Cluster-SkePU: A Multi-Backend Skeleton Programming Library for GPU Clusters

Mudassar Majeed, Usman Dastgeer, and Christoph Kessler
Department of Computer and Information Sciences (IDA), Linköping University, Sweden
{mudassar.majeed, usman.dastgeer, christoph.kessler}@liu.se

Abstract— *SkePU is a C++ template library with a simple and unified interface for expressing data parallel computations in terms of generic components, called skeletons, on multi-GPU systems using CUDA and OpenCL. The* smart containers *in SkePU, such as Matrix and Vector, perform data management with a lazy memory copying mechanism that reduces redundant data communication. SkePU provides programmability, portability and even performance portability, but up to now application written using SkePU could only run on a single multi-GPU node. We present the extension of SkePU for GPU clusters without the need to modify the SkePU application source code. With our prototype implementation, we performed two experiments. The first experiment demonstrates the scalability with regular algorithms for N-body simulation and electric field calculation over multiple GPU nodes. The results for the second experiment show the benefit of lazy memory copying in terms of speedup gained for one level of Strassen's algorithm and another synthetic matrix sum application.*

Keywords: Structured parallel programming, Skeleton Programming, GPU Cluster, SkePU, Scalability, Scientific Applications

1. Introduction

Many supercomputers in the Top500 list contain Graphics Processing Units (GPUs) for accelerating data parallel computations in large-scale parallel applications. For example, Titan, a Cray XK7 system installed at Oak Ridge, contains 560,640 processors, plus 261,632 NVIDIA K20x accelerator cores [1].

These recent developments in multi- and many-core based, multi-GPU systems and GPU clusters and increasing demands for performance in scientific computing have pushed the change in programming paradigms. In order to exploit the processing power of the aforementioned architectures, legacy serial codes for scientific applications have to be rewritten using parallel programming paradigms. For instance, many scientific applications with computationally intensive data parallel computations have been ported to CUDA and OpenCL for performance reasons. However, CUDA and OpenCL are at relatively low level of abstraction, requiring the explicit offloading of the computationally intensive tasks by transfering their operand data to/from the accelerators, invoking kernels etc. GPU performance tuning requires that programmers are experts in the specific languages and architectural features. Several other programming models such as distributed memory and bulk synchronous parallelism exist that require the knowledge of task and data partitioning, orchestration, communication and synchronization from the application programmer. Furthermore, debugging, maintaining and porting a parallel application to other (and future) architectures requires code modification, leading scientific application programmers to focus more on development details instead of domain specific issues.

We have taken initiative towards a structured approach for writing massively parallel applications with multiple backends. The aim is to develop a rich skeleton library for heterogeneous multi-node architectures with a number of accelerators like GPUs for writing structured and non-trivial large-scale massively parallel applications. Writing large-scale parallel applications in different domains may require different kinds of distributed sparse or regular data structures, like graphs, trees, vectors, matrices, meshes etc [2] and high level computation and communication patterns or *algorithmic skeletons*. In this prototype, we provide the regular data structures with smartness of data management (we refer to such data structures as *smart containers*). Similarly we provide simple high-level *algorithmic skeletons* for expressing relatively regular algorithms in terms of the provided skeletons. In on-going work, we are extending the design and implementation of SkePU so that it can be used for (certain kinds of) irregular applications as well.

Those applications may also require several optimizations at different points, like the communication of data at different levels of granularity, data caching, prefetching and data locality etc. So the library has to be equipped with certain flexibilities for making better (online or offline) choices, like the data partitioning granularity, communication and computation patterns and other important parameters. Initially, for that purpose, we have implemented several real scientific applications with the provided simple *algorithmic skeletons*.

In earlier work [3] we started with the structured parallel programming approach using skeletons as a solution for the portability, programmability and even performance portability problems in GPU-based systems. Skeleton programming frameworks provide generic constructs, so-called *skeletons*, that are based on higher order functions parameterizable in problem-specific sequential code, that express frequently

occurring patterns of control and data dependence, and for which efficient (also parallel and platform specific) expert-provided implementations may exist [4], [5], [6]. The application programmer expresses the scientific computation in terms of the given skeletons. The programming interface remains sequential, all parallelism, data transfer, synchronization and other platform specific details are encapsulated in the skeleton implementations. A number of skeleton programming systems have been developed in the last 20 years, in particular in the form of libraries such as Muesli [7], a C++ skeleton library for clusters, SkelCL [8], a skeleton library for GPU systems, BlockLib [9], a C skeleton library for IBM Cell/B.E., and the C++ based SkePU [3] for multi-GPU systems. Most of these skeleton libraries are specific to a particular backend like BlockLib is for the IBM Cell/B.E. and work for simple kernels.

Muesli [7] was initially designed for MPI/OpenMP clusters and has recently evolved to CUDA and GPU computing. On the contrary, SkePU was initially designed for single-node GPU-based system (OpenMP/CUDA/OpenCL) and is evolving towards MPI based clusters. This difference in approach results in several key programming differences. SkePU supports OpenCL which makes it much more applicable to other GPU and accelerator (FPGA etc.) platforms not supporting CUDA. Although Muesli supports task parallel skeletons for MPI/OpenMP, only a data parallel skeleton (*Map*) for CUDA with few communication variations is supported which limits program portability. SkePU supports a wide range of data parallel skeletons (*Map*, *Reduce*, *MapOverlap*, *MapArray*, *Scan* etc.) uniformly across all backends. There exists no equivalents of the *MapArray*, *MapOverlap* skeletons in Muesli which allow to implement applications ranging from N-body simulation to Conjugate Gradient solver.

In this work, we extend SkePU for providing scalability across multiple nodes of GPU clusters, such that the same SkePU application can now run on several nodes for the provided simple and regular skeletons. Each node, being a complete multi-GPU system, runs one instance of SkePU and the given computation is partitioned among the nodes. By a simple compiler switch, the application programmer can run the code on a GPU cluster e.g. for running the application for larger problem sizes that may not fit in one or two GPUs' device memory space. We perform experiments for four scientific applications and one synthetic matrix sum application by expressing their computation intensive parts in terms of SkePU skeletons. We explain one application in details. Initially, we see that simple algorithms like the brute force implementation of N-body simulation and the calculation of electric field on a 3D grid scale across multiple nodes. We also show that extending the lazy memory copying mechanism across multiple nodes gives benefit in terms of speedup.

The rest of this paper is outlined in the following way. In Section 2, we provide some background knowledge about the SkePU skeleton library. Section 3 presents the extension of SkePU, in our current prototype for all its dataparallel skeletons and the vector container. Section 4 explains one of the four scientific applications rewritten in SkePU and how the extended version of SkePU works for it with respect to data communication, synchronizations and other details. Section 5 gives the experimental results and discussion. Finally, Section 6 concludes the paper.

2. The SkePU Library

SkePU is a C++ template library that provides a simple and unified interface for specifying data- and task-parallel computations with the help of skeletons on GPUs using CUDA and OpenCL [3]. The interface is also general enough to support other architectures, and SkePU implements both a sequential CPU and a parallel OpenMP backend. SkePU provides six data parallel skeletons including *Map*, *MapArray*, *MapOverlap*, *Scan*, *Reduce* and *MapReduce* and one Generate skeleton for data initialization. These skeletons can operate on vector and matrix containers, which encapsulate the skeleton operand data and keep track of copies and thus allow to optimize memory transfers.

An example code written in SkePU using the *MapOverlap* skeleton is shown in Figure 1. The SkePU library provides a way to generate user functions using macros. The user function *over* is written using the *OVERLAP_FUNC* macro, where *over* is the name of the user function, *float* is the return type, *3* is passed as the overlap size parameter, *a* is a reference to an element in the used container. The last parameter is the actual user code that is translated into the selected backend. Notice that the semantics of the *MapOverlap* skeleton requires that here only *3* elements (before and after) the element pointed by *a* can be accessed in the user function *over* in the example SkePU code (Figure 1). The computation in the user function is expressed in terms of these seven values. During execution, the SkePU container transfers the required data according to the same semantics.

During compilation, the macro is converted into actual code based on the compilation flags set for backend selection. The set of SkePU user function variants generated from a macro based specification are placed in a struct *over* with member functions for CUDA and CPU, and strings for OpenCL. An instance of the struct *over* is passed to create an instance of the *MapOverlap* skeleton named *conv* in the *main* function in Figure 1. A vector *v0* is initialized with 40 elements and the user function *over* is applied on every element of *v0* using the skeleton *MapOverlap*. The code in the *main* function looks pretty sequential, but it is executed in parallel according to the selected backend.

The SkePU containers (Vector and Matrix) are implemented using the *STL vector*. These containers are smart and perform the necessary data transfers using a lazy memory copying mechanism. This means that the data is transferred

```
OVERLAP_FUNC(over, float, 3, a,
    return a[-3]*0.8f + a[-2]*0.4f + a[-1]*0.2f +
    a[0]*0.1f + a[1]*0.2f + a[2]*0.4f + a[3]*0.8f;
)
int main()
{
    skepu::Init(NULL,NULL);
    skepu::MapOverlap<over> conv(new over);
    skepu::Vector<float> v0(40, (float)10);
    skepu::Vector<float> r;
    skepu::cout << "v0: " << v0 << "\n";
    conv(v0, r, skepu::CONSTANT, (float)0);
    skepu::cout << "r: " << r << "\n";
    skepu::Finalize();
    return 0;
}
```

Fig. 1: SkePU code for a 1D convolution using the MapOverlap skeleton

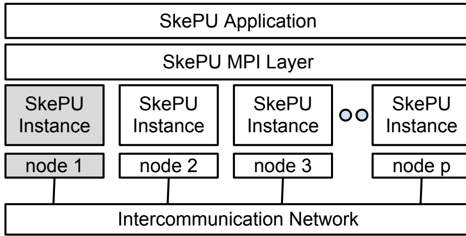

Fig. 2: SkePU instances with MPI layer

only when it is needed for performing computation or saving output data. If the input data is used for some other computation (kernel), it is not copied again. Furthermore, the data for intermediate results remains available for further computation (on GPU memory, with the CUDA/OpenCL backend) and transferred back to the host memory once it is required. The SkePU skeleton library is an on-going work with addition of more containers like sparse matrices for implementing irregular computations and dynamically changing data distributions. Furthermore, auto-tuning for selecting the execution plan has been addressed in [10]. In the current version of SkePU, the code is run on a single multi-GPU node and in the rest of the paper, we present the extension of SkePU for multiple nodes.

3. Extending SkePU by a MPI Layer

In this work we consider the extension of SkePU for the vector container only. On a single multi-GPU system (i.e., one cluster node), the skeletons can execute on one or more GPUs and upload their data to the device memory according to the selected number of GPUs or cores. The data access patterns for the vector container for different skeletons are shown in the upper portion of Figure 3. For example, in case of the *Map* skeleton, the user function f_2 is applied on the ith element of the input vector(s) and the result is stored in the corresponding i^{th} index of the output vector. In the *MapOverlap* skeleton, the neighbouring $overlap$ elements are also used in the calculation of ith output element.

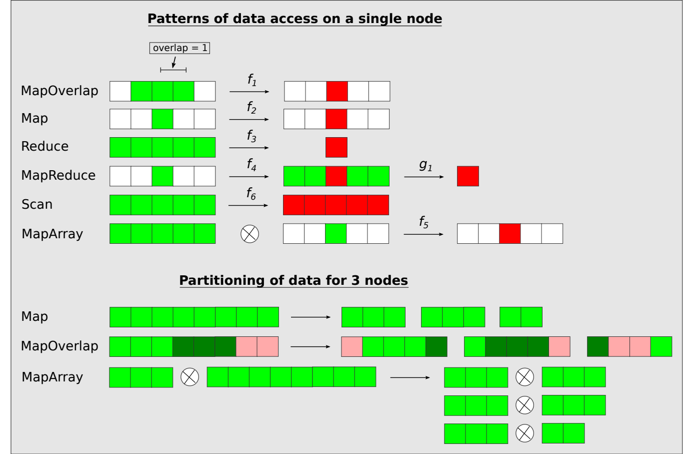

Fig. 3: Data access patterns for single node and data partitioning for 3 nodes.

The MPI Layer: In order to run skeletons on several nodes, each SkePU instance on a node runs the same code but the data is internally partitioned according to the semantics of the selected skeleton. For the necessary communication we add an MPI layer connecting the SkePU instances running on the nodes. The block diagram for the SkePU extension is given in Figure 2. The grey colored box shows the root node that runs the MPI process with rank 0. For broadcasting, scattering and gathering data over several nodes, we use the collective MPI calls, *MPI_BCast*, *MPI_Scatterv* and *MPI_Gatherv* respectively. There can be any number of nodes (and SkePU instances) over the MPI layer that is actually determined by the number of MPI processes selected while executing the parallel application. The MPI layer remains transparent to the user but the same SkePU code for the application will internally use the MPI layer to partition the data and computations.

Data Partitioning over Multiple GPU Nodes: The data partitioning is performed when a skeleton on the vector inputs is called. For example, in case of a *Map* skeleton and 3 nodes, a vector of length 8 is partitioned into 3 parts as shown in the lower portion of Figure 3. The output vector is also initialized on each node with the same lengths (as the lengths of the partitions of input vector). Then each instance of SkePU on each node computes the smaller computation with the same semantics of the *Map* skeleton. Notice that the *MapArray* skeleton has two vector inputs, when the ith element of the output vector is calculated (with the user function f_5, in Figure 3) by using the ith element of the second input vector and all the elements of first input vector. So, while partitioning, the second vector is partitioned in a similar way as in the *Map* skeleton case, but the first vector is broadcasted to each node. In this way, the *MapArray* computation is subdivided into smaller computations (that are performed according to *MapArray* semantics) on the

partitioned data. Similarly, in *MapOverlap*, the partitioning of the vector is similar as in the *Map* skeleton but *overlap* elements are attached in the beginning and the end of each partition as shown in Figure 3.

Data Partitioning Granularity: The partitioning is performed at the granularity level of the vector container and not at finer granularity. So, even for an update of a single element of a vector the whole partition will be communicated (if required). The number of partitions is the same as the number of MPI processes. For more complex containers and finer granularities of data communication, the container will also handle coherency (being addressed in on-going work).

Convolution Example for Multiple Nodes: In the example of Figure 1, we considered a 1D convolution computation using the *MapOverlap* skeleton. The SkePU code for the *MapOverlap* instance (conv), when executed on the GPU cluster, runs on all the nodes. The root process (MPI process with rank 0) initializes the vector *v0* with data and all the other processes (running on other nodes) keep the vector empty. When every process calls the *conv* skeleton, the root process partitions $v0$ in *p* parts where *p* is the number of MPI processes (or nodes used), and scatters the parts to *p* MPI processes. In each part, additional *d* elements, on both sides, are appended where *d* is the overlap size (here d = 3), which is a (statically known) parameter of the skeleton. Every MPI process (running one instance of SkePU) fills its $v0$ vector with the part it receives and performs the *over* kernel on the respective part (on CPU or GPU, according to the preselection made by the programmer). The results are gathered on the root process when they are required.

Lazy Memory Copying on Multiple Nodes: The lazy memory copying mechanism of the vector container is also extended for the cluster implementation of SkePU. In case the SkePU application is executed on a single node with CUDA or OpenCL backend, the input vector containers are uploaded on the device from the host memory and the references of those vectors are maintained in a hash table. Maintaining the hash table adds an extra overhead in the lazy memory copying mechanism but access to the hash table is an expected O(1) operation. If any of these input vectors is required again for another skeleton call, the reference of that vector already resides in the hash table so the vector is not uploaded again.

When the data in the vector is changed on the GPU and then accessed on the CPU the reference is removed from the hash table and the updated vector is downloaded. The lazy memory copying involves only the data uploading and downloading to/from the device memory. Whereas, in case of multiple nodes, data communication over the network is also involved besides the data uploading/downloading on device. As mentioned earlier the computation carried out for a skeleton on multiple nodes is actually done by the same skeleton calls on the different partitions of the operand data (vector) so the hash tables are also maintained on all the nodes including the root node. The root process performs the check whether the data is already distributed or not. It performs this check by finding the reference of the (partitioned) vector in the hash table. Then it either scatters the vector or does nothing depending upon absence or availability of the partitioned vector's reference in the hash table. On all the other nodes, similar checks are performed and the vector data is either gathered or nothing is done depending upon the absence or availability of vector data on the nodes' GPU devices.

In case of a backend not requiring a GPU device, there is no data uploading or downloading but the lazy memory copying is still useful in saving redundant communications over the network. This intelligent mechanism of data management is effective in terms of saving the redundant data transfers over the PCIe bus or the communication network. It can also happen that the capacity of the memory is less than the total required memory for all the operand vector elements used (in a large application) so lazy memory copying will cause an *out of memory* error. We have not considered this constraint in the current extension of SkePU. On the other hand, executing the application on several nodes may resolve this problem because the accumulated storage capacity of multiple GPUs will be larger than for a single GPU device memory and the data will be partitioned (requiring less memory on each GPU device). We will see in the results that the benefit of lazy memory copying depends upon the nature of the computations and data dependencies.

Implicit Barriers: As we are using the blocking collective calls of MPI, like *MPI_BCast* etc, there will be an implicit barrier in each collective call. There is no overlapping of data communications and computations. The semantics of SkePU as suggested by the sequential programming interface requires barrier synchronization where multiple nodes execute the code in an SPMD fashion. Certain applications may require barriers for correct computations so these implicit barriers are helpful. We will discuss possible benefits of these implicit barriers in the discussion of the N-body problem in Section 4.

Utilization of CPU Cores: The SkePU code follows the serial execution semantics between calls and for computation intensive data parallel kernels, it uses the selected backend for parallel execution. So at least one core is used for data uploading, downloading (on one node) and for data distribution (in case of multiple nodes). On each node, only one MPI process executes (irrespective of the number of CPU cores or GPUs) and based on backend selection, like the OpenCL, CUDA or OpenMP, the GPU and CPU cores are used (on each node).

The programmer needs not modify the code for running on the cluster, and in case the code is executed on the cluster, the distribution and synthesis of data is hidden from the programmer. The programmability of SkePU code is not affected, but scalability is achieved. We will discuss the

scalability in the Section 5. For all the other skeletons, the partitioning of the data is performed in the similar way.

4. N-body Simulation

The N-body simulation is a well known problem for understanding the interaction of N bodies (e.g. electric particles or galactic bodies). The time complexity of the naive serial algorithm is $O(KN^2)$ where N is the number of particles and K is the number of timesteps of the whole simulation.

The simple algorithm for N-body simulation in SkePU is shown in Figure 4. The application starts the serial execution and initializes the skeleton instances *nbody_init* and *nbody_simul-ate_step* using the user functions *init_kernel* and *move_kernel* respectively. Then two vectors of size N are created. The skeleton instance *nbody_init* is used to initialize the particles' initial positions, velocities, accelerations and masses. The particles are positioned in a 3D space with some initial velocities and accelerations. *ARRAY_FUNC* and *GENERATE_FUNC* are macros like *OVERLAP_FUNC* as explained in Section 2. After initialization of all the particles, the actual simulation starts in a *for-loop* using the *nbody_simulate_step* skeleton instance of *MapArray*. After every time step, the user function *move_kernel* is called using the *nbody_simulate_step* skeleton instance. The SkePU vector *all_particles* contains the positions, velocities and accelerations of all particles in the previous time step, and *ith_particle* points to the ith particle in *all_particles*. The *move_kernel* updates the ith particle as shown in Figure 4. The nature of the application is such that the output data is updated on the host (or root node in case of multiple nodes) so that the next skeleton call is made on the current state of the system. The skeletons are called *time_steps* times, with the first argument as the updated vector.

Execution using Multiple Nodes: In case the application is executed on a cluster (with multiple nodes), the first vector is internally broadcasted and the second vector is distributed/scattered to every MPI process in each iteration. In this way, the large problem is divided into smaller problem of the same nature. But due to the large computations the partitioning still gives benefit even there is communication overhead.

Synchronizations and Barriers: In each iteration, the skeleton call *nbody_simulate_step* is made two times. This is because the first argument requires the current state of the system of particles. The implicit barriers make it possible that the current state of the system is used after it is computed completely. This synchronization is inherently present in the nature of the computation of the N-body problem but using the current implementation (Cluster-SkePU) this synchronization is also enforced for any other application and overlapping of computation and communication cannot be exploited (we are addressing this in on-going work).

```
GENERATE_FUNC( init_kernel, Particle, index, seed,
   Particle p;
   // initialize location, velocity etc
   return p;
)
ARRAY_FUNC( move_kernel, Particle, all_particles, ith_particle,
   // calculate the force exerted on ith_particle from
   // all the other particles given in all_particles
   // update acceleration, velocity and position of ith_particle
   return ith_particle;
)
int main()
{
   skepu::Generate<init_kernel> nbody_init(new init_kernel);
   skepu::MapArray<move_kernel> nbody_simulate_step(new move_kernel);
   skepu::Vector<Particle> particles(n);
   skepu::Vector<Particle> latest(n);
   nbody_init(n, particles);
   nbody_init(n, latest);
   for(t=0;t<time_steps/2; t=t+1)
   {
      nbody_simulate_step(particles, latest, latest);
      // Update vectors on the host
      nbody_simulate_step(latest, particles, particles);
      // Update vectors on the host
   }
}
```

Fig. 4: SkePU code for N-body Simulation

Ratio of Computations and Communications: In this application, the parallel computations performed by the threads (either CUDA, OpenMP) are $O(N^2)$ in each update of the system of particles whereas the amount of data communicated is $O(N)$.

Effect of Lazy Memory Copying: The nature of computation of N-body simulation does not exploit the benefit of lazy memory copying.

The code for N-body simulation is simply written as serial code in C++ in terms of skeletons but executes on several SkePU backends including the GPU cluster backend.

All the other selected scientific applications are expressed in SkePU in a similar way in terms of *MapArray*, *Generate* and *Map* skeletons by following their semantics. The code looks serial but, following the semantics of the given skeletons, the expressed code can be executed on all backends implemented in SkePU.

5. Experimental Results

We have implemented several scientific applications (expressing their data parallel computations in terms of SkePU skeletons) including, N-body simulation, electric field calculation, smoothed particles hydrodynamics, one-level of Strassen's recursive matrix multiplication algorithm, and a synthetic matrix sum application. We performed experiments on two machines M1 and M2 and used OpenMP/MPI and CUDA/MPI backends respectively. Machine M1 has 805 nodes (for checking the scalability, we use up to 16 nodes only as more than 16 nodes were not accessible to us), each with two 2.33 GHz Xeon quad core chips and at least 16 GiB RAM, running under CentOS5. The nodes are interconnected by Infiniband. Machine M2 has 6 nodes each with 2 Intel Xeon E5620 CPUs, 3 NVIDIA Tesla M2090 GPUs with NVIDIA CUDA Toolkit V.4.0, and nodes

are interconnected by Infiniband. We could use up to 3 accessible nodes with single GPU (with CUDA only) on each node for experiments.

Scalability over Multiple Nodes: In the first experiment, the results show the scalability for the first three scientific applications with CUDA/MPI and/or OpenMP/MPI backends as shown in Figure 5. The horizontal axis in Figure 5 shows the number of particles for N-body simulation, smoothed hydrodynamics and electric field applications. The vertical axis shows the speedup for the three applications. The graphs mentioned with 1C/2C show the speedup for 2 CUDA nodes against a single CUDA node on M2 for each application in Figure 5. We also found that the CUDA/MPI backend with three nodes on M2 gives at most 4X performance than the OpenMP/MPI backend with 16 nodes on M1 for two scientific applications (shown by 16P/3C in Figure 5) besides the performance portability across different parallel architectures without code modification. These three scientific applications are computation intensive such that each node gets considerably large computations to perform for the given amount of communicated data among the nodes. For example, (considering the CUDA/MPI backend), in case of N-body simulation, $O(N)$ parallel tasks (each containing $O(N)$ operations) are performed by P nodes and data communication per iteration of N-body simulation will be $O(N)$. In this case, distributing the computation will have more benefit than the communication overhead. Note also that these computations are quite regular and use brute force algorithms, whereas better $O(N \log N)$ work algorithms exist that we considered in on-going extension work of SkePU.

For the SkePU implementation of other scientific applications with $O(N)$ parallel tasks each of asymptotically less than $O(N)$ work (e.g. $O(\log N)$ operations), we experienced increased communication cost outweighing the benefit of distributing the computation among several GPU nodes. This is because of the regular patterns of (blocking) communication (at the granularity level of containers) hidden in the simple skeletons in which the data parallel computations of the applications are expressed. Here we experience that finer granularity levels of communications are required for optimizing the communication as in [11] and [12]. The authors in [11] and [12] demonstrate the scalability of scientific applications like fluid dynamics, Strassen's algorithm and conjugate gradient method using CUDA and MPI over multiple nodes by using non-blocking (optimized) communication among the nodes. Hence, more complex skeletons and *smart containers* are required to express non-trivial and irregular scientific applications with varying granularity of data partitioning, prefetching and data caching. As we noted above certain computation intensive scientific applications can still get benefits from scaling over multiple nodes with ease of writing the parallel code (we demonstrated three).

Lazy Memory Copying over Multiple Nodes: In the second experiment, we implemented a one-level recursive variant of Strassen's algorithm for matrix muliplication and another synthetic matrix sum application that simply adds 12 $N \times N$ matrices. The results are shown in Figure 6. In the one-level Strassen's algorithm, two matrices A and B with dimensions $N \times N$ are partitioned into submatrices A_{11}, A_{12}, A_{21}, A_{22} and B_{11}, B_{12}, B_{21}, B_{22} respectively. These submatrices are used (more than once, but communicated only once with lazy memory copying) in the computations of intermediate product submatrices $P_1, ..., P_7$. Similarly, the intermediate product submatrices $P_1, ..., P_7$ stay distributed (not communicated) and the final result's submatrices $C_{11}, C_{12}, C_{21}, C_{22}$ are computed (see Figure 7). We see that lazy memory copying reduces the communication and gives speedup against the application that does not use lazy memory copying. Similarly, in our synthetic matrix sum application, the intermediate result matrix R is not communicated until the last addition is done (see Figure 7). Here, we see even more benefit than with Strassen's algorithm. This is because most of this synthetic matrix sum application benefits from lazy memory copying.

We further observe that the benefit of lazy memory copying decreases for the two applications when multiple nodes are used (as shown in Figure 6). For example, we see that the speedup decreases for both the applications when two nodes are used. Whereas, when 3 nodes are used, the benefit decreases even more in the synthetic matrix sum application but increases in case of Strassen's algorithm. This is because of the following reasons. In case of a single node, more data is transferred to the device memory using the PCIe bus (without partitioning of data). Whereas, in case of 2 nodes, first the data is partitioned in to 2 parts (scattered over the network) and then smaller partitions are transferred to device memories (on each node) in parallel. This decreases the overall transfer time on PCIe bus. So if we save these data transfers, we save less data transfer time (on 2 nodes) and hence speedup decreases as compared to a single node. A further increase in the number of nodes (and partitions) decreases the benefit even more in the case of the synthetic matrix sum application because the data is scattered over the network and even more smaller (three) partitions are transferred on three device memories in parallel. But in Strassen's algorithm we are using the *MapArray* skeleton (in several skeleton calls with broadcasts of several matrices) that increases the communication over the network with increasing number of nodes. So saving the communication with lazy memory copying gives more benefit. Although we get a benefit by lazy memory copying on multiple nodes, the nature of the application affects the speedup. The benefit achieved using the lazy memory copying also suggests to explore more smartness in future work on regular and irregular distributed containers.

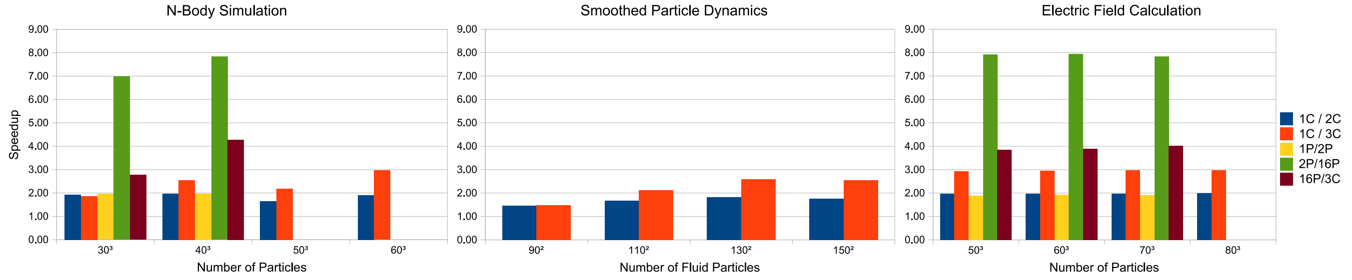

Fig. 5: Speedup of three scientific applications. xC: x nodes are used each with 512 CUDA threads. xP: x nodes are used each with 8 OpenMP threads.

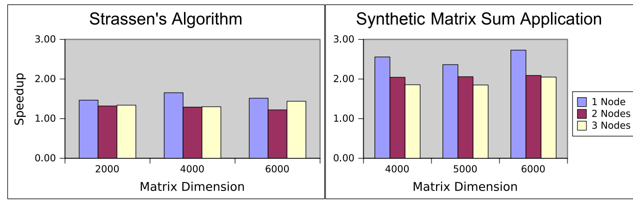

Fig. 6: Speedup gained by using lazy memory copying for one level Strassen's algorithm and a synthetic matrix sum application

Strassen's Algorithm (Recursive step)

$P_1 = (A_{11}+A_{22})*(B_{11}+B_{22})$
$P_2 = (A_{21}+A_{22})*B_{11}$
$P_3 = A_{11}*(B_{12}-B_{22})$
$P_4 = A_{22}*(B_{21}-B_{11})$
$P_5 = (A_{11}+A_{12})*B_{22}$
$P_6 = (A_{21} - A_{11})*(B_{11}+B_{12})$
$P_7 = (A_{12} - A_{22})*(B_{21}+B_{22})$

$C_{11} = P_1 + P_4 - P_5 + P_7$
$C_{21} = P_2 + P_4$
$C_{12} = P_3 + P_5$
$C_{22} = P_1 + P_3 - P_2 + P_6$

Synthetic Matrix Sum App.

$R = A_1 + A_2$
$R = R + A_3$
$R = R + A_4$
$R = R + A_5$
$R = R + A_6$
...
$R = R + A_{12}$

Fig. 7: Possibilities of lazy memory copying in one-level Strassen's algorithm and synthetic matrix sum application

6. Conclusions and Future Work

We have provided the principles and a first prototype for the extension of SkePU for GPU clusters and implemented four scientific and one synthetic matrix sum application. To the best of our knowledge, this is the first skeleton library implementation for GPU clusters that is also evaluated on a GPU cluster (note that the recent framework by Ernsting et al. [7] is evaluated with multiple MPI processes running either on a single GPU node or on a non-GPU cluster). We performed two experiments where the results show scalability, portability and programmability. SkePU code looks serial (easy to maintain and debug) and can be compiled and executed using a number of backends without code modification. We found that certain computation intensive applications (expressed in SkePU skeletons) can scale over multiple nodes even with the current extension of SkePU. The smartness of the containers can give speedup (depending upon the nature of computations). Future work will address improvements in the containers and skeletons in order to make Cluster-SkePU more useful in different domains of scientific computing.

Acknowledgments

This work was partly funded by Higher Education Commission Pakistan (HEC), Swedish e-Science Research Center (SeRC), and EU FP7 project PEPPHER (www.peppher.eu). We are also thankful to Supercomputing Centers NSC (www.nsc.liu.se) and PDC (www.pdc.kth.se) for providing us with the computing resources and technical support.

References

[1] Top500 supercomputer sites. www.top500.org, Nov. 2012.
[2] K. A. Yelick, S. Chakrabarti, E. Deprit, J. Jones, A. Krishnamurthy, and C. po Wen. *Data structures for irregular applications,* DIMACS Workshop on Parallel Algorithms for Unstructured and Dynamic Problems, 1993.
[3] J. Enmyren and C. W. Kessler. *SkePU: a multi-backend skeleton programming library for multi-GPU systems,* Proceedings of the fourth international workshop on High-level parallel programming and applications, pages 5-14, New York, NY, USA, 2010.
[4] M. Cole. *Recursive splitting as a general purpose skeleton for parallel computation,* Proceedings of the Second International Conference on Supercomputing, pages 133-140. 1987
[5] M. Cole. *Algorithmic skeletons: structured management of parallel computation,* MIT Press, Cambridge, MA, USA, 1991.
[6] M. Cole. *A skeletal approach to the exploitation of parallelism,* Proceedings of the conference on CONPAR, pages 667-675, 1988 New York, NY, USA, 1989.
[7] S. Ernsting and H. Kuchen. *Algorithmic skeletons for multicore, multi-GPU systems and clusters,* International Journal of High Performance Computing and Networking, pages 129-138, 2012.
[8] M. Steuwer, P. Kegel, and S. Gorlatch, *SkelCL - A portable skeleton library for high-level gpu programming,* Parallel and Distributed Processing Workshops (IPDPSW-11), pp. 1176-1182. 2011.
[9] M. Alind, M. V. Eriksson, and C. W. Kessler. *Blocklib: a skeleton library for Cell broadband engine,* 1st Int. Workshop on Multicore Software Engineering (IWMSE-08) , pages 7-14, Leipzig, Germany 2008.
[10] U. Dastgeer , J. Enmyren , C. W. Kessler, *Auto-tuning SkePU: a multi-backend skeleton programming framework for multi-GPU systems,* Fourth Int. Workshop on Multicore Software Engineering (IWMSE-11) USA, p.25-32, May 2011.
[11] D. Jacobsen, J. C. Thibault, and I. Senocak. *An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters,* Aerospace Sciences Meeting and Exhibit (AIAA-10), 2010.
[12] N. Karunadasa and D. N. Ranasinghe. *Accelerating high performance applications with CUDA and MPI,* ICIIS'09, pages 28-31.