Managing Overlapping Data Structures for Data-Parallel Applications on Distributed Memory Architectures

Mads Ruben Burgdorff Kristensen and Brian Vinter

***Abstract*— In this paper, we introduce a model for managing abstract data structures that map to arbitrary distributed memory architectures. It is difficult to achieve scalable performance in data-parallel applications where the programmer manipulates abstract data structures rather than directly manipulating memory. On distributed memory architectures such abstract data-parallel operations may require communication between nodes. Therefore, the underlying system has to handle communication efficiently without any help from the user. Our data model splits data blocks into two sets -- local data and remote data -- and schedules the sub-block by availability at runtime.**

We implement the described model in DistNumPy -- a high-productivity programming library for Python. We go on to evaluate the implementation using a representative distributed memory system -- a Cray XE-6 Supercomputer -- up to 2048 cores. The benchmarking results demonstrate scalable good performance.

***Index Terms*—HPC, NumPy, High-Productivity, Data-Parallel, DistNumPy**

I. Introduction

High-productivity programming languages are very popular in the computational scientific community because they enable quickly prototyping of numerical problems. Common for most high-productivity languages is high-level operation on data structures such as vectors and matrices because they increase the productivity and remove a broad range of typical errors. Two high-productivity languages, MATLAB and Python, are popular in the scientific community precisely because of a rich set of high-level vector and matrix operations.

It is possible to execute parallel applications written in a high-productivity language that make use of data parallelism without reducing the productivity[4, 11]. This is because data parallelism is ideal for high-level vector and matrix operations. Data parallelism refers to a parallel model where a single instruction is distributed between processes based on data locality. Therefore, data parallelism provides full knowledge of data distribution and parallelization to all participating processors, which makes it possible for the runtime system to execute vector operations seamlessly in parallel without further assistance from the user. Additionally, the processors need not communicate when performing data dependency analysis and scheduling optimizations at runtime. However, the downside of data parallelism is that it reduces the programmability because the user is restricted to vector operations.

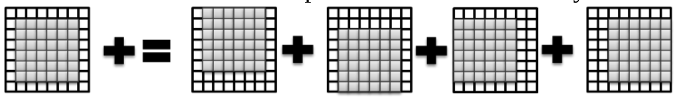
Fig.1, Matrix expression of a simple 5-point stencil computation example. See Figure 2 for the expression in MATLAB and Figure 8 for the expression in Python.

```
I %Number of iterations
A %Input & Output Matrix
SIZE %Symmetric Matrix Size
T = %Temporary array
i = 2:SIZE+1;%Center slice vertical
j = 2:SIZE+1;%Center slice horizontal
for n=1:I,
    T(:) = A(i,j) + A(i+1,j) + A(i-1,j) ...
           + A(i,j+1) + A(i,j-1);
    A(i,j) = T;
end
```

Fig. 2, 5-point stencil application that uses Jacobi Iteration in a fixed number of iterations implemented in MATLAB.

When expressing algorithms through high-level vector and matrix operations, or simply array operations, the user needs a mechanism to specify a subset of an array. E.g., Figure 1and 2 illustrate how one implements a 5-point-stencil computation in MATLAB by operating on *views* of arrays. In contrast, conventional programming languages would require using tedious scalar operations with *for* loops and index arithmetic.

These array views are data structures that maps to arbitrary distributed memory and thus possible overlapping memory. In the context of this paper, we will use array views as a synonym for such abstract data structures that may refer to parts of the same underlying data.

Array views gives rise to a number of important performance challenges when combined with data parallelism where the shared data is distributed across multiple processes. The problem is that operations on views may translate into *non-aligned* distributed array operations, which are difficult to handle efficiently. We define an *aligned* distributed array

Manuscript received February 16, 2012. Mads R. B. Kristensen and Brian Vinter is with Niels Bohr Institute, University of Copenhagen, Denmark. Email: madsbk@nbi.dk/vinter@nbi.dk.

DOI: 10.5176_2010-2283_2.1.142

operation as an operation on arrays that are distributed in a conformable manner, i.e. the arrays use identical data distribution. A non-aligned distributed array operation is then an operation without this property.

In this paper, we will introduce a data model that provides efficient handling of overlapping data structures. We will concretize the data model by implementing efficient array views in the high-productivity language DistNumPy[11], which interprets NumPy applications as data parallel applications in a distributed memory environment. In order to achieve good scalable performance we leverage the work by [14] who introduce an efficient communication latency-hiding model.

A. Related Work

Libraries and programming languages that strive to support parallelism in a high productive manner is a well-known concept. In a perfect framework all parallelism introduced by the framework is completely hided from the user while the performance and scalability archived is optimal. However, most frameworks require the user to specify some kind of parallelism -- either explicitly by using parallel directives or implicitly by using parallel data structures.

High Performance Fortran (HPF)[12] and ZPL[3] are two well-known examples of data-parallel programming languages that supports abstract data structures. HPF is a Fortran-based data-parallel programming language that requires static compilation for distributed-memory systems[10]. To obtain good parallel performance the user must *align* arrays together to reduce communication[1]. Our data model manages computation and communication of abstract data structures at runtime, which enables on-the-fly data dependency analysis. Using our model the user will not have to *align* arrays in order to obtain good parallel performance.

Python extensions, NumPy[13] and SciPy[9], have been successfully used in scientific computing[6] because their high-level abstractions are very close to mathematical formulas and there exist a super rich set of Python packages for almost any common task. Similarly, MATLAB is very popular because of a high-level data structure abstraction support. NumPy, SciPy, and MATLAB are targeting single-node systems where as our model is targeting multi-node systems. There exists extension to MATLAB that targets multi-node systems. MATLAB*P[4] introduces data-parallelism in MATLAB with support for high-level data structure abstraction.

II. TARGET DATA-PARALLEL APPLICATIONS

Data-parallel applications are a class of applications that make use of data parallelism -- either explicitly handled by the programmer or implicitly handled by the programming language or library. In this work, we focus on data-parallel applications written in a high-productivity language where the programming language, scientific library, and/or runtime system handles the data parallelism seamlessly.

We target applications with the following properties:

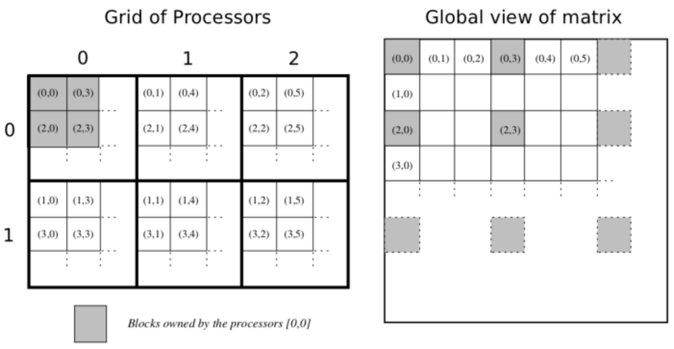

Fig. 3, The Two-Dimensional Block Cyclic Distribution of a matrix on a 2 x 3 grid of processors.

- The application uses high-level array operations instead of explicitly programmed *for* loops.
- The application uses data parallelism to execute vector/array operation in parallel.
- In order to utilize distributed memory architectures, the application distribute data evenly across process using a static distribution scheme.

The application uses data structures that maps to arbitrary distributed memory, e.g. by using data structures, such as array views, that may refer to parts of the same underlying data.

A. Data Distribution

Data parallelism is a classic approach to support distributed memory architectures. It clearly defines how data and computation is distributed across processes when combined with a static distribution scheme. Two-Dimensional Block Cyclic Distribution is a very popular distribution scheme and it is used in numerical libraries such as ScaLAPACK[2] and LINPACK[5]}. It supports matrices and vectors and has a good load balance in numerical problems that have a diagonal computation workflow e.g. Gaussian elimination. The distribution scheme works by arranging all processes in a two dimensional grid and then distributing data-blocks in a round-robin fashion either along one or both grid dimensions (Fig. 3); the result is a well-balanced distribution.

B. Array Operations

High-level array operation is relevant for all kinds of computations. Some array operations are very domain specific and other array operations are very general. Element-wise operations on arrays are an elementary part of most high-productivity languages and libraries. It simplifies the programming because it replaces computation loops, including index arithmetic, with one single operation.

Element-wise operations take a fixed number of scalar inputs and produce a fixed number of scalar outputs. E.g., an element-wise addition takes three array-views as argument: two input arrays and one output array. For each element, the operation adds the two input arrays together and writes the result into the output array. Applying an element-wise operation on a whole array is semantically equivalent to

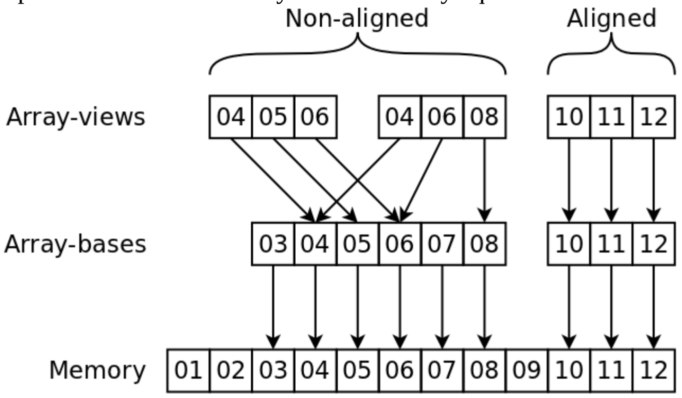

Fig. 4, Reference hierarchy between the two array data structures and the main memory. Only the three array-views at top of the hierarchy are visible from the perspective of the user.

performing the operation on each distributed array block individually. This property makes it possible to perform the distributed element-wise operation in parallel.

C. Array Views

Array views are essential when expressing algorithms through high-level array operations. It makes it possible to apply an operation on a subpart of an existing array without memory copying. Conceptually, array views form a hierarchy where each array view points to an underlying ``base". This ``base" is then an array that maps directly to a contiguous piece of memory. We define the two terms, array-base and array-view, as follows:

- **Array-base** is the base of an array and has direct access to the content of the array in main memory. An array-base is created with all related meta-data when the user allocates a new distributed array, but the user will never access the array directly through the array-base. The array-base always describes the whole array and its meta-data such as array size and data type are constant.
- **Array-view** is a view of an array-base. The view can represent the whole array-base or only a sub-part of the array-base. An array-view can even represent a non-contiguous sub-part of the array-base. An array-view contains its own meta-data that describe which part of the array-base is visible. The array-view is manipulated directly by the user and from the users perspective the array-view is simply a normal contiguous array.

For simplicity, array-views are not allowed to refer to each other, which mean that the hierarchy is flat with only two levels: array-base below array-view. However, multiple array-views are allowed to refer to the same array-base. This hierarchy is illustrated in Figure 4.

III. Non-Aligned Array Operations

Managing overlapping data structures, aka array-view, for data-parallel applications on distributed memory architectures gives rise to a number of important performance challenges.

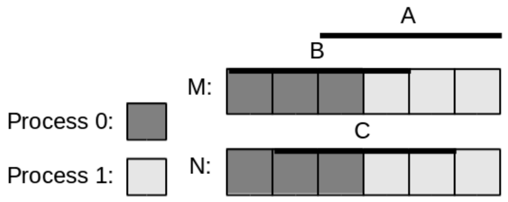

Fig. 5, The data layout of the two arrays M and N and the three array-views A, B and C in the 3-point stencil application. The arrays are distributed between two processes using a block-size of three.

The problem is that element-wise operations on array-views may translate into *non-aligned* distributed array operations, which are difficult to handle efficiently. That is, element-wise operations on array-views that does not map directly to the underlying array-base.

For example, a 3-point stencil application uses three array-views, *A*, *B* and *C*, to express a stencil. When executing on two processes the two underlying array-bases, *M* and *N*, are distributed according to Fig. 5. It is clear that *A* and *C* does not map directly to the underlying array-bases *M* and *N*. Thus, the result is a non-aligned array operation. In order to execute such an application the two processes must exchange data blocks, which mean commutation when executing on a distributed memory architecture. Therefore, an efficient data structure model that minimizes communication is vital for the parallel performance.

IV. Managing Non-Aligned Array Operations

The main contribution in this work is a model for managing non-aligned array operations efficiently. We introduce a hierarchy of data structures that makes it possible to divided non-aligned array operations into aligned blocks at runtime while minimizing the total amount of communication.

The model consists of three kinds of data blocks: base-blocks, view-blocks and sub-view-blocks, which make up a three level abstraction hierarchy (Fig. 6).

- **Base-block** is a block of an array-base and maps directly into one block of memory located on one node. The memory block is contiguous and only one process has exclusive access to the block. The base-blocks are distributed across multiple processes in a round-robin fashion according to the N-Dimensional Block Cyclic Distribution.
- **View-block** is a block of an array-view and from the perspective of the user a view-block is a contiguous block of array elements. A view-block can span over multiple base-blocks and consequently also over multiple processes. For a process to access a whole view-block it will have

to fetch data from possible remote processes and put the pieces together before accessing the block. To avoid this process, which may cause some internal memory copying, we divide view-blocks into sub-view-block.

- **Sub-view-block** is a block of data that is a part of a view-block but is located on only one process. The memory block is not necessarily contiguous but only one process has exclusive access to the block. The driving idea is that all array operation is translated into a number of sub-view-block operations.

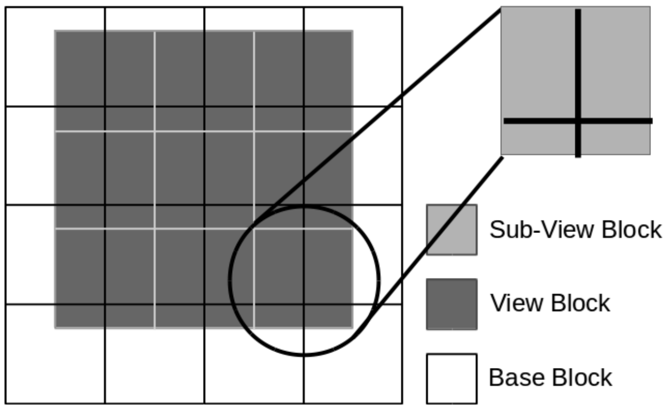

Fig. 6, An illustration of the block hierarchy that represents a 2D distributed array. The array is divided into three block-types: Base, View and Sub-View-blocks. The 16 base-blocks make up the base-array, which may be distributed between multiple processes. The nine view-blocks make up a view of the base-array and represent the elements that are visible to the user. Each view-block is furthermore divided into four sub-view-blocks, each located on a single process.

In this data model, an aligned array is an array that has a direct contiguous mapping through the block hierarchy. That is, a distributed array in which the base-blocks, view-blocks and sub-view-blocks are identical. A non-aligned array is then a distributed array without this property.

It is straightforward to parallelization aligned array operations because each view-block is identical to the underling base-block and is located on a single process. On the other hand, when operating on non-aligned arrays each view-block may be located on multiple processes. Therefore, we have to divide the computation into sub-view-blocks and even into aligned blocks of sub-view-blocks, which makes the operation more complex and introduces extra communication and computation overhead.

At the user level, an array operation operates on a number of input array-views and output array-views. It is the user's responsibility to make sure that the shape of these array-views matches each other. Since all arrays uses the same block size, this guaranties that all involved view-blocks match each other. Thus, it is possible to handle one view-block from each array at a time. In order to compute an array operation in parallel all available processes computes a view-block using the following steps:

1) The process fetches all the remote sub-view-blocks that constitute the involving input view-blocks.
2) The process aligns the sub-view-blocks by dividing them into the smaller blocks that are aligned to each other. If some output sub-view-blocks is not located on the process it will use temporary memory for the output.
3) The process applies operation on these aligned blocks.
4) The process sends temporary output sub-view-blocks back to the original locations.

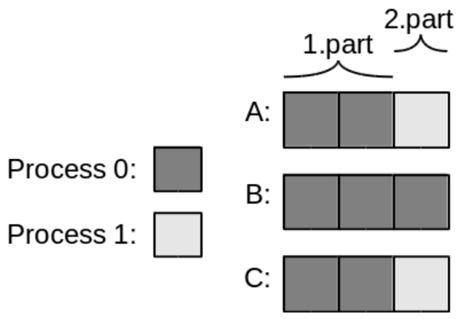

Fig. 7, The sub-view-block alignment of the first view-block in the three array-views *A*, *B* and *C* (Fig. 5).

A. References

To demonstrate how the model works we will walk through the execution of the first block in a small 3-point stencil application. Two processes are executing the stencil application with the two array-bases, *M* and *N*, using a block-size of three elements. This means that three contiguous array elements are located on each process (Fig. 5). The application uses two input array-views, *A* and *B*, and one output array-view, *C*, to compute the 3-point stencil.

In order to compute the first view-block in the three array-views, process 0 divides the computation into two parts (Fig. 7). The first part, which consists of the first two elements, needs no communication since all elements are located locally. The process can therefore apply the operation directly on the first two elements of each array.

The second part, which consists of the third element, needs communication. The two processes will transfer the third element in *A* from process 1 to process 0. Even though the third element in *C* is located remotely, no communication is need now because *C* is the output. Instead, a temporary memory location is used for the output element. The process will apply the operation when the communication the element is finished. When process 0 finishes the computation of part 2 the process transfer the third element back to process 1.

B. Latency-Hiding

It is essential to the performance of non-aligned array operations that the execution hides communication latency behind computation. In order to accomplish this we make use of the Latency-Hiding model introduces in [14]. Using this model, we initiate non-blocking communication at the earliest time and only do computation after all communication has been initiated. Furthermore, we check for communication completion between multiple computation operations to make sure that there is progress in the communication layer. The execution flow is as follows:

1) Initiate all non-depended communication operations.
2) Check if any communication operations has been finished in a non-blocking manner and insert operations that have no dependencies into the ready queue.
3) When only computation operations are ready, execute one of them and move new operations that have no dependencies into the ready queue.
4) Go back to step one if there are unfinished operations or else terminate.

The algorithm maintains the following three invariants:
1) All ready operations are in the ready queue.
2) Computation operations are executed only when there is no communication operation in the ready queue.
3) Communication operations are checked for completion when there is no computation operation in the ready queue.

Table 1. Cray XE-6 Supercomputer

Processor	AMD Opteron 6172
Clock	2.1 GHz
Peak Performance per Core	8.4 Gflops
Cores per NUMA Domain	6
NUMA Domains per Node	4 (packaged in 2 sockets)
Total Cores per Node	24
Private L1 Data Cache	64 KB
Private L2 Data Cache	512 KB
Shared L3 Cache per Socket	12MB
Memory Bandwidth	25.6 GB/s
Memory per Node	32GB DDR3-1066 ECC
Compiler	PGI 11.3
Math Library	Cray Scientific Library 10.5
Interconnect	Gemini 3-D Torus
Peak Bandwidth (per direction)	7 GB/s
MPI	Cray MPI 5.1.4

V. Distributed Numerical Python

In order to demonstrate the efficiency of our model for managing abstract data structures, we optimize the numerical Python library Distributed Numerical Python (DistNumPy) [11] using our model. DistNumPy is a new version of NumPy[13] that parallelizes array operations in a manner completely transparent to the user -- from the perspective of the user, the difference between NumPy and DistNumPy is minimal. DistNumPy can use multiple processors through the communication library Message Passing Interface (MPI)[7]. However, DistNumPy does not use the traditional single-program multiple-data (SPMD) parallel programming model. Instead, the MPI communication in DistNumPy is fully transparent and the user needs no knowledge of MPI or any parallel programming model.

The only difference in the API of NumPy and DistNumPy is the array creation routines. DistNumPy allow both distributed and non-distributed arrays to co-exist thus the user must specify, as an optional parameter, if the array should be distributed. The following illustrates the only difference between the creation of a standard array and a distributed array:

```
#Non-Distributed
A = numpy.array([1,2,3])
#Distributed
B = numpy.array([1,2,3], dist=True)
```

The first version of DistNumPy does not support efficient non-aligned array operations. Its focus was scientific applications that uses aligned distributed array operations, such as Monte Carlo and N-body simulations. To address this shortcoming we introduce our model for managing abstract data structures efficiently. We expect good performance and scalability when combining this implementation with the latency-hiding model introduced in [14].

The implementation of DistNumPy is open-source and freely available (http://code.google.com/p/DistNumPy).

```
I #Number of iterations
A #Input & Output Matrix
SIZE //Symmetric Matrix Size
#Temporary array
T = empty([SIZE]*2,dtype=double ,dist=True)
for i in xrange(I):
  T[:]  = A[1:-1, 1:-1] #Center
  T    += A[1:-1, 0:-2] #Left
  T    += A[1:-1, 2:   ] #Right
  T    += A[0:-2, 1:-1] #Up
  T    += A[2:   , 1:-1] #Down
  A[1:-1, 1:-1] = T
```

Fig. 8, 5-point stencil application that uses Jacobi Iteration in a fixed number of iterations implement in DistNumPy.

VI. Experiments

In this section, we will evaluate the performance impact of our model for managing non-aligned array operations. We conduct all experiments on an Cray XE6 supercomputer (Table 1). The system systems consist of multi-core Non-Uniform Memory Access (NUMA) shared-memory nodes where each node has multiple NUMA domains. CPU cores within the same NUMA domain have uniform data access latency to the local memory while CPU cores of different NUMA domains would have non-uniform data access latencies. We will focus on the MPI communication overhead associated with non-aligned array operation and we will therefore only execute one MPI-process per NUMA domain.

To evaluate the performance, we will compare aligned array operations with non-aligned array operations. We use a 5-point stencil application that uses Jacobi Iteration in a fixed number of iterations. Figure 8 is this application implemented in

Python using the DistNumPy library. It expresses the 5-point stencil using five array views that are shifted one element in each direction and thereby non-aligned operations (Fig. 1). In order to benchmark the efficiency of the data structures hierarchy we introduce in this work, we compare this application with a synthetic version where all operations an aligned and do the same amount of computation. Because

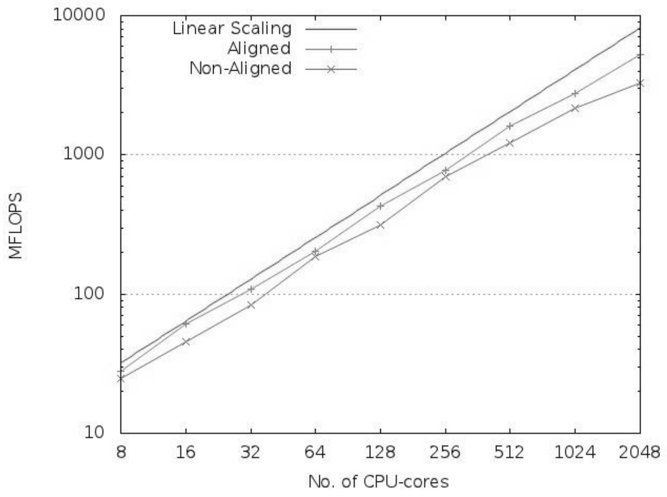

Fig. 9, Weak scaling of aligned and non-aligned array operation.

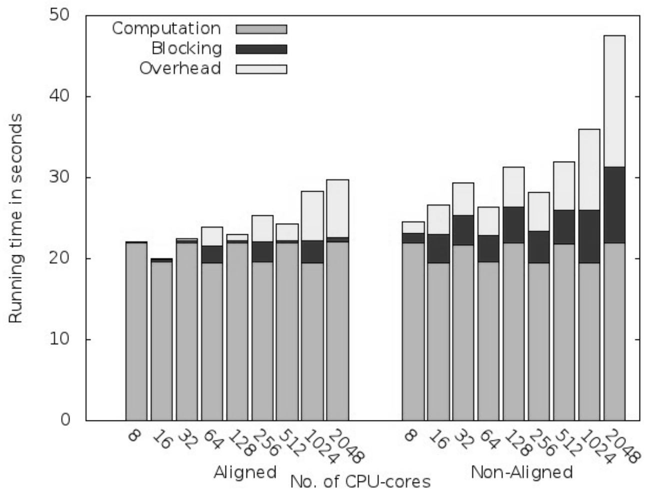

Fig. 10, Weak scaling of aligned versus non-aligned array operation.

of the exclusively use of aligned operation the synthetic version requires no communication. It should be emphasize that the synthetic version is purely for benchmark purposes and do no meaningful work.

The unfavorable computation-communication ratio in the 5-point stencil application makes it difficult to achieve good scaling performance. The asymptotic computational complexity is *O(n)* thus increasing the problem size does not improve the scaling performance significantly.

For the experiment, we calculate the FLOPS based on the floating operation counts of the ideal sequential algorithm and the measured execution times. Additionally, we compare the results with the linearly scaling performance, which we calculate by extrapolating the sequential FLOPS performance of NumPy. We use this comparison as an upper bound of the achievable scalable performance. We perform weak scaling experiments, in which the problem size is scaled with the number of CPU-cores in the executions. The experiment goes from 8 to 2048 CPU-cores where the CPU-cores and problem size doubles between each execution.

A. *Results*

Figure 9 shows the result of the experiment. Overall the result is very promising, we see a linear increase of performance in both the aligned and non-aligned version. The aligned version demonstrates a speedup of 1514 at 2048 CPU-cores compared to a sequential execution, which translates into a CPU utilization of 74%. The non-aligned version demonstrates a speedup of 948 at 2048 CPU-cores compared to a sequential execution, which translates into a CPU utilization of 46%.

To analyze the experiment result further we divide the execution time into three categories in Figure 10. The execution time in each category is the average timing from each process.

- **Computation** is the time used on actually computing element values. It should be fairly static through all the executions. However, variations in the data distribution may result in different execution times.
- **Blocking** is the time used on waiting for communication to finish. Each process will do as much work as possible before interring a blocking state. However, as the number of CPU-cores increases the chances that the job scheduler on the Cray system allocates distant nodes to a job also increases. Furthermore, the torus network performance may suffer from the communication traffics caused by other jobs.
- **Overhead** is the time used on handling the data structures associated with array operations. The overhead is proportional with the number of sub-view-blocks involved in the computation. Since the number of sub-view-blocks increases with the problem size, the overhead also increases. In addition, the number of sub-view-blocks increases even more when executing non-aligned operations.

As expected the blocking time is relatively small for all the aligned operation executions. Even at 2048, the blocking time is less the 2% of the total execution time. On the other hand, the blocking time for the non-aligned version is not as good. At 2048, the blocking time is 18% of the total execution time. This increase in blocking time is primarily because of an increase in communication, but also because of the MPI implementation by Cray. Currently, the Cray MPI for the Cray Gemini network has limited overlapping support for non-blocking MPI communication.

In the aligned operation version, the overhead time increases from 0.4% to 24% of the overall execution time. This overhead incensement is a direct result of the increased problem size. In the non-aligned operation version, the overhead increases more drastically -- going from 6% to 34% of the overall execution time. This is because the non-aligned operations results in four times the number of sub-view-blocks -- one sub-view-block per direction in the stencil computation.

VII. Conclusion

The single execution flow with abstract data operations is both the main strength and weakness of data-parallel programming models: two most notorious types of parallel programming bugs, data races and deadlocks, simply do not exist in data-parallel applications because there is only one execution thread. However, flexible abstract data operations for data-parallel applications require a very efficient runtime system in order to have good scalable performance.

In this work, we have successfully shown that by splitting data blocking based on locality it is possible to efficiently managing abstract data structures that map to arbitrary distributed memory. We demonstrate scalable performance of a Jacobi Iteration application up to 2048 CPU-cores.

Acknowledgment

This research is supported by the Danish Strategic Research Council, grant #09-063770. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

References

[1] S. Benkner, P. Mehrotra, J. Van Rosendale, and H. Zima. High-level management of communication schedules in HPF-like languages. In Proceedings of the 12th international conference on Supercomputing, ICS '98, pages 109–116, New York, NY, USA, 1998. Institute for Software Technology and Parallel Systems, University of Vienna. ISBN 0-89791-998-X.

[2] L. S. Blackford. ScaLAPACK. In Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing 96 Supercomputing 96, page 5, 1996.

[3] B. L. Chamberlain, S.-E. Choi, E. C. Lewis, C. Lin, L. Snyder, and D. Weathersby. ZPL: A Machine Independent Programming Language for Parallel Computers. Software Engineering, 26(3):197–211, 2000.

[4] R. Choy and A. Edelman. MATLAB*P 2.0: A unified parallel MATLAB. Technical report, Massachusetts Institute of Technology, January 2003.

[5] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. LINPACK users' guide. SIAM, 1, 1979.

[6] P. F. Dubois. Guest Editor's Introduction: Python: Batteries Included. Computing in Science Engineering, 9(3):7–9, may-june 2007. ISSN 1521-9615.

[7] W. Gropp, E. Lusk, and A. Skjellum. Using MPI Portable Parallel Programming with the Message Passing Interface. The MIT Press,1994.

[8] M. U. Guide. The MathWorks. Inc., Natick, MA, 5, 1998.

[9] E. Jones, T. Oliphant, P. Peterson, et al. SciPy: Open source scientific tools for Python, 2001.

[10] K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of High Performance Fortran: an historical object lesson. In Proceedings of the third ACM SIGPLAN conference on History of programming languages, HOPL III, pages 7–1, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7.

[11] M. R. B. Kristensen and B. Vinter. Numerical Python for Scalable Architectures. In Fourth Conference on Partitioned Global Address Space Programming Model, PGAS'10. ACM, 2010. ISBN 978-1-4503-0461-0.

[12] D. Loveman. High Performance Fortran. IEEE Parallel & Distributed Technology: Systems & Applications, 1(1):25, 1993.

[13] T. E. Oliphant. Python for Scientific Computing. Computing in Science and Engineering, 9:10–20, 2007. ISSN 1521-9615.

[14] M. Ruben Burgdorff Kristensen and B. Vinter. Managing Communication Latency-Hiding at Runtime for Parallel Programming Languages and Libraries. Arxiv Preprint arXiv:1201.3804v1, jan 2012.

Mads Ruben Burgdorff Kristensen is a PhD student at the Niels Bohr Institute, University of Copenhagen. His primary PhD study is in Supercomputing and Multi-core architectures. Current work includes seamlessly parallelism in scientific Python applications with special focus on exploiting distributed memory architectures.

Brian Vinter is Professor at the Niels Bohr Institute, University of Copenhagen. His primary research areas are Grid computing, Supercomputing and Multi-core architectures. He has done research in the field of High Performance Computer since 1994. Current research includes methods for transparent utilization of parallelism in scientific applications with special focus on exploiting distributed memory architectures such as CELL-BE and BlueGene.