Supercomputing Frontiers and Innovations

2020, Vol. 7, No. 2

Scope

- Enabling technologies for high performance computing
- Future generation supercomputer architectures
- Extreme-scale concepts beyond conventional practices including exascale
- Parallel programming models, interfaces, languages, libraries, and tools
- Supercomputer applications and algorithms
- Distributed operating systems, kernels, supervisors, and virtualization for highly scalable computing
- Scalable runtime systems software
- Methods and means of supercomputer system management, administration, and monitoring
- Mass storage systems, protocols, and allocation
- Energy and power minimization for very large deployed computers
- Resilience, reliability, and fault tolerance for future generation highly parallel computing systems
- Parallel performance and correctness debugging
- Scientific visualization for massive data and computing both external and in situ
- Education in high performance computing and computational science

Editorial Board

Contents

Performance Reduction for Automatic Development of Parallel Applications for Reconfigurable Computer Systems
A.I. Dordopulo, I.I. Levin 4

Long Distance Geographically Distributed InfiniBand Based Computing
K. Niedzielewski, M. Semeniuk, J. Skomial, J. Proficz, P. Sumionka, B. Pliszka, M. Michalewicz 24

Potential of I/O Aware Workflows in Climate and Weather
J.M. Kunkel, L.R. Pedro 35

Bridging the Architecture Gap: Abstracting Performance-Relevant Properties of Modern Server Processors
J. Hofmann, C.L. Alappat, G. Hager, D. Fey, G. Wellein 54

Dawn: a High Level Domain-Specific Language Compiler Toolchain for Weather and Climate Applications
C. Osuna, T. Wicky, F. Thuering, T. Hoefler, O. Fuhrer 79

DOI: 10.14529/jsfi200201

Performance Reduction for Automatic Development of Parallel Applications for Reconfigurable Computer Systems

Aleksey I. Dordopulo[1], *Ilya I. Levin*[2]

In the paper, we review a suboptimal methodology of mapping of a task information graph on the architecture of a reconfigurable computer system. Using performance reduction methods, we can solve computational problems which need hardware costs exceeding the available hardware resource. We proved theorems, concerning properties of sequential reductions. In our case, we have the following types of reduction such as the reduction by number of basic subgraphs, by number of computing devices, and by data width. On the base of the proved theorems and corollaries, we developed the methodology of reduction transformations of a task information graph for its automatic adaptation to the architecture of a reconfigurable computer system. We estimated the maximum number of transformations, which, according to the suggested methodology, are needed for balanced reduction of the performance and hardware costs of applications for reconfigurable computer systems.

Keywords: performance reduction, hardware costs, reconfigurable computer system, parallel applications development, information graph.

Introduction

Most researchers of parallel computing [1–4] admit that parallel programming is a complex area. It is necessary to organize and control a large number of processes that asynchronously run on the nodes of a multiprocessor computer system (MCS). The demanding requirements are decreasing of the calculation time and increasing of the results accuracy. To fulfill these requirements, we increase the number of nodes of a multiprocessor computer system, but at the same time, development of parallel programs becomes more complex.

For a long time, we believed that it is possible to cope with the growing complexity of parallel program development with the help of automatic parallelization of sequential processor (procedural) programs. In this case, a parallelizing compiler [1, 2, 5–10] receives an imperative processor program, reconstructs the natural parallel structure of its initial algorithm, detects its fragments for concurrent execution (e.g. loop iterations suitable for parallelization), and adds all necessary instructions. However, the automatic parallelization of sequential programs is a computationally expensive problem with an extremely large number of variants for analysis.

The parallelizing compiler has to analyze different variants of multiple fragments of the procedural program. At the same time, it analyses distribution of data among the nodes of the multiprocessor computer system according to its switching network. These two reasons complicate automatic parallelization for clusters that are the most widely used multiprocessor computer systems with distributed memory.

Let us have a cluster computer system, which consists of n nodes, and each node processes its local part of data. In this case, we describe the data distribution among the nodes using an n-ary tree. According to the Cayley theorem, we estimate the number of variants of data distribution as the number of different trees for n vertices, i.e. n^{n-2}. For example, the cluster MCS consists of 64 nodes. So, the number of possible distribution variants is $64^{62} = 2^{372}$. Analysis of such number of variants on any existing computer system and during any reasonable time is impossible.

[1]Supercomputers and Neurocomputers Research Center, Taganrog, Russia
[2]Southern Federal University, Taganrog, Russia

Therefore, the most part of research in this problem domain was devoted to heuristic methods of search space reduction (e.g. the analysis of information dependencies [1], loop nests and iterations [4–6], private and reduction variables [7, 8], canonization, loop unrolling/unwinding, loop fusion, loop distribution [9, 10], etc.). Formal transformations and heuristic methods, developed for rejection of inefficient parallel program variants [3], require some recommendations and instructions given by the programmer; otherwise, they cannot provide efficient automatic parallelization of any procedural program.

Nowadays, multi-chip reconfigurable computer systems (RCS) [11] with field-programmable gate arrays (FPGAs) are widely used for solving of computationally expensive problems in various fields of science and technology. RCSs contain multiple FPGAs of a large logical capacity. The FPGAs are connected by a spatial switching system into a single computational field. Within such computational field, we implement calculations as a computing structure [12–14] and decrease the solution time [15, 16] by one or two orders of magnitude at the considerably lower (by a factor of 6–8) processing rate. For certain problem domains [17, 18], RCSs are considerably superior in real performance and power efficiency in contrast with cluster MCSs.

In the paper, we consider a theory which helps to reduce the number of variants parallel calculations for analysis and further synthesis of a computing structure for an RCS. We represent a task as an information graph and then, using performance reduction methods and a relatively small number of steps, we transform it into the form, similar to the architecture of an RCS. For most applications, it is possible to synthesize computing structures and to increase the task solution time owing to the performance reduction methods. In this case, the efficiency of the designed structures is not less than 50 % in comparison with those designed by circuit engineers.

Let us review the structure of the paper. In the first section we describe the forms of parallel calculations, and the task information graph used for structural and procedural calculations on the RCS. In the second section, we consider performance reduction as a way of implementing of the task information graph on the RCS with the lack of its hardware resource. In the third section, we represent the performance reduction methods for decreasing of hardware costs, required for implementing of the information graph, and prove theorems on the applicability of reduction transformations. In the fourth section, we represent the performance reduction principles for mapping of the task information graph on the RCS architecture. Besides, here we estimate the number of computing structures that are to be analysed for adaptation of the initial task information graph to the architecture and hardware resource of the RCS. In the fifth section, we describe the rules, according to which we use the reduction transformations in an experiment for verification of our performance reduction methods. The rules were used in tools for parallel application development. In the conclusion we generalize our results and discuss the directions of our future research.

1. Forms of Calculations

According to the form of calculations, we can reveal the natural parallelism of the task [1, 19]. As a result, it simplifies the task solution and scaling. Usually, parallel calculations are represented in two most common forms – an algorithmic diagram (a flowchart) and a graph [1, 19]. According to the algorithmic diagram [1], calculations are a control transfer among computing devices. Besides, the algorithmic diagram defines the order (or the sequence) of operations performed by a computing device (or devices) using processor instructions. Generally, the algorithmic diagram is the task flowchart or pseudocode, but sometimes it is a control flow graph [1, 5, 8]

When we represent *calculations* as a graph, we describe a task or its fragment in an absolutely parallel form, i.e. as an acyclic oriented graph with input, output, and operation vertices connected by arcs according to the data processing order (but not according to the control transfer). There are various forms of graph models for computational tasks such as algorithm graphs [1], information graphs [1, 19], dependency and influence graphs, and lattice graphs [1]. Arcs of a graph show, how arguments of operation vertices depend on results of calculations, performed by other operation vertices, or arguments, received from input vertices. This is an information connection (or information dependence) that describes relations between two vertices of a graph when the output argument of each vertex is the input of another one. If we speak about multiprocessor architectures [1], then information dependence between two operators means addressing to the same memory cell during their execution. If we speak about dataflow architectures, then it means addressing to one and the same element of a flow.

The most common forms are an algorithm graph [1] and an information graph [1, 19]. The algorithm graph describes a computational task as a set of simple operations (addition, multiplication, division, etc.) distributed into levels. Although, it is possible to use complex composite operations (macro operations) as level vertices [1]. All vertices of the algorithm graph, represented in the canonical parallel form, are distributed into numbered subsets, which form levels. Here, the first vertex of each arc belongs to the level, whose number is less than the number of the level, which contains the last vertex. Besides, arcs cannot connect vertices which belong to the same level.

The theory of structural and procedural calculations [19] deals with a task information graph (TIG). In contrast to the parallel forms of an algorithm graph, the task information graph is a combination of layers and iterations. A layer consists of isomorphic, functionally complete, and information independent subgraphs of a task instead of operation vertices. Iterations describe dependencies among processing data over time without considering latency. Subgraphs from one and the same layer are information independent, i.e. not connected by arcs. Subgraphs, which belong to different iterations, depend on processing data. The number of isomorphic subgraphs in a layer is similar to the level width of the canonical parallel form, and the number of iterations is similar to its height, if we consider isomorphic subgraphs as macro-operations. In comparison with an algorithm graph, a TIG describes a task at a higher level of hierarchy. In this case, we use separate operations, but subgraphs which consist of several operations. The information graph describes the absolutely parallel form of the task. The task parameters define the number of iterations. Therefore, the TIG has no dataflows.

A structural implementation of a TIG on a computer system provides the highest performance. In this case, the number of devices is equal to the number of operation vertices (or operations) of a solving task, and the number of input/output arcs is equal to the number of external memory channels. For the majority of applications, such structural implementation of a TIG is impossible, because the number of devices and channels in any RCS is limited. Therefore, if we map a TIG on a real RCS with its limited hardware resource, we transform this TIG into a computing structure with the lower performance and lower requirements for the number of channels, the number of concurrently functioning devices, and/or the data width in comparison with the structural implementation of this TIG.

A TIG (or its subgraph) describes the logical structure of calculations as vertices and arcs. To implement the TIG on an RCS means to create its computing structure, which consists of hardware-programmed devices with timing characteristics such as latency, data processing

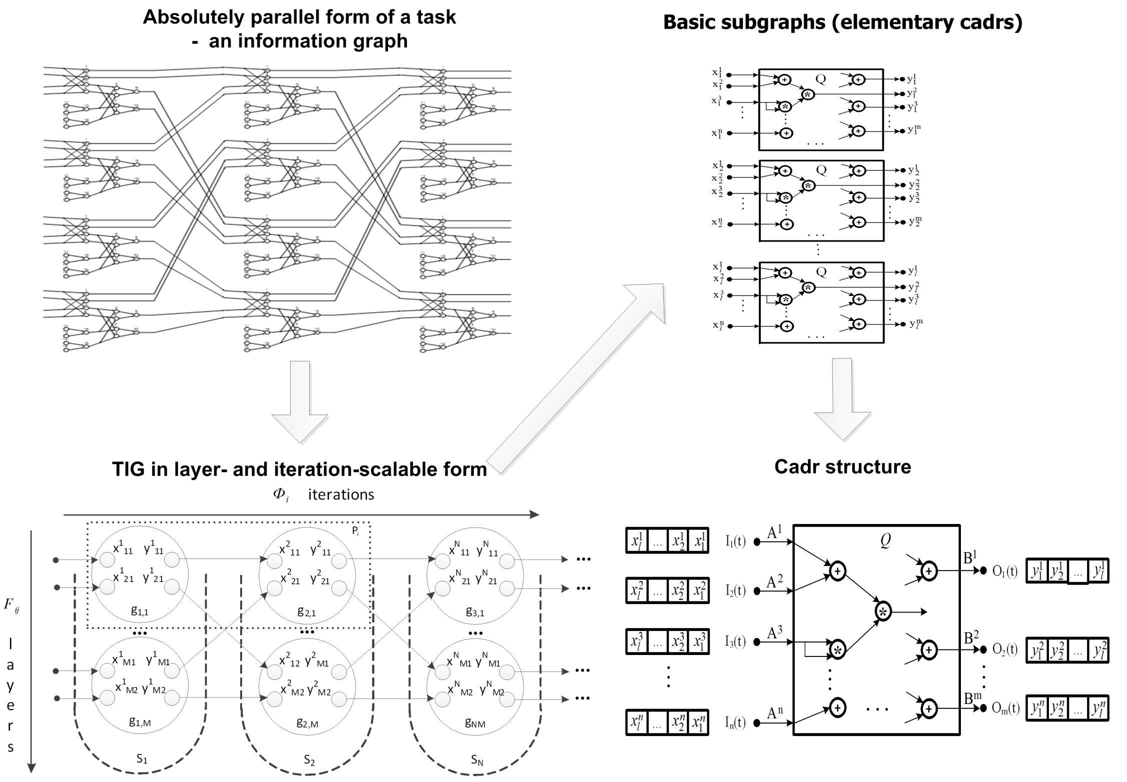

Absolutely parallel form of a task - an information graph
Basic subgraphs (elementary cadrs)
TIG in layer- and iteration-scalable form
Cadr structure
Φi iterations
Fij
layers
S1
S2
SN
Q

Figure 1. Transformation of a TIG into the computational structure via layer- and iteration-scalable form

interval, clock rate, etc. We assume, that the term "implementation of a subgraph on a computer system" means a computing structure which consists of hardware-programmed devices with timing characteristics (or so-called timing component).

Figure 1 shows the transformation a TIG for its structural implementation on an RCS.

To transform the absolutely parallel form of a TIG into the layer- and iteration-scalable form, we obtain the functionally regular form [19] with functions of layer mapping between iterations Φ_i, and functions of isomorphic subgraph ordering in a layer F_{ij}:

$$G = \Phi(F_{ij}(g_{ij})), \tag{1}$$

where g_{ij} is a basic subgraph (a pipeline computing structure); F_{ij} is an ordering function for information-independent subgraphs in a computational layer; Φ_i is a mapping function of information-dependent layers. The composition of functions F_{ij} and Φ_i depends on an available RCS hardware resource A_{RCS}.

Figure 2 shows the task information graph, which consists of information-dependent layers $S_1, ..., S_N$. Each layer consists of isomorphic information-independent subgraphs $G_{1,1}, ..., G_{1,M}, ...G_{N,M}$.

Owing to such form, we easily scale the task computing structure. If we change the number of basic subgraphs g_{ij} in the composition of the functions F_{ij} and Φ_i, then we scale the computing structure both by layers and by iterations. If we increase the number of hardware-programmed information-independent subgraphs within the layer, then we scale the computing structure by layers. If we increase the number of hardware-programmed subgraphs with information dependence among iterations, then we scale the computing structure by iterations.

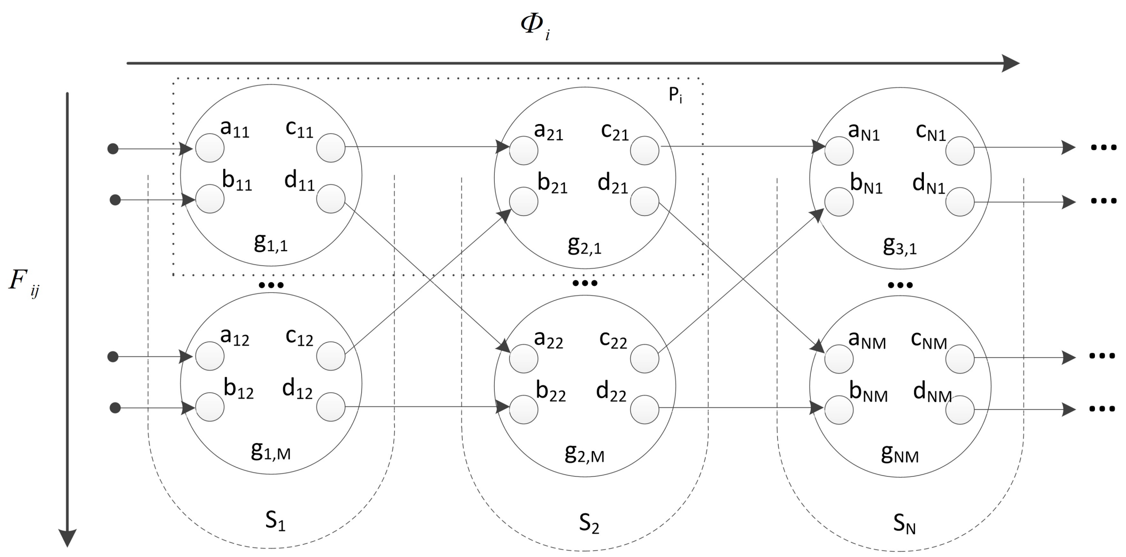


Figure 2. The information graph, its layers F_{ij} and iterations Φ_i

A basic subgraph g_{ij} is a minimal indivisible element of a task. When its computing structure is mapped on an RCS, it is completed with functions of reading, writing, and recursion, derived from F_{ij} and Φ_i functions. The obtained indivisible program structure is called a cadr. For all obtained cadrs we specify an order relation, which, together with the von-Neumann determinism, define the execution sequence of cadrs according to their control program.

A basic subgraph is a functionally completed fragment of a TIG. It consists of subgraphs of one or several subtasks. It is possible to map any basic subgraph on an available RCS hardware resource. Completed with the synthesized read/write functions, a basic subgraph provides solution of a task. Within the theory of structural and procedural calculations, basic subgraphs are selected according to available hardware resource. In this case, selection criteria are not formalized; they are determined by the structure of a task, by available resource, and by the developers experience. To select a basic subgraph, the developer analyzes the TIG and looks for frequently used fragments of the TIG which are typical for a certain problem area. Here are the examples of such frequently used fragments:

- addition, multiplication, and division of matrix elements (linear algebra);
- calculations in mesh points (mathematical physics);
- round transformations with logical “AND”, “OR”, “exclusive OR”, and fixed-size data block offset (symbolic processing);
- the discrete fast Fourier transform operation (digital image and signal processing).

Usually, these standard fragments form basic subgraphs of various tasks. We can select basic subgraphs in procedural programs, using descriptions of loops, because fragments with cyclic processing correspond to functional subgraphs, i.e. to calculations with specified scaling functions by layers F_{ij} and by iterations Φ_i. Here, the operators of a loop body are a basic subgraph. Information dependencies among operators, and cycle description determine the functions of layers F_{ij} and by iterations Φ_i. As a rule, any basic subgraph consists of multiple functional subgraphs, and is a broader concept. However, for some tasks a functional subgraph and a basic one are the same.

2. Mapping of Information Graphs on Reconfigurable Computer Systems

The available hardware resource A_{RCS} defines not only mapping functions F_{ij} and Φ_i, but also the calculations of the basic subgraph. That is why, we can represent (1) as

$$G(A_{RCS}) = \Phi_{par} \overset{A_{RCS}}{\circ} \Phi_{pipe}(F_{par} \overset{A_{RCS}}{\circ} F_{pipe}(g_{str} \overset{A_{RCS}}{\circ} g_{proc})), \tag{2}$$

where "*par*" and "*pipe*" mean parallel and pipeline execution, respectively; g_{str} is the structural form of the basic subgraph; g_{proc} is the procedural form of the basic subgraph; $\overset{A_{RCS}}{\circ}$ is composition of scaling functions, which depends on the available hardware resource A_{RCS}.

Using the dependence between the basic subgraph and the available hardware resource (2), we can describe not only extreme variants of completely structural (g_{str}) and completely procedural (g_{proc}) calculations, but other intermediate ones. However, we cannot obtain the structural form g_{str} for some tasks due to hardware resource limitations, and the procedural form cannot provide results of adequate accuracy in reasonable time. The examples of such tasks are:

- molecular simulation (docking of inhibitors);
- synthesis of new chemical compounds;
- 3D simulation of spatial physical processes (e.g. tomography of the Earth surface);
- high-resolution simulation of physical processes;
- symbolic processing, etc.

Tasks with variable data flow density [20] belong to this type also. For such tasks, the amount of processed data in various TIG subtasks may differ by 2–4 decimal places, and may depend on input data. For such tasks, basic subgraphs from different layers are significantly non isomorphic. If we try to transform them into isomorphic subgraphs, using the union operation, then we need an inaccessible hardware resource for their structural (or structural-procedural) variant. Hence, we cannot solve these tasks using structural, structural-procedural, or procedural calculations.

If we want to solve these tasks during some reasonable time and using some available hardware resource, it is necessary to reduce the hardware costs for g_{str}, not using the completely procedural variant g_{proc}, in order to create the basic subgraph within an RCS, and to provide the specified task performance. Here, the task performance is lower than the one for the structural variant g_{str}, but higher than the one for the procedural variant g_{proc}.

Therefore, we consider a basic subgraph as a scalable, not an atomic object of a task. If we reduce the performance and hardware costs, then it is possible to fulfill all requirements of the task and solve it.

For the first time [20], it was suggested to use performance reduction methods for decreasing of hardware costs in case, when RCS hardware resource is insufficient for even one basic subgraph. The main effect of performance reduction is a linear increase of the task solution time, proportional to the reduction coefficient. The main reduction transformations, which provided balanced scaling of molecular docking tasks in [20], are the following:

- R^N – the reduction by number of basic subgraphs. It decreases the number of computing structures, simultaneously mapped on RCS.
- R^{Op} – the reduction by number of computing devices. It decreases the number of concurrently performed operations of the basic subgraph. Similar operations and data of similar

types are combined in one device. Besides, new connections for operands synchronizations are synthesized.

- R^{ρ} – the reduction by data width. It decreases the number of concurrently processed digits. Absolutely parallel processing of digits in each operand is transformed in partly parallel or sequential processing.
- R^{S} – the reduction by data processing interval. It increases the data processing/supply interval; the hardware costs remain unchanged. This type of reduction is used for matching of data flow with different density among different subtasks or information graph fragments.
- R^{Freq} – the reduction by clock rate. It decreases the clock rate of a computing structure, which implements some information graph fragment, and matches data flows with different density.

In [20], all reduction transformations were used to reduce hardware costs for solution of some task. However, we can consider the methods of performance reduction as transformations, which provide scaling of a TIG as a computing structure for further mapping on RCS architecture. Moreover, we often use the reduction transformations to get a computing structure from the absolutely parallel form of a task. As a result, the performance of the obtained computing structure is lower, but the structure requires less number of channels and simultaneously operating devices, and/or less data width. That is why we may consider the computing structure of a TIG, mapped on RCS architecture, as performance reduction. Of course, this is true only in case, when RCS hardware resource is insufficient for task solution.

We efficiently use the methods of performance and hardware costs reduction for information graph mapping on RCS architectures. These methods provide automatic (without the programmers instructions) adaptation of applications to various RCS architectures, and solve the problem of application portability.

3. Methods of Performance and Hardware Costs Reduction

The task solution performance is a number of computing operations performed per time unit during execution of an application. Let us have a computing structure F with N_F basic subgraphs. Each basic subgraph contains Op_F computing devices, which process data of a ρ_F width. The total number NC_F of computing operations, required for processing of a data flow with a length N, is

$$NC_F = N \cdot N_F \cdot Op_F \cdot \rho_F. \tag{3}$$

The task solution time for a computing device with a clock period $\tau = 1/Freq$ and with an interval S is $t = N \cdot S \cdot \tau$. Here, the data processing interval is measured in cycles. Then, the performance of the computing structure F is defined as

$$Perf_F = \frac{NC_F}{t} = \frac{N \cdot N_F \cdot Op_F \cdot \rho_F}{N \cdot S \cdot \tau} = \frac{N_F \cdot Op_F \cdot \rho_F}{S \cdot \tau} = \frac{N_F \cdot Op_F \cdot \rho_F \cdot Freq}{S}. \tag{4}$$

If we carry out the performance reduction with the integer reduction coefficient R, then the performance (2) is reduced by R times:

$$Perf_F(R) = \frac{Perf_F}{R} = \frac{N_F \cdot Op_F \cdot \rho_F}{S \cdot \tau \cdot R} = \frac{N_F \cdot Op_F \cdot \rho_F \cdot Freq}{S \cdot R}. \tag{5}$$

The balance of a result computing structure is one of the main and the most important distinction of the performance reduction methods. It means that data flows and hardware costs for their switching and synchronization are multiply scaled. Concerning (5), it means that the cofactors of the numerator are reduced by the reduction coefficient R (or by its prime cofactors). According to (5), we can reduce the performance of a task computing structure by:

- decreasing of the number N_F of hardware-programmed basic subgraphs in proportion to R (or its prime cofactors). For each mapped BS, the length N of its processed data flow increases. This method is traditional for scalable calculations, performed on RCS and clusters;
- reducing of the number Op_F of computing devices in the task basic subgraph [20] in proportion to R(or its prime cofactors). The number of operations, performed by each computing device, and the number of data processing cycles increase. This method is used for RCS;
- decreasing of the processed data width ρ_F in proportion to R (or its prime cofactors). The method is used for fixed-point data, and used with restrictions for floating-point data. The number of processing cycles multiply increases. At the same time, the number of data processing channels multiply decreases. This reduction is used in case of lack of input data channels (the most typical case for RCS);
- increasing of the data processing interval S;
- decreasing of the clock rate $Freq$ [20].

In the first, second, and third cases, we reduce both the performance and the hardware costs for the computing structure F, if the switching and synchronization costs do not exceed the reduced resources. If we use the two last methods, we only reduce the performance of a task or its fragment. In this case, the hardware costs remain unchanged. We can use these methods for matching of data processing rates in different task fragments.

So, we reduce the hardware costs and the number of RCS channels, needed for the computing structure F, only if the hardware costs for switching and synchronization do not exceed the reduced resources.

The performance reduction methods without hardware costs reduction are the following:

- the reduction by clock rate;
- the reduction by data processing interval.

A multiple integer, unified for all task fragments automatic performance reduction (5) provides a balanced computing structure. Thus, all task fragments are to be reduced not only with the same reduction coefficient R, but the types and coefficients of performed reductions are to be the same. However, for real tasks such requirement is almost impossible.

If we reduce the performance in order to decrease the hardware costs, then all types of reduction transformations are performed in a balanced manner. Here, the reduction coefficient is a positive integer not less than unity. Owing to the reduced computing structure, we can solve the task on lesser hardware resource with longer solution time (in proportion to the reduction coefficient).

In order to describe all reductions of the modifying computing structure, we suggest to use an operation, which rounds rational numbers down to unity [21]. For natural numbers $a \geq 1$ and $b \geq 1$, the operation is defined as

$$\left\lfloor \frac{a}{b} \right\rfloor_1 = \begin{cases} a \ \mathbf{div} \ b, a > b, \\ 1, a \leq b, \end{cases} \tag{6}$$

where **div** is integer division; $\lfloor\ \rfloor_1$ is similar to the standard floor notation $\lfloor\ \rfloor$ [21], and it indicates that the result of the "floor" operation is bounded below by unity.

The result of the floor operation $\lfloor\ \rfloor_1$ corresponds to the physical meaning of the parameters that are being reduced, because the number of basic subgraphs, computing devices, and processed digits cannot be less than unity after the reduction. The traditional "floor" operation $\lfloor\ \rfloor$ has a useful property given in [21]. For the real numbers m, x and the natural number n

$$\left\lfloor \frac{\lfloor \frac{x}{m} \rfloor}{n} \right\rfloor = \left\lfloor \frac{x}{m \cdot n} \right\rfloor . \tag{7}$$

Since the set of natural numbers is a subset of the set of real numbers, and function (4) is monotonic and continuous, equality (7) is valid for the proposed function, too. Taking into account the commutative law, we obtain:

$$\left\lfloor \frac{\lfloor \frac{x}{m} \rfloor_1}{n} \right\rfloor_1 = \left\lfloor \frac{\lfloor \frac{x}{n} \rfloor_1}{m} \right\rfloor_1 = \left\lfloor \frac{x}{m \cdot n} \right\rfloor_1 . \tag{8}$$

Taking into account (8), we prove the following important theorem, which represents the reduction coefficient as a production of coefficients for the sequential reduction transformation. We denote sequential reduction by $\times$.

For example, the sequential reductions by number of basic subgraphs and by number of computing devices we represent as $R_n^T \times R_{Op}^T = R_{n \cdot m}$.

Theorem 1.

Sequential T-type reductions $R_m^T$and R_n^T with natural coefficients $m > 1$ and $n > 1$ are equivalent to the reduction $R_{n \cdot m}^T$ of the same type with a coefficient $(m \cdot n) > 1$:

$$R_n^T \times R_m^T = R_{n \cdot m}^T . \tag{9}$$

Proof. Let F be a task fragment which contains N_F basic subgraphs. Each basic subgraph contains Op_F computing devices and processes data with a width ρ_F. The total amount of calculations NC_F in F is

$$NC_F = N_F \cdot Op_F \cdot \rho_F . \tag{10}$$

Since reduction transformations are independent, then we prove (9) for each type of reduction.

Let us prove condition (9) for the reduction R_n^N by the number of basic subgraphs with the reduction coefficient n. The number of basic subgraphs in F is reduced to $\lfloor \frac{N_F}{n} \rfloor_1$, and the total amount of calculations NC_n^N is:

$$NC_n^N = \left\lfloor \frac{N_F}{n} \right\rfloor_1 \cdot Op_F \cdot \rho_F . \tag{11}$$

The sequential reduction R_m^N of the same fragment provides m-fold decreasing of its number of basic subgraphs. According to (8), we transform (11) and obtain

$$NC_{n \times m}^{N \times N} = \left\lfloor \frac{\lfloor \frac{N_F}{n} \rfloor_1}{m} \right\rfloor_1 \cdot Op_F \cdot \rho_F = \left\lfloor \frac{N_F}{n \cdot m} \right\rfloor_1 \cdot Op_F \cdot \rho_F . \tag{12}$$

The total amount of calculations (12), which we obtain as results of the sequential reductions by number of basic subgraphs with the coefficients n and m, and as results of the reduction $R_{n \cdot m}^N$

by number of basic subgraphs with $n \cdot m$ instead of n in (11), have the same value. This fact proves Theorem 1. In a similar way, we prove (9) for the reduction by number of computing devices, and for the reduction by data width. As a result, we prove Theorem 1 in general. ■

Let us formulate several corollaries for application of reduction transformations.

Corollary 1.1 of Theorem 1. Using factorization of performance reduction coefficients, we decrease the number of steps, required for selection of reasonable coefficients for sequential reductions of the same or different types. Besides, for the specified reduction coefficient we choose the best suited type of reduction transformations according to the parameters of a solving task. If the performance reduction coefficient R is a prime number, which exceeds 2, and if we cannot obtain it by a single reduction, then it is reasonable to perform not an R-fold, but an $(R+1)$-fold reduction. In this case, we obtain fully R times lower hardware costs. Since $(R+1)$ is an even composite number, we sequentially perform reduction transformations with reduction coefficients taken from the prime factorization of $(R+1)$.

Corollary 1.2 of Theorem 1. There is no need to return to the initial basic subgraph, when the reduction coefficient multiply increases during sequential reduction of one and the same. If the result of a reduction transformation is a computing structure, which requires additional multiple (not less than twofold) decreasing of hardware costs, and its reduction type permits multiple increasing of its coefficient, then, according to Theorem 1, sequential reduction with no return to the initial basic subgraph lessen the number of steps to get a final reduced structure.

According to Theorem 1, Corollaries 1.1 and 1.2, the total coefficient of sequential reductions equals to a product, but not to an algebraic sum of reduction coefficients. Therefore, it is impossible to get a reduced computing structure with a coefficient from a structure with a coefficient $(n+1)$ using sequential reductions of any type. Let us prove this statement (or Theorem 2) more strictly for a generalized case with a reduction coefficient $(n+x)$.

Theorem 2.

In the general case, for a basic subgraph reduced with a coefficient n, we cannot obtain a computing structure with a reduction coefficient $(n+x)$ for a prescribed $x \geq 1$, using sequential reductions of a type T with a natural coefficient $k > 1$:

$$R_n^{T1} \times R_k^{T2} \neq R_{n+x}^{T}. \tag{13}$$

Proof. According to Theorem 1, it is possible to fulfil (13) for reductions of the same type T only if

$$n \cdot k = n + x \tag{14}$$

is valid.

Then, we transform (14) and obtain

$$k = 1 + \frac{x}{n}. \tag{15}$$

According to Theorem 2, the numbers n, k and x are positive integers. So, we solve (15) for k only when x is integrally divided by n, but not $\forall x \geq 1$. This fact proves Theorem 2.

If we perform reductions of different types, then the similar computing structures from the left and right sides of (13) have the same total amount of operations

$$NC_{n\times k}^{T1\times T2} = NC_{n+k}^{T}. \tag{16}$$

Therefore,

$$\left\lfloor \frac{NC}{n \cdot x} \right\rfloor_1 = \left\lfloor \frac{NC}{n + x} \right\rfloor_1, \tag{17}$$

which requires

$$n \cdot k \geq n + x \quad \text{and} \quad n + x \geq n \cdot k \tag{18}$$

to be fulfilled.

Both conditions are true only if

$$n \cdot k = n + x. \tag{19}$$

So,

$$k = 1 + \frac{x}{n}. \tag{20}$$

Under hypothesis of Theorem 2, n, k, and x are positive integers. Hence, the solution of (20) for k is possible in the natural domain only when x is integrally divisible by n, but not when $\forall x \geq 1$. This conclusion leads to contradiction and, as a result, proves Theorem 2. ■

Using Theorem 2, we formulate a corollary which is important for application of a sequence of reduction transformations.

Corollary 2.1 of Theorem 2. For a reduced structure, it is impossible to increase the reduction coefficient by an arbitrary value, performing sequential reductions of any types. Hence, in general case, if additional reduction (of hardware costs) is needed, we return to the initial basic subgraph and perform reduction transformations again with a new (increased) reduction coefficient R. As a result, we need more steps to obtain the reduced computing structure.

Let us analyse, how a sequence of reduction transformations of various types influences on a final computing structure.

Theorem 3.

The superposition of reductions of different types (e.g. a reduction $T1$ with a coefficient n, and a reduction $T2$ with a coefficient m) is commutative. Therefore, if we change the order of reductions of different types for a task fragment, then the result information graph of the fragment remains unchanged:

$$R_n^{T1} \times R_m^{T2} = R_m^{T2} \times R_n^{T1},$$

where T_1 and T_2 are the types of reduction transformations; n and m are the reduction coefficients.

Proof. Let us prove commutativity of sequential reductions. The first reduction is performed by number of basic subgraphs, and the second one – by number of computing devices:

$$R_n^N \times R_m^{Op} = R_m^{Op} \times R_n^N. \tag{21}$$

After the reduction R_n^N, which is performed by number of basic subgraphs and has the reduction coefficient n, the total amount of calculations NC_n^N over binary digit bits is

$$NC_n^N = \left\lfloor \frac{N_F}{n} \right\rfloor_1 \cdot Op_F \cdot \rho_F. \tag{22}$$

Since the number of basic subgraphs and the number of computing devices in each basic subgraph are independent values, then the sequential reduction R_m^{Op} by number of computing devices with the coefficient m decreases only the number of devices, and the total amount of calculations over binary digit bits is

$$NC_{n\times m}^{N\times Op} = \left\lfloor \frac{N_F}{n} \right\rfloor_1 \cdot \left\lfloor \frac{Op_F}{m} \right\rfloor_1 \cdot \rho_F. \tag{23}$$

For the right side of (21), the sequential reductions R_m^{Op} and R_n^N lead to the same total amount of calculations over binary digit bits:

$$NC_{m\times n}^{Op\times N} = \left\lfloor \frac{N_F}{n} \right\rfloor_1 \cdot \left\lfloor \frac{Op_F}{m} \right\rfloor_1 \cdot \rho_F. \tag{24}$$

We prove commutativity for all other possible combinations of sequential reductions in the same way. As a result, this fact proves Theorem 3. ■

Using Theorem 3, we define a corollary for estimation of the number of reduction steps.

Corollary 3.1 of Theorem 3. If the order of reduction transformations is changed, it is not necessary to return to the initial basic subgraph in order to decrease the number of steps.

4. Performance Reduction Methods for Information Graphs Mapping on Reconfigurable Architectures

Taking into account the proved theorems and corollaries, let us formulate the main rules information graph adaptation to RCS architectures.

1) To decrease the number of steps of reduction transformations, it is reasonable to choose coefficients of each type of reduction from the prime factorization of the reduction coefficient.

2) If the number of basic subgraphs in an information graph is more than 1, then it is reasonable to perform the reduction by number of basic subgraphs as the first step of reduction transformations.

In this case, we linearly decrease the required hardware resource such as the number of FPGA logic cells, and the number of channels for data parallelization.

3) If we perform the reduction by number of computing devices, and by data width to decrease the number of steps of reduction transformations, it is reasonable to perform reductions of each type until the value, specified by reduction criteria, is reached. After that, we perform reduction of another type. Here, the value is chosen according to the cofactors of the reduction coefficient of an information graph.

In this case, we reduce additional overhead of switching hardware.

Owing to the performance reduction methods, which we use for information graphs mapping on RCS architectures, it is possible to divide the set of parallelization variants into several classes that consist of isomorphic computing structures. As a result, we have few variants for analysis.

Let us estimate the number of steps of reduction transformations, which we need to adapt an information graph to a reconfigurable architecture. We consider the most general case, when it is necessary to perform all types of reduction transformations (by number of basic subgraphs, by number of computing devices, and by data width) for reduction of hardware costs.

To define the initial value of a performance reduction coefficient R of a computing structure, we use its approximate value the coefficient of necessary hardware costs reduction R_T, defined as a proportion of the hardware resource, needed for hardware-programmed information graph, to the available RCS resource A_{RCS}. The hardware resource A_T for hardware-programmed information graph is equal to the sum of hardware costs of all task subgraphs for each architecture component of an FPGA (the number of Look-UP Tables (LUTs), Memory LUTs (MLUTs), Flip-Flops (FFs), the number of Digital signal processor blocks (DSPs) and Block RAM (BRAMs)). For an RCS we use the parameters of FPGA chips as follows:

$$A_T = \{A_T^{LUT}, A_T^{MLUT}, A_T^{FF}, A_T^{DSP}, A_T^{BRAM}\},$$

$$A_{RCS} = \{A_{RCS}^{LUT}, A_{RCS}^{MLUT}, A_{RCS}^{FF}, A_{RCS}^{DSP}, A_{RCS}^{BRAM}\}. \tag{25}$$

A hardware costs reduction coefficient for each resource is a proportion of the hardware costs, needed for task solution, to the available resource. We select the task hardware costs reduction coefficient as the maximum value among the calculated values:

$$R_T = Max(\frac{A_T^{LUT}}{A_{RCS}^{LUT}}, \frac{A_T^{MLUT}}{A_{RCS}^{MLUT}}, \frac{A_T^{FF}}{A_{RCS}^{FF}}, \frac{A_T^{DSP}}{A_{RCS}^{DSP}}, \frac{A_T^{BRAM}}{A_{RCS}^{BRAM}}). \tag{26}$$

The initial value of the performance reduction coefficient R_0 is equal to the coefficient R_T, rounded up to the nearest integer: $R_0 = \lceil R_T \rceil$.

For linear and iterative computing structures, used in tasks of symbolic processing and linear algebra, respectively, the hardware costs reduction coefficient R_T and the performance reduction coefficient R_0 can be the same. In most cases, it turns out that for performance reduction with a coefficient, which is equal to the hardware costs reduction coefficient, it is necessary to increase the performance reduction coefficient even more, due to unforeseen switching costs. Since the overall reduction coefficient is nonadditive for sequential reduction (according to Theorem 2), then it is necessary to increment R_0 by 1, and to perform reduction with a new coefficient.

Performance reduction is carried out for the initial value of the performance reduction coefficient $R_0 > 1$, which, according to the fundamental theorem of arithmetic, and to Corollaries 1.1 and 3.1, is a product of prime cofactors:

$$R^0 = \prod_i r_{0i}. \tag{27}$$

To perform reduction transformations, taking into account the task parameters, and the prime factorization of the reduction coefficient R_0, we represent it as a product of three coefficients of reduction transformations:

$$R^0 = R_0^N \cdot R_0^{Op} \cdot R_0^{\rho}. \tag{28}$$

If R_0 is a prime number, we increment it by 1 according to Corollary 1.1.

Since in our case all reductions are performed, then all reduction coefficients R_0^N(by number of basic subgraphs), R_0^{Op}(by number of computing devices) and R_0^{ρ}(by data width) exceed unity.

In the first step, it is reasonable to perform the performance reduction by number of basic subgraphs with the coefficient R_0^N. In the second step, we perform the reduction by number of computing devices with the coefficient R_0^{Op}. The extreme case of subgraph reduction by number of computing devices means sequential execution of its operations as g_{proc} (2) in one device (a processor). If the coefficient R_0^{Op} is less than the number of devices in a subgraph, then, according to the type and number of used operations, several variants of computing structures are possible (with the different latency time and data supply interval). The reduced computing structure must provide data equivalency of results. Therefore, each of the considered variants contains devices that perform the operations of the basic subgraph, in order to perform all its operations within the reduced computing structure.

Let us consider reduction by number of devices for a basic operation of the fast Fourier transform with calculation of coefficients. Its information graph contains 16 operations such as 8 multipliers, 4 adders, and 4 subtractors (see Fig. 3a). Taking into account, that hardware-programmed addition and subtraction are identical, we claim that 8 multipliers and 8 adders are enough for the hardware-programmed information graph.

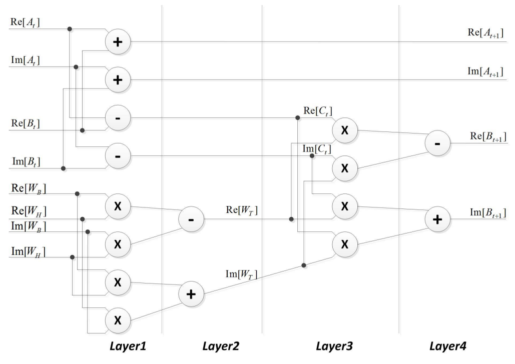
In the case of reduction by number of devices for a basic operation of the fast Fourier transform it is possible to suggest not less than 5 different variants called m-subgraphs. Each m-subgraph is characterized by its own data processing interval and hardware costs:

1. An m-subgraph μ_1 (minimal, Fig. 3d) contains not more than one device for each type of the operations of the subgraph. For our example, μ_1 contains 2 devices – a multiplier and an adder.
2. An m-subgraph μ_2 (multiple) represents a multiple reducing of the number of devices in the subgraph, and is similar to factoring out. Several variants of μ_2 are possible, such as 8 devices (4 multipliers, 4 adders, Fig. 3b), 4 devices (2 multipliers, 2 adders, Fig. 3c), and 2 devices (1 multiplier, 1 adder, Fig. 3d).
3. An m-subgraph μ_3 contains all devices from a layer with the maximum total number of operations. If it is necessary, the set of operations is complemented with devices to keep data equivalency. In this case, the layer with the maximum total number of operations is involved entirely, and it is executed during one clock cycle. For our example, μ_3 contains 8 devices from the first layer (4 multipliers, 4 adders, Fig. 3b).
4. An m-subgraph μ_4 is formed by a layer with the maximum number of operation types. If it is necessary, the layer is complemented with devices to keep data equivalency. For our example, μ_4 is similar to μ_3. It contains 8 devices from the first layer (4 multipliers, 4 adders, Fig. 3b).
5. An m-subgraph μ_5 (the improved minimal one) is the minimal μ_1 with one supplementary device that performs the most repeated operation of a basic subgraph. For some subgraphs, it provides approximately twofold decrease in the data processing interval for the reduced computing structure. For our example, μ_5 contains 3 devices (2 multipliers, 1 adder).

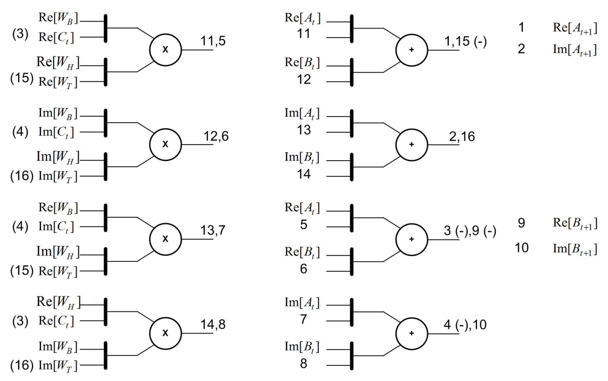
We formed the list of m-subgraphs on the base of tasks from such problem domains as digital signal processing, symbolic processing, linear algebra, and molecular docking. It is possible to add to the list some new strategies of m-subgraphs synthesis for tasks from other problem domains. However, the total number of possible strategies hardly ever exceeds 10, because the number of problem domains of RCS application is limited. Here, μ_1, μ_2 and μ_5 are the most interesting m-subgraphs. The m-subgraph μ_1 is the most common variant of basic subgraphs from various problem domains; μ_2 is the most acceptable for scaling of computing structures, but not always suitable due to the task structure; μ_5 is the most time-optimal, if hardware resource is sufficient for additional hardware-programmed device.

After the reduction by devices, in the third step of transformations, the reduction by data width with the coefficient R_0^ρ is performed for each synthesized m-subgraph. Here, the number of possible variants of reduction by data width for possible data types does not exceed 2:

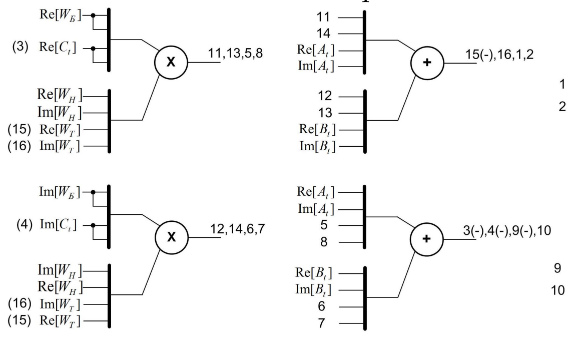
- For the reduction by width of logical and integer data (fixed-point data), the decrease in hardware costs is linearly proportional to the reduction coefficient. Therefore, the reduction is performed with the specified coefficient that does not exceed the width of processing data.



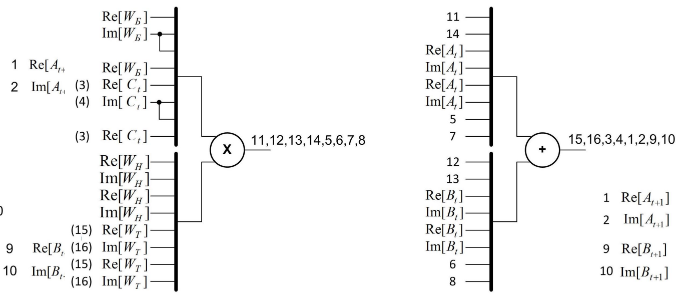
(a) The information graph of hardware-programmed fast Fourier transform basic operation



(b) The computational structure of m-subgraphs μ2-8 devices, μ3 and μ4



(c) The computational structure of m-subgraphs μ2-4 devices, μ3 and μ4



(d) The computational structure of m-subgraphs μ1 and μ2-2 devices

Figure 3. The information graph of hardware-programmed fast Fourier transform basic operation and m-subgraphs μ_1, μ_2, μ_3, μ_4, μ_5

- If floating-point data are reduced, then it is reasonable to perform 2-fold reduction by data width for 32-digit data, and 2- and 4-fold reduction by data width – for 64-digit data. It is caused by the exponential growth of the overhead expenses for processing of a mantissa and an order of magnitude for other reduction coefficients.

Thus, after reduction by data width, the number of m-subgraph variants is equal to $5 \cdot 2 = 10$. For each variant, it is necessary to analyze the required hardware resource, and the data processing interval, which defines the task solution time. Sometimes, when the hardware costs A_T of the reduced task structure exceed the available RCS hardware resource A_{RCS}, we perform the additional or fourth step of transformations. Such situation occurs due to additional switching costs, required for the reduction by number of computing devices and for the reduction by floating-point data width, because hardware costs are decreasing non-linearly. For the reduction by number of computing devices, we cannot always calculate the reduction coefficient R_0^{Op} before the transformations. Therefore, the coefficients R_0^{Op} and R_0^{ρ} may demand correction after the reduction.

After all reduction transformations, we evaluate the achieved reducing of hardware costs for task solution. Two variants are possible. We map the reduced computing structure on the available RCS hardware resource, or we additionally reduce hardware costs due to growth of expenses. In the first case, we perform the reduction transformations to map the information graph on the RCS architecture, and it takes 3 steps with analysis of 10 variants. In the second case, we return to the initial information graph (according to Theorem 2), and perform the performance reduction (steps 1–3) with the increased coefficient $R_1 = R_0 + 1$. Or, if it is possible, we perform multiple reductions by one parameter. Obviously, in the second case, the number of analysed variants is duplicated and equal to 20. Even if the task structure consists of several fragments, then the number of variants for justification performed by additional reduction transformations, and by methods of data processing, is few. Here, the additional reduction transformations consist in variation of the clock rate and data processing interval, and data processing can be parallel, pipelined, or can be represented as a macropipeline or a nested pipeline. Practically, the number of different fragments in the most part of tasks does not exceed 3–5; hence, the total number of variants of reduction transformations for such tasks hardly ever exceeds 60.

When a sequential program for a multiprocessor computer system with distributed memory is parallelized automatically, the compiler evenly distributes all calculations among the nodes without any splitting into several subtasks. It is necessary to analyse data distribution into nodes to avoid data nonlocality that may occur during automatic distribution of calculations disregarding dependencies (Read-After-Write, Write-After-Read, Write-After-Write, Read-After-Read) [21, 22] of the source program. So, the parallelizing compiler selects one parameter the parallelizing coefficient according to the number of used multiprocessor computer system nodes, the data spatial locality criterion, and the dependencies of the source program.

Reduction of performance and hardware costs of a RCS is performed with a reduction coefficient, which is the same for all subtasks. As a result, the reduced computing structure is balanced. For an RCS, it is possible to reduce the performance by such parameters as the number of devices, data width, and interval of processing data, unavailable for processor computer systems. For an RCS, in contrast to processor architectures, the overall reduction coefficient for each subtask is represented as a product of reduction coefficients (by number of basic subgraphs, devices, data width and interval). Owing to the fact, that we use a specific combination of reduction coefficients for each subtask, it is possible to take into account parameters of subtasks, choosing the most rational coefficients of reduction transformations, and to decrease the variety of reduced computing structures. Using such approach, we considerably decrease both the number of analyzed variants, and the time of information graph adaptation to the architecture and configuration of the given RCS.

5. Order of Reduction Transformations For Synthesis of Computing Structures

We created software tools for application development [24], based on our principles of automatic mapping of information graphs on RCS architectures, and on our performance reduction methods. With the help of the software, any sequential C-program is transformed into the absolutely-parallel information graph form. After that, the information dependencies among the task subgraphs are analyzed, and performance reduction of the subgraphs is performed for further adaptation to the RCS architecture, selected by the user. The methodology of all these transformations is the topic for another paper, and it transcends the scope of this work. Therefore, let us represent the basic rules, which we use for reduction of tasks, containing several subgraphs. To justify the speed of data processing in all subtasks of the information graph, and to select the most rational form of calculations for each subgraph, taking into account computing structures of other subgraphs and the whole task, we use the following order of reduction for computing structure synthesis:

1. Scaling and performance reduction of the information graph starts from the biggest subgraph. Here, "the biggest subgraph" means the subgraph with the highest hardware costs. The number of memory channels, and the data flow density of the biggest subgraph define all these parameters for all the rest subgraphs.
2. For basic subgraphs partition during analysis of its hardware resource, it is reasonable to compare it with the minimum resource, which is definitely implementable in one FPGA chip. In this case, there is no need to scale the subgraph with the help of the methods of reduction by number of devices, and by data width. If the given minimum resource is sufficient for the subgraph, then the subgraph is hardware-programmed without scaling.
3. The first transformation is decreasing of the number of memory channels. It is performed with the help of the reduction by number of subgraphs for data independent subgraphs. Then, according to the reduction coefficient, all reviewed reduction transformations are performed. Here, we take into account that the order and priority of reduction transformations for different types of tasks can be different.
4. For subgraphs with low weights, it is reasonable to perform hardware implementation. Here, a low weight is not more than a-priori specified value, for example, 5 % from the total hardware costs of the task. If it is necessary to reach the specified reduction coefficient, we use the reduction by data processing interval. Such subgraphs have no considerable influence on exceeding of task hardware resource. Besides, the reduction by number of devices, and by data width can both complicate hardware-programming, and increase hardware costs for switching structure, and, as a result, lead to additional steps of reduction transformations for all task subgraphs.
5. If reduction transformations are the same, but used with different coefficients and in different subtasks, it is necessary to synchronize data flows density (is performed automatically). As a rule, such synchronization leads to additional hardware costs, because hardware programming of synchronization blocks is based on multiplexers/demultiplexers, buffers, internal dual-port memory (BRAM).
6. When we perform the reduction by number of subgraphs, we keep at least one loop structure, because this is the way to decrease the task solution time. Besides, it does not increase the number of distributed memory channels, and it occupies hardware resource, which is available and rather large. If it is impossible, then we program a multipipeline structure. It inevitably contains a feedback, and larger data processing interval; hence, the task solution time grows. Reduction of the data processing interval in such computing structure is possible, if the structure is optimized, i.e. transformed into a nested pipeline or into a macropipeline. In this case, the multipipeline computing structure contains the number of layers equal to the latency of iterative rungs. Then, the computing structure can be reduced to one pipeline, and the feedback sequence is completed with registers. The number of registers is equal to the latency.
7. If the information graph layers have a data dependence, which is possible in the case of functionally irregular graph, then basic subgraphs are reduced to the sequential form.

We experimentally verified all these rules with the help of our compiler prototype and testing tasks of linear algebra, symbolic processing and digital signal processing, such as SLAE solution by the Gaussian method, SLAE solution by the Jacobi method, SLAE solution by lower-upper-decomposition, the basic operation of fast Fourier transform with coefficients calculation. For all these problems, the number of steps of reduction transformations, calculated according to the suggested methodology, does not exceed 16. The obtained values of reduction coefficients, numbers of transformation steps, and practical results for the scaled tasks, prove that the reduction transformation methods for automatic creation of parallel RCS applications, reviewed in the paper, are correct and efficient. The efficiency of solutions, created with the suggested methods, is not less than 50–75 % in comparison with optimal solutions, designed by circuit engineers.

Conclusion

The task information graph, used as the absolutely parallel form of a task for an RCS, provides the maximum performance with the maximum hardware costs. When a task is hardware-programmed on an RCS, the user transforms its information graph into a computing structure which provides lower performance and occupies smaller hardware resource. This transformation, decreasing the performance and hardware costs, is performed by reducing the number of subgraphs, computational devices, the processing data width, by increasing the data processing interval, and by reducing the rate. We use performance reduction not only for those tasks, that need more resource, than it is available, but also as a method of mapping (or adaptation) of an information graph to an RCS architecture. Owing to the performance reduction methods for RCS, it is possible to use reduction by number of devices, by data width and interval. This is unachievable for processor computer architectures.

Owing to the proved theorems on reduction transformations, we defined the main principles, and suggested the methodology of information graphs mapping on RCS architectures with the help of the performance reduction methods. Besides, we estimated the number of performed reduction transformations. Performance reduction does not change the total number of variants of a parallel application, but helps us to distribute these variants into several classes for further analysis. It is sufficient to analyze only one variant from each class, not the whole class. The obtained estimation of the number of analyzed variants of the computing structure, synthesized as a result of reduction of performance and hardware costs, is considerably less than the similar indicator for a multiprocessor computer system with distributed memory. We explain it by decomposition of the whole set of variants into topologically isomorphic groups of solutions, performed during reduction. Decrease of the number of analyzed variants to a single computing structure from each class considerably decreases the creation time of a parallel application, adapted to a RCS architecture (or configuration).

Further research will be directed at extension of classes from various problem domains, at mapping of information graphs on RCS architectures with the help of the reviewed methods of automatic reduction of performance and hardware costs.

References

1. Voevodin, V.V., Voevodin Vl.V.: Parallel computing. BHV-Petersburg (2002)

2. Palkowski, M., Bielecki, W.: TRACO Parallelizing Compiler. In: Wiliski, A., Fray, I., Peja, J. (eds.) Soft Computing in Computer and Information Science. Advances in Intelligent

Systems and Computing, vol. 342, pp. 409–421. Springer, Cham (2015), DOI: 10.1007/978-3-319-15147-2_34

3. SAPFOR system. https://www.keldysh.ru/dvm/SAPFOR/, accessed: 2020-05-22
4. Bielecki, W., Palkowski, M.: Perfectly Nested Loop Tiling Transformations Based on the Transitive Closure of the Program Dependence Graph. In: Wiliski, A., Fray, I., Peja, J. (eds) Soft Computing in Computer and Information Science. Advances in Intelligent Systems and Computing, vol. 342, pp. 309–320. Springer, Cham (2015), DOI: 10.1007/978-3-319-15147-2_26
5. Devan, P.S, Kamat, R.K.: A Review – LOOP Dependence Analysis for Parallelizing Compiler. International Journal of Computer Science and Information Technologies 5(3) (2014) https://www.ijcsit.com/docs/Volume%205/vol5issue03/ijcsit20140503305.pdf, accessed: 2020-05-22.
6. Jensen, N., Karlsson, S.: Improving Loop Dependence Analysis. ACM Transactions on Architecture and Code Optimization 14(3), 1–24 (2017), DOI: 10.1145/3095754
7. Solihin, Y.: Fundamentals of parallel computer architecture: multichip and multicore systems. Chapman and Hall/CRC (2016)
8. Cooper, K.D., Torczon, L.: Engineering a Compiler. Morgan Kaufmann (2005)
9. Kennedy, K., Allen, R.: Optimizing Compilers for Modern Architectures: A Dependence-based Approach. Morgan Kaufmann (2001)
10. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann (1997)
11. Levin, I., Dordopulo, A., Fedorov, A., Kalyaev, I.: Reconfigurable computer systems: from the first FPGAs towards liquid cooling systems. Supercomputing Frontiers and Innovations 3(1), 22–40 (2016), DOI: 10.14529/jsfi160102
12. Liu, S., Liu Z., Huang, H.: FPGA implementation of a fast pipeline architecture for JND computation. In: Proceedings of 5th International Congress on Image and Signal Processing, 16-18 Oct. 2012, Chongqing, China. pp. 577–581. IEEE (2012), DOI: 10.1109/CISP.2012.6469995
13. Trimberger, S.M.: Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. Proceedings of the IEEE 103(3), 318–331 (2015), DOI: 10.1109/JPROC.2015.2392104
14. Wahiba, M., Abdellah, S., Aichouche, B.: Implementation of parallel-pipeline H.265 CABAC decoder on FPGA. In: Proceedings of the First International Conference on Embedded & Distributed Systems, EDiS 2017, 17-18 Dec. 2017, Oran, Algeria. pp. 1–6. IEEE (2017), DOI: 10.1109/EDIS.2017.8284037
15. Khatami, R.I., Ahmadi, M.: High throughput multi pipeline packet classifier on FPGA. In: Proceedings of the 17th CSI International Symposium on Computer Architecture & Digital Systems, 30-31 Oct. 2013, Tehran, Iran. pp. 137–138. IEEE (2013), DOI: 10.1109/CADS.2013.6714253

16. Prihozhy, A., Bezati, E., Ab Rahman, A.A., Mattavelli, M.: Synthesis and Optimization of Pipelines for HW Implementations of Dataflow Programs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 34(10), 1613–1626 (2015), DOI: 10.1109/TCAD.2015.2427278

17. Korcyl, G., Korcyl, P.: Investigating the Dirac Operator Evaluation with FPGAs. Supercomputing Frontiers And Innovations 6(2), 56–63 (2019), DOI: 10.14529/jsfi190204

18. Qu, Y.R., Prasanna, V.K.: High-Performance and Dynamically Updatable Packet Classification Engine on FPGA. IEEE Transactions on Parallel and Distributed Systems 27(1), 197–209 (2016), DOI: 10.1109/TPDS.2015.2389239

19. Kalyaev, I.A., Levin, I.I., Semernikov, E.A., Shmoilov, V.I.: Reconfigurable multipipeline computing structures. Nova Science Publishers, New York, USA (2012)

20. Sorokin, D.A., Dordopulo, A.I., Levin, I.I., Melnikov, A.K.: Solving problems with essentially variable intensity of data flows on reconfigurable computing systems. Bulletin of computer and information technologies 2, 49–56 (2012), DOI: 10.14489/issn.1810-7206

21. Graham, R.L., Knuth, D.E., Patashnik, O.: Concrete Mathematics: A foundation for computer science (2nd ed.). Addison-Wesley Professional (1994)

22. Patterson, D., Hennessy, J.: Computer Architecture: A Quantitative Approach (5th ed.). Morgan Kaufmann (2011)

23. Unnikrishnan P., Shirako J., Barton K., Chatterjee S., Silvera R., Sarkar V.: A Practical Approach to DOACROSS Parallelization. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds) European Conference on Parallel Processing, 27-31 Aug. 2012, Rhodes Island, Greece. Euro-Par 2012 Parallel Processing, Lecture Notes in Computer Science, vol. 7484, pp. 219–231. Springer, Berlin, Heidelberg (2012), DOI: 10.1007/978-3-642-32820-6_23

24. Levin I., Dordopulo, A., Gudkov, V., Gulenok, A., Bovkun A., Yevstafiyev, G., Alekseev, K.: Software Development Tools for FPGA-Based Reconfigurable Systems Programming. In: Voevodin, Vl., Sobolev, S. (eds) Russian Supercomputing Days, 23-24 Sept., Moscow, Russia. Supercomputing, Communications in Computer and Information Science, vol. 1129, pp. 625–640. Springer, Cham (2019), DOI: 10.1007/978-3-030-36592-9_51

DOI: 10.14529/jsfi200202

Long Distance Geographically Distributed InfiniBand Based Computing

Karol Niedzielewski*[1]*, Marcin Semeniuk*[1]*, Jarosław Skomiał*[1]*,
***Jerzy Proficz*[2]*, Piotr Sumionka*[2]*, Bartosz Pliszka*[2]*, Marek Michalewicz*[1]**

Collaboration between multiple computing centres, referred as federated computing is becoming important pillar of High Performance Computing (HPC) and will be one of its key components in the future. To test technical possibilities of future collaboration using 100 Gb optic fiber link (Connection was 900 km in length with 9 ms RTT time) we prepared two scenarios of operation.

In the first one, Interdisciplinary Centre for Mathematical and Computational Modelling (ICM) in Warsaw and Centre of Informatics – Tricity Academic Supercomputer & networK (CI-TASK) in Gdańsk prepared a long distance geographically distributed computing cluster. System consisted of 14 nodes (10 nodes at ICM facility and 4 at TASK facility) connected using InfiniBand. Our tests demonstrate that it is possible to perform computationally intensive data analysis on systems of this class without substantial drop in performance for a certain type of workloads. Additionally, we show that it is feasible to use High Performance Parallex [1], high level abstraction libraries for distributed computing, to develop software for such geographically distributed computing resources and maintain desired efficiency.

In the second scenario, we prepared distributed simulation - postprocessing - visualization workflow using ADIOS2 [2] and two programming languages (C++ and python). In this test we prove capabilities of performing different parts of analysis in seperate sites.

Keywords: HPC, distributed computing and systems, InfiniBand, federated supercomputing, geographically distributed workflows, ADIOS, HPX, High Performance Parallex.

Introduction

Growth of computing capabilities in connection with big data manipulation and analysis gives us new tools for broadening knowledge and bringing new scientific breakthroughs. However new possibilities introduce new challenges. Great scale of stored information requires new approches to data storage and manipulation. Sometimes data movement between data centres requires a lot of time (measured in days or months) or is even impossible because of property rights (the data is owned by one entity and cannot be shared). Such cases occur more and more frequently and demand close cooperation between data centres. Collaboration between multiple computing centres is referred to as federated computing and will be one of the key components of High Performance Computing (HPC) in the future.

Between 2014–2016, A*STAR Computational Resource Centre (A*CRC) in Singapore engaged in exploration of long-range InfiniBand technology to build globally distributed concurrent computing system called InfiniCortex. These exploration led to integrating computing resources over four continents and six countries connected with RDMA enabling InfiniBand fabric [3–6].

The technology for long-haul global reach extended InfiniBand has been created by two companies: Obsidian Strategics, a Canadian company [7, 8], which apparently is not in operation anymore, and Bay Microsystems which was recently bought by Vcinity [9].

Mellanox Technologies built MetroX InfiniBand long-haul extenders [10], but they have limited range of about 40 km.

[1]Interdisciplinary Centre for Mathematical and Computational Modelling (ICM), University of Warsaw, Warsaw, Poland
[2]Centre of Informatics – Tricity Academic Supercomputer & networK (CI TASK), Gdańsk University of Technology, Gdańsk, Poland

The extended range InfiniBand has been initially used for remote storage, and for moving very large data (so-called "Large Data") between the sites. In 2007 it was reported that *"Obsidian Research Corporation's Longbow Campus products have enabled NASA to relocate 15 percent (1,536 processors) of its high-ranking SGI Altix-based Columbia Supercomputer to another facility and connect both locations without any performance degradation."* [11].

The early instances of long range InfiniBand connectivity were implemented at:

1. NASA: connecting Pleiades at NASA Ames Research Center and Hyperion elements at Lawrence Livermore National Laboratory with Obsidian Longbow extenders.
2. NASA: secure wire-speed storage synchronisation between NASA Ames Research Center in California and NASA Goddard Space Flight Center in Maryland.
3. Arizona State University testbed [12].
4. Swiss Supercomputer Center: *"We evaluated the Obsidian Longbow InfiniBand Range Extender with the overall goal to ensure continuous availability of GPFS through the complete CSCS relocation period by running one single GPFS file system over both sites. The geographical distance between the current and the future location is about 3 km, the measured distance of dark fiber is 10 km. The evaluation results for the range extender are encouraging and are in line with our expectations and requirements."* [13].
5. IT Centers at the Heidelberg to Mannheim Universities [14].

InfiniCortex built by A*CRC over three year period 2014–2016 was by far the largest and the most extensive, global scale InfiniBand distributed concurrent computing system ever built. A notable application created on the top of InfiniCortex was InfiniCloud – a globally distributed cloud infrastructure used to run cancer mutation calling pipeline over four continents [15, 16].

Recently an idea of Superfacilities was formulated within the US Department of Energy Labs. Superfacilities would encompass supercomputing resources with large scale data storage, large scale experimental facilities, mathematical methods, software and human expertise – and, of course, with all infrastructure elements connected with super-efficient network fabric [17–19].

In the words of Gregory Bell, with creation of Superfacilities *"Scientific progress will be completely unconstrained by the physical location of instruments, people, computational resources, or data."* [20].

It should be noted, that InfiniCortex created several years earlier was a precursor of a DoE defined Superfacility. European prototype, named Fenix Infrastructure is currently being built by five major supercomputing centres [21].

Based on the success and experiences of the global scale InfiniCortex infrastructure, Singapore implemented a country-wide STAR-N Singapore InfiniBand Fabric connecting Nanyang Technological University, Singapore National University and A*STAR into one 100 Gbps InfiniBand network. The fabric is based on the shorter range Mellanox MetroX extenders. It allows for easy access to ASPIRE Supercomputer based at the National Supercomputer Centre at the A*STAR central location through login nodes at remote locations, and efficient data transfer between the sites [22].

The main objectives of our project, reported here, were to:

- establish the first long-haul InfiniBand connection between two Polish HPC centres, which will serve as the first step towards federating all Polish HPC centres;
- test Vcinity 40 Gbps long-range InfiniBand technology over distance of 900 km;
- run High Performance ParalleX enabled application over long-haul distributed network;
- test ADIOS workflows over this distributed infrastructure.

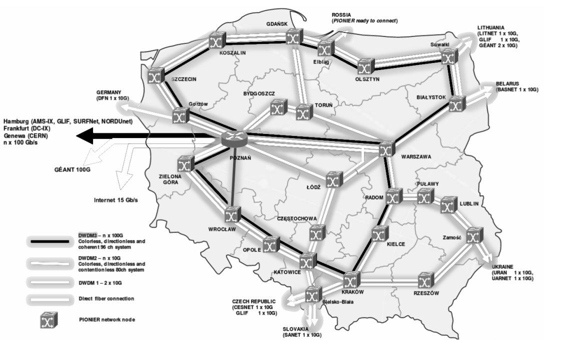
To test technical possibilities of future collaboration, ICM and TASK teams decided to test 40 Gb InfiniBand connection over optic fiber link in various scenarios.

The first step was to prepare a long distance geographically distributed computing cluster and to examine its data analysis capabilities. We demonstrate possibilities of using high level abstraction libraries for distributed computing (High Performance Parallex) to develop software for such clusters. What is more we show that some of the workflows (with low comunication requirements) can perform without drop in performance on such distributed clusters. More comprehensive tests involving MPI all-reduce algorithms on this distributed computing cluster were presented in a separate conference report [23]. The second test was focused on distributing different parts of data analysis workflow between seperate sites. Here we show that it is feasible to implement efficient distributed workflows using geographically distributed hardware configuration.

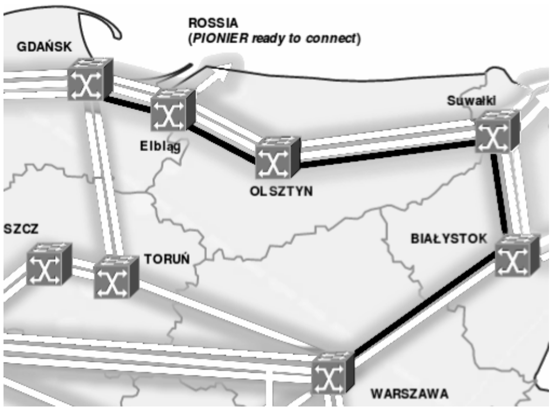
The paper is organized as follows. In Section 1 we describe our distributed computing cluster in two separate locations. It includes hardware, software and storage specification. In Section 2 we present the results of data analysis performed using geographically distributed computing cluster. Section 3 presents the capabilities of distributed simulation-postprocessing-visualization ADIOS workflow on the distributed infrastructure. The last section, Conclusions, contains a summary of the study and provides some hints to the future activities.

1. Testbed

For testing purposes ICM in collaboration with TASK, prepared a distributed computing cluster that consisted of the nodes located at ICM datacenter in Warsaw, and some nodes at TASK datacenter in Gdańsk. Facilities which are about 350 km apart were connected using Pionier academic network fiber optic link running in a round-about way over ∼900 km path (Fig. 1a and 1b).



(a) Map of Poland with Pionier network architecture and 100 gb links marked in black



(b) Zoom in on the map of Poland with Pionier network architecture. Link used in tests is marked in black

Figure 1. Pionier network architecture map (courtesy of Artur Binczewski, PSNC, Pozna)

1.1. Hardware

There were 10 compute nodes at ICM site. Each node was a dual socket HUAWEI RH-1288 v3 server with two Intel E5-2680 v3 CPUs, four 6 TB SATA drives and 128 GB DDR4 RAM. Each CPU has 12 cores and operate at 2.50 GHz clock frequency. At TASK facility there were four nodes. Each diskless node was HPE ProLiant XL230a Gen9 server with two Intel E5-2670 v3 processors and 128 GB DDR4 RAM. Each CPU has 12 cores operating at 2.3 GHz clock frequency.

InfiniBand interconnect in both clusters consisted of Mellanox SX6025 – InfiniBand switching system with 36 (FDR) 56 Gb/s ports and 4 Tb/s aggregate switching capacity. Each server was equipped with Mellanox FDR (56 Gb/s) Connect-X3 interface card used for the InfiniBand link and 1GE link for management. InfiniBand Extenders used in this tests were IBEX G40 – QDR InfiniBand RDMA based Extension Platform[3] and each was equipped with one QDR InfiniBand interface and one 40 GE port. IBEX G40 form factor is 1U rack unit and its power consumption is less than 140 W. Total buffer capacity allows extending InfiniBand connection up to 15,000 km.

The 100 GE circuit spanned between Warsaw and Gdańsk is routed via Balystok and was delivered by Pionier Polish National Research and Education Network in cooperation with Poznan Supercomputing and Networking Center. The ∼900 km long circuit introduces 9 ms RTT latency that is consistent with theoretical results calculated using (1):

$$ping = \frac{l}{V_{glass}} = \frac{1800\ km}{200\ \frac{km}{ms}} = 9\ ms, \tag{1}$$

where l – length of optic fiber connection, V_{glass} – velocity of light in glass.

Storage was located at ICM and shared using Network File System (NFS) technology, therefore nodes on TASK side had to download dataset before analysis.

1.2. Software

For testing purposes we decided to use Multidimensional Feature Selection algorithm implemented together with High Performance Parallex (HPX) [24]. We chose this application because of its very good parallel scaling on our Okeanos Cray XC40 supercomputer (each node equipped with 24 Intel Xeon E5-2690 v3 cpu cores). The scaling results are presented in Fig. 2. We can see that analysis of Madelon dataset exhibits almost perfect parallel scaling up to 64 nodes (1,536 cores), and then deteriorates due to the size of the problem being too small (starvation). Possible applications of Multidimensional Feature Selection exhaustive search include many domains of science such as genomics, economics, social sciences and others. Full details of this work can be found in [24].

For MPI connectivity openMPI v3.1.4 [25] was used. HPX was built using this openMPI library. MPI processes count was equal to the number of used computing cores (number of cores * number of nodes).

Tests on a distributed cluster were performed using Madelon dataset. Madelon [26] is a synthetic dataset with 2,000 objects and 500 variables that can be accessed from the UCI Machine Learning Repository [27] that was prepared in csv format. Data was located on ICM side, therefore nodes on TASK side had to download dataset before analysis. Jobs were invoked on ICM side, therefore latency of the excecution on TASK side was ∼5 ms caused by the connection latencies.

[3]The test InfiniBand Range Extenders were provided by Vcinity, Inc. and 2CRSI SA.

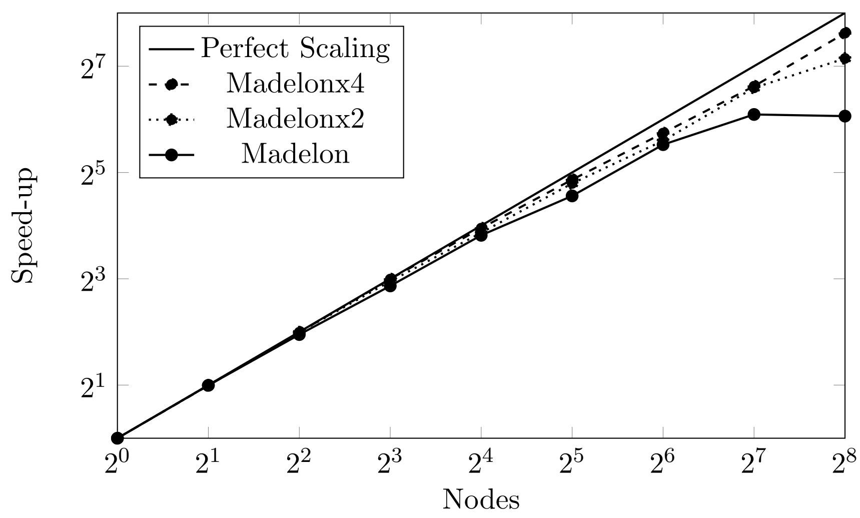


Figure 2. Measured speedup of 3-Dimensional analysis with 100 discretizations on different Madelon dataset sizes and with first algorithm implementation [24] (e.g. dataset that is twice as big has twice the number of variables. Additional variables are copies of variables from the original data set). Each node was equiped with 24 Intel Xeon E5-2690 v3 cores

2. Results

We tested the first implementation of MDFS [24] on groups of nodes of varying sizes and locations. Scenarios were prepared so that the amount of work was split evenly between sites (ICM and TASK) or was performed on nodes located only at one site.

We decided to perform 2-Dimensional analysis tests because it is the minimal size of the problem that fits well on up to 4 nodes. The measured time of the analysis performed on different configurations of nodes is presented in Fig. 3 and in Tab. 1. Speed-up of analysis is seen in Fig. 4 and the Tab. 2.

The results are presented using following coding (2):

$$G[Number\ of\ nodes\ on\ TASK\ side\ (Gdansk)]W[Number\ of\ nodes\ on\ ICM\ side\ (Warsaw)]$$
$$Example:\ G2W3\ -\ 2\ nodes\ on\ TASK\ sideand\ 3\ nodes\ on\ ICM\ side. \qquad (2)$$

Table 1. Measured time of Madelon dataset analysis at ICM-TASK

Nodes configutation	Number of nodes	Measured time [s]
G1W0	1	66.4
G0W1	1	60.9
G2W0	2	33.4
G1W1	2	33.5
G0W2	2	30.8
G2W2	4	16.9
G0W4	4	16.0

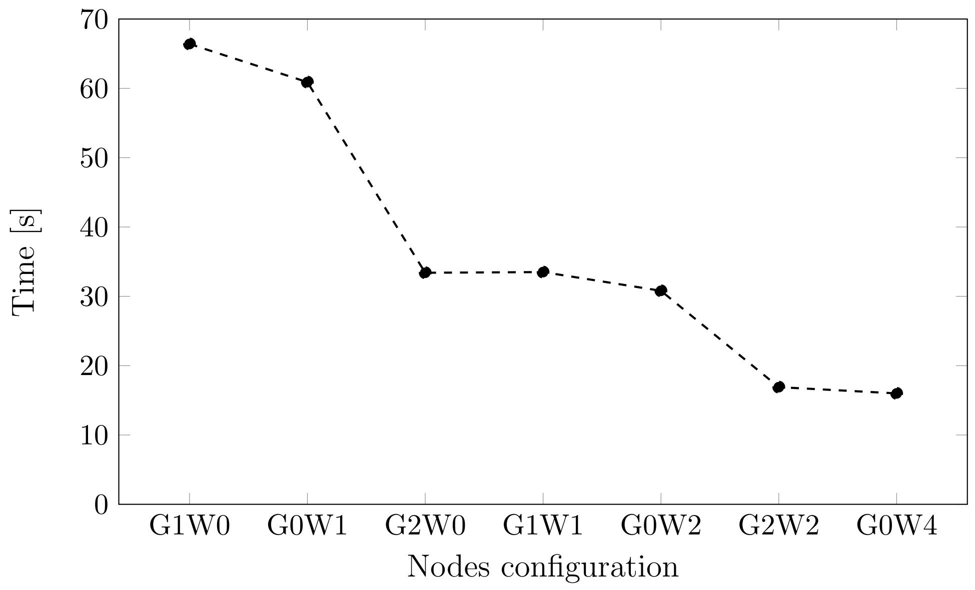


Figure 3. Measured time of madelon dataset analysis at ICM-TASK

Table 2. Measured speedup of madelon dataset analysis at ICM-TASK

Nodes configuration	Number of nodes	Measured speedup
G1W0	1	0.9
G0W1	1	1
G2W0	2	1.8
G1W1	2	1.8
G0W2	2	1.9
G2W2	4	3.6
G0W4	4	3.8

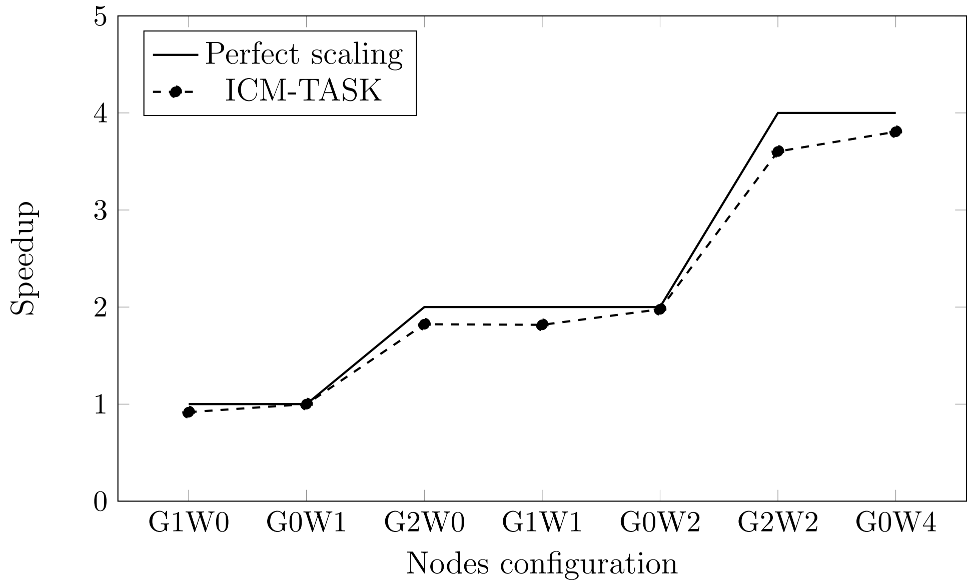


Figure 4. Measured speedup of madelon dataset analysis at ICM-TASK

It is clear that location of computations affects time of the analysis. Please see Tab. no. 1. Speedup changes are the outcome of analysis time changes (Tab. 2). Computations at ICM are faster (better performance) for the following reasons:

1. Jobs were invoked on the ICM side, therefore the excecution is delayed as well as receiving of the results is delayed. Globally the latency will be at least ∼9 ms because of the connection latencies.
2. Data was located on the ICM side, therefore nodes on the TASK side had to download dataset using NFS before analysis. Here again we observe minimum ∼9 ms latencies.
3. Nodes on the ICM side and the TASK side were equiped with different hardware (CPUs, RAM, Network card, etc.). This results in different computation times which are slower on the TASK side.

Nevertheless, location of the computations affect analysis time (performance) no more than 10 % and could be reduced by selection of optimal load balancing (less computations on TASK side). This brings us to conclusions that the differences of analysis time are not significant. Adventages (speedup) of computations on 'distributed' cluster overcome disadventages and can be beneficial in the future. We can observe linear scalability of MDFS method up to 4 nodes.

3. Simulation - Postprocessing - Visualization Distributed Workflow

We prepared simple simulation - postprocessing - visualization distributed workflow using the Gray-Scott MiniApp [28] and ADIOS 2 (version 2.4.0). ADIOS 2 (The Adaptable Input Output System version 2) is a framework dedicated for data I/O to write and read data when and where required. Its design introduces new approach to high level API that allows easy building of the data dependencies between components of applications. Important feature is possibility to build dependencies in distributed manner that makes ADIOS really interesting and powerful tool.

ADIOS2 remote IO between ICM and TASK was based on RDMA connection using SST files. This approach allows efficient reads of remote files and synchronous staging of sequences of simulation steps. Therefore none of the steps of the simulation data was omitted during postprocessing and visualization.

Distributed workflow is presented in Fig. 5 where we can see its several components written in C++ and python:

1. Gray-Scott (C++) – 3-D simulation of Gray-Scott reaction diffusion model [29] (using 4 mpi processes). Simulation and staging of data is run at the TASK site.
2. PDF Analysis (C++) – postprocessing of simulation data that prepares pdf images (using 1 mpi process). Run at the ICM site.
3. 2-D visualization (python) – 2-D cross section visualization of 3-D simulation (using 1 mpi process). Run at the ICM site. Example frame can be seen in Fig. 6a.
4. PDF ploting (python) – visualization of the plots from PDF Analysis (using 1 mpi process). Run at the ICM site. Example frame can be seen in Fig. 6b.

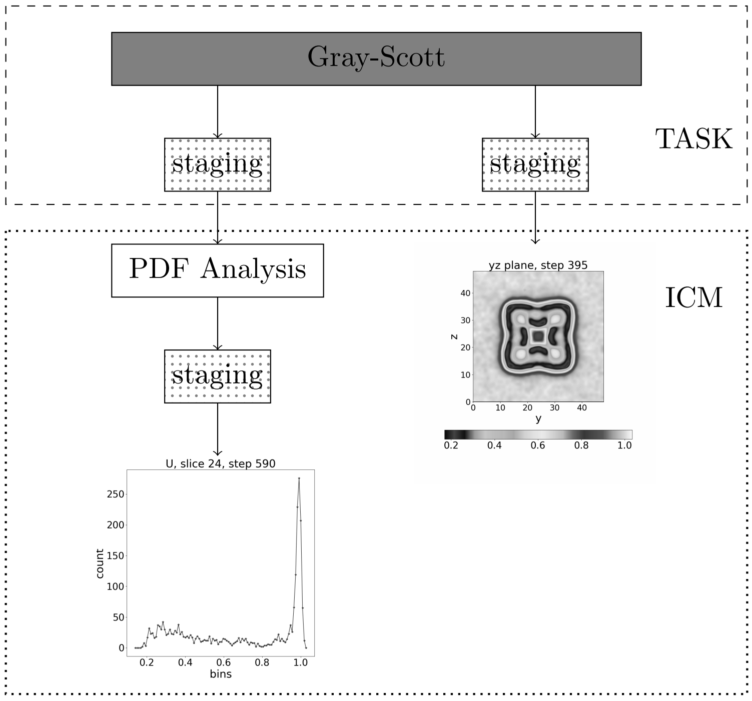
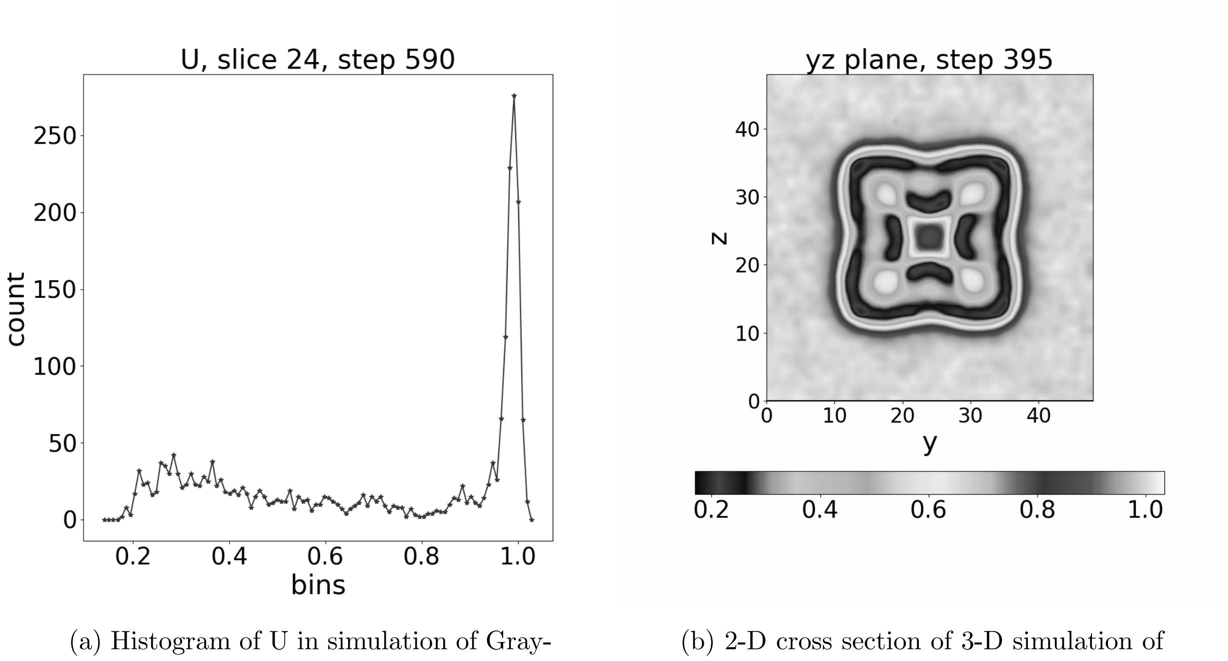


Figure 5. Workflow diagram



(a) Histogram of U in simulation of Gray-Scott reaction diffusion model

(b) 2-D cross section of 3-D simulation of Gray-Scott reaction diffusion model

Figure 6. Visualization examples

Conclusions

Our tests demonstrate that it is possible to perform computationally intensive data analysis on long distance geographically distributed computing cluster without substantial drop in performance. Additionally, we demonstrate that it is feasible to use high level abstraction libraries for distributed computing, such as High Performance Parallex, to develop software for geographically distributed clusters and to maintain computational performance comparable to a cluster in a single location. Moreover our application has potential to be used in many domains such as genomics, economics or social sciences. Our approach is not limited to feature selection methods and can be applied to many other data analysis and machine learning workflows that have low communication requirements.

In second test we present capabilities of using simulation - post-processing - visualization distributed workflow to execute parts of application in geographically separated sites. As a consequence, it opens new ways for sharing of data and distributing various components of applications.

Our successful tests of the connection between ICM and TASK present new technical possibilities and potential benefits of future collaboration between computing centres and federated computing in general.

Furthermore, presented solutions can be widely used and are not limited to the two centres listed above. We envisage a Polish InfiniCortex federating all six top Polish HPC centres into the Polish National (Distributed, Concurrent) Supercomputer utilising the Pionier fibre-optic fabric and six new generation InfiniBand range extenders offering 100 Gbps bandwidth and unlimited range.

Acknowledgements

This research was carried at the Interdisciplinary Centre for Mathematical and Computational Modelling (ICM), University of Warsaw, and at the Centre of Informatics – Tricity Academic Supercomputer & networK (CI TASK) at Gdańsk University of Technology. We acknowledge support of Vcinity® Inc. and 2CRSI SA who provided us with the InfiniBand Range Extenders. This project would not be possible without infrastructure and support of personnel of Pionier, a Polish National Research and Education Network (NREN) – providing 100 G connectivity to all academical HPC centers in Poland.

References

1. Kaiser, H., Lelbach aka wash, B.A., Heller, T., Bergé, A., et al.: STEllAR-GROUP/hpx: HPX V1.3.0: The C++ Standards Library for Parallelism and Concurrency (2019), DOI: 10.5281/zenodo.3189323

2. The Adaptable Input Output System version 2, `https://github.com/ornladios/ADIOS2/`, accessed: 2020-02-08

3. Orłowski, Ł., Deng, Y., Michalewicz, M.: Galaxies of supercomputers and their underlying interconnect topologies hierarchies. In: International Supercomputer Conference, Leipzig, Germany (2014), DOI: 10.13140/2.1.4798.2728

4. Michalewicz, M., Southwell, D., Tan, T., Poppe, Y., et al.: InfiniCortex: concurrent supercomputing across the globe utilising trans-continental InfiniBand and Galaxy of Supercomputers. In: Supercomputing 2014: The International Conference for High Perfor-

mance Computing, Networking, Storage and Analysis, At New Orleans, LA, USA (2014), DOI: 10.13140/2.1.3267.7444

5. Michalewicz, M.T., Lian, T.G., Seng, L., Low, J., et al.: InfiniCortex: Present and Future Invited Paper. In: Proceedings of the ACM International Conference on Computing Frontiers, May 2016, Como, Italy. pp. 267–273. Association for Computing Machinery, New York, NY, USA (2016), DOI: 10.1145/2903150.2912887

6. Noaje, G., Davis, A., Low, J., Lim, S., et al.: InfiniCortex-From Proof-of-concept to Production. Supercomputing Frontiers and Innovations 4(2), 87–102 (2017), DOI: 10.14529/jsfi170207

7. Obsidian Strategics Inc., `https://www.cybersecurityintelligence.com/obsidian-strategics-106.html`, accessed: 2020-06-01

8. Obsidian Strategics Inc., `https://obsidianstrategics.com/index.html`, accessed: 2020-06-01

9. Vcinity Inc., `https://vcinity.io/`, accessed: 2020-06-01

10. Mellanox MetroX®-2 Systems, `https://www.mellanox.com/products/long-haul`, accessed: 2020-06-01

11. Obsidian Longbow Campus Solutions Extend Its Columbia Supercomputer across Multiple NASA Locations, `https://www.militaryaerospace.com/home/article/16725502/obsidian-longbow-campus-solutions-extend-its-columbia-supercomputer-across-multiple-nasa-locations`, accessed: 2020-06-01

12. Eikenberry, S., Lindekugel, K., Stanzione, D.: Long Haul InfiniBand Technology: Implications for Cluster Computing, Arizona State University (2006), `https://obsidianstrategics.com/archives/2006/asu_stanzione_ccs.pdf`, accessed: 2020-06-28

13. El-Harake, H.N., Gamboni, C., Gorini, S., Schoenemeyer, T.: Evaluation of infiniband range extension offered by obsidian (2011)

14. Richling, S., Kredel, H., Hau, S., Kruse, H.G.: A long-distance infiniband interconnection between two clusters in production use. In: State of the Practice Reports, November 2011, Seattle, Washington. Association for Computing Machinery, New York, NY, USA (2011), DOI: 10.1145/2063348.2063368

15. Ban, K., Chrzeszczyk, J., Howard, A., Li, D., Tan, T.W.: InfiniCloud: Leveraging the Global InfiniCortex Fabric and OpenStack Cloud for Borderless High Performance Computing of Genomic Data. Supercomputing Frontiers and Innovations 2(3), 14–27 (2015), DOI: 10.14529/jsfi150302

16. Chrzeszczyk, J., Howard, A., Chrzeszczyk, A., Swift, B., Davis, P., Low, J., Tan, T.W., Ban, K.: InfiniCloud 2.0: distributing High Performance Computing across continents. Supercomputing Frontiers and Innovations 3(2), 54–71 (2016), DOI: 10.14529/jsfi160204

17. Antypas, K.: Superfacility: How new workflows in the DOE Office of Science are influencing storage system requirements? (2016), `https://storageconference.us/2016/Slides/KatieAntypas.pdf`, accessed: 2020-06-01

18. NERSC Superfacility, `https://www.nersc.gov/research-and-development/superfacility/`, accessed: 2020-06-01

19. Creating Super-facilities: a Coupled Facility Model for Data-Intensive Science, Internet 2 Global Summit 2015, `http://meetings.internet2.edu/2015-global-summit/detail/10003679/`, accessed: 2020-06-01

20. Bell, G.: The Energy Sciences Network: Overview, Update, Impact (DoE) - presentation, `https://science.osti.gov/-/media/ascr/ascac/pdf/meetings/20150324/Bell_ESNet.pdf?la=en&hash=46C0168F7ADAB232EC32E4452C49A159453859C9`, accessed: 2020-06-01

21. Fenix Research Infrastructure, `https://fenix-ri.eu/about-fenix`, accessed: 2020-06-01

22. Noaje, G.: InfiniCortex, InfiniBand nation-wide and world-wide, a talk given at Journee Scientifique ROMEO'2016, Reims, France (2016), `https://romeo.univ-reims.fr/news/208/Journee_Scientifique_ROMEO_2016_le_9_juin_2016_a_REIMS`, accessed: 2020-06-01

23. Proficz, J., Sumionka, P., Skomiał, J., Semeniuk, M., Niedzielewski, K., Walczak, M.: Investigation into MPI All-Reduce Performance in a Distributed Cluster with Consideration of Imbalanced Process Arrival Patterns. In: International Conference on Advanced Information Networking and Applications, 15-17 April, Caserta, Italy. pp. 817–829. Springer (2020), DOI: 10.1007/978-3-030-44041-1_72

24. Niedzielewski, K., Marchwiany, M.E., Piliszek, R., Michalewicz, M., Rudnicki, W.: Multidimensional feature selection and high performance parallex. SN Computer Science 1(1), 40 (2020), DOI: 10.1007/s42979-019-0037-5

25. Open MPI: Open source high performance computing, `https://www.open-mpi.org/`, accessed: 2020-02-08

26. Guyon, I., Gunn, S., Ben-Hur, A., Dror, G.: Result analysis of the nips 2003 feature selection challenge. In: Saul, L.K., Weiss, Y., Bottou, L. (eds.) Advances in Neural Information Processing Systems 17, pp. 545–552. MIT Press (2005), `http://papers.nips.cc/paper/2728-result-analysis-of-the-nips-2003-feature-selection-challenge.pdf`

27. Dua, D., Graff, C.: UCI machine learning repository (2017), `http://archive.ics.uci.edu/ml`

28. Application examples for the ADIOS2 I/O library, `https://github.com/ornladios/ADIOS2-Examples`, accessed: 2020-02-08

29. Pearson, J.E.: Complex Patterns in a Simple System. Science 261(5118), 189–192 (1993), DOI: 10.1126/science.261.5118.189

DOI: 10.14529/jsfi200203

Potential of I/O Aware Workflows in Climate and Weather

Julian M. Kunkel[1], *Luciana R. Pedro*[1]

The efficient, convenient, and robust execution of data-driven workflows and enhanced data management are essential for productivity in scientific computing. In HPC, the concerns of storage and computing are traditionally separated and optimised independently from each other and the needs of the end-to-end user. However, in complex workflows, this is becoming problematic. These problems are particularly acute in climate and weather workflows, which as well as becoming increasingly complex and exploiting deep storage hierarchies, can involve multiple data centres.

The key contributions of this paper are: 1) A sketch of a vision for an integrated data-driven approach, with a discussion of the associated challenges and implications, and 2) An architecture and roadmap consistent with this vision that would allow a seamless integration into current climate and weather workflows as it utilises versions of existing tools (ESDM, Cylc, XIOS, and DDN's IME).

The vision proposed here is built on the belief that workflows composed of data, computing, and communication-intensive tasks should drive interfaces and hardware configurations to better support the programming models. When delivered, this work will increase the opportunity for smarter scheduling of computing by considering storage in heterogeneous storage systems. We illustrate the performance-impact on an example workload using a model built on measured performance data using ESDM at DKRZ.

Keywords: workflow, heterogeneous storage, data-driven, climate/weather.

Introduction

High-Performance Computing (HPC) harnesses the fastest available hardware components to enable the execution of tightly coupled applications from science and industry. Typical use-cases include numerical simulation of physical systems and analysis of large-scale observational data. In the domain of climate and weather, there is a considerable demand for the orchestration of ensembles of simulation models and the generation of data products. A service such as the operational weather forecast workflow in Met Office writes around 200 TB and reads around 600 TB every day. In total, at the Met Office, on average 1.5 PB and 14 PB are written and read per day, respectively, for all climate and weather forecasts across all HPC clusters.

Based on the needs of climate and weather researchers, the HPC community has developed a software ecosystem that supports scientists to execute their large-scale workflows. While the current advances correspond to a big leap forward, many processes still require experts. For example, porting a workflow from one system to another requires adjusting runtime parameters of applications and deciding on how data is managed.

Since performance is of crucial importance to large-scale workflows, careful attention must be paid to exploit the system characteristics of the target computing centre. For instance, a data-driven workflow may benefit from the explicit and simultaneous use of a locally heterogeneous set of computing and storage technologies. This aspect means that substantial changes may be required to a workflow to tailor it to a particular supercomputer environment in order to obtain the best performance.

Knowing the capabilities, interfaces, and performance characteristics of individual components are mandatory to make the best use of them. As the complexity of systems expands and alternative storage and computing technologies provide unique characteristics, it becomes in-

[1]University of Reading, Reading, United Kingdom

creasingly difficult, even for experts, to manually optimise the usage of resources in workflows. In many cases, modifications are not performed because: 1) They are labour intense: any change to the workflow requires careful validation which may not pay off for small scale runs; 2) It is a one-time explorative workflow and; 3) Users are not aware of the potential of the complex system.

In this paper, we illustrate how knowing the Input/Output (I/O) characteristics of workflow tasks and overall experimental design helps to optimise the execution of climate and weather workflows. Exploiting this information automatically may increase the performance, throughput and cost-efficiency of the systems, providing an incentive to users and data-centres that cannot be neglected any longer. Our approach intends to reduce the burden on researchers and, at the same time, optimise the decisions about jobs running on HPC systems.

This paper is structured as follows. First, we describe the software stack involved in executing workflows in climate and weather in Section 1. Related work in heterogeneous storage environments and solutions for workflow processing is presented in Section 2. Next, the vision for including knowledge about data requirements and characteristics is sketched in Section 3 outlining the potential benefit the automatic exploitation might bring. Our design, based on existing components in climate and weather, is described in Section 4. An example use case demonstrates the impact on running a workload at the Mistral supercomputer in Section 5. The paper is concluded in Section 5.

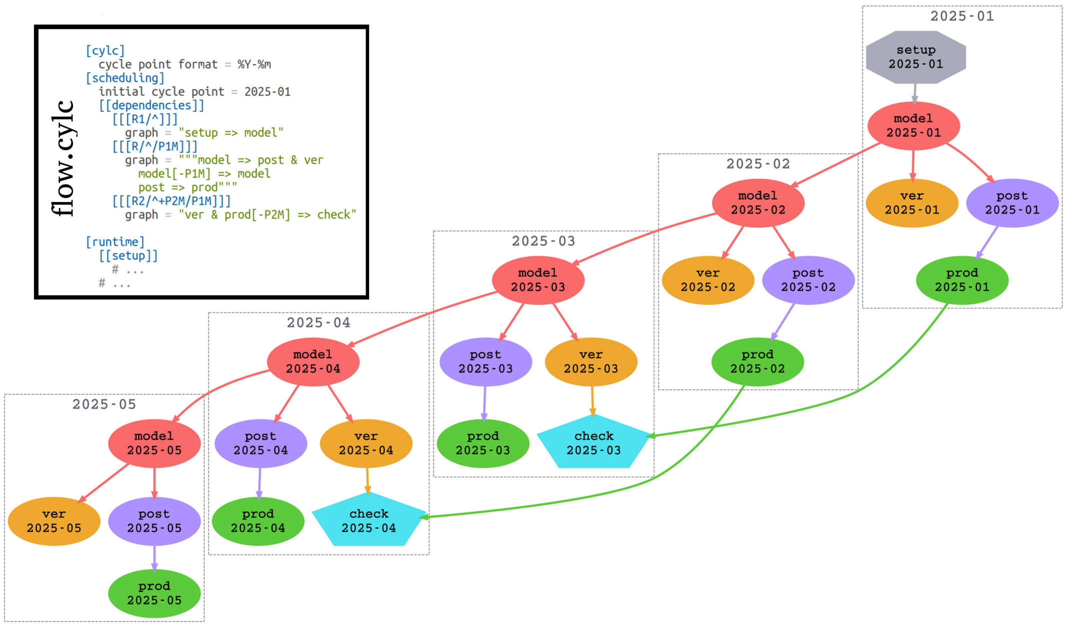
1. Workflows in Climate/Weather

In this section, we describe how workflows are executed in a conventional software stack and the typical hardware and software environment involved in running a climate and weather application.

1.1. Cylc

Cylc [19] is a general-purpose workflow engine in charge of executing and monitoring cyclic workflows in which each step is submitted to the batch scheduler of a data centre. With Cylc, tasks from multiple cycles may be able to run concurrently without violating dependencies and preventing the issue of delays that cause one cycle to run into another. Cylc was written in Python and built around a new scheduling algorithm that can manage infinite workflows of cycling tasks without a sequential cycle loop. At any point during workflow execution, only the dependence between the individual tasks matters, regardless of their particular cycle points. The information Cylc uses to control a given workflow is the task dependency. In a script file, the developers define, for each task, the parallelism settings and where data is to be stored.

Consider the Cylc workflow for a toy monthly cycling workflow in Fig. 1. In this workflow, an atmospheric model (labelled as `model` in the figure) simulates the physics from a current state to predict the future, for example, a month later. In climate research, this process is repeated in the model to simulate years into the future. Once the simulation of any month is computed, data for this month becomes available and can now be analysed. In this workflow, the task `model` is followed by tasks postprocessing (`post`), forecast verification (`ver`), and product generation (`prod`), all specified as a workflow in a Cylc configuration file (`flow.cylc`).



flow.cylc

```
[cylc]
  cycle point format = %Y-%m
[scheduling]
  initial cycle point = 2025-01
  [[dependencies]]
    [[[R1/^]]]
      graph = "setup => model"
    [[[R/^/P1M]]]
      graph = """model => post & ver
        model[-P1M] => model
        post => prod"""
    [[[R2/^+P2M/P1M]]]
      graph = "ver & prod[-P2M] => check"

[runtime]
  [[setup]]
    # ...
  # ...
```

Figure 1. Example of a Cylc workflow with its configuration file [19]

1.2. Workflow Execution

While Cylc is directing the execution of workflows, several components are presented in the implementation. The software stack involved in a general workflow is depicted in Fig. 2. Next, each stage of the execution is further described.

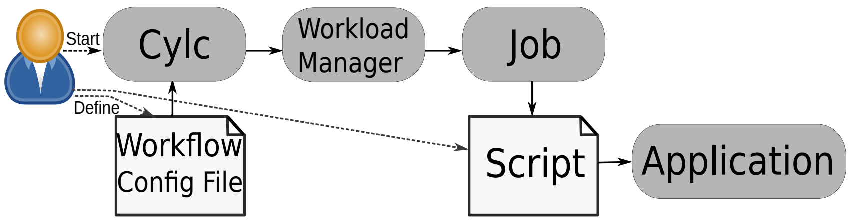


Figure 2. Software stack and stages of execution

1. **Scientist** specifies the workflow and provides a command or a script for each task. As part of the Cylc configuration, the command(s) to be run, any environment variables used by these application(s), and any workload manager directives. After that, the user enacts Cylc to start the workflow.
2. **Cylc** parses the workflow configuration file (`flow.cylc`), generates tasks dependencies, defines a schedule for the execution, and monitors the progress of the workflow. Once a task can be executed (dependencies are fulfilled), the workflow engine submits a *job script* for the workload manager with the required metadata that will run the Cylc task script.
3. **Workload Manager** such as Slurm [10] is responsible for allocating compute resources to a batch job and performing the job scheduling. The selected tool queues the job that represents the Cylc task and plans its execution, considering the scheduling policy of the data centre. Once the job is scheduled to be dispatched, i.e., resources are available, and the job priority is the highest, it is started on the supercomputer.
4. **Job** provides the environment with the resources and runs the user-provided program or script on one of the nodes allocated for it. Local variables containing information about the

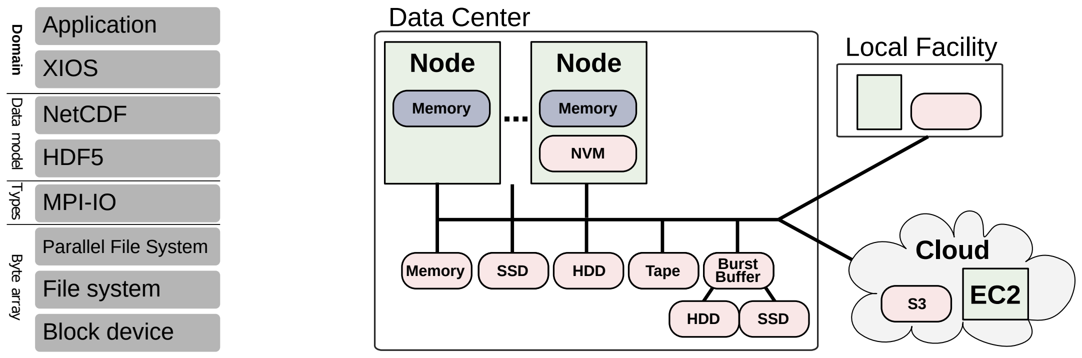
environment of the batch job, e.g., the compute nodes allocated, enact the Cylc provided script on the node.

5. **Script** starts the commands sequentially (a command can be a parallel application). During the creation of the script, Cylc has included variables that describe the task in the workflow. The information is typically fed into the application(s) representing the task and defining the storage location. The script uses commands to generate filenames considering the cycle and may store data in a workflow-specific shared directory. Either these commands are set in the Cylc workflow and then injected as environment variables or directly utilised as a part of the user-provided script.
6. **Application** is executed taking the filenames set by the script.

1.3. I/O Stack of a Parallel Application

Climate applications may have complex I/O stacks, as can be seen in Fig. 3a. In this case, we assume the application uses XIOS [17], which is providing domain-specific semantics to climate and weather. It may gather data from individual fields distributed across the machine (exploiting MPI for parallelism) and then uses NetCDF [4] to store data as a file. Under the hood, NetCDF uses the HDF5 API with its file format. Internally, HDF5 uses MPI and its data types to specify the nature of data stored. Finally, data is stored on a parallel file system like Lustre which, on the server-side, stores data in a local file system on block devices such as SSDs and HDDs.

Different applications involved in a workflow may use different I/O stacks to store their outputs. Naturally, the application which uses previously generated data as its inputs must use a compatible API to read the specific data format. In Fig. 3a, for example, XIOS may perform parallel I/O via the NetCDF API, allowing subsequent processes to read data directly using NetCDF. Within the ESiWACE project[2], we are developing the Earth System Data Middleware (ESDM) [14] to allow applications with this kind of software stack to exploit heterogeneous storage resources in data centres. The goal of ESDM is to provide parallel I/O for parallel applications with advanced features to optimise subsequent read accesses. Implemented as a standalone API, it also provides NetCDF integration allowing its usage in existing applications. Hence, in Fig. 3a, the HDF5 layer can be replaced with ESDM.



(a) I/O path for an MPI application (b) Example of an heterogeneous HPC landscape

Figure 3. Typical hardware and software environment for applications

[2]`https://www.esiwace.eu/`

1.4. Data Centre Infrastructure

At present-day, data centres provide an infrastructure consisting of computing and storage devices with different characteristics, making them more efficient for specific tasks and satisfying the needs of different workflows. Take, for example, the supercomputer Mistral at DKRZ, that consists of 3,321 nodes[3] and offers two types of compute nodes equipped with different CPUs and GPU nodes. Each node has an SSD for local storage, and DKRZ has additionally two shared Lustre file systems with different performance characteristics. Individual users and projects are mapped to one file system explicitly, and users can access it with `work` or `scratch` semantics. While data is kept on the `work` file system indefinitely, available space is limited by a quota. The `scratch` file system allows storing additional data, but data is automatically purged after some time.

Future centres are expected to have even more heterogeneity. A variety of accelerators (GPU, TPU, FPGAs), active storage, in-memory, and in-network computing technologies will provide further storage and processing capabilities. Fig. 3b shows such a system with a focus on computation and storage. Some of these technologies might be locally (specific compute nodes) or globally available. Depending on the need, the storage characteristics range from predictable low-latency (in-memory storage, NVMe) to online storage (SSD, HDD), and also cheap storage for long-term archival (tape). The tasks within any given workflow could benefit from utilising different combinations of storage and computing infrastructure.

1.5. Data Management

Usually, the scripts representing tasks define how data is placed on the available storage system. What happens in many current workflows is that they ignore the benefits of using multiple file systems concurrently and data locality between tasks to colocating them. On top of that, in the current state-of-the-art scientists optimise the available storage resources intuitively and compile the information about this decision-making process manually.

If a user knows the workflow and the system characteristics, data placement decisions can be optimised. Consider, for instance, the situation where each computing node has access to three file systems: a fast `scratch` file system on which data may reside only for a week, a slower `work` file system, and a local file system. Most current workflows utilise `work` and `scratch` systems. When a task is set to run, the corresponding dataset would be moved from `work` to `scratch`, processed, and the resulting dataset would be transferred back to `work`. If the `scratch` file system reaches its capacity, the dataset would be moved back to `work`, and the task would continue running until it is finished, which might be inefficient. In this situation, there are many straightforward opportunities to utilise data migration to optimise performance, and also other criteria (e.g., costs). However, with a multitude of file systems that differ at each data centre, such optimisations would be difficult to achieve manually by users. Policy-driven systems and burst buffers perform such optimisations automatically to some extent. However, as they lack information about the workflow, they cannot optimise workflows altogether.

[3]`https://www.dkrz.de/up/systems/mistral`

2. State-of-the-Art

Related work to the proposed approach can be categorised into: 1) Technology that exploits heterogeneous storage environments and supports user-directed policies and 2) Solutions for workflow processing.

Technology. Manual tiering requires the user or application to control data placement, i.e., storing data typically in the form of files on a particular storage system and, usually, moving data between storage by scripts. One limitation of such an approach is that decisions about how data are mapped and packaged into files are made by the producing application, and cannot be changed without manual intervention by a downstream application.

Burst buffer solutions provide a tiered storage system that aims to exploit a storage hierarchy. They can be integrated into hardware capabilities such as DDN's Infinite Memory Engine (IME) [2] or simple software solutions. A policy system, e.g., deployed on a burst buffer [22], aims to simplify data movement for the user, but typically migrates objects in the coarse granularity of files. File systems and data management software such as IBM Spectrum Scale [23], HPSS [26], BeeGFS [5], and Lustre [3] (e.g., using the progressive file layouts feature) provide hierarchical storage management allowing to store data on different storage technology according to administrator-provided policies. However, the semantic information that can be used by this type of system to make decisions is limited, e.g., data location, file extension, file age, etc.

The storage community had also adjusted various higher-level software to support storage tiering on top of several storage systems. For instance, ADIOS provides in-memory staging that had been exploited by a variety of applications [24]. Hermes [12] introduces a multi-tiered I/O buffering system with pre-fetcher that provides several data placement policies. iRODS [21] is a rule-oriented data system that allows scientists to organise data into shareable collections and provides several patterns for workflows considering data locality and data migration/replication. Finally, there have also been extensions to batch schedulers to perform data staging for utilising node-local storage, for example, NORNS [18] as an extension to Slurm.

Workflows. A good overview of the flavours of Scientific Workflow Management Systems (SWfMS) and their application to data-intensive workflows is given in [15]. The article states that SWfMS should enable the parallel execution of data-intensive scientific workflows and exploit vast amounts of distributed resources. Existing solutions recognise challenges in data variety (formats of the input data), opportunities to optimise the schedule by moving code to data, specification of the data dependencies for tasks, and they even may consider the capacity of the available data storage. The execution engine Dryad [9], for example, allows transferring data between tasks via files or directly using TCP connections and attempts to schedule tasks on the same nodes or racks. In [16], an approach was presented to monitor and analyse I/O behaviour of HPC workflows. Swift/T [27], a scripting language for describing dataflow processing enabling the execution of ensembles of applications, is now openly used as a prototype platform [20]. Recent improvements aim to migrate data to a local cache allowing to exploit locality. For instance, in [6], information about locality is proposed to be stored in extended attributes.

Several early research in grid workflows and, lately, cloud workflows, use cases of interest to maximise data locality. Economic factors (including storage costs) for workflow execution are discussed in [1]. In [7], the authors discuss the role of Machine Learning (ML) for workflow execution and elaborate a general potential for resource provisionings such as optimisation of runtime parameters, data movements, and hierarchical storage. In [25], an ML model that stages data for in-situ analysis by exploiting the access patterns is introduced.

Workflow systems can also be specifically utilised to reproduce scientific results, i.e., recompute the results. Those scalable workflow solutions typically utilise a container solution to allow execution in an arbitrary software environment. Popper [11], Snakemake [13], and Nextflow [8] provide a language to specify workflows and to execute them. Snakemake is interesting as it supports definition and inference of input and output filenames.

While various aspects of our vision have been addressed individually by related work for different domains, the high level of abstraction that we aim for and the potential it unleashes goes beyond the capabilities of existing approaches.

3. Vision for I/O-Aware Workflows

Nowadays, in order to run a job in an HPC environment efficiently, researchers have to develop profound knowledge, not only about their workflow, which is expected, but also about decisions regarding storage, communication, computing, and considerations regarding cost-efficiency of those operations. However, applied scientists should not spend much time understanding hardware characteristics and operational knowledge of running a data centre, but using their expertise to develop their work and just collect and analyse the results of their experiments.

We aim for achieving an automatic and dynamic mapping of I/O resources to workflows. Once we have an automated decision about where the job will run and how the storage will be managed, scientists can then reuse their workflow specification on any system without further modification and even without previous knowledge about the system architecture.

There are several approaches to implement the technology for the vision proposed in this work, and changes are needed in the software components to realise it. In Section 4, we will discuss a specific design for our transitional roadmap considering climate and weather workflows and tools scientists from this field already use in their routine research.

Our vision for I/O-aware workflows requires two additional pieces of information. Firstly, the user must augment the workflow description with information about I/O specifications and explicitly annotate dependencies to datasets. Secondly, details about the storage architecture must be available.

3.1. System Information

While many optimisations are possible once an abstraction is in place, the improvements we discuss here are related to the life cycle and placement of datasets into specific storage according to system performance characteristics and workflow specification. To achieve that, the system information shall comprise of all available storage systems, the system topology, and details of each available component. Simplified and complex models of the components can be included to approximate expected performance for specific I/O patterns. It is expected that the data centre (or expert user) can create such a configuration file, e.g., by using vendor-provided information or by executing benchmarks. With this information, a scheduler can make the initial data placement, transformation, and migration decisions for individual datasets during their life cycle. This separation of concerns allows us to abstract from the workflow what is essential and what a system should optimise to ensure smart usage of the available resources.

3.2. Extended Workflow Description

In general, climate and weather workflows allow specifying tasks and dependencies among them. We aim to enhance the current information with characteristics for input and output, i.e., the datasets. An example workflow with `N` cycles containing input datasets and (intermediate) products is illustrated in Fig. 4. Round nodes represent tasks, squared nodes represent data, and arrows indicate dependencies. In the example, Task 1 needs two datasets to perform its work, it produces Product 1, and directly communicates with Task 2. For each new cycle, the `checkpoint` from the previous cycle (Product 1) is used as input to starting the next cycle. Most of the workflow can run automatically, except for the manual quality control of the products and the final data usage of Product 3. This last step represents the typical uncertainty of data reuse, i.e., it is unclear how Product 3 will be used further. In the approach proposed in this work, each task is annotated with the required input datasets and the generated products must include metadata such as data life cycle, the value of data, and how long it should be kept. The idea here is to embrace the concept that tasks dependencies are really imposed by datasets dependencies.

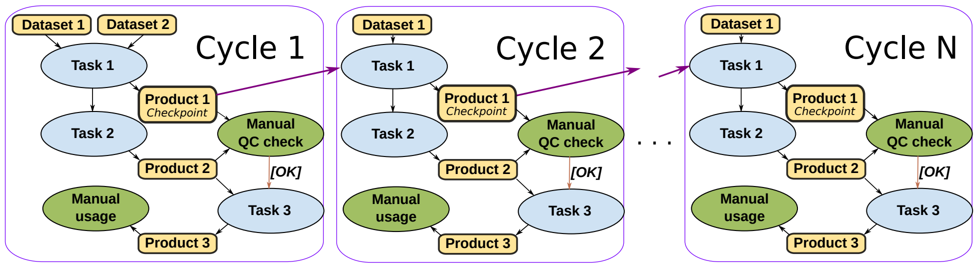


Figure 4. Example of a high-level workflow with tasks and data dependencies

3.3. Smarter I/O Scheduling

The abstraction and automation of the I/O inside a workflow allow a runtime system to improve data placement and apply data reduction on heterogeneous storage systems. Taking into consideration the architecture and workflow information, a smarter schedule can now be realised by exploiting the additional information. Value and priority can influence fault-tolerance strategies and imply the quality of service for performance and availability. Aspects like data reproducibility (can it be recomputed easily), type of the experiment (test, production), and runtime constraints for the overall and potential workflow could allow reducing costs and, hence, increase scientific output. Next, we outline two core strategies and the potential the proposed vision can bring to the improvement of current workflows. In the design proposed in this work (Section 4), we will focus on the data placement strategy.

Strategy: Data Placement Data placement encompasses all data movement-related activities such as transfer, staging, replication, space allocation and de-allocation, registering and unregistering metadata, locating and retrieving data[4]. The general idea is to host a dataset on the storage system that is most favourable in terms of performance, cost-effectiveness, and availability for the access pattern observed in the workflow. Here we are considering the optimisation of data locality, where locality is twofold, spatial and temporal, on a variety of characteristics. For optimising data placement, we introduce four approaches: data allocation, data migration, data replication and direct-coupling.

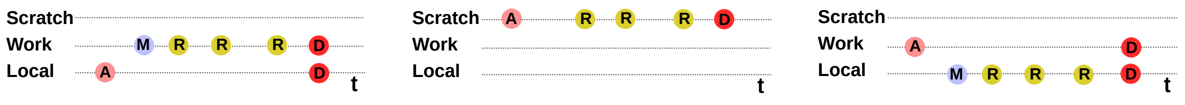
[4]`https://www.igi-global.com/dictionary/data-aware-distributed-batch-scheduling/6782`

Data Allocation is the assignment of a specific area of an available storage system to particular data. In current workflows, the user usually has a script for each task defining the filenames with a prefix that places datasets generated by the same task into a specific storage[5]. Because there is one script responsible for generating the configuration, the decision in which directory the dataset will be stored is somewhat fixed. Such configuration is done manually and with restricted information about the system architecture. It would be interesting to explore storage options for the datasets and, e.g., to have datasets from different cycles placed at different storage systems. For instance, in Fig. 4, alternating the storage location for Product 2 into two `scratch` file systems is something that would be a simple job for an I/O-aware scheduler. However, currently, it implies providing scripts for that task and all tasks depending on it with information about the different storage placement.

Data Migration is the process of transferring data from one storage system to another. Typically, it involves to delete data, but this decision can be delayed to provide read access to multiple storage systems. Data movement involves a significant overhead, both in terms of latency and energy-efficient computing, as data must be read on one storage and written to another. Hence, it needs to be considered carefully. Figure 5 introduces three possible life cycles for a specific dataset and explains how migrations can be done to improve datasets accessibility. In Fig. 5a, the dataset could be first stored on the local storage to avoid congestion on the `work` file system, then it is migrated to `work` file system where subsequent tasks of the workflow may read it multiple times. In the end, this dataset may be an intermediate product that can then be deleted. In Fig. 5b, the dataset is stored on the `scratch` file system immediately and accessed there. However, the last read access must happen before files on `scratch` are automatically removed. Alternatively, Fig. 5c presents the case where the dataset is created on `work` and it is copied to a local node. This local node allows reading accesses of subsequent tasks which might be beneficial for small random accesses. For the last two scenarios, subsequent tasks would have to be placed on the same node where previous data was stored.

Data Replication in computing involves sharing information to ensure consistency between redundant resources, such as software or hardware components, to improve reliability, fault-tolerance, or accessibility.

Data might be replicated by enabling the system to rerun parts of the workflow in case of a data loss. In addition, the system may combine the replication of data by **transforming** data into a different representation allowing to achieve better performance considering a variety of access patterns.



(a) Local and `work` file systems (b) `Scratch` file system only (c) Local and `work` file systems

Figure 5. Alternative life cycles for mapping a dataset to storage and the operations: **A**llocation, **M**igration, **R**eading, and **D**eleting

[5]Complicated scripts would have allowed changing the storage type depending on the cycle. Still, it is a significant burden to the user.

Direct-Coupling replaces I/O by communicating data between subsequent steps of a workflow directly without storing intermediate data products on persistent storage. As an example, in Fig. 4, the outcome of Task 1 may be used directly by Task 2. Data may also be kept in memory and cached to achieve a certain level of independence between producer and consumer.

Strategy: Data Reduction Data reduction decreases the amount of data stored. We discuss here two potential optimisations: data compression and data recomputation.

Data Compression is the process of encoding information using fewer bits than the original representation. Knowing the characteristics of data production and usage allows scientists to annotate the required precision of data in those workflows. The storage system can exploit such information by reducing the precision of data and automatically picking an appropriate compression algorithm.

Data Recomputation Climate/weather scientists are trading recomputation with space usage manually. By knowing how to rerun the workflow behind the data creation, a smarter storage system can automatically trade data availability for potential recomputation opportunities to optimise the cost-efficiency of the system. Intermediate states could be rerun by utilising virtualisation and container technologies. Consider Fig. 4 again and that, at every `K` cycles of the workflow, the generated Product 3 (from Cycle 1 to Cycle `K`) are used in a validation task, called here `check`. From the workflow, we know that P_3C_1[6] will be used to construct P_3C_2 and then `check`. This dataset would probably be stored somewhere, and it will not be used until the workflow reaches the `K`-th cycle. One alternative is to delete it after it was first used and then recompute it when time is right. The cost of doing that is storing `checkpoint` and then use it to reconstruct P_3C_1. If, for instance, P_3C_1 is a large dataset, `checkpoint` is small, and computing time is short, it is easy to see that deleting and recomputing it may improve the costs of running the workflow. That is just an example, and, currently, scientists perform those optimisations manually.

3.4. Benefit

The benefits of the proposed vision are:

Abstraction Providing the abstraction that enables a separation of concerns. Once the I/O characteristics of a workflow are defined, the user does not have to know the architecture of the target system on which the workflow will run. Thus, this level of abstraction can remove the specialist from the decision-making process of individual workflows.

Optimisation The workflow will be optimised specifically for the available system infrastructure and information about data. In particular, by exposing the heterogeneous architecture, potential runtime characteristics can be considered. By using information about the value of data, policies for data management (storage resilience, recomputation, replication, etc.) can be decided.

Performance-portability With both abstraction and optimisation, the user can specify the I/O requirements only once for the tasks of a specific workflow, and the I/O-aware workflow

[6]The P_iC_j notation represents the Product i generated in the Cycle j.

can now run with optimised data storage on any system without user intervention. Even more, if the system characteristics change, e.g., it gets upgraded, an additional storage tier becomes available, or if storage degrades, the I/O-aware workflow could automatically adapt and make use of this new environment.

4. Design

This section describes our first approach to incrementally extend workflows for climate and weather that realises parts of our vision. While individual components such as ESDM and Cylc exist, we have not implemented the described scheduler, yet. To automatically make scheduling decisions, the software stack needs to:

1. deliver information about dataset life cycle together with the workflow, and
2. adapt the resulting workflow, individual scripts, and application executions to consider the potential for data placement strategies.

4.1. System Information

The system information of the design is realized using already available capabilities in the ESDM middleware. We assume ESDM is used as the I/O middleware in the parallel application (with NetCDF or directly) and orchestrates the I/O according to a simplified ESDM configuration file (`esdm.conf`). This file contains information about the available technology in the data centre, its I/O characteristics, and will be used to make decisions about how to prioritise I/O targets. In the example presented in Fig. 6, we have three storage targets: two global accessible file systems (`lustre01` and `lustre02`), and one local file system in `/tmp` that can be accessed via the POSIX backend. Each of them comes with a lightweight performance model and the maximum size of data fragments. The metadata section (Line 24) utilises here a POSIX interface to store the information about ESDM objects. Internally, ESDM creates so-called containers and dataset objects to manage data fragments.

ESDM manages a pool of threads that should be created per compute node to achieve better performance and delegates the assignment of optimal block sizes to the storage backend. Since ESDM supports several (non-POSIX) storage backends, an application can utilise all available storage systems without any modifications to the code. The configuration file is inquired by the application utilising ESDM and steers the distribution of data during I/O. To elucidate the system's behaviour, ESDM distributes a single dataset across multiple storage devices depending on their characteristics. While the current system information and performance model are based on latency and throughput only, ESDM shows that automatic decision making can be made on behalf of the user.

4.2. Extended Workflow Description

The user now has to provide information about the datasets required as input and the generated output for each workflow task in a file called I/O-workflow configuration file (`io.cylc`). An example of an `io.cylc` file is shown in Fig. 7. In this file, information about Task 1 is given as an example, and we expect the extra information about all tasks in the same file. This file could define a cycle flexibly to be a month or a year according to the file `flow.cylc`. The notation

```
"backends": [
    {"type": "POSIX", "id": "work1", "target": "/work/lustre01/projectX/",
        "performance-model" : {"latency" : 0.00001, "throughput" : 500000.0},
        "max-threads-per-node" : 8,
        "max-fragment-size" : 104857600,
        "max-global-threads" : 200,
        "accessibility" : "global"
    },
    {"type": "POSIX", "id": "work2", "target": "/work/lustre02/projectX/",
        "performance-model" : {"latency" : 0.00001, "throughput" : 200000.0},
        "max-threads-per-node" : 8,
        "max-fragment-size" : 104857600,
        "max-global-threads" : 200,
        "accessibility" : "global"
    },
    {"type": "POSIX", "id": "tmp", "target": "/tmp/esdm/",
        "performance-model" : {"latency" : 0.00001, "throughput" : 200.0},
        "max-threads-per-node" : 0,
        "max-fragment-size" : 10485760,
        "max-global-threads" : 0,
        "accessibility" : "local"
    }
],
"metadata": {"type": "POSIX",
        "id" : "md",
        "target" : "./metadata",
        "accessibility" : "global"
    }
```

Figure 6. Example of an ESDM configuration file (`esdm.conf`)

is similar to the specification of Cylc workflows using a nested INI format and, ultimately, files `io.cylc` and `flow.cylc` can be merged.

For each task, inputs and outputs are defined. In the **input** section, each entry specifies the virtual name that is used by ESDM as a filename inside NetCDF. Line 5, for example, it defines that the filename `topography` is mapped to a specific input file and that this dataset does not depend on any previous step of the workflow. The next line specifies that the input filename `checkpoint` should be mapped to the output of Task 1 `checkpoint` dataset from the previous cycle (e.g., the checkpoint generated after completing the last year's production). For the initial cycle, the checkpoint file will be empty, and the application will load `init` data. In the **output** section, the datasets are annotated with their characteristics more precisely. For each variable, a pattern defining how frequently data is output according to the workflow must be provided. Most data is input and output in the periodicity of the cycle. Still, we can have variables with different patterns, such as `varA`, which is output per day regardless of the cycle.

Next, we formally define the expected annotations in all the fields envisioned in the I/O-workflow configuration file:

Name A primary name for the field/data generated. It is extended by a pattern defined in a variable (Lines: 11, 19, 26).

Pattern The frequency of data output (Lines: 12, 20).

Lifetime How long data must be retained on storage (if at all) (Lines: 13, 21).

Type The class type of data, i.e., checkpoint, diagnostics, temporary (Lines: 14, 22, 27).

Datatype The data type of data (Lines: 15, 23, 28).

Size An estimate of data size[7] (Lines: 16, 29).

[7]This field can be inferred if dimension and data type are provided.

```
1  [Task 1]
2
3    [[inputs]]
4
5      topography = "/pool/input/app/config/topography.dat"
6      checkpoint = "[Task 1].checkpoint$(CYCLE - 1)"
7      init = "/pool/input/app/config/init.dat"
8
9    [[outputs]]
10
11     [[[varA]]] # This is the name of the variable
12       pattern = 1 day
13       lifetime = 5 years
14       type = product
15       datatype = float
16       size = 100 GB
17       precision.absolute_tolerance = 0.1
18
19     [[[checkpoint]]]
20       pattern = $(CYCLE)
21       lifetime = 7 days
22       type = checkpoint
23       datatype = float
24       dimension = (100,100,100,50)
25
26     [[[log]]]
27       type = logfile
28       datatype = text
29       size = small
```

Figure 7. External Cylc I/O-workflow configuration file (`io.cylc`)

Dimension The data dimension (Line: 24).

Accuracy Characteristics quantifying the required level of data precision (Line: 17).

Note that the user may not be able to provide all required information which can be handled by assuming a default safe behaviour. For instance, in the case of missing data precision, data should be retained in the original form. Knowing the dimension or size a priori might be difficult for scientists, e.g., the log file size is unclear. In this case, the user may insert relevant information like small or big, indicating that any information is better than no information at all. In future, we will explore ways to infer the output volume from the input automatically. For instance, by running the workflow without I/O specification and monitoring I/O accesses for one cycle, we can propose an I/O description to the user to simplify the specification and generate an experimental I/O configuration file.

4.3. Smarter I/O Scheduling

From the list of opportunities, we realise data placement and migration in a heterogeneous (multi-storage) environment. These goals will be achieved via the proposed I/O-aware scheduler, called here EIOS (ESDM I/O Scheduler). EIOS will make the schedule considering Cylc workflow and ESDM provided system characteristics. We are working together with Cylc Team in developing how EIOS interfaces with Cylc. While Cylc schedules the workflow, EIOS can provide hints about colocating tasks which generate the opportunity for keeping data in local storage. Our design imposes only minor changes to Cylc as normal functionalities cover the core requirements:

The ability to dynamically set the job (Slurm) directives for a task

This will be achieved by calling an external command (run on the Cylc scheduler host) which adds additional directives to be used by the job. This command, provided by EIOS, will determine attributes of previous tasks through simple SQL queries to the Cylc database. We plan on using the Cylc broadcast functionality to change the instructions used by a task by running an external program before any task where we may want to alter the directives.

The ability to dynamically set storage locations

This will be achieved by defining environment variables in the job script which are set to the output of another external command (run on the job host). This command, also provided by EIOS, will have access to all the standard Cylc environment variables with details about the current task.

We plan on utilising DDN's IME API to pin data in IME and trigger migrations between IME and a storage backend explicitly. Decisions about data locality will not be made for a whole (and potentially big) workflow. Instead, the system will make decisions by looking ahead to several steps of the workflow, allowing reacting to the observed dynamics of the execution. Ultimately, when a user-script runs, the information about the intended I/O schedule is communicated from EIOS through a modified filename, which is then used by the ESDM-aware application to determine the data placement.

4.4. Modified Workflow Execution

The steps to execute a workflow enriched with I/O information and perform smarter scheduling are depicted in Fig. 8. Components of EIOS are involved in different steps of the workflow and the I/O path. The suggested alterations can be seen in boxes pointed by red arrows, and the remaining components are the current state-of-the-art for workflows in climate and weather from Fig. 2. In the following, we describe the modifications we propose in this vision paper for each component involved in the software stack.

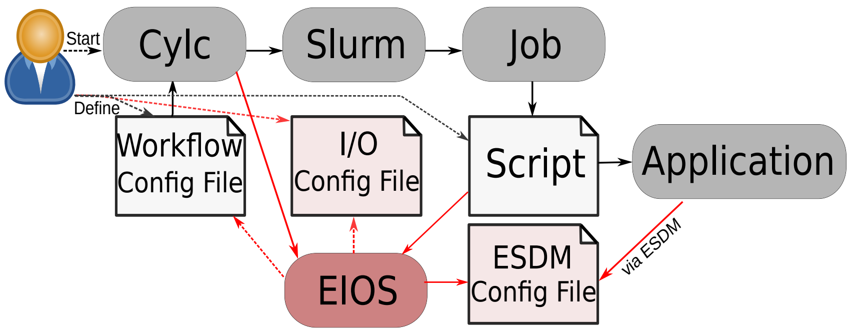


Figure 8. Software stack and stages of execution with the I/O-aware scheduler (EIOS). The red arrows and boxes indicate additions to the workflow compared to Fig. 2

1. **Scientist** The user now has to provide an additional file that covers the I/O information for each task and slight changes have to be made to the current scripts.
2. **Cylc** EIOS is invoked by Cylc to identify potential optimisations in the schedule before generating the Slurm script.

3. **EIOS** The ESDM I/O Scheduler reads the information about the workflow (`flow.cylc` and `io.cylc` configuration files) and acts depending on the stage of the execution. EIOS consists of several subcomponents:
   - The high-level scheduler that interfaces with Cylc.
   - A tool to generate pseudo filenames used by the ESDM-aware applications.
   - A data management service (not shown in the figure) that migrate and purge data at the end of the life cycle.

   EIOS components use knowledge about the system by parsing the `esdm.conf` file. EIOS may decide that subsequent jobs shall be placed on the same node, reorder the execution of some jobs, and provide information for conducting data migration.
4. **Slurm** Cylc may now have added an optimisation identified by EIOS which is promptly handled by a modified Slurm. Also, if migrations have to be performed, Slurm will administer them according to the specification in the job script.
5. **Job** A job might run on the same node as a previous job to utilise local storage.
6. **Script** Filenames are now generated by a replacement command that calls EIOS to create a pseudo filename. This filename will encode additional information for ESDM about how to prioritise data placement according to data access.
7. **Application** The application may either use XIOS, NetCDF with ESDM support or ESDM directly to access datasets. ESDM loads the file `esdm.conf` that contains the information about the available storage backends and their characteristics. ESDM extracts the long-term schedule information from the generated pseudo filenames and employs them during the I/O scheduling to optimise the storage considering data locality among tasks. Basically, ESDM can now change the priorities for data placement in the different storage locations that would typically be encoded in Cylc's configuration file.

5. Potential Benefit

In this section, we discuss the potential performance benefit that our vision for I/O-aware workflows may have considering DKRZ Mistral supercomputer, ESDM current version and a hypothetical workload related to the workflow in Fig. 1. In our scenario, we compare the usage of the node-local file system[8] with a globally shared Lustre file system to store intermediate data. We focus on the model execution and subsequent verification and postprocessing steps. Firstly, checkpoints of a long model execution chain could be stored locally and restart from there. When the subsequent jobs require the whole data to generate a product, they can be run on the same nodes.

Figure 9 shows the read/write performance when using ESDM to store a time series of 10 steps of a variable with 200 k $\times$ 200 k dimension (about 1 km global resolution of the model, equivalent to 3 TB of data in total). Note that, while we only consider the volume data for one variable with one level and ten timesteps, this value could be multiplied by a sensible number of levels and timesteps of the model. Data is stored on either Lustre02 or both Lustre file systems – ESDM splits data of a single variable internally and distributes them across the file systems. While the mapping is not yet optimal, the figure shows that the write performance benefits from this approach. In our observations, performance will not improve beyond 500 nodes, which

[8]We assume the availability and fault-tolerance of the nodes is non-uncritical for the particular workload – typically nodes can be repaired and returned to the pool within days.

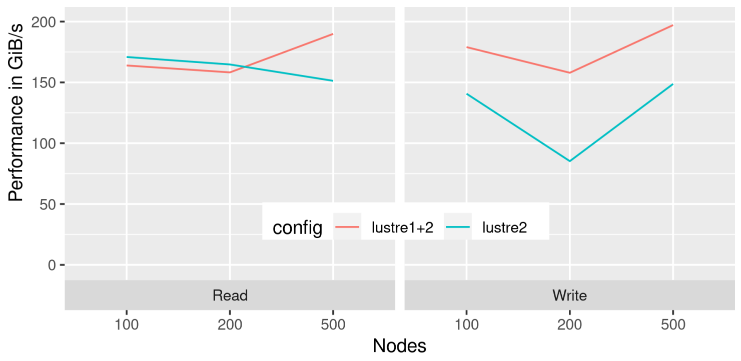


Figure 9. Lustre performance for 100, 200, and 500 nodes

might be due to the fact that each Lustre file system has 128 Object Storage Targets (256 in total).

Another improvement can be achieved by using local storage. Each local SSD of Mistral is a Micron M600 MTFD[9] (256 GB), which has a nominal sequential read/write performance slightly above 500 MB/s. Hence, with 500 nodes, we could achieve 250 GB/s, which surpasses the Lustre performance observed in Fig. 9. Even more, our 3 TB of data would be about 6 GB per node, which could be cached in memory and overlap with the computation phases. An additional benefit of using local storage is that the interference with I/O activities of other jobs would be minimised. Actually, for all reasonable sizes of the experimental data with 500+ nodes, the observed performance of node-local storage would be higher, and thus, improving the workflow execution time. Since DKRZ has more than 3,000 nodes, using the local SSD would sum up to 1.5 TB/s speeding up the IO phase by 7x. For model runs with 1 km resolution, such configurations would be reasonable. It might also be suitable to couple the model with a parallel analysis process directly using an in-memory file system such as `tmpfs`. In this case, the performance per node can be assumed to be 3 GiB/s, making it a viable option for smaller runs.

Users can always use the node-local storage manually and create specific run-scripts to reproduce similar behaviours utilising the local storage. However, this would be tedious and error-prone. The purpose of our proposal is to establish an abstraction layer to allow for semi-automatic decision making and reduce, or even remove manual intervention.

Conclusions

In the domain of climate and weather, organising data placement on storage tiers is performed by the users or via policies, often leading to suboptimal decisions. Additionally, manual optimisation and hard-coding of storage locations are non-portable and an error-prone task. We believe users must be able to express their workflows abstractly. By increasing the abstraction level for scientists, not only tedious manual optimisation could be automatised, but also strategies for data placement and data reduction can be harnessed. With knowledge about the data pattern, the runtime system could generate optimised execution plans and monitor their execution. In this work, we describe the overall vision and a specific design for the software stack in the domain of climate and weather that we work on in the ESiWACE project. The proposed changes increase the opportunity for smarter scheduling of storage in heterogeneous storage environments by considering the characteristics of data and system architecture in the workflow.

[9] `https://www.anandtech.com/show/8528/micron-m600-128gb-256gb-1tb-ssd-review-nda-placeholder`

Acknowledgements

This project is funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 823988. We thank our collaborators Bryan Lawrence, Glenn Greed, David Matthews, and Hua Huang for their input to this paper, and the NGI initiative for contributions to the vision.

References

1. Alkhanak, E.N., Lee, S.P., Rezaei, R., Parizi, R.M.: Cost optimization approaches for scientific workflow scheduling in cloud and grid computing: a review, classifications, and open issues. Journal of Systems and Software 113, 1–26 (2016), DOI: 10.1016/j.jss.2015.11.023

2. Betke, E., Kunkel, J.: Benefit of DDN's IME-Fuse and IME-Lustre file systems for I/O intensive HPC applications. In: Yokota, R., Weiland, M., Shalf, J., Alam, S. (eds.) High Performance Computing: ISC High Performance 2018 International Workshops, Frankfurt/Main, Germany, 28 June, 2018, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11203, pp. 131–144. ISC Team, Springer (2019), DOI: 10.1007/978-3-030-02465-9_9

3. Braam, P.: The Lustre storage architecture. CoRR abs/1903.01955 (2019), `http://arxiv.org/abs/1903.01955`

4. Center, U.P.: Network Common Data Form (NetCDF), DOI: 10.5065/D6H70CW6

5. Chowdhury, F., Zhu, Y., Heer, T., Paredes, S., Moody, A.T., Goldstone, R., Mohror, K.M., Yu, W.: The parallel I/O architecture of the high-performance storage system (HPSS). In: Proceedings of the 48th International Conference on Parallel Processing, August 2019, Kyoto, Japan. pp. 1–10 (2019), DOI: 10.1145/3337821.3337902

6. Dai, D., Ross, R., Khaldi, D., Yan, Y., Dorier, M., Tavakoli, N., Chen, Y.: A cross-layer solution in scientific workflow system for tackling data movement challenge. CoRR abs/1805.061675 (2018), `https://arxiv.org/abs/1805.06167`

7. Deelman, E., Mandal, A., Jiang, M., Sakellariou, R.: The role of machine learning in scientific workflows. The International Journal of High Performance Computing Applications 33(6), 1128–1139 (2019), DOI: 10.1177/1094342019852127

8. Di Tommaso, P., Chatzou, M., Floden, E.W., Barja, P., Palumbo, E., Notredame, C.: Nextflow enables reproducible computational workflows. Nature Biotechnology 35, 316–319 (2017), DOI: 10.1038/nbt.3820

9. Isard, M., Budiu, M., Yu, Y., Birrell, A., Fetterly, D.: Dryad: Distributed data-parallel programs from sequential building blocks. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, March 2007, Lisbon, Portugal. p. 59–72. Association for Computing Machinery, New York, NY, USA (2007), DOI: 10.1145/1272996.1273005

10. Jette, M.A., Yoo, A.B., Grondona, M.: SLURM: Simple Linux Utility for Resource Management. In: Proceedings of Job Scheduling Strategies for Parallel Processing, 24 June, Seattle, WA, USA. Lecture Notes in Computer Science, vol. 2862, pp. 44–60. Springer, Berlin, Heidelberg (2002), DOI: 10.1007/10968987_3

11. Jimenez, I., Sevilla, M., Watkins, N., Maltzahn, C., Lofstead, J., Mohror, K., Arpaci-Dusseau, A., Arpaci-Dusseau, R.: The popper convention: making reproducible systems evaluation practical. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops, 29 May-2 June 2017, Lake Buena Vista, FL, USA. pp. 1561–1570. IEEE (2017), DOI: 10.1109/IPDPSW.2017.157

12. Kougkas, A., Devarajan, H., Sun, X.H.: I/O acceleration via multi-tiered data buffering and prefetching. Journal of Computer Science and Technology 35(1), 92–120 (2020), DOI: 10.1007/s11390-020-9781-1

13. Köster, J., Rahmann, S.: Snakemake: a scalable bioinformatics workflow engine. Bioinformatics 28(19), 2520–2522 (2012), DOI: 10.1093/bioinformatics/bts480

14. Lawrence, B.N., Kunkel, J.M., Churchill, J., Massey, N., Kershaw, P., Pritchard, M.: Beating data bottlenecks in weather and climate science. In: Extreme Data Workshop – Forschungszentrum Jülich, Proceedings, IAS series. vol. 40, pp. 31–36 (2018)

15. Liu, J., Pacitti, E., Valduriez, P., Mattoso, M.: A survey of data-intensive scientific workflow management. Journal of Grid Computing 13(4), 457–493 (2015), DOI: 10.1007/s10723-015-9329-8

16. Lüttgau, J., Snyder, S., Carns, P., Wozniak, J.M., Kunkel, J., Ludwig, T.: Toward understanding I/O behavior in HPC workflows. In: IEEE/ACM 3rd International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, 12 Nov. 2018, Dallas, Texas. pp. 64–75. IEEE Computer Society, Washington, DC, USA (2019), DOI: 10.1109/PDSW-DISCS.2018.00012

17. Meurdesoif, Y., Caubel, A., Lacroix, R., D'erouillat, J., Nguyen, M.H.: XIOS Tutorial (2016), `http://forge.ipsl.jussieu.fr/ioserver/raw-attachment/wiki/WikiStart/XIOS-tutorial.pdf`

18. Miranda, A., Jackson, A., Tocci, T., Panourgias, I., Nou, R.: NORNS: extending Slurm to support data-driven workflows through asynchronous data staging. In: 2019 IEEE International Conference on Cluster Computing, 23-26 Sept. 2019, Albuquerque, NM, USA. pp. 1–12. IEEE (2019), DOI: 10.1109/CLUSTER.2019.8891014

19. Oliver, H., Shin, M., Matthews, D., Sanders, O., Bartholomew, S., Clark, A., Fitzpatrick, B., van Haren, R., Hut, R., Drost, N.: Workflow automation for cycling systems: the Cylc workflow engine. Computing in Science Engineering 21(4), 7–21 (2019), DOI: 10.1109/MCSE.2019.2906593

20. Ozik, J., Collier, N.T., Wozniak, J.M., Spagnuolo, C.: From desktop to large-scale model exploration with Swift/T. In: 2016 Winter Simulation Conference, 11-14 Dec. 2016, Washington, DC, USA. pp. 206–220. IEEE (2016), DOI: 10.1109/WSC.2016.7822090

21. Rajasekar, A., Moore, R., Hou, C.y., Lee, C.A., et al.: iRODS primer: integrated rule-oriented data system. Synthesis Lectures on Information Concepts, Retrieval, and Services 2(1), 1–143 (2010), DOI: 10.2200/S00233ED1V01Y200912ICR012

22. Romanus, M., Ross, R.B., Parashar, M.: Challenges and considerations for utilizing burst buffers in high-performance computing. CoRR abs/1509.05492 (2015), `http://arxiv.org/abs/1509.05492`

23. Schmuck, F., Haskin, R.: Gpfs: A shared-disk file system for large computing clusters. In: Proceedings of the 1st USENIX Conference on File and Storage Technologies, Monterey, CA. pp. 231–244. USENIX Association, USA (2002), DOI: 10.5555/1083323.1083349

24. Slawinska, M., Clark, M., Wolf, M., Bode, T., Zou, H., Laguna, P., Logan, J., Kinsey, M., Klasky, S.: A Maya use case: adaptable scientific workflows with ADIOS for general relativistic astrophysics. In: Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery, July 2013, San Diego, California, USA. pp. 1–8. Association for Computing Machinery, New York, NY, USA (2013), DOI: 10.1145/2484762.2484795

25. Subedi, P., Davis, P.E., Parashar, M.: Leveraging machine learning for anticipatory data delivery in extreme scale in-situ workflows. In: 2019 IEEE International Conference on Cluster Computing, 23-26 Sept. 2019, Albuquerque, NM, USA. pp. 1–11. IEEE (2019), DOI: 10.1109/CLUSTER.2019.8891003

26. Watson, R.W., Coyne, R.A.: The parallel I/O architecture of the high-performance storage system, 11-14 Sept. 1995, Monterey, CA, USA. In: Proceedings of IEEE 14th Symposium on Mass Storage Systems. pp. 27–44. IEEE (1995), DOI: 10.1109/MASS.1995.528214

27. Wozniak, J.M., Armstrong, T.G., Wilde, M., Katz, D.S., Lusk, E., Foster, I.T.: Swift/T: Large-Scale Application Composition via Distributed-Memory Dataflow Processing. In: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, 13-16 May 2013, Delft, Netherlands. pp. 95–102. IEEE (2013), DOI: 10.1109/CCGrid.2013.99

DOI: 10.14529/jsfi200204

Bridging the Architecture Gap: Abstracting Performance-Relevant Properties of Modern Server Processors

***Johannes Hofmann*[1], *Christie L. Alappat*[2], *Georg Hager*[2], *Dietmar Fey*[1], *Gerhard Wellein*[2]**

We propose several improvements to the execution-cache-memory (ECM) model, an analytic performance model for predicting single- and multicore runtime of steady-state loops on server processors. The model is made more general by strictly differentiating between application and machine models: an application model comprises the loop code, problem sizes, and other runtime parameters, while a machine model is an abstraction of all performance-relevant properties of a processor. Moreover, new first principles underlying the model's estimates are derived from common microarchitectural features implemented by today's server processors to make the model more architecture independent, thereby extending its applicability beyond Intel processors.

We introduce a generic method for determining machine models, and present results for relevant server-processor architectures by Intel, AMD, IBM, and Marvell/Cavium. Considering this wide range of architectures, the set of features required for adequate performance modeling is surprisingly small.

To validate our approach, we compare performance predictions to empirical data for an OpenMP-parallel preconditioned CG algorithm, which includes compute- and memory-bound kernels. Both single- and multicore analysis shows that the model exhibits average and maximum relative errors of 5 % and 10 %. Deviations from the model and insights gained are discussed in detail.

Keywords: microarchitecture comparison, Intel, AMD, ARM, IBM, performance evaluation, performance modeling, analytic modeling, execution-cache-memory model.

Introduction

The architectural differences among processor models of different vendors (and even among models of a single vendor) lead to a diverse server-processor landscape in the high-performance computing market. On the other hand, several analytic performance models, such as the Roofline model [10, 25] and the execution-cache-memory (ECM) model [6, 13], show that many relevant performance features can be described using a few key assumptions and a small set of numbers such as bandwidths and peak execution rates. In this work we introduce a structured method of establishing and describing those assumptions and parameters that best summarize the features of a multicore server processor. It has satisfactory predictive power in terms of performance modeling of (sequences of) steady-state loops with regular access patterns but is still simple enough to be carried out with pen and paper. The overarching goal is to allow comparisons among microarchitectures not based on benchmarks alone, which have narrow limits of generality, but based on abstract, parameterized performance models that can be used to attribute performance differences to one or a few parameters or features. As a consequence, reasoning about code performance from an architectural point of view becomes rooted in a scientific process.

Main contributions

We describe an abstract workflow for predicting the runtime and performance of sequential and parallel steady-state loops (or sequences thereof) with regular access patterns on multicore

[1]Friedrich-Alexander-University Erlangen-Nuremberg, Germany
[2]Erlangen Regional Computing Center, Germany

server CPUs. The core of the method is an abstract formulation of the ECM model, which is currently the only analytic model capable of giving accurate single- and multicore estimates.

We show that a separation between the *machine model*, which contains hardware features alone, and the *application model*, which comprises loop code and execution parameters, is possible with some minor exceptions.

We describe a formalized way to establish a machine model for a processor architecture and present results for Intel Skylake SP and, for the first time, for AMD Epyc, IBM Power9, and Marvell/Cavium ThunderX2 CPUs. The degree of data-transfer overlap in the memory hierarchy is identified as a key parameter for the single-core in-memory performance of data-bound code.

The feasibility of the approach is demonstrated by predicting runtime and performance of a preconditioned conjugate-gradient (PCG) solver and comparing estimates to empirical data for all investigated processors. ECM predictions for the AMD, Cavium, and IBM CPUs have not been published before.

Outline

This paper is structured as follows. In Section 1 we detail our testbed and methodology. Section 2 describes, in general terms, our modeling approach including application model, machine model, and the modeling workflow. Section 3 shows how machine models can be constructed by analyzing data from carefully chosen microbenchmarks and gives results for the four CPU architectures under consideration. In Section 4 we validate the model by giving runtime and performance predictions for a PCG solver and comparing them to measurements. Finally, Section 5 puts our work in the context of existing research and Section 5 summarizes and concludes the paper.

1. Methodology and Testbed

In this section we point out some relevant high-level properties, while details will be discussed later. Note that we generally take care to run the optimal instruction mix for all benchmark kernels (i.e., using the most recent instruction sets available on the hardware at hand, with appropriate unrolling in place to enable optimal instruction-level parallelism). Compiler peculiarities are commented on where necessary. To minimize interference from the operating system, NUMA balancing was disabled. Transparent huge pages were used by default. Measurements were carried out on repeated loop traversals so timer resolution was not an issue. Run-to-run variations were small (generally below 2 %) and will thus not be reported.

An overview of the investigated processors is provided in Tab. 1. The AMD Epyc 7451 (EPYC) has a hierarchical design comprising four ccNUMA nodes per socket and six cores per domain. L3 cache segments of 8 MiB each are shared among the three cores of a core complex (CCX). The Uncore of the processor (i.e., the L3 cache, memory interface, and other I/O circuitry) is clocked at a fixed 2.66 GHz. Although the cores support the AVX2 instruction set, 32-byte (B) wide SIMD instructions are executed in two chunks of 16 B by only 16-B wide hardware, so that an effective SIMD width of 16 B applies.

Although the Intel Xeon Skylake Gold 6148 (SKL) has a base core frequency of 2.4 GHz and a wide range of Turbo settings, we fix the clock speed to 2.2 GHz in all our experiments in order to avoid the automatic clock-speed reduction when running AVX-512 code [16]. The AVX-512 SIMD extensions were introduced with the SKL architecture and provide 64-B wide vector registers and execution units. The Uncore frequency is set to its nominal value of 2.4 GHz.

Table 1. Key specifications of testbed machines

Microarchitecture	Zen (EPYC)	Skylake-SP (SKL)	Vulcan (TX2)	Power9 (PWR9)
Chip Model	Epyc 7451	Gold 6148	ThunderX2 CN9980	8335 GTX EP0S
Supported core freqs	1.2–3.2 GHz	1.2–3.7 GHz	2.2–2.5 GHz	2.8–3.8 GHz
De-facto freq.	2.3 GHz	2.2 GHz	2.2 GHz	3.1 GHz
Supported Uncore freqs	2.66 GHz	1.2–2.4 GHz	1.1 GHz	N/A
Cores/Threads	24/48	20/40	32/256	22/88
SIMD extensions	AVX2	AVX-512	NEON	VSX-3
L1 cache capacity	24×32 KiB	20×32 KiB	32×32 KiB	22×32 KiB
L2 cache capacity	24×512 KiB	20×1 MiB	32×256 KiB	11×512 KiB
L3 cache capacity	8×8 MiB	27.5 MiB	32 MiB	110 MiB
Memory Configuration	8 ch. DDR4-2666	6 ch. DDR4-2666	8 ch. DDR4-2400	8 ch. DDR4-2666
Theor. Mem. Bandwidth	170.6 GB/s	128.0 GB/s	153.6 GB/s	170.6 GB/s

These choices are not a limitation of generality since all procedures described in this work can be carried out for any clock-speed setting. SKL also features a boot-time configuration option of sub-NUMA clustering (SNC), which splits the 20-core chip into two ccNUMA nodes, each comprising ten cores (while the full L3 is still available to all cores). This improves memory-access characteristics and is thus a recommended operating mode for HPC in our opinion. The last-level cache (LLC) prefetcher was turned on for the same reason.

The Cavium/Marvell ThunderX2 CN9980 (TX2) implements the ARMv8.1 ISA with 128-bit NEON SIMD extensions that support double-precision floating-point arithmetic for a peak performance of two 16-B wide FMA instructions per cycle and core. The 32-core chip runs at a fixed 2.2 GHz clock speed, while the L3 cache runs at half the core speed. The victim L3 cache is organized in 2 MiB slices but shared among all cores of the chip.

The Power9 processor used for our investigations is part of an IBM 8336 GTX data analytics/AI node. Being an implementation of the Power ISA v3.0, the core supports VSX-3 SIMD instructions, corresponding to 16-B wide vector registers. A 512 KiB L2 cache is shared between each pair of cores. The victim L3 cache is segmented, with eleven slices of 10 MiB each, and each slice can act as a victim cache for others [20].

High-level language code for both the Intel and AMD processors was compiled with the Intel C compiler (version 19.0 update 2). On the Marvell and IBM processors the ARM clang (version 19) and the IBM XL C (version 16.1.0) compilers were used, respectively. To get the compiler generate an appropriate instruction mix, the `-O3`, `-xHost`, `-mavx2`, and `-mavx` compiler flags are required for the AMD Epyc processor. For the Intel Skylake processor, the `-O3`, `-xCORE-AVX512`, and `-qopt-zmm-usage=high` flags were used. For the Marvell TX2 processor, the `-Ofast` and `-mtune=native` flag were employed. Finally, for the IBM Power9 processor, the `-O5`, `-qarch=pwr9`, and `-qsimd=auto` flags were used.

Note that the particular choice of compilers was to some extent arbitrary, because it is not our intention to provide a comprehensive compiler comparison. It must be understood that compilers may fail to produce "optimal" code for a loop, but modeling procedures like the one we show here can be used to pinpoint such deficiencies.

The likwid suite [5] version 4.3.3 was used in several contexts: `likwid-pin` for thread-core affinity, `likwid-perfctr` for counting hardware performance events, and `likwid-bench` for low-level loop benchmarking (with customized kernels for TX2 and PWR9). Instruction latency and throughput were measured using the `ibench` tool [11]. Where compiled code was required, we used the compiler versions and flags indicated in Tab. 1.

2. Modeling Approach

Just like the Roofline model, the ECM model is an analytic performance model for streaming loop kernels with regular data-access patterns and a uniform amount of work per loop iteration. Unlike Roofline, however, ECM favors an analytic approach. As a result, the model can give single- and multicore estimates with high accuracy without relying on a large number of measurements. Moreover, the analytic nature enables the evaluation of different hypotheses with respect to a processor's performance behavior by investigating which of them lead to a model that best describes empirical performance, thereby enabling deeper insights than measurement-based approaches such as Roofline. See Section 5 for a more thorough comparison of the models and their predictive powers.

Two major shortcomings of the ECM model concern its loose formulation and the resulting lack of portability: in its current form, the model mixes general first principles and Intel-specific microarchitectural behavior into a set of rules that make it difficult to apply it to other processors. In the following, we untangle the original model: First, several truly general (i.e., microarchitecture-independent) first principles and their rationales are laid out. Next, application and machine models that address code- and microarchitecture-specific properties are covered (in addition, we provide general instructions on how to determine machine models for new microarchitectures in Section 3). Finally, the workflow of the new model is demonstrated.

2.1. Model Assumptions

The model assumes that the single-core runtime is composed of different runtime components. These include the time required to execute instructions in the core (T_{core}) and the runtime contributions that result from carrying out the necessary data transfers in the memory hierarchy (e.g., T_{RegL1} the time to transfer data between the register file and the L1 cache, T_{L1L2} for L1-L2 transfers, and so on). Depending on the architecture, some or all of these components may overlap. The single-core runtime estimate is therefore derived from the runtime components by putting them together according to the architecture's overlap capabilities.

If no shared resources are involved, single-core performance is assumed to scale linearly with the number of active cores for the multicore estimate. In practice, however, at least one shared resource (the memory interface) will be involved. The model takes conflicts on shared resources into account by modeling contention and the resulting waiting times in an analytical way. In the following, some particularities of modern server processors that simplify runtime modeling are discussed.

Today's server processors typically feature superscalar, out-of-order cores that support speculative execution and implement pipelined execution units. Figure 1 shows the execution of instructions corresponding to a simple vector sum (`C[i]=A[i]+B[i]`) for a data set in the L1 cache on a hypothetical core. The core has a two-cycle latency for add and load instructions. When the loop begins execution, each of the two load units can execute a load instruction. Since there is a two-cycle load latency, inputs for the add instruction will only be available after two cycles. However, due to speculative execution, the core can continue to execute two load instructions from the next loop iterations in each cycle. Once input data is available, the

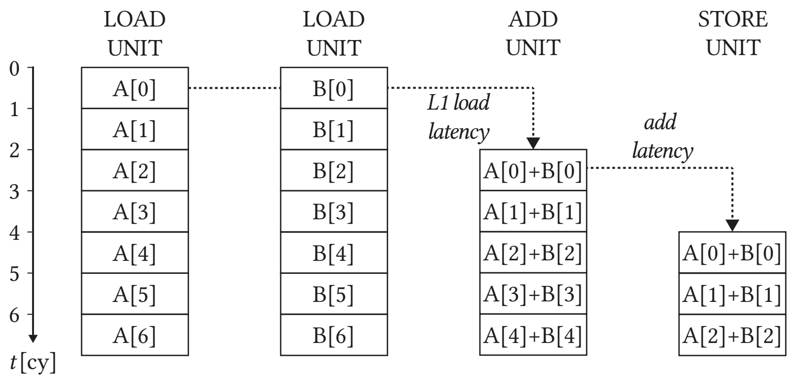


Figure 1. Loop execution on a hypothetical core with load and add latencies of two cycles each

core can begin executing an add instruction in each cycle. Eventually, after another two-cycle latency (that of the add instruction), the core can begin executing a store instruction in each cycle. Once this latency-induced wind-up phase of four cycles is complete, instruction latency no longer impacts runtime; instead, the runtime is determined by the throughput of instructions. Although latencies might be higher on real processors, the wind-up phase can be neglected even for short loops with only hundreds of iterations. This leads to one of the key assumptions of the ECM model: in the absence of loop-carried dependencies and data-access delays from beyond the L1 data cache, the runtime of a single loop iteration can be approximated by the time that is required to retire the instructions of a loop iteration. With loop-carried dependencies in place, the inter-iteration critical path is a good estimate of the runtime. Due to speculative execution, load/store instructions are decoupled from the arithmetic instructions of a particular loop iteration. This leads to the further assumption that the time to retire arithmetic instructions and the time to retire load/store instructions can overlap.

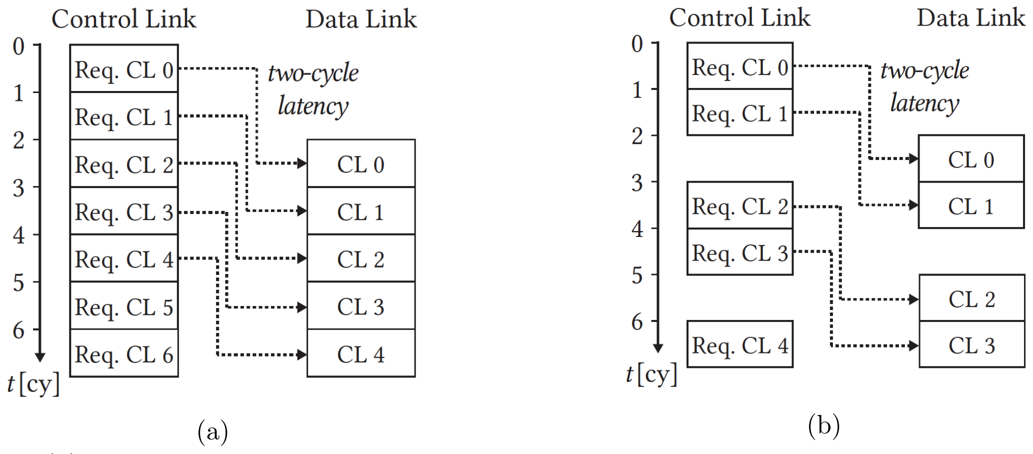


Figure 2. (a) Inter-cache data transfers for a design with more than two buffers to track outstanding cache-line (CL) transfers; (b) design with only two buffers

The next set of assumptions concerns data transfers in the memory hierarchy. The relationship between latency and bandwidth is well understood, so most designs typically provide a sufficient number of buffers to track outstanding cache-line transfers to allow for the saturation of the data-transfer link between adjacent cache levels. Figure 2a shows such a design with more than two buffers to track outstanding transfers to hide a two-cycle latency. Sometimes, however, the number of buffers is insufficient, leading to a deterioration of bandwidth. Figure 2b shows a variant with only two buffers: after two cycles, no more transfer-tracking buffers are available, which prevents the initiation of new transfers. Only after a previous transfer completes and the buffer tracking this transfer is freed can a new transfer request be initiated. As a result, the data link is idle for one cycle, reducing the attainable bandwidth in practice to two-thirds of the theoretical value. On some of the investigated processors this problem can be observed for transfers between the LLC and main memory. This can be attributed to significant latencies caused by the increasingly complex on-chip networks required to accommodate the growing number of cores of modern CPUs.

The model assumes that data links can typically be fully saturated because a sufficient amount of buffers are available and adequate prefetching (be it hardware, software, or both) results in full utilization of these buffers. As a result, runtime contributions of data transfers can typically be calculated by dividing data volumes by the theoretical bandwidths of the corresponding links; the model does, however, include an optional latency penalty to cover edge cases such as the one shown in Fig. 2b. Therefore, the runtime contribution of data transfers between memory hierarchy levels i and j is the sum of the actual data transfer time and an optional latency penalty: $T_{ij} = T_{ij}^{\mathrm{data}} + T_{ij}^{\mathrm{p}}$.

2.2. Application Model

An application model condenses all of the code-related information required to give runtime estimates for a particular loop.

It comprises all operations carried out during one loop iteration as well as parameters that influence data transfers in the memory hierarchy. Most prominently, the latter includes the data-set size(s), which determine in which level of the memory hierarchy data resides, yet it may also cover information about blocking size(s) and the scheduling strategy.

2.3. Machine Model

Machine models comprise selected key information about processors. Despite being limited to few architectural properties, the data included in machine models is sufficient to give meaningful performance estimates. With respect to scope, the contents of machine models can be separated into two parts: the execution capabilities of cores, and details about the memory hierarchy. In the following, each of the two components is discussed in detail.

The part concerning in-core execution capabilities deals with the cores' properties that determine the runtime contribution of instruction execution. As discussed in Section 2.1, throughput is a key determinant for single-core runtime, so throughput limits (in operations per cycle) of relevant operations are included. To address loop-carried dependencies, latencies for the corresponding instructions must be included. Moreover, the machine model includes information about potential bottlenecks that limit operation throughput: On most architectures, different functional units share the same execution port, which implies that operations associated with units served by the same port cannot cannot begin execution in the same cycle. Finally, most modern core designs have some architectural deficiency that prevents them from fully utilizing the core's load/store units[3].

[3]Most modern cores feature one store and two load units but only have two address-generation units (AGUs), which means that in each cycle only two of the three load/store units can be supplied with memory addresses if complex addressing modes (e.g., base plus scaled offset) are used. In addition to the two-AGU shortcoming, the EPYC's cores have only two data paths between the register file and the L1 cache.

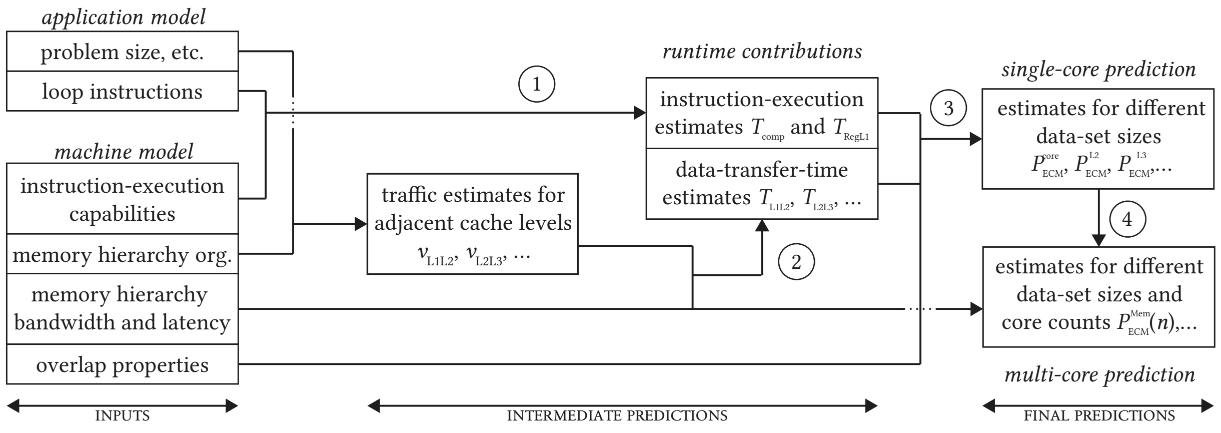

application model
problem size, etc.
loop instructions
machine model
instruction-execution capabilities
memory hierarchy org.
memory hierarchy bandwidth and latency
overlap properties
1
2
3
4
traffic estimates for adjacent cache levels v_{L1L2}, v_{L2L3}, ...
runtime contributions
instruction-execution estimates T_{comp} and T_{RegL1}
data-transfer-time estimates T_{L1L2}, T_{L2L3}, ...
single-core prediction
estimates for different data-set sizes P^{core}_{ECM}, P^{L2}_{ECM}, P^{L3}_{ECM},...
estimates for different data-set sizes and core counts $P^{Mem}_{ECM}(n)$,...
multi-core prediction
INPUTS
INTERMEDIATE PREDICTIONS
FINAL PREDICTIONS

Figure 3. Overview of the performance prediction workflow, including application model, machine model, and runtime contributions

The second part of the machine model covers information about the cache hierarchy. This entails everything needed to calculate the volume of data transfers for a loop: the number of cache levels, their effective[4] sizes, write-through vs. write-back policy, victim/exclusive vs. inclusive, etc. For example, a victim cache typically implies additional traffic since it receives both modified and unmodified cache lines (CLs) from the overlying cache, whereas a non-victim cache only receives modified CLs. In order to get from data volumes to runtime contributions of individual data paths, the machine model also requires data about the available bandwidth between adjacent caches, and whether transfers take place over a single bi-directional link or over two uni-directional links. Moreover, if an architecture provides an inadequate number of buffers to track outstanding transfers, the corresponding latency penalties must be included. Finally, the second part of the machine model contains a description of which transfers in the memory hierarchy can occur simultaneously[5].

2.4. Performance Prediction Workflow

An overview of the performance-prediction workflow is provided in Fig. 3. As indicated in the figure, the process can be divided into four steps: first, the runtime contribution of performing operations in the core (with all data coming from L1) is determined. Next, the runtime contributions of data transfers in the memory hierarchy are calculated (to this end, data transfer volumes in the memory hierarchy need to be determined). In a third step, the previously determined runtime contributions are put together to arrive at a single-core runtime estimate. Finally, based on the single-core estimate from the previous step, multicore predictions can be given. In the following, each of the steps is discussed in detail.

[4]For several reasons (imperfect cache replacement strategies, prefetchers preempting data that could have otherwise been reused, etc.) the effective capacity of a cache is lower than its nominal size. In practice, the heuristic of halving the theoretical cache size delivers good estimates for the effective size.

[5]As will be demonstrated later, we find that in practice, this rarely discussed architectural feature turns out to be much more important for single-core in-memory performance than other more prominent features such as SIMD width or cache bandwidths.

2.4.1. Contributions of instruction execution in the core

The fact that some architectures cannot overlap data transfers between the register file and the L1 cache on one hand and the L1 and L2 caches on the other makes it necessary to separate the runtime contribution of operations into two components: T_{comp}, which are cycles in which no data transfers between registers and L1 cache occur, and T_{RegL1}, which are cycles in which at least one load or store operation retires. Unless otherwise indicated, all arithmetic and load-store operations handle double-precision floating-point operands.

To estimate T_{RegL1}, first the numbers of load and store operations (n_{LD} and n_{ST}) are determined by counting their occurrences in the loop body; the numbers are then divided by the respective throughputs, τ_{LD} and τ_{ST}, taking additional constraints specified in the machine model into account (e.g., a limited throughput for the overall number of load/store operations per cycle, $\tau_{\text{LD/ST}}$, caused by a limited number of AGUs). The corresponding runtime contribution is the maximum of all components:

$$T_{\text{RegL1}} = \max\left(\frac{n_{\text{LD}}}{\tau_{\text{LD}}}, \frac{n_{\text{ST}}}{\tau_{\text{ST}}}, \frac{n_{\text{LD}} + n_{\text{ST}}}{\tau_{\text{LD/ST}}}\right). \tag{1}$$

The number of cycles in which no load/store operations are carried out is determined in a similar way: operation counts are found in the loop body. Each count is then divided by the operation's throughput documented in the machine model. As before, additional constraints have to be considered: For example, execution-port conflicts (cf. Section 2.3) can be addressed by summing up the contributions of functional units that share the same execution port (this is demonstrated in the equation below, where MUL and DIV units are assumed to be assigned to the same execution port). The fact that cores have an upper limit to the number of instructions they can retire per cycle can be modeled by dividing the total number of operations by a corresponding instruction-throughput limit τ_{total}. Finally, loop-carried dependencies are accounted for by including the contribution of the longest cross-iteration dependency chain, T_{dep}, when determining the overall runtime by applying the maximum to all individual contributions:

$$T_{\text{comp}} = \max\left(\frac{n_{\text{ADD}}}{\tau_{\text{ADD}}}, \frac{n_{\text{MUL}}}{\tau_{\text{MUL}}} + \frac{n_{\text{DIV}}}{\tau_{\text{DIV}}}, \ldots, \frac{\sum_i n_i}{\tau_{\text{total}}}, T_{\text{dep}}\right). \tag{2}$$

2.4.2. Contributions of data transfers in the memory hierarchy

Before the runtime contributions of data transfers can be determined, the data volumes transferred over the various data paths in the memory hierarchy need to be established. To this end, the location of the data set(s) in the memory hierarchy is derived from the data-set size(s) specified in the application model. Then, the load/store operations documented in the application model are revisited: for each operation, the corresponding data set is identified, and the transfers required to get the data from its current location in the memory hierarchy to the L1 cache are recorded. Along with the required transfers, the data volume is determined (e.g., four bytes per single- or eight bytes per double-precision floating-point number). Note that full CL transfers need to be taken into account even when CLs are only partially read or written (e.g., for strided but regular access). In case of truly random access patterns, latency contributions will dominate. This case is not part of the ECM model yet, although it is possible to incorporate it in a phenomenological way [2]. Extending the analytic model towards random accesses is part of future work.

Note that determining data-transfer volumes requires keeping track of previous data accesses to detect possible data reuse. While this can be done manually for kernels with simple data-access patterns, analysis of complex patterns is best left to cache simulators (e.g., pycachesim [8]). To this end, per-loop traffic estimates from cache simulators can be used as inputs in Eq. 3. If necessary, the resulting numbers can be validated by measuring the actual data volumes using hardware performance events (e.g., with PAPI [23] or LIKWID [5]).

Once the data volumes have been established, the runtime contribution T_{ij} of data transfers between levels i and j of the memory hierarchy can be calculated:

$$T_{\text{ij}} = \max/\text{sum}\left(\frac{v_{i\to j}}{b_{i\to j}}, \frac{v_{i\leftarrow j}}{b_{i\leftarrow j}}\right) + T^{\text{p}}_{ij} = T^{\text{data}}_{\text{ij}} + T^{\text{p}}_{ij}. \tag{3}$$

The process works by first calculating the time the data link(s) connecting levels i and j are actually busy transferring data. To calculate this data-link busy time, T^{data}_{ij}, the data volumes, v, transferred in each direction are divided by the bandwidth, b, of the link over which the data is transferred. The two directional components $T_{i\to j}$ and $T_{i\leftarrow j}$ are then combined according to the information provided in the machine model. If there is a single bi-directional link over which transfers in both directions take place, the combined data-link busy time is the *sum* of both contributions. If there are two dedicated uni-directional links over which the transfers can take place, the overall data-link busy time is the *maximum* of both contributions. The overall data-transfer time, T_{ij}, is given by the sum of the previously determined data-link busy time and (if applicable) the corresponding latency penalty specified in the machine model.

2.4.3. Combination of runtime contributions for single-core estimate

To arrive at a single-core runtime prediction, the previously determined components are put together according to the overlap capabilities specified in the machine model. To this end, first, all non-overlapping components are added up. The result is then included in the set of overlapping components, and the total runtime estimate is the maximum of the resulting set:

$$T = \max\left(\overbrace{T_{\dots}, \cdots, T_{\dots}}^{\text{overlapping}}, \overbrace{T_{\dots} + \cdots + T_{\dots}}^{\text{non-overlapping}}\right). \tag{4}$$

The following example will clarify the process: when discussing the model assumptions in Section 2.1, it was established that T_{comp} and T_{RegL1} overlap on all processors. Let us further assume that the architecture under consideration has a multi-ported L1 cache, which enables the cache to simultaneously communicate with the register file and the L2 cache. Assuming no overlap of other transfers, the runtime estimate for an in-memory data set on this processor would be $T = \max(T_{\text{comp}}, T_{\text{RegL1}}, T_{\text{L1L2}}, T_{\text{L2L3}} + T_{\text{L3Mem}})$.

The runtime estimate T can be converted into a performance estimate P by dividing the amount of work W carried out in one loop iteration by the runtime estimate for the same, and multiplying the result with the core frequency: $P = f_{\text{core}} \cdot W/T$.

For our investigations f_{core} was fixed, so converting from runtime to performance estimates is trivial. In practice, however, f_{core} is often set dynamically on the authority of the operating system, the processor, or even the user. However, f_{core} is virtually constant during the execution of a particular steady-state loop. This is because the metric used by the underlying mechanism (e.g., DVFS) to select f_{core} does not change while the processor is in a steady state. For a particular kernel, f_{core} can thus be measured via hardware performance events. For each kernel of a multi-loop application, f_{core} value must be determined individually. See [13] for an investigation of the model's ability to deal with different core and Uncore frequencies.

2.4.4. Multicore prediction based on single-core estimate

Multicore estimates require as inputs the single-core runtime estimate T, and the time the memory interface is busy transferring data $T_{\text{Mem}}^{\text{data}}$, which is the sum of all data-link busy times that involve the main memory (e.g., in a memory hierarchy with a victim L3 cache, where memory sends data to L2 and receives modified CLs from L3, $T_{\text{Mem}}^{\text{data}} = T_{\text{L2Mem}}^{\text{data}} + T_{\text{L3Mem}}^{\text{data}}$).

In the absence of shared resources (e.g., if the entire data set fits into core-private or scalable[6] shared caches), single-core performance P is expected to scale linearly with the number of active cores n, so the multicore estimate for n active cores is just $P(n) = nP$. If shared resources, such as the main memory interface, are involved, resource conflicts and the resulting waiting times must be considered. Here we employ a statistical model that is motivated by first principles: the utilization of the memory bus u is the probability of another core encountering a busy bus. For a single core, the utilization is given by the ratio of the time the memory interface is busy transferring data and the overall runtime estimate: $u(1) = T_{\text{Mem}}^{\text{data}}/T$. If multiple cores are active, the utilization is expressed recursively:

$$u(n) = \min\left(1, \frac{nT_{\text{Mem}}}{\max(T_{\text{comp}}, \ldots, T_{\text{Mem}} + \underbrace{u(n-1)(n-1)p_0}_{T_{\text{conf}}})}\right). \tag{5}$$

In the numerator, the memory-bus busy time is multiplied with the number of active cores, n, since multiple cores are using the memory interface. The denominator is the expanded expression for the runtime estimate, T, where a conflict time has been added to the data-transfer time involving the memory interface. This conflict time represents the average time that a core encountering a busy memory bus has to wait for the bus to become available to it. The conflict time encountered in a scenario with n active cores is given by multiplying the probability of a core hitting a busy memory bus, which corresponds to the memory utilization of the remaining cores, $u(n-1)$, with the time the other $n-1$ cores are using the interface. This results in $T_{\text{conf}} = u(n-1)(n-1)p_0$, with p_0 being an empirical fit parameter[7].

For performance estimates, the memory-bus utilization is multiplied with the performance to be expected with fully saturated bandwidth: $P(n) = u(n)P^{\text{Sat}}$. The memory-saturation performance, P^{Sat}, corresponds to the bandwidth limitation of the Roofline model and is determined by dividing the amount of work per loop iteration by the memory-bus busy time, and multiplying the result with the core frequency: $P^{\text{Sat}} = W/T_{\text{Mem}} \cdot f_{\text{core}}$.

3. Machine Model Construction

3.1. Method to Determine Machine Models

In the ideal case, all of the data required for a machine model would be available in vendor data sheets. In practice, however, this is rarely the case because important information is

[6]Scalable means a parallel efficiency close to one for all relevant degrees of parallelization (i.e., up the maximum number of cores sharing the cache).

[7]Although p_0 can also be modeled analytically employing the data used to derive T_{Mem}, we find that the level of detail required to reliably estimate the parameter outweighs the benefits of using an analytical approach.

deemed irrelevant or, more likely, intellectual property and therefore omitted from specifications. Moreover, the interaction of different parts of the processor might lead to situations in which vendor-specified numbers are not attainable (see, e.g., the discussion on load/store throughput in Section 2.3). In the following, a method is presented that allows to establish machine models in cases where relevant information is missing, or the documented specifications turn out to be impractical for some reason.

3.1.1. Instruction throughput and latency

At fixed core clock speed f_{core}, the time t it takes the core to execute a large number n of independent[8] instructions of type i is measured. The throughput of the instruction is then $\omega_i = n/(tf_{\text{core}})$. Since we will usually use a work unit of one (high-level) loop iteration in the modeling procedure, the instruction throughput is multiplied by the appropriate SIMD width w_{SIMD} to get the *operation throughput*:

$$\tau_i = w_{\text{SIMD}} \times \omega_i = w_{\text{SIMD}} \times n/(tf_{\text{core}}). \tag{6}$$

To measure latency, an artificial data-dependency chain is introduced by making each instruction use the output of the previous instruction as its input. This forces each new instructions to be held at a reservation station until the previous instruction has completed. The holding time, $\Lambda = n/(tf_{\text{core}})$, corresponds to the instruction's latency. The measured instruction latency is divided by the appropriate SIMD width to get the *operation latency*:

$$\lambda_i = \Lambda_{\text{i}}/w_{\text{SIMD}} = n/(tf_{\text{core}}w_{\text{SIMD}}). \tag{7}$$

While implementing these two strategies sounds simple in theory, deriving a suitable instruction mix from a high-level language implementation can be difficult in practice because compiler optimizations get in the way. We solve this problem by side-stepping the compiler and hand-crafting the necessary code in assembly language. To automate the process of determining latencies and throughputs, the `ibench` tool [11] was developed, which comprises a measurement framework and a number of assembly-code files for the most widespread instructions of AMD, IBM, ARM, and Intel processors.

3.1.2. Topology and data flow in the memory hierarchy

Information about the topology of the memory hierarchy, such as the number of caches, their sizes and properties (write-back vs. -through, victim vs. non-victim) are often well documented in vendor data sheets. Even if this is not the case, the data is easy to obtain, for most processors provide access to it over a well-defined interface. In case of x86, for instance, the `cpuid` instruction can be used to extract detailed information about the memory hierarchy, including the capacity, associativity, number of sets, inclusiveness, cache-line size, and more for each level in the hierarchy. Other processors offer similar mechanisms, and the Linux `sysfs` file system provides an architecture-independent interface to obtain the necessary data.

Information about data flow (i.e., the path data takes from a particular level in the memory hierarchy to reach a core's L1 cache) can be derived from the topology information. In most cases, only stores require special attention to determine whether store-misses trigger a write-allocate

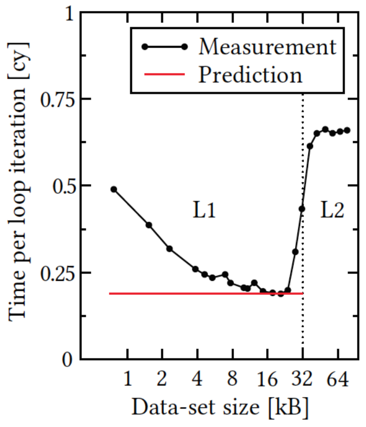
[8]Independent means that there are no data dependencies between the different instances of the instruction.

for the missed CL or if some optimization (as the one implemented in Marvell's ThunderX2) detects whether a full CL is written to avoid the write-allocate. Such details can be derived with the help of hardware performance events, which can be used to record the data volumes exchanged between different levels of the memory hierarchy.

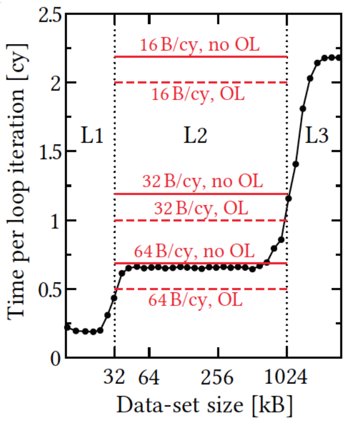
3.1.3. Bandwidth, latency, and overlap in the memory hierarchy

Typically, cache bandwidths are well-documented by vendors. In some cases, however, vendors only specify bandwidths for selected caches. In these instances, cache bandwidths can be determined by selecting a set of reasonable bandwidth candidates (e.g., 16, 32, and 64 B/cy), and examining which of the corresponding estimates best agrees with empirical data. To the best of our knowledge, no vendor publishes data on the overlap properties of their processors' memory hierarchies, so this data needs to be determined in a similar way.

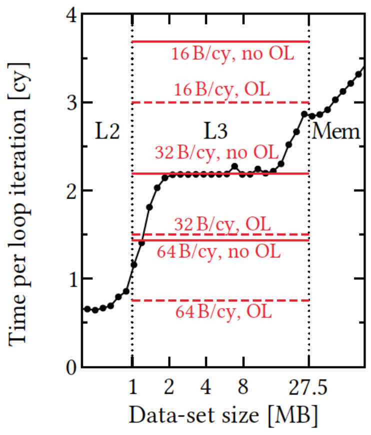
The process of comparing estimates to empirical data is iterative: once the bandwidth and overlap properties for a particular memory level have been established, the numbers can be used as input for different bandwidth and overlap assumptions in the next memory level. In the following, the process is demonstrated on the SKL processor for the well-known STREAM triad [17].



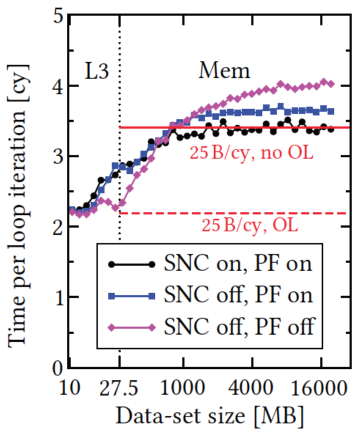
(a) L1



(b) L2



(c) L3



(d) Main memory

Figure 4. Comparison of model estimates to empirical data for the STREAM triad on SKL for data sets in (a) L1, (b) L2, and (c) L3 caches, and (d) main memory

On the SKL processor, one loop iteration of the STREAM triad (`A[i]=B[i]+s*C[i]`) comprises two loads (one from each of the input arrays `B` and `C`), one fused multiply-add (FMA) (to calculate the result), and one store (to write the result to the output array `A`). Using `ibench`, the following operation throughputs were established: $\tau_{\mathrm{FMA}} = 16/\mathrm{cy}$, $\tau_{\mathrm{LD}} = 16/\mathrm{cy}$, $\tau_{\mathrm{ST}} = 8/\mathrm{cy}$, and $\tau_{\mathrm{LD/ST}} = 16/\mathrm{cy}$. According to Equations (1), (2), and (4), for a data set in the L1 cache this leads to a single-iteration runtime estimate of

$$\begin{aligned} T_{\mathrm{L1}} &= \max\left(\overbrace{\frac{1\,\textsc{fma}/\mathrm{it}}{16\,\textsc{fma}/\mathrm{cy}}}^{T_{\mathrm{comp}}}, \overbrace{\frac{2\,\textsc{ld}/\mathrm{it}}{16\,\textsc{ld}/\mathrm{cy}}, \frac{1\,\textsc{st}/\mathrm{it}}{8\,\textsc{st}/\mathrm{cy}}, \frac{3\,\textsc{ld}/\textsc{st}/\mathrm{it}}{16\,\textsc{ld}/\textsc{st}/\mathrm{cy}}}^{T_{\mathrm{RegL1}}}\right) \\ &\approx 0.19\,\mathrm{cy/it}. \end{aligned}$$

In Fig. 4a we compare this prediction to measurements. Note that the estimate corresponds to the lower limit of runtime, which is actually attained by the running code if the loop is long enough.

If the data set resides in the L2 cache, a total of 32 B are transferred between the L1 and L2 caches per iteration: 8 B for each of the double-precision floating-point numbers from the input arrays `B` and `C`, 8 B for the write-allocate to `A`, and 8 B for evicting the updated element of `A` to the L2 cache. Bandwidth assumptions of 16, 32, and 64 B/cy yield estimates for T_{L1L2} of two, one, and one-half cycle, respectively. Figure 4b compares the estimates to empirical data. The assumptions of no overlap and a bandwidth of 64 B/cy match the measurements strikingly well; incidentally, the L1-L2 cache bandwidth as advertised by Intel is also 64 B/cy. With L1-L2 cache bandwidth and overlap properties established, we can move on to the L3 cache. The data exchanged between the L2 and L3 caches is 48 B because each of the three eight-byte reads from L3 (two from the input arrays `B` and `C`, one write-allocate from the target array `A`) triggers the eviction of data replaced in the L2 cache to the victim L3. L2-L3 bandwidth assumptions of 16, 32, and 64 B/cy yield estimates for T_{L2L3} of 3, 1.5, and 0.75 cy, respectively. Figure 4c compares estimates derived from the different bandwidth and overlap assumptions to empirical data for a data set in the L3 cache. In this case we find that that assumptions of no overlap and a bandwidth of 32 B/cy agree very well with the measurement. Finally, for in-memory data sets, only different overlap assumptions must be made, since the sustained memory bandwidth is determined by measurement (55 GB/s for one SNC domain, which for $f_{\mathrm{core}} = 2.2\,\mathrm{GHz}$ is 25 B/cy). Figure 4d compares the resulting estimates to empirical data (black line) and we find that in memory, too, no overlap of data transfers occurs.

In addition to runtime measurements obtained with SNC mode and the LLC prefetcher (PF) enabled, Fig. 4d also shows data where these features were disabled. This is to demonstrate that in some settings, bandwidth and overlap are not sufficient to describe the empirical behavior in a satisfying manner. Then, a latency penalty must be added to data transfer times (see Section 2.1).

3.2. Results for Investigated Processors

Table 2 shows the machine models that result from applying the previously introduced method to the processors from the testbed.

The upper part of the table lists relevant operation throughput (τ) and instruction latency (λ) values. The center part lists bandwidths and latency penalties (if applicable) in the memory hierarchy. Note that in cases where two numbers are provided (e.g., 64+16 B/cy for PWR9's L1-

Table 2. Machine models determined for the investigated processors

Microarchitecture	Skylake-SP (SKL)	Zen (EPYC)	Vulcan (TX2)	POWER9 (PWR9)
τ_{ADD}, τ_{MUL}, τ_{FMA} [/cy]	16, 16, 16	4, 4, 4	4, 4, 4	4, 4, 4
τ_{LD}, τ_{ST}, $\tau_{\mathrm{LD/ST}}$ [/cy]	16, 8, 16	4, 2, 4	4, 2, 4	4, 4, 4
λ_{ADD}, λ_{MUL}, λ_{FMA}	0.5, 0.5, 0.5	1.5, 2, 2.5	3, 3, 3	3, 3, 3
$b_{\mathrm{L1}\leftrightarrow\mathrm{L2}}$	64 B/cy	32+32 B/cy	64 B/cy	64+16 B/cy
$b_{\mathrm{L2}\leftrightarrow\mathrm{L3}}$	32 B/cy	32 B/cy	32 B/cy	32 B/cy
$b_{*\leftrightarrow\mathrm{Mem}}$	25–28 B/cy	13–16 B/cy	47–56 B/cy	41–45 B/cy
Data-transfer penalties	—	—	—	$T^{\mathrm{p}}_{\mathrm{Mem}} = 0.04\,\mathrm{cy/B}$
Non-overlapping transfers	all	L2-L3, L2-Mem, L3-Mem	all (if Mem involved)	L2-Mem, L3-Mem
Write-through/ victim caches	Victim L3	Victim L3	Victim L3	Write-through L1, Victim L3

L2 bandwidth), two uni-directional data paths exist between the caches. In such instances, the first number corresponds to the bandwidth of sending data from the underlying to the overlying cache, and second number to the bandwidth in the opposite direction. Note that listed memory bandwidth corresponds to that of a single NUMA node (SNC node on SKL, Zeppelin on EPYC, full-chip on TX2 and PWR9). Memory bandwidths are specified as ranges, since different data-access patterns exhibit slightly varying sustained memory bandwidths. The last part of the table contains overlap capabilities and additional information on cache types.

4. Case Study: PCG

We use a matrix-free PCG solver to demonstrate the viability of our approach in real-world scenarios. The solver is preconditioned using the well-known symmetric Gauss-Seidel iteration and relies on the second-order finite-difference method for discretization. We use it to solve the steady-state heat equation in 2D. The sparse matrix entries are not stored explicitly but hard-coded into a 2D five-point stencil representation. Hence, the solver is similar to the well-known HPCG but shows a more interesting phenomenology: as opposed to HPCG, where all loops are limited by data transfers due to explicit matrix storage, our preconditioner is bound by in-core pipeline hazards. All computations and data storage are in double precision.

Algorithm 1 shows the entire PCG method. It is composed of a matrix-free sparse-matrix-vector multiplication (SpMVM) which we refer to as STENCIL, a symmetric Gauss-Seidel preconditioner (GS), and three BLAS-1 routines: DOT product, vector NORM, and DAXPBY. The code is implemented in C++ and parallelized with OpenMP. The Gauss-Seidel kernels, which have loop-carried dependencies, are parallelized using a well-known wavefront technique that preserves the numerical behavior of the serial code [7]. The preconditioner can be vectorized by, e.g., coloring methods, but this would alter the convergence and render the loops data bound, which is not the scenario we want to showcase (see above).

Algorithm 1 PCG algorithm: solve for $x : Ax = b$

```
1: $r = b - Ax$
2: $r_{\text{norm}} = \langle r, r \rangle$
3: $p = z = Pr$
4: $\alpha_0 = \langle r, z \rangle$
5: $i = 0$
6: **while** $(i < n_{\text{iter}})$ && $(r_{\text{norm}} > \varepsilon^2)$ **do**
7:     $v = Ap$                                    STENCIL operation (SpMVM)
8:     $\lambda = \frac{\alpha_0}{\langle v,p \rangle}$                    DOT
9:     $x = x + \lambda p$                          DAXPBY
10:    $r = r - \lambda v$                          DAXPBY
11:    $r_{\text{norm}} = \langle r, r \rangle$     NORM
12:    $z = Pr$                                     GS preconditioner
13:    $\alpha_1 = \langle r, z \rangle$            DOT
14:    $p = z + \frac{\alpha_1}{\alpha_0} p$        DAXPBY
15:    $\alpha_0 = \alpha_1$
16:    $i = i + 1$
```

Algorithm 2 High-level representation of STENCIL

```
1: **for** $j = 1 : n_j - 1$ **do**
2:     **for** $i = 1 : n_i - 1$ **do**
3:         $v_{j,i} = w_c p_{j,i} + w_y(p_{j-1,i} + p_{j+1,i}) + w_x(p_{j,i-1} + p_{j,i+1})$
```

4.1. Application Models

The total problem size ($n_i \times n_j$) was chosen to be $n_i = 25000$ (inner, leading dimension) and $n_j = 2000$ (outer dimension), so that all arrays reside in main memory. In the following, application models for all of the PCG components are presented.

Features important for the considered example include the number of loads and stores, floating-point operations, and loop structures. For simple streaming loops, all of these details can be derived from high-level code. The DAXPBY kernel (`y[i]=a*x[i]+b*y[i]`) entails two loads, one FMA, one multiplication, and one store. The DOT product (`d+=x[i]*y[i]`) and NORM (`n+=x[i]*x[i]`) have two and one load(s), respectively, along with an FMA. These kernels can be fully and effectively vectorized by all compilers.

For kernels with cache reuse such as STENCIL and GS, reuse-distance analysis (best done using the layer condition [3, 22]), blocking factors, parallelization strategies, and scheduling techniques have to be taken into account. The STENCIL kernel is shown in Algorithm 2, with w_* representing different weights obtained from the matrix A. The kernel requires two FMAs, two additions, one multiplication, one store, and five load operations. SIMD vectorization is straightforward, but in contrast to the BLAS kernels, different loads can hit different memory hierarchy levels depending on the reuse distance. For the considered inner dimension of $n_i = 25000$ and outer (j) loop parallelization employed in our code, the layer condition would require $4n_i$ elements per thread to fit in a cache. The lowest (i.e., outermost) cache that satisfies this criterion will only have a miss for one of the four elements on the right-hand side, while the cache levels above it will have three. On all processors under investigation, the layer condition is satisfied in the last-level cache (LLC). Changing the inner problem dimension would certainly change

Algorithm 3 High-level representation of GS forward sweep

```
1: for $j = 1 : n_j - 1$ do
2:     for $i = 1 : n_i - 1$ do
3:         $z_{j,i} = w_c(r_{j,i} + w_y z_{j-1,i} + w_x z_{j,i-1})$
```

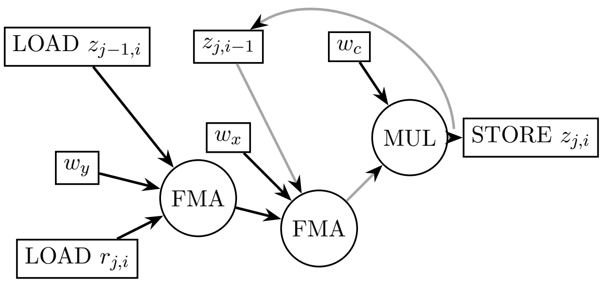


Figure 5. Dependency chain of the GS forward kernel when using the Intel compiler. Critical path shown in gray

the prediction; the ECM model has been demonstrated to yield accurate results in all these cases [14, 22], so we restrict ourselves to a single size only here. Storing to v implies a write-allocate through the whole memory hierarchy on all processors, and, at some point, the writing back of the newly-computed data to memory.

The GS kernel is a symmetric operator comprising a forward and a backward sweep. The forward sweep (GS$_\text{F}$) is shown in Algorithm 3, and requires two FMAs, one multiplication, one store, and three load operations. The kernel is similar to STENCIL, but it reads from $z_{j,i-1}$ and writes to $z_{j,i}$, causing a loop-carried dependency. A wavefront technique can be used to parallelize the kernel [7], and the corresponding layer-condition criterion requires $3n_i$ elements to fit in a cache. The outermost cache that satisfies this condition will have only two load misses on the right-hand side, while the others would have three. The GS backward sweep (not shown here for brevity) is similar, but loops are traversed in reverse direction and $w_c r_{j,i}$ in GS$_\text{F}$ is replaced with $z_{j,i}$. The analysis of the kernel follows the same approach, but there is one less load miss.

Both GS loops have loop-carried dependencies, preventing SIMD vectorization. As a result, a critical path analysis is required. In GS$_\text{F}$ the element $z_{j,i}$ written in a particular iteration is read in the next as $z_{j,i-1}$. The actual delay caused by this dependency can vary depending on the code generated by the compiler. Figure 5 shows the result when using the Intel compiler and the critical path of the generated instruction mix includes one FMA and one multiplication. The ARM clang compiler produces code that does not keep $z_{j,i}$ in a register across loop iterations, leading to an extra delay caused by storing and loading the element. Due to its particular unrolling strategy, IBM's XLC compiler generates a combination of the two previous variants.

4.2. Runtime Predictions

In the following, the proposed model is validated by comparing the model estimates to empirical performance for the DOT product, DAXPBY and GS$_\text{F}$ kernels, as well as the full PCG algorithm. The DOT kernel is also used to exemplify how the simultaneous multi-threading (SMT) feature of modern processors can be incorporated in the model. Note that estimates correspond to the runtime of a single high-level (i.e., scalar) loop iteration.

4.2.1. Simultaneous multi-threading on SKL

As discussed in Section 4.1, one loop iteration of the DOT product entails two loads and one FMA. According to the machine model documented in Tab. 2, the SKL processor can perform 16 loads and 16 FMAs per cycle for AVX-512 code. Combining application and machine models according to Eq. (1) yields a contribution of $T_{\text{RegL1}} = n_{\text{LD}}/\tau_{\text{LD}} = 2/16\,\text{cy} = 0.125\,\text{cy}$ for data transfers between the register file and the L1 cache. With respect to computational cycles, it is worth pointing out that the kernel contains a loop-carried dependency. Each FMA uses as one of its inputs the result of the previous FMA. Without modulo-variable expansion (MVE) and SMT, the impact of the dependency corresponds to the FMA latency, so $T_{\text{dep}} = \lambda_{\text{FMA}} = 0.5\,\text{cy}$. According to Eq. (2), computational cycles therefore amount to $T_{\text{comp}} = \max(n_{\text{FMA}}/\tau_{\text{FMA}}, T_{\text{dep}}) = \max(1/16\,\text{cy}, 0.5\,\text{cy}) = 0.5\,\text{cy}$. Since both contributions can overlap, the runtime estimate for a data set in the L1 cache according to Eq. (4) is $T = \max(T_{\text{RegL1}}, T_{\text{comp}}) = 0.5\,\text{cy}$.

Using either two-way unrolling with MVE or 2-SMT, the impact of the dependency is cut in half, so $T_{\text{dep}} = 0.25\,\text{cy}$. The overall runtime estimate in this case becomes $T = 0.25\,\text{cy}$.

The combination of two-way unrolling with MVE and 2-SMT again halves the impact of the dependency, so $T_{\text{dep}} = 0.125\,\text{cy}$, and the overall runtime becomes $T = 0.125\,\text{cy}$. Note that at this point, the contribution of the loop-carried dependency is identical to $T_{\text{RegL1}} = 0.125\,\text{cy}$. This means that additional unrolling will no longer effect a reduction in runtime since runtime is now limited by data transfers between the register file and the L1 cache. Note as well that reducing T_{dep} to 0.125 cy can also be achieved without SMT by applying four-way unrolling with MVE to the code. In fact, it is possible to run most loop-based streaming codes at the lower runtime limit without SMT if the executed instruction mix is optimized appropriately and sufficient physical registers are available.

Table 3. Comparison of model estimates to empirical data (in cycles per loop iteration) for the DOT product on the SKL CPU for data-set sizes of 25 kB (L1), 127 kB (L2), 9772 kB (L3), and 1022 MB (Mem) as function on the degree of simultaneous multi-threading (SMT) and unrolling with modulo-variable expansion (MVE)

Degree of		Model estimate for				Measurement for			
SMT	MVE	L1	L2	L3	Mem	L1	L2	L3	Mem
1	1	0.500	0.500	1.375	1.975	0.501	0.500	1.411	2.096
1	2	0.250	0.375	1.375	1.975	0.250	0.379	1.411	2.085
2	1	0.250	0.375	1.375	1.975	0.250	0.359	1.411	2.028
2	2	0.125	0.375	1.375	1.975	0.136	0.360	1.411	2.030
1	4	0.125	0.375	1.375	1.975	0.136	0.376	1.413	2.090
2	4	0.125	0.375	1.375	1.975	0.136	0.364	1.411	2.029

Table 3 summarizes the estimates discussed above, as well as estimates for the remaining levels of the SKL processor's memory hierarchy. As predicted, no unrolling and SMT results in a runtime of 0.5 cy per loop iteration for a data set in the L1. Moreover, either two-way unrolling or SMT results in a halving of the runtime to 0.25 cy. Combining both optimizations further reduces the runtime by a factor of two to 0.125 cy. The data shows that the same result can be achieved without SMT when applying four-way unrolling. Furthermore, the data in the last row supports the model prediction that additional unrolling (or SMT, if the core supported it) would not lead to further reductions in runtime since at this point T_{RegL1} dominate.

When the data set resides in the L2 cache, 16 bytes (8 bytes for each of the double-precision floating-point numbers from the two input arrays) must be transferred between the L1 and L2 caches per loop iteration. The machine model (see Tab. 2) lists a L1-L2 bandwidth of 64 B/cy for the SKL processor, so the data-transfer time is $T_{\text{L1L2}} = 0.25\,\text{cy}$. All data transfers are non-overlapping, so the runtime estimate according to Eq. (4) becomes $T = \max(T_{\text{comp}}, T_{\text{RegL1}} + T_{\text{L1L2}})$ with an aggregated transfer time of $T_{\text{RegL1}} + T_{\text{L1L2}} = 0.125\,\text{cy} + 0.25\,\text{cy} = 0.375\,\text{cy}$. For the version without unrolling and SMT, $T_{\text{comp}} = 0.5\,\text{cy}$ is higher than the combined contribution of the runtime, resulting in an overall runtime of $T = \max(0.5\,\text{cy}, 0.375\,\text{cy}) = 0.5\,\text{cy}$. For all other versions, the overall runtime is dominated by the combined data-transfer time, so $T = \max(T_{\text{comp}}, 0.375\,\text{cy}) = 0.375\,\text{cy}$.

With data in the L3 cache, 16 bytes are to sent from the L3 to the L2 cache in each loop iteration. At the same time, 16 bytes are preempted from the L2 cache into the victim L3 cache. The total amount of data transferred is therefore 32 bytes, which takes 1 cy according to the documented bandwidth of 32 B/cy (cf. Tab. 2). Considering that data transfers are non-overlapping, the overall runtime estimate becomes $T = \max(T_{\text{comp}}, T_{\text{RegL1}} + T_{\text{L1L2}} + T_{\text{L2L3}})$. According to the model, the runtime of all variants is dominated by the contribution of data transfers of $T_{\text{RegL1}} + T_{\text{L1L2}} + T_{\text{L2L3}} = 0.125\,\text{cy} + 0.25\,\text{cy} + 1\,\text{cy} = 1.375\,\text{cy}$. Consequently, the runtime estimate for all variants is $T = \max(T_{\text{comp}}, 1.375\,\text{cy}) = 1.375\,\text{cy}$.

Finally, in the case of input data residing in main memory, 16 bytes have to be sent from memory to the L3 cache. The bandwidth of about 26.5 B/cy documented in the machine model (cf., again, Tab. 2) implies a contribution of $T_{\text{L3Mem}} \approx 0.6\,\text{cy}$. As before, the non-overlapping data-transfer contributions dominate the overall runtime for all versions, resulting in an estimate of $T = \max(T_{\text{comp}}, 0.125\,\text{cy} + 0.25\,\text{cy} + 1\,\text{cy} + 0.6\,\text{cy}) = 1.975\,\text{cy}$.

4.2.2. Single-core

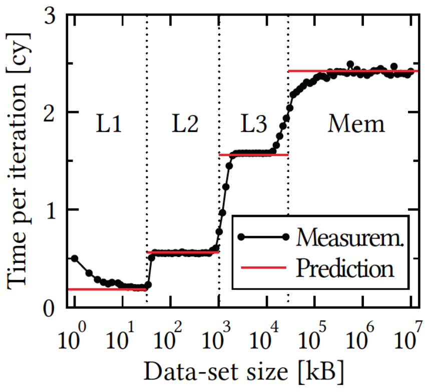
On the SKL processor, retiring the DAXPBY kernel's multiplication and FMA operations takes $T_{\text{comp}} \approx 0.0625\,\text{cy}$. The one store and two load operations take $T_{\text{RegL1}} \approx 0.1875\,\text{cy}$. Per-iteration data-transfer volumes are 24 B between the L1 and L2 caches (one load each from `x` and `y`, one write to `y`), 32 B between the L2 and L3 caches (one load each from `x` and `y`, and two corresponding evicts since the L3 is a victim cache), and 24 B between L3 and main memory (see L1-L2 transfers). Using the bandwidths documented in the machine model, this results in contributions of $T_{\text{L1L2}} = 0.375\,\text{cy}$, and $T_{\text{L2L3}} = 1\,\text{cy}$. For the measured memory bandwidth of 60 GB/s, which for $f_{\text{core}} = 2.2\,\text{GHz}$ corresponds to a bandwidth of 27.3 B/cy, T_{L3Mem} is 0.88 cy. Since all data transfers are non-overlapping, the runtime estimates are $T_{\text{L1}} = 0.1875\,\text{cy}$, $T_{\text{L2}} = 0.5625\,\text{cy}$, $T_{\text{L3}} = 1.5625\,\text{cy}$, and $T_{\text{Mem}} = 2.4425\,\text{cy}$.

Intermediate and final single-core estimates for DAXPY on SKL, and all other processors, are given in Tab. 4. Cases where data volumes change in the victim L3 cache (depending on whether the input data resides in the L3 or main memory) are indicated by listing two numbers in the table, the former corresponding to the data-transfer time estimate for data in the L3, the latter for data in memory.

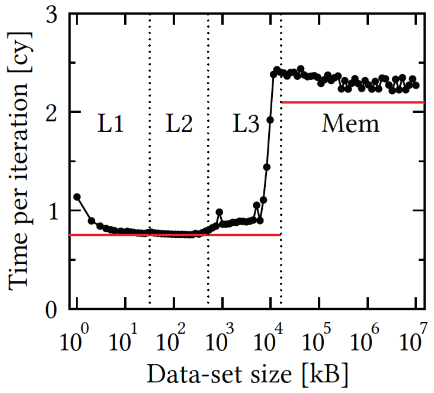
These single-core estimates are compared to empirical data in Fig. 6. The data indicates that the model manages to describe empirical performance on all investigated processors with high accuracy.

Table 4. Single-core estimates for DAXPY on all investigated processors

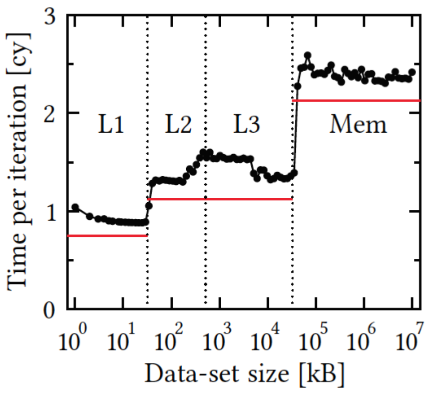
CPU	SKL	EPYC	TX2	PWR9
T_{comp} [cy/it]	0.0625	0.25	0.25	0.25
T_{RegL1} [cy/it]	0.1875	0.75	0.75	0.75
T_{L1L2} [cy/it]	0.375	0.5	0.375	0.5
T_{L2L3} [cy/it]	1	0.75 | 0.25	1 | 0.5	1 | 0.5
T_{L2Mem} [cy/it]	—	1.23	0.29	0.36
T_{L3Mem} [cy/it]	0.88	0.62	0.14	0.18
T_{L1} [cy/it]	0.1875	0.75	0.75	1.25
T_{L2} [cy/it]	0.5625	0.75	1.125	1.25
T_{L3} [cy/it]	1.5625	0.75	1.125	1.25
T_{Mem} [cy/it]	2.4425	2.1	2.06	2.1



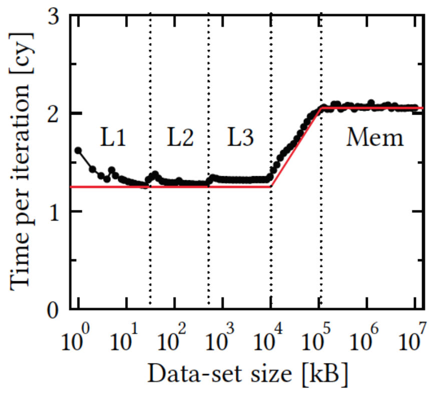
(a) Intel Skylake-SP



(b) AMD Epyc



(c) Cavium ThunderX2

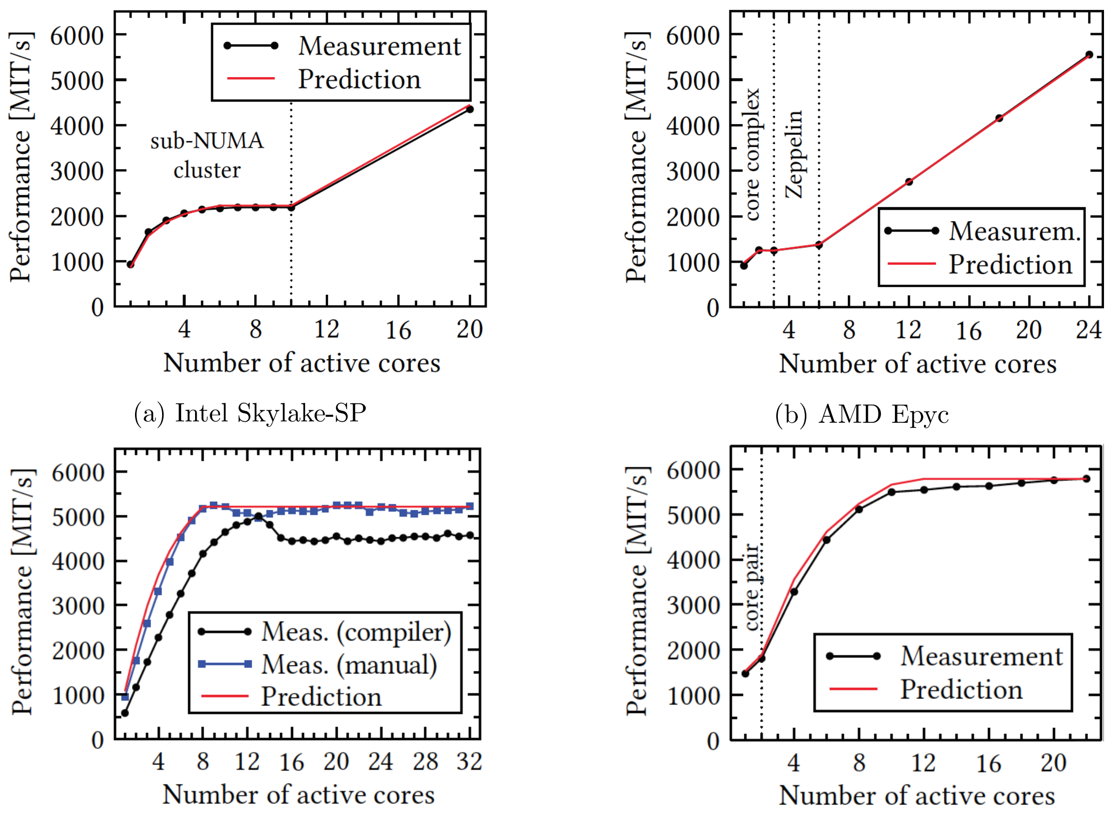


(d) IBM Power9

Figure 6. Comparison of model estimates to empirical data for DAXPY on (a) SKL, (b) EPYC, (c) TX2, and (d) PWR9

4.2.3. Multicore

The DAXPBY and GS$_{\mathrm{F}}$ kernels were selected to investigate the model's capability to accurately describe multicore performance. Being a data-bound streaming kernel, DAXPBY proves particularly suitable to investigate the memory subsystem of the investigated processors and



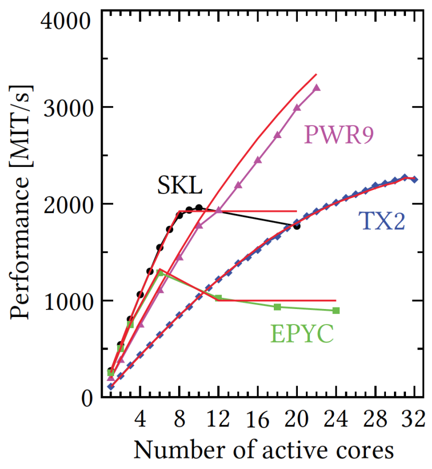
(a) Intel Skylake-SP (b) AMD Epyc (c) Cavium ThunderX2 (d) IBM Power9

Figure 7. Comparison of performance models to empirical data for intra-socket scaling of DAXPBY on (a) SKL, (b) EPYC, (c) TX2, and (d) PWR9. Performance is given in 10^6 iterations per second (MIT/s)

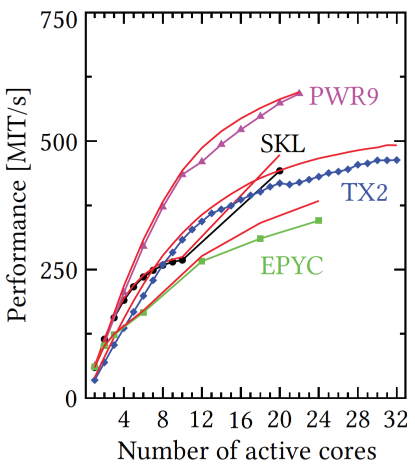
their scaling behavior. GS_F, on the other hand, is core bound for all architectures when executed on a single core. However, when increasing the number of cores, NUMA properties turn out to have a significant impact on performance.

Figure 7 shows the multicore scaling of DAXPBY on all architectures up to a full socket using "close" thread affinity (i.e., filling cores consecutively through ccNUMA domains). For SKL we observe the typical saturation behavior (at $\approx 2.2\,\text{GIT/s} = 53\,\text{GB/s}$) of bandwidth-bound code within a single SNC domain. Using the second SNC domain doubles the bandwidth and hence performance by a factor of two as predicted by the model. The scaling behavior of EPYC exposes its main hardware features: within a single CCX (three cores) the shared L3 bandwidth does not scale across the cores and hits a maximum of 32 B/cy. The best bandwidth attained on a single CCX is 30 GB/s compared to 33 GB/s for the entire ccNUMA domain (a "Zeppelin" die); we speculate that this is a faint echo of non-scalable L3 cache. Scaling across the Zeppelin dies is linear, as expected. On the TX2, we initially observed a significant deviation: The compiler-generated code (black line) fell short of the model by as much as 40 % for a single core and 10 % after saturation. The prompted investigation revealed that the TX2's hardware prefetchers have some deficiency: data was not prefetched in time, so runtime is subject to additional latency. The issue could be resolved by manually adding software prefetch instructions to the compiler-generated code to work around the flawed hardware prefetchers (blue line). This demonstrates how the model can be used to identify bottlenecks or other shortcomings that limit performance (in this case, the compiler). Note that the optimization is not part of our PCG code; we use the compiler versions for all further comparisons. On PWR9, the scaling within a core pair is similar to that observed within a CCX of EPYC. This is due to the shared and non-scalable L2 and L3 cache segments per core. The multicore model accommodates this behavior by keeping the L2 and L3 data-transfer rates constant for the two cores sharing the resources. Scaling across core pairs (i.e., running with 2, 4, 6, etc. cores) is only limited by bandwidth saturation as can be observed by the measurements and respective model prediction.

The GS_F kernel is latency bound due to the loop-carried dependency discussed in Section 4.1. There are two peculiarities that make predictions of the parallel GS_F kernel challenging: first, the wavefront parallelization requires a barrier synchronization after each inner loop traversal. For the chosen problem size, the corresponding OpenMP-barrier was found to cause non-negligible overhead. We addressed this by benchmarking the OpenMP barrier for all relevant compiler-hardware combinations and included the barrier time as additional overhead. Secondly, although parallel first-touch page placement works fine for all other loops, the parallel-wavefront algorithm accesses data in parallel across the inner dimension. Since data placement is done with static OpenMP scheduling across the outer dimension, this leads to all threads accessing the same ccNUMA domain most of the time during the GS sweeps. It turns out that this effect can be incorporated into the model as well. To this end, the sustained memory bandwidth is measured across all ccNUMA domains with data residing in only one domain. This data can then be used as a bandwidth limit when using multiple ccNUMA domains on SKL and EPYC. Figure 8a compares performance estimates to measurements for GS_F across the cores of a socket on all architectures. The deviation from the model is generally smaller than 10 % when using multiple NUMA domains, and below 5 % when looking at a single ccNUMA domain. The results indicate that the model with enhancements described above (barrier overhead, ccNUMA contention) delivers a good qualitative and quantitative description of the performance behavior.



(a) GS forward kernel



(b) The full PCG algorithm

Figure 8. Comparison of estimates to empirical data for (a) GS forward kernel and (b) the full PCG algorithm

4.2.4. Composition

With estimates for individual kernels in place we can now present multicore-scaling data for the full PCG algorithm. Composing the model from single-loop predictions is simple due to the time-based formulation of the ECM model [21]. In the case of PCG we have three invocations of DAXPBY, two of DOT, one GS forward- and backward-sweep each, as well as one of STENCIL. Figure 8b shows the comparison of the model with measurements for all four architectures. Again, the general model error is below 10 %, and less than 5 % when looking at single ccNUMA domains. The slightly larger deviation beyond 12 cores on TX2 can be attributed to the fact that we use compiler-generated code instead of hand-crafted assembly for the CG solver on this machine. The lack of prefetching causes a 10–15 % performance breakdown of data-bound loops beyond the saturation point (see Fig. 7c), which we ignore in the model. On EPYC and SKL we observe very low performance for OpenMP reductions across ccNUMA domains (much larger than the considered OpenMP barrier) with the Intel compiler, causing the slight deviation beyond one domain.

5. Related Work

There are two capable analytic (in the sense of "first principles") performance models for steady-state loop code on multicore CPUs: the Roofline model [10, 25] and the ECM model [6, 13, 22]. Both have been subject to intense study, refinements, and validation, and their areas of applicability are well understood. However, while there is ample data available for Roofline on a wide variety of architectures [15, 18], one drawback of previous applications of the ECM model [2, 4, 12, 21, 22, 24, 26] is that they were mostly restricted to Intel processors. We provide the first thorough cross-architecture study of the model.

The Roofline model has the attractive property that it can be easily separated into a machine part (memory and cache bandwidths, peak performance) and an application part (computational intensity). There is no previous work that has done the same with the ECM model. A comparison between Roofline and ECM for several stencil algorithms can be found in [22]. A drawback of the Roofline model is that it requires a large amount of phenomenological input such as measured bandwidths for all core counts and all memory hierarchy levels [15], while the ECM model only needs the saturated memory bandwidth and the machine model (i.e., overlap assumptions).

Advanced curve-fitting and machine-learning techniques combined with hardware performance monitoring data have been used in the past to model the performance of code [1, 19]. Although these approaches are useful in practical settings, e.g., for predicting program runtimes with a goal of optimized resource scheduling, the deepest insights are gained through first-principles models such as Roofline or ECM.

Conclusion

We have shown that it is possible to set up a well-defined workflow for modeling the serial and parallel runtime of steady-state (sequences of) loops with regular data access patterns using the analytic ECM performance model. One can, with minor exceptions, cleanly separate machine properties from application properties. Four multicore server processors were investigated, and we could demonstrate that despite their obvious differences the main properties needed to set up a useful machine model can be summarized in a few parameters. The performance, including scalability across cores and ccNUMA domains, of an OpenMP-parallel preconditioned CG solver with wavefront-parallel Gauss-Seidel sweeps could be described with a modeling error of 5 % or less in most cases. We have to emphasize that no other first-principles model is capable of delivering such predictions with comparable accuracy and generality.

We found the overlapping property of transfers across data paths in the cache hierarchy to be the pivotal architectural feature governing single-core performance for data-bound loops. A design with very strong in-core performance (e.g., via wide SIMD execution) but a non-overlapping memory hierarchy may well be inferior to a weak core with strong overlap, as our comparison of Skylake SP and AMD Epyc shows. The architecture with the lowest in-core computational performance, Power9, came out first in serial and parallel memory-bound performance. The Cavium ThunderX2 processor can compensate its rather low in-core performance with good memory bandwidth and a large core count.

All modeling procedures carried out in this paper were done by hand. Some components, e.g., the construction of a runtime prediction from code and a (given) machine model, can be supported by tools [9]; others, such as the derivation of overlapping properties, would be very hard to automate. However, the purpose of performance modeling is not just prediction but also insight, and manual analysis sharpens the view on the relevant details.

Acknowledgements

We thank Thomas Gruber for helping to port the likwid tool suite to IBM's Power9 architecture.

We also thank the Center for Information Services and High Performance Computing (ZIH) at TU Dresden for providing access to their Power9 cluster.

References

1. Alam, S.R., Bhatia, N., Vetter, J.S.: An exploration of performance attributes for symbolic modeling of emerging processing devices. In: Perrott, R., Chapman, B.M., Subhlok, J., de Mello, R.F., Yang, L.T. (eds.) High Performance Computing and Communications. Lecture Notes in Computer Science, vol. 4782, pp. 683–694. Springer, Berlin, Heidelberg (2007), DOI: 10.1007/978-3-540-75444-2_64

2. Cremonesi, F., Hager, G., Wellein, G., Schrmann, F.: Analytic performance modeling and analysis of detailed neuron simulations. The International Journal of High Performance Computing Applications 34(4), 428–449 (2020), DOI: 10.1177/1094342020912528

3. Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Optimization and performance modeling of stencil computations on modern microprocessors. SIAM Review 51(1), 129–159 (2009), DOI: 10.1137/070693199

4. Gmeiner, B., Rüde, U., Stengel, H., Waluga, C., Wohlmuth, B.: Performance and scalability of hierarchical hybrid multigrid solvers for Stokes systems. SIAM Journal on Scientific Computing 37(2), C143–C168 (2015), DOI: 10.1137/130941353

5. Gruber, T., et al.: LIKWID performance tools (2019), `http://tiny.cc/LIKWID`

6. Hager, G., Treibig, J., Habich, J., Wellein, G.: Exploring performance and power properties of modern multicore chips via simple machine models. Concurrency Computat.: Pract. Exper. 28(2), 189–210 (2013), DOI: 10.1002/cpe.3180
7. Hager, G., Wellein, G.: Introduction to High Performance Computing for Scientists and Engineers. CRC Press, Inc., Boca Raton, FL, USA, 1st edn. (2010)
8. Hammer, J.: pycachesim – Python Cache Hierarchy Simulator (2019), `https://github.com/RRZE-HPC/pycachesim`
9. Hammer, J., Eitzinger, J., Hager, G., Wellein, G.: Kerncraft: A tool for analytic performance modeling of loop kernels. In: Niethammer, C., Gracia, J., Hilbrich, T., Knüpfer, A., Resch, M.M., Nagel, W.E. (eds.) Tools for High Performance Computing 2016: Proceedings of the 10th International Workshop on Parallel Tools for High Performance Computing, October 2016, Stuttgart, Germany. pp. 1–22. Springer International Publishing, Cham (2017), DOI: 10.1007/978-3-319-56702-0_1
10. Hockney, R.W., Curington, I.J.: $f_{1/2}$: A parameter to characterize memory and communication bottlenecks. Parallel Computing 10(3), 277–286 (1989), DOI: 10.1016/0167-8191(89)90100-2
11. Hofmann, J.: ibench – measure instruction latency and throughput (2019), `https://github.com/hofm/ibench`
12. Hofmann, J., Fey, D.: An ECM-based energy-efficiency optimization approach for bandwidth-limited streaming kernels on recent Intel Xeon processors. In: Proceedings of the 4th International Workshop on Energy Efficient Supercomputing, 14 Nov. 2016, Salt Lake City, UT, USA. pp. 31–38. IEEE Press, Piscataway, NJ, USA (2016), DOI: 10.1109/E2SC.2016.010
13. Hofmann, J., Hager, G., Fey, D.: On the accuracy and usefulness of analytic energy models for contemporary multicore processors. In: Yokota, R., Weiland, M., Keyes, D., Trinitis, C. (eds.) High Performance Computing. pp. 22–43. Springer International Publishing, Cham (2018), DOI: 10.1007/978-3-319-92040-5_2
14. Hornich, J., Hammer, J., Hager, G., Gruber, T., Wellein, G.: Collecting and presenting reproducible intranode stencil performance: INSPECT. Supercomputing Frontiers and Innovations 6(3), 4–25 (2019), DOI: 10.14529/jsfi190301
15. Ilic, A., Pratas, F., Sousa, L.: Cache-aware roofline model: Upgrading the loft. IEEE Comput. Archit. Lett. 13(1), 21–24 (2014), DOI: 10.1109/L-CA.2013.6
16. Intel Corporation: Intel Xeon Processor Scalable Family (2019), `http://tiny.cc/IntelSP`
17. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter pp. 19–25 (1995)
18. Ofenbeck, G., Steinmann, R., Cabezas, V.C., Spampinato, D.G., Püschel, M.: Applying the roofline model. In: IEEE International Symposium on Performance Analysis of Systems and Software, 23-25 March 2014, Monterey, CA, USA. pp. 76–85. IEEE (2014), DOI: 10.1109/ISPASS.2014.6844463

19. Peraza, J., Tiwari, A., Laurenzano, M., Carrington, L., Ward, W.A., Campbell, R.: Understanding the performance of stencil computations on Intel's Xeon Phi. In: 2013 IEEE International Conference on Cluster Computing, 23-27 Sept. 2013, Indianapolis, IN, USA. pp. 1–5. IEEE (2013), DOI: 10.1109/CLUSTER.2013.6702651

20. Sadasivam, S.K., Thompto, B.W., Kalla, R., Starke, W.J.: IBM Power9 processor architecture. IEEE Micro 37(2), 40–51 (2017), DOI: 10.1109/MM.2017.40

21. Seiferth, J., Alappat, C., Korch, M., Rauber, T.: Applicability of the ECM performance model to explicit ODE methods on current multi-core processors. In: Yokota, R., Weiland, M., Keyes, D., Trinitis, C. (eds.) High Performance Computing, 24-28 June 2018, Frankfurt, Germany. Lecture Notes in Computer Science, vol. 10876, pp. 163–183. Springer International Publishing, Cham (2018), DOI: 10.1007/978-3-319-92040-5_9

22. Stengel, H., Treibig, J., Hager, G., Wellein, G.: Quantifying performance bottlenecks of stencil computations using the Execution-Cache-Memory model. In: Proceedings of the 29th ACM International Conference on Supercomputing, June 2015, Newport Beach, CA, USA. ACM, New York, NY, USA (2015), DOI: 10.1145/2751205.2751240

23. Terpstra, D., Jagode, H., You, H., Dongarra, J.: Collecting performance data with PAPI-C. In: Müller, M.S., Resch, M.M., Schulz, A., Nagel, W.E. (eds.) Tools for High Performance Computing 2009. pp. 157–173. Springer Berlin Heidelberg, Berlin, Heidelberg (2010), DOI: 10.1007/978-3-642-11261-4_11

24. Wichmann, K.R., Kronbichler, M., Löhner, R., Wall, W.A.: Practical applicability of optimizations and performance models to complex stencil based loop kernels in CFD. International Journal of High Performance Computing Applications 33(4), 602–618 (2018), DOI: 10.1177/1094342018774126

25. Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. Commun. ACM 52(4), 65–76 (2009), DOI: 10.1145/1498765.1498785

26. Wittmann, M., Hager, G., Zeiser, T., Treibig, J., Wellein, G.: Chip-level and multi-node analysis of energy-optimized lattice Boltzmann CFD simulations. Concurrency and Computation: Practice and Experience 28(7), 2295–2315 (2016), DOI: 10.1002/cpe.3489

DOI: 10.14529/jsfi200205

Dawn: a High Level Domain-Specific Language Compiler Toolchain for Weather and Climate Applications

Carlos Osuna[1]***, Tobias Wicky***[1]***, Fabian Thuering***[1,2]***, Torsten Hoefler***[3]***, Oliver Fuhrer***[1]

High-level programming languages that allow to express numerical methods and generate efficient parallel implementations are of key importance for the productivity of domain-scientists. The diversity and complexity of hardware architectures is imposing a huge challenge for large and complex models that must be ported and maintained for multiple architectures combining various parallel programming models. Several domain-specific languages (DSLs) have been developed to address the portability problem, but they usually impose a parallel model for specific numerical methods and support optimizations for limited scope operators. Dawn provides a high-level concise language for expressing numerical finite difference/volume methods using a sequential and descriptive language. The sequential statements are transformed into an efficient target-dependent parallel implementation by the Dawn compiler toolchain. We demonstrate our approach on the dynamical solver of the COSMO model, achieving performance improvements and code size reduction of up to 2x and 5x, respectively.

Keywords: GPGPU computing, DSL, weather and climate, code optimization, compiler, performance portability.

Introduction

High resolution weather and climate simulations are subject of an unprecedented scientific interest due to the urgent need to reduce uncertainties of climate projections. Despite the progress achieved in the last years, the uncertainties have remained large, and improvements in the projections are crucial for designing and adopting efficient mitigation measures.

There is clear evidence in the literature that increasing resolution is one of the key factors to reduce the uncertainty. The horizontal resolution of current state-of-the-art climate model projects range between 50 km and 100 km. At these resolutions, several physical processes of key importance, e.g. the formation and evolution of deep convection must be parametrized. However, these processes can be explicitly resolved on the computational grid at horizontal resolutions around one kilometer. Unfortunately, when employing explicit numerical solvers that need to respect the Courant-Friedrichs-Lewy (CFL) condition, an increase of 2x in horizontal resolution implies a factor 8x in computational cost. Since the resolution of climate models is constantly improving [24], they will quickly become a major workload on supercomputers which requires increased attention to their performance [25].

Although some of the most powerful supercomputers provide an extraordinary computational power, weather and climate models cannot take full advantage of these leadership class systems. The end of Dennard's scaling [7] has led to the adoption of many-core accelerators, hybridization and diversity of supercomputers. Weather and climate models are complex numerical codes that contain from hundred thousands to millions lines of codes, and the community is struggling to migrate and maintain the models for multiple computing architectures. Due to the lack of standard parallel programming models that can be used by compilers to parallelize models

[1]Federal Institute of Meteorology and Climatology MeteoSwiss, Zurich, Switzerland
[2]NVIDIA, Berlin, Germany
[3]Department of Computer Science, ETH Zurich, Zurich, Switzerland

implemented with sequential programming languages like Fortran on any architecture, domain scientists are forced to combine multiple pragma based models like OpenMP and OpenACC, in addition to usage of distributed memory parallelization with MPI. The result is a mixture of multiple compiler directives with redundant semantic and numerical algorithms, often combined with attempts to customize data layouts for different architectures using preprocessor conditionals. In an attempt to tackle the portability problem, numerous domain-specific languages (DSLs) are being developed and used to port parts of weather and climate models. However, they still require to specify parallel programming model semantics that is crucial to generate correct parallel implementations and instruct the DSL compiler with necessary information to apply key optimizations. This generates a significant amount of boilerplate and code redundancy and impacts the scientific productivity of the model developer. The result are error-prone implementations where the user must be careful to avoid data races when new computations are inserted into pre-existing templates. Even if the use of these DSLs is an important step towards providing a solution that allows to retain a single source code, they have not significantly improved the ease of use, safety in parallel programming and programmer productivity for domain scientists. Additionally, the scope of computations covered by existing DSLs is very limited, since they cannot deal with the analysis and optimizations of large numerical discretization algorithms with complex data dependencies. We present Dawn, a DSL compiler toolchain that offers a descriptive, high-level language for the domain of partial differential equation solvers using finite difference or volume discretizations. The Dawn DSL language provides a parallel programming language for weather applications with sequential semantics where the user does not have to consider data dependencies. The highly descriptive language where optimization and parallelization are abstracted significantly reduces the amount of necessary code to express the numerical algorithms in a parallel architecture and consequently improves maintainability and productivity of the domain scientist. In contrast to other high-level frameworks (e.g., PETSc or MATLAB), the domain scientist retains full control over the discretizations and solvers employed.

The Dawn DSL compiler aims at porting full components, within the scope of the language. This allows the toolchain to apply data locality optimizations across all the components of a model. The authors are not aware of any other DSL or programming model that enables global model optimizations with all the functionality required by the weather and climate domain. We demonstrate the Dawn DSL compiler on the full dynamical core of the COSMO weather and climate model [4]. The dynamical core of a weather and climate model solves the Navier-Stokes equations of motion of the atmosphere and its discretization generates the most complex computational patterns of a model. Our results show that it is feasible to port entire dynamical cores to a high-level descriptive DSL obtaining more efficient implementations and maintainable codes.

The contributions of this paper are as follows:

- We introduce Dawn, an open-source DSL compiler toolchain including front-end language and a comprehensive set of optimizers and code generators.
- We propose a high-level intermediate representation to interoperate tools and DSL front ends.
- We demonstrate the usability and performance of the Dawn DSL compiler on the dynamical core of COSMO, a representative weather and climate code.

The document is organized as follows: Section 2 describes the language and main features supported by the DSL for weather and climate models. Section 3 provides a comprehensive description of the dawn compiler and the algorithms of the different DSL compiler passes. Finally Section 4 shows performance results for the dynamical core of COSMO and comparisons with the operational model running on GPUs.

1. Related Work

Numerous DSLs for stencil computations have been developed and proposed in the last decade in order to solve the performance portability problem. In the image processing field, Halide [22] provides a language and an autotuning framework to find an optimized set of parameters and strategies. PolyMage [18] provides instead a performance model heuristic. However, they lack in general functionality required for the specific domain of weather and climate, like 3D domains and a special treatment of the vertical dimension. Kokkos [5] provides a performance portable model for many core devices which has been demonstrated on the E3SM model [1]. The programming model contains useful parallel loop constructs and data layout abstractions. The CLAW DSL [3] is a Fortran based DSL for column based applications. It can apply a large set of transformations and generate OpenACC or OpenMP codes. However, it is limited to single column type of computations, like physical parametrizations, and not suitable for dynamical cores. STELLA [10] (and its successor GridTools) has been the first DSL running operationally for a weather model in a GPU-based supercomputer. The DSL supports finite differences methods on structured grids and is embedded in C++ using template metaprogramming. PSyclone [21] is a Fortran based DSL for finite elements/volumes dynamical cores being demonstrated for the NEMO ocean model and the LFric weather model. It relies on metadata provided together with the main stencil operators, in order to apply transformations and optimizations like loop fusion. Various tools and approaches such as Patus [2], Modesto [11], or Absinthe [12] tune low-level stencil implementations and could be combined with a stencil DSL. However, all of the existing approaches known to the authors provide a language where the user has the responsibility to declare the data dependencies via metadata, boilerplate of the language or need to resolve explicitly the data dependencies by choosing the computations that are fused into the same parallel component.

2. Abstractions for Weather and Climate

2.1. The Weather and Climate Domain

The domain we target are computational patterns of weather and climate models on structured grids where each grid cell can be addressed by an index triplet (i, j, k). We focus on algorithmic motifs with direct addressing where a series of operators are applied to update grid point values. Further, no explicit dependency of the operator on the horizontal positional indices (i, j) is assumed (with the exception of boundary conditions). This domain contains discretizations of partial differential equations using finite difference or volume methods as well as physical parametrizations. The main computational patterns resulting from these numerical discretizations are compact horizontal stencils in the horizontal plane and implicit solvers like tridiagonal solve in the vertical dimension.

In the following sections we introduce the Dawn DSL frontend (GTClang) and an intermediate representation (IR) designed as a minimal set of orthogonal concepts that can be used to represent these computational patterns with a high-level of abstraction.

2.2. GTClang Frontend

GTClang [20] is a DSL frontend that provides a high-level descriptive language for expressing finite difference/volume methods on structured grids. GTClang takes the view of a series of computations at a single horizontal location (i, j). Unlike other approaches such as the STELLA DSL [10], the language assumes a sequential model where all the data dependencies and data hazards will be resolved by the toolchain when constructing a parallel implementation from the sequential specification. The frontend language is embedded in C++ using the Clang compiler [15] to parse and analyze the C++ abstract syntax tree (AST). It provides a high level of interoperability, allowing to escape the DSL language within the same translation unit.

Figure 1 shows the main language elements of the DSL for an horizontal stencil example, while Fig. 2 shows how to execute the generated backend specific implementation from a C++ driver program.

```
globals {
  double eddlat, eddlon;
  double r_earth = 6371.229e3;
}
stencil_function avg {
  storage data;
  direction d;
  Do {
    return (data[d+1] - data[d-1])*0.5;
  }
}
stencil_function delta {
  storage data;
  offset off;
  Do {
    return (data[off] - data);
  }
}
stencil_function laplacian {
  storage data;
  Do{
    return data[i+1] + data[i-1] +
           data[j+1] + data[j-1] -
           4.0*data;
  }
}

stencil hd_smag {
 // Input-Output fields
 storage u, v;

 // Input fields
 storage hdmaskvel;
 storage[j] crlat;

 // Temporaries
 var T_sqr_s, S_sqr_uv;

 Do {
  vertical_region(k_start, k_end) {
   var frac_1_dx = crlat * eddlon;
   var frac_1_dy = eddlat / r_earth;

   var T_s = delta(j-1, v) * frac_1_dy -
       delta(i-1, u) * frac_1_dx;
   T_sqr_s = T_s * T_s;

   var S_uv = delta(j+1, u) * frac_1_dy +
       delta(i+1, v) * frac_1_dx;
   S_sqr_uv = S_uv * S_uv;

   var smag_u = math::sqrt((avg(i+1,
       T_sqr_s) +
       avg(j-1, S_sqr_uv))) - hdmaskvel;

   smag_u = math::min(0.5, math::max(0.0,
        smag_u));

   var smag_v = math::sqrt((avg(j+1,
       T_sqr_s) +
       avg(i-1, S_sqr_uv))) - hdmaskvel;
   smag_v = math::min(0.5, math::max(0.0,
        smag_v));

   u += smag_u * laplacian(u);
   v += smag_v * laplacian(v);
}}}
```

Figure 1. Smagorinsky diffusion operator example implemented using the GTClang front-end language. For simplicity some functions like delta, avg are omitted

The main language elements shown in the example are the following:

- **`stencil`** is the main computation concept that contains the declaration of all the input-output (`storage`) and temporary (`var`) fields and the `Do` body with the sequence of stencil-like computations.
- **`vertical_region`** allows to specialize computations for different regions of the vertical dimension. Atmospheric codes require to specialize equations at the boundaries or at custom regions of the vertical dimension. Since weather models do not need to specialize computations for regions of the horizontal plane, the semantic of this keyword is restricted to the

```

// define a runtime domain as a triplet of
// sizes and halos on each direction/dimension
domain dom(128,128,80,halos{3,3,3,3,0,0});

// declare all storages
metadata storage_info(128,128,80);
storage u(storage_info, "u"), v(storage_info,"v");
//...

hd_smag(domain, u, v, hdmaskvel, crlat);
hd_smag.run();
```

Figure 2. Execution in a C++ program of the smagorinsky computation declared in Fig. 1

vertical dimension. `k_start/k_end` are builtin identifiers marking the vertical levels of the top/bottom of the atmosphere.

- **`var`** declares a variable for a temporary computation. The dimensionality and type of memory where the temporary field will be stored is derived by specific analysis passes (Section 3.2.2).
- **`stencil_function`**, allows to define numerical functions that can be used within the `vertical_region` to increase readability of the numerical algorithm. They can be parametrized with fields (line 20), grid-point offsets (line 14) and dimensions (line 7).
- **`storage[dim]`**, declares fields of certain dimensionality (default is a 3D storage). The grid dimensionality of var declarations is deduced by an analysis pass of the toolchain, while it is explicitly for input-output storage fields.
- **`i,j,k`** are builtin identifiers for each of the cartesian dimensions.
- Neighbor grid point field access operator **`[]`**, like `u[i+1]` allows to access fields at neighboring grid points of the center of the stencil operation.
- **`global`** defines global scalar parameters, like model configuration variables, with a global scope. They can be defined at compile time or runtime.

The language assumes an array-like notation, where a loop over the entire domain is implicit and indices on dimensions are only used when accessing neighbor grid points. A GTClang statement:

```
b = a[i+1]
```

would be equivalent to the explicit array notation

```
b[h:end-h] = a[h+1:end-h+1],
```

where `h` is the halo size.

The main drawback of the array notation is that for a parallel compiler is not trivial to determine in general which statements can be fused in the same parallel region due to data dependencies. Indeed not all the statements of Fig. 1 can be inserted in the same (i, j) parallel region due to dependencies, e.g., between lines 53 and 46. Other languages or DSLs like GridTools or Kokkos delegate the responsibility for splitting the computations into parallel regions that should not contain data dependencies to the user.

Since GTClang does not expose these parallel concepts in its language, the numerical methods can be implemented in a sequential manner without considering data hazards or having to split computations into different parallel region components. This increases safety and productivity of the scientific development compared to other programming models that expose parallel constructs in their language.

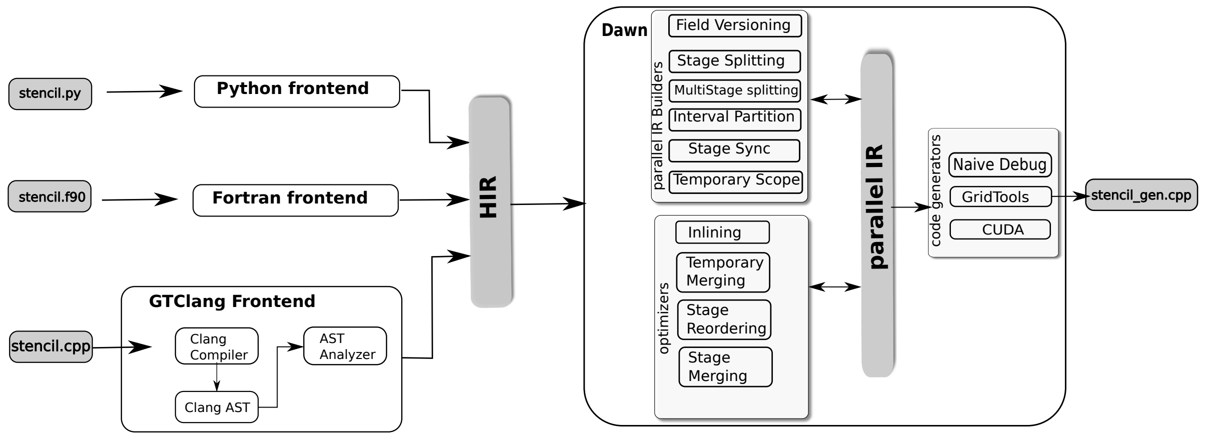


Figure 3. Architecture design of the dawn compiler

2.3. The High-Level Intermediate Representation

The high-level intermediate representation (HIR) is a representation that captures all (and only) the concepts required to express a high level language for weather and climate applications like the GTClang language presented in Section 2.2. The aim of the HIR is to provide a standard, programming language agnostic representation that enables sharing and reusing tools and optimizers such as Dawn across multiple frontend languages. Other DSLs and code-to-code translators like Fortran-based DSLs [3, 17] with the ability to parse and generate an IR will be able to transform the parsed code into HIR and use the Dawn compiler toolchain. The Dawn implementation uses Google protocol buffers to provide a specification of the HIR and thus communicates in a language-neutral manner between the DSL frontend and the DSL toolchain.

A comprehensive description of the HIR specification can be found in [19].

3. Compiler Structure

In this section, we will present the dawn [20] compiler structure and a description of all the components from the HIR to the code generation. The different layers of the compiler toolchain, including the GTClang front end, the standard interface HIR and the Dawn compiler, are shown in Fig. 3.

3.1. The Dawn Parallel IR

The HIR represents the user specification of computations in a sequential form. In order to be able to generate efficient parallel implementations from the HIR, we need to define a parallel model and map the HIR computations into that parallel model.

The Dawn compiler uses a parallel model that consist of a pipeline of computations (*Stages*) that are executed in parallel for the horizontal plane (i, j). *MultiStages* contains a list of *Stages* that are sequentially executed for each horizontal plane. Finally each *MultiStages* is executed over a vertical range. The vertical data dependencies in the user-specified computations determine the execution strategy of the vertical looping: forward, backward, or parallel. Multiple stages are then fused within the same parallel kernel connected by temporary computations stored in on-chip memory. The horizontal plane is tiled in order to fit the temporary computations into limited-size caches. However, stencil computations accessing data from grid cells of neighboring tiles will create data races in the case of horizontal data dependencies, since parallel tile computations can not be synchronized in general. We construct a parallel model based on redundant computations [10], where all intermediate computations are computed privately by each tile.

In order to code-generate implementations that follow this parallel model, Dawn defines a parallel IR that enriches the HIR with additional concepts such as *Stages* and *MultiStages* of the parallel model. Different optimizer-passes are responsible for creating a valid parallel IR representation from the HIR. The main data structure of the parallel IR is defined as a tree, shown in Fig. 4.

Computation:
- Scope of a stencil computation, contains one or more parallel kernels and fields allocated to it
- -*[MultiStage]*:
  - A parallel kernel over the entire 3D domain, where the vertical loop direction can be specified
  - -vertical loop strategy: (forward —— backward —— parallel)
  - -*[IntervalComputation]*:
    - A set of stage computations defined within a vertical region
    - -range: [k0,k1]
    - -*[Stage]*:
      - A group of computations performed in parallel in the horizontal (i, j) plane
      - -*[AST]*:
        - The AST of the arithmetic statements

Figure 4. Parallel IR data model tree. The operator [] denotes an array of nodes

Figure 5 shows the parallel IR representation of the smagorinsky example. Since the example does not contain vertical dependencies, the organization of the HIR computations into *MultiStages* and interval computations information is trivially computed.

```

-Computation:
  -MultiStage:
    -vertical_loop_strategy: parallel
    -IntervalComputation: [k_start,k_end]
      -Stage:
        var frac_1_dx = crlat * eddlon;
        var frac_1_dy = eddlat / r_earth;

        var T_s = delta(j-1, v) * frac_1_dy -
          delta(i-1, u) * frac_1_dx;
        T_sqr_s = T_s * T_s;
        var S_uv = delta(j+1, u) * frac_1_dy +
          delta(i+1, v) * frac_1_dx;
        S_sqr_uv = S_uv * S_uv;
      -Stage:
        var smag_u = math::sqrt((avg(i+1, T_sqr_s) +
        avg(j-1, S_sqr_uv))) - hdmaskvel;
        smag_u = math::min(0.5, math::max(0.0, smag_u));
      -Stage:
        var smag_v = math::sqrt((avg(j+1, T_sqr_s) +
        avg(i-1, S_sqr_uv))) - hdmaskvel;
        smag_v = math::min(0.5, math::max(0.0, smag_v));

        u += smag_u * laplacian(u);
        v += smag_v * laplacian(v);
}}}
```

Figure 5. Parallel IR of the smagorinsky diffusion operator defined in Fig. 1

The parallel IR defines different types of data storages:

- User-declared fields: N dimensional fields that are owned by the user with a scope that goes beyond the *Computation*. The GTClang front end declares them using the *storage* keyword.
- Temporary fields: storage with a scope limited to any of the levels of the parallel IR. The data allocation and dimensionality of the field will depend on the scope and will be managed by the toolchain.
- global parameters: scalar basic types to describe non gridded data, like model configuration switches.

In addition to the parallel IR tree, a program that assembles operators in a model requires a control flow that can schedule the different computations. The *control flow AST* contains the sequence of AST nodes to define a control flow (conditionals, loop iterations, etc.) and nodes for calls to

- *Computations* defined in the parallel IR;
- boundary conditions;
- halo exchanges.

All the analyses and optimizations are performed across multiple *Computation* nodes without control flow dependencies like conditionals or iterations. Therefore program dependence graphs and control dependence analyses are not used by the toolchain. The *control flow AST* is only stored as part of the parallel IR for code generation purposes.

3.2. Optimization Process

A comprehensive list of compiler passes are responsible for organizing the HIR statements into a valid parallel IR and run optimization algorithms to prepare the IR for efficient code generation. The passes are organized in three different categories: parallel IR builders, optimization passes and safety checkers. The safety checkers contain checks for detection of ill-formed numerical codes like access to uninitialized temporaries or write-after-write (WAW) data hazards. The following will describe the most relevant passes of the first two categories:

3.2.1. Parallel IR Builders

Field Versioning. Numerical discretizations often update a field reusing the same field identifier for readability of the code and to minimize memory footprint. In the following PDE example

$$\frac{\delta u}{\delta t} = f(u) + g(u), \tag{1}$$

that is discretized to

$$u_{t+1} = u_t + \Delta t(f(u_t) + g(u_t)) \tag{2}$$

if $f(u_t)$ or $g(u_t)$ involves an access to neighbour grid points in the parallel dimension, the field update can generate data races if the right-hand side (RHS) and the left-hand side (LHS) are evaluated in the same parallel region. Often this is solved by using a double buffering technique in the model implementation:

$$u\_out = u\_in + dt * (f(u\_in) + g(u\_in)). \tag{3}$$

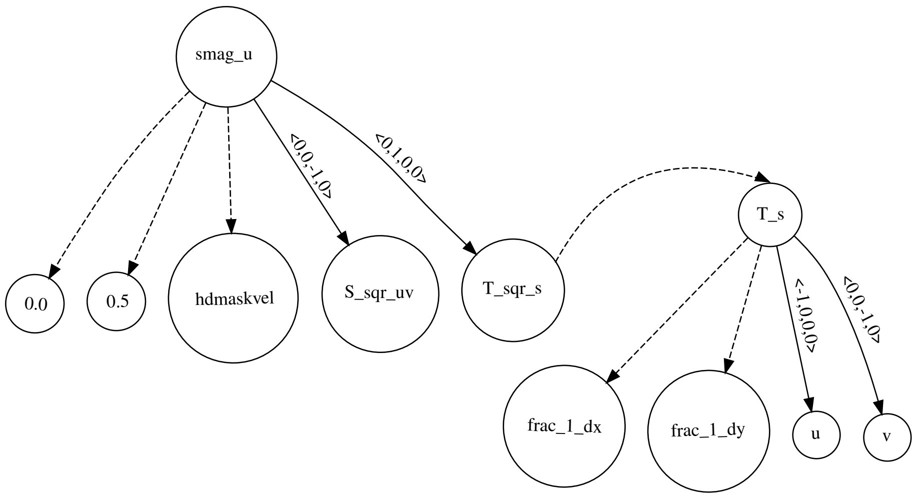


Figure 6. Horizontal data dependency DAG of a subset of the smagorinsky diffusion operator (Fig. 1)

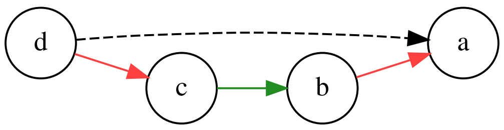


Figure 7. Vertical data dependency DAG of the anti dependent pattern example (Fig. 8)

Since the high-level DSL abstracts away the details of parallelization, it is not possible to know if RHS and LHS are evaluated within the same parallel region. Therefore GTClang allows to update fields with stencil computations that generate write-after-read (WAR) data dependencies. The field versioning pass will create versions of the field to ensure data hazards are resolved in parallel regions. Read-after-write (RAW) are resolved by the stage splitting pass and write-after-write (WAW) are legal but should be protected with a compiler warning, since the numerical algorithm might be ill-formed.

The field versioning pass operates on the following DAG formulation: field accesses are represented by nodes in the DAG where edges connect fields according to data dependencies. Edges are annotated with the horizontal *stencil extent* with the notation `<iminus, iplus, jminus, jplus>`.

There are two type of edges: solid lines edges for data dependencies with a (not null) horizontal stencil extent and dashed lines for connecting data with null extents or literals (see Fig. 6).

The algorithm identifies WAR hazards by finding strongly connected components (SCC) in the DAG where at least one of the nodes is a non temporary field and contains at least a solid edge. Temporary fields have a special allocation with redundant halos per block which allow WAR, unless they are within a single statement.

Stage Splitting. The stage splitting pass will organize the original sequence of statements, ($stmt_n$), into *stages* of the parallel IR in order to resolve (RAW) data dependencies that would introduce data races in the horizontal parallelization.

An example of such data races is observed in the horizontal diffusion smagorinsky example in the following lines:

```
T_sqr_s = T_s * T_s;
var smag_u = math::sqrt((avg(i+1, T_sqr_s) + avg(j-1, S_sqr_uv))) - hdmaskvel;
```

Algorithm 1: Stage splitting algorithm

```
for $stmt\ in\ J(stmt_n)$ do
    $D' \leftarrow D$ ;
    insert stmt accesses in D';
    if $\exists\{i,j\}$ such that $D_{ij} == 1$ and $\exists j'$ such that $D_{jj'} != 0$ then
        create new stage from D';
        $D \leftarrow \varnothing$ ;
        insert stmt accesses in D ;
    else
        $D \leftarrow D'$
```

The stage splitting pass resolves this by placing statements with data dependencies in separate *stages*.

The algorithm finds a partition of the DAG (see Fig. 6) where any solid edge can only be connected to leaves of the DAG. Let D_{ij} be the adjacency matrix of the DAG, where elements of the matrix can take two values for elements that are connected: 1 for a solid edge and 2 for dashed edges. The algorithm (Algorithm 1) iterates over the statements in the reverse sequence of statements, $J(stmt_n)$, where J is the backward identity matrix.

Multistage Splitting. It is common in weather and climate applications to find vertical implicit solvers that introduce a loop-carried dependence (see Fig. 8).

Anti dependence patterns are also allowed for read-only fields or field accesses before a write.

```
vertical_region(k_start , k_start) {
  phi = 0;
}
vertical_region(k_start , k_end) {
  phi = phi[k-1];
}
```

```
vertical_region ( k_start+1 , k_end-1 ) {
  b = (a[k+1])*0.5;
  c = b[k-1];
  d = c[k+1]*a;
}
```

Figure 8. Examples of (left) vertical solver and (right) vertical antidependence pattern

On the contrary, an anti dependence pattern on a field after a write statement would access outdated or uninitialized data. The Multistage splitting identifies anti dependence patterns on temporary accesses and fixes them by splitting and creating a new multistage with reverse vertical loop ordering.

The algorithm is similar to stage splitting. It processes a graph where edges are colored green for in loop data dependence and red for anti dependencies (see Fig. 7).

The algorithm traverses each edge of the graph. If a red edge is found, the loop order is reversed to fix the anti dependence. If a red and a green edges are traversed, then the multistage is split. Edges on leaves are ignored, since they connect to read-only data.

Interval Computation Partition. The user DSL code is provided as an ordered sequence of (in general) overlapping *vertical region* computations. In order to be able to fuse computations of the different interval regions into a single vertical loop, the interval computation partition pass will reorganize the ordered set into a non overlapping ordered set of *interval computations* of the parallel IR (Section 3.1). The sequence can be described as an interval graph where edges connect interval nodes that are overlapping. From there, the interval partitioning algorithm derives a set of non-overlapping *interval computations* as follows:

Let e(X,Y) be any edge that connects two nodes X and Y.

```
for $e(X,Y) : G$ do
    $G' \leftarrow \{G \setminus \{X,Y\}, (X \cap Y), (X \ominus Y)\}$ ;
```

where the set operations on *interval computations* are defined as:

$$\{I^x, (stmt^x_n)\} \cap \{I^y, (stmt^y_n)\} = \{I^x \cap I^y, (stmt^x_n, stmt^y_n)\} \tag{4}$$

$$\{I^x, (stmt^x_n)\} \ominus \{I^y, (stmt^y_n)\} = \begin{cases} \{I^x \setminus I^y, (stmt^x_n)\} \\ \{I^y \setminus I^x, (stmt^y_n)\} \end{cases} \tag{5}$$

assuming that $IC^x = \{I^x, (stmt^x_n)\}$ and $IC^x <= IC^y$ in the ordered set of *interval computations*. Transformations are iteratively applied until non of the nodes are connected.

Scope of temporaries. The toolchain will size the dimensionality of the temporary storages according to the scope of the temporary usages in the parallel IR tree (Fig. 4). The scope of temporaries pass will determine the lifetime and scope of each temporary, and optimize the dimensionality required accordingly.

3.2.2. Optimization passes

The first instance of the parallel IR would generate legal parallel implementations but is still non optimized and requires further transformations in order to produce efficient implementations. This is done with a set of optimizations presented in this section. All the optimizations performed in the toolchain are domain specific based on analyses of the domain specific, high-level information stored in the parallel IR.

Stage Reordering. The compiler passes discussed in Section 3.2.1 are necessary to map a sequential description of the numerical algorithms into the parallel IR model from Section 3.1. However, the splitting algorithms tend to generate a large number of *Stages* and *MultiStages*. The stage reordering will reorder *Stages* according to data dependencies in order to group and merge them together, increasing the data locality of the algorithms. The algorithm operates on the stage dependency DAG, where a stage S_1 depends on stage S_2 if and only if:

- The vertical intervals where S_1 and S_2 are defined overlap.
- Both access at least one common field with the policies defined as in Tab. 1.

Table 1. IO policies to consider a data hazard between stages

		S1 policy		
		Input	Output	Input-Output
S2 policy	Input		x	x
	Output	x		x
	Input-Output	x	x	x

Table 1 extends the definition of write-after-read (WAR), write-after-write (WAW) and read-after-write (RAW) hazards [14] for a compiler framework without single static assignment, where input-output accesses to a field are allowed for a single statement.

The algorithm iterates over all the stages in a reverse order and finds the leftmost position in the tree where the stage can be moved, accordingly to the following criteria:

- If the *Stage* is moved into another *MultiStage*, the loop orders must be compatible. A forward order *Stage* can be inserted into a parallel but not into a backward order *MultiStage*.
- A *Stage* S_1 can be moved over S_2 only if there is no path from S_1 to S_2 in the stage dependency DAG.
- Placing a stage into a new *MultiStage* should not introduce any anti dependence in the vertical dimension (see multistage splitting pass).

Stage Merging. As a result of the stage and multistage splitting pass, potentially many fine grained stages might be generated. The stage reordering pass will naturally group stages together that are connected via data dependencies, increasing data locality. Since every stage requires synchronize, the stage merging pass will merge various stages into a single one. It contains two modes:

- merge stages that are trivially mergeable since they specify the same level of redundant computations [10];
- merge stages even if they specify different level of redundant computations.

Temporary Merging. As a result of the field versioning pass, or usage of many temporaries, the parallel IR usually contains more temporary allocations than required. This increases the memory footprint and the load on the scratchpad memory. The temporary merging pass will reduce the number of temporaries to the minimum required. The pass operates on a reduced version of the data dependency DAG (Fig. 6) where only temporary fields are represented as nodes, and where two nodes are connected if there is a path that connects them in the original DAG. A coloring algorithm of the DAG of temporaries will identify the required number of temporary fields, where nodes with the same color will share the same temporary identifier.

Inlining. The stage splitting pass will generate a pipeline of stage computations that require synchronization of the parallel computational units, since stages are connected via horizontal data dependencies. The stage computations are executed in a tiled decomposition of the domain, allowing to cache the data that connects the different stages in some type of fast on-chip memory.

```
stencil hori_diff {
 storage dphi, phi, c;

 Do {
  vertical_region(k_start, k_end){
   lap = -4*phi + c*(phi[i+1] +
       phi[i-1] + phi[j+1] + phi[j-1]);
   dphi = -4*lap + c*(lap[i+1] +
       lap[i-1] + lap[j+1] + lap[j-1]);
}}};
```

```
stencil hori_diff {
 storage dphi, phi, c;

 Do {
  vertical_region(k_start, k_end){
   dphi = (16+4*c*c)*phi
     -8*c*(phi[i-1]+phi[i+1]+phi[j-1]+phi[j+1])+
     c*c*(phi[i+2]+phi[i-2]+phi[j+2]+phi[j-2] +
     2*c*c*(phi[i+1,j+1]+phi[i+1,j-1]+
     phi[i-1,j+1]+phi[i-1,j-1]));
}}};
```

Figure 9. Fourth-order diffusion operator as a two stage Laplacian operator in DSL form (left) and the dawn inlined version (right)

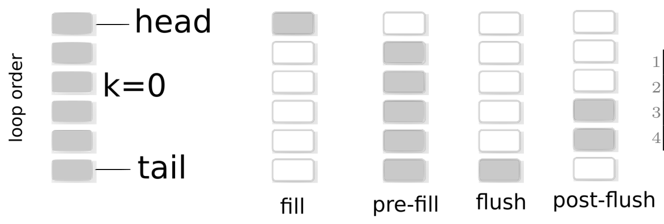
Alternatively, any intermediate computation that is not part of the input/output field declaration of the computation can be inlined, avoiding the memory operations but generating a larger stencil matrix computation. An example of a double Laplacian stage computation of a fourth-order diffusion operator is inlined in Fig. 9.

The algorithm traverses the reverse sequence of statements of each interval computation finding assignments on temporary fields that are only accessed in the parallel dimensions (i, j). The right hand side of the assignment is stored as the definition of a function that computes the temporary and the assignment stmt is removed. If the assignment statement depends on local variable declarations, they need to be promoted to temporary fields (with the right dimensionality) before recursively inlining all data dependencies of the RHS of the assignment.

A second iteration over the reverse sequence of statements will replace all field accesses to the temporary by the function definition evaluated at the stencil offset of the temporary access.

The pre-computation and inlined algorithms exhibit completely different arithmetic intensity. Performance of the different versions will depend on the computation (stencil shape, memory accesses, ...) and the hardware architecture and its memory subsystem.

Vertical register caching detection. An important optimization for implicit vertical solvers is to cache in registers a ring buffer of near k accesses reused through the vertical iteration, as in the following example: The ring buffer keeps values of vertical levels that will still be accessed from the vertical iteration of neighbour vertical levels, and it will synchronize values with the field in main memory whenever required according to the input-output pattern of the computation. Often the levels at the head of the ring buffer needs to be filled from the main memory for input accesses while values of the tail of the ring buffer must be flushed into main memory from the ring buffer. Additionally a pre-fill/post-flush operation of the ring buffer before/after processing the *Interval Computation* might be needed to synchronize the required sections of the ring buffer (Fig. 10).



```
vertical_region(k_start+3, k_end-2) {
  a = (a[k+2] + a[k+1] + a +
       a[k-1] + a[k-2])*0.2;
}
```

Figure 10. (Left) representation of the ring buffer and the different sections that will require synchronization with main memory for the vertical average example operation (right)

3.3. Code Generation

The last step in the toolchain is the code generation. The optimizer passes have organized the original HIR into a structured parallel IR that contains all the components required for generating an efficient parallel implementation with stencil computations, halo exchanges and boundary conditions. Dawn supports three code generators:

- a naive C++ sequential generator, used for debugging purposes;
- a GridTools DSL generator;
- a specialized CUDA generator.

All the existing generators make use of different GridTools components like the multidimensional storage facility and the halo exchange library. In particular the use of the multidimensional storage allows to escape the DSL language by using the storage objects that abstract away the details of memory layouts of the fields.

4. Experimental Results

In this section we present performance results obtained for the most relevant dynamical core operators of COSMO. They provide a diverse set of computational patterns of weather and climate application in Cartesian grids. The performance baseline of the dynamical core of COSMO has been analyzed in detail for NVIDIA GPUs [8].

The benchmarks are collected on the Piz Daint system at the Swiss National Supercomputing Centre (CSCS). The nodes are equipped with a 12-core Intel Xeon CPU (E5-2690 v3 @ 2.60 GHz) and an NVIDIA Tesla P100. Each node has 16 GB of HBM2 memory on the GPU. The nominal peak memory bandwidth of the GPU is 732 GB/s. All executables were compiled using gcc 7.3 and the nvvc compiler shipped with CUDA 10.0. Timing information is collected using CUDA events. All the experiments are verified by comparing the output of the optimized generated code with the naive C++ code generation on input data artificially generated using smooth mathematical functions.

4.1. COSMO Dynamical Core Results

Figure 11 shows the performance comparison between the production GPU code using the STELLA DSL and Dawn CUDA generation, for the most relevant stencils of the dynamical core of COSMO and a domain size of 128x128x80. Although Dawn does not support yet the STELLA tracer functionality that performs the same computation on multiple tracer fields in the same parallel kernel, two tracer operators (AdvPDBottX/Y) were added where the optimization in STELLA has been disabled for the comparison purposes. The data shown in the plots are showing the harmonic mean $\tilde{x}^{(h)} = \frac{n}{\sum_{i=1}^{n}(1/x_i)}$ [6, 13]. Errors were removed from Fig. 11 since they are not perceptible at the scale of the figure.

The performance of the Dawn CUDA backend obtained on P100 GPUs outperforms the STELLA optimized GPU production code, with performance improvements that vary from 2.62x (for HoriAdvPPTP operator) to same performance.

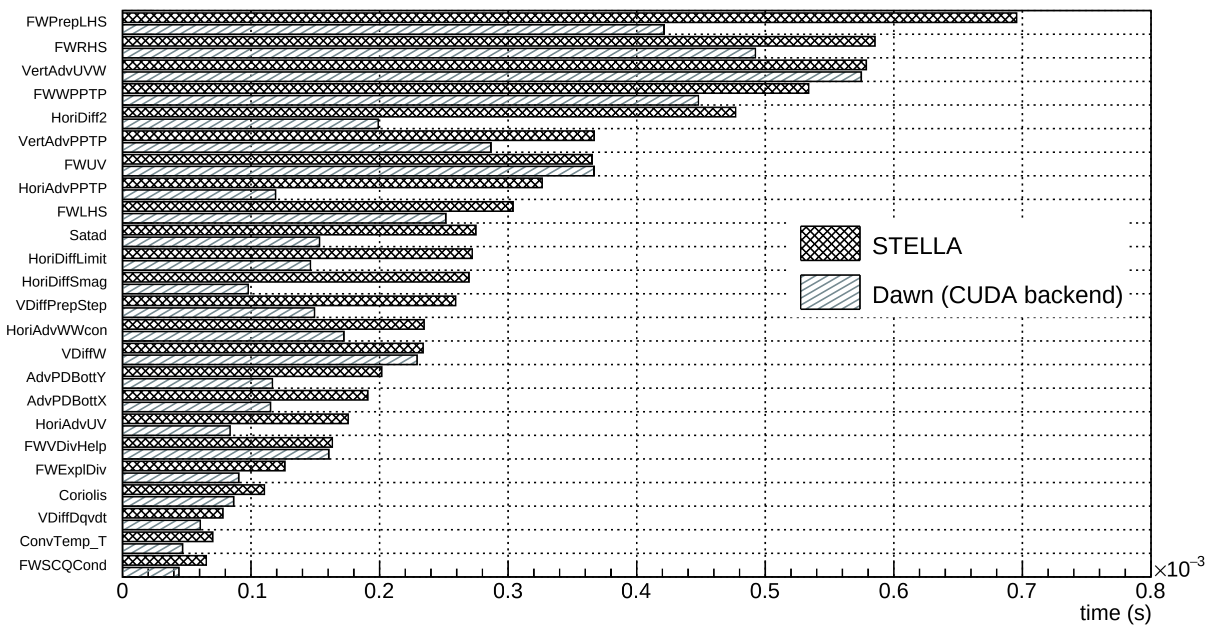


Figure 11. Performance of individual stencil objects of the COSMO dynamical core on P100 NVIDIA GPUs

In order to understand importance of the most relevant optimizers of Dawn, Tab. 2 evaluates the impact of disabling them for some of the components of the dynamical core of COSMO.

Table 2. Time (ms) for P100 GPU of some components of the dynamical core of COSMO measured with some of the Dawn optimizations disabled

	Full Opt	No stage reordering/stage merging
FastWaves	2.6	4.5
		No vert register caching
VertUVW	0.57	0.86
		No temporary merging
HoriDiffSmag	0.098	0.13

Table 3. Computational time (ms) of one time step of the fast waves solver of the dynamical core of COSMO for a P100 GPU

	time (ms)
STELLA	3.08902
Dawn	2.680

4.2. Global Optimizations

The STELLA based dynamical core of COSMO [10] was implemented as a set of complex stencil operators (see Fig. 11) that are glued together by a C++ driver code. The driver code connects all stencil components, implements time iterations, manage data storages, and performs other administrative functions. Contrary to Dawn, STELLA requires the user to organize the code in components that do not contain data dependencies.

Therefore, there is a compromise in designing how many computations are fused within each stencil component that will increase data locality but at the same time increases the complexity of the data dependencies that need to be understood by the user to produce correct and efficient code. Most of the stencils STELLA stencils of the dynamical core define no more than 4–5 *Stages* and 2–3 *MultiStages*. This approach has the following drawbacks:

- It limits the data locality optimizations that can be performed by the DSL since only a limited scope of computations are expressed within a DSL stencil object.
- It requires infrastructure that glues all the DSL stencil objects together increasing boiler-plate required to composed a full dynamical core.

One of the advantages of GTClang/Dawn compared to other approaches is the possibility to express entire models within the DSL language. We demonstrate this on the fast waves component of the dynamical core of COSMO, that is composed by 11 STELLA stencil objects, and a total size of approximately 4K lines of code.

The GTClang/Dawn implementation of the fast waves reduces the amount of lines of code to approximately 800. The performance results are summarized in Tab. 3.

The main optimizer pass that allows to integrate large stencil computations is the *stage reordering pass*. The same fast waves without the stage reordering pass takes 4.5 ms (Tab. 2). As shown in Tab. 3, the CUDA implementation generated by Dawn outperforms the version in production using the STELLA DSL. However, the *stage reordering pass* is an algorithm that tends to fuse computations together as much as possible within the same kernel, which in general will not lead to the most efficient implementation. An optimal solution should be driven by a performance model that minimizes HBM2/DDR memory accesses and at the same time minimizes on-chip resource consumption like shared memory or registers. The optimization problem is NP-hard and not solved for the complexity of a full dynamical core, although there are approaches that find solutions for a smaller problem sizes [11, 12, 23].

4.3. Maintainability and Model Development Productivity

One of the recurring parameters for maintainability across various models is lines of code (LOC). In order to quantify the gain in maintainability, we measured this for the fast waves of the dynamical core of COSMO, where the GTClang implementation (800 LOC) requires a factor 5x less than the operational code (4200 LOC). With similar order of magnitude the original Fortran implemention of the COSMO consortium requires 5000 LOC, as well as the optimized CUDA generated code of Dawn.

Therefore, GTClang/Dawn considerably reduces the amount of code required to express numerical algorithms which will increase the model development productivity and improve the model maintainability. However, there are other metrics in addition to LOC that contribute to significantly improve the maintainability:

Lack of Parallelism. Since the Dawn DSL does not expose parallelism to the user, a significant amount of boilerplate code as well as code complexity is no longer present in the user code. This does not only decrease the LOC but also increases safety, since parallel errors typical of programming models like OpenMP/OpenACC cannot occur.

Lack of Optimization. Since all the optimizations are performed by the Dawn toolchain, the GTClang language mostly expresses only the numerical algorithm. All optimizations and hardware dependent code, such as tiling, register or scratch-pad software managed cache, loop nesting, etc., hinder the scientific development. The use of a high-level language like GTClang increases considerably the readability of the numerical algorithm and model development productivity.

Reduction of Driver Code. As shown for the fast waves, the possibility to develop full models within the DSL, removes the necessity of complex infrastructure to glue all the stencil objects, data management and driver code that increases the overall maintenance of the model.

Conclusions

We have presented Dawn, a high-level DSL compiler toolchain that solves the performance portability problem of weather and climate applications. The DSL compiler is designed as a modern modular compiler. We demonstrated the usage of Dawn on the dynamical core of COSMO and presented performance results for the CUDA back end of Dawn on P100 GPUs. All the stencil computations outperform the optimized production code using the STELLA DSL, obtaining speedup factors of up to 2x. This significantly reduces the amount of code required (up to a factor of 5x). More importantly, the lack of explicit parallelism in the HIR and GTClang language provides a safe (against parallel errors) and highly productive scientific development environment.

The dynamical core is the most computationally expensive component of the model, accounting for 60 % of the total simulation time. Furthermore, it is the most complex in terms of computational patterns. Other components like the physical parametrizations contain column based only computations that are subset of the patterns supported by dawn for dynamical cores. Therefore the DSL toolchain proposed is applicable to entire models. Although the version of dawn presented here is demonstrated on the COSMO model, its applicability is not restricted to regional models. Additional development projects are porting the advection operators of the operational ocean model of NEMO [9]. Furthermore, the dawn collaboration is extending the toolchain in order to support two new categories of global weather and climate models:

- Cube sphere grids (adding support for corner and edge specializations). Current developments are working on porting the dynamical core of the FV3 model to DSL using dawn [16].
- Global models on triangular grids. New language extensions will allow to port models that can not be described with Cartesian operators. A version of the dynamical core of ICON [26] model is being developed using the DSL based on dawn.

Dawn is the only high-level DSL compiler known to the authors that allows to express an entire weather and climate model using a concise, simple and sequential language, and delivers an optimized model implementation that outperforms operational models even on modern GPU architectures.

Future developments will allow to apply this novel DSL language and compiler to a wider range of global models, and demonstrate its applicability on two of the most renowned models like FV3 and ICON.

References

1. Bertagna, L., Deakin, M., Guba, O., Sunderland, D., Bradley, A.M., Tezaur, I.K., Taylor, M.A., Salinger, A.G.: Hommexx 1.0: A performance portable atmospheric dynamical core for the energy exascale earth system model. Geoscientific Model Development Discussions 2018, 1–23 (2018), DOI: 10.5194/gmd-2018-218

2. Christen, M., Schenk, O., Burkhart, H.: Patus: A code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures. In: 2011 IEEE International Parallel Distributed Processing Symposium, 16-20 May 2011, Anchorage, AK, USA. pp. 676–687. IEEE (2011), DOI: 10.1109/IPDPS.2011.70

3. Clement, V., Ferrachat, S., Fuhrer, O., Lapillonne, X., Osuna, C.E., Pincus, R., Rood, J., Sawyer, W.: The CLAW DSL: Abstractions for performance portable weather and climate models. In: Proceedings of the Platform for Advanced Scientific Computing Conference, Basel, Switzerland. pp. 2:1–2:10. ACM, New York, NY, USA (2018), DOI: 10.1145/3218176.3218226

4. Doms, G., Baldauf, M.: A Description of the Nonhydrostatic Regional COSMO-Model – Part I: Dynamics and Numerics. COSMO – Consortium for Small-Scale Modelling (2015), `http://cosmo-model.org/content/model/documentation/core/cosmoDyncsNumcs.pdf`

5. Edwards, H.C., Trott, C.R., Sunderland, D.: Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of Parallel and Distributed Computing 74(12), 3202–3216 (2014), DOI: 10.1016/j.jpdc.2014.07.003

6. Fleming, P.J., Wallace, J.J.: How not to lie with statistics: the correct way to summarize benchmark results. Communications of the ACM 29(3), 218–221 (1986)

7. Frank, D.J., Dennard, R.H., Nowak, E., Solomon, P.M., Taur, Y., Wong, H.S.P.: Device scaling limits of Si MOSFETs and their application dependencies. Proceedings of the IEEE 89(3), 259–288 (2001), DOI: 10.1109/5.915374

8. Fuhrer, O., Chadha, T., Hoefler, T., Kwasniewski, G., et al.: Near-global climate simulation at 1 km resolution: establishing a performance baseline on 4888 GPUs with COSMO 5.0. Geoscientific Model Development 11(4), 1665–1681 (2018), DOI: 10.5194/gmd-11-1665-2018

9. Gurvan, M., Bourdall-Badie, R., Bouttier, P.A., Bricaud, C., et al.: NEMO ocean engine (2017), DOI: 10.5281/zenodo.3248739

10. Gysi, T., Osuna, C., Fuhrer, O., Bianco, M., Schulthess, T.C.: STELLA: a domain-specific tool for structured grid methods in weather and climate models. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 15-20 Nov. 2015, Austin, TX, USA. pp. 1–12. IEEE (2015), DOI: 10.1145/2807591.2807627

11. Gysi, T., Grosser, T., Hoefler, T.: MODESTO: Data-centric Analytic Optimization of Complex Stencil Programs on Heterogeneous Architectures. In: Proceedings of the 29th International Conference on Supercomputing, Newport Beach, CA, USA. pp. 177–186. ACM, New York, NY, USA (2015), DOI: 10.1145/2751205.2751223

12. Gysi, T., Grosser, T., Hoefler, T.: Absinthe: Learning an Analytical Performance Model to Fuse and Tile Stencil Codes in One Shot. In: Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques, 23-26 Sept. 2019, Seattle, WA, USA. pp. 370–382. IEEE (2019), DOI: 10.1109/PACT.2019.00036

13. Hoefler, T., Belli, R.: Scientific Benchmarking of Parallel Computing Systems. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 15-20 Nov. 2015, Austin, TX, USA. pp. 1–12. ACM (2015), DOI: 10.1145/2807591.2807644

14. Kuck, D.J., Kuhn, R.H., Padua, D.A., Leasure, B., Wolfe, M.: Dependence graphs and compiler optimizations. In: Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Williamsburg, Virginia. pp. 207–218. ACM, New York, NY, USA (1981), DOI: 10.1145/567532.567555

15. Lattner, C.: LLVM and Clang: Next generation compiler technology. In: The BSD conference, May 2008, Ottawa, Canada. vol. 5 (2008)

16. Lin, S.J.: A "vertically Lagrangian" finite-volume dynamical core for global models. Monthly Weather Review 132(10), 2293–2307 (2004), DOI: 10.1175/1520-0493(2004)132¡2293:AVLFDC¿2.0.CO;2

17. Melvin, T., Mullerworth, S., Ford, R., Maynard, C., Hobson, M.: LFRic: Building a new Unified Model. In: EGU General Assembly Conference Abstracts. EGU General Assembly Conference Abstracts, vol. 19, p. 13021 (2017)

18. Mullapudi, R.T., Vasista, V., Bondhugula, U.: Polymage: Automatic optimization for image processing pipelines. SIGARCH Comput. Archit. News 43(1), 429–443 (2015), DOI: 10.1145/2786763.2694364

19. Osuna C., Clement V.: MeteoSwiss-APN/HIR 0.0.1 (2019), DOI: 10.5281/zenodo.2629314

20. Osuna C., Thuering F., Wicky T., Dahm J., et al.: MeteoSwiss-APN/dawn: 0.0.2 (2020), DOI: 10.5281/zenodo.3870862

21. Porter, A.R., Appleyard, J., Ashworth, M., Ford, R.W., Holt, J., Liu, H., Riley, G.D.: Portable multi- and many-core performance for finite-difference or finite-element codes – application to the free-surface component of NEMO (NEMOLite2D 1.0). Geoscientific Model Development 11(8), 3447–3464 (2018), DOI: 10.5194/gmd-11-3447-2018

22. Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., Amarasinghe, S.: Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, Seattle, Washington, USA. pp. 519–530. ACM, New York, NY, USA (2013), DOI: 10.1145/2491956.2462176

23. Rawat, P.S., Rastello, F., Sukumaran-Rajam, A., Pouchet, L.N., Rountev, A., Sadayappan, P.: Register optimizations for stencils on GPUs. ACM SIGPLAN Notices 53(1), 168–182 (2018), DOI: 10.1145/3178487.3178500

24. Schär, C., Fuhrer, O., Arteaga, A., Ban, N., et al.: Kilometer-scale climate models: Prospects and challenges. Bulletin of the American Meteorological Society 101(5), E567–E587 (2020), DOI: 10.1175/BAMS-D-18-0167.1

25. Schulthess, T., Bauer, P., Fuhrer, O., Hoefler, T., Schaer, C., Wedi, N.: Reflecting on the goal and baseline for exascale computing: a roadmap based on weather and climate simulations. Computing in Science and Engineering (CiSE) 21(1), 30–41 (2019), DOI: 10.1109/MCSE.2018.2888788

26. Zangl, G., Reinert, D., Ripodas, P., Baldauf, M.: The ICON (ICOsahedral Non-hydrostatic) modelling framework of DWD and MPI-M: Description of the non-hydrostatic dynamical core. Quarterly Journal of the Royal Meteorological Society 141(687), 563–579 (2015), DOI: 10.1002/qj.2378