Numerical Python for Scalable Architectures

Kristensen, Mads Ruben Burgdorff; Vinter, Brian

Published in:
PGAS '10 Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model

DOI:
10.1145/2020373.2020388

Publication date:
2010

Document version
Early version, also known as pre-print

Numerical Python for Scalable Architectures

Mads Ruben Burgdorff Kristensen
Brian Vinter

eScience Centre
University of Copenhagen
Denmark
madsbk@diku.dk/vinter@diku.dk

Abstract

In this paper, we introduce DistNumPy, a library for doing numerical computation in Python that targets scalable distributed memory architectures. DistNumPy extends the NumPy module[15], which is popular for scientific programming. Replacing NumPy with DistNumPy enables the user to write sequential Python programs that seamlessly utilize distributed memory architectures. This feature is obtained by introducing a new backend for NumPy arrays, which distribute data amongst the nodes in a distributed memory multiprocessor. All operations on this new array will seek to utilize all available processors. The array itself is distributed between multiple processors in order to support larger arrays than a single node can hold in memory.

We perform three experiments of sequential Python programs running on an Ethernet based cluster of SMP-nodes with a total of 64 CPU-cores. The results show an 88% CPU utilization when running a Monte Carlo simulation, 63% CPU utilization on an N-body simulation and a more modest 50% on a Jacobi solver. The primary limitation in CPU utilization is identified as SMP limitations and not the distribution aspect. Based on the experiments we find that it is possible to obtain significant speedup from using our new array-backend without changing the original Python code.

Keywords NumPy, Productivity, Parallel language

1. Introduction

In many scientific and engineering areas, there is a need to solve numerical problems. Researchers and engineers behind these applications often prefer a high level programming language to implement new algorithms. Of particular interest are languages that support a broad range of high-level operations directly on vectors and matrices. Also of interest is the possibility to get immediate feedback when experimenting with an application. The programming language Python combined with the numerical library NumPy[15] supports all these features and has become a popular numerical framework amongst researchers.

The idea in NumPy is to provide a numerical extension to the Python language. NumPy provides not only an API to standardized numerical solvers, but a possibility to develop new numerical solvers that are both implemented and efficiently executed in Python, much like the idea behind the MATLAB[8] framework.

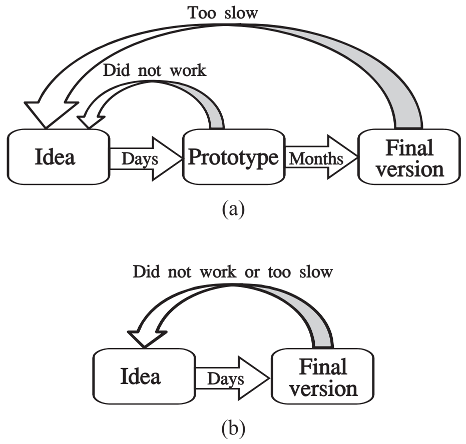


Figure 1. Development workflow. (a) is a typical workflow that involves two languages: one for the prototype and one for the final version. In (b) only one language is used in the workflow.

NumPy is mostly implemented in C and introduces a flexible N-dimensional array object that supports a broad range of numerical operations. The performance of NumPy is significantly increased when using array-operations instead of scalar-operations on this new array.

Parallel execution is supported by a limited set of NumPy functions, but only in a shared memory environment. However, many scientific computations are executed on large distributed memory machines because of the computation and memory requirements of the applications. In such cases, the communication between processors has to be implemented by the programmer explicitly. The result is a significant difference between the sequential program and the parallelized program. DistNumPy eliminates this difference by introducing a distributed version of the N-dimensional array object. All operations on such distributed arrays will utilize all available processors and the array itself is distributed between multiple processors, which makes it possible to expand the size of the array to the aggregated available memory.

1.1 Motivation

Solutions to numerical problems often consist of two implementations: a prototype and a final version. The algorithm is developed and implemented in a prototype by which the correctness of the algorithm can be verified. Typical many iterations of development are required to obtain a correct prototype, thus for this purpose a high productivity language is used, most often MATLAB. However, when the correct algorithm is finished the performance of the implementation becomes essential for doing research with the algorithm. This performance requirement presents a problem for the researcher since highly optimized code requires a fairly low-level programming language such as C/C++ or Fortran. The final version will therefore typical be a reimplementation of the prototype, which involves both changing the programming language and parallelizing the implementation (Fig. 1a).

The overall target of DistNumPy is to provide a high productivity tool that meets both the need for a high productivity tool that allows researcher to move from idea to prototype in a short time, and the need for a high performance solution that will eliminate the need for a costly and risky reimplementation (Fig. 1b). It should be possible to develop and implement an algorithm using a simple notebook and then effortlessly execute the implementation on a cluster of computers while utilizing all available CPUs.

1.2 Target architectures

NumPy supports a long range of architectures from the widespread x86 to the specialized Blue Gene architecture. However, NumPy is incapable of utilizing distributed memory architectures like Blue Gene supercomputers or clusters of x86 machines. The target of DistNumPy is to close this gap and fully support and utilize distributed memory architectures.

1.3 Related work

Libraries and programming languages that support parallelization on distributed memory architectures is a well known concept. The existing tools either seek to provide optimal performance in parallel applications or, like DistNumPy, seek to ease the task of writing parallel applications.

The library ScaLAPACK[2] is a parallel version of the linear algebra library LAPACK[1]. It introduces efficient parallel operations on distributed matrices and vectors. To use ScaLAPACK, an application must be programmed using MPI[7] and it is the responsibility of the programmer to ensure that the allocation of matrices and vectors comply with the distribution layout ScaLAPACK specifies.

Another library, Global Arrays[13], introduces a distributed data object (global array), which makes the data distribution transparent to the user. It also supports efficient parallel operations and provides a higher level of abstraction than ScaLAPACK. However, the programmer must still explicitly coordinate the multiple processes that are involved in the computation. The programmer must specify which region of a global array is relevant for a given process.

Both ScaLAPACK and Global Arrays may be used from within Python and can even be used in combination with NumPy, but it is only possible to use NumPy locally and not with distributed operations. A more closely integrated Python project IPython[16] supports parallelized NumPy operations. IPython introduces a distributed NumPy array much like the distributed array that is introduced in this paper. Still, the user-application must use the MPI framework and the user has to differentiate between the running MPI-processes.

Co-Array Fortran[14] is a small language extension of Fortran-95 for parallel processing on Distributed Memory Machines. It introduce a Partitioned Global Address Space (PGAS) by extending Fortran arrays with a *co-array* dimension. Each process can access remote instances of an array by indexing into the co-array dimensions. A similar PGAS extension called Unified Parallel C (UPC)[3] extent the C language with a distributed array declaration. Both languages provide a high abstraction level, but users still program with the SPMD model in mind, writing code with the understanding that multiple instances of it will be executing cooperatively.

A higher level of abstraction is found in projects where the execution, seen from the perspective of the user, is represented as a sequential algorithm. The High Performance Fortran (HPF)[12] programming languages provide such an abstraction level. However, HPF requires the user to specify parallelizable regions in the code and which data distribution scheme the runtime should use.

The Simple Parallel R INTerface (SPRINT)[9] is a parallel framework for the programming language R. The abstraction level in SPRINT is similar to DistNumPy in the sense that the distribution and parallelization is completely transparent to the user.

2. NumPy

Python has become a popular language for high performance computing even though the performance of Python programs is much lower than that of compiled languages. The growing popularity is because Python is used as the coordinating language while the compute intensive tasks are implemented in a high performance language.

NumPy[15] is a library for numerical operations in Python which is implemented in the C programming language. NumPy provides the programmer with an N-dimensional array object and a whole range of supported array operations. By using the array operations, NumPy takes advantage of the performance of C while retaining the high abstraction level of Python. However, this also means that no performance improvement is obtained otherwise e.g. using a Python loop to traverse a NumPy array does not result in any performance gain.

2.1 Interfaces

The primary interface in NumPy is a Python interface and it is possible to use NumPy exclusively from Python. NumPy also provides a C interface in which it is possible to access the same functionality as in the Python interface. Additionally, the C interface also allows programmers to access low level data structures like pointers to array data and thereby provides the possibility to implement arbitrary array operations efficiently in C. The two interfaces may be used interchangeably through the Python program.

2.2 Universal functions

An important mechanism in NumPy is a concept called Universal function. A universal function (ufunc) is a function that operates on all elements in an array independently. That is, a ufunc is a vectorized wrapper for a function that takes a fixed number of scalar inputs and produces a fixed number of scalar outputs. Using ufunc can result in a significant performance boost compared to native Python because the computation-loop is implemented in C.

2.2.1 Function broadcasting

To make ufunc more flexible it is possible to use arrays with different number of dimensions. To utilize this feature the size of the dimensions must either be identical or have the length one. When the ufunc is applied, all dimensions with a size of one will be *broadcasted* in the NumPy terminology. That is, the array will be duplicated along the *broadcasted* dimension (Fig. 2).

It is possible to implement many array operations efficiently in Python by combining NumPy's ufunc with more traditional numerical functions like matrix multiplication, factorization etc.

2.3 Basic Linear Algebra Subprograms

NumPy makes use of the numerical library Basic Linear Algebra Subprograms (BLAS) [11]. A highly optimized BLAS implementation exists for almost all HPC platforms and NumPy exploits

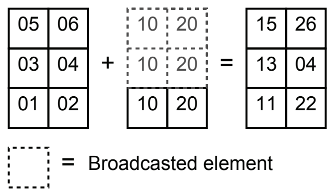


Figure 2. Universal function broadcasting. The ufunc `addition` is applied on a 3x2 array and a 1x2 array. The first dimension of the 1x2 array is broadcasted to the size of the first dimension of the 3x2 array. The result is a 3x2 array in which the two arrays are added together in an element-by-element fashion.

this when possible. Operations on vector-vector, matrix-vector and matrix-matrix (BLAS level 1, 2 and 3 respectively) all use BLAS in NumPy.

3. DistNumPy

DistNumPy is a new version of NumPy 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, we have chosen not to follow the standard MPI approach in which the same user-program is executed on all MPI-processes. This is because the standard MPI approach requires the user to differentiate between the MPI-processes, e.g. sequential areas in the user-program must be guarded with a branch based on the MPI-rank of the process. In DistNumPy MPI communication must be fully transparent and the user needs no knowledge of MPI or any parallel programming model. However, the user is required to use the array operations in DistNumPy to obtain any kind of speedup. We think this is a reasonable requirement since it is also required by NumPy.

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)
```

3.1 Interfaces

There are two programming interfaces in NumPy – one in Python and one in C. We aim to support the complete Python interface and a great subset of the C interface. However, the part of the C interface that involves direct access to low level data structures will not be supported. It is not feasible to return a C-pointer that represents the elements in a distributed array.

3.2 Data layout

Two-Dimensional Block Cyclic Distribution is a very popular distribution scheme and it is used in numerical libraries like 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 MPI-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.

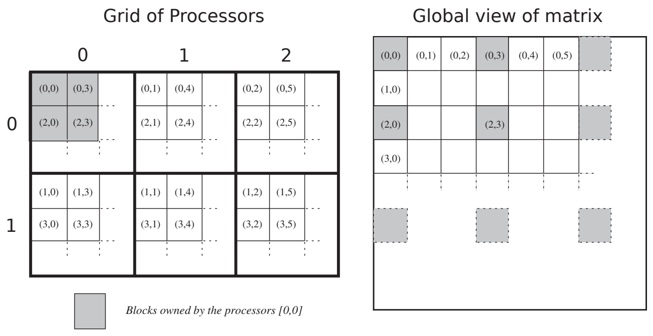


Figure 3. The Two-Dimensional Block Cyclic Distribution of a matrix on a 2 x 3 grid of processors.

NumPy is not limited to matrices and vectors as it supports arrays with an arbitrary number of dimensions. DistNumPy therefore use a more generalized N-Dimensional Block Cyclic Distribution inspired by High Performance Fortran[12], which supports an arbitrary number of dimensions. Instead of using a fixed process grid, we have a process grid for every number of dimensions. This works well when operating on arrays with the same number of dimensions but causes problems otherwise. For instance in a matrix-vector multiplication the two arrays are distributed on different process grid and may therefore require more communication. ScaLAPACK solves the problem by distributing vectors on two-dimensional process grids instead of one-dimensional process grids, but this will result in vector operations that cannot utilize all available processors. An alternative solution is to redistribute the data between a series of identically leveled BLAS operations using a fast runtime redistribution algorithm like [18] demonstrates.

3.3 Operation dispatching

The MPI-process hierarchy in DistNumPy has one MPI-process (master) placed above the others (slaves). All MPI-processes run the Python interpreter but only the master executes the user-program, the slaves will block at the `import numpy` statement.

The following describes the flow of the dispatching:

1. The master is the dispatcher and will, when the user applies a python command on a distributed array, compose a message with meta-data describing the command.
2. The message is then broadcasted from the master to the slaves with a blocking MPI-broadcast. It is important to note that the message only contains meta-data and not any actual array data.
3. After the broadcast, all MPI-processes will apply the command on the sub-array they own and exchange array elements as required (Point-to-Point communication).
4. When the command is completed, the slaves will wait for the next command from the master and the master will return to the user's python program. The master will return even though some slaves may still be working on the command, synchronization is therefore required before the next command broadcast.

3.4 Views

In NumPy an array does not necessarily represent a complete contiguous block of memory. An array is allowed to represent a subpart of another array i.e. it is possible to have a hierarchy of arrays where only one array represent a complete contiguous block of memory and the other arrays represent a subpart of that memory.

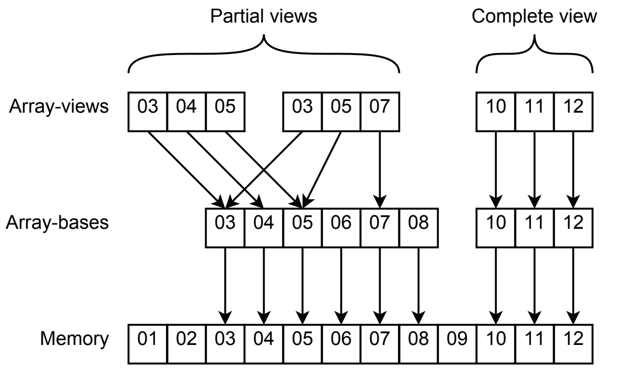


Figure 4. Reference hierarchy between the two array data structures and the main memory. Only the three array-views at top of the hierarchy are visible from the perspective of the user.

Inspired by NumPy, DistNumPy implements an array hierarchy where distributed arrays are represented by the following two data structures.

- **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 and it can add non-existing 1-length dimensions to the array-base. The array-view is manipulated directly by the user and from the users perspective the array-view is the array.

Array-views are not allowed to refer to each other, which means 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.

3.5 Optimization hierarchy

It is a significant performance challenge to support array-views that are not aligned with the distribution block size, i.e. an array view that has a starting offset that a not aligned with the distribution block size or represents a non-contiguous sub-part of the array-base. The difficulty lies in how to handle data blocks that are located on multiple MPI-processes and are not aligned to each other. Such problems can be handled by partitioning data blocks into sub-blocks that both are aligned and located on a single MPI-process. However, in this work we will not focus on problems that involve non-aligned array-views, but instead simply handle them by communicating and computing each array element individually.

In general we introduce a hierarchy of implementations where each implementation is optimized for specific operation scenarios. When an operation is applied a lookup in the hierarchy determines the best suited implementation for that particular operation. All operations have their own hierarchy some with more levels than others, but at the bottom of the hierarchy all operations have an implementation that can handle any scenario simply by handling each array element individually.

```
from numpy import *
(x, y) = (empty([S], dist=True), \
          empty([S], dist=True))
(x, y) = (random(x), random(y))
(x, y) = (square(x), square(y))
z = (x + y) < 1
print add.reduce(z) * 4.0 / S #The result
```

Figure 5. Computing Pi using Monte Carlo simulation. `S` is the number of samples used. We have defined a new ufunc (`ufunc_random`) to make sure that we use an identical random number generator in all benchmarks. The ufunc uses ”`rand()/(double)RAND_MAX`” from the ANSI C standard library (`stdlib.h`) to generate numbers.

3.6 Parallel BLAS

As previously mentioned NumPy supports BLAS operations on vectors and matrices. DistNumPy therefore implements a parallel version of BLAS inspired by PBLAS from the ScaLAPACK library. Since DistNumPy uses the same data-layout as ScaLAPACK, it would be straightforward to use PBLAS for all parallel BLAS operations. However, to simplify the installation and maintenance of DistNumPy we have chosen to implement our own parallel version of BLAS. We use SUMMA[6] for matrix multiplication, which enable us to use the already available BLAS library locally on the MPI-processes. SUMMA is only applicable on complete array-views and we therefore use a straightforward implementation that computes one element at a time if partial array-views are involved in the computation.

3.7 Universal function

In DistNumPy, the implementation of ufunc uses three different scenarios.

1. In the simplest scenario we have a perfect match between all elements in the array-views and applying an ufunc does not require any communication between MPI-processes. The scenario is applicable when the ufunc is applied on complete array-views with identical shapes.
2. In the second scenario the array-views must represent a continuous part of the underlying array-base. The computation is parallelized by the data distribution of the output array and data blocks from the input arrays are fetched when needed. We use non-blocking one-side communication (`MPI_Get`) when fetching data blocks, which makes it possible to compute one block while fetching the next block (double buffering).
3. The final scenario does not use any simplifications and works with any kind of array-view. It also uses non-blocking one-side communication but only one element at a time.

4. Examples

To evaluate DistNumPy we have implemented three Python programs that all make use of NumPy's vector-operations (ufunc). They are all optimized for a sequential execution on a single CPU and the only program change we make, when going from the original NumPy to our DistNumPy, is the array creation argument `dist`. A walkthrough of a Monte Carlo simulation is presented as an example of how DistNumPy handles Python executions.

4.1 Monte Carlo simulation

We have implemented an efficient Monte Carlo Pi simulation using NumPy's ufunc. The implementation is a translation of the Monte Carlo simulation included in the benchmark suite SciMark 2.0[17],

```
1 h = zeros(shape(B), float, dist=True)
2 dmax = 1.0
3 AD = A.diagonal()
4 while(dmax > tol):
5   hnew = h + (B - add.reduce(A * h, 1)) /
        AD
6   tmp = absolute((h - hnew) / h)
7   dmax = maximum.reduce(tmp)
8   h = hnew
9 print h #The result
```

Figure 6. Iteratively Jacobi solver for matrix `A` with solution vector `B` both are distributed arrays. The `import` statement and the creation of `A` and `B` is not included here. `tol` is the maximum tolerated value of the diagonal-element with the highest value (`dmax`).

which is written in Java. It is very simple and uses two vectors with length equal the number of samples used in the calculation. Because of the memory requirements, this drastically reduces the maximum number of samples. Combining multiple simulations will allow more samples but we will only use one simulation. The implementation is included in its full length (Fig. 5) and the following is a walkthrough of a simulation (the bullet-numbers represents line numbers):

1: All MPI-processes interpret the `import` statement and initiate DistNumPy. Besides calling `MPI_Init()` the initialization is identical to the original NumPy but instead of returning from the import statement, the slaves, MPI-processes with rank greater than zero, listen for a command message from the master, the MPI-process with rank zero.

2-3: The master sends two `CREATE_ARRAY` messages to all slaves. The two messages contain an array shape and unique identifier (UID), which in this case identifies `x` and `y`, respectively. All MPI-processes allocate memory for the arrays and stores the array information.

4: The master sends two `UFUNC` messages to all slaves. Each message contains a UID and a function name `ufunc_random`. All MPI-processes apply the function on the array with the specified UID. A pointer to the function is found by calling `PyObject_Get AttrString` with the function name. It is thereby possible to support all ufuncs from NumPy.

5: Again the master sends two `UFUNC` messages to all slaves but this time with function name `square`.

6: The master sends a `UFUNC` messages with function name `add` followed by a `UFUNC` messages with function name `less_than`. The scalar `1` is also in the message.

7: The master sends a `UFUNC_REDUCE` messages with function name `add`. The result is a scalar, which is not distributed, and the master therefore solely computes the remainder of the computation and print the result. When the master is done a `SHUTDOWN` message is sent to the slaves and the slaves call `exit(0)`.

4.2 Jacobi method

The Jacobi method is an algorithm for determining the solutions of a system of linear equations. It is an iterative method that uses a spitting scheme to approximate the result.

Our implementation uses ufunc operations in a while-loop until it converges. Most of the implementation is included here(Fig. 6).

Table 1. Hardware specifications

CPU	Core 2 Quad	Nehalem
CPU Frequency	2.26 GHz	2.66 GHz
CPU per node	1	2
Cores per CPU	4	4
Memory per node	8 GB @ 6.5 GB/s	24 GB @ 25.6 GB/s
Number of nodes	8	8
Network	Gigabit Ethernet	Gigabit Ethernet

4.3 Newtonian N-body simulation

A Newtonian N-body simulation is one that studies how bodies, represented by a mass, a location, and a velocity, move in space according to the laws of Newtonian physics. We use a straightforward algorithm computing all body-body interactions. The NumPy implementation is a direct translation of a MATLAB program[4]. The working loop of the two implementations take up 19 lines in Python and 22 lines in MATLAB thus it is too big to include here. However, the implementation is straightforward and use universal functions and matrix multiplications.

5. Experiments

In this section, we will conduct performance benchmarks on DistNumPy and NumPy[1]. We will benchmark the three Python programs presented in Section 4. All benchmarks are executed on two different Linux clusters – an Intel Core 2 Quad cluster and an Intel Nehalem cluster. Both clusters consist of processors with four CPU-cores, but the number of processors per node differs. Intel Core 2 Quad cluster has one CPU per node whereas the Intel Nehalem cluster has two CPUs per node. The interconnect is Gigabit Ethernet in both clusters. (Table 1).

Our experiments consist of a speedup benchmark, which we define as an execution time comparison between a sequential execution with NumPy and a parallelized execution with DistNumPy while the input is identical. Strong-scaling is used in all benchmarks and the input size is therefore constant.

5.1 Monte Carlo simulation

A Distributed Monte Carlo simulation is embarrassingly parallel and requires a minimum of communication. This is also the case when using DistNumPy because ufuncs are only applied on identically shaped arrays and it is therefore the simplest ufunc scenario. Additionally, the implementation is CPU-intensive because a complex ufunc is used as random number generator.

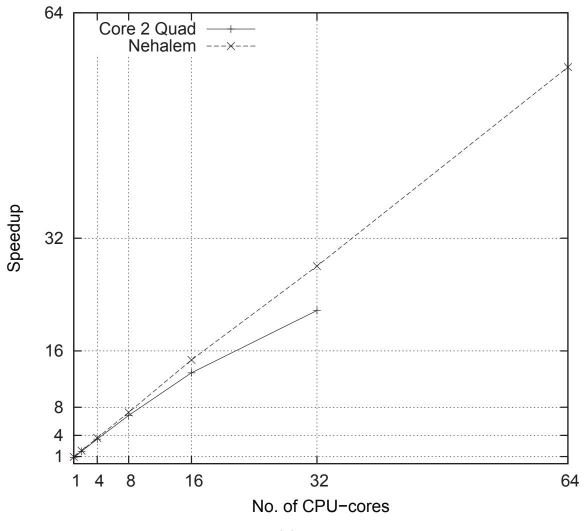
The result of the speedup benchmark is illustrated in Figure 7. We see a close to linear speedup for the Nehalem cluster – a CPU utilization of 88% is achieved on 64 CPU-cores. The penalty of using multiple CPU-cores per node is noticeable on the Core 2 architecture – a CPU utilization of 68% is achieved on 32 CPU-cores.

5.2 Jacobi method

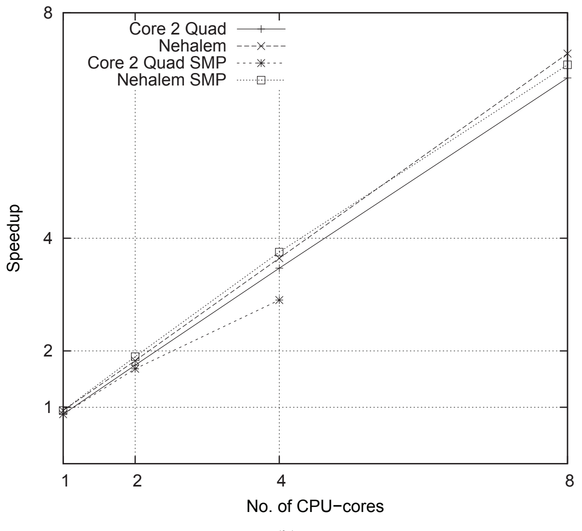
The dominating part of the Jacobi method, performance-wise, is the element-by-element multiplication of `A` and `h` (Fig. 6 line 5). It consists of $O(n^2)$ operations where as all the other operations only consist $O(n)$ operations. Since scalar-multiplication is a very simple operation, the dominating ufunc in the implementation is memory-intensive.

The result of the speedup benchmark is illustrated in Figure 8. We see a good speedup with 8 CPU-cores and to some degree also with 16 Nehalem CPU-cores. However, the CPU utilization when

[1] NumPy version 1.3.0



(a)



(b)

Figure 7. Speedup of the Monte Carlo simulation. In graph (a) the two architectures uses a minimum number of CPU-cores per node. Added in graph (b) is the result of using multiple CPU-cores on a single node (SMP).

using more than 16 CPU-cores is very poor. The problem is memory bandwidth – since we use multiple CPU-cores per node when using more than 8 CPU-cores, the aggregated memory bandwidth of the Core 2 cluster does only increase up to 8 CPU-cores. The Nehalem cluster is a bit better because it has two memory buses per node, but using more than 16 CPU-cores will not increase the aggregated memory bandwidth.

5.2.1 Profiling of the Jacobi implementation

To investigate the memory bandwidth limitation observed in the Jacobi execution we have profiled the execution by measuring the time spend on computation and communication (Fig. 9). As expected the result shows that the percentages used with communication increases when the number of CPU-cores increases. Furthermore, a noteworthy observation is the almost identical communication overhead at eight CPU-cores and sixteen CPU-cores. This is because half of the communication is performed through the use of shared memory at sixteen CPU-cores, which also means that the communication, just like the computation, is bound by the limited memory bandwidth.

5.3 Newtonian N-body simulation

The result of the speedup benchmark is illustrated in Figure 10. Compared to the Jacobi method we see a similar speedup and CPU utilization. This is expected because the dominating operations are also simple ufuncs. Even though there are some matrix-multiplications, which have a great scalability, it is not enough to significantly boost the overall scalability.

5.4 Alternative programming language

DistNumPy introduces a performance overhead compared to a lower-level programming language such as C/C++ or Fortran. To investigate this overhead we have implemented the Jacobi benchmark in C. The implementation uses the same sequential algorithm as the NumPy and DistNumPy implementations.

Executions on both architectures show that DistNumPy and NumPy is roughly 50% slower than the C implementation when executing the Jacobi method on one CPU-core. This is in rough runtime numbers: 21 seconds for C, 31 seconds for NumPy and 32 seconds for DistNumPy.

Obviously highly hand-optimized implementations have a clear performance advantages over DistNumPy. For instance by the use of a highly optimized implementation in C [10] demonstrates extreme scalability of a similar Jacobi computation – an execution by 16384 CPU-cores achieves a CPU utilization of 70% on a Blue Gene/P architecture.

5.5 Summary

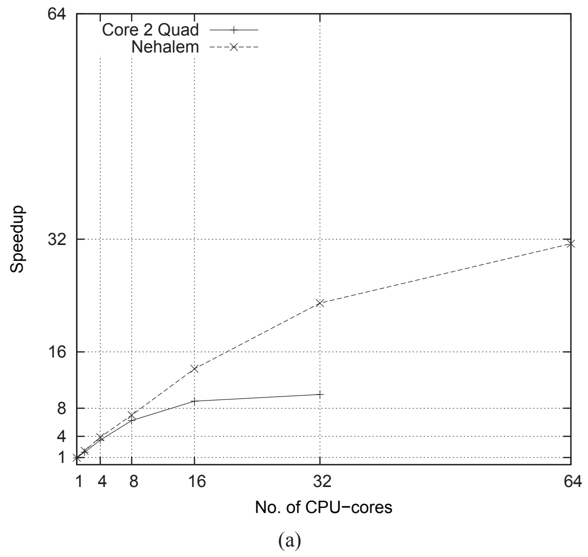
The benchmarks clearly show that DistNumPy has both good performance and scalability when execution is not bound by the memory bandwidth, which is evident from looking at the CPU utilization when only one CPU-core per node is used. As expected the scalability of the Monte Carlo simulation is better than the Jacobi and the N-body computation because of the reduced communication requirements and more CPU-intensive ufunc operation.

The scalability of the Jacobi and the N-body computation is drastically reduced when using multiple CPU-cores per node. The problem is the complexity of the ufunc operations. As opposed to the Monte Carlo simulation, which makes use of a complex ufunc, the Jacobi and the N-body computation only use simple ufuncs e.g. add and multiplication.

As expected the performance of the C implementation is better than the DistNumPy implementation. However, by utilizing two CPU-cores it is possible to outperform the C implementation in the case of the Jacobi method. This is not a possibility in the case of the Monte Carlo simulation where the algorithm does not favor vectorization.

6. Future work

In its current state DistNumPy does not implement the NumPy interface completely. Many specialized operations like Fast Fourier transform or LU factorization is not implemented, but it is our intention to implement the complete Python interface and most of the C interface.



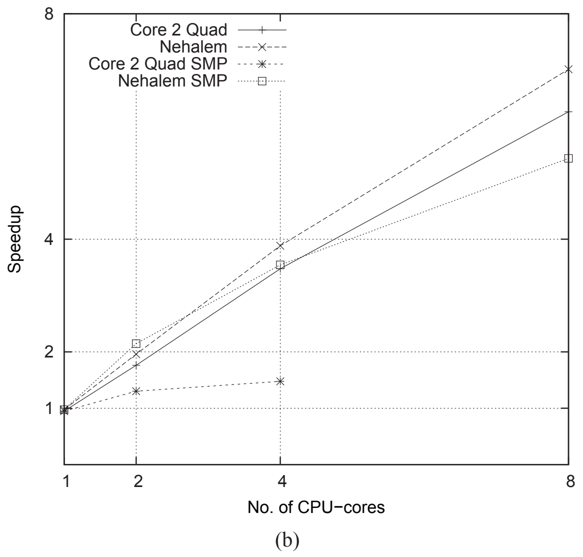
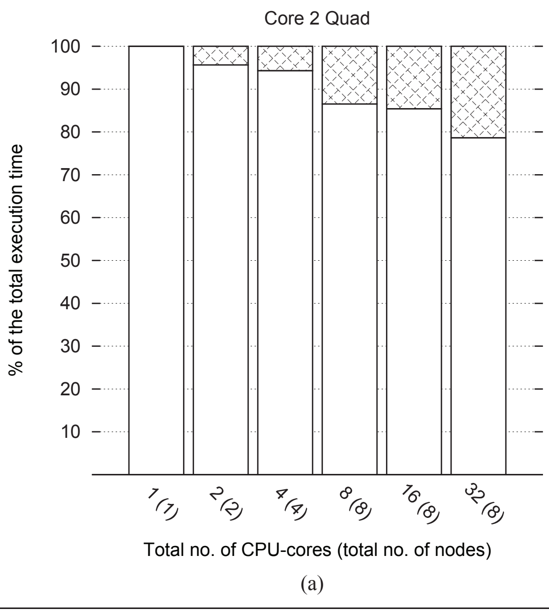


Figure 8. Speedup of the Jacobi solver. In graph (a) the two architectures uses a minimum number of CPU-cores per node. Added in graph (b) is the result of using multiple CPU-cores on a single node (SMP).



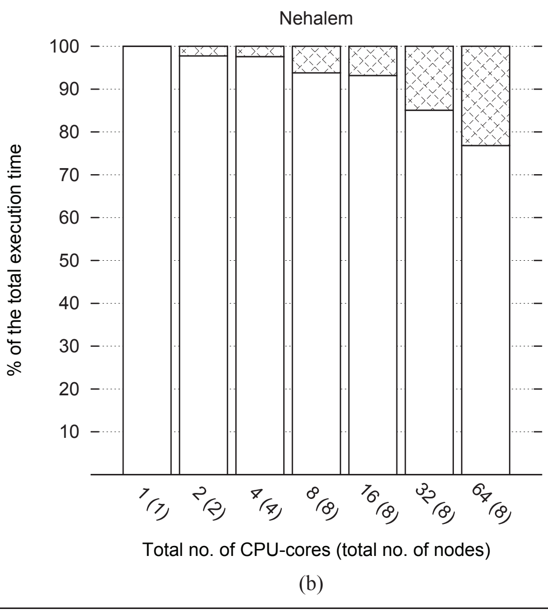
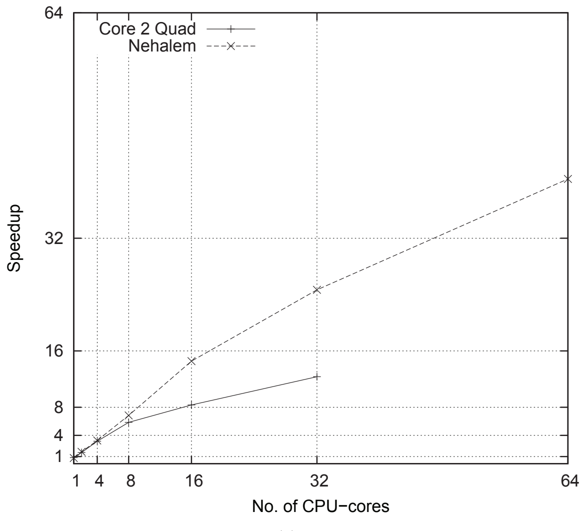
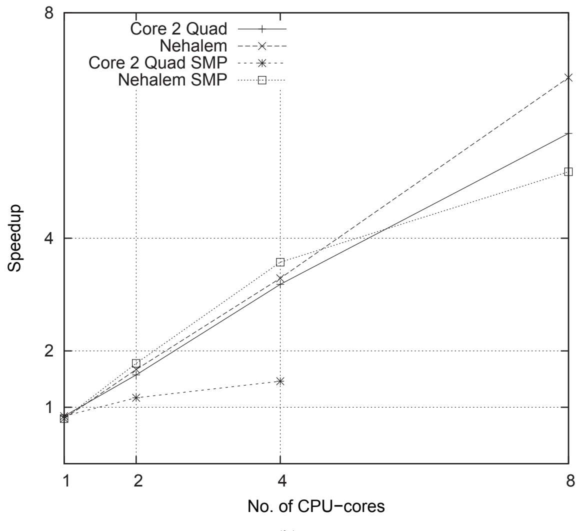


Figure 9. Profiling of the Jacobi experiment. The two figures illustrate the relationship between communication and computation when running on the Core 2 Quad architecture (a) and the Nehalem architecture (b). The area with the check pattern represent MPI communication and the clean area represent computation. Note that these figures relates directly to the Jacobi speedup graph (Fig 8a).



(a)



(b)

Figure 10. Speedup of Newtonian N-body simulation. In graph (a) the two architectures uses a minimum number of CPU-cores per node. Added in graph (b) is the result of using multiple CPU-cores on a single node (SMP).

The performance of NumPy programs that make use of array-views that are not aligned with the distribution block size is very poor because each array element is handled individually. This is not a problem for a whole range of NumPy programs, including the experiments presented in this paper, since they do not use non-aligned array-views. However some operations, such as stencil operations, require non-aligned array-views and an important future work is therefore to support all array views with similar efficiency.

Other important future work includes performance and scalability improvement. As showed by the benchmarks, applications that are dominated by non-complex ufuncs easily become memory bounded. One solutions is to merge calls to ufuncs, that operate on common arrays, together in one joint operation and thereby make the joint operation more CPU-intensive. If it is possible to merge enough ufuncs together the application may become CPU bound rather than memory bound.

7. Conclusions

In this work we have successfully shown that it is possible to implement a parallelized version of NumPy[15] that seamlessly utilize distributed memory architectures. The only API difference between NumPy and our parallelized version, DistNumPy, is an extra optional parameter in the array creation routines.

Performance measurements of three Python program, which make use of DistNumPy, show very good performance and scalability. A CPU utilization of 88% is achieved on a 64 CPU-core Nehalem cluster running a CPU-intensive Monte Carlo simulation. A more memory-intensive N-body simulation achieves a CPU utilization of 91% on 16 CPU-cores but only 63% on 64 CPU-cores. Similar a Jacobi solver achieves a CPU utilization of 85% on 16 CPU-cores and 50% on 64 CPU-cores.

To obtain good performance with NumPy the user is required to make use of array operations rather than using Python loops. DistNumPy take advantage of this fact and parallelizes array operations. Thus most efficient NumPy applications should be able to benefit from DistNumPy with the distribution parameter as the only change.

We conclude that it is possible to obtain significant speedup with DistNumPy. However, further work is needed if shared memory machines are to be fully utilized as nodes in a scalable architecture.

References

[1] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. Lapack: a portable linear algebra library for high-performance computers. In *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press. ISBN 0-89791-412-0.

[2] L. S. Blackford. Scalapack. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM) - Supercomputing 96 Supercomputing 96*, page 5, 1996. doi: 10.1145/369028.369038.

[3] W. W. Carlson, J. M. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to upc and language specification. Technical Report CCS-TR-99-157, Bowie, MD, May 1999.

[4] H. Casanova. N-body simulation assignment, Nov 2008. URL `http://navet.ics.hawaii.edu/~casanova/courses/ics632_fall08/projects.html`.

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

[6] R. A. v. d. Geijn and J. Watts. Summa: scalable universal matrix multiplication algorithm. *Concurrency - Practice and Experience*, 9 (4):255–274, 1997.

[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] J. Hill, M. Hambley, T. Forster, M. Mewissen, T. M. Sloan, F. Scharinger, A. Trew, and P. Ghazal. Sprint: a new parallel framework for r. *BMC Bioinformatics*, 9:558, 2008. doi: 10.1186/1471-2105-9-558.

[10] M. R. B. Kristensen, H. H. Happe, and B. Vinter. Gpaw optimized for blue gene/p using hybrid programming. *Parallel and Distributed Processing Symposium, International*, 2009. doi: 10.1109/IPDPS.2009.5160936.

[11] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979. doi: 10.1145/355841.355847.

[12] D. Loveman. High performance fortran. *IEEE Parallel & Distributed Technology: Systems & Applications*, 1(1):25, 1993. doi: 10.1109/88.219857.

[13] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10(2), 1996. doi: 10.1007/BF00130708.

[14] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998. ISSN 1061-7264. doi: 10.1145/289918.289920.

[15] T. E. Oliphant. Python for scientific computing. *Computing in Science and Engineering*, 9:10–20, 2007. ISSN 1521-9615. doi: 10.1109/MCSE.2007.58.

[16] F. Pérez and B. E. Granger. Ipython: a system for interactive scientific computing. *Comput. Sci. Eng.*, 9(3):21–29, may 2007.

[17] R. Pozo and B. Miller. Scimark 2.0, 12 2002. URL `http://math.nist.gov/scimark2/`.

[18] L. Prylli and B. Tourancheau. Fast runtime block cyclic data redistribution on multiprocessors. *J. Parallel Distrib. Comput*, 45(1):63–72, 1997.