The 2-Dimensional Swept Rule Applied on Heterogeneous Architecture

Anthony Walker, Dr. Kyle Niemeyer[1]

Abstract

This paper contain is a study of the swept rule in two dimensions on heterogeneous architecture.

Keywords: `elsarticle.cls`, LaTeX, Elsevier, template

2010 MSC: 00-01, 99-00

you can do something like: 1. solving multidimensional PDEs for practical problems requires using shared and/or distributed memory systems 2. bandwidth and latency between processing units, particularly in distributed systems, can be a bottleneck. separately (but related), when you have things like GPUs, you want to do as much local work as possible before communicating between the independent processors 3. how have others tried to solve this? 4. what past work has been done specifically on the swept approach? 5. what are the objectives of this study?

1. Introduction

Unsteady multidimensional PDES often require the use of shared and/or distributed memory systems to obtain a solution with practical grid resolution or scale in a reasonable time frame. The solution of many problems at any point in the grid is ineherently dependent on the neighboring points in each dimension. This dependence necessitates communication between computing nodes. Each of these communications incurs a minimum cost regardless of the

[*]Fully documented templates are available in the elsarticle package on CTAN.

amount of information communicated known as network latency. This latency can be siginificantly restrictive on the solution's time to completion, especially when using distributed systems. This barrier to scaling is referred to as the latency barrier and is particularly impactful in large scale simulations advancing many time steps (i.e. ones that requires a large amount of communication) [CITE ALHUBAIL AND WANG]. The barrier a bottle neck in the system which can limit the performance regardless of the architecture.

There are several strategies that exist for latency reduction such as Communication avoidance (CA) algorithms... [READ MORE ON THIS].

In this paper, PySweep, a 2 dimensional PDE solver was implemented and tested on heterogeneous (i.e. utilizes both a CPU and GPU) architecture. This solver is openly avaliable on GitHub at [ADD GITHUB REFERENCE]. This differs from past work in a few ways. The swept rule was originally developed by ALHUBAIL [SWEPT PAPER] who published a 1 dimensional CPU only swept PDE solver which concluded that..[CONCLUSION][CITATION]. This differs from PySweep in multiple ways. The most prominent are the dimensionality, architecture, implementation strategies. [ADD MORE].

These authors followed this work with a 2 dimensional CPU only swept PDE solver which concluded that...[CONCLUSION][CITATION]. PySweep differs from this solver primarily in architecture and implementation. [ADD MORE].

Magee et al. created multiple 1 dimensional PDE solvers which includes a GPU only and heterogeneous solver which this work is essentially an extension of. They concluded that [CONCLUSION][CITATION]. The difference in PySweep is the dimensionality and implementation strategy. [ADD MORE].

The objective of this study are to implement the swept rule in 2 dimensions on heterogeneous architecture and study the performance capabilities of such a solver based on system parameters. The first parameter of interest in the block size Magee et al. showed a dependence on the block size so it is of interest to determine the change in behavior when added another dimension which the solvers are not identitical, the theory is the same and comparison can provide valuable insight into the effects of the extra dimension. Magee et al. likewise showed a performance dependence on the GPU affinity, Array size, and [READ PAPERS].

The effect of added dimensionality on these parameters is a pragmatic interest and can be considered from multiple perspectives. The primary goal is speed up of simulations requiring high performance computing (HPC) through the reduction of network latency. In [CITATIONS] many motivations are described from aerodynamics, compressible flows, and ... to other applications such as heat diffusion and so on [FIX ME]. While many simplifications exist to reduce the dimensionality of these problems, the most practical approach is what is experienced in reality which is 3 dimensions. While this solver is not 3 dimensional, it is a step in that direction which can provide insight to the performance of the swept rule in multiple dimensions. In the case of this specific solver, it can provide insight to the performance of the heterogeneous solver in 2 dimensions that employ this latency reduction technique. This insight can offer the chance to optimize system usage and promote faster design and prototype of the aforementioned applications.

In the event that time is not the primary concern, avaliable resources is another important consideration. Computing architecture typically comes with a hefty price tag with the cost of [WELL KNOWN SC AND ITS PRICE][CITATION]. [MAYBE ADD SOME OTHER ARCHITECTURE COSTS]. Network latency is maximized in applications requiring a lot of communication because each communication takes a finite amount of time regardless of the data size. However, that does not mean that improvement cannot be obtained on smaller systems. This claim is supported by findings from Magee et al. as they tested their code on a work station with a single GPU and CPU and obtained speed up [CITATION]. While this is not the primary focus, an optimized solver that reduces latency would require less computing resources and more practical applications could potentially be solved on smaller less costly computing clusters.

Hopefully, it is clear at this point that latency reduction is important in high performance computing and scientific applications from multiple perspectives as this is the intention of this work. The work in this paper is presented in a traditional manner as follows. The implementation of the swept rule will be discussed along with the design decisions, strategies applied, and restrictions with respect to this specific code. The testing procedure and guidelines for code usage will be presented. The results of testing the solver will be discussed. Finally, the paper will close with conclusions and recommended future work.

2. Implementation

The implementation of the 2D swept rule on heterogeneous architecture, referred to as PySweep on GitHub, involves multiple languages, architectures, strategies, and problems which will be discussed along with decisions made during the implementation process. The code was developed on a workstation with 2 Nvidia GPUs (Tesla K40c and Quadro K620) and a CPU with 16 cores. It was tested on a small computing cluster at Oregon State University which includes (ADD OSU CLUSTER GPU AND CPUS HERE). Network communication over Infiniband? The Tesla K40c.. The Quadro K620.. The CPUs... [FIX THIS]

To run the code, the following inputs are required. These inputs are described here for reference to future discussion of the implementation. an array of initial conditions with dimension (v, x, y) where v is the number of variables necessary to solving the system. The x and y dimensions of this array are required to be evenly divisible by given the block size (b_s). This is a necessary restriction to prevent added logic for handling blocks of varying sizes which would be far more complicated with little added benefit.

The next argument is a tuple with the first 3 arguments being $(t_0, t_f, \Delta t)$ and subsequent arguments being anything necessary to solve the given problem (e.g. when solving the heat diffusion equation the thermal diffusivity (α) is a necessary contant, so the arguments may be something like $(t0, tf, dt, dx, dy, \alpha)$). These arguments will be passed to a user defined function subsequently described. The next argument is a tuple which contains the time scheme order (O_{ts}), the number of points in on each side of a central point in a stencil (N_{pts}), b_s, the GPU affinity (a_g), the GPU source code (S_g), and the CPU source code

(S_c) in that order. The given N_{pts} is a single integer which ensures that the stencil size is odd and symmetrical. The source codes provided are required to have 3 functions. A CPU step function written in python and named step. This function must accept 4 inputs which are the current local array of the system, an Iterable object containing the point indicies to be solved, the current swept time step and a simulation time step. Another CPU function is required to set any global CPU and GPU variables that are to be defined at runtime. This function must be written in python, named "set_globals", and accept as arguments the first input tuple described, a source module created by PySweep, and a boolean which dictates whether the rank will use a CPU or GPU to solve it's block(s). The GPU step function must be written in CUDA and accept as arguments a shared array of floats, an index integer, and a global time step integer. The given shared array is an originally 4 dimensional (time, variables, x, and y), flattened array. The index integer selects a point from this given array and the global time step integer provides the user with a value to assist in implementation of time schemes requiring multiple steps (e.g. Second Order Runge-Kutta).

The block size is restricted by both the code and the hardware used. The code accepts only square blocks with even block sizes. The maximum size is limited by the GPU's maximum threads per block (T_{max}) and, the minimum size is limited by the spatial stencil. This can be expressed as

$$\sqrt{T_{max}} => b_s >= 2(N_{pts}+1). \tag{1}$$

The square root is a result of the system being two dimensional. For example, in the case of the work station that PySweep was developed on, the maximum threads per block is 1024. This means that the block size can be a maximum of 32 because the block must be a 2 dimensional square. The datatype is the last input variable of importance. This is set to a numpy $float32$ but can be adjusted as necessary. However, adjustment may impose block size or other memory restrictions. Further discussion on these potential restrictions can be found in later paragraphs along with discussion on other restrictions. Finally, there are other avaliable inputs to allow the user to control the filename of data written out and any GPUs (by device id) to be excluded from the calculation.

PySweep was implemented primarily in python as it is a simple yet powerful high level programming language that has a multitude of open source packages used for scientific computing and is frequently used for such applications. There is also a well known GPU computing package, PyCuda, that provides a simple interface for working with Nvidia GPUs. The GPU kernels were specifically written in CUDA in lieu of using GPU Arrays (A PyCuda abstraction) due to the results of simple testing and comparsion between the two strategies. The GPU Array was significantly slower in solving a simple test function so utilization of CUDA was justified. It is believed that this slowdown is due to the GPU Array requiring more frequent communication with the CPU as it performs operations on the CPU. Since a major part of the project was speed improvement through reduced communication, it was appropriate to write kernels in CUDA. The CPU parallelism was implemented with mpi4py, a MPI (Message Passing Interface) package for python, so that the code can be applied to distributed computing systems. Several other open-source packages were used but are not important to this discussion. A list of these packages can be found in the PySweep GitHub repository.

PySweep is capable of running across GPUs and CPUs given a GPU affinity which designates the amount of work to be handled by the GPU out of the total work (i.e. an affinity of 1 means that the GPU handles 100% of the work). This is expressed as

$$a_{gpu} = \frac{W_{gpu}}{W_{total}}. \tag{2}$$

This does not account for the number of GPUs and CPU cores avaliable but simply divides the given input array on a column basis. The GPU portion and CPU portion obtained from affinity based decomposition are then divided amongst the avaliable ranks of each architecture type. For example, the initial workstation that the code was written on has 2 GPUs. If both GPUs are included and the code is run with 6 MPI processes then 2 of the 6 processes will be GPU ranks (i.e. processes controlling the GPUs) and the remaining processes will be CPU ranks. This decomposition strategy requires the array dimensions (n, m) be equally divisible by the given block size. However, this imposed restriction does not guarentee good load balancing as ranks are assigned blocks strictly based on the number of blocks and ranks avaliable. Caution should be exercised when selecting block, rank, and array sizes to maximize potential speed up. a few rough guidelines for sizes will be discussed later with the results. If there are too many ranks provided, the code will automatically terminate ranks that are not assigned blocks.

To recap, the swept rule is a latency reduction technic that focuses on obtaining a solution to unsteady PDEs at as many possible locations and times prior to communicating with other computing nodes (ranks). Discussion of the swept rule can at times be convoluted as there are time steps, swept steps, and other actions that can be referred to as steps. To simplify this, the primary two actions of the code will be referred to as swept steps. Individual time steps and intermediate time steps such as those that occur during multi-step schemes will be referred to as so. The other actions that occur will be referred to as phases of the simulation.

There are two major steps involved in the process that can be separated for clarity. The first step is the calculation step which occurs in multiple phases during the swept simulation. The next step is the communication step which also occurs during multiple phases. However, in this implementation it is not necessarily communication between ranks but assignment of each rank to a region and shifted region for the different solution phases. These regions correspond to a section of the original array which is stored in CPU shared memory; a capability implemented in MPICH 3 or later [ADD CITATION HERE]. a shared GPU memory approach was implemented in [CITE DAN AND KYLE] which showed benefit. This same strategy is implemented on the GPU and provided the reasoning for adaptation to the CPU portion of the solver. Once the region is determined, each rank then uses its region or shifted region to locate and copy the appropriate points from the shared array into a local array which computation is performed on. The corresponding updated values are then written back into the shared array. Certain values are also shifted in the array and written to disk at select times through out the simulation. In summary, the communication step consists of managing the values in the shared memory array and writing to disk. The computation step consists of solving the appropriate points. The exact progression of thse steps vary depending on which phase of the swept solution the simulation is in.

During the swept solution process there are 4 phases that occur. In the code, these phases are referred to as UpPyramid, Bridge (X/Y), Octahedron, and DownPyramid so that convention will be continued here. They are so named as these shapes are produced during the solution procedure if you visual the solutions progression in 3 dimensions (x,y,t). The specific shape that forms is a function of the spatial stencil and time scheme chosen to solve a given problem. Note that both the communication and calculation step have a role in each of these phases. The solution process begins with a "pre-sweep" phase which includes decomposition, rank handling, and the determination of communication regions which includes regions to read and write for the appropriate phases.

The process then enters it's first calculation step where the UpPyramid is solved. This height of the UpPyramid (time) is dependent on the mimimum spatial dimension of the block size. This dependence is the reasoning for restricting the blocks to be square in x and y; The added complexity of managing rectangular blocks simply does not add value to the solver. The number of steps in the UpPyramid (H_u) can be determined from N_{pts} by

$$H_u = \frac{x}{2N_{pts}+1} + 1. \tag{3}$$

The extra step is avaliable due to the presence of ghost nodes. A dynamic programming approach is used on the CPU portion to calculate the UpPyramid in lieu of conditional statements. Specifically, the indicies of each block for each time are stored locally in a set which is accessed during solving. The GPU portion implements a conditional statement to accomplish this same task because it's speed typically dwarfs that of the CPU regardless. If the code were to eventually be optimized, it could be beneficial to consider a dynamic programming approach such as that implemented on the CPU. In both cases, conditional or dynamic, the code removes N_{pts} from each boundary after every time step. If the time scheme requires multiple intermediate time steps then these points are removed after each intermediate step. The initial boundaries of the UpPyramid are the given block size. The final boundaries are of little importance but could be determined from the aforementioned variables. After the UpPyramid is completed, the appropriate points are copied to the shared memory array using pre-determined regions which completes the first part of the communication step.

The next step in the process is referred to as the Bridge which occurs independently in each dimension. The number of bridges is equal to twice the number of UpPyramids. The height of the bridges can be determined as

$$H_b = H_u - 1. \tag{4}$$

The dimension in which the bridge grows as time passes is the reference dimension for the bridge (e.g. the bridge that becomes larger in the x dimension is refered to as the X-Bridge). To complete the aforementioned communication process, Bridge shifted regions in each dimension are used to copy data from shared memory to local arrays. Each rank then computes both a X-Bridge and Y-Bridge where a very similar GPU and CPU strategy to the UpPyramid is taken for both bridges. The primary difference is that 1 boundary of each bridge grows in lieu of both boundaries shrinking. Interestingly enough, with the given restrictions, the X-Bridge sets are the inverse of the Y-Bridge sets (i.e. the X-Bridge starts with the set that the Y-Bridge ends on). This is beneficial as only one set of swept step indicies needs to be determined in order to obtain the indicies necessary to solve both bridges. Also, the Bridge phase differs in the fact that it will be executed multiple times over through the simulation as needed to obtain a complete solution between the specified initial and final time. After the calculation is completed, each rank copies the bridge values into the shared memory array to begin the next phase.

The next phase of the process is the Octahedron which is essentially super position of the DownPyrmaid and UpPyramid. The DownPyramid indicies are also determined during the pre-sweep phase and they are super imposed with the UpPyramid indicies to form the Octahedron indicies. The height of the DownPyramid can be determined as

$$H_d = H_u - 1 \tag{5}$$

The height of the Octahedron H_o is then a sum of H_u and H_d. The DownPyramid alway begins with a block that is $2N_{pts}$ and grows by N_{pts} on each boundary with every passing time step. The start is a natural consequence of removing these points during the UpPyramid phase. It is completed upon passing the top of the previous UpPyramid or Octahedron at which time the next UpPyramid portion of this phase begins. The entire Octahedron is calculated in the same fashion as the UpPyramid on both CPU and GPUs. While the steps are described separately for clarity, they are performed in a single calculation step without communication between ranks. The Octahedron step is also a recursive step which is applied as many times as necessary to complete the desired simulation.

The final phase of the swept process is the DownPyramid which is fairly well described during the Octahedron phase. This is the ending phase of the simulation and is only performed a single time. The strategy is the same as that of the UpPyramid for both the CPU and GPU. The only difference is that all boundaries grow. To summarize the swept phases, the UpPyramid is calculated a single time; The Bridges and Octahedron phases are then repeated until the simulation has reached a value greater than or equal to that of the desired final time. The actual final time is determined by the number of ”swept” steps, (N_{ss}) or phases that the simultation is run. This is determined by the given final time and H_o which is the minimum time possible of a simulation. It is this way because the simulation only stops after the completion of a phase. These phases occur on both the GPU and CPU with respect to the given affinity. In between each calculation step, a communication step occurs which consists of shared memory data management and writing out to disk.

The shared memory data management of the communication step as well as the writing to disk involve a couple of nuances worth mentioning. It includes shifting of the data which is a strategy implemented for handling boundary blocks in the array. PySweep was implemented with periodic boundary conditions based on the targeted test problems. The boundary blocks of the array form half of the shape it would normally in the direction orthogonal to the boundary (e.g. During the Octahedron phase on the boundary where $x = 0$, only half the Octahedron will be formed in the x direction). As expected, the corner will form a fourth of the respective shape. In lieu of added logic for handling these varying shapes, a shifting strategy was developed which allows the majority of the same functions and kernels to be used. The strategy works by writing the data to designated points at the end of the array in each dimension. The shared memory array is exacly $2N_{pts} + b_s/2$ points larger in x and y than the given initial conditions array. The shifting of data occurs after every Octahedron phase so that the simulation oscillates between use of its region and shifted region. The boundary points are able to be solved as if they were in the center of a block with this strategy. This strategy comes at the expense of determining the extra regions and storing the extra points in memory. However, the added memory expense of this is minimal due to the strategy for writing to disk and block size restriction. It also mitigates the need for added logic to handle the natural shifting of the shapes and boundaries.

PySweep writes to disk after every Octahedron step as it is the ideal time. The code uses parallel HDF5 (h5py) so that each rank can write it's region to disk independently of other ranks. The shared memory array is the height of an Octahedron in time plus the order of the time scheme so that the intermediate steps of the scheme may be used for future steps if necessary. The first H_u steps following the O_{ts} steps are written to disk if the step completes the cycle of the specific time scheme (i.e. it is the final step in the time scheme and the values are the desired result). The data is then shifted down in the time dimensions of the array and so that the next phase can be calculated from the region or shifted region. When the write phase occurs after the shifted calculation, the data is shifted back prior to writing so that the writing does not require any conditional statements. This is also necessary for the next calculation step so it is not an added expense.

In summary of this swept rule implementation, PySweep is an extensible PDE solver that implements the swept rule in 2 dimensions on heterogeneous architecture. It consists of 2 primary steps which are calculation and communication. The calculation computes points based on a predictable patterns referenced as the phases of the simulation. The communication step is atypical as it does not communicate persay but shifts the perspective of each rank and copies the working data to a local array and then writes it into a shared memory array after the calculation step [MAKE SURE THIS IS ACCURATE, CITE]. This shifting strategy provides simple communication logic at a minimal memory expense. There are multiple phases where both steps occur during the simulation which are the UpPyramid, Bridge (X/Y), Octahedron, and DownPyramid. The solver has a few restrictions based on architecture and implementation which have been previously described. It is currently implemented for periodic boundary conditions only but updating it for other conditions would be fairly straight forward. The solver is also capable of handling given CPU functions and GPU kernels so that it may be used for any desired application that falls within the guidelines presented here. There are some restrictions on the time and spatial schemes namely that the spatial stencilbut it would be fairly straight forward to handle other cases. PySweep could certainly be further optimized but it is presently sufficient to test and determine the performance capabilities of the 2 dimensional heterogeneous swept rule in comparison to a traditional solver.

3. Testing

4. Results

5. Conclusions

6. Acknowledgements

References