APES on SX-ACE

Harald Klimach, Jiaxing Qi and Sabine Roller

Abstract We report on first experiences in deploying the APES framework on the NEC SX-ACE vector system. In APES there are two solvers available, implementing different numerical schemes. Musubi is a Lattice-Boltzmann solver that can be used to simulate incompressible flows. This numerical method is attractive as it allows the explicit computation for incompressible flows with good scalability and robust treatment of highly comlex geometries. The second solver, Ateles, implements a high-order Discontinuous Galerkin method and can be used to solve hyperbolic conservation laws, including linear equations like acoustics and non-linear equations like compressible Navier-Stokes. The NEC SX-ACE vector system offers a memory bandwidth to operation ratio of 1 Byte per floating point operation, which is an interesting deployment option for many numerical schemes. Though, there are a lot of experiences with earlier systems of the SX series, the latest installment comes with new features and an overhaul of the programming environment.

1 Introduction

The APES framework [1] is a collection of applications and libraries to enable large scale numerical simulations of fluid dynamics on distributed memory systems. It is written in Fortran and utilizes some Fortran 2003 features. Though, some features from the Fortran 2003 standard are extensively used throughout the code, the development tried to stay off from various language constructs that were troublesome with various compilers over a long time. One of the notable requirements from the Fortran 2003 standard is the ISO-C-Binding, which is used to incorporate the Lua scripting language for configuration files.

H. Klimach (✉) · J. Qi · S. Roller
University of Siegen, 57076 Siegen, Germany
e-mail: harald.klimach@uni-siegen.de

J. Qi
e-mail: jiaxing.qi@uni-siegen.de

S. Roller
e-mail: sabine.roller@uni-siegen.de

M.M. Resch et al. (eds.), *Sustained Simulation Performance 2016*,
DOI 10.1007/978-3-319-46735-1_4

There are mainly two fluid dynamic solvers developed within APES. One is the Lattice-Boltzmann solver Musubi [2], suitable for incompressible flows. The other is the high-order Discontinuous Galerkin solver Ateles [3] for compressible flows, but also other hyperbolic conservation laws. Both rely on a large part of shared infrastructure provided by the TreElM library [4]. Meshes are described with the help of Octree data structures, but in a sparse sense, where only elements are stored explicitly which are part of the computational domain. This sparsity results in the need for an indirection when accessing neighboring elements, like in unstructured mesh representation. The mesh format has been chosen specifically to cater large scale distributed parallel systems, besides the known topology of the Octree, a space-filling curve is used to partition the mesh. Using a space-filling curve together with the Octree enables a completely distributed partitioning, where each process can locally compute the elements it is going to work on.

Both solvers have been deployed on various large systems and have been shown to be capable of utilizing at least 100 thousand MPI processes. The APES tools have never been deployed on any earlier NEC SX vector system. Here we report some first experiences of porting this flexible framework to the NEC SX-ACE system. The NEC SX series of vector computers have a long standing tradition in high performance computing with a well-established development environment, especially for Fortran. A high memory bandwidth in relation to the floating point operation speed yields a well balanced system with 1 Byte per floating point operation. If we can exploit vector instructions in the APES solvers, we expect both solvers to benefit from the fast memory access and, thus, a high sustained performance.

Though, the APES solvers have been developed on and for cluster machines with mostly an Intel x86 architecture, vectorization was always a considered point in the development. Vectorization in the implementation is of increasing importance, as larger and larger vector instructions get into the processor architectures. The AVX2 instructions, for example, allow a single instruction to process four double precision numbers and AVX-512 extends this to eight. However, the 256 double precision numbers processed per instruction on the NEC SX processor is a completely different quality of vectorization, which requires a really strong level of vectorization in the implementation.

At least for the Lattice-Boltzmann kernel itself a good vectorization is known, due to the straight forward single loop. We therefore start out with the porting of Musubi to the NEC SX-ACE and have a look at other important parts besides the kernel. After that we move on to the little more involved Discontinuous Galerkin implementation in Ateles, where the computational effort is distributed across multiple important kernels and a vectorization is less obviously achieved.

2 Porting of Musubi

We started our porting efforts on Musubi, as this algorithm is known to perform well on the NEC SX architecture and the main kernel is straight forward to understand. The first step in porting an application is to ensure that the code can be compiled

Listing 1: No vectorization due to small intermediate arrays

```
1 do iVal=1,nVals
2   tmp(1) = a(iVal)
3   tmp(2) = b(iVal)
4
5   result = var(1) + var(2)
6 end do
```

for the target system. Luckily, the compiler environment on the NEC SX-ACE provides a Fortran 2003 compiler. This new compiler is a complete overhaul of the old compiler that only provided a very limited subset of the Fortran 2003 standard. One drawback of this new compiler is a less sophisticated optimization and vectorization, when compared to the previous more restricted compiler. Nevertheless, we found the compiler to work pretty well on Musubi and were able to tune the implementation towards a high sustained performance of about 30 % of the peak performance in the Lattice-Boltzmann kernel.

2.1 Porting of the Kernel

The Lattice-Boltzmann kernel basically is just a single loop over all lattice points. It is an explicit scheme and a double buffer is used to hold the values of the new and old time step. Therefore, there is no dependency and the expectation was a straight forward vectorization of the loop. Thanks to the NEC sxf03 compiler the initial porting without optimization was straight forward. Only very minor issues arose with some modern Fortran constructs that were easily resolved. However, the performance was disappointing and especially the vector length in vector instructions was far below the possible maximum of 256. As it turned out, the implementation of the Lattice-Boltzmann loop in Musubi made use of smaller arrays to hold temporary values. This convenience confused the compiler and had it vectorizing these arrays instead of the outer loop. To allow the compiler to put those temporary values into vector data registers, they had to be turned into individual scalars.

An illustration of the code layout where the long loop over all lattices is not vectorized is given in Listing 1. And the accordingly transformed code to allow vectorization is shown in Listing 2. For Lattice-Boltzmann the number of intermediate scalar values is usually in the range of 20 values. There explicit declaration is a little more cumbersome than employing arrays, but all in all this is a rather minor code transformation. Finally we add a hint to the compiler in Listing 2 that the outer loop iterations are indeed independent via the NODEP compiler directive. The independence is obscured by the use of indirection to address the lattices.

With the changed loop from 2, we already achieve a good vectorization and high sustained performance. For the standard collision operator, called BGK, the

Listing 2: Vectorizable loop by using scalar temporary variables

```
!CDIR NODEP
do iVal=1,nVals
  tmp1 = a(iVal)
  tmp2 = b(iVal)

  result = tmp1 + tmp2
end do
```

vectorization ratio reached 99.83 % with an average vector length of 256, which results in a performance of 19.87 GFLOPs on a single SX-ACE core.

For Musubi this change in code is already sufficient to achieve a satisfactory sustained performance in the Kernel on a single core. Unfortunately, for real problems we also need to consider other parts of the code, as any one of them might pose a potential bottleneck, prohibiting the execution of simulations on the machine.

2.2 *Porting of the Initialization*

While the kernel ran fine with the above minimal code changes, we hit a wall in the initialization for larger problems. The initialization takes care of constructing neighborhood information and setting up the indirect addressing accordingly. To achieve this a dynamic data structure is used, which allows for the addition of new elements with fast access afterwards. These are called growing arrays in Musubi and make use of amortized allocation costs by doubling the memory when the array is full.

This datatype is illustrated in Listing 3. The shown code makes use of CoCo preprocessing to define this growing array for arbitrary data types. Each array is accompanied by a counter `nVals` to track the actual number of entries in the possibly larger array. New entries are appended at the end of the list and the counter is increased accordingly. However, if the current size of the array is reached, the array is copied into a larger array to automatically allow for the addition of the new element. This is shown done by the shown `expand` routine.

For some data a dynamically growing array is not sufficient, instead it needs to be possible to search for values in the given array. To achieve this, we employ a very similar data structure, but with the addition of a ranking array to maintain a sorting of array entries and allow for binary searches. This kind of data structure has the additional complication, that for newly added values, we need to perform an insertion in the ranking array. Though the necessary copying to shift the entries is vectorizable here, the compiler needs again some hints to actually perform the vectorization.

Not too surprisingly this dynamic data structures yield only little performance on the NEC SX-ACE system, however the heavy costs getting prohibitive large for

Listing 3: Growing array to deal with dynamic data

```
! Class definition
type grw_?tname?Array_type
  integer :: nVals = 0
  integer :: ContainerSize = 0
  ?tstring?, allocatable :: Val(:)
end type

! Double the array
subroutine expand(me, pos)
  ...
  me%containerSize = me%containerSize*2
  if ( me%nVals > 0 ) then
    allocate(swpval(me%ContainerSize))
    swpval(:me%nVals) = me%Val(:me%nVals)
    call move_alloc( swpval, me%Val )
  else
    if ( allocated(me%Val) ) deallocate( me%Val )
    allocate( me%Val(me%containerSize) )
  end if
  ...
end subroutine expand

subroutine append(me, newval)
  ...
  if (me%nvals == me%containerSize) call expand(me)
  me%nVals = me%nVals + 1
  me%val(me%nVals) = newval
  ...
end subroutine append
```

larger problems were not expected. To overcome the long running times for the initialization the following strategies have been employed:

- Avoid many small allocations: Use an initial size for the growing arrays that is in the range of expected array entries.
- Minimize the utilization of these data structures: Out of convenience the data structures where used in places where some code reorganization allowed for single allocation of fixed sized arrays instead.
- Instead of employing the data structures for arbitrary complicated derived datatype, restrict there usage to arrays of intrinsic Fortran datatypes.
- Add a NODEP compiler directive to allow the compiler the vectorization in the shifting of the ranked array.

These changes indeed cut down the initialization times to a reasonable amount, and computations of large problems became feasible. The taken steps for the Kernel and the initialization so far are crucial for all simulations. They were apparent for the most simple simulation setups, where only minimal IO had to be performed. However, for meaningful simulations it usually is necessary to load a mesh that describes a more complicated geometry. Thus, after resolving the fundamental performance issues to this point, we are now able to move on to those more involved setups.

2.3 Porting of the IO

Most simulations require the loading of meshes to describe the geometrical setup of the computation and the boundary conditions. As it turned out, the loading of meshes in Musubi was awfully slow and took in serial several minutes for a small mesh file of 32 MB. For the reading of meshes Fortran direct IO was used, which in itself so far did never pose a problem. Some further investigation revealed that the old sxf90 compiler was around 400 times faster with the same reading task as the sxf03 compiler. The explanation for this can be found in the buffering mechanism for the IO. Due to the nature of the mesh data, each read only loads 8 Bytes of data. However, the system reads 4 MB at once. The sxf90 compiler recognizes the consecutive reads and reuses the loaded 4 MB, while the new sxf03 compiler seems to not recognize it and reads the 4 for every read, resulting in the huge observed overhead. This will probably be fixed in a later release of the compiler. However, the schedule for this is not fixed yet.

To overcome this issue we now make use of MPI-IO for the reading of all data. Most IO operations already made use of MPI-IO beforehand, but for the simple reading this was not considered necessary up to now. After implementing the mesh reading also via MPI-IO, the loading time for meshes was also cut down with the sxf03 compiler.

2.4 Porting of Boundary Conditions

Related to the usage of more complicated meshes is also the treatment of boundary conditions. Indeed, after resolving the issues in the initialization and loading of the mesh, the major performance hurdle was encountered in the boundary conditions. The boundary conditions use conditionals to decide what to do in a loop over boundary lattices. As this is only badly vectorizable an alternative implementation has been put into place. To allow for a vectorization also in the boundary treatment some additional memory is introduced to maintain lists of lattices with the same boundary condition. This enables the vectorized processing of each boundary condition and finally yields an implementation that is capable of running non-trivial simulation setups with a high sustained performance on the NEC SX-ACE system.

2.5 Parallel Performance

As shown, we were able to port Musubi to the NEC SX-ACE system with relatively little effort. Missing now is the parallel performance of Musubi on the system. The NEC SX-ACE provides 4 cores per node and we employ a MPI parallelization strategy to utilize the parallelism offered by the system. For the scaling analysis we

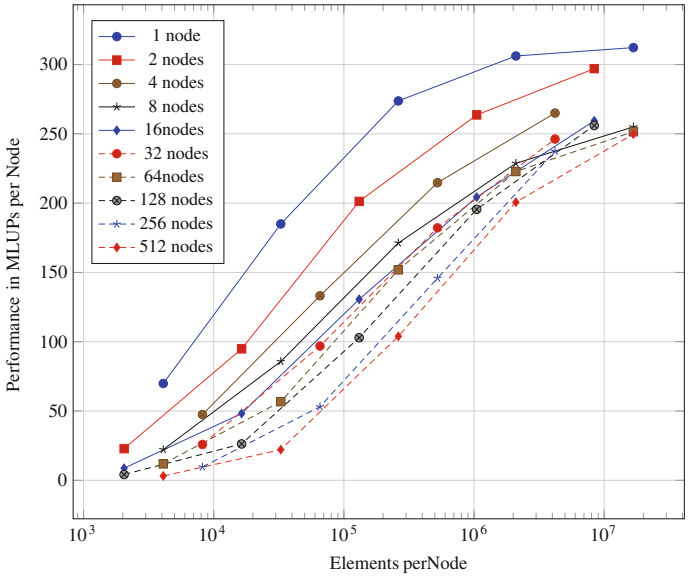

Fig. 1 Performance map of Musubi on the Tohoku NEC SX-ACE installation with up to 512 nodes (2048 MPI processes)

are using the machine of the Tohoku University in Sendai, where up to 512 nodes can be used in a parallel computation job.

To assess the parallel performance of Musubi, we use a performance map, which shows the performance per node over the problem size per node for various node counts. This graph is shown in Fig. 1. As a measure for the performance we use million lattice updates per second (MLUPs), which can also be translated into floating point operations per second. It shows a strong dependency of the performance on the problem size. For small problem sizes the performance gets diminished.

The performance map in Fig. 1 provides a full picture of the performance behavior of Musubi. We can extract the weak scaling for given problem sizes per node out of it by comparing the vertical distance between the lines for different node counts. And the strong scaling can be seen by starting on the right for a single node and moving to the left for larger and larger node counts. Thus, we recognize that weak scaling appears to rather poor, but still reasonable, while strong scaling always suffers from the strong dependency of the performance on the problem size.

Figure 2 shows the parallel efficiency when doing a weak scaling with 4 million elements per node. As can be seen the parallel efficiency drops immediately when the network is utilized (step from 1 node to 2 nodes). However, overall the drop in performance is moderate and on 512 nodes a parallel efficiency around 70 % is achieved.

A more concerning behavior is observed with respect to the performance in dependency on the problem size. Even on a single node with 4 MPI processes the

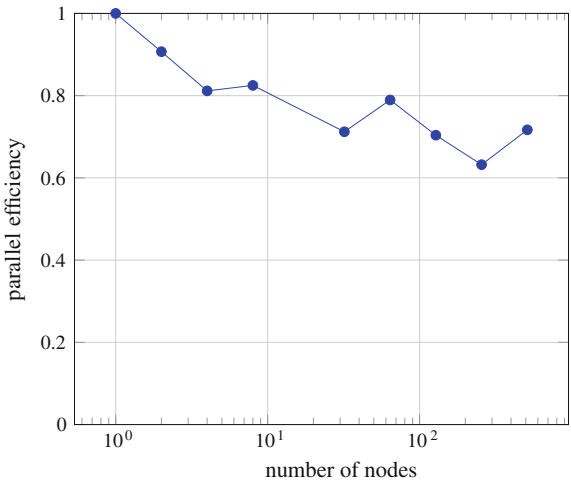

Fig. 2 Weak scaling efficiency for 4 million elements per node

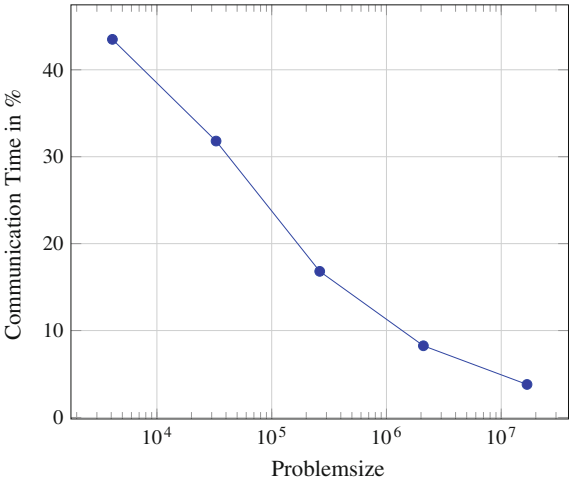

Fig. 3 Communication time in relation to the overall running time for increasing problem sizes on a single node (4 MPI processes)

performance drops drastically for small problem sizes. This is not observed for serial runs and we can trace this drop in performance back to the communication overhead. Figure 3 illustrates the communication overhead on a single node with 4 MPI processes. For the smallest problem of only 4096 elements we observe a communication time of around 44 % of the overall running time. This large fraction of time for the communication only slowly decreases for larger problems, which results in the bad parallel performance for a wide range of problem sizes per node.

This high communication cost is probably due to a flushing of the ADB, which becomes necessary for the MPI communication. For small problems the data from the ADB is relatively large and the costs for purging it in the communication calls get the dominating factor for the running time.

Unfortunately we are currently stuck with this situation as all attempts to overcome this performance bottleneck failed so far. One hope would be to avoid MPI communication by utilizing OpenMP parallelism within each node. However, the current sxf03 compiler seems to deactivate most vectorization as soon as OpenMP is activated, resulting in a poor performance in comparison to the MPI-only implementation even with the shown communication overheads. Another hope to speed the communication up was the utilization of global memory, which is available via the `MPI_alloc_mem` call of MPI. Incorporating this special memory did also not yield any benefit.

Despite the relatively poor scaling behavior due to these issues, the execution on the NEC SX-ACE compares quite well with more common large scale high performance computing systems because of the high sustained performance of nearly 30 % of the peak performance on a single node. This is illustrated in the comparison Fig. 4 by showing the sustained performance over the theoretical peak performance of utilized machine fractions in a strong scaling setup for more than 16 million elements. The comparison was done on the german systems:

- *Kabuki*: a small installation of NEC SX-ACE at the HLRS Stuttgart.
- *Hornet*: a Cray XC40 system with Intel Xeon E5-2680 v3 processors at the HLRS.
- *SuperMUC*: a Lenovo NeXtScale nx360M5 WCT system with Intel Xeon E5-2697 v3 processors located at the LRZ in Munich.
- *Juqueen*: a IBM BlueGene/Q system at the FZJ in Jülich.

As can be seen in Fig. 4, the scaling is on the NEC SX-ACE system Kabuki not as good as on the other many-core systems. However, due to the higher sustained

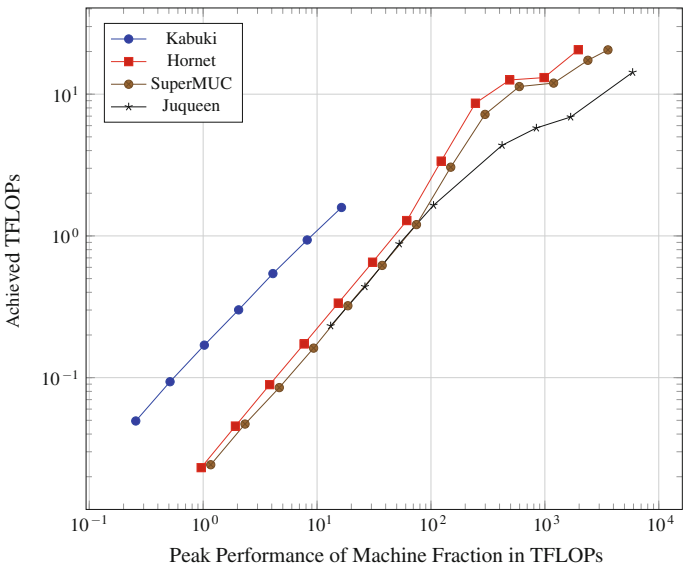

Fig. 4 Comparison of achieved performance by Musubi on different machines

performance the other systems need to utilize a much larger peak performance to obtain the same performance. Thus, even with the observed bad scaling at the moment, the NEC SX-ACE system appears to be an attractive execution option for a wide range of simulation setups. Especially, when considering the power consumption for a given simulation with Musubi, the NEC SX-ACE system shows a large advantage over the other high performance computing systems.

3 Some Notes on the Porting of Ateles

In contrast to Musubi, Ateles is not so straight forward to port to the vector system. The computational load is spread across multiple kernels and the data organization is more involved. Nevertheless, we hope to utilize the NEC SX-ACE system also with this solver and its porting is ongoing work. Though the work is not yet complete we want to share some first experiences with this Discontinuous Galerkin solver on the NEC SX-ACE.

Ateles implements a high order Discontinuous Galerkin scheme with many degrees of freedom per element. While the elements are unstructured like in Musubi, the internal organization of degrees of freedom in each element is highly regular. Our hope, therefore, is to allow a vectorized processing of the data within elements.

As a large part of the code is shared with Musubi, the initial porting without optimizations did not yield big surprises, and a first measurement could be done for a discretization of 128th order. This revealed multiple routines with significant contributions to the compute time. The three most expensive routines used respectively 32, 22 and 13 % of the compute time. For this high order, a good average vector length of 254 was achieved. However, the vector operation ratio only reached 11 % in the most expensive routine. Upon investigation, we found that a major problem in the code are again smaller loops that are put inside longer loops. Mostly these issues can be overcome by loop exchanges.

At first our attempts to remedy these problems with short loops did not yield the benefits, we would hope for. For a much smaller problem with only 16th order in the discretization scheme, the vectorization ratio only increased from 14.5 to 38 % with an average vector length of 140. A surprising discovery here was that vectorization seemed to be limited because of the size of source file. Splitting the Fortran module and using a smaller source file yielded a vectorization ratio of 99 %. This appears to be a compiler shortcoming and hopefully will be fixed in future revisions of the new compiler.

A compiler problem also seems to prohibit further vectorization of the next most important routine. This routine contains a loop, which is only partially vectorized in the new compiler, but in the old sxf90 compiler this loop gets fully vectorized. Thus, the performance porting of Ateles on the NEC SX-ACE seems to be more demanding for the vectorizing compiler and for further improvements we are looking forward to new compiler releases.

4 Summary

The porting effort of the APES solvers to the NEC SX-ACE proofed to be surprisingly smooth. For the Lattice-Boltzmann implementation in Musubi a high sustained performance of around 30 % on a single node was achieved. It has been shown that for real simulations not only the kernel needs to perform well, but also bottlenecks in the supporting infrastructure need to be overcome to allow actual simulations with complex setup. We have explained which roadblocks we encountered for the complete porting of Musubi and how they have overcome. The parallel performance has been investigated and a problem with the MPI communication was uncovered. This scalability issue remains the major shortcoming of the NEC SX-ACE for Musubi. However, despite the limited scalability, the system offers an attractive alternative to other systems due to the high sustained performance.

For the Discontinuous Galerkin solver only very first experiences could be shared, as the performance optimization got stuck early on due to a shortcoming of the current compiler version. Work on further improvements for Ateles are ongoing and we hope to be able to utilize the vectorization for the high order discretization with future compiler versions.

Acknowledgements We would like to thank Holger Berger from NEC for his ongoing kind support. This work would not have been possible without the possibility to use the NEC SX-ACE system at the Cyberscience Center, Tohoku University in Sendai and we are deeply grateful to Ryusuke Egawa to provide us with access to the system and Kazuhiko Komatsu for his support in running the jobs on the system. We also thank Uwe Küster for insightful discussions and the possibility to make use of the SX-ACE testsystem Kabuki at the HLRS in Stuttgart. Finally, we thank the Gauss-Center for supercomputing for providing the computing resources on Hornet, SuperMUC and Juqueen.

References

1. Roller, S., et al.: An adaptable simulation framework based on a linearized octree. In: Resch, M., Wang, X., Bez, W., Focht, E., Kobayashi, H., Roller, S. (eds.) High Performance Computing on Vector Systems 2011. Springer, Heidelberg (2011)
2. Hasert, M., et al.: Complex fluid simulations with the parallel tree-based lattice Boltzmann solver Musubi. J. Comp. Sci. **5**, 784–794 (2014)
3. Zudrop J., et al.: A fully distributed CFD framework for massively parallel systems. In: Cray User Group 2012. Stuttgart (2012)
4. Klimach, H., et al.: Distributed octree mesh infrastructure for flow simulations. In: Eberhardsteiner, J. (ed.) Eccomas 2012 - European Congress on Computational Methods in Applied Sciences and Engineering, e-Book Full Papers. Vienna (2012)