Hindawi Publishing Corporation
Scientific Programming
Volume 2015, Article ID 648752, 13 pages
http://dx.doi.org/10.1155/2015/648752

Research Article

Methods to Load Balance a GCR Pressure Solver Using a Stencil Framework on Multi- and Many-Core Architectures

Milosz Ciznicki,[1,2] Michal Kulczewski,[1] Piotr Kopta,[1] and Krzysztof Kurowski[1]

[1] *Applications Department, Poznań Supercomputing and Networking Center, 61-139 Poznań, Poland*
[2] *Poznań University of Technology, 60-965 Poznań, Poland*

Correspondence should be addressed to Milosz Ciznicki; miloszc@man.poznan.pl

Received 16 July 2015; Accepted 19 October 2015

Academic Editor: Can Özturan

The recent advent of novel multi- and many-core architectures forces application programmers to deal with hardware-specific implementation details and to be familiar with software optimisation techniques to benefit from new high-performance computing machines. Extra care must be taken for communication-intensive algorithms, which may be a bottleneck for forthcoming era of exascale computing. This paper aims to present a high-level stencil framework implemented for the EULerian or LAGrangian model (EULAG) that efficiently utilises multi- and many-cores architectures. Only an efficient usage of both many-core processors (CPUs) and graphics processing units (GPUs) with the flexible data decomposition method can lead to the maximum performance that scales the communication-intensive Generalized Conjugate Residual (GCR) elliptic solver with preconditioner.

1. Introduction

The recent advent of novel multi- and many-core architectures, such as GPU and hybrid models, offer notable advantages over traditional supercomputers [1]. However, application programmers have to deal with hardware-specific implementation details and must be familiar with software optimisation techniques to benefit from new high-performance computing machines. It is therefore of great importance to develop expertise in methods and algorithms for porting and adapting the existing and prospective modelling software to these new, yet already established, machines.

Elliptic solvers of elastic models are usually based on standard iterative algorithms for solving linear systems, for example, CG, GMRES, or GCR. Numerous reports on porting them to modern architectures are available [2]. However, in an anelastic solver for geophysical flows, fast-acting physical processes may enter the elliptic problem implicitly. Furthermore, formulation of the boundary conditions is not trivial and therefore it is not feasible to use standard iterative solvers from linear algebra packages. For simulating physical experiments with a high degree of anisotropy, additional preconditioning is necessary to improve matrix conditioning. Such preconditioner for anisotropic geometries often relies on the direct inversion using the Thomas algorithm. A comprehensive study on implementations of tridiagonal solvers on GPU found that it is possible to implement solvers which perform exceptionally well in the range of grid nodes [3].

EULAG [4], an anelastic model for simulating low Mach number flows under gravity, developed in the National Center for Research, is widely used in an international community and has a rich portfolio of applications. It features nonoscillatory forward-in-time (NFT) numerics, which are original and unique. It also employs preconditioned, nonsymmetric, generalised conjugate-residual type "Krylov" scheme [5–8] to solve an elliptic boundary value problem, reported to be among the most effective methods for solving difficult elliptic problems [9]. Based on variational principles, Krylov solvers provide a hierarchical framework, which assures an asymptotic convergence rate in inverse proportion to the square root of the condition number of the linear operator, resulting from the numerical formulation of the model. The hierarchical design of Krylov solvers relies on the operator preconditioning, the goal of which is to accelerate the convergence of the main solver beyond the theoretically optimal limit.

Our work utilises the C++ framework to distribute the stencil computations on multiple CPUs and GPUs simultaneously. Previous systems distributed stencil computations with simple decomposition methods with uniform partition where each processor and accelerator receives subdomains of the same size. Unlike our work, previous approaches do not allow careful load balancing of the domain decomposition between heterogeneous architectures.

The framework is based on a domain-specific language (DSL) that expresses the stencil computations. It has a similar semantics to Fortran to ease the transition for the end users. It is specialized for partial differential equations defined on the regular multidimensional Cartesian grids. The usage of the standard C++ language allows avoiding the nonstandard language expression and nonstandard programming models and requires no external libraries. The flexible architecture of the framework is suited for the computations on different configurations of the clusters that contain diverse number of CPUs and GPUs within the computational node. The framework is able to efficiently spread computations on the CPU-only clusters including the NUMA machines, architectures with global shared memory, on the GPU-only clusters with fat nodes containing one GPU per single CPU core as well as on the hybrid clusters with powerful CPUs and GPUs. It provides the seamless subdomain decomposition to the blocks to efficiently utilise processors' memory hierarchy. Additionally, it allows the end user to manually tune the blocking size for the future processor architectures. The framework contains the communication library with the unified interface that allows for the efficient intranode and internode communication. The automatic parallelisation for multi-CPU, multi-GPU, and hybrid resources allows the users to write stencil functions that are translated to selected architectures. The usage of the C++ templates and provision of as much as possible of static information for the compiler improves the optimisation during compilation. With the new generations of compilers the code will scale for the future architectures.

Our research is to provide novel methods to adapt scientific code to novel hardware architectures, taking EULAG as an example. In this paper we focus on efficient communication methods with efficient load balancing to scale the elliptic solver along with the preconditioner.

2. Related Works

A number of previous works have been focused on accelerating stencil computations on GPUs [10–12]. The works presented in [10, 11] used autotuning techniques to efficiently parallelise stencils on multicore CPU and on many-core GPU. Other approaches employ multiple GPUs in solving stencil computations. They treat each GPU as an accelerator associated with separate MPI process where CPU acts as a management entity that does only the communication. Authors in [13] employed compiler based approach for automatic parallelisation of a code written in a domain-specific language into the Compute Unified Device Architecture (CUDA) programming standard. The work in [12] distributes stencil computations across multiple GPUs with explicit attention to the PCI express configuration.

Our work is more related to approaches that utilise multiple CPUs and GPUs simultaneously. Previous systems distributed stencil computations with simple decomposition methods with uniform partition where each processor and accelerator receives subdomains of the same size. For example, work in [14] utilises a high-level problem description to parallelise the code on the CPU and GPU clusters by combining OpenMP and CUDA. On the other hand, authors in [15] provide a framework that allows programmers to partition the data contiguously between CPU and GPU within single node. Unlike our work, their approach does not allow careful load balancing of the domain decomposition between heterogeneous architectures.

```
gcrk() {
   prforc()
   divrhs()
   precon()
   reduction()
   laplc()
   for it=1..solver_iterations {
        reduction()
        if(exit) quit_for_loops;
        precon()
        laplc()
   }
}
```

Algorithm 1: The body of the elliptic solver code.

3. Description of the GCR Solver

The body of the elliptic solver code consists of five major routines (Algorithm 1). The main routine advances the solution iteratively by calling other major computational routines. The routines *prforc* and *divrhs* initialise the solver. The former routine evaluates the first guess of the updated velocity, by combining the explicit part of the solution and the estimation of the generalised pressure gradient, while imposing an appropriate boundary condition. The latter routine evaluates the density weighted divergence of the velocity, and thus the initial residual error of the elliptic problem for pressure is computed. Among the most computationally intensive routine of the GCR solver is *laplc* that iteratively evaluates the generalised Laplacian operator (a combination of divergence and gradient) acting on residual errors. Another important part of the solver is the *precon* routine that accelerates the convergence of the variational scheme. By performing the direct matrix inversion in the vertical dimension of the grid; it is especially useful for large-scale simulations on thin spherical shells with grids characterised by a large anisotropy. The routine *precon* employs the sequential Thomas algorithm [16] to solve tridiagonal systems of equations with the right-hand side consisting of the horizontal divergence of the generalised horizontal gradient. This gradient is evaluated by *nablaCnablaxy*, which also belongs to the most computationally intensive routines of the GCR solver. With regard to the data access pattern, the computational loops within the elliptic solver can be simply divided into three categories: (i) reductions, (ii) implicit methods of the Thomas algorithm, and (iii) explicit methods of the stencils. In our work we focus on the explicit methods.

4. Multicore and Many-Core Architectures

This section describes two computing architectures utilised in our work: the cache-coherent NUMA machine with global shared memory and heterogeneous CPU-GPU supercomputer. Many of nowadays multi- and many-core architectures are NUMA, machines equipped with nonuniform memory access where memory access times depend on the memory location relative to the processor: one processor can access its local memory faster than the memory of another processor. Although NUMA is used in a symmetric multiprocessing (SMP) systems in particular, it is also used in multi- and many-core nodes, equipped with Intel Xeon and Itanium processors (that use QPI, QuickPath architecture) and AMD Opteron (using HyperTransport). To keep a consistent image, a cache-coherent protocol is used (ccNUMA), which simplifies the use of multiple cores, while exhibiting complex performance properties [17]. The recent advent of novel multi- and many-core architectures, such as GPU, offers notable advantages over traditional supercomputer. In our work we use the Piz-Daint supercomputer with heterogeneous CPU-GPU nodes. Each node contains the Intel Xeon E5-2670 CPU with 32 GB of RAM and single Nvidia Tesla K20X GPU. For the tests of large ccNUMA architecture we use the Chimera, the SGI Ultraviolet machine equipped with 256 Intel Xeon E7-8837 processors (a.k.a. NUMA nodes, 8 cores each), and 16 TB of RAM.

5. High-Level Stencil Framework

This section describes the design goals of the proposed framework. The major objectives during development are described as follows.

The GCR solver code was ported from the Fortran 77 language. The framework is based on a domain-specific language (DSL) that expresses the stencil computations. It has a similar semantics to Fortran to ease the transition for the end users. The usage of the standard C++ language allows avoiding the nonstandard language expression and nonstandard programming models and requires no external libraries. The flexible architecture of the framework is suited for the computations on different configurations of the clusters that contain diverse number of CPUs and GPUs within the computational node. The framework is able to efficiently spread computations on the CPU-only clusters including the NUMA machines, architectures with global shared memory, and on the GPU-only clusters with fat nodes containing one GPU per single CPU core as well as on the hybrid clusters with powerful CPUs and GPUs. It provides the seamless subdomain decomposition to the blocks to efficiently utilise processors' memory hierarchy. Additionally, it allows the end user to manually tune the blocking size for the future processor architectures. The framework contains the communication library with the unified interface that allows for the efficient intranode and internode communication. Depending on the decomposition of a computational domain between processors, on a position of the actual processor and on the periodicity of the boundaries, the library transparently chooses the most efficient communication method. The automatic parallelisation for multi-CPU, multi-GPU, and hybrid resources allows the users to write stencil functions that are translated to selected architectures. The usage of the C++ templates and provision of as much as possible of static information for the compiler improves the optimisation during compilation. With the new generations of compilers the code will scale for the future architectures.

```
Geometry *geom = Geometry::init<
  DomainSize<NP,MP,LP>,
  HaloSize<HLEFT,HRIGHT,HBOT,
    HTOP,HGND,HSKY>,
  ProcessorGridSize<NPX,NPY,NPZ>>();
Communicator *comm =
  Communicator::init(geom);
```

Algorithm 2

5.1. Programming Model. The following sections describe the implementation of the framework. Firstly, the structure of the framework is outlined with the initialisation of the necessary resources. Then, the methodology of the writing and running user stencils with an example is given. Lastly, the domain decomposition method with the communication model is illustrated.

5.2. Framework Structure. The idea behind the parallelisation of the computation between the processors is based on the data decomposition where each process updates the fixed part of the global domain called a subdomain. Since the stencil computations require the neighbour points to update a point, the boundaries of the subdomains have to be communicated between processors. The communicated boundaries are saved in a designated buffer called a halo region. In order to efficiently utilise the data locality the OpenMP and MPI models are employed for the intranode and internode communication. Each CPU and GPU have assigned separate MPI processes that are pinned to the selected cores. The GPU parallelisation is done using CUDA whereas the CPU parallelisation employs the OpenMP model.

5.3. Initialisation. During the initialisation of the framework user has to create the computational subdomain for each MPI process by using the `Geometry` and `Communicator` classes (See Algorithm 2).

The `Geometry` class initialises the 3D domain decomposition by using the `DomainSize`, `HaloSize`, and `ProcessorGridSize` classes. The `ProcessorsGridSize` class specifies the number of the subdomains in each

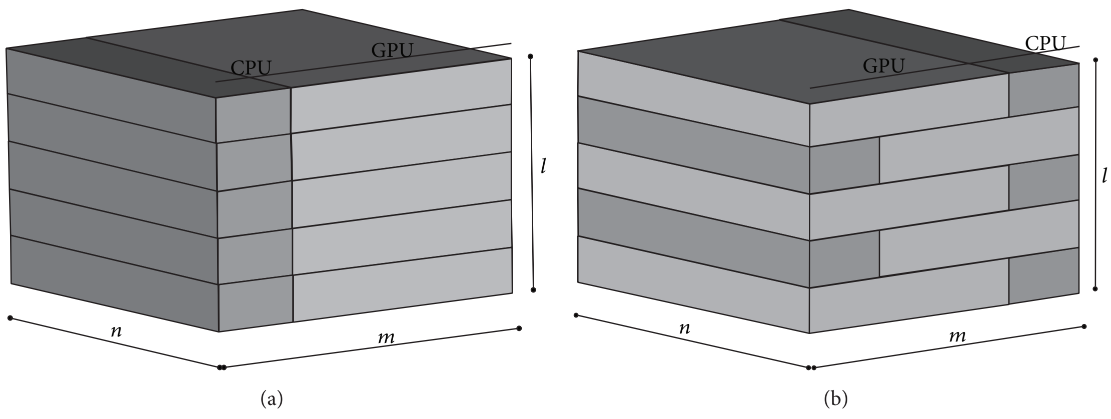


Figure 1: Domain decomposition: (a) allowed decomposition, (b) forbidden decomposition.

dimension. The decomposed subdomains may have different sizes with a restriction that each pair of the neighbouring processors sharing boundary in the single dimension have the same boundary size in that dimension; see Figure 1.

The three values `NP`, `MP`, and `LP` in `DomainSize` describe the size of the subdomain whereas the `HaloSize` class characterises the size of the halo region on each side. Furthermore, the `Geometry` class creates $n - 1$ OpenMP threads for CPU with n cores where the MPI processor of GPU is pinned to the nth core only if GPU is used. The `Communicator` class creates the specific communicator depending on the processor's architecture. To hide the communication time with the computation on GPU the communicator utilises the CUDA streams to concurrently exchange the boundary data during the computation of the subdomain. The communicator based on the position of the processor within the global domain handles communication in a specific way. The processors inside the domain always communicate the data while the processors on the boundaries communicate the data for the periodic boundaries only and do not communicate them for the nonperiodic boundaries. Additionally, for the decomposed domain with the single processor in a given dimension, the data is exchanged using only the processor's local memory.

5.4. Stencils. The task of computing the stencils can be essentially divided into two parts. First, the stencil with an access pattern updating the domain point has to be defined. Second, the range within the computational domain on which stencil will be executed has to be provided. To enable this in the framework, the user defines the stencil functions and executes them through a kernel.

5.4.1. Writing Stencils. In the framework the stencils are defined as the C++ functors called the stencil functions. The 3D Laplacian function is defined as shown in Algorithm 3.

The `DEFINE_DO` macro allows quickly defining the functor. The stencil access pattern on the 3D domain is described by using the `IN3D` macro. There also exist `IN1D` and `IN2D` macros that allow operating on the 1D and 2D domains. The functor through template parameters passes information

```
struct LaplcStencil {
 DEFINE_DO(const T *__restrict__ in_p,
    T *__restrict__ out_p,const
    T &cCoeff){
  IN3D(out_p, 0, 0, 0) = cCoeff * (
   IN3D(in_p, -1, 0, 0, CACHED) +
   IN3D(in_p, 1, 0, 0, CACHED) +
   IN3D(in_p, 0, -1, 0, CACHED) +
   IN3D(in_p, 0, 1, 0, CACHED) +
   IN3D(in_p, 0, 0, -1, CACHED) +
   IN3D(in_p, 0, 0, 1, CACHED));
 }
};
```

Algorithm 3

about the domain dimensions and index parameters i, j, k to the macros. The first parameter of the `IN3D` macro takes a pointer to an array; the parameters from the second to the fourth take the positions of the domain point related to currently updated point. For example, `IN3D(p, −1, 2, 0)` returns indices of $(i - 1, j + 2, k)$. The last parameter of the `IN3D` macro is optional and gives a hint to the framework that the following point should be cached in the shared memory of GPU to improve efficiency. The decision which points of which array should be cached is based on how many points of a given array are accessed. Typically, the array with the largest number of accessed points should have its values cached to reduce the number of main memory accesses. However, if only single point is accessed per array, the `CACHED` macro should not be used as it would degrade performance. Due to the small size of the shared memory, the points of the single array can be cached at a time. The function parameters of stencil functions must begin with the pointer to the array that is cached. In case of 3D Laplacian example the pointer to the `in_p` is first. The details of the algorithm that does stencil computations can be found in our previous work [18] (See Algorithm 4).

The comparison of the C++ stencil function to the sequential Fortran 77 code shows that the framework's code

```
      do 3 k=2,lp-1
      do 3 j=2,mp-1
      do 3 i=2,np-1
    3 out_p(i,j,k)=cCoeff*(
     · in_p(i+1,j,k)+in_p(i-1,j,k)+
     · in_p(i,j+1,k)+in_p(i,j-1,k)+
     · in_p(i,j,k+1)+in_p(i,j,k-1))
```

Algorithm 4

Table 1: Number of SLOC (source lines of code) for the Fortran 77 MPI code and the C++ MPI + OpenMP + CUDA code.

Language	SLOC
F77 MPI	11238
C++ MPI + OpenMP + CUDA	10066

is compact and enforces regular neighbour access pattern. Table 1 compares the source lines of code for the original MPI Fortran code with the code developed with the framework. The source code developed with the framework has similar size comparing to the manually written MPI Fortran version. Although it handles three programming models comparing the F77 code, still Fortran is powerful for writing the scientific code. However, the source code lines are not an ideal metric as the higher source code size does not necessarily mean lower productivity; still it is easy to understand metric to compare the program complexity.

5.4.2. Running Stencils. In order to apply the stencil function, the framework provides the `kernel_conf` function that is used to invoke the 3D Laplacian on the 3D computation domain as shown in Algorithm 5.

The `kernel_conf` function is initialised with the type of the floating-point calculations `type_t` such as float or double. Similarly to the `Geometry` class the user provides `DomainSize` and `HaloSize`. The `StencilSize` class describes the number of accessed neighbouring points on each side of the stencil function. In our example the 3D Laplacian provides `StencilSize<1,1,1,1,1,1>`. To restrict the computation region of the stencil function the `ComputeRegion` class with the range parameters is used. The stencil function will be applied from the `SLEFT` index to the index `NP-1-SRIGHT` where NP is size of the subdomain in the i direction. The `updateInner/updateLeft/...` enum allows parallelising the update of the boundary points with the inner points of the subdomain by the utilisation of the GPU streams. In our 3D Laplacian stencil example the inner points use `updateInner` whereas the boundaries updates are modelled with `updateLeft/updateRight...` so that the separate kernel calls are utilised for each side. The `Cache` class drives the usage of the cache. The `TRUE` and `FALSE` macros switch on and off the caching of the neighbouring points of the stencil function, respectively. The cache size is automatically determined by the framework. However, if needed, the optional parameter `CSIZE` allows manually controlling the cache size in bytes. The order of the

```
kernel_conf<
  type_t, DomainSize<NP,MP,LP>,
  StencilSize<SLEFT,SRIGHT,SBOT,
    STOP,SGND,SSKY>,
  HaloSize<HLEFT,HRIGHT,HBOT,
    HTOP,HGND,HSKY>,
  ComputeRegion<0+SLFET,NP-1-SRIGHT,0+SBOT,
    MP-1-STOP,0+SGND,LP-1-SSKY>,
  LaplcStencil,
  updateInner,
  Cache<TRUE,CSIZE>
>(in_p, out_p, cCoeff);
```

Algorithm 5

parameters of `kernel_conf` must be the same as in the stencil function.

The `kernel_conf` function executes the stencil functions using OpenMP for CPU while for GPU the CUDA kernel functions are called.

5.5. Domain Decomposition. There are a number of the different decomposition strategy options available. Typically systems use MPI-all parallelisation scheme with a uniform partition where each individual core maps to the MPI process with no utilisation of the shared memory on a compute node. This scheme is very simple to implement and straightforward to run as it requires no knowledge about the NUMA topology of the physical node. On the other hand, the number of MPI messages required to exchange is a multiple of the number of cores; thus the communication overhead is substantial. Another choice is a strategy which assigns single MPI process to the whole node. It decomposes the obtained subblock for a specified number of processors and accelerators and minimizes the number of MPI processes, thus the communication overhead; see the methodology described in [19]. The drawback of this method is that the inner part of the subblock is only decomposed in one dimension; hence it is not flexible in balancing the load between the accelerators and processors. Additional strategy performs a uniform decomposition where pair of CPU and GPU is mapped to single MPI process. In this case the boundaries of the subdomain are updated and communicated by CPU whereas the inner points are handled by GPU. In this scenario CPU serves as a management entity and does not execute any computations thus it inefficiently uses CPUs. Our framework utilises single MPI process per each processor with a flexible and efficient decomposition strategy to make best use of the hybrid architectures. The scheme can partition the domain in all three dimensions to nonuniform subblocks for the arbitrary number of the processors and accelerators. This scheme enables computing on different cluster configurations that contain diverse number of CPUs and GPUs within the computational node. The framework is able to efficiently decompose the domain on the CPU-only clusters including the NUMA machines, architectures with global shared memory, and on the GPU-only clusters with fat nodes containing one GPU per single CPU core as well as on the hybrid clusters with powerful CPUs and GPUs. The partition mechanism is employed once before the compilation of the code for the target architecture thus the obtained decomposition is static during computations. This static decomposition allows the compiler to optimise the code for the stencil loops by utilisation of various techniques such as loop unrolling and vectorisation. Once the subblock for each processor is obtained, it is further decomposed to the optimal size for the cache blocking thus receiving the optimal size for the processor. The details of the subblock decomposition are described in our previous work [18].

5.6. Load Balancing. Careful load balancing is essential in order to find the good partition of the computational domain between the heterogeneous resources. The goal is to minimize the difference of the finish time of computing the portion of the partition between CPU and GPU. To reach this goal the two-level decomposition method is employed. First of all, the domain is partitioned to equally sized subdomains between the machine nodes. The partition of the domain may be one-, two-, or three-dimensional. If for the given partition the domain dimension size is not evenly divisible by the number of the machine nodes, the last node obtains resized subdomain to fit the domain size. Please note that it is assumed that each node contains the same resources. Next, the obtained subdomain size is utilized to benchmark CPU and GPU to measure the computational time and calculate the size of the subblocks assigned to each processor within the node. For example, the size of the subblock obtained from the one-dimensional decomposition is calculated as follows:

$$\begin{aligned} f_{\text{cpu}} &= \frac{t_{\text{gpu}}}{t_{\text{cpu}} + t_{\text{gpu}}}, \\ n_{\text{cpu}} &= n_n, \\ m_{\text{cpu}} &= m_n * f_{\text{cpu}}, \\ l_{\text{cpu}} &= l_n. \end{aligned} \tag{1}$$

The t_{cpu} and t_{gpu} values equal to the computational times necessary to update the subdomain are assigned to the cluster node on CPU and GPU, respectively. The f_{cpu} value determines the chunk of the subdomain assigned to the processor. The subdomain size is specified by the n_n, m_n, and l_n values. The final block size is described by the n_{cpu}, m_{cpu}, and l_{cpu} values.

5.7. Communication and Computation Overlap. The framework utilises the MPI communication to exchange messages between processors. The halo data transfer between the accelerators is conducted through CPU. CPU acts as a bridge that receives the data from the GPU and then packs it to the MPI message and sends it to the host CPU of the target accelerator. The host CPU unpacks data and transfers it through PCI express to the target GPU. Please notice that the framework currently does not support GPUDirect RDMA to directly exchange data between GPUs located on different nodes without using CPUs as we did not have access

```
Communicator *comm =
  Communicator::init(geom);
comm->update(p_in,sizeLeft,shiftLeft,
  sizeRight,shiftRight,updateLeft);
// or
comm->update_beg(p_in,sizeLeft,shiftLeft,
  sizeRight,shiftRight,updateLeft);
// kernel execution
comm->update_end(p_in,sizeLeft,shiftLeft,
  sizeRight,shiftRight,updateLeft);
```

Algorithm 6

```
// left processor call
comm->update(p_in,1,-1,1,0,updateLeft);
// right processor call
comm->update(p_in,1,0,1,-1,updateLeft);
```

Algorithm 7

to the machine that supports it. The framework provides the aforementioned `Communicator` class to transfer data between processors and accelerators. The example usage of the class is showed in Algorithm 6.

The class provides two methods of sending MPI messages: synchronous and asynchronous. The update method is used to apply synchronous communication whereas for the asynchronous communication type the pair of methods `update_beg` and `update_end` are exploited. The kernel execution is surrounded by asynchronous MPI calls to overlap communication with the computation. The `Communication` class is able to flexibly exchange boundaries on each side of the domain. The `update` method requires six parameters where `p_in` is pointer to the 3D array containing data. The following pair of parameters defines the size and the shift of the boundary data received from the left processor whereas the next pair determines the size and the shift of the boundary data sent to the right processor. The last parameter of the `update` method specifies the side of the update in this case the left update. For example, the domain is decomposed to the two processors in the *i* direction. There are two defined stencil functions on the domain. The computation range of the first stencil called the boundary stencil is constrained to left boundary of the domain while the second stencil called the inner stencil updates the inner points of the domain. The boundary stencil models the boundary condition and requires single point from the left shifted by one position; it is the `i-2` index. The inner stencil demands single point from the left with the `i-1` index. To fulfil the stencils requirements for the left and right processors the `update` method is as shown in Algorithm 7.

From the perspective of the sender the left processor sends its right boundary to the right processor whereas the right processor sends its right boundary to the left processor; see Figure 2. This communication method simplifies the

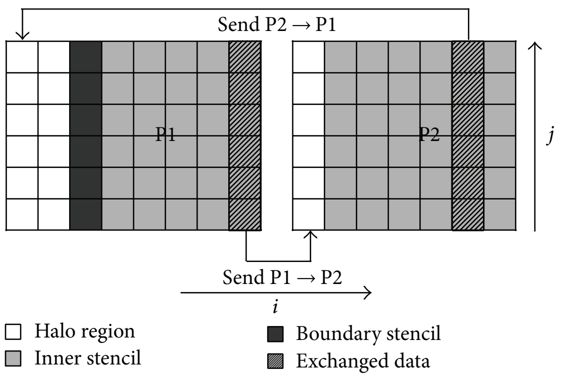


Figure 2: Example of the communication between two processors.

exchange of the boundaries for the sophisticated stencils that require the values from the distant neighbours. To efficiently scale the code on large number of processors and accelerators the framework utilises the overlapping of the communication with the computation. The idea of the overlapping is based on the separation of the computation of the boundary regions and the inner region. The separate kernel calls are employed for each side and the inside of the subdomain. The communication of the boundaries with the computation of the inside of the subdomain is overlapped and what is more the boundary kernels are computed in parallel to more efficiently utilise the accelerator resources. Figure 3 shows the flow of the overlapping method based on GPU of the 3D Laplacian stencil. To concurrently update the boundaries with the inside of the subdomain the seven streams are utilised. In case of more sophisticated stencils up to 27 streams are used. The kernel with the zero-copy memory is used to copy boundary data from GPU to the host CPU. The kernel with the zero-copy memory allows us on the fly to change the 3D layout of the boundary data to linear ordered without using an intermediate buffer. The linear ordered data is directly passed to the MPI send function. The order of copying the boundaries is as follows: first, the left and right boundaries are copied, next the bottom and top boundaries, and finally the ground and sky boundaries. The order is specified by the time needed to send the data through PCI express. The kernel updating inner of the subdomain is executed concurrently with the copy kernels. After the first boundary is copied to CPU it is sent through MPI to the proper processor, depicted as Communication in the figure. As soon as the halo region is received, it is sent back to GPU. Finally, the stencil updating the boundary is executed. This overlapping methodology allows us to concurrently execute five events: copying from and to GPU, computing the inner subdomain, MPI communication, and computing the boundaries. Depending on the decomposition of the computational domain between processors, on the position of the actual processor, and on the periodicity of the boundaries the library transparently chooses the most efficient communication method. The processors inside the domain always communicate the data while the processors on the boundaries communicate the data for the periodic boundaries only and do not communicate them for the nonperiodic boundaries. Additionally, for the decomposition with the single processor in a given dimension, the data is exchanged using only the processor's local memory.

5.8. MPI Scheduler for NUMA. With the novel multi- and many-core architectures it is of great importance to place properly all processes and threads on the underlying hardware. In [20] we proposed a method for mapping application topology to cluster. We used the MPI ping-pong benchmark to measure the sustained latency and bandwidth between all nodes to calculate the cost matrix. Next, the minimum path is calculated and, eventually, the application topology is mapped to the hardware, using the Hilbert curve to calculate the spatial locality [21]. We have extended this functionality especially to NUMA architectures, taking into account proper placement of MPI and OpenMP threads across system. The developer can rely on Intel or GNU facilities for proper thread placement and KMP and GOMP environments variables, respectively. It allows users to give a hint to the system about on which cores OpenMP threads should be placed, as well as not moving threads between cores during the run. To pin MPI processes accordingly, one has to rely on Intel or OpenMPI facilities, which are not always available. To automate processes and threads placement on underlying hardware, we use the HWLOC [22], a portable hardware locality software package that provides a portable abstraction of the hierarchical topology of modern architectures. Using aforementioned HWLOC, we calculate distances between each pair of NUMA nodes (which may be whole node in traditional cluster or socket or a processor in more SMP-like environment). Next, we find the minimum path as previously, and using the Hilbert curve we place each MPI process and its OpenMP threads on cores, where for most cases one MPI process per NUMA node is the most efficient allocation.

5.9. Integration with Fortran. Together the GCR solver with the MPDATA method are a dynamical core of the EULAG software. As described above, EULAG is developed in Fortran thus the stencil framework has to provide a flexible way of the integration of different programming models. It is of high importance to flexibly define the memory layout of the data to avoid the cost of the transposition. In case of the GCR solver the memory layout is the same as in the Fortran code i, j, k where the i index is linearly ordered in the memory for both the CPU and GPU code. To avoid the cost of the data movement between CPU and GPU all arrays are allocated on GPU before executing the time loop and persist during the entire run. The framework provides the Fortran bindings to ease the combination with C++, OpenMP, and CUDA. The arrays can be easily shared between Fortran and the framework. The arrays may be allocated in two different ways: (a) with a pointer that is allocated in Fortran and is reused in the framework; (b) with a copy where the framework allocates the aligned copy of the array and copy it when it is used in the Fortran code.

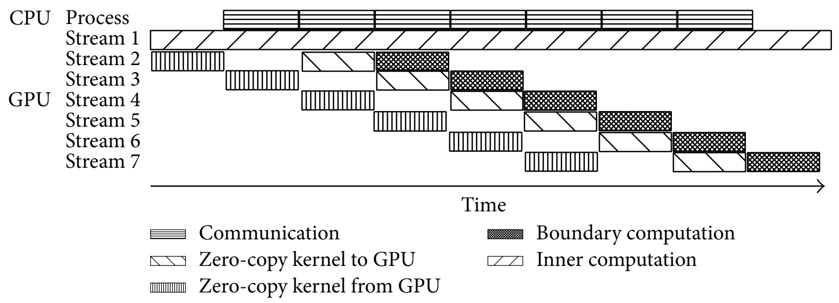


Figure 3: Flow of the overlapping method based on GPU.

6. Experimental Evaluation

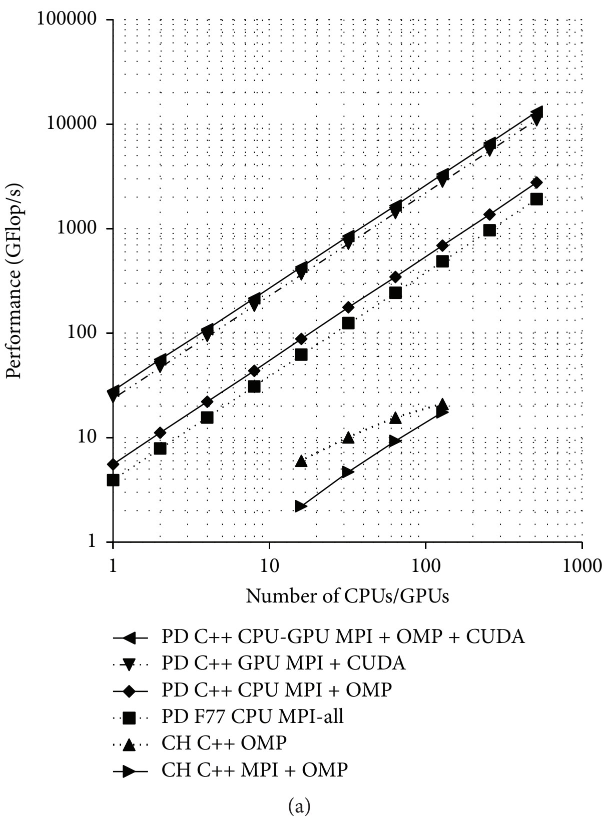
In this section, the strong and weak scaling results are presented using the GCR solver on the Piz-Daint supercomputer and on the Chimera SMP machine. The Piz-Daint supercomputer contains 5272 nodes equipped with an Intel Xeon E5-2670 CPU with 32 GB of RAM and Nvidia Tesla K20X GPU. The Chimera machine has 2048 CPU cores with Intel Xeon E7-8837 CPUs clocked at 2.66 GHz with 16 TB of RAM.

The implementation of the GCR solver is validated using a standard benchmark test case for incompressible flow solvers. We simulate decaying turbulence of a homogeneous incompressible fluid. Here, only the simplified setup proposed by Taylor and Green [23] is considered. The details of the problem can be found in our previous work [18].

6.1. Weak Scaling. The GCR solver is tested using five different versions of the code. The Fortran CPU MPI-all version is the original code developed in Fortran 77. The remaining variants including C++ CPU MPI+OpenMP, C++ OpenMP, C++ GPU MPI+CUDA, and C++ CPU-GPU MPI+OpenMP+CUDA are implemented with our framework. In order to evaluate the performance of all codes the number of floating-point operations is counted by calculating their occurrence in the source code. The MPI ranks in the MPI-all code are pinned to individual cores whereas for the MPI+OpenMP version single MPI rank is used for each CPU. The work is distributed across cores by using the OpenMP threads. In case of the GPU code single MPI rank is used. For the heterogeneous CPU-GPU case two MPI ranks are pinned to single CPU. The first MPI rank executes seven OpenMP threads where the second MPI rank is pinned to the last CPU core and handles the execution of GPU. For ccNUMA architecture, OpenMP threads are distributed across all the allocated cores. For the MPI+OpenMP version, MPI ranks are placed on separate sockets (a.k.a. NUMA nodes), with the corresponding set of OpenMP threads. Figure 4 shows the weak scaling results. All the codes almost reach the perfect linear scaling up to 512 nodes. Using 8 CPUs on the 488^3 domain size the C++ MPI+OpenMP version is 1.4x faster than the Fortran MPI-all code. Moving the code to GPUs for the same domain leads to 6x speedup comparing to the original Fortran code. The heterogeneous CPU-GPU code further improves speedup to 7x by distributing subdomains to CPUs. Figure 5 presents the performance per watt for the weak scaling case. The greatest number of GFlop/s per watt is obtained for the GPU-only code. In case of using 8 nodes the GPU code is 2.13x more power efficient than the CPU code. The hybrid CPU-GPU code is 1.88x more power efficient than the CPU code. In summary, in our case it is more power efficient to run code only on GPUs instead of using the hybrid CPU-GPU code.

6.2. Strong Scaling. Figure 4 shows the strong scaling results for a fixed grid size 244^3 with the varying number of processors and accelerators for two-dimensional domain decomposition along the y and z directions. The results are presented for the best domain decompositions for all code variants. The CPU codes scale up to more than 100 nodes; however the run with GPU saturates at 128 GPUs count. In order to efficiently use the GPU resources it requires a minimal number of domain points to saturate the memory bandwidth. The similar saturation can be observed with the CPU-GPU code. Figure 5 demonstrates the performance per watt for the strong scaling with the same grid size. Up to the 64 computational nodes the GPU code is the most power efficient whereas the CPU code reaches high power efficiency with the number of nodes equal to or larger than 128. For higher number of the nodes the power efficiency of the GPU code and the hybrid CPU-GPU code is converging. For both the weak scaling and the strong scaling case it is more power efficient to use the GPU code than the hybrid CPU-GPU code.

6.3. Load Balancing. As described in Section 5 the two-level decomposition is employed to balance the load between the computational resources. Figure 6 shows the performance of the first-level decomposition of the domain to the subdomains distributed between the cluster nodes for a fixed grid size 614^3. The decomposition strategies include the 2D partition through j and k dimensions and the 3D partition through i, j, and k dimensions. The 3D partition provides up to 1.16x better performance for the larger number of nodes. The second-level decomposition of the subdomain between the CPU and GPU processors is presented in Figure 6. For the subdomain size larger than 10^4 grid cells on average the 80% of the subdomain is assigned to GPU. When the subdomain



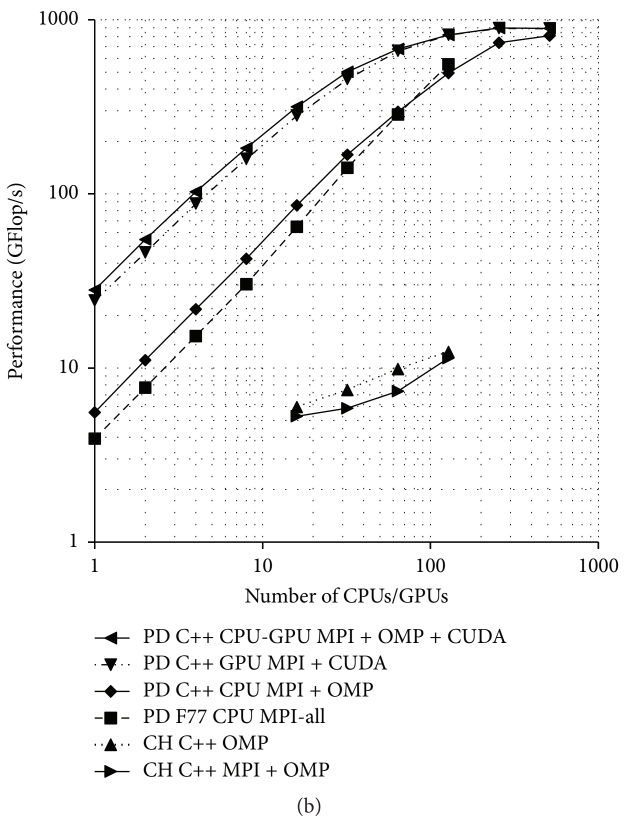
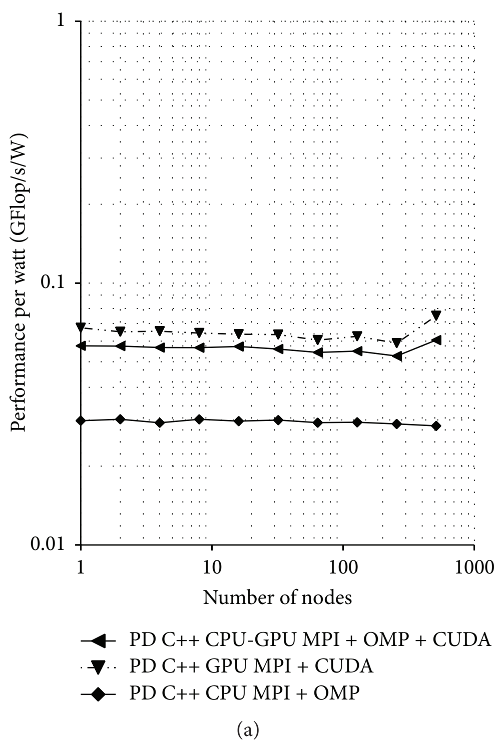


FIGURE 4: Performance. (a) Weak scaling for the 244^3 domain size per CPU/GPU; (b) strong scaling for the total 244^3 domain size. Used machines: PD: Piz-Daint cluster; CH: Chimera SMP.



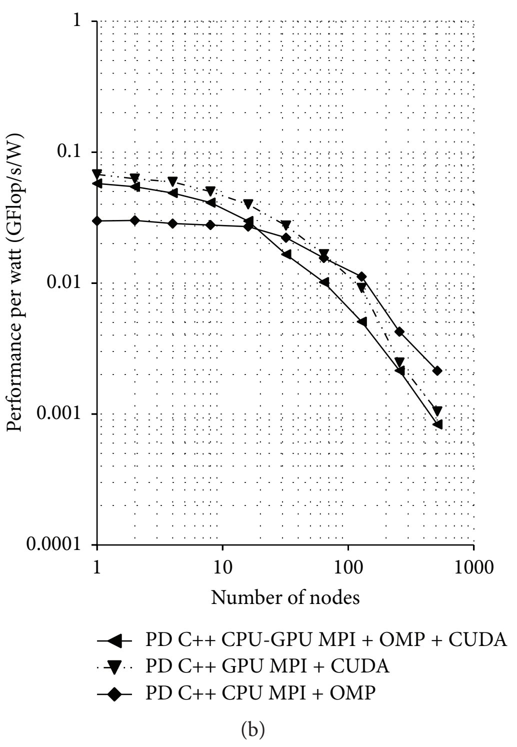


FIGURE 5: Performance per watt. (a) Weak scaling for the 244^3 domain size; (b): strong scaling for the total 244^3 domain size on the Piz-Daint cluster.

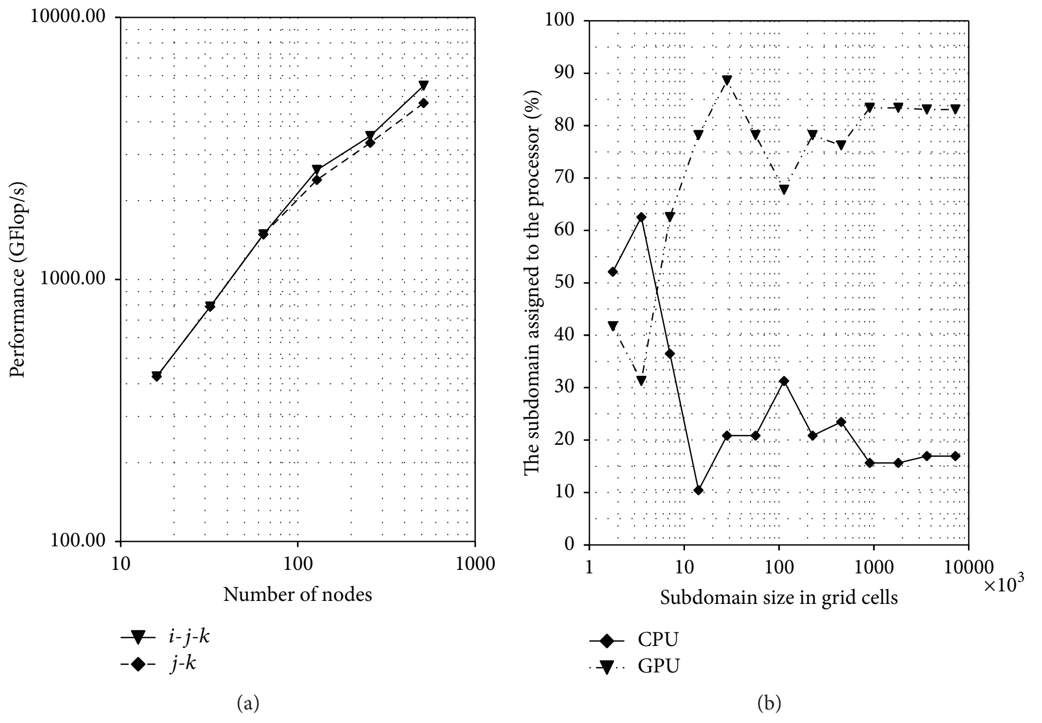


Figure 6: (a) First-level decomposition for the strong scaling for the 614^3 domain size. (b) Second-level decomposition for the subdomains obtained from the first-level decomposition on the Piz-Daint cluster.

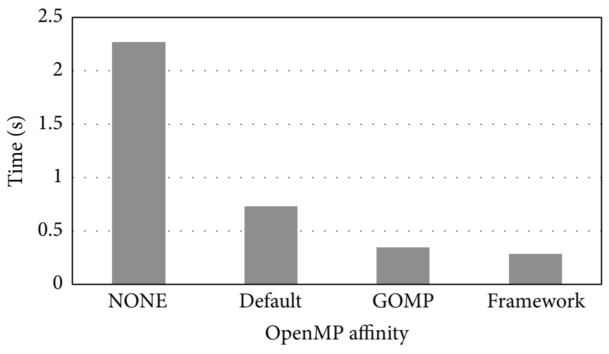


Figure 7: GCR C++ OpenMP version: different OpenMP affinity versus execution time.

size is smaller than 10^4 grid cells the size of the subdomain assigned to CPU grows. However, for such a small subdomain size, it is generally not efficient to employ GPUs.

6.4. MPI Scheduler Impact. The scheduler was tested on ccNUMA machine equipped with 256 nodes (8 cores each) and global shared memory. MPI and OpenMP allocation impact is presented using different version of OpenMP-only and MPI+OpenMP GCR solver implementation: with and without aforementioned framework. In the MPI+OpenMP version, MPI ranks are placed on separate sockets (NUMA nodes), with the corresponding set of OpenMP threads. For OpenMP-only version, threads are distributed across all the allocated cores with different scheduling policy. Figure 7 presents how different OpenMP affinity policies impact execution time of the C++ OpenMP-only implementation of GCR. The $128 \times 128 \times 128$ problem is solved by 128 cores. The weakest result is for NONE affinity, where no thread affinity is set, while threads are allowed to migrate between cores during the execution. The default policy set in the system typically means that threads can be run even on the single core, while possible migration is up to the system. The GOMP policy, which improves execution time greatly, means that the GOMP-CPU-AFFINITY environment variable was used to affect thread affinity (set to GATHER). The best result however is achieved for framework version of thread affinity, which places subsequent threads on sockets based on distances between them (minimal path is selected). Various

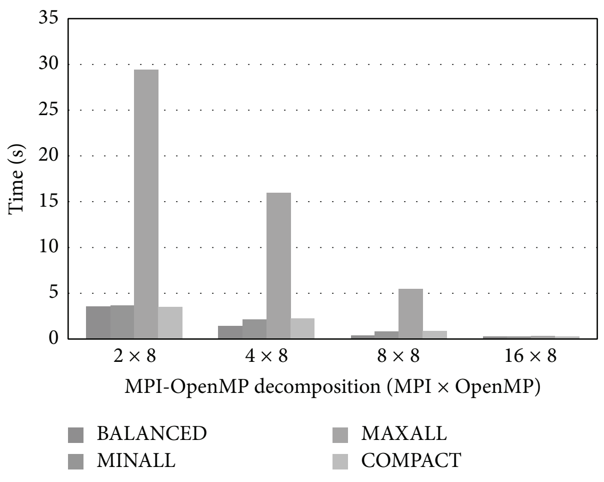


Figure 8: MPI+OpenMP decomposition, Fortran version.

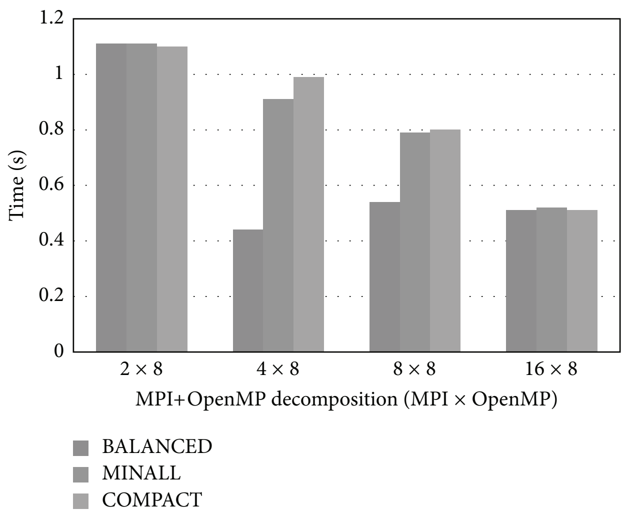


Figure 9: MPI+OpenMP decomposition, framework version.

types of MPI processes scheduling were tested: (i) MINALL: the minimum path between all NUMA nodes, (ii) MAX-ALL: the maximum path between all NUMA nodes, (iii) COMPACT: the minimum path between allocated NUMA nodes (side by side in performed tests), and (iv) BALANCED: placement balanced over all available NUMA nodes (one processor per board except for the 128 cores being used). The results are presented in Figure 8 for Fortran version and Figure 9 for C++ version. When all NUMA nodes are used (128 cores), the execution time for all MPI/OpenMP placement types is the same, except for the MAXALL which uses the longest path between MPI ranks. Where less MPI ranks are present, attention to and careful study of Figure 10 are needed. It presents Chimera's hardware topology of one board equipped with 2 processors (8 cores each). There were 8 such boards available for testing purposes. For example, when using 32 cores (i.e., 4 MPI ranks and 8 OpenMP threads each), BALANCED placement is using 0, 4, 8, and 12 processors (each MPI rank is run in its own distant board) and MINALL utilises 0, 1, 4, and 5 processors (there are 2 MPI ranks running on the same boards; the boards are distant to each other), while COMPACT uses 0, 1, 2, and 3 processors; (i.e., there are 2 boards near each other; each is running 2 MPI ranks). For the hardware architecture like Chimera, BALANCED placement seems to be the best, even if it does not form a minimal path between MPI ranks, while COMPACT should be used every time all allocated cores are used for calculations.

7. Conclusions

In this work, the stencil framework is presented that utilises a domain-specific language to simplify the development of stencil computations on multi- and many-cores architectures. The framework is written with C++ templates that provide portable code with no need for the additional dependencies. The C++ templates with the static domain decomposition allow the compiler to efficiently optimise the prepared code. The flexible domain decomposition scheme with the sub-domain partition to fit the memory hierarchy of the target architecture supports load balancing the work between an arbitrary number of CPUs and GPUs. The resulting code with the communication overlap method achieves high scalability and a 7x speedup against the Fortran MPI-all code. However, the GPU code is 1.14 more power efficient than the hybrid CPU-GPU code. The framework can be used with a good outcome on NUMA machines, including those equipped with global shared memory, and on the heterogeneous CPU-GPU supercomputers.

The results from the evaluation tests showed that the heterogeneous cluster configurations promise relatively large energy savings. For future work, we want to develop the scheduling methods to dynamically allocate stencil tasks to various unit blocks to optimize the energy efficiency. We want to take into account the communication between processors to better predict the runtime and the energy usage of stencil computations.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work is supported by the Polish National Center of Science under Grant no. UMO-2011/03/B/ST6/03500. This research was supported in part by PL-Grid Infrastructure. This work was supported by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID d25.

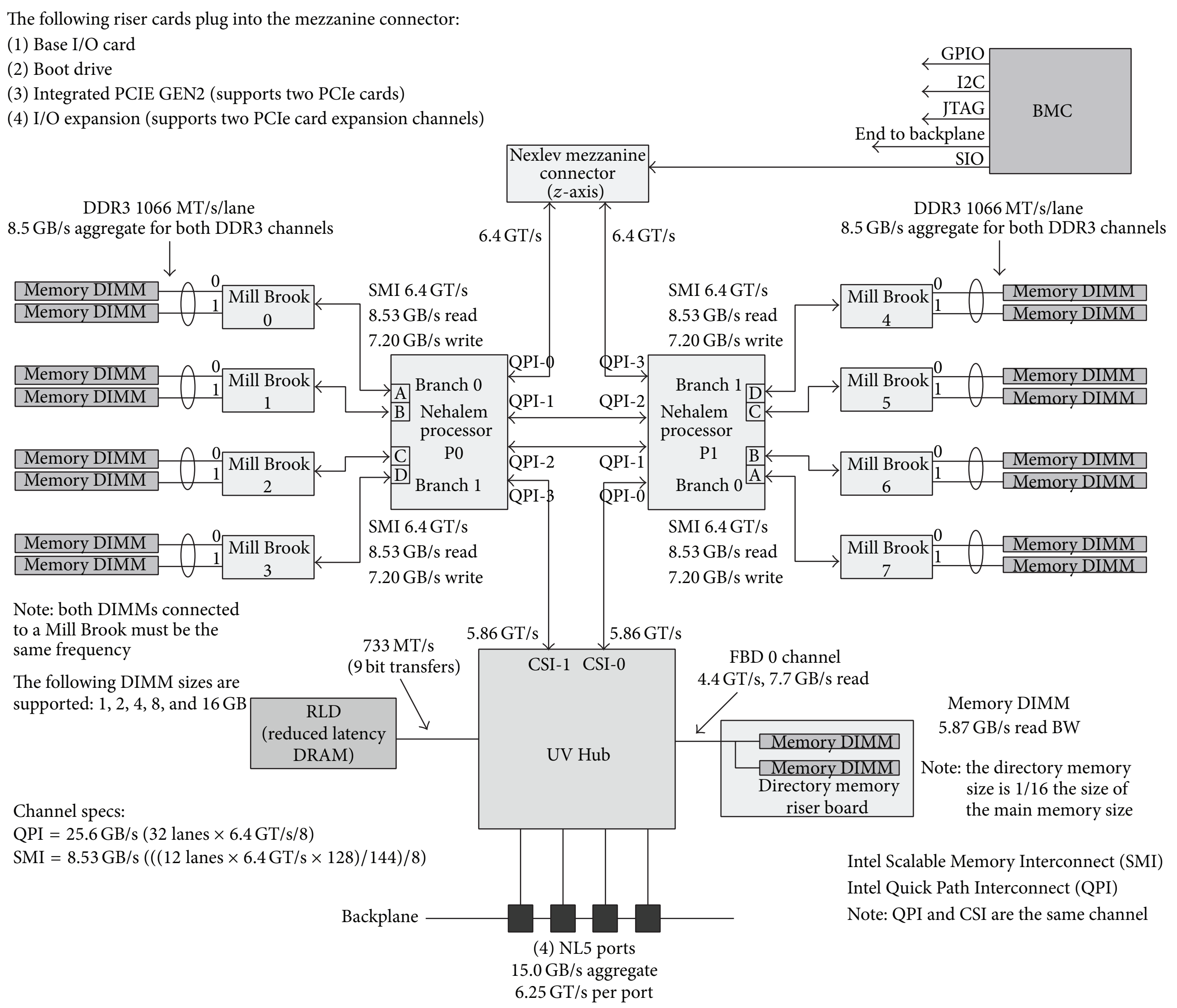

The following riser cards plug into the mezzanine connector:
(1) Base I/O card
(2) Boot drive
(3) Integrated PCIE GEN2 (supports two PCIe cards)
(4) I/O expansion (supports two PCIe card expansion channels)
GPIO
I2C
JTAG
End to backplane
SIO
BMC
Nexlev mezzanine connector (z-axis)
DDR3 1066 MT/s/lane
8.5 GB/s aggregate for both DDR3 channels
6.4 GT/s
6.4 GT/s
Memory DIMM
Mill Brook 0
Mill Brook 1
Mill Brook 2
Mill Brook 3
Mill Brook 4
Mill Brook 5
Mill Brook 6
Mill Brook 7
SMI 6.4 GT/s
8.53 GB/s read
7.20 GB/s write
QPI-0
QPI-1
QPI-2
QPI-3
Branch 0
Branch 1
Nehalem processor P0
Nehalem processor P1
A B C D
Note: both DIMMs connected to a Mill Brook must be the same frequency
The following DIMM sizes are supported: 1, 2, 4, 8, and 16 GB
733 MT/s (9 bit transfers)
5.86 GT/s
CSI-1 CSI-0
UV Hub
RLD (reduced latency DRAM)
FBD 0 channel
4.4 GT/s, 7.7 GB/s read
Memory DIMM
5.87 GB/s read BW
Directory memory riser board
Note: the directory memory size is 1/16 the size of the main memory size
Channel specs:
QPI = 25.6 GB/s (32 lanes × 6.4 GT/s/8)
SMI = 8.53 GB/s (((12 lanes × 6.4 GT/s × 128)/144)/8)
Intel Scalable Memory Interconnect (SMI)
Intel Quick Path Interconnect (QPI)
Note: QPI and CSI are the same channel
Backplane
(4) NL5 ports
15.0 GB/s aggregate
6.25 GT/s per port

Figure 10: SGI UV board (Chimera).

References

[1] J. Kurzak, D. Bader, and J. Dongarra, *Scientific Computing with Multicore and Accelerators*, CRC Computer and Information Science Series, Chapman & Hall, 2010.

[2] S. Georgescu and H. Okuda, "Conjugate gradients on multiple GPUs," *International Journal for Numerical Methods in Fluids*, vol. 64, no. 10-12, pp. 1254–1273, 2010.

[3] Y. Zhang, J. M. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the GPU," *ACM SIGPLAN Notices—PPoPP '10*, vol. 45, no. 5, pp. 127–136, 2010.

[4] J. M. Prusa, P. K. Smolarkiewicz, and A. A. Wyszogrodzki, "EULAG, a computational model for multiscale flows," *Computers & Fluids*, vol. 37, no. 9, pp. 1193–1207, 2008.

[5] P. K. Smolarkiewicz and L. G. Margolin, "Variational elliptic solver for atmospheric applications," *Applied Mathematics and Computer Science*, vol. 4, pp. 527–551, 1994.

[6] P. K. Smolarkiewicz, V. Grubisić, and L. G. Margolin, "On forward-in-time differencing for fluids: stopping criteria for iterative solutions of anelastic pressure equations," *Monthly Weather Review*, vol. 125, no. 4, pp. 647–654, 1997.

[7] W. C. Skamarock, P. K. Smolarkiewicz, and J. B. Klemp, "Preconditioned conjugate-residual solvers for Helmholtz equations in nonhydrostatic models," *Monthly Weather Review*, vol. 125, no. 4, pp. 587–599, 1997.

[8] P. K. Smolarkiewicz and L. G. Margolin, "Variational methods for elliptic problems in uid models," in *Proceedings of the ECMWF Workshop on Developments in Numerical Methods for Very High Resolution Global Models*, vol. 7, pp. 137–159, Reading, UK, June 2000.

[9] S. J. Thomas, J. P. Hacker, P. K. Smolarkiewicz, and R. B. Stull, "Spectral preconditioners for nonhydrostatic atmospheric models," *Monthly Weather Review*, vol. 131, no. 10, pp. 2464–2478, 2003.

[10] S. Kamil, C. Chan, L. Oliker, J. Shall, and S. Williams, "An auto-tuning framework for parallel multicore stencil computations," in *Proceedings of the 24th IEEE International Parallel and*

Distributed Processing Symposium (IPDPS '10), pp. 1–12, IEEE, Atlanta, Ga, USA, April 2010.

[11] M. Christen, O. Schenk, and H. Burkhart, "PATUS: a code generation and autotuning framework for parallel iterative stencil computations on modern microarchitectures," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS '11)*, pp. 676–687, IEEE, Anchorage, Alaska, USA, May 2011.

[12] T. Lutz, C. Fensch, and M. Cole, "PARTANS: an autotuning framework for stencil computation on multi-GPU systems," *Transactions on Architecture and Code Optimization*, vol. 9, no. 4, article 59, 2013.

[13] N. Maruyama, T. Nomura, K. Sato, and S. Matsuoka, "Physis: an implicitly parallel programming model for stencil computations on large-scale gpu-accelerated supercomputers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11)*, pp. 1–12, IEEE, Seatle, Wash, USA, November 2011.

[14] M. Blazewicz, I. Hinder, D. M. Koppelman et al., "From physics model to results: an optimizing framework for cross-architecture code generation," *Scientific Programming*, vol. 21, no. 1-2, pp. 1–16, 2013.

[15] A. D. Pereira, L. Ramos, and L. F. Góes, "PSkel: a stencil programming framework for CPU-GPU systems," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 4938–4953, 2015.

[16] J. C. Strikwerda, *Finite Difference Schemes and Partial Differential Equations*, SIAM, 2004.

[17] S. Ramos and H. Torsten, "Cache line aware optimizations for ccNUMA systems," in *Proceedings of the 2th International ACM Symposium on High-Performance Parallel and Distributed Computing*, Portland, Ore, USA, June 2015.

[18] K. A. Rojek, M. Ciznicki, B. Rosa et al., "Adaptation of fluid model eulag to graphics processing unit architecture," *Concurrency and Computation: Practice & Experience*, vol. 27, no. 4, pp. 937–957, 2015.

[19] W. Xue, C. Yang, H. Fu et al., "Enabling and scaling a global shallow-water atmospheric model on Tianhe-2," in *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS '14)*, pp. 745–754, IEEE, Phoenix, Ariz, USA, May 2014.

[20] M. Ciznicki, P. Kopta, M. Kulczewski, K. Kurowski, and P. Gepner, "Elliptic solver performance evaluation on modern hardware architectures," in *Parallel Processing and Applied Mathematics: 10th International Conference, PPAM 2013, Warsaw, Poland, September 8–11, 2013, Revised Selected Papers, Part I*, vol. 8384 of *Lecture Notes in Computer Science*, pp. 155–165, Springer, Berlin, Germany, 2014.

[21] S.-I. Kamata, R. O. Eason, and Y. Bandou, "A new algorithm for N-dimensional Hilbert scanning," *IEEE Transactions on Image Processing*, vol. 8, no. 7, pp. 964–973, 1999.

[22] F. Broquedis, J. Clet-Ortega, S. Moreaud et al., "Hwloc: a generic framework for managing hardware affinities in HPC applications," in *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP '10)*, IEEE Computer Society Press, Pisa, Italia, February 2010.

[23] G. I. Taylor and A. E. Green, "Mechanism of the production of small eddies from large ones," *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 158, no. 895, pp. 499–521, 1937.