Article:
Saxena, G, Jimack, PK and Walkley, MA orcid.org/0000-0003-2541-4173 (2018) A quasi-cache-aware model for optimal domain partitioning in parallel geometric multigrid. Concurrency and Computation: Practice and Experience, 30 (9). e4328. ISSN 1532-0626

https://doi.org/10.1002/cpe.4328

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2010; **00**:1–30
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

A Quasi-Cache-Aware Model for Optimal Domain Partitioning in Parallel Geometric Multigrid

Gaurav Saxena [*,†], Peter K. Jimack and Mark A. Walkley

School of Computing, University of Leeds, Leeds LS2 9JT, UK

SUMMARY

Stencil computations form the heart of numerical simulations to solve Partial Differential Equations using Finite Difference, Finite Element and Finite Volume methods. Geometric Multigrid is an optimal $\mathcal{O}(N)$, hierarchical tool employing stencil computations in its chief constituents, namely smoothing, restriction, and interpolation. When Multigrid is parallelized over distributed-shared memory architectures, traditionally the domain partitioning creates cubic partitions of the mesh to minimize overall communication. Thus, the orthodox approach considers only load-balancing and communication minimization for completely determining the domain partitioning. In this article, we show that these two factors are not sufficient to obtain optimal partitions for Parallel Geometric Multigrid. To this effect, we develop and validate a high level analytical model to show that "close to 2-D" partitions for Geometric Multigrid can give higher performance than the partitions returned by the `MPI_Dims_create()` function which minimizes the communication volume by default. We quantify sub-domain level cache-misses in Parallel Geometric Multigrid and obtain families of optimal domain partitions. We conclude that the sub-domain level cache-misses for the application-specific stencil computational kernel and communicated planes should be taken into account in addition to communication minimization/load-balance to obtain optimal partitions for Parallel Geometric Multigrid. Copyright © 2010 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Domain partitioning ; Geometric Multigrid ; Quasi-cache-aware ; Topology ; Stencil ; Cache Misses

1. INTRODUCTION

Partial Differential Equations (PDEs) [1, 2, 3] can be used to model natural phenomena which involve the rate of change of variables with respect to other quantities. Since analytical solutions are not generally possible, they are solved numerically by simulation methods to obtain an approximate solution. Before simulating, the continuous physical domain must be discretized so that it can be represented on digital systems. Finite Difference Methods (FDMs), Finite Element Methods (FEM) and Finite Volume Methods (FVM) [4, 1, 5] are commonly used to discretize a PDE on such domains. Iterative methods like Jacobi, weighted Jacobi (ω-Jacobi), Gauss-Seidel (GS), Red-Black Gauss-Seidel (RBGS), Conjugate Gradient (CG) and other numerical methods can then be used to compute the solution [5, 4, 6]. Due to the slow rate of convergence of these iterative methods and the time taken to solve large systems, multilevel algorithms have been created that accelerate the rate of convergence to the solution. Multigrid methods are a class of multilevel algorithms that have been shown to be extremely effective in solving Elliptic PDEs, a class of PDEs which are completely specified using boundary conditions [1, 3, 5].

*Correspondence to: Gaurav Saxena, School of Computing, University of Leeds
†E-mail : scgs@leeds.ac.uk

The Multigrid [7, 8] method is an optimal hierarchical method which can be used for solving sparse systems of linear equations that arise from local discretization of Elliptic PDEs in $\mathcal{O}(N)$ time, where N is the number of unknowns or the degrees of freedom in the system. The discretization operator, in Finite Difference methods for example, can be expressed as a stencil - a fixed geometrical figure which is utilized to update the solution at various points in the discretized domain. The hierarchy in Multigrid consists of several systems corresponding to discretizations on several levels of grids of decreasing resolution, where the finest level grid represents the actual problem to be simulated. It accelerates the convergence of the solution by eliminating low frequency error components on the series of coarse grids. To further decrease the solve time of Multigrid methods, they are parallelized on distributed, shared memory or hybrid architectures to allow simulation of extremely large scale problems [9, 10, 11], where the unknown variables can be of the order of billions or trillions. These parallel processes typically communicate using the MPI (Message Passing Interface) [12] library, the de-facto standard for distributed programming. It is the parallelization of Multigrid that is extremely challenging and requires a very careful design and implementation to achieve near perfect linear Weak Scaling and preserve its optimality.

Overheads in the form of communication, load-imbalance, limited memory-bandwidth, imbalance between processor and memory speeds, network and memory latencies, complex memory hierarchies necessitate careful optimization [13, 14, 15, 16, 17, 18, 19, 20]. These can be broadly classified as serial overheads or parallel overheads. Serial overheads, i.e. overheads which would still be present in the absence of multicore architectures, can be considered as a subset of parallel overheads (overheads which come into existence only because of the introduction of multicores). For example, Cache misses, TLB (Translational Lookaside Buffer) misses and memory latencies are examples of serial overheads which we shall collectively refer to as *Serial Control Parameters* (SCPs). Operating System inter-process latencies, network bandwidth/latencies, non-optimal process placement, non-optimal domain partitions and cache conflicts in shared caches are examples of parallel overheads, which we shall refer to as *Parallel Control Parameters* (PCPs). Clearly, SCPs are a subset of PCPs. Figure 1 shows the various SCPs and PCPs. More research has explored SCPs and their optimization compared to PCPs, whilst there is a complex interaction of SCPs and PCPs which has little literature. Our focus is to investigate this. Due to the large interaction space between SCPs and PCPs, we restrict ourselves to the single most important SCP which we practically (and from the literature [17, 18, 19, 21, 22, 23]) identify to be Cache Misses. We take the first step in connecting Cache misses to Domain Partitioning (shown in rectangular boxes in Figure 1) on Hybrid architectures and then extend the model developed in [24] to parallel Geometric Multigrid (GMG). In general, domain partitioning is governed by two factors : load-balance and communication minimization. Cache optimizations are typically considered only *after* the domain partitioning is performed, i.e. at the sub-domain level. We develop a high level analytical model based upon [24] which is "quasi-cache aware", i.e. it is formulated in a cache-aware way by taking into account the cache-line size but produces a cache-oblivious result. The model analyzes cache-misses and uses this as a guiding factor to perform domain partitioning. Thus, the model looks at cache-optimization from a higher level compared to temporal and spatial cache-optimizations at the sub-domain level alone. Further, our model seeks to put this in the context of all the factors that might influence the choice of sub-domain shape and size. Thus, we qualitatively and quantitatively consider factors such as cache-misses, prefetching, cache-eviction policy, Vectorization etc., and explore their effect on determining optimal sub-domain dimensions. Though these factors have been separately well explored in the literature, the focus of our work is on establishing a *connection* between them and domain partitioning. We present the results of our investigations to evaluate the model's effectiveness in a multiple grid scenario and discuss its limitations to open further research avenues.

DOI: 10.1002/cpe

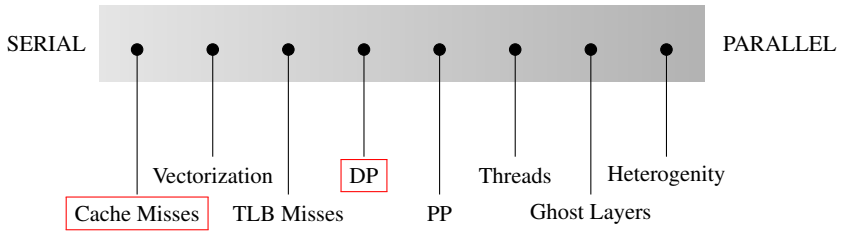


Figure 1. Serial Control Parameters (SCPs) Vs Parallel Control Parameters (PCPs), DP (Domain Partitioning), PP (Process Placement). Our focus : Cache Misses and Domain Partitioning

2. BACKGROUND AND RELATED WORK

We sub-divide this section into three logical parts which can be thought of as the constituent parts of parallel Geometric Multigrid (GMG). The first part recaps the main concepts behind Multigrid. The second part elaborates the concept of stencils, while emphasizing cache optimization. Finally, we describe the role of domain partitioning in the parallelization of GMG.

2.1. Multigrid

Multigrid [7, 5, 25, 8] is a multilevel convergence acceleration concept that involves using coarser forms [7, 25, 26] of the given fine grid to remove the low-frequency errors and provide a better estimate of the approximated solution. Local Iterative schemes [7, 8, 5, 25] can remove high frequency error components quickly (known as smoothing) but decrease the low-frequency error spectrum very slowly. These low-frequency components can be represented as relatively high frequency components on coarser grids [7, 8, 27]. The smoothing properties of iterative methods and the equivalent system of equations at various levels form the basis of Multigrid [27]. Depending on the pattern of the traversal between grids, two common types of cycles are categorized as *V-cycles* and *W-cycles* [7, 8].

Let $A^h u^h = f^h$ denote a linear system of equations arising from a local discretization of an elliptic PDE, where the superscript h denotes the grid spacing. Successive grid levels (finest to coarsest) are represented as: $\Omega^h \rightarrow \Omega^{2h} \rightarrow \Omega^{4h} ... \rightarrow \Omega^{2^i h}$. We use standard coarsening in our implementations which reduces the degrees of freedom by one-eighth on the immediate coarser grid. After ν_1 pre-smoothing operations on Ω^h, an *approximation* to u^h is obtained (denoted by v^h) and the *residual* is then calculated as $r^h = f^h - A^h v^h$. A *restriction* (I_h^{2h}) operator transfers this residual to the next immediate coarser grid (Ω^{2h}). In the 2-grid method, the error e^{2h} is obtained after solving $A^{2h} e^{2h} = r^{2h}$ (*error* equation) exactly on the coarser grid. This error is then transferred back to the finest grid using the *interpolation/prolongation* operator (I_{2h}^h) to obtain a better approximation to the solution on the finer grid, followed by ν_2 *post-smoothing* operations. For Multigrid, the error equation is not solved exactly, instead it is replaced by a recursive use of the 2-grid method to update the estimated error. Only at the coarsest level is an exact solve used. The *recursive* algorithm halts when the ratio of the current *norm* of the residual ($||r_k||$) to its initial *norm* ($(||r_0||)$) becomes less than a *specified tolerance*. Typically the pre-smoothing (ν_1) and post-smoothing (ν_2) iterations of the smoother vary between one and three for most practical problems [27].

2.2. Stencils and Cache Optimization

When PDEs are discretized using FDMs, the weighted contributions of the values of the neighbours of a point in geometrical space are used to approximate the differential operator at the point. This fixed geometrical shape known as a *Stencil* is then cycled through the entire domain. A 5-pt stencil in 2-D and a 7-pt stencil in 3-D are very commonly used. In a 7-pt stencil the differential operator at the central point is approximated as the weighted contribution of its six immediate neighbours, two in each direction. A 19-point and 27-point stencil may also be used in discretized

DOI: 10.1002/cpe

problems representing a 3-dimensional space [28, 17, 18]. Typically each iterative update of a point on a 7-pt stencil results in eight floating point operations (flops) when constant coefficients are considered [18]. Stencil computations are typically memory-bound as compared to compute-bound because the vertical memory bandwidth limits their performance/arithmetic intensity rather than computations. However, in some applications due to e.g. varying coefficients, nonlinearities, or more complex grids this *may not* be the case. Since data is mapped out linearly in memory regardless of the dimension of an array data structure, the non-contiguous access pattern produced by stencils increases the cache-misses [24]. Efforts have been made to optimize and exploit spatial and temporal principles of the cache memory hierarchy to bridge the gap between the fast processor speed and the comparatively slower memory access times [21, 17, 18, 19, 29, 20]. The Restriction and Interpolation inter-grid transfer operators in Multigrid are also Stencils. We make use of 27-pt stencils in 3-D for both these operators.

It is advisable to optimize implementations to fetch a higher fraction of data from the higher levels of memory (registers and L1 cache) while reducing the fraction of data fetched from lower level L3 cache (generally shared) and main memory [20]. *Cache tiling/blocking* techniques aim at bringing a sub-domain of data into the cache instead of traversing the entire domain in a single iteration [21, 29, 20]. The effectiveness of these cache tiling/blocking techniques in modern microprocessors has decreased due to advances in compiler technology and increasing size of on-chip caches [30], which enables data fetching from fixed sized memory regions. Fusion techniques are used in the Red-Black Gauss-Seidel (in 2-D, 5-pt stencil) method to combine the update of red and black points in a single sweep by updating red points in row i followed by black points in row $i-1$. Further, the red and black points for unknowns and the corresponding right-hand side values can be stored in different arrays to reduce the traffic between various cache hierarchies, although the total traffic to the main memory remains the same [31]. Initial ground-breaking work proposed the use of *partial 3-D blocking* for 3-D loops which maximizes the size of the dimension which has continuous data [28]. Analytical cost models for cache tiling fail to address the difference between load and store operations [21]. Further, cache conflict misses occur when the data is read from and written to different grids represented by multi-dimensional arrays in the memory as in the case of *Jacobi* (not in-place) updates [22]. However, these cache optimization techniques also interfere with automatic optimization techniques implemented in the hardware and software in the modern microprocessors [24]. These automatic *streaming* techniques consist of SIMD (Single Instruction Multiple Data) instructions (also called *Vectorization*) and *prefetching*. *Microbenchmarks* like the *Stanza Triad* (STriad) and *Stencil Probe* have been created that attempt to act as a proxy for modelling the prefetch behaviour of the actual program [21, 18]. These benchmarks do not account for the packing or unpacking times and the changing latency in the context of using *derived datatypes* in the *Message Passing Interface* (MPI) [32, 12] implementations. Researchers have used hardware performance counters like cache-misses, *Translational Look-Aside Buffers* (TLB) misses, mispredicted branches, hardware prefetches and *regression* analysis to predict the performance of stencil codes [23]. *Cache oblivious/transcendental* [33] algorithms have been proposed which ignore the hardware characteristics of caches as opposed to *Cache aware* algorithms which use cache specifications to minimize cache misses. The idea behind every memory optimization is to maximize the re-use of data while it still resides in the cache memory [20].

2.3. Domain Partitioning

The first step in the parallel implementation of a problem is division of computational work or data among processes/cores. We use a *Domain* partitioning (data division) approach where the largest data-structures representing the problem domain are assigned to processes/cores [6, 34]. In a distributed architecture where each sub-domain resides with a separate process, *ghost cells/halo data* must be introduced to exchange the neighbouring data for stencil computations [6, 32, 5]. Figure 2 shows the division of a 4x4 vertex centered domain into four sub-domains, each having a local size of 4x4 including ghost/halo cells. For structured stencil problems, e.g. solving a PDE on a unit cube using the flat MPI model and using a Cartesian topology [12], each process has a

DOI: 10.1002/cpe

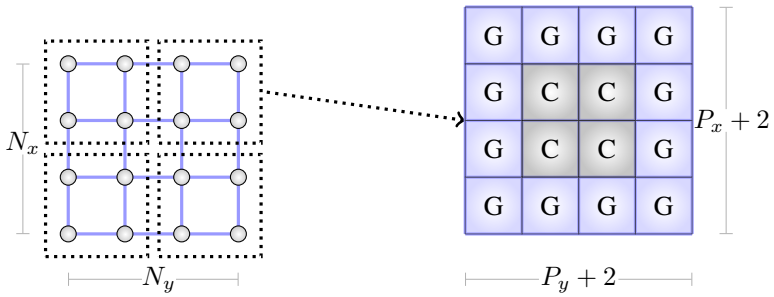


Figure 2. A 4x4 Vertex Centered Grid (VCG) is partitioned among 4 cores. The result is a 4x4 sub-domain with 4 original 'C' cells and added ghost layer cells 'G'. Each dotted partition is converted to a 4x4 sub-domain on an individual core. Domain size = $N_x \times N_y$, sub-domain size = $(P_x + 2) \times (P_y + 2)$

.

maximum of 6 neighbours if a 7-pt stencil is used with a 1-element deep ghost zone. Increased layers of ghost cells can allow more iterations to be performed between communication steps, but at the expense of increased memory usage [35]. Further, it is not necessary that each process will contain an equal number of sub-domain points, especially domains which have dimensions of the form $(2^l + 1) \times (2^m + 1) \times (2^n + 1)$ (in 3-D) [27, 7]. This creates a certain amount of load-imbalance between processes. Another type of load imbalance arises in domains where the work done per-grid point is variable. An example of the latter category is a *Dirichlet-Neumann* [8] boundary value problem where a boundary point adjacent to the Dirichlet boundary has to perform less compute work as compared to a boundary point which is immediately adjacent to the Neumann boundary. Parallelization further introduces a bottleneck when coarser grids in multigrid are solved due to the lower ratio of computation to communication. Communication aggregation and vertical traffic avoidance do not offer substantial benefits at coarser levels [35]. Further, for a very large core count, it can even be the coarsest grid which contributes to the maximum percentage of run-time [36] and the scalability of `MPI_Waitall()` becomes a bottleneck. Scalability of the coarsest level solvers is an important issue [35, 37]. The coarsest level solver is generally a direct solver [16] e.g. *MUMPS* (Multifrontal Massively Parallel sparse direct Solver, [9]) or *SuperLU* (SupernodalLU) [38], though recent research suggests that more efficient approaches can be used in practice [9, 16].

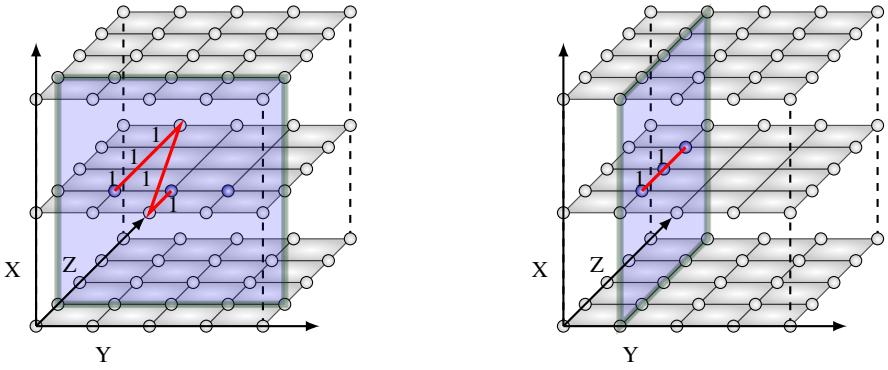
3. TERMINOLOGY AND PROBLEM DESCRIPTION

This section introduces the notation and assumptions on which our model is based, and gives a brief low-level description of the problem under consideration. This is followed by a description of the test problem that we use for our experiments.

3.1. Notation and Assumptions

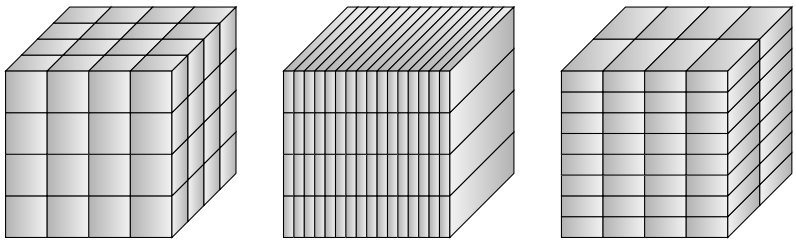
A structured 3-D grid having dimensions $N_x N_y N_z$ can be divided among P parallel processes running on individual cores in several ways. In general, D_i represents the number of processes along direction i where $i = x, y, z$. Thus, P can be decomposed as any valid permutation of D_x, D_y and D_z such that $P = D_x D_y D_z$, and for simplicity, we assume that *mod*(N_i, D_i) = 0. In the following we consider cuts/partitions parallel to the Cartesian axes and the model assumes a 7-pt stencil with a 1-element deep ghost zone. Each sub-domain with a single element deep ghost/halo zone has dimension $(P_x + 2)(P_y + 2)(P_z + 2)$. The 3-D sub-domain on each core can be viewed as 3 parts : the inner *Independent Computational* (IC) kernel which needs no data from neighbouring processes (zone 1), the next-to-boundary layer (*Dependent Planes*) which requires data from neighbouring processes for its update (zone 2) and the *buffer/ghost/halo* region (zone 3) [24]. Thus, in the worst case, a sub-domain will need to pass six planes to its nearest neighbour processes. Without loss of generality we assume that the unit stride dimension is in the Z-direction (see Z-axis in Figure 3a or 3b) and the data is in row-major order. It can be noted that four of the six nearest neighbours

DOI: 10.1002/cpe



(a) Front Z-plane (b) Left Y-plane

Figure 3. Single Z and Y-Plane, data continuity and distance between adjacent points shown in red



(a) 4x4x4 (b) 4x16x1 (c) 8x4x2

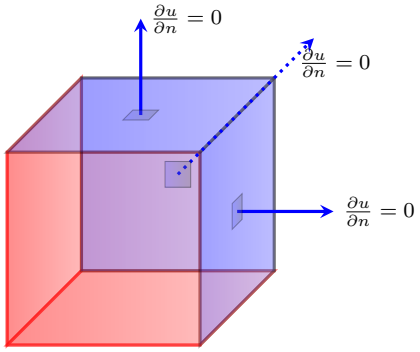
Figure 4. Examples of 3-D Process Topologies.

in the 7-pt stencil are not contiguous in memory. We collectively refer to the two YZ planes as X-planes, the two XY planes as Z-planes, and the two XZ planes as Y-planes. Figures 3a and 3b show the position of a single Z and Y-plane. The distance between adjacent mesh points for a Z-plane is $P_z + 2$ and *1* for the Y-plane (except at the boundaries where it is $(P_z + 2) * (P_y + 1) + 2$).

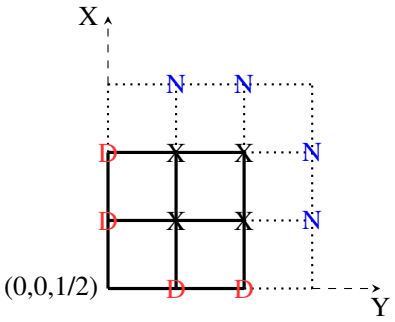
3.2. Problem Description

This division of P as $D_x D_y D_z$ can have a large effect on the packing/unpacking times of data which is to be sent to/received from the neighbouring sub-domains, the update of dependent planes, and the compute times of the Independent Compute kernel. There are several permutations of D_x, D_y and D_z which satisfy $P = D_x D_y D_z$. We refer to a valid permutation as a Topology or a Process Topology (MPI Cartesian Process Topology [12]) in this article. For example, a total of 28 Process Topologies exist for $P = 64$, three of which namely, $D_x \times D_y \times D_z = 4 \times 4 \times 4$, $D_x \times D_y \times D_z = 4 \times 16 \times 1$ and $D_x \times D_y \times D_z = 8 \times 4 \times 2$ are shown in Figure 4. These process topologies decide the sub-domain data dimensions of the hierarchy of grids. Typically and traditionally, the topology which minimizes the communication volume to be sent, created by the default `MPI_Dims_create()`, is chosen as the preferred topology for domain partitioning/mesh partitioning. We investigate the optimality of partitions returned by `MPI_Dims_create()` and whether *only* minimizing communication is *sufficient* to obtain optimal sub-domain dimensions for parallel GMG. Our work in [24] demonstrated the dependence of domain partitioning for single grids on cache-misses in computation and communication. Since parallel GMG is significantly more complex than a single grid and incorporates further stencil operators, the current research examines the efficacy of extending the model to parallel GMG.

DOI: 10.1002/cpe



(a) Dirichlet (Left, Bottom, Front - in red) and Neumann (Top, Back, Right - in blue) boundaries with outward normals on a unit cube



(b) A plane at Z=1/2 depicting Dirichlet (D), Neumann (N) boundary, unknowns (X), given problem (solid lines), halo layer for Neumann boundary (dotted lines).

Figure 5. Dirichlet-Neumann mixed Boundary Value Problem

3.3. Test Problem

Parallel Geometric Multigrid has been implemented for a 3-D *mixed Dirichlet-Neumann* boundary value problem on a unit cube to solve $-\nabla^2 u = -\frac{3\pi^2}{4} \sin\frac{\pi x}{2} \sin\frac{\pi y}{2} \sin\frac{\pi z}{2}$, which has a smooth solution $u(x, y, z) = \sin\frac{\pi x}{2} \sin\frac{\pi y}{2} \sin\frac{\pi z}{2}$. We use a vertex-centered finite-difference scheme in our implementation. Dirichlet boundary conditions ($u = 0$) are applied to the front, left and bottom faces of the cube whereas Neumann boundary conditions ($\frac{\partial u}{\partial n} = 0$) are present at the top, right and back faces. A halo layer is added to the Neumann boundaries as the Neumann boundaries are also considered as unknown values [1]. These halo layers need to be updated at each iteration according to the neighbouring point inside the physical sub-domain. Figure 5a shows the Dirichlet-Neumann boundaries on the domain $\partial\Omega$ specified on a unit cube. Figure 5b shows a cut through $Z = \frac{1}{2}$. Clearly, $\partial\Omega = \Gamma^D \cup \Gamma^N$ and $\Gamma^D \cap \Gamma^N = \phi$.

A *full 27-pt Restriction* scheme was implemented to transfer data to the coarser grid. This needs communication steps to make the *corner points* available to a process. Therefore, the communication pattern is different from that used in the smoother. The case of Trilinear Interpolation onto the finer grid is similar. At the finest grid level, the l_2 norm of the residual can be calculated after each V-cycle and the execution stops when the ratio of the current norm to the initial norm becomes less than a specified tolerance i.e. $\frac{||r_k||}{||r_0||} < TOL_N$. However, for performance analysis purposes, it is sufficient to fix the number of V-cycles. The levels are numbered from the highest to the lowest - starting at the finest grid (level L) to *level zero* corresponding to the *coarsest* grid. The coarsest grid problem can be solved till convergence (or a fixed number of iterations can be performed depending on the experiment). The iterative scheme used is *Jacobi* (or ω-Jacobi) with the option to change the weighting factor (ω) for both smoothing (fine and coarser grids) and solve (coarsest grid) operations. The general optimum values of ω for 1-D, 2-D, 3-D are $\frac{2}{3}$, $\frac{4}{5}$ and $\frac{6}{7}$, respectively (for pure Dirichlet boundaries) [8] but for our mixed Dirichlet-Neumann test case we found $\omega = 1$ to be optimal. Although the number of unknowns per process is equal, the problem is slightly load-imbalanced because the processes containing the upper, right and back Neumann boundary have to perform *slightly* more work than processes containing the lower, left and bottom Dirichlet boundaries. This is because in addition to the points to be updated, the former category of processes must also adjust the Neumann boundary before the values of the boundary points can be updated. However, such processes do not send/receive planes to/from other processes at the Neumann boundary. The number of iterations in the downward phase of the V-cycle is ν_1 (pre-correction smoothing) and ν_2 on the upcycle (post-correction smoothing). The complete V-cycle is written as $V(\nu_1, \nu_2)$ where typically we use $\nu_1 = \nu_2 = 3$.

DOI: 10.1002/cpe

4. OUR MODEL

We now discuss the derivation of our quasi-cache aware model that utilizes the cache-line length and the data access pattern in a 7-pt stencil. Our work in [24] exhaustively identified and quantified cache-misses as the single most important factor influencing domain partitioning of structured *single* level grids and thus, while extending the model, our focus remains on the cache-misses in the update/packing/unpacking of the dependent planes and the update of the Independent Compute kernel. We further elaborate on the super-set of factors influencing cache-misses directly or indirectly to shed light on the complexity of attaining truly optimal sub-domain dimensions for high performing partitions in Parallel Geometric Multigrid. It is to be noted that our high level model is different from the analytical models used to model multigrid cycle times and performance. Classical analytical models have attempted to model the execution timing and analyze the overall Weak Scaling using only the relaxation phase of semi-coarsening Multigrid with 1-D, 2-D and 3-D processor topologies/partitions [39, 40]. A baseline model with penalties in parallel settings has been formulated for modelling the cycle of Algebraic Multigrid [40, 41]. An analytical/empirical comparison for the execution times of an iteration of Newton-Multigrid and FAS (Full Approximation Scheme) has been carried out [42]. Performance prediction of Multigrid codes on large number of cores by benchmarking the code on a very small number of processes presents another alternative [43]. Most of these models take into account only the algorithmic characteristics and not the hardware parameters. Our model is different from these models in the sense that we take into account the cache-line characteristics but obtain a cache-oblivious result. Further, our model does not predict the execution timings but *predicts the topologies* which outperform the standard `MPI_Dims_create()` topology.

4.1. Deriving the Model

We consider an *elliptic, linear* PDE : $\nabla^2 u = f$. The discretized form is $Au = f$, with A being the *discretization* matrix and u representing the *vector* of unknowns. The key component of the smoothing phase of Multigrid consists of an iterative method such as the "out-of-place" weighted Jacobi ($\omega-$Jacobi) as shown in Equation (1) below :

$$v_{i,j,k} = (1-\omega)u_{i,j,k} + \omega(u_{i\pm1,j,k} + u_{i,j\pm1,k} + u_{i,j,k\pm1} + h^2 f_{i,j,k}) \tag{1}$$

The *Red Black Gauss-Seidel (RBGS)* updates "in-place", however, the observations that we make will still hold in principle (though with some quantitative differences). The advantage of RBGS is that the local working set consists of only two arrays which reduces the memory traffic and the cache conflict misses. The disadvantage of RBGS is that the red and black points are communicated separately and hence it requires twice the message exchanges as ω-Jacobi, resulting in twice the latency of messages as a penalty. The worst case for Neumann updates occurs at the top back right boundary process which has three Neumann boundaries. The cache misses for updating the three boundaries in the X, Y and Z-direction are $\frac{P_y P_z}{8}$, $\frac{P_x P_z}{8}$ and $P_x P_y$, respectively. It is to be noted that in this case both the read and write array are the same. However, the planes which undergo Neumann updates are not communicated to any other process and nor do they receive data from other processes. Thus, the packing/unpacking cost of such planes is zero. Since the cache-misses for packing *and* unpacking planes is more than that of Neumann updates for the plane, we can safely consider processes which send and receive data from other processes to derive the upper bound for cache misses. Such a process does not touch any boundary and sends/receives all six planes to/from neighbouring processes.

Assuming that the cache-line length is $\mathcal{L}$ bytes and the width of a `double` element is $\mathcal{D}$, the number of elements fetched from the memory to the cache are $\frac{\mathcal{L}}{\mathcal{D}}$. For example, for the system used here $\mathcal{L} = 64$ bytes and $\mathcal{D} = 8$ bytes and thus $\frac{\mathcal{L}}{\mathcal{D}} = 8$. Assuming a minimal number of cache-lines for accommodating the *six* different read streams (and *one* write stream) in Equation (1) and disregarding the loop invariant terms, namely, ω and h^2 (square of mesh spacing), the cache-misses for *update/packing/unpacking* of dependent planes (S_P) can be summarized in Table I. The total

DOI: 10.1002/cpe

Table I. PREDICTED CACHE-MISSES: Cache read/write/update misses for the dependent X, Y and Z-plane

PLANE	READ MISSES			WRITE MISSES		TOTAL
	Pack	Update	RHS	Unpack	Update	
Z-plane	P_xP_y	$5P_xP_y$	P_xP_y	P_xP_y	P_xP_y	$9P_xP_y$
X-plane	$\frac{P_yP_z}{8}$	$\frac{5P_yP_z}{8}$	$\frac{P_yP_z}{8}$	$\frac{P_yP_z}{8}$	$\frac{P_yP_z}{8}$	$\frac{9P_yP_z}{8}$
Y-plane	$\frac{P_xP_z}{8}$	$\frac{5P_xP_z}{8}$	$\frac{P_xP_z}{8}$	$\frac{P_xP_z}{8}$	$\frac{P_xP_z}{8}$	$\frac{9P_xP_z}{8}$

cache-misses of the Independent Computation (S_I) kernel can be calculated as :

$$S_I = (P_x - 2)(P_y - 2)(P_z - 2)(\frac{5}{8} + \frac{1}{8} + \frac{1}{8}) \tag{2}$$

where the $\frac{5}{8}$, $\frac{1}{8}$ and $\frac{1}{8}$ terms give the read misses in updating, write misses in updating and right hand side term read misses, respectively. The total cache misses ($S = S_I + S_P$) for the Independent Computation and dependent planes are :

$$S = \gamma(P_x - 2)(P_y - 2)(P_z - 2) + \alpha P_xP_y + \beta P_z(P_x + P_y) \tag{3}$$

where $\gamma = \frac{7}{8}$, $\alpha = 9$ and $\beta = \frac{9}{8}$ (see Table I) and are dependent on the computational kernel and the length of the cache-line.

We now consider a Multigrid V-cycle with $L + 1$ levels, where the level $k = 0$ denotes the coarsest grid and $k = L$ the finest grid. We assume the following: (i) the costliest operation is smoothing; (ii) the cost of grid transfer operators is proportional to the smooths; and (iii) the cost of a solve on the coarsest level may be neglected compared to the fine grid smoothing cost.

Let the cache-misses at level k be denoted by S_k where $S_k = S$ at level $k = L$ as in Equation (3). The sum of cache-misses at all levels (S_T) is bounded above by S_∞, where

$$S_T = \sum_{k=0}^{L} S_k < S_\infty = \sum_{k=0}^{\infty} S_k \tag{4}$$

Summing two separate infinite geometric series with common ratios $\frac{1}{8}$ and $\frac{1}{4}$ yields the expression for S_∞ as shown in Equation (5) below :

$$S_\infty = \frac{8\gamma}{7}(P_x - 2)(P_y - 2)(P_z - 2) + \frac{4}{3}(\alpha P_xP_y + \beta P_z(P_x + P_y)) \tag{5}$$

By considering $\frac{\partial S_\infty}{\partial P_x} = \frac{\partial S_\infty}{\partial P_y} = 0$ to minimize the total cache-misses with respect to sub-domain dimensions, we obtain $P_x = P_y$ for optimality (*condition one*) but this does not yield any information regarding P_z. Since we can generate all D_x, D_y and D_z such that $P = D_xD_yD_z$ because P_x, P_y and P_z are dependent on N and D_x, D_y, D_z, we exhaustively find that $D_z = 1$ minimizes S_∞. Thus, cache misses are minimized by maintaining a balance between the X/Y dimensions of the sub-domain and maintaining an unaltered unit stride dimension (theoretically). Further, the communication minimizing condition to minimize the surface area of planes implies $P_x = P_y = P_z$ (*condition two*). Taking the intersection of the cache-misses and communication volume minimization conditions yields a *strong* (common) condition : $P_x = P_y$. Further, when $S_I >> S_P$ in Equation (3), S is minimized with $P_x = P_y = P_z = \frac{N}{P^{\frac{1}{3}}}$. These two limits i.e. cache-miss dominated and communication volume dominated imply that $1 \leq D_{z_{optimal}} \leq P^{\frac{1}{3}}$ (assuming P is a perfect cube).

DOI: 10.1002/cpe

4.2. Pruning the Topology Search Space

Out of all the topologies which are possible, a small set can be examined keeping in mind the balance between the X and Y sub-domain i.e. P_x and P_y dimensions and the minimization of D_z. Thus, if P is a perfect square then $D_z = 1$ and $D_x = D_y = \sqrt{P}$ else we find $min(|D_x - D_y|)$ such that $D_x D_y = P$. To alleviate the effect of process placement we introduce a factor ρ that represents the deviation from the balanced pair of (D_x, D_y) i.e. assuming $P = 64$ and $\rho = 1$, we start with $D_x = 8$, $D_y = 8$ and $D_z = 1$ and then consider $(8 \times 2^1) \times (\frac{8}{2^1}) \times 1 = 16 \times 4 \times 1$ and $(\frac{8}{2^1}) \times (8 \times 2^1) \times 1 = 4 \times 16 \times 1$. For $\rho = 2$, we would *also* consider $(8 \times 2^2) \times (\frac{8}{2^2}) \times 1 = 32 \times 2 \times 1$ and $(\frac{8}{2^2}) \times (8 \times 2^2) \times 1 = 2 \times 32 \times 1$. In practice our experiments show that $\rho = 1$ is sufficient for obtaining optimal topologies. Assuming P to be a perfect cube and a power of two, the maximum number of candidates for optimal topologies is $(2^\rho + 1) \log_2 P^{\frac{1}{3}}$. Thus, with $\rho = 1$ and $P = 64$, theoretically we only have 6 candidates for optimal topologies.

4.3. Factors affecting sub-domain dimensions

To place the above model in its true context, we now discuss all of the factors influencing selection of sub-domain dimensions (assuming the data streams at no point are too large to fit into the cache). We discuss their impact in isolation and with respect to other factors. The discussion primarily brings out the need for a fine balance between multiple factors for optimizing the domain partitions and sheds light on their interplay. That is, that the problem of domain partitioning is much more subtle than *just* minimizing the communication.

Independent Compute (IC): This represents the sub-domain zone that does not need data from other processes for updating the mesh points. To update the solution at all mesh points contained in a plane, three planes i.e. the plane under consideration and the two planes above and below it are needed for a 7-pt stencil. The smaller the total size of these 3 planes, the more is the probability that they would fit into the Last Level Cache (LLC)/Cache-hierarchy. We define the quantity *Working Plane Set Size* (WPSS) as $3 \times (P_y + 2) \times (P_z + 2) \approx 3P_yP_z$ elements. The Independent Compute (IC) tries to minimize the WPSS by minimizing P_y but not P_z as the latter adversely affects the Vectorization and prefetch efficiency. Thus, it is preferable to decrease P_y rather than reducing P_z to decrease the overall WPSS. But when P_y is decreased (or D_y is increased as $P_y = \frac{N}{D_y}$) to some value $<< P_x$, it violates the cache-minimizing condition ($P_x = P_y$) which in turn leads to much higher communication and update times for the Y-plane that contains P_xP_z elements. Ideally, the MPI implementation should hide the entire communication cost behind the cost of executing the IC kernel. Practically, this is never the case as the computation and packing/unpacking of planes is carried out by the same thread or process (assuming no separate core for communication exists) that may result in switching between the two tasks : computation and packing/unpacking. This switching may also lead to an increased cache-contention and conflict misses as different data streams from computation and packing/unpacking are brought into the cache. Thus, decreasing P_y optimizes the execution time for the Independent Compute (IC) but increases the transmission times of the Y-plane. Further, when $P_y << P_x$, both the communication volume and cache-minimization conditions are violated.

Communication Volume (V): Minimizing communication implies $P_x = P_y = P_z$. For simplicity of discussion we assume the number of processes P is a perfect cube and thus for a domain of size N^3 this implies that the Cartesian process dimensions $D_x = D_y = D_z = \frac{N}{P^{\frac{1}{3}}}$. Since the default `MPI_Dims_create()` returns $D_{sx} \geq D_{sy} \geq D_{sz}$, the worst case growth rate of the Z-plane size becomes $P^{\frac{1}{3}}$ - leading to an increase in its communication and update time. Our model shows that $1 \leq D_{zoptimal} \leq D_{sz}$ and hence minimizing *only* the communication volume is *insufficient*.

Prefetch: For any topology, updating the Independent Computation (IC) kernel involves multiple contiguous data streams and thus prefetch reduces the latency. Since prefetch *usually* exploits spatial locality and assumes streaming fetches, maximizing P_z should increase the utilization of

DOI: 10.1002/cpe

the prefetched cache lines. The L1d cache has two hardware prefetchers in the Intel Sandy Bridge architecture (see Section 5). The first one, the Data Cache Unit (DCU) prefetcher, prefetches data in an ascending order from the address which has recently been loaded. Thus, assuming an address *A* has been loaded i.e. a cache-line is populated by `double` elements from A to A+7, the DCU prefetches the data from A+8 to A+15 in another cache line. The second prefetcher, the Instruction Pointer (IP)-based stride prefetcher, detects the stride in different load instructions and prefetches a cache line from the current address which is the sum of the current address and a stride. A stride of upto 2KB can be detected (or equivalently 256 `double` elements) [44]. The two prefetchers that bring data into the L3 cache are called the Streamer and the Spatial Prefetcher. The data *may not always* be brought into the L2 cache due to pending read/write misses. The Spatial Prefetcher fetches an additional 64 bytes into the unified L2 cache when a cache-line is brought into the L2. The Streamer monitors cache misses from the L1d, hardware prefetch requests from the L1d, L1i Instruction cache requests and can maintain upto 32 streams of ascending/descending data [44]. Thus, efficient prefetching demands maximizing the value of P_z. This condition violates the volume minimizing condition and increases the WPSS. With an increase in the WPSS, there is a danger that the three planes required for the update of a single plane *may not* fit into the Last Level Cache.

Least Recently Used (LRU) Eviction: This cache eviction policy replaces the cache-lines which have not been used recently. The distance between the mesh point $u_{i,j,k}$ and $u_{i,j+1,k}$ (see Equation (1)) is $P_z + 2$ and typically for a large enough P_z, they will belong to a different cache-line. Thus, larger the value of P_z, greater the clock cycles elapsed between re-accessing/re-using the mesh point $u_{i,j+1,k}$ to update $v_{i,j+1,k}$. This translates to having a higher probability for the eviction of this cache-line before it is re-used when P_z increases. This factor is different from all the other factors in the sense that it requires minimization of P_z to achieve maximum efficiency.

Planes Cache Misses: To minimize the cache-misses in packing/unpacking/updating planes our model indicates that $P_x = P_y$ and $1 \leq D_{zoptimal} \leq D_{sz}$. This indicates a partition which is *close* to a 2-D partition. As discussed above, when P_z is large, the Least Recently Used (LRU) policy used to evict cache-lines negatively affects the re-use of a cache-line. Further, increasing P_z increases the product $3P_yP_z$ (WPSS), possibly causing the combined size of 3 planes required to update a plane to become larger than the cache capacity. It is to be noted that the *Effective* WPSS (EWPSS) evaluates to $5P_yP_z$ as it involves 3 planes of the array u and one plane each from the arrays v and f (see Equation (1) Section 4.1). Further, when $P_z \geq 256$ `double` elements, the IP-based stride prefetcher of the L1d cache is rendered ineffective. Thus, when packing/unpacking/updating a Z-plane for a sub-domain that has $P_z \geq 256$, neither the DCU nor the IP-based stream prefetcher are effective - resulting in increased cache-misses.

Cache Line Utilization: We define this to mean the number of data elements used in a cache line which is fetched. Thus, $0 \leq$ *Cache Line Utilization (CLU)* ≤ 1. As an example consider the packing of an X-plane which has contiguous data (except for the near-to-boundary data points next to ghost/halo/boundary region). Consider a cache-line which fetches data elements far-away from the boundary. All the elements in this cache line will be used and hence the CLU = 1. But for a cache-line which has been prefetched/loaded containing the two ghost points, the CLU = $\frac{6}{8} = 0.75$. Thus, theoretically if $P_z \longrightarrow \infty$, almost all cache-lines will have a CLU = 1 while packing the X-plane. Same is the case with the Independent Compute and packing/unpacking the Y-planes. The worst CLU is seen with the Z-plane. Assuming $P_z > 8 = \mathcal{L}$, where $\mathcal{L}$ denotes the cache-line length, the minimum CLU = 0 and the maximum CLU = $\frac{1}{8} = 0.125$ for packing the Z-plane. Thus, whereas increasing P_z increases the CLU for the IC and X/Y planes, it decreases it for the Z-plane. Even when the data completely fits into the cache hierarchy, accessing elements from a different cache-line incurs a penalty as compared to accessing the data from the same cache-line.

DOI: 10.1002/cpe

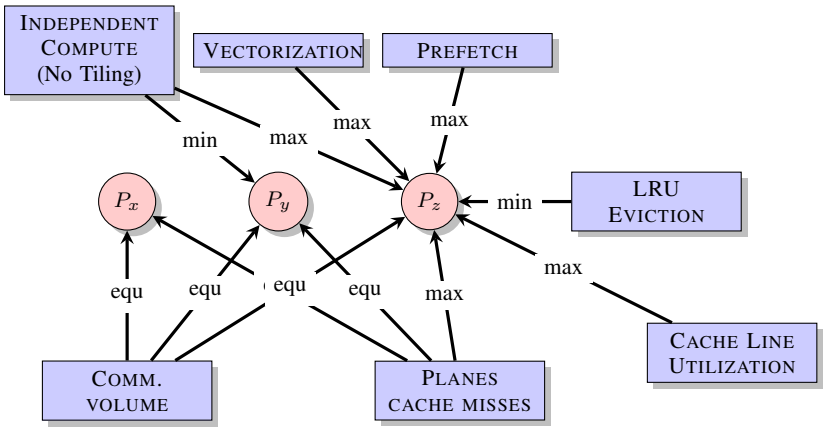


Figure 6. Factors affecting selection of sub-domain dimensions

Vectorization: is a combination of loop unrolling and packed SIMD instructions - 256 bit AVX instructions in case of Intel Sandy Bridge architecture. These work on streaming data and thus, maximizing P_z is a step in this direction. With Independent Compute (IC), the ghost data acts as bubbles in the data stream i.e. the ghost points are fetched as part of a cache line but are not used in the IC. Smaller the value of P_z and larger the value of P_y, greater will be the number of such junctions where ghost data forms a part of the cache line fetched. Thus, Vectorization demands a maximal P_z which again is in direct contradiction with the LRU policy discussed above and also deviates from the condition required for minimizing the communication volume.

The essence of our discussion on the multiple factors influencing sub-domain dimensions is summarized in Figure 6. The cache-misses minimization condition, particularly maximizing P_z, as derived in our model in Section 4.1 is re-enforced by many factors, namely, Independent Compute, Vectorization, Prefetch, Cache Line Utilization, Plane Cache Misses but is opposed by the LRU Eviction policy and the Communication Volume minimization condition. The Independent Compute opposes the equalization condition imposed on P_y by the Communication Volume and the Plane Cache Misses condition but minimizing P_y i.e. maximizing D_y may increase the communication costs as $|D_x - D_y|$ increases. The least constrained sub-domain dimension is the X-dimension i.e. P_x which needs to satisfy the condition $P_x = P_y$ as dictated by the Communication Volume and Plane Cache Misses conditions. We further emphasize that the major benefit in deviating from a cubic sub-domain shape can be attributed to the decreasing Z-plane packing/unpacking/updating cache-miss costs. At the same time, the increasing cost of communication volume cannot be neglected when the unit-stride dimension (i.e. P_z) grows - even though the performance can increase due to the Vectorization, Prefetch and Cache Line Utilization (CLU) factors. Table II shows the trade-off between increasing communication volume and decreasing cache-misses of the Z-plane. As the unit-stride dimension increases, the Z-plane cache-misses decrease at the expense of increasing communication volume. Thus, the decrease in cost due to the Z-plane cache-misses (most significant), improved Vectorization, Prefetch, and Cache Line Utilization must outweigh the cost of increased communication volume along with the extra cache-misses due to the LRU cache-eviction policy.

5. EXPERIMENTAL TESTBED

Our first experimental testbed is the ARC2 (Advanced Research Computing) facility at the University of Leeds. It is a CentOS 6 Linux based computing facility with a total of 3040 cores. Each compute node consists of 2 Intel Xeon E5-2670 Sandy Bridge processors and there are a total of 380 nodes housed in 190 blades. Each processor has 8 physical cores (base frequency 2.6GHz and Turbo 3.2GHz) with hyperthreading and Turbo boost turned off. Each socket has 16GB of

DOI: 10.1002/cpe

Table II. TRADE-OFF: Theoretical Communication Volume Vs Predicted Z-plane Cache-Misses

SUB-DOMAIN DIMS.			Z-PLANE			COMMUNICATION	
P_x	P_y	P_z	Size	Cache-misses	$n = 64$	Volume	$n = 64$
n	n	n	n^2	$9n^2$	36864	$6n^2$	24576
$\frac{n}{\sqrt{2}}$	$\frac{n}{\sqrt{2}}$	$2n$	$\frac{n^2}{2}$	$\frac{9n^2}{2}$	18432	$6.65n^2$	27266
$\frac{n}{\sqrt{4}}$	$\frac{n}{\sqrt{4}}$	$4n$	$\frac{n^2}{4}$	$\frac{9n^2}{4}$	9216	$8.5n^2$	34816
$\frac{n}{\sqrt{8}}$	$\frac{n}{\sqrt{8}}$	$8n$	$\frac{n^2}{8}$	$\frac{9n^2}{8}$	4608	$11.56n^2$	47364

main memory (UMA - Uniform Memory Access), making a total of 32GB per node. Two sockets per compute node create a NUMA region (Non-Uniform Memory Access). Due to 256-bit AVX (Advanced Vector Instructions) each processor delivers $2.6 \times \frac{256}{8\times4} = 166.4$ SP GFLOPS (332.8 GFLOPS per node) and 88.2 DP GFLOPS (166.4 GFLOPS) per processor. Each processor has two bidirectional QPI (Quick Path Interconnect) links of 16GB/sec capacity [45].

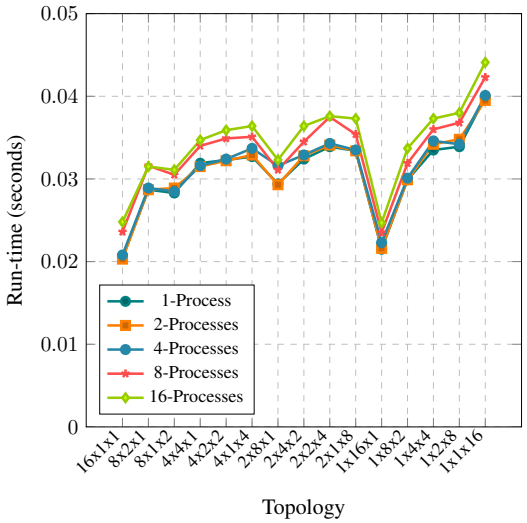
Each core has three levels of cache memory. The L1 instruction cache (L1i) contains a micro-(μ) Decoded Instruction cache (DIcache) which provides decoded instructions at a lower latency. Both the L1i and L1d are 32KB. The L2 cache is an exclusive but unified cache (both data and instructions) of size 256KB. The L3 cache is a shared cache of size 20MB per processor (i.e. 8 cores). Further, the L3 cache is an inclusive cache in the sense that it contains the data contained in the L1 and L2 level cache. All the caches have a cache-line size of 64 bytes, associativity of 8 and use the write-back mechanism. The hierarchy of compute/network elements is as follows : each node is contained in a blade, there are 16 blades in a shelf and 4 shelves are contained in a rack. We use the Intel compiler suite Intel 16.0.2 and OpenMPI 1.6.5 for all our experiments on ARC2.

Our second test platform is the recently added (February 2017) High Performance cluster ARC3 at the University of Leeds. It has a total of 4056 cores of Intel Xeon Broadwell E5-2650v4 processors (12 cores per processor or CPU, 2 processors per node, 24 cores and 128 GB RAM per node in 8 modules of 16 GB each at 2.4GHz) and a total of 22 Tb of RAM. The interconnect is a Mellanox FDR Infiniband operating at 56 Gbits/sec with 2:1 blocking. The Operating system is CentOS 7 with support for AVX2 instructions. Each core delivers 35.2 SP GFLOPS making it a total of 422.4 SP GFLOPS per processor/CPU. The cache memory characteristics per core are exactly the same as that in ARC2 mentioned above. We use the Intel 17.0.1 and GNU 6.3.0 compilers for our experiments on ARC3 in conjunction with OpenMPI 2.0.2 and Mvapich2/2.2.

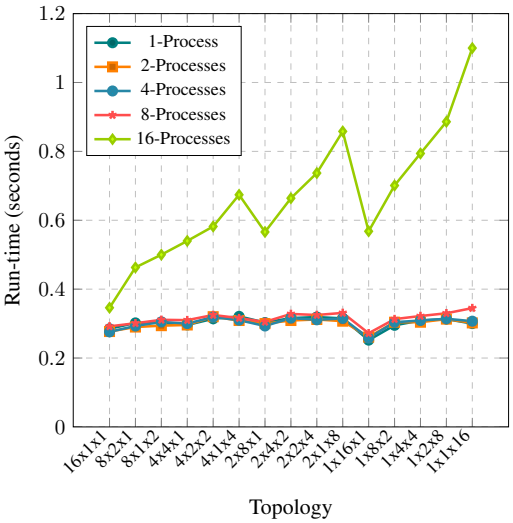
6. EXPERIMENTAL RESULTS

We first carry out a set of performance evaluations using various topologies on a single node, followed by multiple nodes. Our sequence of experiments is as follows : (i) Evaluate and analyze the Independent Compute (IC) for increasing grid sizes and process numbers for characterizing the shared L3 cache behaviour on ARC2 (ii) Optimize the IC using established techniques in order to see if different partitions can yield the same performance on ARC2 (iii) Evaluate and analyze plane communication times for increasing grid sizes with two different intra-node process placement policies on ARC2 (iv) Validate the inferences from our model by combining the IC and plane communication times on ARC2 (v) Evaluate a light-weight, dynamic, tiling heuristic against exhaustive tiling and compiler switches for on-node Parallel Geometric Multigrid on ARC2 and ARC3 (vi) Present performance results for multiple nodes on ARC2 and ARC3 (vii) Observe the relationship between the frequency and size of Z-planes passing through a hierarchy of networking

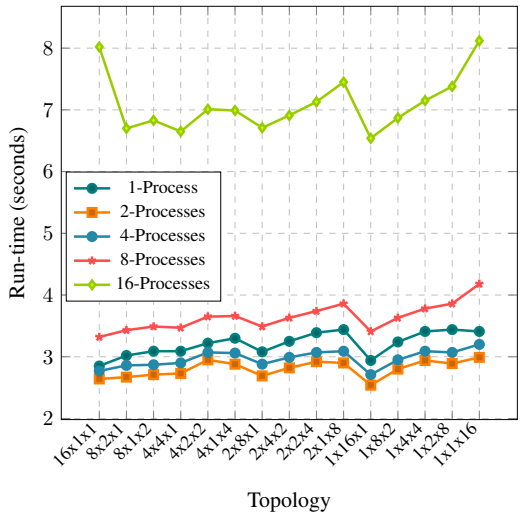
DOI: 10.1002/cpe



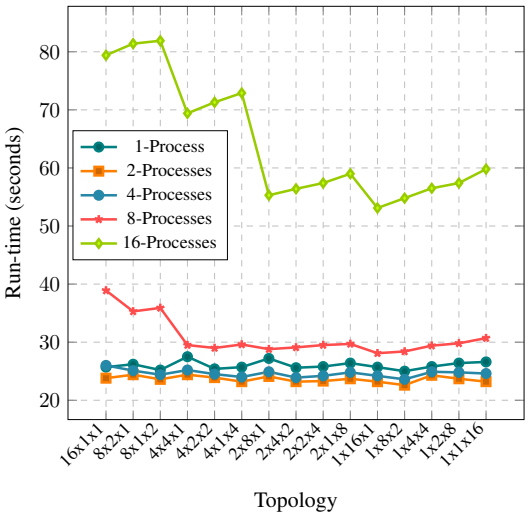
(a) IC run-times for 16384 cells per core and 384 KB working set per process



(b) IC run-times for 131072 cells per core and 3 MB working set per process



(c) IC run-times for 1048576 cells per core and 24 MB working set per process



(d) IC run-times for 8388608 cells per core and 192 MB working set per process

Figure 7. Weak Scaling Independent Compute (IC) for P=1,2,4,8 and 16 processes with $\frac{64^3}{16}$, $\frac{128^3}{16}$, $\frac{256^3}{16}$ and $\frac{512^3}{16}$ cells per core (with no communication) to measure impact of shared Last Level Cache per-socket contention on execution times on ARC2

elements and optimal partitions on ARC2 (viii) Present Weak Scaling and Strong Scaling results for ARC2 and ARC3.

6.1. Single Node

With the growing number of cores in a single node, it becomes important to characterize the intra-node behaviour of the application. Further, in a shared cluster, the traffic generated by multiple user applications does not affect the on-node communication latencies. A single node of our cluster ARC2 consists of 8 cores per-socket with a total of 2 sockets. The default scheduling policy is `--bind-to-core --bysocket` which maximizes the bandwidth per core (the first

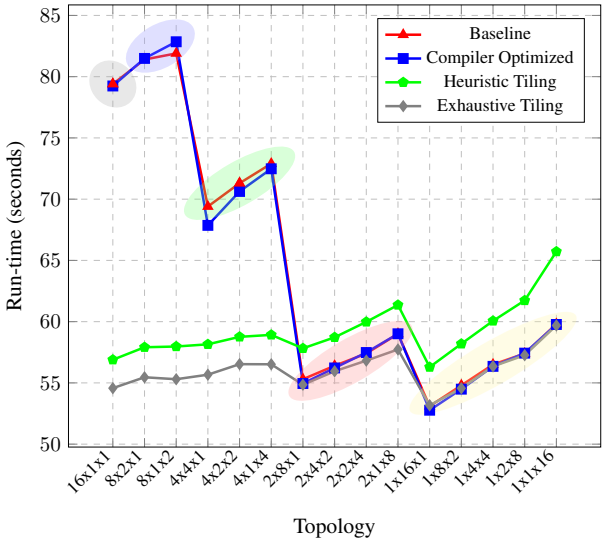
DOI: 10.1002/cpe

process is assigned to core 0 in socket 0, the second process is allocated to core 0 in socket 1, the third process is allocated core 1 in socket 0 and so on). With OpenMPI 1.6.5, `mpiexec --report-bindings` displays the default binding in the standard error file. As the number of processes increase, the contention for the LLC (20 MB/socket) and main memory (16 GB/socket) per socket increases. To study this behaviour, we Weak Scale a problem of given size but with *no communication*. Thus, the problem size per process remains constant as we increase the number of processes. The average execution time of n processes should ideally remain constant as each core executes a same-sized but completely independent problem. In particular, each core updates the Independent Computation (IC) zone of a sub-domain using a 7-pt stencil. This is equivalent to performing smoothing operations on the IC at the finest grid level *only*.

Figure 7 shows the *maximum* execution times of the Independent Compute kernel on any process, with each core (or process) having a sub-domain of size $\frac{64^3}{16}$ (Figure 7a), $\frac{128^3}{16}$ (Figure 7b), $\frac{256^3}{16}$ (Figure 7c) and $\frac{512^3}{16}$ (Figure 7d), respectively. If we run a single process on a 64^3 domain within a 16-core node, then that process handles a sub-domain of size $\frac{64^3}{16}$. If we run 8 processes on the 16-core node, then each process handles a sub-domain of the same size i.e. $\frac{64^3}{16}$. Similar is the case for other domains i.e. 128^3, 256^3 and 512^3. Further, the shape of the sub-domain varies with the different topologies obtainable with $P = 16$. For example, with a topology of $16 \times 1 \times 1$ (and domain 64^3), a sub-domain having dimensions $P_x \times P_y \times P_z = 4 \times 64 \times 64$ is produced, whereas the topology $4 \times 4 \times 1$ produces a sub-domain having shape $16 \times 16 \times 64$. Since there is no communication between processes and each core operates independently on given equal sized sub-domains, the time for the Independent Compute should *ideally* be equal for all processes, irrespective of the number of cores (or processes) we utilize. However, in practice, this is *not true* as increasing the process count leads to an increase in the contention for shared resources such as the Last Level Cache and main memory per-socket. The following discussion elaborates how with an increasing process count, the contention for the above-mentioned shared resources leads to a deterioration of performance within a node, even when the processes operate on independent sub-domains.

With a *Working Set Size* (WSS) of approximately 384 KB i.e. 3 arrays of type `double` with 16384 elements (=$\frac{64^3}{16}$) each, the total WSS remains less than the shared *Last Level Cache* (LLC) per core i.e. 2.5 MB/core. It can be seen from Figure 7a that the characteristics of the curve indicate the unchanging behaviour of the topologies as the process count is increased from one to sixteen. Further, the heavy overlapping indicates that the execution times are approximately equal even when the LLC and shared memory contention increases with an increasing process count. This is expected as the *size(WSS-per-process)* $<$ *size(LLC-per-core)*. An anomaly is that the execution time of a single process is more than that of two and four processes. A plausible reason could be that *CentOS* and *OpenMPI 1.6.5* do not *pin* the single process [46] to a single core. But since we never run a single process per-node in the actual application, we do not investigate this any further. Figure 7b shows the same experiment but with a domain size of 128^3 and 131072 cells/core creating a WSS of $\approx$ 3MB per core. With a per socket shared LLC of 20 MB and with 8 processes per node (i.e. 4 per socket due to the binding `--bind-to-core --bysocket` configuration), the combined WSS of 4 processes is small enough to fit into the per socket LLC. Thus, Figure 7b shows no sudden jumps in the execution times upto 8 processes. But with 16 processes, the cumulative WSS of 48 MB exceeds the LLC and the penalty of accessing the main memory can be clearly seen in the baseline implementation running 16 processes. Figures 7c and 7d show the Weak Scaling of Independent Computation kernel with no communication for domains of sizes 256^3 and 512^3, respectively. In both the cases the WSS per process exceeds the shared LLC per core. The change in the shape of the curve from eight to sixteen processes in Figure 7c shows that the execution timings need not necessarily remain fixed with respect to each other i.e. the execution pattern of topologies in going from a smaller process count to a higher process count may not follow similar curves. A similar change can be seen in Figure 7d for the plot of 4, 8 and 16 processes. It is to be noted that even with the baseline implementation,

DOI: 10.1002/cpe



(a) Execution times

Topology	WPSS
16x1x1	786432
8x2x1	393216
8x1x2	393216
4x4x1	196608
4x2x2	196608
4x1x4	196608
2x8x1	98304
2x4x2	98304
2x2x4	98304
2x1x8	98304
1x16x1	49152
1x8x2	49152
1x4x4	49152
1x2x8	49152
1x1x16	49152

(b) WPSS for Topologies

Figure 8. Baseline/naive implementation, Compiler optimized run-times with `-O3 -xHOST -ip -ansi-alias -fno-alias`, Heuristic square tile for X/Y dimensions (based on Rivera and Tseng [28] square tiles), Exhaustive Tiling for domain of size 512^3 and 16 processes on ARC2, default `MPI_Dims_create()` $= 4 \times 2 \times 2$

there are many topologies at each domain size which outperform the sub-domain created by the standard topology i.e. the topology returned by `MPI_Dims_create()` (henceforth referred to as `MDC` or the *standard* topology). From the results it can be seen that process topologies which have a higher value of D_y outperform other topologies in executing the Independent Computation kernel (IC) with growing data size as predicted in Section 4.3 (see Independent Compute (IC)). The only exception to this is the execution times of a $\frac{128^3}{16}$ sub-domain with 16 processes (see Figure 7b). In this case, the topologies having $D_x > D_y$ outperform other topologies.

The performance of the topologies can be enhanced by using techniques such as optimal compiler switches, cache tiling, Vectorization with appropriate alignment and exclusive SIMD directives. We compare the execution times of various topologies with a domain size of 512^3 with various optimizations. The objective is to optimize the bulk of computation i.e. the Independent Computation kernel of the sub-domain. The results are presented in Figure 8 where the tiled code generally performs better than the code exploiting optimal compiler switches. We create a light-weight, run-time, space tiling heuristic based on the size of the LLC per core and a working set (WSS) of *three equal sized* arrays. Following the work of Rivera and Tseng [28], we assume that square tiles should be used in the X and Y direction i.e. $CX = CY$. Thus, for a single 3-D array having $CX = CY = k$ and an un-cut unit-stride dimension $P_z + 2$, the number of elements should equate to :

$$k^2 \times (P_z + 2) = \frac{2.5}{3} \times \frac{1024 \times 1024}{8}$$

yielding

$$k = \left\lceil \sqrt{\frac{104857.6}{(P_z + 2)}} \right\rceil$$

Although with exhaustive tiling we are able to find tile sizes CX and CY (with $CX \neq CY$ for majority of the topologies) which outperform the heuristic that we create, the tile iteration space becomes huge and thus becomes a time consuming process. The task of optimizing stencils depends

DOI: 10.1002/cpe

heavily upon the hardware parameters like cache sizes, cache line size, prefetch policies, stencil order, data size, and the algorithm employed etc. [47]. The range of relative error between the execution times found using the heuristic and the optimal tile size is $\approx 4-10\%$. An observation is that *most* process topologies outperform the `MDC` topology in the cases of exhaustive and heuristic tiling. Specifically, the compiler optimized version of $1 \times 16 \times 1$ outperforms the Independent Computational kernel created by the standard topology by $\approx 25.2\%$ (see Figure 8a). *Vectorization* and a *32-byte* alignment for the Intel Xeon Sandy Bridge processor on ARC2 produced negligible effects.

To understand the difference in the run-times of the baseline version of the different sub-domains, we group the various process topologies on the basis of the *Working Planes Set Size* (WPSS). The WPSS for a 7-pt stencil is the number of elements in the three planes which are required to update a single plane. Thus, the total elements (`double` type) contained in three planes are $3 \times (P_y + 2) \times (P_z + 2) \approx 3 \times P_y \times P_z$. We cluster the topologies having the same WPSS into a single group (see Figure 8b). To compare the execution times of the IC kernel of two topologies T1 and T2, their WPSS is computed. The WPSS of both T1 and T2 may or may not fit into the $\frac{LLC-per-core}{3}$, where the denominator indicates that the LLC is assumed to be equally divided between three arrays namely, the write array (v), the read array (u) and the array representing the source (or RHS) term (f) (see Equation (1) in Section 4). We can distinguish between at least three cases :

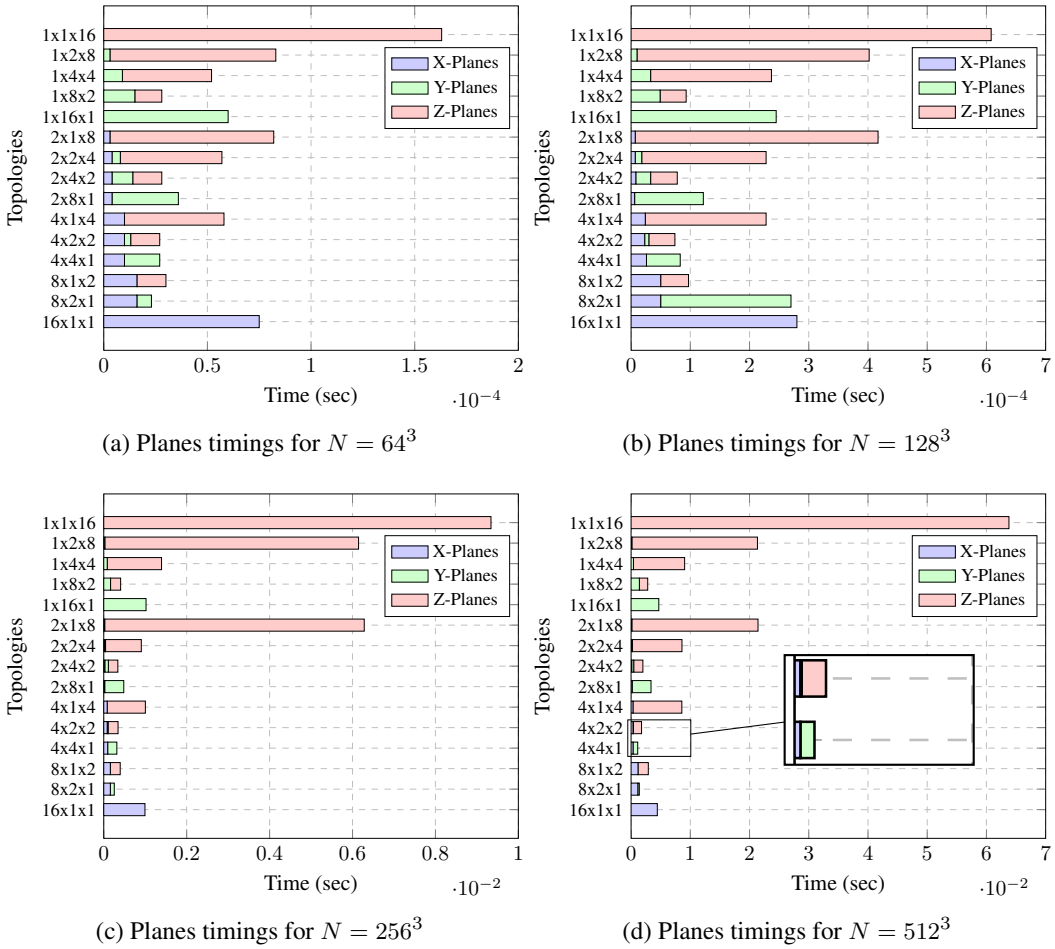
1. *WPSS(T1)* $\neq$ *WPSS(T2)* and both $> \frac{LLC-per-core}{3}$: In this case, more weight is given to the WPSS as compared to the Vectorization factor (i.e. the length of P_z - the larger the better).
2. *WPSS(T1)* = *WPSS(T2)* and *WPSS* $> \frac{LLC-per-core}{3}$: The topology with a higher value of P_z outperforms the other.
3. *WPSS* $< \frac{LLC-per-core}{3}$: Here the demarcation between the performance of topologies becomes blurred and needs more investigation.

The topology $16 \times 1 \times 1$ in Figure 8a deviates from the first rule above and outperforms topologies $8 \times 2 \times 1$ and $8 \times 1 \times 2$ despite having a larger WPSS. Empirically, it is very difficult to exactly determine the working set brought into the different cache levels but still the rules formulated above provide a substantially accurate insight into the relative baseline performance of various topologies.

Figures 9a, 9b, 9c and 9d show the individual maximum time for sending *and* receiving X/Y/Z planes to/from neighbouring processes within an SMP (Symmetric Multiprocessor) of ARC2 for $P = 16$ and increasing plane sizes. The communication times of topologies $16 \times 1 \times 1$, $1 \times 16 \times 1$ and $1 \times 1 \times 16$ form the basis of the following observations :

1. For the same sized X, Y and Z planes, the Z-plane takes the maximum amount of time (as indicated in Table I). For example, topologies $16 \times 1 \times 1$, $1 \times 16 \times 1$ and $1 \times 1 \times 16$ all pass equal-sized inter-socket X, Y and Z planes. At $N = 64, 128$, same sized Z-planes take about `3x` the time as compared to the X/Y planes. At $N = 256, 512$, they take `9x` and `12x` the time, respectively. Our predictions in Table I show that the Z-plane communication is `8x` more expensive than its siblings.
2. At $N = 64, 128$ the same sized X-planes on an average take a factor of `1.2` more time than the Y-planes but at $N = 256, 512$ the Y-planes take a factor of `1.03` more time than the X-planes. Our predictions show that same sized X and Y-planes should take the same amount of time (see Table I in Section 4).
3. When the surface area of planes is quadrupled, the communication times of inter-socket X planes increases by factors of `3.3-4.5`, the inter-socket Y-planes by `4-4.68` whereas the factor is between `3.73-15` for the inter-socket Z-planes. These ranges of times for the X/Y planes are expected but the `15x` jump in timings from $N = 128$ to $N = 256$ for the Z-plane is much greater than the expected theoretical `4` times increase.

DOI: 10.1002/cpe



(a) Planes timings for $N = 64^3$

(b) Planes timings for $N = 128^3$

(c) Planes timings for $N = 256^3$

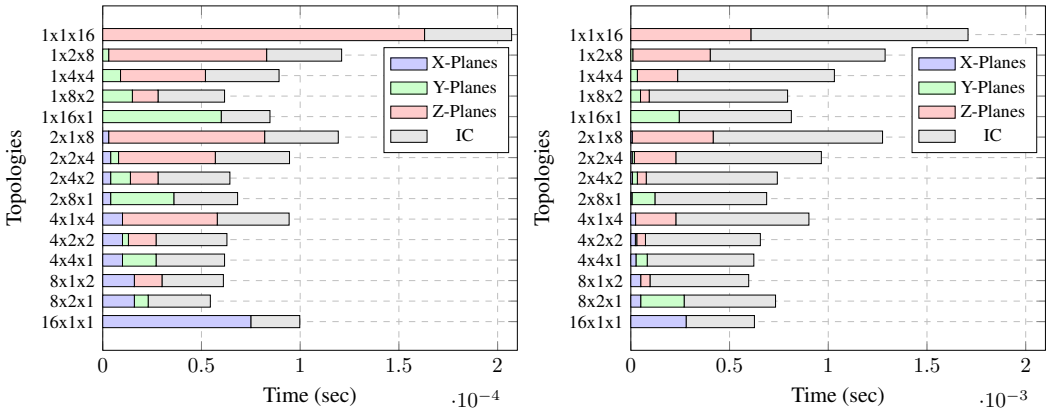
(d) Planes timings for $N = 512^3$

Figure 9. Maximum average time to send and receive X/Y/Z planes separately within a 16-core node for topologies (`--bind-to-core -bysocket`) using Intel 16.0.2 and OpenMPI 1.6.5 on ARC2, default `MPI_Dims_create()` = $4 \times 2 \times 2$

We consider the topology $1 \times 1 \times 16$ to understand the abnormal increase in the communication timings of the Z-plane. The topology $1 \times 1 \times 16$ produces $P_z = 8$ for $N = 128$ and $P_z = 16$ for $N = 256$. The distance between any two adjacent mesh points in the Z-plane ($Z_{adj} = P_z + 2$) then becomes 10 and 18, respectively. The L1 streaming hardware prefetcher (DCU - Data Cache Unit) fetches only *one extra* cache-line with ascending addresses. Thus, effectively two cache lines or $\frac{64+64}{8} = 16$ `double` elements are fetched. With $Z_{adj} = 10$, although only one mesh point is utilized per cache-line, there is no cache-miss to access the second line due to the prefetch mechanism, however with $Z_{adj} = 18$, a cache-miss occurs at every second access. As an approximation for $Z_{adj} = 10$ there is a cache-miss after accessing every 5 or 6 elements and for $Z_{adj} = 18$ a cache-miss on accessing every element even with prefetching. As discussed in Section 4.3, this illustrates how prefetching affects the communication times for a particular choice of sub-domain dimensions.

In summary, the majority of timings for various topologies can be explained and compared on the basis of the following : (i) *Size* of the plane being passed (ii) *Number* of planes being exchanged (iii) *Region of movement* of plane i.e. intra-socket or inter-socket and (iv) *Cache-misses* during packing/unpacking of plane (depends on whether it is an X/Y/Z plane). The timings in Figures 9a, 9b, 9c and 9d do not exactly reflect the actual timings in the real scenario as the X/Y/Z planes in these simulations are being passed and received *separately* i.e. a single type of

DOI: 10.1002/cpe



(a) Combined IC and plane timings, $N = 64^3$ (b) Combined IC and plane timings, $N = 128^3$

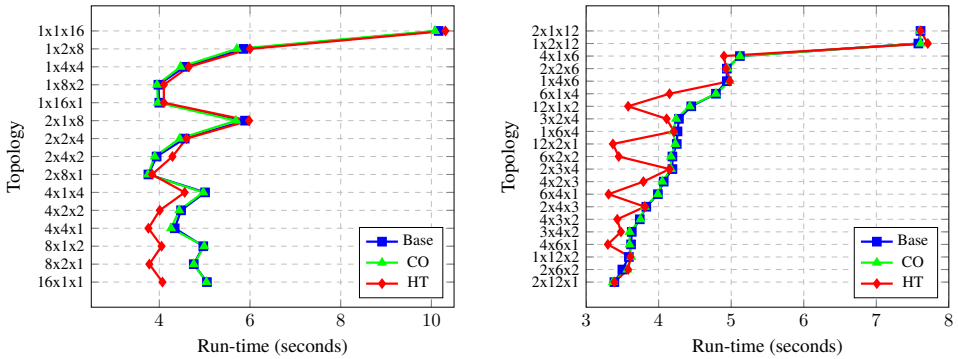
Figure 10. Relative plane communication and Independent computation times for $N = 64$ and $N = 128$ with $P = 16$ ((`--bind-to-core -bysocket`)) using Intel 16.0.2 and OpenMPI 1.6.5 on ARC2, plane update execution times are not shown, default `MPI_Dims_create()` = $4 \times 2 \times 2$

plane (either X or Y or Z) is being handled separately. In a real application, *all* types of planes are passed simultaneously depending on the implementation. Thus, the latter should produce an increased number of simultaneous send/receive requests per process and hence deteriorate the total communication timings further.

Another intra-node process binding scheme, namely `--bind-to-core --bycore`, fills up a single socket with increasing ranks instead of a round-robin policy of utilizing sockets. The key idea is to reduce the cost of communication by increasing the possibility of neighbouring ranks residing on the same socket. For the topologies $16 \times 1 \times 1$, $1 \times 16 \times 1$ and $1 \times 1 \times 16$ and with increasing plane sizes, the communication time of X planes increases by a factor $4-6$, for Y planes by $3.5-5$ and Z planes by $3.5-12$. The abnormal jump by a factor of 12 for Z-planes occurs when the planes size increases from 128×128 to 256×256 elements. Thus, with Z_{adj} changing from 10 to 18, the L1 Streaming hardware prefetcher (Data Cache Unit) is unable to prefetch the cache-line which contains the next mesh point, resulting in a miss for every mesh point when $Z_{adj} = 18$. For equal sized X/Y/Z planes, the communication of the Z plane is a factor $6-15$ more expensive than X/Y planes when the binding policy is `--bind-to-core --bycore`.

Figures 10a and 10b show the relative/combined times for the Independent Compute (IC) and communication of planes. At $N = 64$ and $P = 16$ (16-core node), the communication costs in almost every topology exceeds the IC cost, clearly indicating that the communication cannot be completely hidden within computation at coarser levels of a multigrid solver. We would expect the communication to remain completely hidden within computation at finer grid levels as shown by the larger computation times in Figure 10b but this overlap is completely governed by the OpenMPI implementation and the underlying hardware. These two figures further show that a topology which has the least IC computation time may not yield the optimal partition as it may have a higher communication time as compared to other topologies. For example the topology $1 \times 16 \times 1$ has the least IC execution time at $N = 64^3$, $P = 16$ as can be seen in Figure 10a but its total execution time (disregarding overlap) is more than a topology like $4 \times 4 \times 1$ or $4 \times 2 \times 2$. This observation lends support to our model as the latter topologies have a much more balanced D_x and D_y. Figure 11a shows the Baseline (Base), aggressively Compiler Optimized (CO) (`-O3 -xHOST -ip -ansi-alias -fno-alias`) and a Heuristically Tiled (HT) version of Parallel Geometric Multigrid for the largest problem that we could fit into a *16-core* node of ARC2 i.e. approximately 8 million cells/core (or 0.13 billion dof). It can be noted that a topology like $4 \times 4 \times 1$ outperforms the standard topology $4 \times 2 \times 2$ in all three versions even though the former sends/receives a

DOI: 10.1002/cpe



(a) Topology Run-times for $P = 16$, $N = 512$, Levels = 6, Coarsest iterations = 100, 5 V(3,3) cycles, Intel 16.0.2, OpenMPI 1.6.5, ARC2, default `MPI_Dims_create()` = $4 \times 2 \times 2$

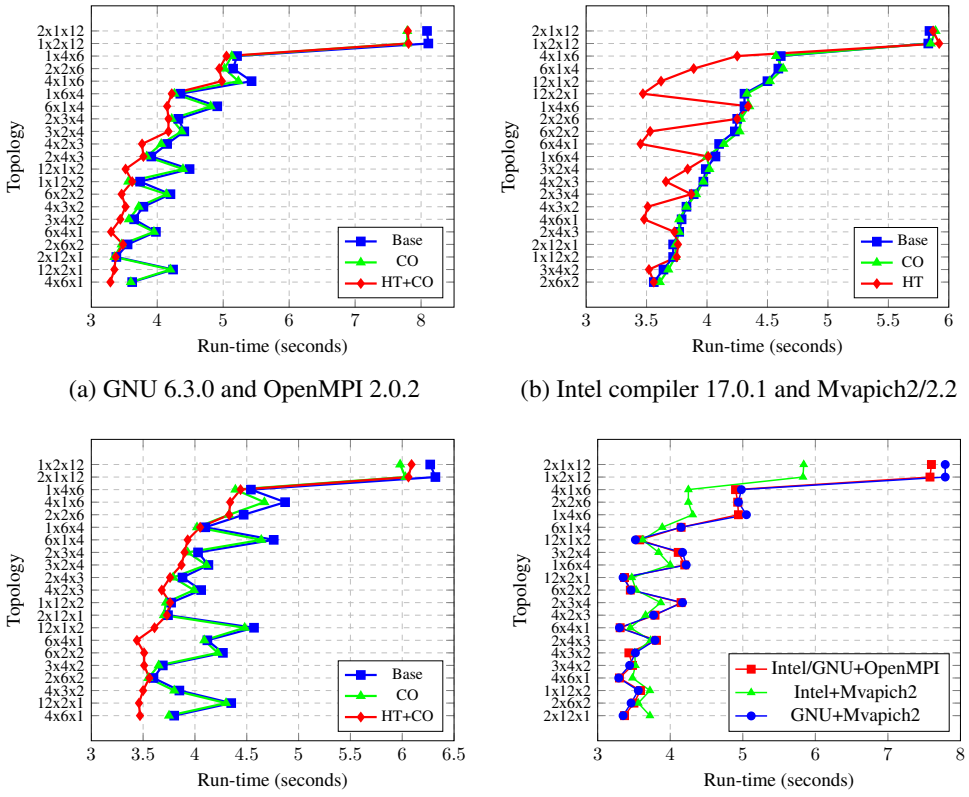
(b) Topology Run-times for $P = 24$, $N = 576$, Levels = 5, Coarsest iterations = 400, 5 V(3,3) cycles, Intel 17.0.1, OpenMPI 2.0.2, ARC3, default `MPI_Dims_create()` = $4 \times 3 \times 2$

Figure 11. Intranode execution times of Parallel Geometric Multigrid using Baseline (Base), aggressive Compiler Optimization (CO) and Heuristically Tiled (HT) versions on ARC2 and ARC3

maximum of 4 inter-socket Y-planes per process that are two times larger than the Y-planes of the latter topology, which sends/receives only intra-socket Y-planes. The reason for this is that the cost of packing/unpacking the Z-plane for $4 \times 4 \times 1$ is zero whereas the standard topology has to pack/unpack/communicate the high cost Z-plane (see the magnified section of Figure 9d). Figure 11b shows the execution time of Parallel Geometric Multigrid on a single node of the ARC3 cluster using the Intel 17.0.1 compiler and OpenMPI 2.0.2 with approximately 8 million cells/core (or 0.19 billion dof). A significant difference between OpenMPI 1.6.5 and OpenMPI 2.0.2 implementation is the change of the shared memory module (`-sm` module) to the `-vader` module, the latter offering performance benefits over the former. With 24 cores, the Heuristically Tiled version of $6 \times 4 \times 1$ and $4 \times 6 \times 1$ outperform the `MPI_Dims_create()` topology of $4 \times 3 \times 2$. The WPSS of $4 \times 6 \times 1$ is less than that of $6 \times 4 \times 1$ and thus is the major factor in contributing to the improved performance of the former within a single node (as process placement effects within a single node can be ruled out). It may be noted that although having a large P_z offers an enhanced opportunity for Vectorization, it decreases the probability of the data remaining in the cache before that data is accessed again due to the Least Recently Used (LRU) eviction policy (see Figure 6). The intranode execution trends of topologies on ARC2 and ARC3 show that our predictions and the behaviour of topologies are consistent *across different hardware*.

Figures 11b, 12a, 12b and 12c show the effect of different combinations of compilers and MPI implementations using a combination of Intel 17.0.1 + OpenMPI 2.0.2 (henceforth called I17O2), GNU 6.3.0 + OpenMPI 2.0.2 (henceforth called G6O2), Intel 17.0.1 + Mvapich2/2.2 (henceforth called I17M2) and GNU 6.3.0 + Mvapich2/2.2 (henceforth called G6M2), respectively, on a domain of size 576^3 on a single node of ARC3. For each of these, three variations in the form of Base version for Intel 17.0.1 (`-O2`) and GNU 6.3.0 (`-O2`), aggressive CO for Intel 17.0.1 (`-O3 -xHOST -ip -ansi-alias -fno-alias`) and GNU 6.3.0 (`-O3 -march=native`) and HT were tested. Since Heuristic Tiling alone provided negligible benefits *without* aggressive compiler based optimization with GNU 6.3.0, it was coupled with the latter (i.e. HT+CO - see Figure 12a and 12c). The curves in I17O2, I17M2, G6O2, and G6M2 are a characteristic of the compiler which is used. The experiments with the Intel 17.0.1 compiler, irrespective of the MPI implementation version, showed negligible difference between the Base version and the CO version while showing the best timings with HT alone. The optimal timings were obtained with a combination of HT+CO with GNU 6.3.0. Overall, the optimal execution timings were obtained with topologies $D_x \times D_y \times D_z = 4 \times 6 \times 1$ and $D_x \times D_y \times D_z = 6 \times 4 \times 1$ - the topologies

DOI: 10.1002/cpe



(a) GNU 6.3.0 and OpenMPI 2.0.2

(b) Intel compiler 17.0.1 and Mvapich2/2.2

(c) GNU 6.3.0 and Mvapich2/2.2

(d) Minimum execution times

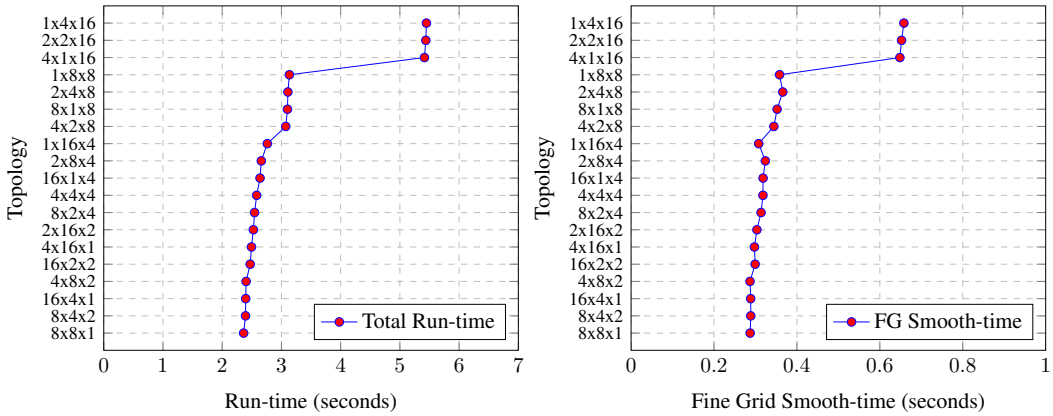
Figure 12. Topology Run-times for $P = 24$, $N = 576$, Levels = 5, Coarsest iterations = 400, 5 V(3,3) cycles and the minimum run times for various combinations of compilers and MPI implementations on ARC3, default `MPI_Dims_create()` = $4 \times 3 \times 2$

which are predicted with our model. For every version (Base, CO, HT, HT+CO) of I17O2, I17M2, G6O2, and G6M2, one of the predicted topologies i.e. either $4 \times 6 \times 1$ or $6 \times 4 \times 1$, outperformed the default `MPI_Dims_Create()` (MDC) topology of $4 \times 3 \times 2$. The performance gains for the versions using Mvapich2/2.2 i.e. I17M2 (1.70%) and G6M2 (1.71%) were smaller as compared to versions using OpenMPI 2.0.2 i.e. I17O2 (3.79%) and G6O2 (6.53%) - possibly suggesting the performance sensitivity of topologies on the efficiency of communication routines in the MPI implementations. Interestingly, the optimal run-time of the OpenMPI versions (I17O2 and G6O2) had a performance gain of approximately 4.36% over the best execution timing of the Mvapich2/2.2 versions (I17M2 and G6M2). Figure 12d shows the minimum timings for I17O2, I17M2, G6O2 and G6M2. The curves for I17O2 and G6O2 almost overlap i.e. have negligible differences and hence are shown as a single curve. The similarity in the shape of curves in Figure 12d shows the *software independence* of our model. This behaviour is ideally expected as our high level abstract model is derived using only the *data layout*, as elaborated in Section 4, and is independent of any particular software or hardware characteristics.

6.2. *Multiple Nodes*

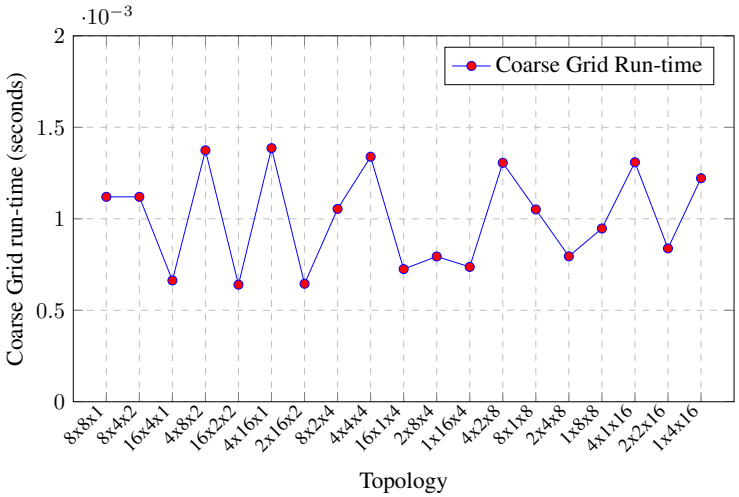
Figure 13a shows the total run-time of parallel geometric multigrid for various topologies which are feasible when the global fine grid size is $512 \times 512 \times 512$ (0.13 billion dof) and the global coarsest grid is $16 \times 16 \times 16$ i.e. 6 levels for $P = 64$ on ARC2. As predicted by our model, there are cache-minimizing topologies which outperform the standard topology $4 \times 4 \times 4$ returned by `MPI_Dims_create()` with $P = 64$. Figure 13b and Figure 13c show the corresponding fine grid

DOI: 10.1002/cpe



(a) Total run-time

(b) Fine Grid (FG) smooth time

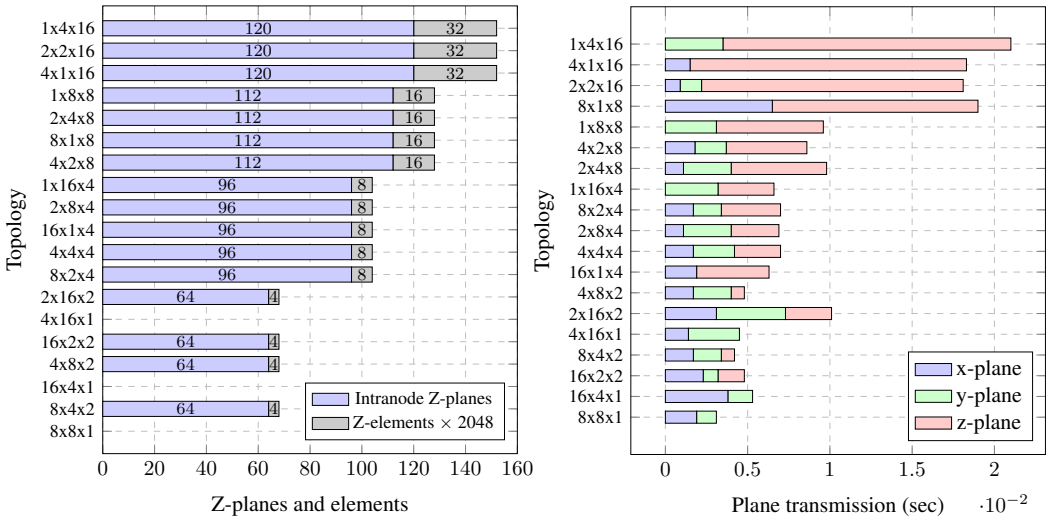


(c) Coarsest Grid run-time

Figure 13. Execution times of Geometric Multigrid for $P = 64$, Fine Grid = 512^3, Levels = 6, Global Coarsest Grid = 16^3, $\nu_1 = \nu_2 = 3$, Fixed Coarsest iterations = 100, Vcycles = 5, Intel 16.0.2, OpenMPI 1.6.5, ARC2, default `MPI_Dims_create()` = $4 \times 4 \times 4$

smooth times and coarsest grid run-times, respectively. The performance improvement of the best performing topology $8 \times 8 \times 1$ over $4 \times 4 \times 4$ in the total run-time is 8.5% whereas in the fine grid smooth time is 9.7%. As the fine grid smoothing time is the major contributor to the total running time, Figure 13a and Figure 13b bear a striking resemblance. The coarsest grid run-times are very small in comparison and appear to be irregular at this level. The cache misses at the coarsest level will have a lesser effect on the running time as compared to the communication time due to process placement and message latency as the *local work-set* of three arrays used in Jacobi updates is 5.1 KB (including the halo cells) for $4 \times 4 \times 4$ and 6.75 KB for $8 \times 8 \times 1$, which can easily fit into the L1d cache. The latter topology passes a maximum of four planes as opposed to a maximum of six by the former. Assuming perfect cache hits (as local-work set fits into L1d cache), it is the message latency which becomes the primary factor in the $8 \times 8 \times 1$'s superior performance over $4 \times 4 \times 4$ at the coarsest level. Our implementation uses *persistent* point-to-point communication at the coarsest level as the number of halo exchanges at the coarsest level $>> (\nu_1 + \nu_2)$ and thus we can expect to see a benefit in *not* destroying the MPI send and receive handles every time data is communicated.

DOI: 10.1002/cpe



(a) Intra-node Z-planes and elements for topologies with $D_z > 1$

(b) Communication times for individual X/Y/Z Planes

Figure 14. $P = 64$, Fine Grid = 512^3, Levels = 6, Global Coarsest Grid = 16^3, $\nu_1 = \nu_2 = 3$, Fixed Coarsest iterations = 100, Vcycles = 5, Intel 16.0.2, OpenMPI 1.6.5, ARC2, default `MPI_Dims_create()` = $4 \times 4 \times 4$

Figure 14a shows the number of intra-node Z-planes being passed for each topology for $P = 64$ on ARC2 at the fine grid level when the topologies are arranged in the ascending order of their total run-times. The number of intra-node/inter-node X/Y/Z planes at all levels for a particular topology are equal *except* for at the coarsest grid. The communication volume decreases by one-fourth in going from a finer level to the next coarser level. Further, we only count the total number of Z-planes which are sent, as it includes the number of Z-planes which will be received. It can be seen from Figure 14a that as the number of Z-planes increase, so does the size of the communicated Z-plane. The number of planes however should not be related directly to the time being taken by a topology as these planes are exchanged *simultaneously*. The majority of the high performing topologies in this case again are the ones which pass a smaller sized Z-plane or do not pass a Z-plane at all.

For $P = 64$, the maximum time taken by any process to communicate X/Y/Z planes was measured on ARC2. It can be seen from Figure 14b that whenever the time taken by X-planes is greater than the time taken by Y-planes, the X plane was larger than the Y plane or the X plane was passed between *racks* and thus the switch hop latency contributed to the total time. Further, whenever equal sized X/Y and Z planes were passed, irrespective of whether it was an intra-node or inter-node plane, the Z-plane communication time exceeded its siblings. The exceptional case was with the topology of $4 \times 4 \times 4$, where an equal sized Y-plane (intra-node) took more time than the X-plane (inter-node). More research is needed to determine the reason for this deviation from the normal.

We can differentiate between various plane categories depending on the hierarchy of network they interact through. Table III divides the X/Y/Z planes into 4 categories each depending on their region of movement. The cheapest communication is intra-node communication and the costliest communication is inter-rack communication. Considering the case of extreme topologies with $P = 64$ processes or cores i.e. $1 \times 1 \times 64$, $1 \times 64 \times 1$ and $64 \times 1 \times 1$, we recorded the exchange of planes on ARC2 as listed in Table IV. It can be seen from Table IV that exactly the same number and size of planes are passed in a particular category. The corresponding running times at the fine grid level and coarsest grid level (global coarsest grid = 64^3) is shown in Table V where it can be

DOI: 10.1002/cpe

Table III. PLANE TYPES: Categories of planes based on network elements that they pass through, namely, node/shelf/rack

CATEGORY	DESCRIPTION
C0	Intra-node X-plane
C1	Inter-node Intra-shelf Intra-rack X-plane
C2	Inter-node Inter-shelf Intra-rack X-plane
C3	Inter-rack X-plane
C4	Intra-node Y-plane
C5	Inter-node Intra-shelf Intra-rack Y-plane
C6	Inter-node Inter-shelf Intra-rack Y-plane
C7	Inter-rack Y-plane
C8	Intra-node Z-plane
C9	Inter-node Intra-shelf Intra-rack Z-plane
C10	Inter-node Inter-shelf Intra-rack Z-plane
C11	Inter-rack Z-plane

Table IV. PLANE FREQUENCY: Number of X/Y/Z Intranode/Intra-shelf/Intra-rack planes for 1-D topologies on ARC2

TOPOLOGY	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11
1x1x64	0	0	0	0	0	0	0	0	120	0	4	2
1x64x1	0	0	0	0	120	0	4	2	0	0	0	0
64x1x1	120	0	4	2	0	0	0	0	0	0	0	0

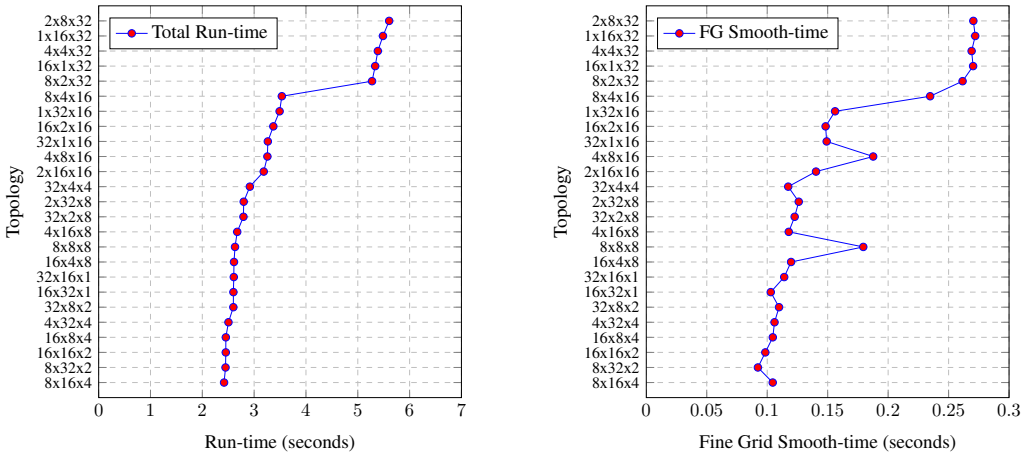
Table V. EXTREME TOPOLOGIES: run-times for $N = 512^3$, $P = 64$, GCG = 64^3, Coarsest iterations = 100, Vcycles = 5, $\nu_1 = \nu_2 = 3, \omega = 1$, FG (Fine Grid), CG (Coarsest Grid), Intel 16.0.2, OpenMPI 1.6.5, ARC2

TOPOLOGY	LEVEL	FG SMOOTH-TIME	CG RUN-TIME
1x1x64	4	1.08 sec	0.028 sec
1x64x1	4	0.39 sec	0.010 sec
64x1x1	4	0.36 sec	0.008 sec

seen that the time taken by the X and Y partition is almost equal but the Z partition is outperformed by a factor of ≈ 3 and 3.5 at the fine grid level and coarsest levels, respectively. This shows that in addition to process placement (which is the same for all partitions in this case), *cache-misses* play a very important factor in the *packing/unpacking/update* times of these planes. The Cache Line Utilization (CLU) factor for the Z-plane is *0.125* at the fine grid level where $P_z = 8$ for $1 \times 1 \times 64$, whereas it is *one* for the X/Y planes. Thus, even when the DCU and IP-based stride prefetcher in the L1d cache are able to hide the latency by prefetching the needed lines, a penalty must be paid as the Z-plane elements reside in different cache lines.

Figure 15a shows the total run-times for topologies with $P = 512$ and a fine grid of size 1024^3 i.e. ≈ 1 billion degrees of freedom. The standard topology of $8 \times 8 \times 8$ is outperformed by several topologies which have $D_z \leq 8$. The highest performing topology outperforms the standard by 8%, whereas in Figure 15b which shows the fine grid smoothing times, it outperforms the standard by 41%. There is clearly a loss of efficiency when comparing the total overall run-times and the fine grid smooth times. Further investigation is needed to ascertain the exact cause. Although all the topologies were examined on the same set of cores, a possibility of increased congestion in the network due to other user jobs cannot be ruled out as the allocated partition by the job scheduler on our test machine ARC2 is not independent. Thus, our reproducible single node experiments are crucial to testing the validity of our model.

DOI: 10.1002/cpe



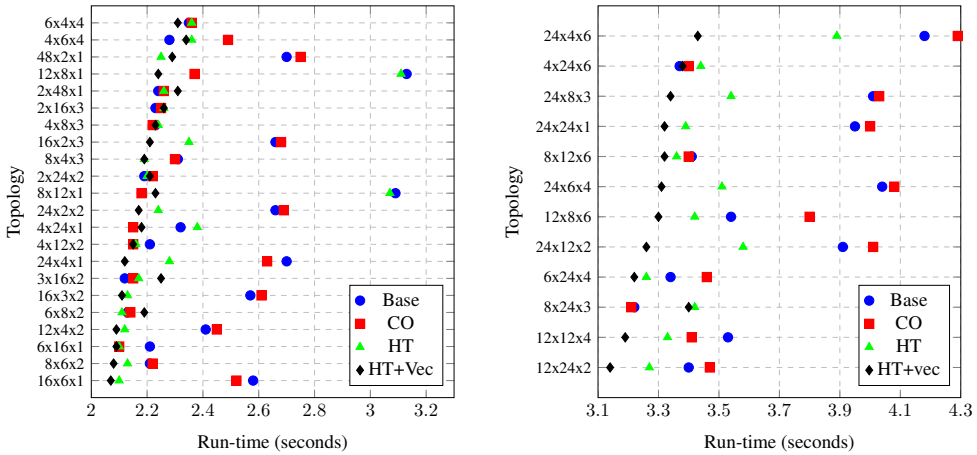
(a) Total run-times (b) Fine Grid smooth-times

Figure 15. Total run-time and Fine Grid smooth-times for $P = 512$, Fine Grid = 1024^3, Levels = 6, Global Coarsest Grid = 32^3, $\nu_1 = \nu_2 = 3$, Fixed Coarsest iterations = 800, Vcycles = 5, Intel 16.0.2, OpenMPI 1.6.5, ARC2, default `MPI_Dims_create()` = $8 \times 8 \times 8$

To elaborate on the trend of topology execution times, Figure 16a and Figure 16b show the multiple node scenario with $P = 96$ and $P = 576$ on ARC3. The Baseline (Base) versions of the predicted topologies $12 \times 8 \times 1$ and $8 \times 12 \times 1$ in Figure 16b are both outperformed by the `MPI_Dims_create()` topology (MDC) of $6 \times 6 \times 4$ by 23.93% but the aggressive CO version of $8 \times 12 \times 1$ outperforms the MDC by 6.89%. The Baseline version suggests that as P_z increases to large values (768 in this case), the LRU policy (see Figure 6) results in the eviction of data in the cache when $D_z = 1$, as a much larger number of cache lines are accessed before the data is utilized again. For example, with $N = 768$ and $D_z = 1$, $P_z = 768$ and approximately $\frac{768}{8} = 96$ cache lines must be accessed before the data point at $u_{i,j+1,k}$ is accessed again after utilizing it to update $v_{i,j,k}$. With Heuristic Tiling and explicit Vectorization (HT+Vec), the compiler is *forced* to vectorize as opposed to issuing only a request for Vectorization at optimization levels `-O2` and `-O3` - the effect of which is evident with the best execution timings being obtained under a combination of Heuristic Tiling and Vectorization. With $P = 576$, the optimal value of D_z shifts to a value of two and again shows that for extremely large domain sizes, an upward shift in the minimal base value of D_z might be needed to avoid mispredictions.

Table VI summarizes the Weak Scaling results by comparing the average performance gain of highest performing topologies with respect to the MDC on up to 1024 cores. Our experiment shows that we are always able to find a topology with a $D_z < D_{sz}$, where D_{sz} is the Z-dimension returned by MDC, that outperforms the standard topology. With $P = 1024$ and $P = 512$, the Z-planes in the MDC topology are still communicated within a node and the cache-minimizing topologies send/receive larger X/Y planes to/from different racks. Despite inter-rack latencies and larger X/Y planes with cache-minimizing topologies, the cost of sending large-sized Z-planes contributes to the higher execution times of the standard topology. As our test facility does not have 4096 cores, we only weak scale up to 1024 cores with ≈ 1 million cells/cores. As opposed to the smaller problem size chosen on ARC2, where tiling and Vectorization yield negligible benefits, we choose a larger problem on ARC3 for Weak Scaling i.e. 18 million cells/core ($\approx$ 29 billion dof). We separately report the Weak Scaling results for the Base, CO, HT and HT+Vec versions as Heuristic Tiling has a significant effect at this problem size (see Table VII). Our HT+Vec scheme decreases the overall run-time of the standard topology by 18.45% but also decreases the gain that cache-minimizing topologies have over the standard topology to approximately 4%. Nonetheless, it is important to note the large gain of approximately 19% in the CO versions.

DOI: 10.1002/cpe



(a) $P = 96$, $N = 768$, 5 levels, Coarsest iterations = 800, 5 V(3,3) cycles, default `MPI_Dims_create()` $= 6 \times 4 \times 4$

(b) $P = 576$, $N = 1536$, 7 levels, Coarsest iterations = 400, 5 V(3,3) cycles, default `MPI_Dims_create()` $= 12 \times 8 \times 6$

Figure 16. Baseline (Base), Compiler Optimized (CO), Heuristically Tiled (HT) and HT + Explicit Vectorization (Vec) total run-time of topologies with Intel 17.0.1, OpenMPI 2.0.2 on ARC3

Table VI. WEAK SCALING ON ARC2: Highest performing Vs standard topology percentage performance gain, Intel 16.0.2, OpenMPI 1.6.5

CORES (Cells/core)	TOTAL RUN-TIME	FINE GRID SMOOTH
64 (≈2 million)	11.1%	14.4%
512 (≈2 million)	17.3%	36.4 %
1024 (≈1 million)	9.6%	8.8%

Table VII. WEAK SCALING ON ARC3: Highest performing Vs standard topology percentage performance gain, TR (Total Run-time), FG (Fine Grid), Base (Baseline), CO (Compiler Optimized), HT (Heuristically Tiled), Vec (explicit Vectorization), Intel 17.0.1, OpenMPI 2.0.2, Coarsest iterations = 200, ≈ 18 million cells/core, Global Coarsest Grid = 48^3

	Base (%)		CO (%)		HT (%)		HT+Vec (%)	
CORES	TR	FG	TR	FG	TR	FG	TR	FG
24	18.56	25.24	18.94	25.81	5.06	2.81	4.10	4.43
192	19.03	25.81	19.51	27.79	4.49	3.81	3.74	3.96
1536	16.71	20.79	18.86	19.75	4.49	1.49	3.76	0.68

Table VIII shows that *Strong* Scaling cache minimizing topologies in Parallel Geometric Multigrid on ARC2 still leads to performance gains up to $P = 256$. The maximum value of EPWSS (Effective Plane Working Set Size) = WPSS + P_yP_z + P_yP_z (for arrays u, v and f in Equation (1) Section 4, respectively) is ≈ 2.5 MB at $P = 128$ but reduces to ≈ 1.25 MB at $P = 256$. Since the *actual inclusive* L3 cache/core is 2.22 MB, similar behaviour of the cache minimizing and standard topology is expected due to the EPWSS completely fitting in the shared Last Level Cache (L3). The Strong Scaling results for ARC3, as shown in Table IX, again show that even with a shrinking problem size per core, the cache-minimizing topologies can outperform the communication volume minimizing topology and thus are also suitable for Strong Scaling till the cores reach a number at which the EPWSS completely fits in the LLC.

DOI: 10.1002/cpe

Table VIII. STRONG SCALING ON ARC2 : % performance gain of Cache Minimizing Topology over Standard Topology for Baseline, Compiler Optimized and Heuristically Tiled versions, N=512, 20 V(3,3) cycles, Coarsest iterations = 100, Levels = 6, Intel 16.0.2, OpenMPI 1.6.5

CORES	BASELINE	COMPILER OPT.	HEURISTIC TILE
16	15.00%	16.14%	6.16%
32	3.88%	4.04%	8.96%
64	12.69%	12.24%	13.30%
128	7.98%	7.29%	7.85%
256	0.82%	-0.82%	5.50%

Table IX. STRONG SCALING ON ARC3 : % performance gain of Cache Minimizing Topology over Standard Topology for Baseline, Compiler Optimized and Heuristically Tiled with Explicit Vectorization versions, N=768, 5 V(3,3) cycles, Coarsest iterations = 400, Levels = 6, Intel 17.0.1, OpenMPI 2.0.2

CORES	BASELINE	COMPILER OPT.	HEURISTIC TILE + VECTORIZATION
48	9.75%	10.10%	8.58%
96	9.05%	9.48%	8.44%
192	14.06%	13.17%	7.62%
384	7.46%	9.09%	6.25%

7. MODEL ACCURACY

We define the accuracy of our model as the fraction of predicted topologies which outperform the default `MPI_Dims_create()` (MDC) topology. Formally, let n_p be the total number of predicted topologies for P cores and $D_z < D_{sz}$, where D_{sz} is the Z-process dimension of the MDC topology and D_z that of the predicted topology. Let t_p be the execution time of the predicted topology and t_{MDC} that of the MDC topology. If $\widetilde{n_p}$ is the number of predicted topologies for which $t_p < t_{MDC}$, then the accuracy of the model is $\frac{\widetilde{n_p}}{n_p} \times 100$ with P processor cores.

Table X shows the accuracy of the model with respect to the hardware clusters ARC2 and ARC3 for various core counts, domain sizes, compilers and MPI implementations. It should be noted that *not all* the predicted topologies are able to reach a pre-defined Multigrid level with a particular domain size and thus such predicted topologies are not counted towards calculating the accuracy. For example, with $P = 768$ only two predicted topologies were experimentally valid (see Table X). Further, it is the predicted topologies with a large $|D_x - D_y|$ that are outperformed by the MDC topology and constitute the *False Positives* (FP). For example, the predicted topology of $24 \times 6 \times 4$ with $P = 576$, $N = 1536^3$, (see Table X) is outperformed by the default MDC $12 \times 8 \times 6$ by a thin margin of 0.30% due to a very high $|D_x - D_y| = 18$. We do not count the *False Negatives* (FN) towards calculating the accuracy. In addition to predicting high performing cache-minimizing topologies, we are *also* able to successfully prune out inefficient topologies with a high degree of accuracy (True Negative accuracy shown in Table X). Further, the *False Negative* topologies i.e. topologies whose performance our model predicts to be worse than the MDC performance but which experimentally outperform the MDC, are the ones which are closer in performance to that of the default `MPI_Dims_create()` topology. As an example, for $P = 96$, $N = 768^3$, using GNU 6.3.0 and Mvapich2 on ARC3, the *False Negative* topology of $24 \times 2 \times 2$ outperforms the MDC by 0.88% only. To convert the *False Negatives* to *True Positives*, the value of D_z can be incremented like $D_z \leftarrow D_z + 1$ instead of $D_z \leftarrow 2 \times D_z$ but this also increases the probability of obtaining more *False Positives*. In this work, increasing D_z as $D_z \leftarrow 2 \times D_z$ was found to be sufficient for obtaining accurate results.

DOI: 10.1002/cpe

Table X. MODEL ACCURACY: P = number of cores, N = Domain size, n_p = Number of predicted topologies, $\widetilde{n_p}$ = Predicted topologies for which $t_p < t_{MDC}$, MDC = `MPI_Dims_create()` topology, Accuracy (True +) = $\frac{\widetilde{n_p}}{n_p} \times 100$

							ACCURACY	
P	N	n_p	$\widetilde{n_p}$	MDC	COMPILER	MPI	True +	True -
				ARC2				
16	512^3	3	3	$4 \times 2 \times 2$	Intel 16.0.2	OpenMPI 1.6.5	100%	77.78%
64	512^3	7	7	$4 \times 4 \times 4$	Intel 16.0.2	OpenMPI 1.6.5	100%	90.90%
512	1024^3	9	8	$8 \times 8 \times 8$	Intel 16.0.2	OpenMPI 1.6.5	88.89%	93.34%
				ARC3				
24	576^3	4	4	$4 \times 3 \times 2$	Intel 17.0.1	OpenMPI 2.0.2	100%	100%
24	576^3	4	4	$4 \times 3 \times 2$	GNU 6.3.0	OpenMPI 2.0.2	100%	100%
24	576^3	4	3	$4 \times 3 \times 2$	Intel 17.0.1	Mvapich2/2.2	75%	100%
24	576^3	4	3	$4 \times 3 \times 2$	GNU 6.3.0	Mvapich2/2.2	75%	100%
48	768^3	10	10	$4 \times 4 \times 3$	Intel 17.0.1	OpenMPI 2.0.2	100%	76%
96	768^3	12	12	$6 \times 4 \times 4$	Intel 17.0.1	OpenMPI 2.0.2	100%	88.89%
96	768^3	12	10	$6 \times 4 \times 4$	GNU 6.3.0	Mvapich2/2.2	83.34%	97.14%
192	768^3	6	6	$8 \times 6 \times 4$	Intel 17.0.1	OpenMPI 2.0.2	100%	82.60%
384	768^3	6	5	$8 \times 8 \times 6$	Intel 17.0.1	OpenMPI 2.0.2	83.34%	100%
576	1536^3	6	4	$12 \times 8 \times 6$	Intel 17.0.1	OpenMPI 2.0.2	66.67%	100%
768	1536^3	2	2	$12 \times 8 \times 8$	Intel 17.0.1	OpenMPI 2.0.2	100%	100%
1536	3072^3	6	4	$16 \times 12 \times 8$	Intel 17.0.1	OpenMPI 2.0.2	66.67%	100%

8. CONCLUSION

Traditionally, domain partitioning has been a function of only load-balance and communication volume. Thus, the orthodox approach aims to achieve maximal load balance and minimize communication volume. We challenge this approach and introduce a third dimension to the problem of domain partitioning : Cache-misses at the sub-domain level. Thus, instead of *only* optimizing cache-misses through spatial and temporal methods *after* the domain partitioning, we analyze the cache-misses at the sub-domain level *before* performing domain partitioning and use them to predict optimal domain partitions in parallel Geometric Multigrid (GMG). To this effect, we develop a high level quasi-cache-aware model which assumes that the interpolation/restriction is proportional to the smoothing time but dominated by the latter. The model estimates the cache-misses for the update of the Independent Compute and the update/packing/unpacking of the dependent planes. Though we develop our model using a 7-pt stencil, the methodology can be applied to a 19-pt or 27-pt stencil. Our numerical tests show the same qualitative results with appropriate quantitative differences. Upon subsequent minimization with respect to sub-domain dimensions, the two most important factors needed to obtain optimal domain partitions that emerge out of the model are : (i) The balance between the X and Y sub-domain dimensions and ; (ii) Maintaining a Cartesian process-dimension $1 \leq D_{zoptimal} \leq D_{sz}$, where D_{sz} is the Z-dimension returned by the default `MPI_Dims_create()` function.

We emphasize and elaborate the factors affecting sub-domain dimensions namely, Independent Compute, Plane cache-misses, Prefetch, Vectorization, Communication Volume, and the LRU eviction policy. We lay stress on maintaining a balance between the cost of growing communication volume when maximizing the unit stride dimension and the growing cost of packing/unpacking/updating the Z-plane when the communication volume is minimized. Our experiments on single and multiple nodes expand on the three most important factors : Independent Compute, dependent plane and communication volume. The single node experiments further

DOI: 10.1002/cpe

show that, even without communication, Weak Scaling a problem on a SMP does not keep the time constant due to the rising contention for the shared Last Level Cache. Topologies efficiently executing the Independent Compute are not optimal when communication is added and thus optimality requires a balance between compute cache-misses and the overhead of communication. Further, we develop a light-weight run-time heuristic for tiling, functioning at all but the coarsest level of GMG, which is close to optimal for high performing topologies, given that exhaustive tiling leads to a combinatorial explosion of the tiling space. The experiments for process placement within a node i.e. `--bind-to-core --bysocket` and `--bind-to-core --bycore` yield similar results for plane communication costs.

Our results on Weak Scaling parallel GMG up to 1536 cores show that the standard partition returned by the default `MPI_Dims_create()` communication volume minimizing function is outperformed by our predicted cache-minimizing partitions. Further, the cache-misses due to the large Working Set Size (WSS) at the finer grid levels contributes maximally to the total execution time. The reducing performance difference between the communication minimizing and cache-minimizing topologies, as shown by Strong Scaling results at higher core counts, demonstrates the efficacy of our predicted optimal topologies. *Most importantly* for *all* our experiments we are able to find optimal topologies which outperform the partitioning imposed by the default `MPI_Dims_create()` function, thus demonstrating that *only minimizing the communication volume is insufficient* for obtaining optimal domain partitions for parallel GMG. We conclude that due to the advances in the hardware and software ecosystem, it has become necessary to fine-tune the domain partitions, based on the cache-misses of the application-specific computational kernels and packing/unpacking/updating costs of communicated planes, to maximize the performance of parallel GMG.

REFERENCES

1. Smith GD. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford University Press, 1985.
2. Strauss WA. *Partial Differential Equations: An introduction*. Wiley, 17 March 1992.
3. Farlow SJ. *Partial differential equations for scientists and engineers*. Courier Corporation, 2012.
4. Saad Y. Iterative Methods for Sparse Linear Systems Second Edition. *Society for Industrial and Applied Mathematics* 2003; .
5. Golub GH, Ortega JM. *Scientific computing: an introduction with parallel computing*. Elsevier, 2014.
6. Quinn MJ. *Parallel Programming*, vol. 526. TMH CSE, 2003.
7. Briggs WL, McCormick SF, *et al.*. *A multigrid tutorial*. Siam, 2000.
8. Trottenberg U, Oosterlee CW, Schuller A. *Multigrid*. Academic press, 2000.
9. Jimack PK, Walkley MA, Zhang J. Scalable Parallel Multigrid Preconditioning for High Fidelity Finite Element and Finite Difference Simulations. *Proceedings of the Fourth International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering* 2015; .
10. Bollada P, Goodyer CE, Jimack PK, Mullis AM, Yang F. Three dimensional thermal-solute phase field simulation of binary alloy solidification. *Journal of Computational Physics* 2015; **287**:130–150.
11. Hülsemann F, Bergen B, Rüde U. Hierarchical hybrid grids as basis for parallel numerical solution of pde. *Euro-Par 2003 Parallel Processing*. Springer, 2003; 840–843.
12. Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, Version 3.0. September 2012.
13. Baker AH, Gamblin T, Schulz M, Yang UM. Challenges of scaling algebraic multigrid across modern multicore architectures. *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, IEEE, 2011; 275–286.
14. Baker AH, Falgout RD, Kolev TV, Yang UM. Scaling hypres multigrid solvers to 100,000 cores. *High-Performance Scientific Computing*. Springer, 2012; 261–279.
15. Gropp WD. Parallel computing and domain decomposition. *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations, Philadelphia, PA*, 1992.
16. Notay Y, Napov A. A massively parallel solver for discrete Poisson-like problems. *Journal of Computational Physics* 2015; **281**:237–250.
17. Datta K, Murphy M, Volkov V, Williams S, Carter J, Oliker L, Patterson D, Shalf J, Yelick K. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, IEEE Press, 2008; 4.
18. Datta K, Yelick KA. Auto-tuning stencil codes for cache-based multicore platforms. PhD Thesis, University of California, Berkeley 2009.
19. Datta K, Kamil S, Williams S, Oliker L, Shalf J, Yelick K. Optimization and performance modeling of stencil computations on modern microprocessors. *SIAM review* 2009; **51**(1):129–159.

DOI: 10.1002/cpe

20. Weiß C, Karl W, Kowarschik M, Rüde U. Memory characteristics of iterative methods. *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, ACM, 1999; 31.
21. Kamil S, Husbands P, Oliker L, Shalf J, Yelick K. Impact of modern memory subsystems on cache optimizations for stencil computations. *Proceedings of the 2005 Workshop on Memory System Performance*, ACM, 2005; 36–43.
22. Sellappa S, Chatterjee S. Cache-efficient multigrid algorithms. *International Journal of High Performance Computing Applications* 2004; **18**(1):115–133.
23. Rahman SMF, Yi Q, Qasem A. Understanding stencil code performance on multicore architectures. *Proceedings of the 8th ACM International Conference on Computing Frontiers*, ACM, 2011; 30.
24. Saxena G, Jimack PK, Walkley MA. A Cache-aware approach to Domain Decomposition for Stencil-based Codes. *International Conference on High Performance Computing and Simulation (HPCS 2016)*, 2016; 875–885.
25. Yavneh I. Why multigrid methods are so efficient. *Computing in science & engineering* 2006; **8**(6):12–22.
26. Wesseling P. Introduction To Multigrid Methods. *Technical Report*, DTIC Document 1995.
27. Hülsemann F, Kowarschik M, Mohr M, Rüde U. Parallel geometric multigrid. *Numerical Solution of Partial Differential Equations on Parallel Computers*. Springer, 2006; 165–208.
28. Rivera G, Tseng CW. Tiling optimizations for 3D scientific computations. *Supercomputing, ACM/IEEE 2000 Conference*, IEEE, 2000; 32–32.
29. Hennessy JL, Patterson DA. *Computer architecture: a quantitative approach*. Elsevier, 2012.
30. Williams S, Kalamkar D, Singh A, Deshpande AM, Straalen BV, Smelyanskiy M, Almgren A, Dubey P, Shalf J, Oliker L. Implementation and Optimization of miniGMG-a Compact Geometric Multigrid Benchmark. *Technical Report*, Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US) 2012.
31. Sturmer M, Treibig J, Rude U. Optimising a 3d multigrid algorithm for the IA-64 architecture. *International Journal of Computational Science and Engineering* 2008; **4**(1):29–35.
32. Gropp W, Lusk E, Skjellum A. *Using MPI: portable parallel programming with the message-passing interface*, vol. 1. MIT press, 1999.
33. Frigo M, Leiserson CE, Prokop H, Ramachandran S. Cache-oblivious algorithms. *Foundations of Computer Science, 1999. 40th Annual Symposium on*, IEEE, 1999; 285–297.
34. Kumar V, Grama A, Gupta A, Karypis G. *Introduction to parallel computing: design and analysis of algorithms*. Benjamin/Cummings Publishing Company Redwood City, CA, 1994.
35. Williams S, Kalamkar DD, Singh A, Deshpande AM, Van Straalen B, Smelyanskiy M, Almgren A, Dubey P, Shalf J, Oliker L. Optimization of geometric multigrid for emerging multi-and manycore processors. *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, IEEE Computer Society Press, 2012; 96.
36. Gmeiner B, Köstler H, Stürmer M, Rüde U. Parallel multigrid on hierarchical hybrid grids: a performance study on current high performance computing clusters. *Concurrency and Computation: Practice and Experience* 2014; **26**(1):217–240.
37. Chow E, Falgout RD, Hu JJ, Tuminaro RS, Yang UM. A survey of parallelization techniques for multigrid solvers. *Parallel processing for scientific computing* 2006; **20**:179–201.
38. Li XS. An Overview of SuperLU: Algorithms, Implementation, and User Interface. *ACM Transactions on Mathematical Software* September 2005; **31**(3):302–325.
39. Brown PN, Falgout RD, Jones JE. Semicoarsening multigrid on distributed memory machines. *SIAM Journal on Scientific Computing* 2000; **21**(5):1823–1834.
40. Kaltenecker C. Comparison of analytical and empirical performance models: A case study on multigrid systems 2016; .
41. Gahvari H, Gropp W, Jordan KE, Schulz M, Yang UM. Modeling the Performance of an Algebraic Multigrid Cycle Using Hybrid MPI/OpenMP. *Parallel Processing (ICPP), 2012 41st International Conference on*, IEEE, 2012; 128–137.
42. Brabazon KJ, Hubbard ME, Jimack PK. Nonlinear multigrid methods for second order differential operators with nonlinear diffusion coefficient. *Computers & Mathematics with Applications* 2014; **68**(12):1619–1634.
43. Romanazzi G, Jimack PK. Parallel performance prediction for multigrid codes on distributed memory architectures. *International Conference on High Performance Computing and Communications*, Springer, 2007; 647–658.
44. Intel 64 and IA-32 Architectures Optimization Reference Manual. http://www.intel.com/content/dam/doc/manual/. Accessed: 2016-12-17.
45. Saini S, Chang J, Jin H. Performance Evaluation of the Intel Sandy Bridge Based NASA Pleiades Using Scientific and Engineering Applications. *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*. Springer, 2014; 25–51.
46. FAQ:General run-time tuning. https://www.open-mpi.org/faq/. Accessed: 2016-11-24.
47. de la Cruz R, Araya-Polo M. Towards a multi-level cache performance model for 3d stencil computation. *Procedia Computer Science* 2011; **4**:2146–2155.

DOI: 10.1002/cpe