ELSEVIER

Available online at www.sciencedirect.com

ScienceDirect

Journal of XXX xx (2016) 1–19

Journal
Logo

Efficient Large-scale Parallel Stencil Computation on multi-GPU and multi-Core clusters

Zhengchun Liu[a,b], Kalyan S. Perumalla[a,*]

[a] *Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA*
[b] *Computer Architecture & Operating Systems, University Autonoma of Barcelona, Barcelona, Spain*

Abstract

The mechanism is designed to accommodate the hierarchical organization as well as heterogeneity of current state-of-the-art parallel computing platforms. We describe our implementation and report preliminary performance results on two distinct parallel platforms: CUDA threads on multiple, networked graphical processing units (GPUs), and OpenMP on multi-core processors. Message Passing Interface MPI is used for inter-GPU as well as inter-socket communication on a cluster of multiple GPUs and multi-core processors. Our test results on Titan supercomputer indicate the benefits of our latency-hiding scheme, delivering as much as 9x faster than conventional implementation without latency-hiding scheme. Agent-based model and scientific model on solving partial differential equation using finite difference method were used as case studies to explore the latency-hiding scheme in hierarchical organization. The case studies have been scaled to 1000 NVIDIA Telsa K20X GPU accelerators and 16,000 CPU cores with total number of 216 billion cells to test the scalability of the framework template.

Keywords: Computational hierarchy, Multi-core, Multi-GPU, Latency hiding, CUDA

1. Introduction

Computer simulation has become one of the most generally applied approach to studying natural systems in physics, chemistry and biology, and human systems in economics and social science. Efficient simulation methods enable researchers to study more scenarios in short time frame. Iterative stencil loops (ISL) are commonly used in scientific programs to implement relaxation methods for numerical simulation and signal processing [1]. Stencil computations are an important class of codes used in a variety of application domains ranging from image and video processing to simulation and computational science applied in several areas of natural science. It is an attractive powerful methods to explore the challenges of massive parallelism in solving partial differential equations. As investigated in Ref. [1–5], tiling is a well-known technique to localize ISL computation and there are usually halo regions need to be updated, exchanged and synchronized among different subdomain processing elements in a parallel architecture.

[☆] Corresponding author at: Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA. Tel.: +1 (865) 241-1315; Fax: +1 (865) 576-0003.
Email address: `perumallaks@ornl.gov` (Kalyan S. Perumalla)

1.1. motivation

Computing devices, e.g., CPU, GPU and Intel MIC, are becoming faster (improved at the rate of roughly 50% per year [6]) and less expensive according to Moore's law. However, memory latencies have not improved as dramatically (improved at a rate of only 7% per year [6]), and memory access times are increasingly limiting system performance. This is a phenomenon known as the *Memory Wall* [7, 8]. Thus, the gap between CPU processing speed and memory access rate is becoming even bigger. To bridge this gap, both efforts in system architecture design and user application are required [5]. As defined in Ref. [9], main memory in a parallel computer is either shared memory (shared between all processing elements in a single address space), or distributed memory (in which each processing element has its own local address space). Distributed memory refers to the fact that the memory is logically distributed, but often implies that it is physically distributed as well.

Yet with the large and growing gap between computation speed and inter-node communication speed in a grid environment, more methods need to be developed to balance out computation and communication. Memory system performance is largely captured by two parameters, latency and bandwidth. Latency is the time from the issue of a memory request to the time the data is available at the processor, and bandwidth, mostly determined by hardware specification, is the rate at which data can be pumped to the processor by the memory system. This work concentrates on the latency hiding of stencil computation model and agent-based model problems on multi-GPU and multi-Core hierarchical organization.

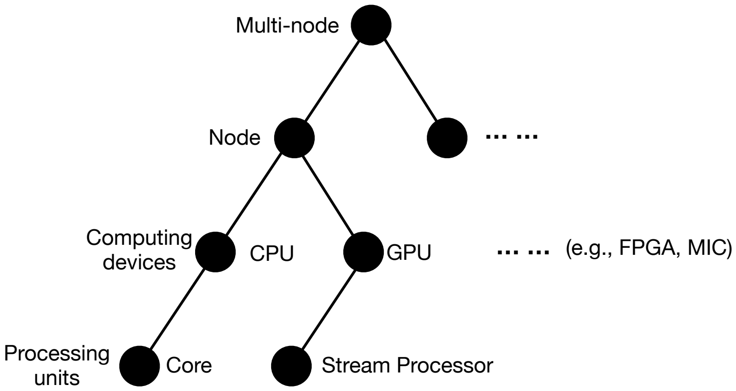


Figure 1: The state-of-the-art parallel computing platform architecture. Computing tasks are hierarchically decomposed. At one level, the compute capability of difference items are different.

The sate-of-the-art parallel computing platform normally contains multiple compute nodes. Those nodes are interconnected by high-speed interconnection, and each node contains multiple CPU cores and accelerators such as GPGPU, Intel MIC and FPGA. Each of these components present significant performance bottlenecks. The question of how best to utilize these resources is an important issue as well. As shown in Figure 1, a large-scale simulation problem needs to be decomposed, i.e., the problem will first decomposed to subdomains for compute nodes, then the subdomain job will be further decomposed to tasks for computing devices on each of the node, finally tasks will be broke down for processing units according to the feature of computing devices.

To deal with large stencil computation model, data parallelism is commonly used for efficiency. Particularly, on a distributed system, each processor holds a subset of the problem domain, referred to as problem subdomains. Each subdomain is padded one or several boundary layers, which are usually called ghost cells [5]. A thorough investigation on the effectiveness of boundary layers with variety of model could help simulation developers more efficiently utilize the parallel computing platform.

1.2. background

Our previous study [10] explored the computation vs. communication trade-off continuum available with the deep computational and memory hierarchies of extant platforms, and presented a novel analytical model of the trade-off. It was focused on 2D simulation, and for those models which update elements according to one-layer neighbors (i.e., computing-dependency order equals one). But in many applications, such as in climate modeling and seismic wave propagation model, updating elements according to three or even more layers are commonly used in a 3D environment. This study continually explore the computation vs. communication trade-off in 3D environment with different computing-dependency order. An analytical model will be provided to find out the optimum computation-communication trade-off point.

1.3. contributions

This study presents a latency-hiding mechanism designed to exploit and seamlessly adapt to the hierarchical organization and heterogeneity of the state-of-the-Art high performance computing platforms. The latency hiding is based on the well-known principle of computation vs. communication tradeoff (or, the duplication of some computation to gain some concurrency to offset communication latencies)[10]. We call it the "*B+2R latency-hiding scheme*" (defined in previous work [10], will be referred by *B2R* in this context), where $B = \{B_x, B_y, B_z\}$ refers to the size of subdomain, and R represents the size of ghost zone. Note that, R denotes the times of computing-dependency order. For example, if a model with computing-dependency order as δ, our B2R scheme will pad $R\delta$ layers to each dimension of the subdomain. These notations will be used throughout the entire article.

Different with our previous study [10], this work extended the B2R latency-hiding scheme to 3D grid environment. In addition, this work also investigated the effectiveness of B2R with different layers of data-dependence (referred by δ in this context) on updating each of the elements' state. An analytical model to easily figure out the optimum B2R parameters, a framework template to update/synchronize subdomain boundaries, and auto-tuning balance scheme for heterogeneous computing devices in grid environment were provided for easy coding. In summary, the main contributions of this work are: compare with our previous study in Ref. [10]: (1) we extended the scheme model from 2D to 3D; (2) from fixed stencil ratio $\delta = 1$ to customizable δ; (3) from computing via CPU or GPU to with both CPU and GPU, and designed an auto-tune self-balancing task-alignment scheme for CPU and GPU. Compared with other related works that investigated either in grid environment with multi-core or in GPU with CUDA, this study explored the latency hiding mechanism to accommodate the hierarchical organization and heterogeneous computing platform. Furthermore, one commonly used benchmark agent-based model, multiple implementations of heat diffusion model with different order of accuracy (vary δ) and a computationally expensive seismic wave propagation model were used to benchmark the proposed latency hiding scheme.

The rest of the paper is structured as follows: section 2 gives a literature review on related work. A detailed description of our latency hiding scheme is given in section 3, in which an analytical performance model is formulated for analyzing the potential optimum configuration. The section 4 describes the implementation details. Then, a comprehensive performance study with several applications are presented in section 5. Finally, section 6 closes the article with our conclusions and potential future contributions.

2. Related work

Stencil codes are compute-intensive algorithms, in which data points arranged in a large grid are being recomputed repeatedly from the values of data points in a predefined neighborhood. This fixed neighborhood pattern is called a stencil [3]. Stencil codes see wide-spread use in computing the discrete solutions of partial differential equations and systems composed of such equations. Given this, how to efficiently conduct stencil codes in the state-of-the-art parallel computing platforms has attracted many investigations. J. Meng et al. [2] thoroughly investigated the effectiveness of ghost zone in CUDA parallel computing platform. A framework template that can automatically incorporate ghost zones to ISL applications in normal CUDA code and optimize it with the selection of trapezoid configurations was proposed. Furthermore, J. Meng et al. [2] also found that smaller stencils with smaller halo width can benefit more from the trapezoid technique, they tested the performance with 1D, 2D and 3D model, the 1-D PathFinder benefits the most, followed by 2-D HotSpot and Poisson model, and the 3-D Cell (Game of Life) does not benefit at all. They finally concluded that the benefit of ghost zones may be more significant for architectures that easily allow for larger tile sizes. However, larger thread block size and also larger shared memory (to host bigger subdomain data and ghost zone data) seems neither possible nor necessary on GPGPU.

Considering the case of using finite difference method to solve partial difference equations on distributed memory computers, processor subdomain boundaries must be updated at each time step. This boundary update process involves many messages of small sizes, therefore large communication overhead. D. Chris et al. [5] investigated the ghost cell layers padding scheme, and thus updates boundaries much less frequently – reducing total message volume via grouping small messages into bigger ones. However, their work has only been studied for message-passing systems in a grid computing environment, it did not involve the state-of-the-art heterogeneous parallel computing platform like GPGPU. With respect to large-scale scientific simulation such as seismic wave propagation, which involves solving multiple partial differential equations on large 3D domain, Ref. [11] presented an implementation and optimization investigation on GPU with CUDA, but they did not explore performance on multiple GPUs. Ref. [12] explored an implementation of earthquake wave propagation model on petascale supercomputer, their implementation has been scaled to more than two hundred CPU-cores. However, they only considered one layer in ghost zone and only with CPU. Multiple ghost zone layers may achieve better tradeoff between communication and computation on large distributed memory system, and with multi-GPU should further accelerate the simulation.

Although the increasing computational speed of GPUs make their use for stencil computations an interesting goal, as numerous publications attest that achieving highly efficient implementations is often nontrivial [13]. Ref. [1] presents a compiler framework for automatic tiling of iterative stencil loops, with the objective of improving the cache performance. There are also some work on performance model based auto-tuning of stencil computations. Such as Hu et al. [13] proposed an analytic performance model for stencil codes on GPUs. Lutz et al. [14] focused on abstracting the complexity of multi-GPU (on one computing node) programming for stencil computation. They demonstrated that adaptation to the given PCI express configuration is a significant factor in achieving high performance overall. Ref. [15] developed a number of effective optimization strategies, and built an auto-tuning environment that searches over optimizations and their parameters to minimize runtime, while maximizing performance portability on multicore architectures. Authors highlighted that auto-tuning is critically important for unlocking the performance potential across a diverse range of chip multiprocessors. Target at keeping the programmer's effort to a lower overhead without significant sacrifice in performance., Zhang et al. [16] developed and evaluated search and optimization techniques for auto-Generation auto-tuning 3D stencil computations on GPUs.

3. Approach description

3.1. Task decomposition

The data parallelism to parallelize the large simulation problem to many computing nodes was commonly used for efficiency. Using this kind of decomposition, the domain is decomposed into small subdomains and distributed among all computing nodes; therefore, each processor solves its own subdomain problems. Since each computing node has CPU and accelerating devices, the subdomain will be further decompose for computing devices. The full 3D domain decomposition process is shown in Figure 2.

More specifically, for decomposition on CPU and GPU based heterogeneous parallel computing platform, the 3D domain is partitioned into a number of subdomains, and each subdomain is reserved for computation on a computing node. Here we use X, Y, Z to represent columns, rows and depth, respectively. The memory for holding the subdomain is managed in row-major, i.e., the first dimension X (column) changes the fastest and the last dimension Z (depth, in this case) changes the slowest. As shown in Figure 2, on each computing node, the subdomain was partitioned into two parts (along Oz), N_{GPU} and N_{CPU} slices for GPU and CPU respectively. Suppose that the 3D subdomain is represented as (N_{bx}, N_{by}, N_{bz}), then CPUs will hold $(N_{bx}, N_{by}, N_{CPU})$ and GPU holds $(N_{bx}, N_{by}, N_{GPU})$, where $N_{bz} = N_{CPU} + N_{GPU}$. Lastly, we need to decompose the CPU domain for CPU cores. As the memory address varies the slowest in Oz direction, to optimize cache hits, the job as partitioned with 1D along Oz direction, i.e., one core responses for several

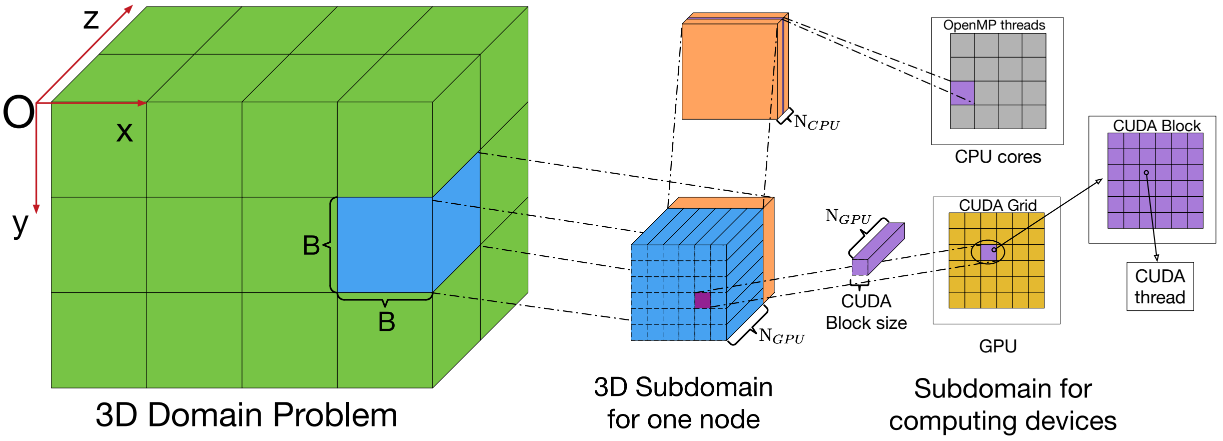


Figure 2: Process of 3D domain decomposition on CPU/GPU based multi-node heterogeneous computing platform: first decompose for nodes, then decompose for computing devices (GPU and CPU, along Oz direction, N_{GPU} and N_{CPU} respectively) on the node, then decompose for processors (CPU cores and GPU streaming multiprocessors).

neighboring Oxy slices. While to partition for GPU streaming multiprocessors (SM), we choose 2D decomposition on the Oxy plane, then each SM takes over the entire computation in Oz direction.

In this decomposition, there is not a node to host the total problem domain. This could make it scalable for very large-scale simulation domain, its scalability for large problem size was investigated in subsection 5.5. We emphasize that the domain and subdomain are not essential to be cubic, can be cuboid or even 2D plane ($N_{bz} = 1$). Other decomposition methods for specific problem, such as the two-layer decomposition described in Ref.[4] that each subdomain takes over the entire computation in the Oz direction could be achieved via setting 1 computing node in Oz direction.

3.2. Ghost expansion

In stencil computation or agent based model, updating one cell at time t requires the current stat of its neighbors, the range of neighbors depends upon specific model (in stencil computing, it is determined by the order of accuracy of the numerical required). In this article, we use δ to represent range (i.e., at a Chebyshev distance [17] of δ) of neighborhood. Thus, each cell has $(2\delta + 1)^n - 1$ and $2\delta n$ cells in a n-Dimension environment ($n = \{1, 2, 3\}$) for Moore neighborhood (e.g., agent-based model), and Von Neumann neighborhood (e.g., finite difference method) respectively. As we propose to synchronize less often with bigger message size in order to reduce communication latency, $R\delta$ layers will be padded to subdomains decomposed in Figure 2 at every level (node, devices, processing unit). Computation on local subdomain can then be increased by R iterations before having to re-synchronize with neighbors. This will result in more computation time thus there will be a trade-off between extra computation and reduced communication latency. The feasibility of this idea and an in-depth analysis of the method will be given in subsection 3.4.

3.3. Boundaries updating

As discussed in subsection 3.1 and subsection 3.2, ghost padding must be synchronized after R local iterations. Table 1 lists the synchronization process in a 3D decomposition. This diagonal communication elimination technique was first implemented in [18] in a slightly different form and then explored in [5] in 2D scenarios. It requires only communications with von Neumann neighbors, and thus can reduce the number of messages from 8 to 4 in 2D, and 26 to 6 in 3D scenario. To be clear, as illustrated in Figure 3, in this technique the corner blocks are moved twice to reach their final destinations in 2D, and the cubic diagonal blocks are moved 3 times to reach their final destinations. Since this technique requires step-by-step synchronizing, according to our experiments, it performs better for small message size than synchronizing directly with all Moore neighbors concurrently. No significant difference was found with big size messages. Its performance also depends on network bandwidth and topology. Since exchange messages between diagonal subdomains are normally small, $(R\delta)^n$ in nD ($n = \{1, 2, 3\}$) case, this diagonal communication elimination technique was used in this study. It is worth to note that if subdomain takes over the entire computation in any of the Ox, Oy, Oz direction, these directions will do not need synchronization. For example, as shown in Figure 2, the CPU/GPU subdomain on the node was partitioned only in Oz (takes entire in Ox and Oy), thus it only needs the third step for updating ghost padding along Oz direction. Similarly, the decomposition for GPU SMs needs synchronization in Ox and Oy direction.

Table 1: Boundary updating steps and message size.

Step	Direction	Valid before updating	Updating size	Valid after updating
1	Ox (Left, Right)	$B_x \times B_y \times B_z$	$R\delta \times B_y \times B_z$	$(B_x + 2R\delta) \times B_y \times B_z$
2	Oy (Up, Down)	$(B_x + 2R\delta) \times B_y \times B_z$	$(B_x + 2R\delta) \times R\delta \times B_z$	$(B_x + 2R\delta) \times (B_y + 2R\delta) \times B_z$
3	Oz (Front, Back)	$(B_x{+}2R\delta){\times}(B_y{+}2R\delta){\times}B_z$	$(B_x{+}2R\delta){\times}(B_y{+}2R\delta){\times}R\delta$	$(B_x{+}2R\delta){\times}(B_y{+}2R\delta){\times}(B_z{+}R\delta)$

More specifically in Table 1, take compute-node level (hierarchy in Figure 1) synchronization as an example and assuming it decomposes in all three directions. There are $B_x \times B_y \times B_z$ valid domain left after R local iterations. $R\delta$ layers data surrounding the valid domain need to be refilled with state information from neighboring subdomains. As shown in Table 1, it requires three consecutive steps to update all boundaries from its neighbors. After synchronization, execution can continue for another R iterations. An illustration of the three-step synchronization was shown in Figure 3.

Figure 3 illustrates the steps on sending data to neighbors. Note that, receiving processes are the same, each node will receive the same amount of data from its neighbors, and the received data will be padded to the corresponding ghost zones. Message Passing Interface (MPI) non-blocking communication functions (*MPI_Isend* / *MPI_Irecv*) were used for this synchronization process.

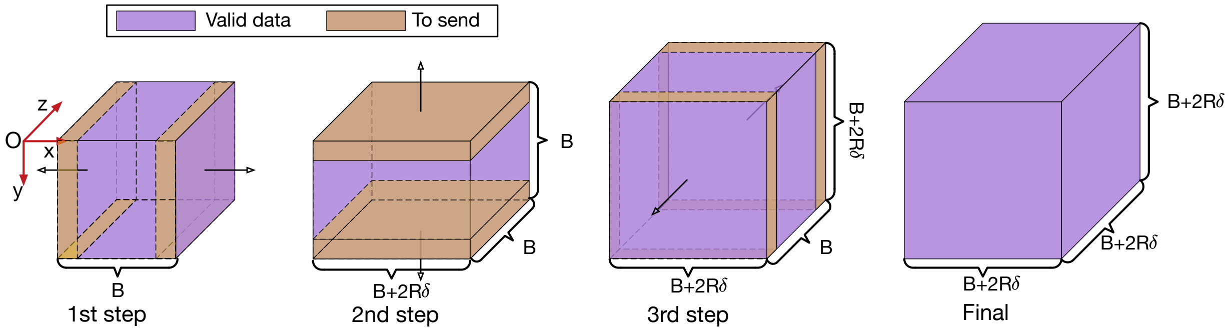


Figure 3: Illustration of updating ghost cells in 3D decomposition. 6 ghost cells blocks are from immediate neighbors.

3.4. Analytical model

In the following, we examine the feasibility of this idea and give an in-depth analysis of the method. To simplify the model description, we assume that all the subdomains are cubic. All the computation and communication time models are formulated as $y = a_{(c,m)}^{(C,G)}x + b_{(c,m)}^{(C,G)}$. Where, the subscript c and m of the parameters denote computation and communication respectively, the superscript C and G represents CPU and GPU respectively. x represents operation (compute or communicate) size, y is the operation time.

When computing on CPU, only the stencil operations within the valid tile are performed in each iteration, i.e., after one iteration, 2δ layers will lose in each dimension. Thus the total computation time for updating R simulation steps on CPU can be formulated as:

$$F_c^{CPU} = a_c^C \sum_{i=1}^{R} [B + 2(R-i)\delta]^3 + b_c^C = a_c^C[2\delta^3 R^4 + (4B\delta^2 - 4\delta^3)R^3 + (3B^2\delta - 6B\delta^2 + 2\delta^3)R^2 + (B^3 - 3B^2\delta + 2B\delta^2)R] + b_c^C \tag{1}$$

Where, a_c^C is the unit time for updating one cell in the grid. Equation 1 can be easily extended to non-cubic grid (different $B \in \{B_x, B_y, B_z\}$ for each domain). However, when execute on GPGPU platform, although the subdomain size decreases 2δ after each iteration in each dimension, as investigated in Ref. [2], the changing boundary-testing increases the amount of computation and leads to more control-flow divergence within warps, which undermines SIMD performance. Therefore, the full domain can be computed as if its tile size does not change along with the loops. At the end, only those elements that fall within the boundary of the shrunk tile are committed to the device memory. This manner results in the following computing time model:

$$F_c^{GPU} = R\{a_c^G[(B + 2R\delta)^3] + b_c^G\} = a_c^G[8\delta^3 R^4 + 12B\delta^2 R^3 + 6B^2\delta R^2 + B^3 R] + b_c^G R \tag{2}$$

To analyze the communication time F_m, we assume it can be approximated by a simple $(message{-}volume)/bandwidth + latency$ model. i.e.,

$$F_m = a_m x + b_m \tag{3}$$

Where a_m denotes the reverse of bandwidth (time per byte), b_m represents the latency. Both a_m and b_m are platform-specific communication overhead constants. Thus the network communication (F_m^{MPI}) and memory transferring time ($F_m^{H2D/D2H}$) for one synchronization operation (for R local iterations) can be expressed as:

$$F_m^{MPI} = a_m^M[(B + 2\delta R)^3 - B^3] + b_m^M = a_m^M[8\delta^3R^3 + 12B\delta^2R^2 + 6B^2\delta R] + b_m^M \tag{4}$$

$$F_m^{H2D/D2H} = a_m^G[(B + 2R\delta)^3 + B(B + 2R\delta)^2] + b_m^G = a_m^G[8\delta^3R^3 + 16B\delta^2R^2 + 10B^2\delta R + 2B^3] + b_m^G \tag{5}$$

Note that MPI communications are done almost in a full-duplex model, i.e., send/receive data size are the same and could be done simultaneously. However, the host-to-device and device-to-host communication are done in sequence, the first and last $R\delta$ layers (only convenient along Oz direction because Z changes the slowest) do not need to be copied to host since they are invalid data after R local iterations. Therefore, runtime time on pushing one simulation time-step ahead could be formulated as:

$$F_{step} = (F_c + F_m^{MPI} + F_m^{H2D/D2H})/R = (a_3R^3 + a_2R^2 + a_1R + a_0) + (b_m^M + b_m^G + 2a_m^GB^3)/R \tag{6}$$

Where, the computation time F_c depends upon the type of computing device, i.e., CPU or/and GPU. If both CPU and GPU are used, the task decomposition scheme should balance their task size, thus $F_c = F_c^{GPU} = F_c^{CPU}$. This scheme will be discussed in subsection 3.5. Here in Equation 6 we use $F_c = F_c^{GPU}$ as an example. Where in Equation 6, $a_0 = a_c^GB^3 + b_c^G + (6a_m^M + 10a_m^G)B^2\delta$, $a_1 = 6a_c^GB^2\delta + (12a_m^M + 16a_m^G)B\delta^2$, $a_2 = 12a_c^GB\delta^2 + 8(a_m^M + a_m^G)\delta^3$, $a_3 = 8a_c^G\delta^3$. Consider Equation 6 as a function of R, it has two parts: polynomial of degree 3 and reverse of R. Since a_1, a_2 and a_3 are very small number (-9 orders of magnitude) but greater than zero, the polynomial will increase with R ($R \in \mathbb{Z}^+$). However, the second part is a non-negative, monotonically decreasing function. Therefore, the second part and a_0 dominant F_{step} when R is small and, conversely, the first part becomes the dominant position after a finite integer. This indicates two traits. The first is that the runtime shall experience a significant decrease as R increases to some finite integer. The second is that there will also exist an R value at which the platform no longer favors computation over communication; in other words, there will be a fixed R for a given B and δ for which optimal performance is achieved. Later, in the performance study (section 5), we in fact observe the fall and rise of runtime with R, as predicted by the analytical model (actual time and analytical model prediction are compared in Figure 6). These inferences are in fact in line with the observation and empirical findings in stencil-based computations as well, although our model is more general in nature.

3.5. Heterogeneous balance

Multi-node is a common architecture in the current state-of-the-art parallel computing platform. On each of the nodes, there are multi-CPU with multi-core, multi-GPU and other parallel computing devices such as field-programmable gate array (FPGA), digital signal processor (DSP) and Intel Xeon Phi. As show in Figure 1, the compute capability of difference items (node, device, core) are different.

With hybrid acceleration there are different resources working on different tasks or parts of the work. When the work load is not balanced, some resources will be idling. Therefore, it is crucial to consider the hierarchical load balancing, an ideal task assignment should make all the computing devices in all hierarchies accomplish their job within the same time frame. In this study, we designed an automatic scheme to balance the jobs on CPU and GPU. The scheme will breakdown the job for GPU and CPU according to their compute capability. Their capabilities are estimated online via counting actually time consuming. Users only need to assign an initial ratio for the decomposition of CPU and GPU, then the scheme will tune itself to balance the load of CPU and GPU.

As shown in Equation 1, for static R and δ, the runtime is cubic to the subdomain size B. Considering that the 3D grid data was organized in an 1D array, in which z-direction varies slowest. It is fairly easier to split task along Oz direction, i.e., unit of task will be slice in Oxy plane. Thus the runtime will be linear to the number of slices. With the goal of assigning task for GPU and CPU to obtain similar runtime per simulation time-step iteration, the task assignment scheme can be describe as following equation group:

$$\begin{aligned} &\frac{T_C}{N_C}x_C - \frac{T_G}{N_G}x_G = 0 \\ &x_C + x_G = N \end{aligned} \tag{7}$$

Where, T_C and T_G is the CPU time and GPU time respectively, N_C and N_G are tasks (number of slices) assigned to CPU and GPU. Values for these four arguments are from previous iteration, x_C and x_G are variables represent the problem size of CPU and GPU respectively for the coming iteration. Our experiments (shown later in Figure 5a) show that the balance scheme could converge in less than five iterations.

4. Implementation

4.1. Inter-node synchronization

The subdomain boundaries synchronization includes two parts: inter-node updating via MPI, and host / device exchanging via *cudaMemcpy* function provided by CUDA. The inter-node updating process was illustrated in Table 1 and Figure 3. After inter-node boundaries updating and CPU/GPU task decomposition (detailed in subsection 3.5), OpenMP threads (equals with number of cores) are invoked for CPU computing and host operation of GPU computing. Specifically, one OpenMP thread will be used to perform CUDA related host operation, e.g., memory transfer, thread synchronization. Other OpenMP threads will perform the tasks assigned to CPU with data parallelism form. Given that there are $R\delta$ layers padded in each dimension of the subdomain, the GPU and CPU task could iterate R time-steps without synchronizing with neighboring subdomains, i.e., there are only $maxIter/R$ subdomain boundaries synchronization needed for $maxIter$ simulation time-steps. The overall flow was shown in Figure 4.

4.2. Benchmark application

With the goal to reduce communication latency at the cost of increased computation, three models (six applications) with different computation complexity and memory access intensities were used to benchmark the B2R latency

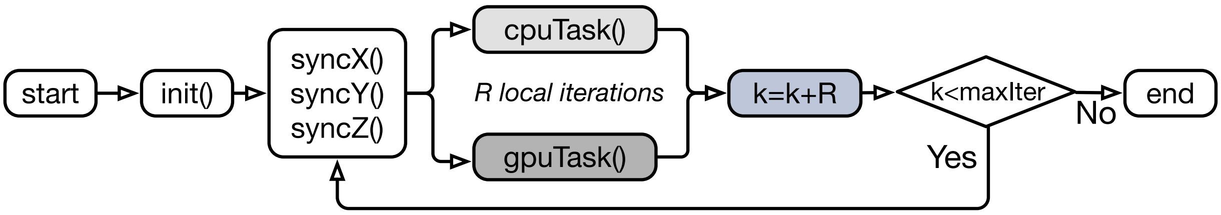


Figure 4: Flow chart of synchronization and computation. As illustrated in Table 1 and Figure 3, *syncX()*, *syncY()* and *syncZ()* are executed in sequential. There are multiple threads in *cpuTask()* for data parallelism, and one thread for *gpuTask()*.

hiding scheme. The first is the common used model for HPC benchmark, namely, heat diffusion model in 3D environment. This model has been implemented by using finite difference method with different order of accuracy (2nd, 4th, 6th, 8th -order accuracy separately). The second benchmark is a relatively well known agent-based model, namely, John Conways Game of Life [19] and, we extended it to 3D. The third is a Seismic wave propagation model. Details on models are discussed next.

1. *Heat diffusion*: It is a parabolic partial differential equation that describes the distribution of heat in a given region over time. More generally, it is to solve a partial differential equation:

$$\partial\mu/\partial t - \alpha\Delta^2\mu = 0 \tag{8}$$

Where, $\mu(x, y, z, t)$ is the current temperature of time variable t at point (x, y, z), α is the thermal diffusivity, Δ^2 denotes the Laplace operator. The Equation 8 was solved with finite difference method which essentially uses a weighted summation of function values at neighboring points to approximate the derivative at a particular point.

2. *3D Game of life*: A 3D extension of Game of Life (devised by John Horton Conway in 1970 in Ref. [19]). In the extended model, element with only 1 or less neighbors (Moore neighborhood, 26-cell cubic neighborhood total in 3D) die, as if by loneliness; If 5 elements surround an empty element, they breed and fill it, and if a cell has 8 or more neighbors, it dies from overcrowding. Therefore, the state of each element is computed as a function of 26 neighboring elements.

3. *Seismic wave propagation* in 3D Cartesian coordinates (x, y, z) for an isotropic and elastic medium can be formulated by equations in velocitystress form [12, 20]:

$$\partial_t v = \frac{1}{\rho}(\nabla \cdot \sigma + f) \tag{9}$$

$$\partial_t \sigma = \lambda\,(\nabla \cdot v)\, I + \mu\left(\nabla v + \nabla v^T\right) \tag{10}$$

Where, $f = \{f_x, f_y, f_z\}$ are the body force components in three directions (i.e., force from oceanic plates colliding), λ and μ are the Lamé coefficient, ρ is the constant density, $v = \{v_x, v_y, v_z\}$ denotes the particle velocity vector and σ represents the symmetric stress tensor. Expanding of Equation 9 and Equation 10 leads to three equations for velocity and six equations for the stress tensor components [20].

In the implementation of the three models, CUDA threads are mapped in Oxy plane and computation iterates along z direction (i.e., compute slice by slice along z direction in a CUDA thread block). For the head diffusion model, a central finite difference method was used for solving Equation 8 that describes the physical laws of heat transfer. It essentially uses a weighted summation of function values at neighboring points to approximate the derivative at a particular point. In 3D problem domain, with a (2δ)th-order accuracy, the next data value of each grid point is calculated based on the current data values of $\pm\delta$ in Ox, Oz, Oz directions and itself, i.e., $(6\delta + 1)$-*stencil computation*. We implemented with $\delta = \{1, 2, 3, 4\}$ in space, i.e., 7-, 13-, 19- and 25-stencil computation to benchmark our latency hiding scheme in multi-GPU and multi-CPU cores environment. In the CUDA based implementation, as described in Ref. [21], shared memory (structured by a 2D array with size: $(blockDim.x + 2\delta) * (blockDim.x + 2\delta)$) was used to buffer an entire slice, registers are used to buffer δ points in-front and δ behind in each thread. The current state values loaded into registers and shared memory will be shifted ahead iteration by iteration in Oz direction. This could dramatically save access to global memory.

Although δ equals 1 in 3D game of life model, its memory access intensity is quite different with the $\delta = 1$ (7-stencil) case of heat diffusion model. In the 3D game of life model, updating one cell requires state of 26 neighbors. Given this, three 2D array (with size: $(blockDim.x + 2) * (blockDim.x + 2)$) in shared memory were used to buffer current layer (in Oxy plane), one layer in-front, and one layer behind. Then iterate along Oz direction and shift the three array ahead. In summary, compare with head diffusion solution with $\delta = 1$, the game of life model requires less computational but more memory access.

In order to solve nine coupled partial differential equations of the seismic wave propagation model, a finite difference method with fourth-order accurate in space and second-order accurate in time was used for solving Equation 9 and Equation 10. This model is a memory-intensive application and twelve 3D variable arrays (three velocity components, three displacement components for visualization, and six symmetric stress tensor components) are involved in the main computation loop. We designed three CUDA kernel functions for computing v_x, v_y and v_z separately, and one for the six stress tensor components. The same as solving Equation 8, shared memory was used to reduce device memory accessing. Compare with the above two models, the seismic wave propagation model is both computational expensive and memory access intensive. That is to say, with the same subdomain synchronization, the seismic wave propagation model needs more time on computation (as later shown in Figure 7 and Figure 9a, the time ratio of $sync/computation$ is smaller).

5. Performance study

As detailed subsection 4.2, an agent-based model and two stencil computation models, both programmed in CUDA, were implemented to benchmark the overall performance of the B2R scheme. The head diffusion model was implemented with different order of accuracy, i.e., $delta = \{1, 2, 3, 4\}$. Since different δ results in quite different computational complexity (formulated in Equation 1 and Equation 2) and memory access intensities, there are six difference models in fact. This section will discuss the benefit of B2R latency hiding with different models and different scales.

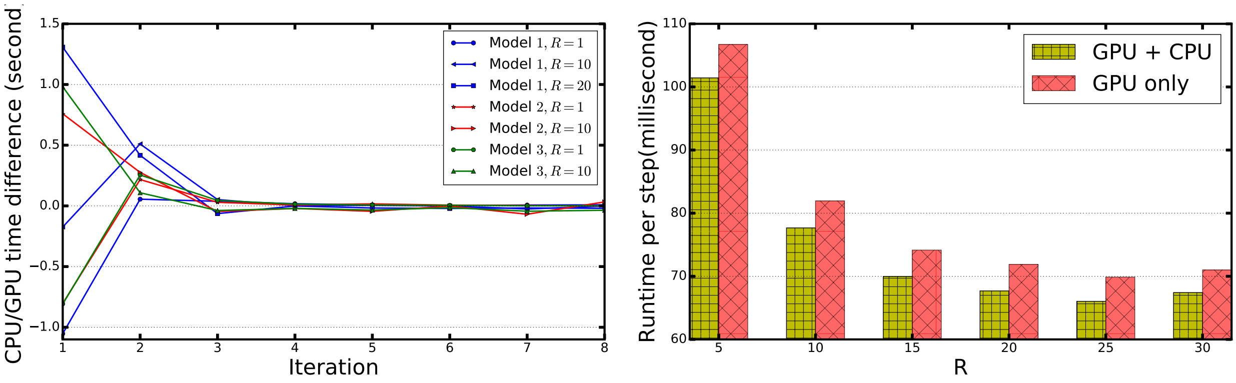
5.1. Hardware & Software

All the experiments have been run on Titan supercomputer at Oak Ridge National Laboratory. Each computing node is equipped with one AMD 16-core Opteron 6274 processor and one NVIDIA Tesla K20X GPU accelerator with 6 GB GDDR5 ECC memory. The CPU has access to 32 gigabytes of DDR3 memory, and is connected to a Cray Gemini interconnect through a hyper-transport 3 interface. Two computing nodes share one Cray Gemini high-speed interconnect router.

In the implementation, we use the NVIDIA Toolkit and SDK (nvcc compiler) for the GPGPU part. Concurrent execution on multi-core is achieved using OpenMP supported by PGI compiler. The inter-node communication for multi-CPU and multi-GPU is handled through MPI. Host part code, include MPI and OpenMP libraries, was compiled with PGI compiler. All runs were conducted on 64-bit Linux environment.

5.2. CPU/GPU load balance

The heterogeneous balance scheme described in subsection 3.5 has been tested with different models and different R. The total problem size is: $B_x = B_y = 200$, $B_z = 8000$ for all the models. As described in subsection 3.5, tasks for CPU and GPU are split in Oz direction. Figure 5a shows the convergence of time difference between CPU and GPU.



(a) Performance test of the automatic load balance scheme.

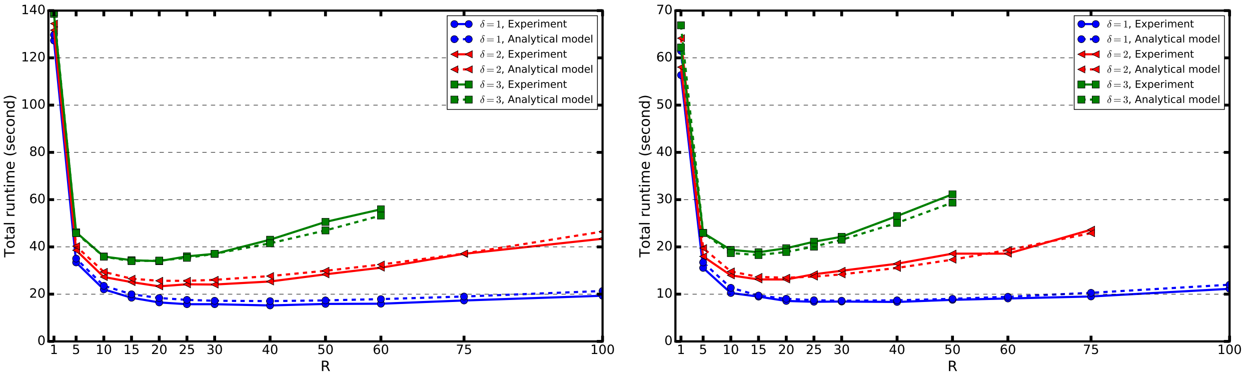
(b) Average runtime, GPU-only versus CPU+GPU.

Figure 5: Performance test of the automatic load balance scheme on different model. Different model represents different order of finite difference method for solving heat diffusion eqaution. Note: in Figure 5a the iteration was in node-level (Figure 1), i.e., one iteration denotes R simulation time-step. In Figure 5b, vertical Y-axis represents runtime for updating one simulation times-step.

As shown in Figure 8, it is clear that the balance could converge in less than 5 steps. The load balance scheme works in node-level (as shown in Figure 1), different models or R result in different runtime. Since the initial task ratio for CPU was set as 0.1, the runtime difference at the first step varies in different cases. As shown in Figure 5b, CPU+GPU could benefit a little. There is not that much improvement on runtime with CPU+GUP because the performance difference between CPU and GPU in our test computing node is too big (i.e., CPU undertakes few task). We believe that further benefit could be obtained on computing node with more powerful CPU.

5.3. Analytical model validation

To validate our analytical model formulated in Equation 1 – Equation 5, we carried out six experiments with heat diffusion model. The problem size are the same, namely, $1200 \times 1200 \times 1200$. The head diffusion model with $\delta = \{1, 2, 3\}$ were run with different number of nodes: (a) 27 nodes, 3 in each dimension, and (b) 64 nodes, 4 in each dimension. Thus subdomain size are different between (a) and (b). The results from experiment with $\delta = 2$ in (b) were used to fit the analytical model parameters described in Equation 6. Then with the fitted parameters, actual runtime and analytical model prediction were compared in Figure 6 on different experiment scenarios.



(a) Node = 27 ($3 \times 3 \times 3$).

(b) Node = 64 ($4 \times 4 \times 4$).

Figure 6: Total runtime, experiment results versus analytical model. The data in case with $\delta = 2$, Node = $4 \times 4 \times 4$ was used to fit parameter values. Note: problem size are $1200 \times 1200 \times 1200$, 600 iterations for each run.

It is clear to see from Figure 6 that, although our analytical model is quite general in nature, it is effective to describe the runtime complexity.

5.4. Speedup benefit

The runtime benefit or B2R is clear in Figure 6. This section details the distribution of the total runtime and, the speedup to $R = 1$ (conventional implementation without latency hiding scheme). The total runtime was roughly divided as three parts: inter-node communication for updating boundaries, memory transferring between host and device, and computation in GPU device. The game of life model runtime distribution under different R was illustrated in Figure 7.

As shown in Figure 7, dramatical communication time was reduced after small R. Although this results in a little longer computation time, the total runtime was reduced. However, there is an optimal R, after which the increased computing time dominates the communication time, i.e., can not profit indefinitely.

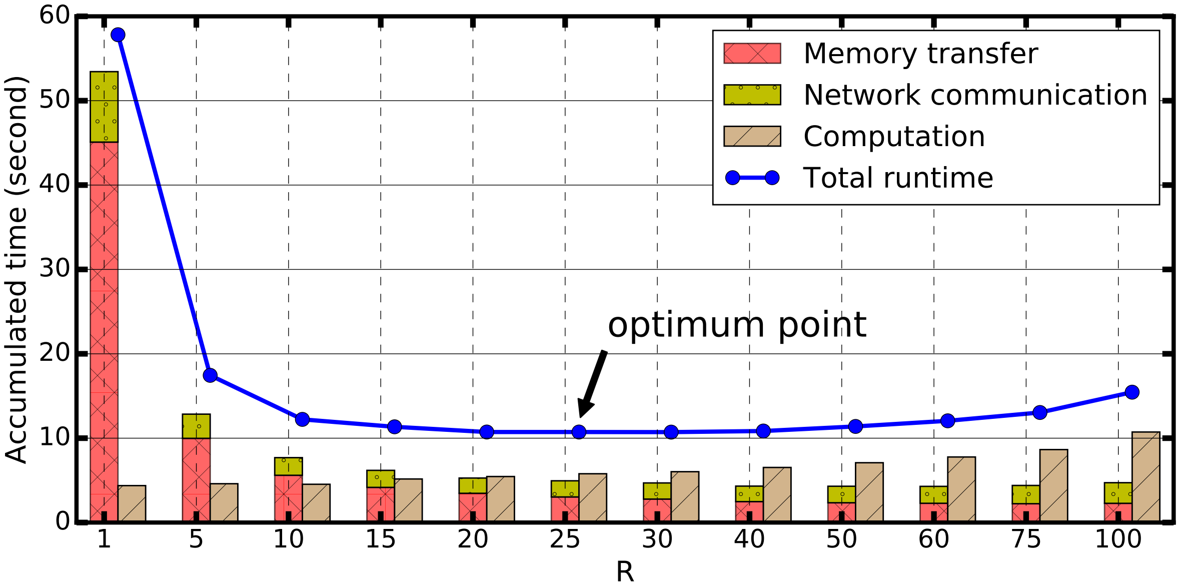
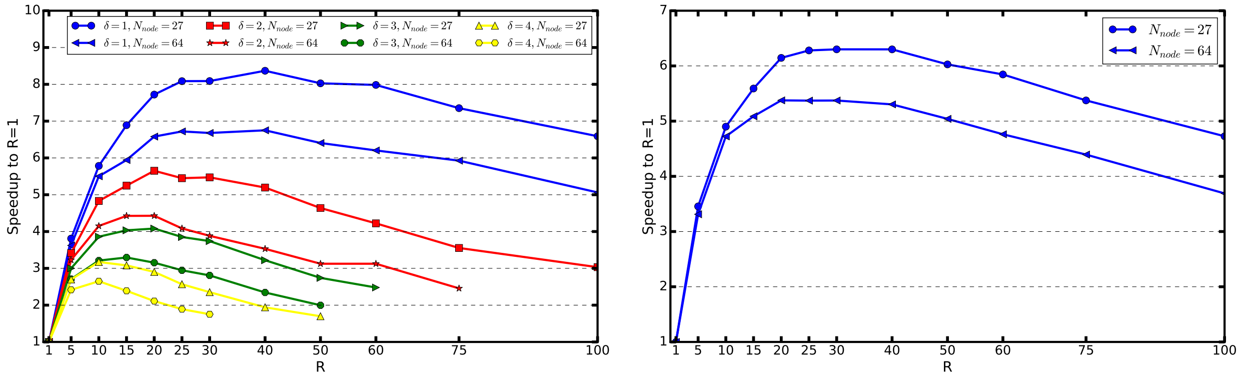


Figure 7: The proportion of communication time and computation time with different R.

To explore the benefit of B2R with different models, speedup to $R = 1$ (no latency hiding scheme) was used as metric. The speedup obtained with different R was shown in Figure 8. Where, the total problem size is $1200 \times 1200 \times 1200$, and iterated 600 times in all testing scenarios. Figure 8a shows the speedup benefit in solving heat diffusion model with different accuracy order of finite difference method. Since the condition: $R\delta \leq B$, $B = 400$ and 300 in 27 and 64 nodes scenario respectively, range of R is restricted in big δ scenarios. Moreover, Figure 8b shows the speedup benefit in solving the 3D game of life model.

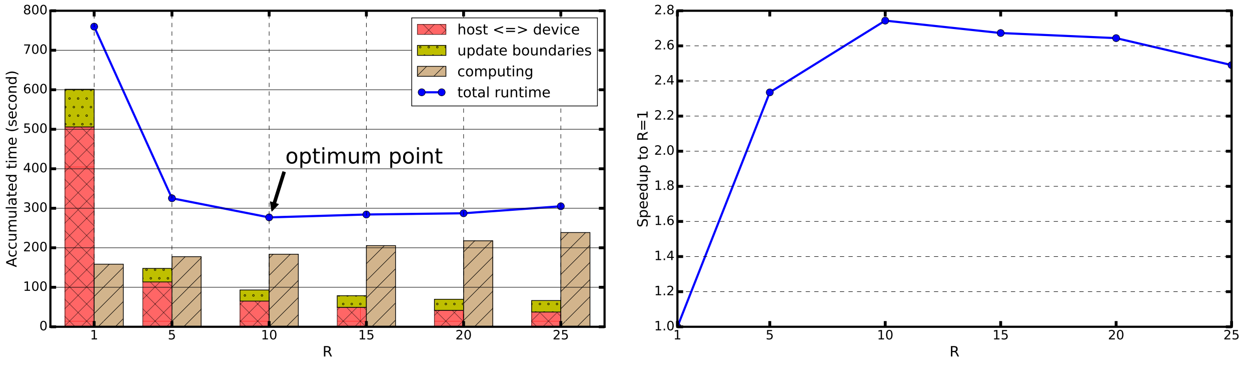


(a) Speedup versus R on solving heat diffusion model.

(b) Speedup versus R on 3D game of life model simulation.

Figure 8: Speedup versus R with different δ (Figure 8a) and subdomain size. Note: the global grid size of the two models are $1200 \times 1200 \times 1200$, and iterated for 600 times.

It is clear that smaller δ (results in less computation but less accurate) can benefit more from B2R latency hiding scheme, each scenario has a significant optimum points (i.e., the best tradeoff between computation and communication). The speedup in Figure 8 clearly highlight the effectiveness of the B2R scheme. For the same model, since the total problem size is the same, decompose to more nodes results in smaller subdomain size. Thus, compare the benefit between experiments with the same δ but different computing node. It is evident that bigger subdomain can benefit more speedup from the latency hiding because computing node hold larger portion of data in local memory. Compare Figure 8b with $\delta = 1$ in Figure 8a, although both scenarios require current state of neighbors within one layer, the game of life model benefit less. As analyzed in subsection 4.2, this is mostly because the game of life requires more memory access, which is more time-consuming than arithmetic operations [22]. That is to say the time ratio of *sync*/*computation* is smaller than heat diffusion model. Figure 9 illustrates the runtime proportion and speedup benefit on solving the seismic wave propagation model.



(a) Communication time versus computation time.

(b) Speedup versus R.

Figure 9: The proportion of communication time and computation time with different R and the speedup benefit versus R on solving seismic wave propagation model. The problem size is $3000 \times 200 \times 3000$ stencil points, it was decomposed to $25 = 5 \times 1 \times 5$ computing node.

Compare Figure 9a with Figure 7, it is clear that the communication to computation ratio of seismic wave propagation model is higher. Although B2R scheme is able to save a lot of synchronization time by performing a little more computation, the final speedup benefit is not as good as the other two models. However, about 2.8x speedup is still appealing on large-scale simulation.

In summary, the B2R scheme is able to save total runtime by synchronizing less frequently. The actually benefaction is affected by the communication to computation ratio, i.e., time ratio of *sync*/*computation*. The computationally expensive model will benefit less from B2R.

5.5. Large scale execution — strong & weak scaling

To test the scalability of B2R scheme, we conducted: (1) strong scaling test, which is defined as how the solution time varies with the number of processors for a fixed total problem size; (2) weak scaling test, which is defined as how the solution time varies with the number of processors for a fixed problem size per node. The 3D game of life model was utilized for the two tests. Figure 10 shows the results of strong scaling test.

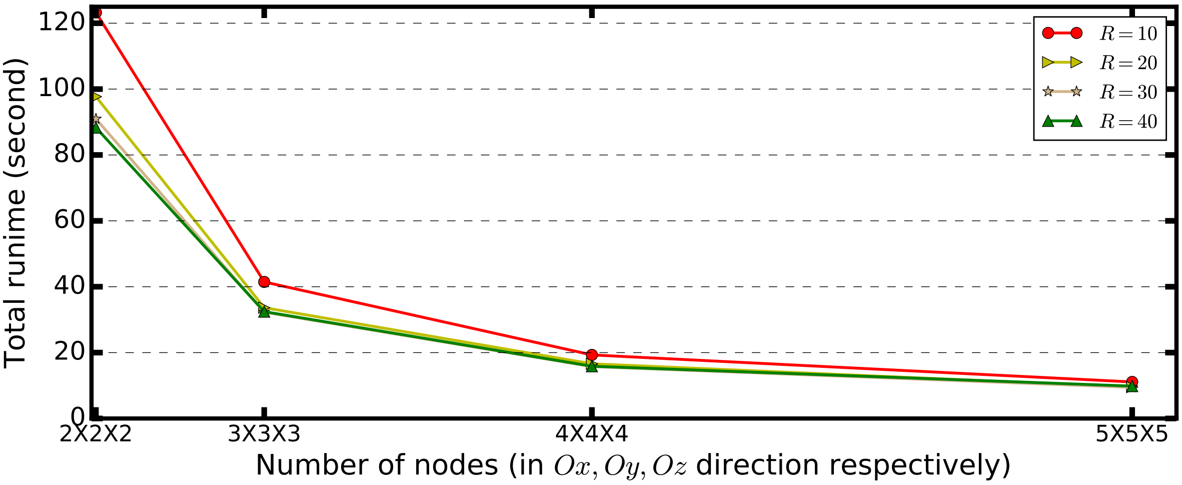
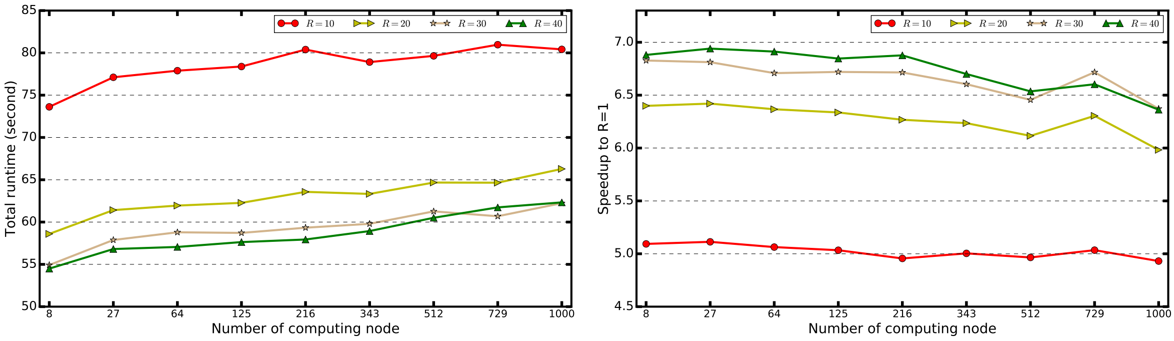


Figure 10: Strong scaling test of the B2R scheme. Total problem size: 1440 × 1440 × 1440.

The strong scaling test was carried out with total problem size of 1440 × 1440 × 1440, and with different R. As shown in Figure 10, it is clear that the total runtime decreased dramatically with increasing number of nodes, but not theoretical linear to the number of nodes. This is because, as discussed in analytical model and verified in Figure 8, the benefit of B2R also depends on subdomain size (B), less benefit could be obtained with smaller subdomain size. The weak scaling test results were shown in Figure 11.



(a) Total runtime versus problem size.

(b) Speedup benefit versus problem size.

Figure 11: Weak scaling study of the B2R scheme. The problem size per computing node is: 600 × 600 × 600.

The weak scaling was conducted with fixed problem size of 600 × 600 × 600 per node. So, more computing nodes could carry out bigger total problem size. The last case scales to 216 nodes that has 46.656 billion cells. As total runtime shown in Figure 11a, due to synchronization between nodes, the runtime increased slightly with bigger total problem size. In addition to the speedup benefit of B2R, as shown in Figure 11b the weak scaling almost does not affect the benefits of B2R scheme.

6. Summary and Future Work

We have investigated the latency hiding scheme on multi-CPU and multi-GPU cluster. With a few expectations, significant benefit could be obtained from the B2R scheme. Several times of speedup without adding any hardware resources is attractive for simulation users because it could dramatically reduce the large scientific simulations time (e.g., several days to one day).

ACKNOWLEDGMENTS

ack to OLCF

This research has been supported by the XXXX, and partially supported by the MINECO Spain, under contract TIN2014-53172-P, and a grant from the China Scholarship Council (CSC) under reference number: 201306290023. Performance tests were carried out using the Titan supercomputer at Oak Ridge National Laboratory.

References

[1] Z. Li, Y. Song, Automatic Tiling of Iterative Stencil Loops, ACM Trans. Program. Lang. Syst. 26 (6) (2004) 975–1028. doi:10.1145/1034774.1034777.

[2] J. Meng, K. Skadron, Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs, in: Proceedings of the 23rd International Conference on Supercomputing, ICS '09, ACM, New York, NY, USA, 2009, pp. 256–265. doi:10.1145/1542275.1542313.

[3] C. Lengauer, M. Bolten, R. D. Falgout, O. Schenk, Advanced Stencil-Code Engineering (Dagstuhl Seminar 15161), Dagstuhl Reports 5 (4) (2015) 56–75. doi:10.4230/DagRep.5.4.56.

[4] J. Zhou, Y. Cui, E. Poyraz, D. J. Choi, C. C. Guest, Multi-gpu implementation of a 3d finite difference time domain earthquake code on heterogeneous supercomputers, Procedia Computer Science 18 (2013) 1255 – 1264, 2013 International Conference on Computational Science. doi:10.1016/j.procs.2013.05.292.

[5] C. Ding, Y. He, A ghost cell expansion method for reducing communications in solving PDE problems, in: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, SC '01, ACM, New York, NY, USA, 2001, pp. 50–50. doi:10.1145/582034.582084.

[6] S. L. Lu, T. Karnik, G. Srinivasa, K. Y. Chao, D. Carmean, J. Held, Scaling the memory wall, in: Computer-Aided Design (ICCAD), 2012 IEEE/ACM International Conference on, 2012, pp. 271–272.

[7] A. Saulsbury, F. Pong, A. Nowatzyk, Missing the memory wall: The case for processor/memory integration, SIGARCH Comput. Archit. News 24 (2) (1996) 90–101. doi:10.1145/232974.232984.

[8] M. V. Wilkes, The memory wall and the cmos end-point, SIGARCH Comput. Archit. News 23 (4) (1995) 4–6. doi:10.1145/218864.218865.

[9] J. L. Hennessy, D. A. Patterson, Computer Architecture, Fifth Edition: A Quantitative Approach, 5th Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[10] B. G. Aaby, K. S. Perumalla, S. K. Seal, Efficient simulation of agent-based models on multi-gpu and multi-core clusters, in: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques, SIMUTools 10, 2010, pp. 29:1–29:10. doi:10.4108/ICST.SIMUTOOLS2010.8822.

[11] J. Zhou, D. Unat, D. J. Choi, C. C. Guest, Y. Cui, Hands-on performance tuning of 3d finite difference earthquake simulation on {GPU} fermi chipset, Vol. 9, 2012, pp. 976 – 985, proceedings of the International Conference on Computational Science, {ICCS} 2012. doi:10.1016/j.procs.2012.04.104.

[12] Y. Cui, K. B. Olsen, T. H. Jordan, K. Lee, J. Zhou, P. Small, D. Roten, G. Ely, D. K. Panda, A. Chourasia, J. Levesque, S. M. Day, P. Maechling, Scalable earthquake simulation on petascale supercomputers, in: 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC), 2010, pp. 1–20. doi:10.1109/SC.2010.45.

[13] Y. Hu, D. M. Koppelman, S. R. Brandt, F. Löffler, Model-driven auto-tuning of stencil computations on GPUs, in: A. Größlinger, H. Köstler (Eds.), Proceedings of the 2nd International Workshop on High-Performance Stencil Computations, Amsterdam, The Netherlands, 2015, pp. 1–8.

[14] T. Lutz, C. Fensch, M. Cole, Partans: An autotuning framework for stencil computation on multi-gpu systems, ACM Trans. Archit. Code Optim. 9 (4) (2013) 59:1–59:24. doi:10.1145/2400682.2400718.

[15] 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: High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for, 2008, pp. 1–12. doi:10.1109/SC.2008.5222004.

[16] Y. Zhang, F. Mueller, Auto-generation and auto-tuning of 3d stencil codes on gpu clusters, in: Proceedings of the Tenth International Symposium on Code Generation and Optimization, CGO '12, ACM, New York, NY, USA, 2012, pp. 155–164. doi:10.1145/2259016.2259037.

[17] C. D. Cantrell, Modern mathematical methods for physicists and engineers, Cambridge University Press, 2000.

[18] H.-Q. Ding, Simulating lattice QCD on a caltech/JPL hypercube, International Journal of High Performance Computing Applications 5 (2) (1991) 74–81. doi:10.1177/109434209100500205.

[19] M. Gardner, Mathematical games: The fantastic combinations of john conway's new solitaire game "life", Scientific American 223 (4) (1970) 120–123.

[20] Y. Qin, Y. Wang, H. Takenaka, X. Zhang, Seismic ground motion amplification in a 3D sedimentary basin: the effect of the vertical velocity gradient, Journal of Geophysics and Engineering 9 (6) (2012) 761. doi:10.1088/1742-2132/9/6/761.

[21] P. Micikevicius, 3D finite difference computation on gpus using cuda, in: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2, ACM, New York, NY, USA, 2009, pp. 79–84. doi:10.1145/1513895.1513905.

[22] C. Nvidia, Nvidia cuda c programming guide, Nvidia Corporation 1 (2015) 2–261.
URL http://docs.nvidia.com/cuda/cuda-c-programming-guide