Applying the swept rule for explicit partial differential equation solutions on heterogeneous computing systems

Daniel J. Magee\textsuperscript{a,1}, Kyle E. Niemeyer\textsuperscript{a,∗}

\textsuperscript{a}School of Mechanical, Industrial, and Manufacturing Engineering, Oregon State University, Corvallis, OR, USA

Abstract

Applications that exploit the architectural details of high performance computing (HPC) systems have become increasingly invaluable in academia and industry over the past two decades. The most important hardware development of the last decade in HPC has been the General Purpose Graphics Processing Unit (GPGPU), a class of massively parallel devices that now contributes the majority of computational power in the top 500 supercomputers. As these systems grow, small costs such as latency—the fixed cost of memory accesses—accumulate in a large simulation and become a significant barrier to performance. The swept time-space decomposition rule is a communication-avoiding technique for time-stepping stencil update formulas that attempts to reduce latency costs. This work extends the swept rule by targeting heterogeneous, CPU/GPU architectures representative of current and future HPC systems. We compare our approach to a naive decomposition scheme with two test equations using an MPI+CUDA pattern on 40 processes over two nodes containing one GPU. We show that the swept rule produces a 4–18 × speedup with the heat equation and a 1.5–3.2 × speedup with the Euler equations using the same processors and work distribution. These results show the potential effectiveness of the swept rule for different equations and numerical schemes on massively parallel compute systems that incur substantial latency costs.

∗Corresponding author

Email address: kyle.niemeyer@oregonstate.edu (Kyle E. Niemeyer)

1Current address: Los Alamos National Laboratory, Los Alamos, New Mexico 87545, USA

Preprint submitted to Elsevier November 21, 2018
Keywords: Domain decomposition, Heterogeneous computing, partial differential equations, computational fluid dynamics, communication-avoiding algorithms

1. Introduction

Computational fluid dynamics (CFD) simulations lie at the heart of technological development in industries vital to high and rising standards of living around the world. However, performing simulations at the level of fidelity necessary for continual insight consumes more resources than individual workstations can reasonably accommodate. As a result these simulations are typically performed on high performance computing (HPC) systems, distributing problems across many nodes that each contain multiple multicore central processing units (CPUs), and increasingly in combination with other specialized “accelerator” co-processors. These heterogeneous—that is, containing more than one processing architecture—computing systems have become ubiquitous in areas of research dependent on large amounts of data, complex numerical transformations, or densely connected systems of constraints. Steady progress in addressing these problems requires developing algorithms that consider hardware capabilities (e.g., computational intensity) and limitations (e.g., bandwidth/communication).

In many ways recent improvements in computational capacity have been sustained by the development of accelerators or co-processors, such as general purpose graphics processing units (GPGPUs) or the Intel Xeon Phi many-core processor, that augment the computational capabilities of the CPU. These devices have grown in power and complexity over the last two decades, leading to an increasing reliance on them for enabling efficient floating-point computation on HPC systems [1]. As these systems grow in complexity, computational power, and physical size, latency and bandwidth costs limit the performance of applications that require regular inter-node communication—such as CFD simulations. Bandwidth is the amount of memory that can be communicated per unit of time, and latency is the fixed cost of a communication event: the travel time of the leading bit in a message. Solving partial differential equations (PDEs) on HPC systems using explicit numerical methods requires domain decomposition (a heuristic for dividing the computational domain across the processors), with required inter-node communication of small data packets for boundary information at every time
step. The frequency of these communication events renders their fixed cost, latency, a significant barrier to the performance of these simulations.

Our work is aligned with the overall goals of the HPC development community and seeks to address, however nascently, two of the challenges on the route to exascale computing systems recently identified by Alexandrov: the need for “novel mathematical methods... to hide network and memory latency, have very high computation/communication overlap, have minimal communication, have fewer synchronization points”, and “mathematical methods developed and corresponding scientific algorithms need to match these architectures [standard processors and GPGPUs] to extract the most performance. This includes different system-specific levels of parallelism as well as co-scheduling of computation” [1].

In this article we describe the development and performance analysis of a PDE solver targeting heterogeneous computing systems (i.e., CPU/GPU) using the swept rule, a communication-avoiding, latency-hiding domain decomposition scheme [2, 3]. Section 2 describes recent work on domain decomposition schemes with particular attention to applications involving PDEs and heterogeneous systems. Section 3 lists the questions this study seeks to answer. Section 4 introduces swept time-space decomposition and discusses the experimental hardware, procedure, and factors used to evaluate performance. In Section 5 we present the results of the tests and describe the hardware and the testing procedures used; lastly in Section 6 we draw further conclusions, describe future challenges, and outline plans for prioritizing and overcoming them.

2. Related work

In our previous work [4] we investigated methods for exploiting the memory hierarchy on a single GPU in a swept time-space solver for stencil computations. Alhubail et al. [2, 3, 5] first developed the swept rule for CPU-based operation in one and two dimensions; Wang also demonstrated how complex numerical schemes can be decomposed into “atomic” update formulas, a series of steps requiring only a three-point stencil, suitable for the swept rule [6]. We use this technique, which we refer to as “lengthening”, in the implementation of the swept rule discussed here and contrast it with another method for dealing with complex schemes, “flattening”, which we used in our previous GPU-only study [4]. Section 5.1 quantitatively compares the two techniques. In addition, Alhubail and Wang applied this procedure to
automatically generate C source code for solving the heat and Kuramoto–Sivashinsky equations using the swept rule on CPU-based systems [7]. These articles comprise the body of work on the swept rule to date, upon which this paper expands.

Memory hierarchies are defined by a series of locations where memory is scarce and fast, to where it is plentiful and slow. By working on data in the limited fast-memory space as long as possible, communication-avoiding algorithms accelerate computations by reducing inter-process communication or accesses to global memory in parallel programs. Swept time-space decomposition is a type communication-avoiding algorithm because it seeks to reduce the number of communication events between the processor and less-accessible memory resources. Unlike most communication-avoiding algorithms, it does not perform redundant operations. The heterogeneous communication-avoiding LU factorization algorithm presented by Baboulin et al. [8] splits the tasks between the GPU and CPU and minimizes inter-device communication. Their results show an appreciable benefit from splitting the types of tasks performed on the CPU and GPU, which reduces overall communication and effectively overlaps computation and communication.

Studies of stencil optimization techniques over the last decade often address concerns closely related to the work presented here. Datta et al. [9] explored domain decomposition with various launch parameters on heterogeneous architectures and nested domain decomposition within levels of the memory hierarchy. Malas et al. [10] previously explored similar diamond tiling methods by using the data dependency of the grid to improve cache use.

Swept time-space decomposition is also conceptually related to parallel-in-time methods [11], such as multigrid-reduction-in-time [12]. These algorithms overcome the interdependence of solutions in the time domain to parallelize the problem as if spatial. This class of techniques iterates over a series of fine and coarse grids using an initial guess for the entire solution domain and effectively smooths out the errors in the solution. Historically, parallel-in-time methods were considered unsuitable for nonlinear problems since the use of coarse grids degraded efficiency and accuracy [2]. However, recent developments applying optimization and auto-tuning techniques have matched the scaling of linear solvers [13]. Parareal is a parallel-in-time method that solves multiple time steps in parallel on a fine grid and corrects the results on a coarse grid until the solution converges, resulting in a solution with the accuracy of the fine grid. Wu and Zhou proposed a new local time-integrator
for this method that shows considerable promise for accelerating convergence rates in fractional differential equations [14].

As these distributed, remote, multi-node systems and have become increasingly heterogeneous in recent years, implementing CFD codes effectively on these systems has grown more complex. Furthermore, as distributed-memory HPC systems continue to grow in size, latency between nodes will continue to impede time-to-solution. As a result, domain decomposition on these systems has received a good deal of recent attention. For example, Huerta et al. [15] used methods from process engineering, including experimental design and non-continuous linear models in an experimental parameter space paradigm, to investigate the performance of a well-known workload division benchmark used to rank HPC clusters on a heterogeneous system. This technique shows considerable promise for future studies of the swept rule with a more-mature code base. However, at our current stage, such a detailed analysis would not provide actionable insights beyond what we have already gleaned from our comparatively simpler methods.

3. Objectives

This study applies swept time-space decomposition to explicit stencil computations intended for distributed-memory systems with heterogeneous architecture, that is, systems with one or more CPUs combined with co-processors. In this study we use systems with multiple CPUs and Nvidia GPUs. The software implementing the swept scheme for heterogeneous systems, $hSweep$, is written in C++ and CUDA and uses the Message Passing Interface (MPI) library [16] to communicate between CPU processes and the CUDA API to communicate between GPU and CPU. As described in Appendix Appendix A this software is openly available, and we archived the version used to produce the results in this study.

While stencil computation is a relatively simple procedure, i.e., applying linear operations to points on a grid, the complexities introduced by computing-system heterogeneity and swept time-space decomposition require a significant number of design decisions. In this work we investigated the performance impact of the most immediately salient and configurable decisions and constrained other potential variations with reasonable or previously investigated values. Our investigations focus on answering the following questions:
1. Does the swept rule reduce time cost using the most favorable launch configurations over a large range of grid sizes?
2. How much work should we give to the GPU in a heterogeneous system?
3. How should we organize the stencil update formula for multi-step methods (discussed in Section 5.1)?
4. Does the size of the domain of dependence substantially affect performance?

4. Methodology

4.1. Swept decomposition

The swept rule exhausts the domain of dependence—the portion of the space-time grid that can be solved given a set of initial values, referred to here as a “block”—before passing the grid points on the borders of each process. We refer to the program that implements the swept rule as Swept and to the program that uses naive domain decomposition (i.e., that passes all boundary between processes at each time step) as Classic\textsuperscript{2}. This way the simulation may continue until no spatial points have available stencils; the required values may then be passed to the neighboring process (i.e., neighboring subdomain) in a single communication event. Our prior work, as well as the original swept decomposition study by Alhubail and Wang, provide detailed explanations and graphical depictions of the swept rule in one dimension, for various architectures [2, 4].

Beyond the ordering of computations, however, the swept scheme uses the same numerical method as the classic scheme. Briefly, for a one-dimensional domain, the heterogeneous one-dimensional swept rule begins by partitioning the computational grid and allocating space for the working array in each process. In this case, the working array is of type states, a C struct that contains the dependent and intermediate variables needed to continue the procedure from any time step. Working array size is determined by the number of domains of dependence controlled by the process, \(N_{\text{blocks}}\), and the number of spatial points covered by a domain of dependence, \(N_{\text{threads}}\), the number of threads per block. Here we use “block” to represent a domain of dependence; it comes from the GPU/CUDA construct representing a collection of threads. The program allocates space for \(N_{\text{blocks}} \times N_{\text{threads}} + (N_{\text{threads}} + 2)/2\)

\textsuperscript{2}Alhubail and Wang [2] use the term “straight” decomposition where we use Classic.
spatial points and initializes the first $N_{\text{blocks}} \times N_{\text{threads}} + 2$ points. The initialized points require two extra slots so the edge domains can build a full domain width on their first step. Interior domains in the process share their edges with their neighbors; there is no risk of race conditions since even the simplest numerical scheme requires at least two values in the state struct, which allows the procedure to alternate reading and writing those values. Therefore, even as a domain writes an edge data point its neighbor must read, the value the neighbor requires is not modified.

The first cycle completes when each domain has progressed to the sub-time step $N_{\text{threads}}/2$ where it has computed two values at the center of the spatial domain. At this point each process passes the first $N_{\text{threads}}/2 + 1$ values in its array to the left neighboring process. Each process receives the neighbor’s buffer and places it in the last $N_{\text{threads}}/2 + 1$ slots; that is, starting at the $N_{\text{blocks}} \times N_{\text{threads}}$ index. It proceeds by performing the same computation on the centerpoints, starting at global index $N_{\text{threads}} - 1$ (adjusted index $N_{\text{threads}}/2 - 1$), of the new array and filling in the uncomputed grid points at successive sub-time steps with a widening spatial window until it reaches a sub-time step that has not been explored at any spatial point and proceeds with a contracting window. Geometrically, the first cycle completes a triangle and the second cycle completes a diamond. After completing the diamond, the program passes the last $N_{\text{threads}}/2 + 1$ time steps in the array and inputs the received buffer starting at position 0. Now it performs the diamond procedure again, with identical global and adjusted indices starting at index $N_{\text{threads}}/2 - 1$.

The procedure continues in this fashion until reaching the final time step, when it stops after the expanding window reaches the domain width and outputs the now-current solution at the same time step within and across all domains and processes. Therefore, the triangle functions are only used twice if no intermediate time step results are output, while the rest of the cycles are completed in a diamond shape.

Our program uses the MPI+CUDA paradigm and assigns one MPI process to each core. We considered using an MPI+OpenMP+CUDA paradigm by assigning an MPI process to each socket and launching threads from each process to occupy the individual cores, but recent work has shown that this approach rarely improves performance on clusters of limited size for finite volume or finite difference solvers [17, 18]. This conclusion has led widely used libraries, such as PETSc, to opt against a paradigm of threading within processes [19], and we followed this decision.
4.2. Experimental method

We will address the questions presented in Section 3 by varying three elements of the swept decomposition: block size (the number of spatial points in each domain), GPU affinity (the number of spatial points assigned to a GPU as a multiple of the number assigned to a CPU process), and grid size. We repeatedly executed our two test equations, the heat equation and Euler equations, over the experimental domain of these variables using the swept and classic decomposition methods.

In our one-dimensional swept program for heterogeneous systems, hSweep, the size of the domain-of-dependence or “block” is synonymous with number of threads per block, because it launches the solution of each domain using a block of threads on the GPU, where each thread handles one spatial point. In GPU/CUDA terms, a block is an abstract grouping of threads that share an execution setting, a streaming multiprocessor, and access to a shared-memory space, which is a portion of the GPU L1 cache. hSweep uses the swept rule to avoid communication between devices and processes and exploits the GPU memory hierarchy to operate on shared memory quantities closer to the processor. Since this multi-level memory scheme influences the swept-rule performance and GPU execution, the resulting effects are difficult to predict.

The independent variables grid size and GPU affinity are more straightforward: the grid size is the total number of spatial points in the simulation domain, and the GPU affinity is the portion of the computational grid that is processed by the GPU. We express the GPU affinity as the ratio of the number of domains-of-dependence assigned to the GPU to those assigned to a single MPI process (on a CPU core). Since a GPU can handle a larger portion of the total grid than a single MPI process, GPU affinity is specified as an integer greater than one.

In our previous study of the swept rule on a single GPU [4], the properties of GPU architecture clearly defined the experimental domain. Here, because a warp contains 32 threads and a block cannot exceed 1024 threads, we constrained the number of threads per block to be a multiple of 32 from 32–1024; this is also the width of the domain of dependence. To enforce regularity, we constrained our experimental problem size—the number of spatial points in the grid—to be a power of 2 larger between 1024 and $2^{21}$.

The addition of GPU affinity as an independent variable introduces further complication to the experimental domain. While our experiments are constrained by GPU architecture in threads per block and by the number
of processes and blocks in problem size, we initially have no clear indication of what the experimental limits of GPU affinity should be—so we took an iterative approach. First, we ran a screening study for a broad range of conditions: eight block sizes from 64–768, 11 GPU affinities from 0–80, and four grid sizes from $5 \times 10^5$–$1 \times 10^7$. This showed us that the best affinity falls between 20–60 and that all threads per block values could provide the best performance. (This was somewhat disappointing, since we had hoped to narrow the range for both GPU affinity and threads per block further to experiment on a finer increment of grid size in a reasonable amount of time.) For the final experiment, we used the same block sizes, GPU affinity values from 20–60 in increments of 4, and seven grid sizes over the same range.

In this study, we solve the one-dimensional heat equation using a first-order forward in time, central in space method, and the Euler equations using a second-order finite-volume scheme with minmod limiter. Explanations of these methods can be found in the appendix of our previous article [4].

5. Results

5.1. Primary data structure

Implementing the swept rule for problems amenable to single-step PDE schemes is straightforward, but dealing with more realistic problems often requires more complex, multi-step numerical schemes. Managing the working array and developing a common interface for these schemes provokes design decisions substantially impact performance. In this article we consider two strategies for dealing with this complexity: “flattening” and “lengthening”, distinguished via code snippets in Figure 1.

The flattening scheme flattens the domain of dependence in the time dimension by using wider stencils and fewer sub-timesteps. This strategy is more memory efficient for the working array, which contains instances of the primary data structure at each spatial point, but it cannot easily accommodate different methods and equations. It also introduces additional complexity from parsing the arrays and requires additional register memory for function and kernel arguments and ancillary variables. Figure 1a depicts the flattening approach applied to the Euler equations using classic domain decomposition.

In the new implementation shown here we use the lengthening strategy, also referred to as “atomic decomposition”, which is instantiated as a struct to generalize the stages into a user-defined data type [6] as shown in Figure 1a.
void classicStep(const double *s_in, double *s_out, bool final) {
    int gid = blockDim.x * blockIdx.x + threadIdx.x;
    // number of spatial points - 1
    int lastidx = ((blockDim.x * gridDim.x));
    int gids[5];
    for (int k = -2; k < 3; k++) {
        gids[k+2] = (gid + k) % lastidx;
    }
    // Final is false for predictor step, true otherwise.
    if (final) {
        s_out[gid] += finalStep(s_in, gids);
    } else {
        s_out[gid] = predictorStep(s_in, gids);
    }
}

(a) flattening method. The sub-timesteps are compressed to a step with a wider stencil. The two arrays which alternate reading and writing are explicitly passed and traded in the calling function.

(b) lengthening method. The states struct contains all information to step forward at any point. A user only needs to write the eulerStep() and pressureRatio() functions and accessing the correct members based on the time step.

Figure 1: Brief code skeletons for the flattening and lengthening methods, applied to solving the Euler equations using classic domain decomposition.

It requires more memory for the primary data structure; for instance, our flattening version of the Euler equations carries six doubles per spatial point since the pressure ratio used by the limiter was rolled into the flattened step. By restricting the stencil to three points, the lengthening method requires the pressure ratio to be stored and passed through the memory hierarchy, meaning the data structure carries seven doubles per spatial point for the Euler equations.

To gauge the influence of our choice for primary data structure, we implemented each combination of the classic and swept decomposition techniques using the flattening and lengthening data structures. We applied these methods to solve the Euler equations using the discretization and conditions described in Section 4.2. For these tests we used a single GPU as the sole processor—this differs from the other results shown in this article, which use CPUs and GPUs in concert on a heterogeneous platform.

Figure 2 compares the performance of the memory-storage techniques in experiments executed on a workstation with an Intel Xeon 2630-E5 v3 and an Nvidia Tesla K40c. Figure 2a shows the speedup, the ratio of the aver-
(a) Speedup of flattening compared to the same scheme using lengthening.

(b) Speedup of Swept program compared to Classic using each data structure strategy.

Figure 2: Performance comparison of the flattening and lengthening strategies solving the Euler equations on a single GPU.

...age time per timestep, of a lengthening implementation compared with a flattening implementation for each domain decomposition scheme. The flattening strategy is faster than lengthening for both decomposition methods, and the speedup grows as the number of spatial points increases, but the performance difference and trend is more pronounced for Classic decomposition than Swept. Much of the reason that lengthening performs worse for both decomposition methods is particular to GPU architecture: the array of structures used in lengthening amplifies the performance sensitivity to irregularity in memory access patterns. As a result, it is difficult to generalize these results to heterogeneous systems. The extra memory requirements of the lengthening method also consume limited shared-memory resources on the GPU, which diminishes both occupancy (the number of threads active on the GPU at any given time) for Swept and the L1 cache capacity used to accelerate global-memory accesses on Kepler-generation GPUs for Classic. While locality is a significant issue for effective CPU memory accesses, it has a larger impact on GPU performance.

These issues explain the general benefit of the flattening strategy, but they do not explain why these benefits are more pronounced for Classic. First, the lengthening strategy, which requires more sub-steps per time step to limit the stencil to three points, does not affect the number of kernels launches for Swept, but may increase the occurrence of these events for Classic. For the Euler equations, our scheme uses four sub-steps per
time step using \textbf{lengthening}, and two using \textbf{flattening}; this causes the \textbf{Classic} \textbf{lengthening} scheme to launch twice as many kernels as it would with \textbf{flattening}. Also, the aforementioned structure of arrays paradigm used in the \textbf{lengthening} strategy increases the stride for memory accesses, which has little effect on the \textbf{Swept} scheme since it uses mostly shared memory. This access pattern does not produce bank conflicts, but it does prevent global memory accesses from coalescing, which incurs a significant cost in the \textbf{Classic} program.

Figure 2b shows the affect of choice in primary data structure on the speedup of \textbf{Swept} compared with \textbf{Classic} for solving the Euler equations. These results match what we found in our previous study [4]: \textbf{Swept} performs worse than \textbf{Classic} using the \textbf{flattening} method on a single GPU solving the Euler equations. However, examining the current results, now we see that when using the \textbf{lengthening} strategy the swept decomposition scheme improves performance over the classic approach. In any experimental algorithm, especially those involving emerging, parallel computational platforms, performance depends on a multitude of implementation details, and we feel that it is important to present these findings so that others who implement the swept rule will have a more thorough understanding of the tradeoffs inherent in particular design choices. For this study, we selected the \textbf{lengthening} method due to its benefits for the swept scheme, as well as better extensibility and regularity.

\subsubsection{5.2. Swept decomposition}

We compiled all test programs with the CUDA compiler \texttt{nvcc v8} and launched using \texttt{mpirun v3.1.4}. Our study collects the average time per time step over 4000 time steps. The timing measurements include the one-time costs of device memory allocation, plus initial and final host-device memory transfers; this does not include the time cost of host-side memory management, grid generation, file I/O, and initial condition calculations. The heterogeneous swept rule algorithms and test cases were implemented in \texttt{hSweep v2.0} [20].

We performed the tests on the \textbf{Classic} and \textbf{Swept} programs on two nodes of the Oregon State University College of Engineering cluster. Each node contains two sockets with Intel Xeon 2660-E5 v3 processors with ten cores each operating at 2.60 GHz. An Nvidia Tesla K40m GPGPU is available to one of these nodes through a PCI connection.
Figure 3: A map of the time cost per time step of the heat equation at four grid sizes. The red dot signifies the best performance.
Figure 4: A map of the time cost per timestep of the Euler equations at four grid sizes. The red dot signifies the best performance.
First, we measured the performance of the Swept and Classic decomposition schemes using the one-dimensional heat and Euler equations over the complete range of operating conditions, as shown in Figures 3 and 4 respectively. By creating similar maps in the first stage of the experiment, we narrowed the range of experimental variables to capture the best performance for each problem, algorithm, and grid size. Figure 4 shows how the algorithms vary in performance by runtime configuration and decomposition method. Notably, these results show that, for the Euler equations, the Swept scheme achieves best performance at GPU affinities between 45–55 and with 768 grid points in the domain of dependence, but performs particularly poorly with 512 points. From other perspectives the performance profile of the Swept Euler implementation is regular, so a sudden drop in performance followed by a substantial increase is unexpected. The consistency of the influence of GPU affinity allows further studies to explore more granular performance characteristics. While the saddle along the threads per block axis is problematic for our general recommendations, we also observe that the program consistently exhibits similar performance for 192–384 threads per block.

We also observe from these maps that Swept produces a more orderly performance profile than Classic. For grids with less than $10^6$ spatial points, the Classic decomposition technique exhibits peaks and valleys in performance over a wide range of block sizes and GPU affinities. On larger grids, performance regularizes and the method performs best for smaller blocks and higher GPU affinities. This suggests that we may have truncated the GPU affinity dimension prematurely in our experiments. The regularity we observe in the experimental grid for Swept also extends to grid size as seen in Figures 5 and 6. The Swept decomposition performs following a power-law curve for each problem. In fact, we performed a least-squares fit of these results to a power law, as shown in Table 1, and found excellent agreement.

Next, using the best launch conditions identified previously, Figure 5a compares the computational time per timestep of the heat equation using the Classic and Swept decomposition methods. These results are generally consistent with our expectations for several reasons. First, we observed in previous studies [2, 4] that Swept applied to simpler problems perform better than Classic decomposition, but reach max performance at smaller grid sizes. This difference in the capacity for increasing grid sizes narrows the performance advantage of Swept swiftly, and, since both algorithms scale similarly with grid size, the relative speedup of Swept declines as well. How-
ever, Swept performs faster than Classic for all grid sizes by about 4–18×.

Second, although the minimum grid size in this study is about 100× larger than our previous study, and the maximum grid size is about 10× larger, Figure 5b shows a similar trend in the speedup of swept decomposition. The heterogeneous Swept implementation studied here offers double the speedup of the GPU-only case studied previously [4] for the same grid size. The improvement in relative speedup of the heterogeneous swept rule is expected since latency is a much larger cost in internode communication than within
Table 1: Coefficients for power-law fit of grid size vs time per time step \((y = Ax^b)\) of Swept performance at best runtime configuration.

| Equation | A       | b       | \(R^2\)  |
|----------|---------|---------|---------|
| Euler    | \(3.55 \times 10^{-3}\) | 0.976   | 0.999   |
| Heat     | \(1.08 \times 10^{-4}\) | 0.949   | 0.999   |

the GPU memory hierarchy or between the CPU and GPU.

Figure 6 compares the time cost per timestep of the Classic and Swept schemes applied to the Euler equations; the performance trends match those of the heat equation. In this case, the communication costs that the program avoids are significant enough that swept decomposition provides a tangible benefit of 1.5–3.2× despite the extra complexity, management, and memory resources that it requires. This shows that swept time-space domain decomposition is a viable method for complex equations in one dimension on systems with substantial communication costs and various architectures. This contrasts our prior GPU-only results [4] where the swept decomposition scheme always performed slower than classic.

Though we emphasize that these results represent narrow experimental conditions, architectures, and design settings, the regularity of the Swept program performance allows us to present fitted results in Table 1, corresponding to the data points presented in Figures 5a and 6a, that may illuminate and guide future work on this and similar topics.

6. Conclusions

We examined the performance characteristics of design choices that must be made when applying the swept rule to partial differential equations on heterogeneous computational architectures using swept-time space decomposition. These design choices are: how many threads per block—i.e., points per domain—to assign, what proportion of the total domain to assign to a GPU, and how to efficiently and generally store the working array throughout the simulation.

We aimed to answer the primary questions concerning these design choices laid out in Section 3. First, we found that the best number of grid points to
assign to each domain varies with the algorithm, governing equation(s), and grid size. To achieve the best performance on repeated similar runs, any program should be tested over a limited number of time steps and tuned to the best result; however, we recommend choosing a block size that is a multiple of 32 between 96–384. This is consistent with our previous results for the GPU-only implementation of the swept rule [4]. Next, we concluded that while a GPU affinity is best chosen after a similar tuning experiment, for more complex problems an affinity from 40–60 performs well and simpler problems perform best with an affinity between 20–40. Next, there is a significant tradeoff between extensibility and performance associated with the primary data structure and compression scheme applied to the working quantity in the simulation. The lengthening approach offers both performance benefits to the swept decomposition scheme as well as simplified development. Finally, although any conclusions drawn from an experiment on only two nodes are limited, we showed a significant relative improvement over our previous results for the Euler equations using a fine-tuned GPU-only program [4].

Future work in this project will continue adapting the swept rule to higher dimensions, architecture types, and grid formations. For example, while Alhubail and Wang demonstrated the two-dimensional swept rule for CPU-based clusters [3], we have not yet extended this to heterogeneous systems. In addition, we recognize the need to develop new experiments that examine the scaling characteristics of the program as additional computational resources are added. We plan on executing those experiments on cloud systems like Amazon Web Services, Microsoft Azure, or Nvidia GPU Cloud. As we conduct these experiments, we hope to gain greater insight into the factors affecting performance and develop a more robust performance model for swept time-space decomposition.

Acknowledgments

This material is based upon work supported by NASA under award No. NNX15AU66A under the technical monitoring of Drs. Eric Nielsen and Mujeeb Malik. We also gratefully acknowledge the support of NVIDIA Corporation for donated Tesla K40c GPU used for this research.

Appendix A. Availability of material

The software package hSweep v2.0 used to perform this study is available openly [20]; the most recent version can be found at its GitHub repository.
References

[1] V. Alexandrov, Route to exascale: Novel mathematical methods, scalable algorithms and computational science skills, Journal of Computational Science 14 (2016) 1–4, the Route to Exascale: Novel Mathematical Methods, Scalable Algorithms and Computational Science Skills. doi:10.1016/j.jocs.2016.04.014.

[2] M. Allhubail, Q. Wang, The swept rule for breaking the latency barrier in time advancing PDEs, Journal of Computational Physics 307 (2016) 110–121. doi:10.1016/j.jcp.2015.11.026.

[3] M. M. Allhubail, Q. Wang, J. Williams, The swept rule for breaking the latency barrier in time advancing two-dimensional PDEs, arXiv:1602.07558 [cs.NA] (2016).

[4] D. J. Magee, K. E. Niemeyer, Accelerating solutions of one-dimensional unsteady pdes with gpu-based swept time–space decomposition, Journal of Computational Physics 357 (2018) 338–352. doi:10.1016/j.jcp.2017.12.028.

[5] M. Allhubail, Q. Wang, KSIDSwept, git commit e575d73, https://github.com/hubailmm/K-S_1D_Swept (2015).

[6] Q. Wang, Decomposition of stencil update formula into atomic stages, arXiv:1606.00721 [math.NA] (2017).

[7] M. Allhubail, Q. Wang, Improving the strong parallel scalability of CFD schemes via the swept domain decomposition rule, in: 55th AIAA Aerospace Sciences Meeting, American Institute of Aeronautics and Astronautics, Grapevine, Texas, 2017. doi:10.2514/6.2017-1218.

[8] M. Baboulin, S. Donfack, J. Dongarra, L. Grigori, A. Rémy, S. Tomov, A class of communication-avoiding algorithms for solving general dense linear systems on CPU/GPU parallel machines, Procedia Computer Science 9 (2012) 17–26. doi:10.1016/j.procs.2012.04.003.
[9] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, 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, SC ’08, IEEE Press, Piscataway, NJ, USA, 2008, pp. 4:1–4:12.

[10] T. Malas, G. Hager, H. Ltaief, H. Stengel, G. Wellein, D. Keyes, Multicore-optimized wavefront diamond blocking for optimizing stencil updates, SIAM Journal on Scientific Computing 37 (4) (2015) C439–C464. doi:10.1137/140991133.

[11] M. J. Gander, 50 years of time parallel time integration, in: T. Carraro, M. Geiger, S. Körkel, R. Rannacher (Eds.), Multiple Shooting and Time Domain Decomposition Methods, Vol. 9 of Contributions in Mathematical and Computational Sciences, Springer, Cham, 2015, pp. 69–113. doi:10.1007/978-3-319-23321-5_3.

[12] R. Falgout, S. Friedhoff, T. Kolev, S. MacLachlan, J. B. Schroder, Parallel time integration with multigrid, PAMM 14 (2014) 951–952. doi:10.1002/pamm.201410456.

[13] R. D. Falgout, T. A. Manteuffel, B. O’Neill, J. B. Schroder, Multigrid reduction in time for nonlinear parabolic problems: A case study, SIAM Journal on Scientific Computing 39 (5) (2017) S298–S322. doi:10.1137/16M1082330.

[14] S.-L. Wu, T. Zhou, Parareal algorithms with local time-integrators for time fractional differential equations, Journal of Computational Physics 358 (2018) 135–149. doi:10.1016/j.jcp.2017.12.029.

[15] Y. A. Huerta, B. Swartz, D. J. Lilja, Determining work partitioning on closely coupled heterogeneous computing systems using statistical design of experiments, in: 2017 IEEE International Symposium on Workload Characterization (IISWC), 2017, pp. 118–119. doi:10.1109/IISWC.2017.8167766.

[16] L. Clarke, I. Glendinning, R. Hempel, The MPI message passing interface standard, in: Programming Environments for Massively Parallel Distributed Systems, Birkhäuser Basel, 1994, pp. 213–218. doi:10.1007/978-3-0348-8534-8_21.
[17] D. A. Jacobsen, I. Senocak, Multi-level parallelism for incompressible flow computations on gpu clusters, Parallel Computing 39 (1) (2013) 1–20. doi:10.1016/j.parco.2012.10.002.

[18] F. Lu, J. Song, F. Yin, X. Zhu, Performance evaluation of hybrid programming patterns for large cpu/gpu heterogeneous clusters, Computer Physics Communications 183 (6) (2012) 1172–1181. doi:10.1016/j.cpc.2012.01.019.

[19] R. T. Mills, K. Rupp, M. Adams, J. Brown, T. Isaac, M. Knepley, B. Smith, H. Zhang, Software strategy and experiences with manycore processor support in PETSc, SIAM Pacific Northwest Regional Conference (Oct. 2017).

[20] D. J. Magee, K. E. Niemeyer, Niemeyer-Research-Group/hSweep: MS Thesis Official (Jun. 2018). doi:10.5281/zenodo.1291212.