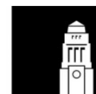


UNIVERSITY OF LEEDS

This is a repository copy of *A Cache-Aware Approach to Domain Decomposition for Stencil-Based Codes*.

White Rose Research Online URL for this paper:
http://eprints.whiterose.ac.uk/100732/

Version: Accepted Version

Proceedings Paper:
Saxena, G, Jimack, PK and Walkley, MA (2016) A Cache-Aware Approach to Domain Decomposition for Stencil-Based Codes. In: 2016 International Conference on High Performance Computing and Simulation (HPCS 2016). 2016 International Conference on High Performance Computing and Simulation (HPCS 2016), 18-22 Jul 2016, Innsbruck, Austria. IEEE . ISBN 978-1-5090-2088-1

https://doi.org/10.1109/HPCSim.2016.7568426

© 2016 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

Reuse
Unless indicated otherwise, fulltext items are protected by copyright with all rights reserved. The copyright exception in section 29 of the Copyright, Designs and Patents Act 1988 allows the making of a single copy solely for the purpose of non-commercial research or private study within the limits of fair dealing. The publisher or other rights-holder may allow further reproduction and re-use of this version - refer to the White Rose Research Online record for this item. Where records identify the publisher as the copyright holder, users can verify any specific terms of use on the publisher's website.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

A Cache-Aware Approach to Domain Decomposition for Stencil-Based Codes

Gaurav Saxena
School of Computing
University of Leeds
Leeds, LS29JT
scgs@leeds.ac.uk

Peter K. Jimack
School of Computing
University of Leeds
Leeds, LS29JT
P.K.Jimack@leeds.ac.uk

Mark A. Walkley
School of Computing
University of Leeds
Leeds, LS29JT
M.A.Walkley@leeds.ac.uk

***Abstract*—Partial Differential Equations (PDEs) lie at the heart of numerous scientific simulations depicting physical phenomena. The parallelization of such simulations introduces additional performance penalties in the form of local and global synchronization among cooperating processes. Domain decomposition partitions the largest shareable data structures into sub-domains and attempts to achieve perfect load balance and minimal communication. Up to now research efforts to optimize spatial and temporal cache reuse for stencil-based PDE discretizations (e.g. finite difference and finite element) have considered sub-domain operations *after* the domain decomposition has been determined. We derive a cache-oblivious heuristic that minimizes cache misses at the sub-domain level through a quasi-cache-directed analysis to predict families of high performance domain decompositions in structured 3-D grids. To the best of our knowledge this is the first work to optimize domain decompositions by analyzing cache misses - thus connecting single core parameters (i.e. cache-misses) to true multicore parameters (i.e. domain decomposition). We analyze the trade-offs in decreasing cache-misses through such decompositions and increasing the dynamic bandwidth-per-core. The limitation of our work is that currently, it is applicable only to structured 3-D grids with cuts parallel to the Cartesian Axes. We emphasize and conclude that there is an imperative need to re-think domain decompositions in this constantly evolving multicore era.**

***Keywords*—PDEs, Domain Decomposition, Stencil, Quasi-cache-directed, Cache-oblivious**

I. Introduction

The introduction of parallel program design, standardized Application Programming Interfaces (API) and enormous support from advancements in hardware of shared and distributed memory machines has instigated researchers to redesign, reimplement and optimize current algorithms. To take advantage of the several CPU cores available in a parallel computer, an existing problem must be partitioned and assigned to these cores. When partitioning/decomposing a problem the focus is typically on computations and data [1], aiming to equidistribute the former and minimize communication of the latter [2]. It is the programmer's responsibility to choose an appropriate decomposition. Standard architecture today varies from a shared memory machine that lets each process access a global address space [3] to a distributed architecture that purely uses message passing for communication/synchronization. MPI

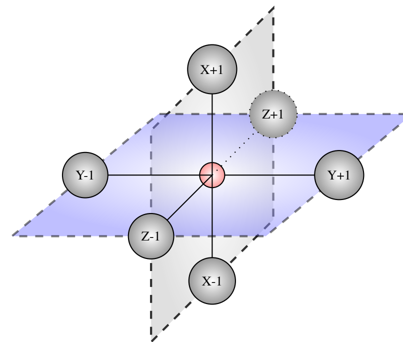


Fig. 1: A 7-pt stencil in 3-D.

(Message Passing Interface) [4] is the de-facto standard for programming distributed memory machines. Hybrid architectures consisting of several shared memory nodes, interconnected by a high speed network like Infiniband [5], have become the norm.

Due to the difficulties in obtaining analytical solutions to PDEs, a good numerical approximation of the solution is needed. *Finite Difference Methods (FDMs)* are a numerical approximation method to estimate derivatives of any order [6]. Although we use FDM, results obtained in this work apply to other stencil-based discretization schemes such as the *Finite Volume Method (FVM)* or the *Finite Element Method (FEM)*. A stencil in FDM is a fixed geometric figure which is used to approximate the value of the dependent variable in the PDE by the weighted contributions of its neighbouring points. In a 7-point stencil (Figure 1), the central point is updated by the weighted average of six of its neighbours. This stencil then moves to the next point to cover the entire domain. Iterative methods like Jacobi, weighted Jacobi ($\omega - Jacobi$), Gauss-Seidel, Conjugate Gradient [6], [7], [8] are used to update the unknowns. Thus, iterative methods are parallelized to obtain stencil based solutions to PDEs. The decomposition of a d-dimensional domain into sub-domains and subsequent assignment to cores *inherently* imposes a logical geometrical arrangement (topology) of the CPU cores as well. Ghost layers/halo data are appended to sub-domains to buffer incoming data from neighbouring processes (and this data must be

Require: Sub-domains with set Dirichlet boundary
while Not converged **do**
`MPI_Irecv` (ghost data)
`MPI_Isend` (next-to-boundary data)
Update (see Figure 3) interior independent values using 7-pt stencil
`MPI_Wait ( )`
Update next-to-boundary values using 7-pt stencil
`MPI_Allreduce` (convergence test)
end while

Fig. 2: High level iterative parallel PDE solver like Jacobi

```
new[i][j][k]=alpha *
           (old[i-1][j][k]+old[i+1][j][k]+
            old[i][j-1][k]+old[i][j+1][k]+
            old[i][j][k-1]+old[i][j][k+1]);
```

Fig. 3: Jacobi iteration kernel, `alpha`=constant, `new` and `old` are 3-D data arrays

received before next-to-boundary points may be updated - so called dependent layers). A high level description of a parallel iterative algorithm, like parallel Jacobi, for solving PDEs is illustrated in Figure 2, while the Jacobi iteration for a stencil with equal weights is shown in Figure 3.

For a given core count, a spatial domain can be divided in several ways. For example, given 64 cores, a total of 28 Cartesian process topologies exist in 3-D. Performance optimization can start with domain decomposition at the macro-level. Figure 4 illustrates that traditional optimizations only consider reducing the cache misses [9] *after* performing domain decomposition [10], [11], [12], [13], [14]. We take a reverse approach in the sense that we derive a domain decomposition based on optimization of cache-misses. Our final objective is then to optimize a sub-domain using an efficient domain decomposition and encourage the use of sub-domain level optimizations.

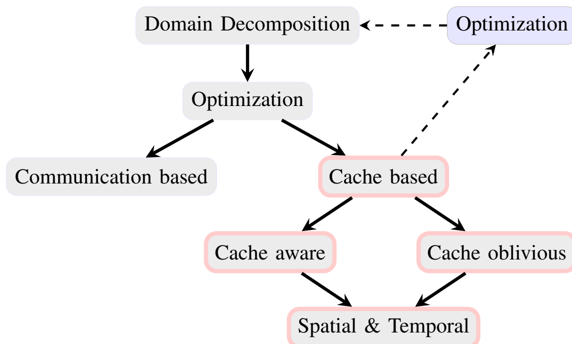


Fig. 4: Traditional optimization (solid arrows), our approach (dashed + solid arrows)

II. Related Work

Domain Decomposition is a partitioning technique where large shareable data structures are split into smaller parts and assigned to processes [1]. Domain decompositions are problem dependent, e.g., matrix multiplications favour a block decomposition whereas LU factorization does not [15]. Partial Differential Equations (PDEs) [6] are prevalent in scientific calculations which model natural phenomena. An associated physical domain must be discretized before simulation, for example, a 2-D metal plate is divided into a 2-D mesh [15]. Second order elliptic PDEs can be completely specified by defining boundary conditions, namely, Dirichlet, Neumann or Robin [1], [7], [8]. The partial derivatives can be approximated at various points using Finite Difference Methods (FDMs) [6]. Iterative methods can then be used to update the value of the approximated solution [7], [8]. Depending on the number of neighbours which are considered for updating, including the point itself, an x-point geometrical stencil is formed. In 3-D, we commonly consider a 7-point or a 27-point stencil [10], [12], [16]. Variable or constant weighted contributions of these neighbours represent the discretized coefficients of the given PDE for a particular data point. Typically stencil based codes achieve poor performance due to low arithmetic intensity [13], [17]. This suggests that arithmetic intensity and cache optimization ought not be neglected when selecting a suitable decomposition.

Parallel efficiency is inherently connected to an optimized serial code and there have been numerous efforts to optimize the re-use of data in the cache memory [10], [11], [12], [13], [16]. Cache blocking/tiling optimizations for maximum cache reuse have focussed both on using appropriate block sizes of data to improve spatial locality as well as enhancing data locality between adjacent time steps or iterations [9], [10], [11], [12], [13], [16]. Partial 3-D blocking has been proposed in the literature to overcome the issue of frequent gaps in the block of memory being considered and to show that highest efficiency is achieved when the blocking factor has the maximum size in the dimension which has contiguous data [10]. Traditionally and universally, optimization in parallel codes is considered *only after* the decomposition has been selected as shown in Figure 4 (solid arrows).

Cost models for cache tiling have been developed but the coarse-grained models do not distinguish between the cost of load and store operations [11]. Further, in Jacobi type iterations the grid that is written is *different* from the one that is read as compared to a grid being updated by the Gauss-Seidel method - a cause of increase in cache-conflict misses [18]. Serial microbenchmarks like the *Stanza Triad (STriad)*, *Stencil Probe* act as a proxy for the actual code, for studying automatic prefetch policies and assessing the performance for larger stencil codes. Furthermore, changing memory hierarchy has reduced the efficacy of cache tiling/blocking [11] and hand-coded optimizations might interfere with streaming memory

mechanisms like software/hardware prefetching [11], [13]. Factors like Translation Look Aside Buffers (TLB) misses, mispredicted branches and hardware prefetches, etc. have also been used to predict the stencil code performance using statistics from performance counters [14]. Cache-aware [10], [11], [12] and Cache Oblivous/transcendental [19] algorithms form an orthogonal approach, with the former taking into account the architectural details of the cache memory hierarchy and the latter completely ignoring them.

III. Our Focus and Contribution

We focus on predicting the best domain decomposition(s) by minimizing a combination of communication elements *and* cache-misses. Our analysis utilizes minimal cache parameters i.e. cache line size. We introduce the term *quasi-cache-aware* to mean that although the analysis is minimally cache-aware, the end result is *cache-oblivious*. Our experiments show that the *same* optimization does not yield the same performance benefits with a *sub-optimal* domain decomposition. The following are our contributions:

- An in-depth analysis and worst-case prediction of read-/write cache-misses due to the local computations in the independent computation kernel and the dependent layers, along with packing/unpacking cache-misses involved in communication of data (see Section IV and V).
- Prediction of high performance families of process topologies (see Section V).
- To emphasize that a hand-coded optimization at sub-domain level can interfere with compiler optimizations (see Section VI).
- Predict and demonstrate that given the same amount of data in an X/Y/Z-plane, communication of Z-planes is the most expensive (see Section VI).
- Investigate trade-offs in determining the best topology for a given core count and problem size (see Section VI and Section VII).

IV. Notation and Experimental Testbed

We represent the size of the input problem as $N_xN_yN_z$, where (N_i+1) is the number of grid points in direction i, and $i=x,y,z$. The outermost points at the boundary do not constitute unknowns (Dirichlet boundary). Hence, we have a system of linear equations in $(N_x-1)(N_y-1)(N_z-1)$ unknowns. Without any loss of generality, we assume $N_x=N_y=N_z=N$. The number of processes (or cores) is = P and any regular Cartesian domain decomposition satisfies $D_xD_yD_z=P$, where D_i is the number of processes in the ith dimension. The number of unknowns per process is $P_xP_yP_z$, where $P_i=\frac{N_i-1}{D_i}$. Allocating a 1-element deep ghost layer to buffer data from neighbouring processes, the 3-D sub-domain size becomes $(P_x+2)(P_y+2)(P_z+2)$. Each sub-domain is composed of three layers: the *ghost layer*, the *dependent layer* - consisting of near-to-boundary values (see Figure 2) that needs data from other processes for updating unknowns - and the *independent layer* (computational kernel) consisting of interior values (see Figure 2) which needs

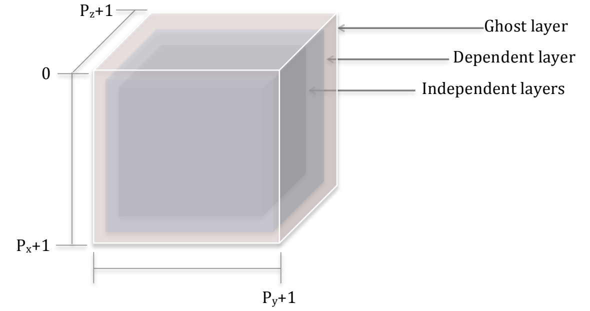


Fig. 5: Subdomain of a process with independent, dependent, ghost layers and indexing from 0

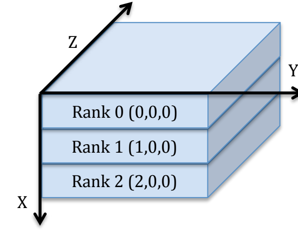


Fig. 6: 3x1x1 Process Grid Decomposition and Coordinate Axes with process ranks/coordinates

no data from neighbouring processes. The ghost layer acts as a true boundary layer when the neighbour process is `MPI_PROC_NULL` (A dummy destination/source process as given by the MPI standard [4]). Figure 5 shows the three basic layers and associated dimensions. Figure 6 shows the process decomposition reference axes, X decomposition and process coordinates given by `MPI_Cart_coords()` (A function in MPI which gives the process coordinates in a Virtual Topology [4]). Our X-axis for the domain is in the opposite direction as the X-axis for process coordinates. The planes are referred to as: X_UP (upper YZ), X_DOWN (lower YZ), Y_LEFT (left XZ), Y_RIGHT (right XZ), Z_TOWARDS_U (XY plane towards reader) and Z_AWAY_U (XY plane away from reader), respectively. The 3-D data layout is depicted in Figure 7.

Our experimental testbed is the ARC2 (Advanced Research Computing) facility at Leeds - a Linux based HPC (High Performance Computing) facility based on CentOS6 distribution. Each compute node consists of 2 Xeon E5-2670 Sandy Bridge processors, each with 8 compute cores (base clock frequency 2.6 GHz, Turbo 3.2 GHz), 16 GB shared memory per processor making a total of 32 GB per compute node. The peak theoretical FLOPS delivered by each processor is 166.4 GFLOP/sec (332.8 GFLOPS/sec per node). Each processor is housed in a socket and has two QPI links, with each link running at 16 GB/sec in each direction simultaneously [20]. There are a total of 190 blades consisting of 380 nodes, making a total of 3040 compute cores (though we use no more than 1024 in this paper). The L1d and L1i cache are 32 KB each,

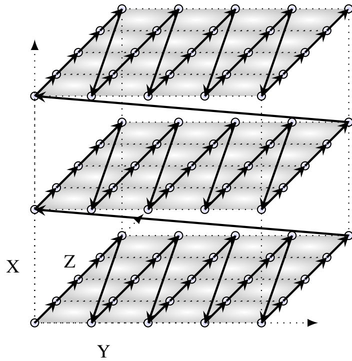


Fig. 7: 3-D data layout : Z direction - contiguous data

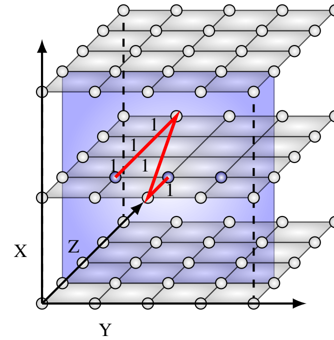


Fig. 8: Dependent Z_TOWARDS_U (shaded vertical rectangle), adjacent points distance (thick solid red line $\approx P_z$) and boundary (unshaded circular points).

L2 cache is 256 KB and 8 cores in a socket share the last level cache (LLC) or L3 of 20 MB. L1d and L2 have a cache line size of 64 bytes and associativity of 8 while L3 has the same cache line size but an associativity of 20. Each node server has a main memory of 32GB of type 1600MHz DDR3 and peak memory bandwidth per node of 102.4 GB/sec. Each socket (CPU) forms a NUMA (Non-Uniform Memory Access) region. The network is QDR Connect-X delivering 40Gbit/sec to the compute blades and storage. The software stack supports Intel C, GNU and PGI compilers with OpenMPI 1.6.5 library and hyperthreading is turned off.

V. Creating a model for prediction

We focus on establishing a relation between minimizing cache misses and domain decomposition by considering the internal layout of data of a sub-domain and the cache line size. Our high level analysis allows us to ignore contention of shared resources, processor architecture, cache-line replacement policies - factors that contribute to cache misses but are extremely difficult to quantify because of the multitude of interactions between contending processes. We always decompose along Cartesian Axes directions i.e. perpendicular cuts along X, Y and Z dimensions (block partitions) and start the analysis by considering the planes consisting of near-to-boundary values (see Figure 2 and Figure 5). In practice, it is not possible to determine the exact number of cache lines being used to contain the data in the working set. The *minimum* number of cache lines which can contain 2 *contiguous* data elements in the Z-direction, 2 *non-contiguous* data elements in the X-direction and 2 *non-contiguous* data elements in the Y-direction is 5. Thus, at any point in time while updating we deal with 2 planes and assume 5 dedicated cache lines. Further, except for Figure 18, we utilize all the cores of a node.

A. Z-Plane

This plane has the greatest effect on the running time as no dimension has contiguous data here i.e. X and Y dimensions. Using a 1-element ghost zone, 2-D data from the dependent layer is packed implictly (using `MPI_Type_subarray()`) in the sending process and sent to the receiver. While *packing*, read-misses (reading from user array and writing to MPI buffer) become significant and while unpacking, write-misses (reading from MPI buffer and writing to user array) become significant. Both are significant when updating an element using its neighbouring elements. Since the cache line size in our experimental testbed is 64 bytes, it can store either 8 `double precision` (DP) values or 16 `floating point/single precision` (FP) values. Figure 8 shows the update of a Z-plane. The near-to-boundary points (in blue) have a minimum distance of P_z between them and hence do not represent contiguous data. When a data point is updated, the cache logic tries to exploit spatial locality. But the greater the value of P_z, and smaller the length of the cache line, the lesser the probability that the next needed element will be found in the cache. Assuming $P_z + 2 > \frac{cache_line_size(64)}{sizeof(FP)}$ for large problem sizes, there is a cache miss for a write on *every* element. Hence, probabilty of a write-miss is $\frac{P_xP_y}{P_xP_y} = 1$. For all practical purposes we assume $P_z + 2 > \frac{cache_line_size}{sizeof(FP)}$. Table I shows the various parameters for Z-planes. Assuming $P_z > 16$ (for `double` values assume $P_z > 8$) there are 2 read-misses in X and Y directions and 1 read-miss in the Z direction. Hence, there is a total of 5 cache read-misses in updating one element (for a large problem), making it a total of $5P_xP_y$ misses for the entire Z-plane. Our analysis of the data access pattern for the Z-plane is for a standard implementation i.e. the sub-domain consists of the dependent layers as well as the ghost layers. A different data layout is possible where the Z-plane is contained in a separate contiguous 1-D array.

TABLE I: Parameters for Z-plane

Description	**Value**
Total elements[1]	P_xP_y
Max. 2 element gap	$P_z + 2 \approx P_z$, if $P_z >> 2$
Total gap both Z-planes	$2(P_x - 1)[(P_y + 1)(P_z + 2) + 2]$
Probability cache write-miss	1, if $P_z + 2 > \frac{cache_line_size}{sizeof(FP)}$
Total cache write-misses	$\approx P_xP_y$, if $P_z + 2 > \frac{cache_line_size}{sizeof(FP)}$
Probability cache read-miss	1, if $P_z + 2 > \frac{cache_line_size}{sizeof(FP)}$
Total update cache read-misses	$5P_xP_y$ if $P_z + 2 > \frac{cache_line_size}{sizeof(FP)}$

[1] Either `float` or `double` data

TABLE II: Parameters for X-planes

Description	Value
Total elements	$P_y P_z$
Max. 2 element gap	2
Total gap/unwanted elements for both X-planes	$2[2(P_y - 1)]$
Probability of cache write-miss	$1/16$
Total cache write-misses	$P_y P_z/16$
Probability of a cache read-miss	$1/16$
Total update cache read-misses	$\frac{5}{16} P_y P_z$

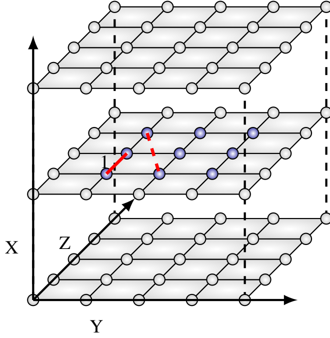


Fig. 9: X-plane update: Data elements are contiguous (solid thick red line) except at boundary (dashed thick red line)

Such a design will increase implementation complexities, may increase cache-conflict misses by altering the working set and further, move the problem to the *next-to-boundary* plane.

B. X-plane

Both X_UP and X_DOWN have contiguous data in the Z direction (blue points in Figure 9). Irrespective of the value of P_z, the gap between the *last element* updated in the Z direction and the *first next element* is always two (two ghost data points). The total gap for $P_y P_z$ points is exactly $2[2(P_y - 1)]$ for both the X-planes. The various parameters for X-planes are shown in Table II. All updates proceed in the Z direction where data is contiguous and hence after a cache-write miss, data would be fetched into the cache according to the cache line size. Thus, there is a cache write-miss after every 16 elements (=$\frac{cache_line_size(64)}{sizeof(FP)}$). Further, there are 5 cache read-misses every 16th element, making a total of $\frac{5}{16} P_y P_z$ cache read-misses for the entire plane in the worst case (assuming no *aggressive* prefetching).

C. Y-plane

The planes Y_LEFT and Y_RIGHT have contiguous data in the Z direction but not in the X direction. The gap between the last updated element (x^{th} row) and the first element in the next row (i.e. $(x+1)^{th}$) row is $(P_z + 2)(P_y + 1) + 2$. Table III shows the parameters for the Y-plane. Data here is contiguous in the Z-direction and hence there is a cache write-miss every 16 elements (= $\frac{cache_line_size}{sizeof(FP)}$ in the worst case), making the probability of a cache write-miss $\frac{1}{16}$. The total cache write-misses are then $\frac{1}{16} P_x P_z$. But unlike the constant maximum distance of 2 elements in updating the X-plane, the

TABLE III: Parameters for Y-plane

Description	Value
Total elements	$P_x P_z$
Max. 2 element gap	$(P_z + 2)(P_y + 1) + 2.$
Total gap both planes	$2(P_x - 1)[(P_z + 2)(P_y + 1) + 2]$
Probability cache write-miss	$\frac{1}{cache_line_size/sizeof(FP)} = 1/16$
Total cache write-misses	$(1/16) P_x P_z$
Probability cache read-miss	$\frac{1}{cache_line_size/sizeof(FP)} = 1/16$
Total update cache read-misses	$\frac{5}{16} P_x P_z$

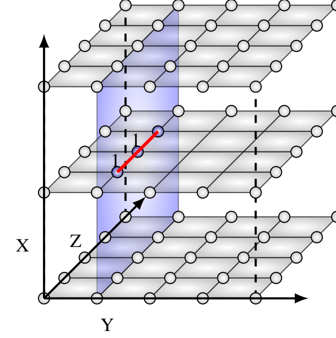


Fig. 10: Dependent Y_LEFT plane (vertical shaded rectangle) and distance between two adjacent points (solid red thick line).

distance here is variable and depends on the Z and Y direction. The higher the value of $(P_z + 2)(P_y + 1) + 2$, the lower the probability that the prefetched data will be available in cache while updating a Y-plane. Figure 10 shows the Y_LEFT plane and a constant gap of 1 element in the Z direction. If we are currently in the xth row, then to reach the first element in the $(x+1)^{th}$ row, we need to cross the ghost boundary and then traverse through $\approx P_y$ elements. We move along the contiguous Z direction to update the Y plane and hence the data for the next element is available if the gap between the current and next element is less than the size of the cache line. Hence, the total cache read-misses is $\frac{5}{16} P_x P_z$ (2 for X neighbours, 2 for Y neighbours and 1 for Z neighbours). If there is prefetching involved then the X decomposition should perform better as there is a maximum constant gap of 2 between any two updated elements and there is higher probability that prefetching will cover that gap of 2 instead of a gap of $(P_z + 2)(P_y + 1) + 2$.

TABLE IV: Parameters for independent computational kernel

Description	Value
Computational elements	$(P_x - 2)(P_y - 2)(P_z - 2)$
Max. 2 element gap	4
Total gap	$4(P_x - 3)[(P_y + P_z)]$
Probability cache write-miss	$1/16$
Total cache write-misses	$\frac{1}{16}(P_x - 2)(P_y - 2)(P_z - 2)$
Probability cache read-miss	$1/16$
Total update cache read-misses	$\frac{5}{16}(P_x - 2)(P_y - 2)(P_z - 2)$

TABLE V: Cache read/write misses for X, Y and Z-plane

Plane	Pack read-misses	Unpack write-misses	Update read-misses	Update write-misses	Total
Z-plane	P_xP_y	P_xP_y	$5P_xP_y$	P_xP_y	$8P_xP_y$
X-plane	$\frac{P_yP_z}{16}$	$\frac{P_yP_z}{16}$	$\frac{5P_yP_z}{16}$	$\frac{P_yP_z}{16}$	$\frac{P_yP_z}{2}$
Y-plane	$\frac{P_xP_z}{16}$	$\frac{P_xP_z}{16}$	$\frac{5P_xP_z}{16}$	$\frac{P_xP_z}{16}$	$\frac{P_xP_z}{2}$

D. Independent computation

Irrespective of the dimensions, in the subdomain interior we have a maximum gap of 4 elements between the last updated element and the next element to be updated. As P_z decreases and P_x and P_y increase, the *total* gap/unwanted elements will increase. In any topology, a write-miss is expected only after approximately 16 elements. Hence, probability of a write-miss is approximately $\frac{1}{16}$, which makes a total of $\frac{1}{16}(P_x - 2)(P_y - 2)(P_z - 2)$ write misses. Since this is the same for all topologies, a uniform cache-miss rate is expected irrespective of the size of the cubic sub-domain but the *total number* of cache misses is a function of the size of the sub-domain. The case for cache read-misses is similar. Table IV shows the parameters for the independent computation kernel. Note that the independent computation kernel is the part of the sub-domain where computation can be overlapped with communication (see Figure 2) using the non-blocking communication routines. When the data is being packed by the communication progress engine, the cache is being used for two purposes: to bring in data for independent computations, and to bring in data from the dependent planes which are being packed if neighbour $\neq$ `MPI_PROC_NULL`. Since the cache is now being used for both the purposes mentioned above, the cache miss rate is likely to go up because of cache pollution. Similar is the case of unpacking of data if the MPI implementation decides to unpack it *before* `MPI_Wait()` is executed. If the data is unpacked at the point of executing the wait call, we are sure that the independent computational core has *already* been updated.

E. Packing, Unpacking and Updating

In general, the number of cache write-misses for unpacking will be the same as cache read-misses while packing data. While updating data, the number of cache write-misses will be different from cache read-misses because of the 7-point stencil. Table V shows the total number of cache misses in the worst case without agressive prefetch, theoretically predicted by our model when a plane is packed, unpacked and updated.

F. Deriving the heuristic by minimization

We proceed to minimize the cache-misses that we derived in the previous sections. The total cache misses for the three planes using Table V can be written as:

$$S = 8P_xP_y + \frac{1}{2}P_xP_z + \frac{1}{2}P_yP_z = \alpha P_xP_y + \beta P_z(P_x + P_y)$$

where α and β are dependent on the length of the architecture-specific cache line (here $\alpha = 8$, $\beta = \frac{1}{2}$) . Our goal is to minimize this expression to obtain the least value of S. By manipulating $\frac{\partial S}{\partial P_x}$ and $\frac{\partial S}{\partial P_y}$, we obtain $P_x = P_y$ but this does not yield any relation to P_z. Since N is constant, the values of P_x, P_y and P_z are dependent on the values of D_x, D_y and D_z such that $D_xD_yD_z = P$, where P is the number of processes or cores. Clearly, we can find all possible combinations of D_x, D_y and D_z and thus find all possible values of $\alpha P_xP_y + \beta P_z(P_x + P_y)$. Minimization of this expression suggests a minimization of a quadratic problem but by observing that we know the various permutations of D_x, D_y, D_z for a given P, we can find the minimum value of S by an exhaustive search by substituting the value of P_x, P_y, P_z in S. Our solution implies that for S to remain minimum, we need $D_x = D_y$ and $D_z = 1$. In the worst case when all six planes are sent, the volume of data is given by:

$$V = 2(P_xP_y + P_yP_z + P_zP_x)$$

Minimizing V by manipulating $\frac{\partial V}{\partial P_x}, \frac{\partial V}{\partial P_y}$ and $\frac{\partial V}{\partial P_z}$, we obtain $P_x = P_y = P_z$. The intersection of conditions for minimization of the sum of communicated elements and minimization of cache misses leads to a *common* condition $P_x = P_y$. This implies that $D_x = D_y$ when $N_x = N_y = N_z = N$. In the more general case where $N_x \neq N_y \neq N_z$, the ratio $\frac{(N_x-1)}{D_x} = \frac{(N_y-1)}{D_y}$ must be maintained.

As the problem size increases, the inner independent computational kernel increases faster than the surface area of planes. For example, when the problem increases 8 times, the independent computational domain increases 8 times as compared to a 4 times increase in the surface area. Our derivation in Section V is based on the assumption that the cache misses due to the independent computation kernel should not be much larger than the sum total of cache misses incurred by the planes. If this case is violated i.e. $\tilde{S} = \frac{(5+1)}{16}(P_x - 2)(P_y - 2)(P_z - 2) >> S$, then the optimal topology moves *towards* the topology given by `MPI_Dims_create()`. This *does not* mean that the topology determining optimal domain decomposition is *always* the one returned by `MPI_Dims_create()` but rather that the optimal topology will be found at a higher $D_z \leq D_{sz}$, where D_{sz} is the Z-dimension returned by `MPI_Dims_create()`. Since minimizing $\tilde{S}$ yields $D_x = D_y = D_z$ and minimizing S gives $D_x = D_y$, $D_z = 1$, thus $1 \leq D_{z_optimal} \leq D_{sz}$. In other words, `MPI_Dims_create()` returns the *upper limit* of the search space of highest performing topologies.

VI. Experimental Results

We implement the Laplace equation $\nabla^2 u = 0$ where $u = u(x, y, z)$ - an elliptic, linear, homogeneous PDE of order two. Dirichlet boundary conditions for boundary $\partial\Omega$ is $u = 1$. Implicit equations in $(N_x - 1)(N_y - 1)(N_z - 1)$ unknowns are solved using a finite difference 7-point stencil on $\Omega = (0, 1)^3$. Without loss of generality and for

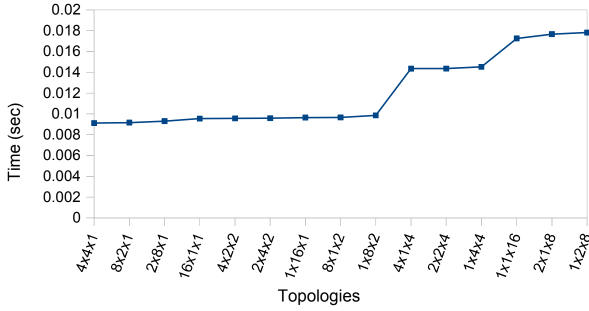


Fig. 11: Time/iteration Vs Topology for 16 processes (SMP), problem size=257^3, 1048576 cells/process

simplicity, the simulation assumes that $(N_i - 1)\%D_i = 0$ for $i = x, y, z$. When $(N_i - 1)\%D_i \neq 0$, it produces a load imbalance and complicates an unbiased study of the effect of domain decompositions. Further, we always use a Jacobi computational kernel (Figure 3) in 3-D for discussions.

Single Node: For 16 processes decomposed in 3-D, the cache equations yield an optimal decomposition of 4x4x1 instead of the 4x2x2 topology given by `MPI_Dims_create()`. The performance of various topologies for 16 processes is shown in Figure 11. Points at the same horizontal level can be visualized as a single family and hence *at least* three families can be observed. The performance gain for the best topology (4x4x1) over the topology minimizing communication (4x2x2) is approximately 4%, while compared to the worst topology (1x2x8), it is approximately 48%. Using our model we search for the solution of $D_x D_y = 16$ and find that $D_x = 4$, $D_y = 4$ satisfies it such that $P_x = P_y$. It may be noted that it is not always possible that $D_x = D_y$. When $D_x \neq D_y$, we find the *closest* D_x, D_y such that the equation holds while keeping $D_z = 1$. When $D_x = D_y$ can be found, we systematically consider the next best topologies to have the X and Y components as $2D_x$ and $\frac{1}{2}D_y$ or $\frac{1}{2}D_x$ and $2D_y$ while keeping $D_z = 1$. Applying this rule to Figure 11 we predict the next highest performing topologies to be 8x2x1 and 2x8x1, which coincides with the experimental values. The topology yielding the lowest communication elements per process is 4x2x2 (minimum surface area) and 4x4x1 but a topology like 8x2x1 and 2x8x1 yields better performance than 4x2x2 *due to cache effects*. Various compiler optimizations were tried in order to bring down the timing of the worst decomposition (among 16x1x1, 1x16x1 and 1x1x16) with a problem of size 161^3, i.e. a decomposition of 1x1x16. We list the results in Table VI. It can be noted from Table VI that even with the *-O2* flag, the compiler generates almost optimal code and that hand optimization interferes with compiler optimization (*-O3* with Rivera et. al. [10] 2-D tiling). Table VII shows the predicted cache misses using our model and the actual cache misses. Even without incorporating prefetching in our model, the predictions are extremely accurate. We

TABLE VI: Time per iteration with different compiler options for problem size=161x161x161 and Processors=16

Compiler Optimization	Time/iteration (10^{-5} secs)
-O2	373
-O3	372
-O3 -xhost	384
-O3 -fp-model fast=1	361
-O3 -fp-model precise -fp-model source	374
-O3 -fimf-precision:low	370
-O3 -unroll4	374
-O3 -opt-prefetch=4	368
-O3, Tile Size=50, Rivera et. al. [10]	394
-O2, Tile Size=50, Rivera et. al. [10]	363

TABLE VII: Predicted Cache Misses (PCM) and Actual cache misses for Problem Size=161^3, Processors=16, Iterations=19353, Independent Compute Elements (ICE)=199712, PCM for ICE=62410

Topology	PCM-planes			Total PCM	Observed Misses	
	Z	X	Y		L1	L2
16x1x1	0	12800	0	1.45E+9	1.8E+9	4.0E+8
1x1x16	204800	0	0	5.16E+9	5.0E+9	1.4E+9
1x16x1	0	0	12800	1.45E+9	1.4E+9	5.3E+8

combine the cache misses of only the functions that contribute significantly towards the total cache misses. The profiler TAU (Tuning and Analysis Utilities) [21] was used to obtain the PAPI (Performance Application Programming Interface) counters like `PAPI_L1_DCM` and `PAPI_L2_DCM` [14]. Table VII shows that the Z decomposition is the worst, with maximum predicted and actual cache misses. This serves as both a motivation and verification for considering topologies like $(2D_x)(\frac{D_y}{2})D_z$ and $(\frac{D_x}{2})(2D_y)D_z$. The observed L1 cache misses for the Y decomposition is less than for the X decomposition although our predictions show that they should be equal. Further investigation is needed to ascertain the exact cause. The order of predicted and observed cache misses is both 10^9 - instilling further confidence in our prediction (we do not predict L1 and L2 cache misses separately but use TAU [21] to capture them individually.).

Multiple Node: Here both local and global communication take place via the Infiniband interface - leading to an increase in communication time due to an added message latency (hops) and increased data in-flight time. We further note that because of the difference in the number of communication elements between a topology which minimizes local cache misses and a topology that minimizes communication elements specifically, the time gap between the execution for our experiment is expected to reduce when the communication time between processes increases due to inter-node communication.

Weak Scaling: Figure 12 shows the results for weak scaling for 8, 64, 216 and 512 processors for 10^6 cells/core. The best topology (minimizing cache misses) for each processor count and problem size is plotted against the

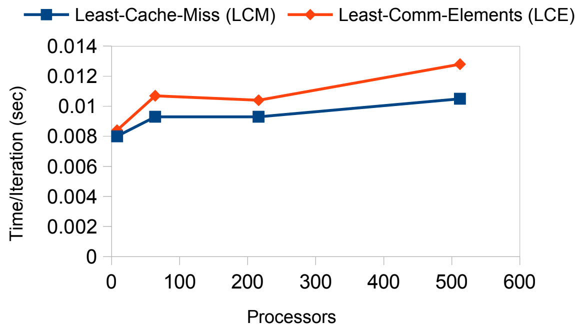


Fig. 12: Weak scaling for 8, 64, 216, 512 processors, Cells/processor = 10^6, Iterations=10000. Best topologies (4x2x1, 16x4x1, 6x12x3 and 8x32x2) Vs (2x2x2, 4x4x4, 6x6x6 and 8x8x8), respectively.

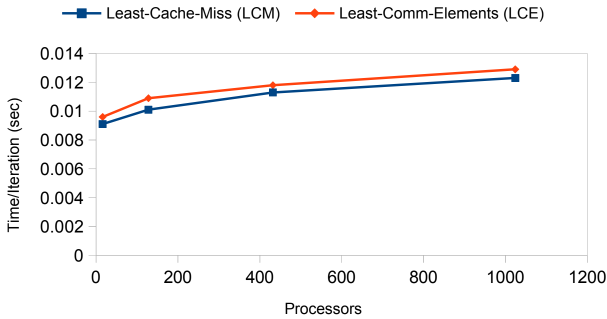


Fig. 13: Weak scaling for 16, 128, 432, 1024 processors, Cells/processor=1048576, Iterations=10000. Best topologies (4x4x1, 16x8x1, 12x12x3, and 16x32x2) Vs (4x2x2, 8x4x4, 12x6x6, and 16x8x8), respectively.

topology `MPI_Dims_create()`. It can be seen that the cache-minimizing topology outperforms the communication minimizing topology consistently and the gap even tends to increase. It was not possible to obtain all possible permutations of decompositions for 216 processors as our implementation assumes that $(N_i - 1)\%D_i = 0$. Figure 13 shows the weak scaling between the two types of topologies for 16, 128, 432 and 1024 processors for a total of 1048576 unknowns per core. The difference between this case and the previous case is that the number of processors is not a perfect cube and hence the `MPI_Dims_create()` may return/returns $D_x \neq D_y \neq D_z$. A smaller gap between the two categories of topologies is *possibly* because D_z is not the cube-root of the processor count and hence may be less than D_x and D_y in the Least Communication Elements (LCE) case, whereas if the processor count is a perfect cube then D_z (= $D_x = D_y$) grows exactly as $P^{\frac{1}{3}}$ (MPI_Dims_create() returns the maximum value of D_z) for the LCE decomposition. Since the process placement also plays a very important role when we venture out of the SMP, we show in Figure 14 (individual topologies not shown because of lack of space) the difference between two runs of the same problem size with

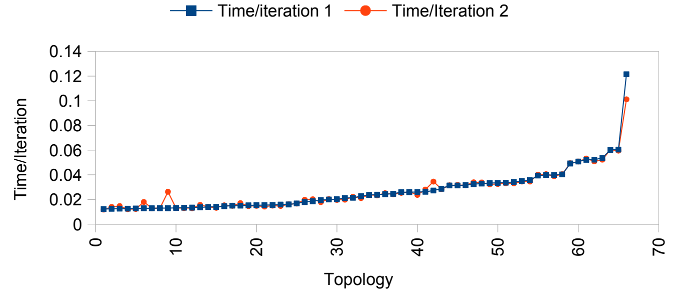


Fig. 14: Topology Timings for two runs of Problem Size=1025^3, P=1024

identical number of processors but *random node allocation*. The topology which minimizes communication i.e. 16x8x8 has a variation of approximately 0.0051 seconds per iteration which shows that obtaining an optimal process placement is just as important as minimizing communication. A detailed discussion of topology mapping/process placement is outside the scope of this paper.

Strong scaling: As the number of cores increase, the communication time increases whereas the computation time decreases due to the decreasing problem size per core. In our experiments, we observed that in addition to the problem size per core, the number of cells in the Z-direction i.e. P_z also affects the optimal decomposition. Further, a tile size of $\frac{N}{2D_z}$ in the Z-direction with D_z is *not* the same as having no tiling with $2D_z$. For example, a tile size of 512 in the Z-direction with $P_z = 1024$ is not equivalent to having no tiling with $P_z = 512$. Further, when we increase D_z, we *trade-off* an increase in the Z-plane update with a decrease in update in the independent computational kernel due to enhanced caching. Table VIII and IX show our results for problems of sizes 1025^3 and 513^3 up-to 512 cores. In *all* the cases we were able to predict high performing decompositions in either 1 or 2 deterministic steps. At $P = 512$, our predicted topology for problem size 1025^3 is 12.38% and for problem size 513^3 is 11.97% more efficient than the standard decomposition (in terms of speed-up), respectively. For a problem of size 2049^3 (approx. 8.6 billion cells, results not shown), we are able to find a higher performing topology (8x16x4) in 3 steps for $P = 512$ that outperforms the standard (8x8x8) in time by 8.6% and at $P = 128$ a topology (4x8x4) that outperforms the standard (8x4x4) by 18.33%. Interestingly, at $P = 512$ and problem size = 2049^3, the three topologies which take the least amount of time to update the independent computational kernel are: 1x512x1, 512x1x1 and 1x1x512, but are outperformed by other topologies due to the *imbalance* in the X/Y dimensions of the aforementioned trio.

Prefetch: In modern microprocessors software and hardware controlled prefetching is used to hide latency and is abstracted away from the user, unlike a cell processor [12] where it can be controlled. As long as the prefetching policy

TABLE VIII: Strong scaling for problem size=1025^3, Iterations=500, steps taken to predict first topology better than MDC i.e. `MPI_Dims_create()`, S_{MDC}=Speed-up of MDC relative to the best, S_{pred}=Speed-up of predicted relative to the best. Best, MDC and Predicted measured in *seconds*

Cores	Best	MDC	Predicted	Steps	S_{MDC}	S_{pred}
16	228.99	235.62	230.41	2	0.97	0.99
32	115.64	116.12	116.12	2	1.97	1.97
64	58.59	63.57	58.59	2	3.60	3.91
128	29.78	31.94	30.22	2	7.17	7.58
256	15.39	16.39	15.39	2	13.97	14.88
512	8.19	9.57	8.36	2	23.93	27.39

TABLE IX: Strong scaling for problem size=513^3, Iterations=500, steps taken to predict first topology better than MDC i.e. `MPI_Dims_create()`, S_{MDC}=Speed-up of MDC relative to the best, S_{pred}=Speed-up of predicted relative to the best. Best, MDC and Predicted measured in *seconds*.

Cores	Best	MDC	Predicted	Steps	S_{MDC}	S_{pred}
2	198.26	199.58	198.26	1	0.99	1.00
4	100.89	100.89	100.89	1	1.97	1.97
8	52.13	54.99	52.13	1	3.61	3.80
16	28.11	30.58	28.11	1	6.48	7.05
32	14.33	15.03	14.33	1	13.19	13.84
64	7.49	8.40	7.49	1	23.60	26.47
128	4.06	4.38	4.09	1	45.26	48.47
256	2.25	2.31	2.25	2	85.83	88.12
512	1.31	1.67	1.47	1	118.72	134.87

remains uniform for every topology, the inclusion or exclusion of prefetching does not affect our model for prediction as it has the *same relative effect* on different decompositions. The probability of a cache miss increases with increasing data gaps (unused elements), theoretically being zero for 0-stride data. Table VI shows that the approximate timing with *aggressive* software prefetch remains almost the same as without it. Theoretically, for independent computation (similarly for X/Y/Z planes), the ratio of unwanted to total elements gives us the *inverse efficiency* of prefetching (η^{-1}) i.e. $\eta_{in}^{-1} = \frac{4(P_x-3)(P_y+P_z)}{4(P_x-3)(P_y+P_z)+(P_x-2)(P_y-2)(P_z-2)}$. For example, if $N = 161$, $D_xD_yD_z = 4 \times 2 \times 1$ then (1-η_{in}^{-1}) = 99.93%. When $D_xD_yD_z = 2 \times 4 \times 1$, (1-$\eta_{in}^{-1}$) = 99.89%. This shows the theoretical superiority of a topology of type $m \times n \times p$ over $n \times m \times p$ where $m > n$. This is also reflected by the execution times of topologies (see Figure 11) in an SMP (but not multiple nodes as process placement plays a significant role or when the independent computational kernel becomes extremely large as then higher cuts in the Y-dimension are preferred i.e. the second fastest changing index). More research is needed to gain an exact understanding of how prefetching works at different data sizes.

Timing comparison for X/Y/Z planes: Figure 15 (Log scale on Y-axis) shows the average time taken by each process to send an equal amount of data in the X, Y and Z planes and is the maximum for the Z-plane. The topology

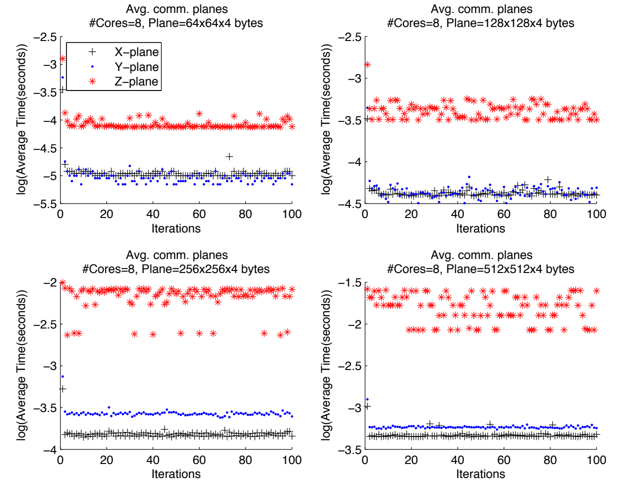


Fig. 15: Average time to send X/Y/Z planes with topology=2^3, plane sizes 64x64x4, 128x128x4, 256x256x4 and 512x512x4 bytes

chosen for this experiment was 2x2x2, ensuring the same number of X, Y and Z neighbours for each process. ARC2 uses a default `--bind-to-core --bysocket` policy and thus the Z-planes are communicated across sockets using dedicated Quick Path Interconnect (QPI) as opposed to shared memory communication for X and Y planes. The mapping can be changed but we prefer to keep the default mapping and not venture into the domain of process placement to limit the scope of the current work.

Figure 16 illustrates the same for inter-node communication with 4 nodes (64 cores) and Figure 17 shows the corresponding cache misses for equal sized X/Y/Z planes with 4 nodes. The Y/Z-planes are sent to neighbour processes on the same node but X-planes travel across SMP's (using Infiniband). The Y planes thus, take less time than X-planes on an average. The Z-planes still take more time than the X-planes, although the former use shared memory for communication. The major contributing component in the average timings of Z-planes is then due to the cache-misses incurred during its packing/unpacking and the contention for shared memory among processes.

Increasing bandwidth per-core: When a node is completely utilized, the memory bandwidth per core is minimal as all the 8 cores of a socket share the same Last Level Cache (LLC) and the main memory module. Since simulation of a PDE using stencil based methods is a memory-bandwidth intensive procedure, we experiment with partial utilization of nodes. Though an *under-utilization* of resources, this can find a potential application in solving the coarsest grid on a subset of processes in parallel multilevel methods like geometric multigrid (for example [7], [22]). Our experiments with P = 64 processors and a problem of size 401^3 is shown in Figure 18. As the *processes-per-node* (ppn) decrease, the application performance increases. Theoretically, there should come a

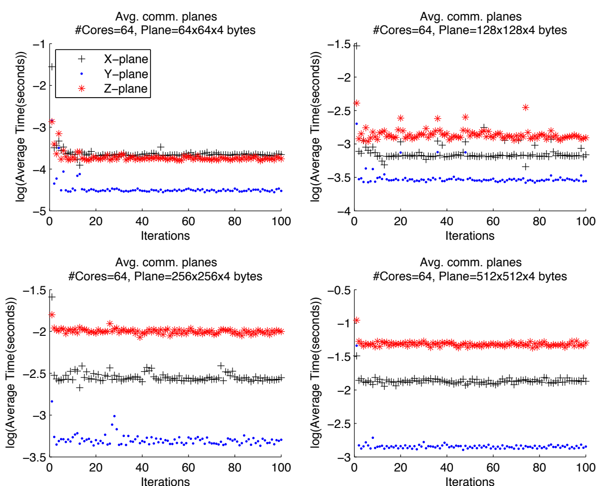
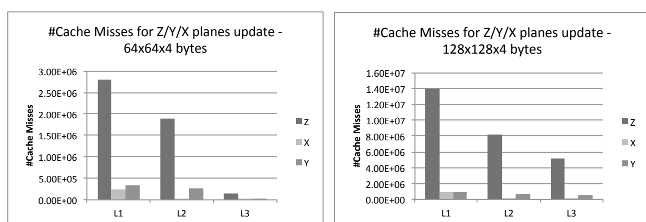


Fig. 16: Average time taken to send X, Y and Z planes with Processors=64 (topology=4x4x4) and plane sizes of 64x64x4, 128x128x4, 256x256x4 and 512x512x4 bytes



(a) Cache Misses for update of planes of size 64x64x4bytes

(b) Cache Misses for update of planes of size 128x128x4bytes

Fig. 17: Cache Misses for updation of Z/Y/X planes of equal sizes with Processors P=64

point where the benefits of increasing memory bandwidth per core will be balanced by the increasing global and local synchronization time. The experiment proves that stencil codes are memory bandwidth intensive.

VII. Conclusion and Future Work

We analytically derive a heuristic for predicting high performing topologies by using a minimally cache-aware

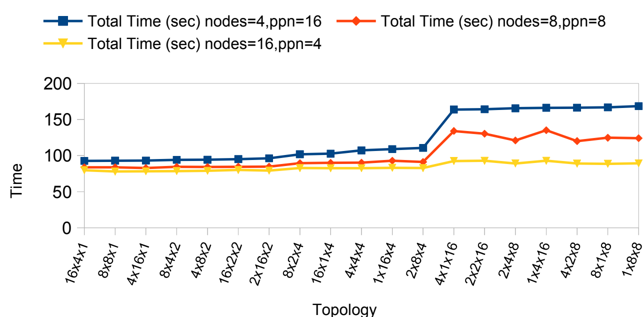


Fig. 18: Topology Timings for Processors=64, Problem Size=401^3, Iterations=10000, Cells/core=10^6 for varying Memory Bandwidth per core

TABLE X: High level summary of tradeoffs in finding optimal domain decomposition, ↓ - a decrease, ↑ - an increase

Factor	Impact
Cache Misses ↓	Communication ↑
Virtual bw./core ↑	Communication time ↑ & Resource utilization ↓
Communication ↓	Computation ↑ & Cache-misses ↑

analysis. The layout of data of a sub-domain is used and experiments demonstrate that certain predicted topologies lead to a higher application performance due to fewer cache misses. Thus, to obtain maximum performance, it is necessary to optimize the domain decomposition *at the macro level* and then implement sub-domain level temporal and spatial optimizations. In this work we only used spatial locality to derive cache-miss governing equations on a standard algorithm. Optimal decompositions not only depend on communication and load balance but also on cache misses incurred due to the memory access pattern in the algorithm, problem size, balance between cuts in the X/Y dimension, data-points in the contiguous-data direction and process placement. Table X summarizes the tradeoffs in optimizing domain decompositions. In general the best performing topologies have higher communication overhead than the topology returned by `MPI_Dims_create()` - the latter incurring higher cache misses. Using nodes partially (Figure 18) increases memory bandwidth per core but increases communication as the domain is spread on a larger number of nodes. Further, locally predicting the ghost values to cut communication increases computation/cache-misses. Our experiments show that a standard decomposition is generally *not* the optimal decomposition for stencil codes.

We plan to enhance this model by incorporating latency, bandwidth, cache capacity, and physical topology factors. Further research is needed to model the interaction of tiling with cache misses in the planes. Although stencil codes offer low temporal locality, a future study to modify the model to incorporate its effects looks interesting. Logical future directions are to apply the current work to multilevel methods, such as parallel geometric multigrid [7], [22] and block-structured Adaptive Mesh Refinement (AMR) [23]. This technique can be exploited in parallel geometric multigrid at two levels : (1) at the fine grid level (2) at the coarsest level when using a subset of processes/cores. Further, with increasing nodes, the effects of process placement become significant and we plan to incorporate this variable in predicting optimal decompositions at run-time. We emphasize and conclude that in the light of growing size of on-chip memories, enhanced bandwidth of interconnects, shrinking latencies, optimizations in the stacks of distributed/shared memory APIs, it is important to re-think domain decompositions for a given core count and problem size.

REFERENCES

[1] M. J. Quinn, *Parallel Programming*. TMH CSE, 2003, vol. 526.

[2] W. D. Gropp, "Parallel computing and domain decomposition," in *Fifth International Symposium on Domain Decomposition Methods for Partial Differential Equations, Philadelphia, PA*, 1992.

[3] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: portable parallel programming with the message-passing interface*. MIT press, 1999, vol. 1.

[4] Message Passing Interface Forum., "MPI: A Message-Passing Interface Standard, Version 3.0." September 2012.

[5] "Infiniband Trade Association : Home," http://www.infinibandta.org, accessed: 2015-05-30.

[6] G. D. Smith, *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford University Press, 1985.

[7] W. L. Briggs, S. F. McCormick *et al.*, *A multigrid tutorial*. Siam, 2000.

[8] G. H. Golub and J. M. Ortega, *Scientific computing: an introduction with parallel computing*. Elsevier, 2014.

[9] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2012.

[10] G. Rivera and C.-W. Tseng, "Tiling optimizations for 3D scientific computations," in *Supercomputing, ACM/IEEE 2000 Conference*. IEEE, 2000, pp. 32–32.

[11] S. Kamil, P. Husbands, L. Oliker, J. Shalf, and K. Yelick, "Impact of modern memory subsystems on cache optimizations for stencil computations," in *Proceedings of the 2005 workshop on Memory system performance*. ACM, 2005, pp. 36–43.

[12] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, p. 4.

[13] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM review*, vol. 51, no. 1, pp. 129–159, 2009.

[14] S. M. F. Rahman, Q. Yi, and A. Qasem, "Understanding stencil code performance on multicore architectures," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*. ACM, 2011, p. 30.

[15] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to parallel computing: design and analysis of algorithms*. Benjamin/Cummings Publishing Company Redwood City, CA, 1994.

[16] K. Datta and K. A. Yelick, "Auto-tuning stencil codes for cache-based multicore platforms," Ph.D. dissertation, University of California, Berkeley, 2009.

[17] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[18] S. Sellappa and S. Chatterjee, "Cache-efficient multigrid algorithms," *International Journal of High Performance Computing Applications*, vol. 18, no. 1, pp. 115–133, 2004.

[19] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *Foundations of Computer Science, 1999. 40th Annual Symposium on*. IEEE, 1999, pp. 285–297.

[20] S. Saini, J. Chang, and H. Jin, "Performance Evaluation of the Intel Sandy Bridge Based NASA Pleiades Using Scientific and Engineering Applications," in *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*. Springer, 2014, pp. 25–51.

[21] "TAU - Tuning and Analysis Utilities," https://www.cs.uoregon.edu/research/tau/home.php, accessed: 2015-05-18.

[22] F. Hülsemann, M. Kowarschik, M. Mohr, and U. Rüde, "Parallel geometric multigrid," in *Numerical Solution of Partial Differential Equations on Parallel Computers*. Springer, 2006, pp. 165–208.

[23] P. Bollada, C. E. Goodyer, P. K. Jimack, A. M. Mullis, and F. Yang, "Three dimensional thermal-solute phase field simulation of binary alloy solidification," *Journal of Computational Physics*, vol. 287, pp. 130–150, 2015.