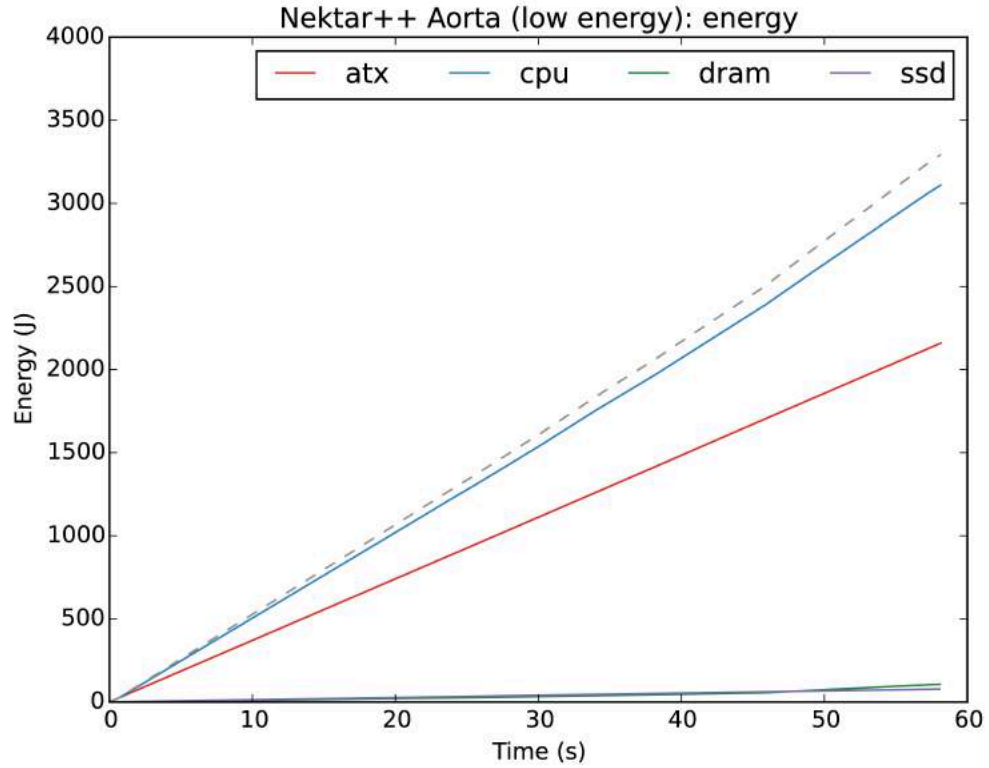

Figure 37: Energy usage over time for the Aorta test case using the low-energy pre-conditioner.

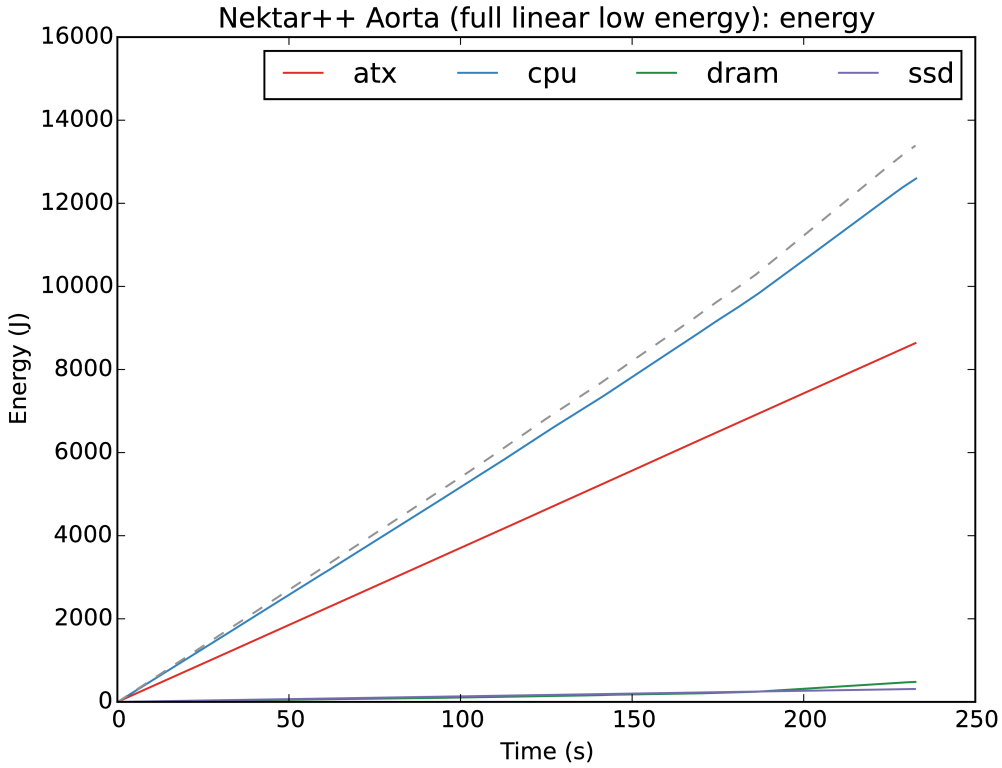

Figure 38: Energy usage over time for the Aorta test case using the full-linear low-energy pre-conditioner.

6.3.2 Power profiles

For a more complete understanding of the energy consumption, it is possible to look at the point-in-time power consumption over the run-time of the test-case. Figure 39, Figure 40, Figure 41 and Figure 42 show the power profile of the four pre-conditioners. What is most evident from these figures, as compared to the energy consumption figures, is:

- there are two distinct phases of CPU activity, a low-power phase in which the power consumption sits at a little over 50W and a high-power phase in which the power consumption rises to close to 70W, and;
- the DRAM power phase matches that of the CPU fairly well in all cases.

It would not be unreasonable then, as a next step, to attempt to determine the computational activity in both phases and see if energy savings can be made by either reducing the run-time of the high-power phase or operating more in the low-power phase, even if it increases run-time, with the restriction that it should save energy overall.

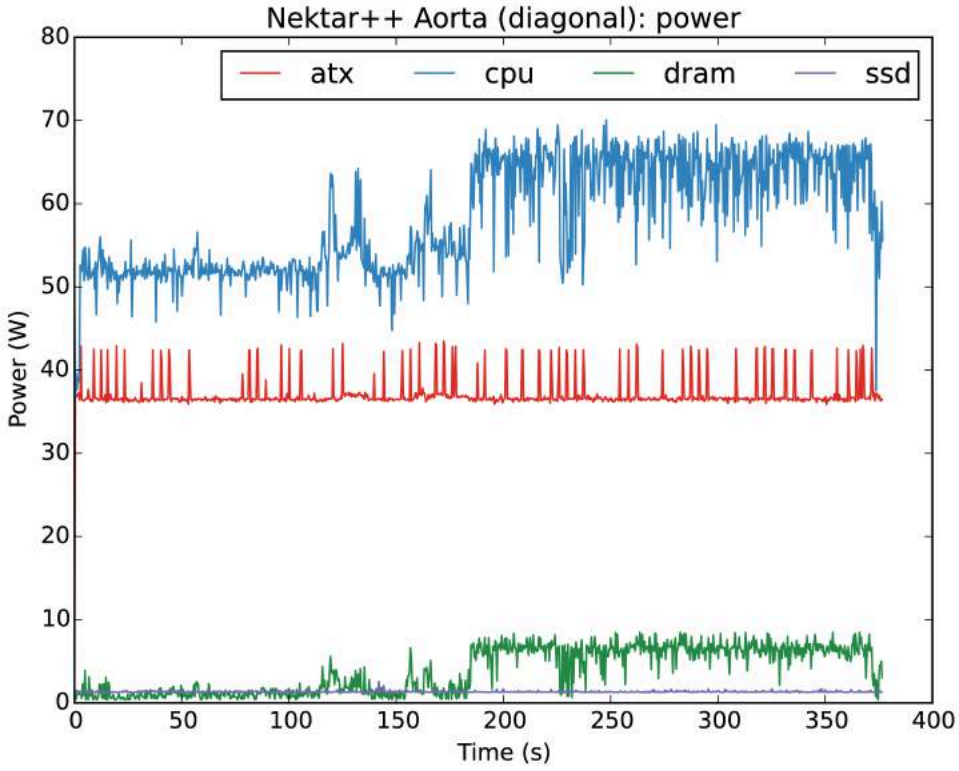

Figure 39: Power profile over time for the Aorta test case using the diagonal pre-conditioner.

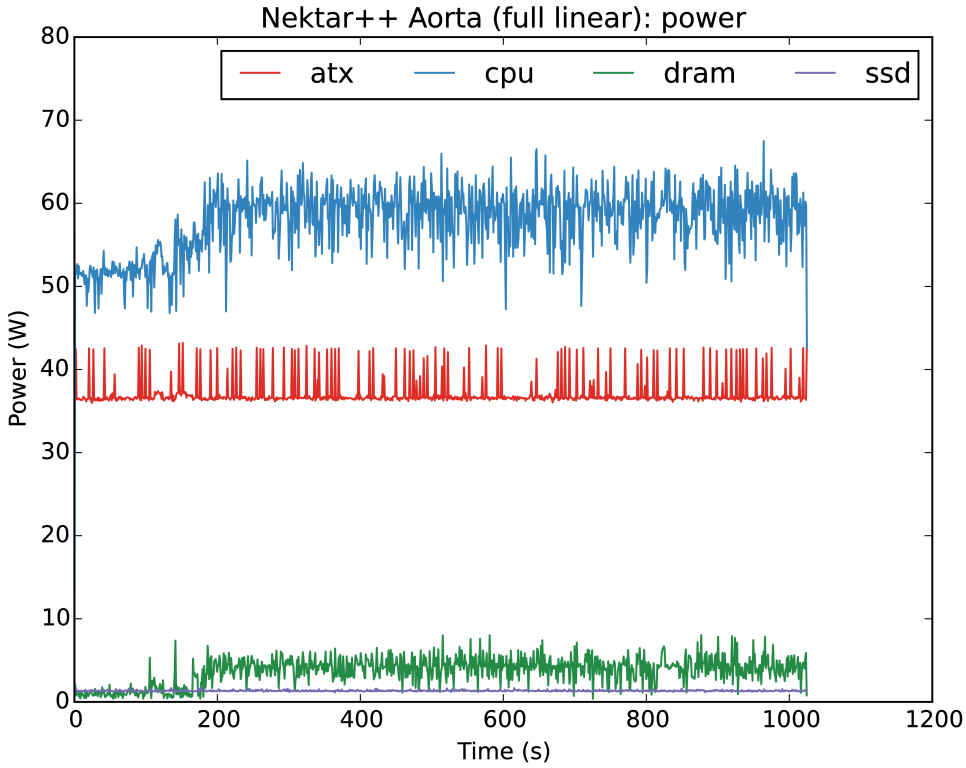

Figure 40: Power profile over time for the Aorta test case using the full-linear pre-conditioner.

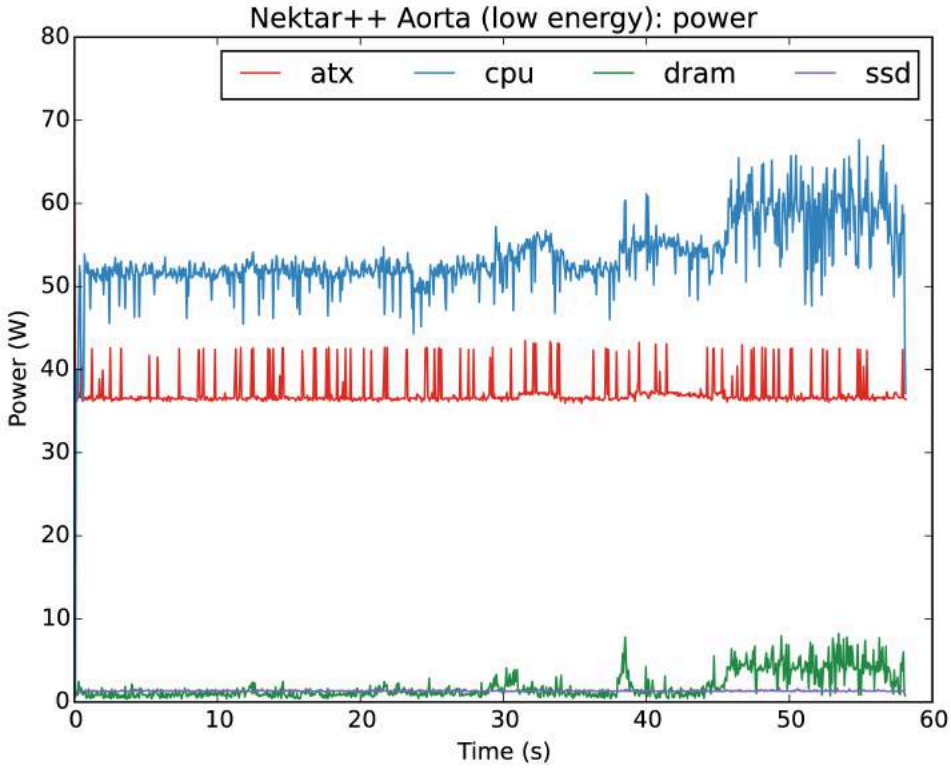

Figure 41: Power profile over time for the Aorta test case using the low-energy pre-conditioner.

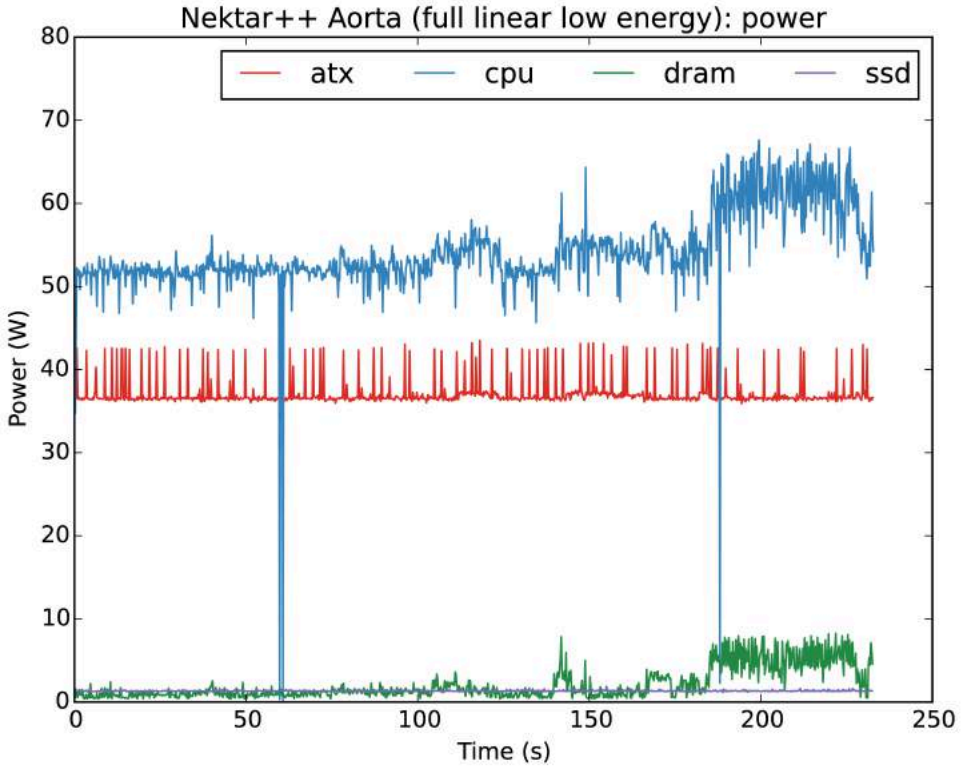

Figure 42: Power profile over time for the Aorta test case using the full-linear low-energy pre-conditioner.

7 Data Management & IO

Exascale computing will serve for very large capability jobs as well as for workflows with many instances of large-scale simulations. Implied, in any case, is an extremely large I/O consumption for reading and writing data as well as for storing these on a large-scale filesystem. This applies in particular to fluid-flow simulations. However, data I/O is an emerging bottleneck in high performance computing (irrespective of application/discipline) because of diverging hardware speed-ups between computation and I/O. This will remain true even with new I/O technologies like burst buffers. Also, non-volatile memory will only gradually help. To reduce the amount of data for storage and handling we propose two solution paths: parallelization of I/O, and I/O data reduction and compression via application-dependent filtering. The main objective of both is alleviation of performance bottlenecks caused by data transfer from memory to disk.

So far, we have focused on the development of the data-reduction algorithms, which are reported in the WP1 deliverable. In order to investigate parallelization of I/O, some small test cases were carried out with the high-level I/O libraries NetCDF (Network Common Data Format) and HDF5 (Hierarchical Data Format) on a Cray XC40 (Hazel Hen) at HLRS. Here we give a short introduction to the I/O libraries.

NetCDF is developed at the Unidata Program Center (UPC) and provides applications with a common data access method for the storage of structured datasets. The original NetCDF API was designed for serial data access and insufficient parallel performance. Parallel netCDF (PnetCDF) is developed by Northwestern University and Argonne National Laboratory (ANL) to provide a parallel API to access NetCDF files with better performance. PnetCDF is built on top of MPI-IO, and allows users to benefit from several optimizations in existing MPI-IO implementations.

HDF5 is developed at the National Center for Supercomputing Applications (NCSA). As a portable file format and software, HDF5 could store multidimensional arrays with ancillary data in a self-describing file format. HDF5 supports parallel I/O and is designed for storing, retrieving, analysing scientific data, etc. Parallel access could use MPI-IO by setting the file access property.

JPEG-2000 is an image compression standard which is typically utilized to store natural and computer generated images of any bit depth and colour space (i.e. 16-bit grey scale images). The standard is based on the wavelet transform, which is typically computed on the entire dataset. However, we are planning to exploit JPEG-2000's capability of decomposing the dataset into so called tiles, which can be distributed among separate CPUs. This tile based parallelization might impair the overall reconstruction quality of the numerical dataset by introducing so called block artefacts, but we hope that this approach will drastically reduce the computational time and speed up the overall I/O.

8 Conclusion and Future Work

In this deliverable, we have seen that there are three strands to efficient implementations that show promise towards the goal of future efficient use of exascale machines: improvements to algorithm implementations; methods to reduce energy to solution without adversely affecting computation time; and improvements in I/O allowing for a reduction in disk space and bandwidth required.

As this deliverable is mid-way through the project, there are number areas of future work for the partners involved in this deliverable:

- Other key features required for the simulation of the compressible NACA-4412 use-case, such as characteristic and wall boundary conditions, support for generalised coordinates, and multi-block capabilities.
- The evaluation of the performance of the various algorithms discussed in **[17]** on other architectures such as GPUs.
- Extending the nonconforming solver to support curved external element boundaries. In this case we have used Gordon-Hall transformation to properly shift elements internal degrees of freedom according to the face deformation.
- Concurrently, re-implementing the hybrid multigrid-Schwarz pre-conditioner for nonconforming meshes.
- The Hypre code, mainly on parallelisation, therefore reducing the setup time further. In a second phase, the setup might be integrated directly in Nek5000 and be performed online. A complete integration would enable the use of the AMG solver in the framework of adaptive mesh refinement.
- Exploration of data analysis strategies, which are developed in WP1, will be implemented ExaFLOW's use cases to analyse parallel I/O.
- Further additions to the HDF5 support in Nektar++ to reduce time taken in checkpoint/restart operations.

9 Bibliography

1. p4est http://www.p4est.org/
2. ParMETIS http://glaros.dtc.umn.edu/gkhome/metis/parmetis/overview/
3. Peplinski, A., Fischer, P. F., Schlatter, P. Parallel Performance of H-type Adaptive Mesh Refinement for Nek5000, in Proceedings of the Exascale Applications and Software Conference 2016, Stockholm, Sweden
4. Kovasznay, L. Laminar flow behind a two-dimensional grid, Proc. Cambr. Philos. Soc. 44, 58–62, 1948
5. Mavriplis, C. A Posteriori Error Estimators for Adaptive Spectral Element Techniques, in Proceedings of the Eighth GAMM-Conference on Numerical Methods in Fluid Mechanics, 333–342. Vieweg+Teubner Verlag, Wiesbaden, 1990
6. OpenSBLI https://github.com/opensbli/opensbli
7. Jacobs et al. The Journal of Computational Science: https://dx.doi.org/10.1016/j.jocs.2016.11.001
8. Jammy *et al.* (In Press), http://dx.doi.org/10.1016/j.jocs.2016.10.015
9. Hypre user's manual.
10. Hypre 2.11.1. http://computation.llnl.gov/projects/hypre-scalable-linear-solvers-multigrid-methods/software.
11. P Fischer, J Lottes, D Pointer, and A Siegel. Petascale algorithms for reactor hydrody-namics. Journal of Physics: Conference Series, 125(1):012076, 2008.
12. P.F. Fischer. An overlapping Schwarz method for spectral element solution of the incompressible Navier-Stokes equations. J. of Comp. Phys., 1997.
13. James William Lottes. Towards Robust Algebraic Multigrid Methods for Nonsymmetric Problems. PhD thesis, University of Oxford, 2015.
14. H.M Tufo and P.F Fischer. Fast parallel direct solvers for coarse grid problems. Journal of Parallel and Distributed Computing, 61(2):151 { 177, 2001.
15. Ivanov, I., Gong, J., Akhmetova, D., Peng, I. B., Markidis, S., Laure, E., … Fischer, P. (2015). Evaluation of parallel communication models in Nekbone, a Nek5000 mini-application. *Proceedings - IEEE International Conference on Cluster Computing, ICCC*, *2015–Octob*, 760–767.
16. PAT MPI library https://github.com/cresta-eu/pat_mpi_lib
17. S. P. Jammy, C. T. Jacobs, D. J. Lusher, N. D. Sandham (Submitted). Energy efficiency of finite difference algorithms on multicore CPUs and Intel Xeon Phi processors, *ParCFD 2017 Extended Abstracts*.
18. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt