SF2A 2012
S. Boissier, P. de Laverny, N. Nardetto, R. Samadi, D. Valls-Gabaud and H. Wozniak (eds)

PYOPERATORS: OPERATORS AND SOLVERS FOR HIGH-PERFORMANCE COMPUTING

P. Chanial[1] and N. Barbey[1]

Abstract.
PyOperators is a publicly available library that provides basic operators and solvers for small-to-very large inverse problems (`http://pchanial.github.com/pyoperators`). It forms the backbone of the package PySimulators, which implements specific operators to construct an instrument model and means to conveniently represent a map, a timeline or a time-dependent observation (`http://pchanial.github.com/pysimulators`). Both are part of the Tamasis (Tools for Advanced Map-making, Analysis and SImulations of Submillimeter surveys) toolbox, aiming at providing versatile, reliable, easy-to-use, and optimal map-making tools for Herschel and future generation of sub-mm instruments. The project is a collaboration between 4 institutes (ESO Garching, IAS Orsay, CEA Saclay, Univ. Leiden).

Keywords: Methods: numerical, Techniques: image processing

1 Introduction

The PyOperators and PySimulators packages provide linear and non-linear operators that can be thought as the building blocks of an instrument model H used to simulate data acquisition, and a variety of solvers to estimate the signal $\mathbf{x}$ given an observation $\mathbf{y}$ through this instrument, i.e. to solve the equation

$$\mathbf{y} = H\mathbf{x} + \mathbf{n}, \tag{1.1}$$

where $\mathbf{n}$ is the noise of covariance matrix N.

These operators can easy be combined to form block diagonal, block column and block row operators (section §3). The chaining of the operations relies on a memory manager (section §4) and has been optimised to minimise the number of memory allocations and the memory footprint, and to maximise the cache locality by relying on operator's properties such as whether or not it can perform in-place operations on its input or on its output (section §5). Solvers will be presented in section §6. The code is massively parallel and hybrid (MPI + OpenMP) (section §7). It is written in Fortran and C (for the number crunching) and Python (for the abstraction and interactivity). The Python overhead has been reduced to a mostly negligeable fraction.

2 Operators

The Operator class is a factory of multi-dimensional functions that, if linear, behave like matrices with a sparse storage footprint. Operator instances can be transposed, added, multiplied and composed in a natural way so that they can be used as the subsystems of a complex instrument acquisition model. Rules can be attached to operators through properties (such as symmetric, hermitian, involutary, orthogonal...) or through a flexible mechanism, to trigger algebraic simplifications, for additional speedups. For example, let us assume that H in Eq. 1.1 is a convolution and that the noise follows a uniform gaussian distribution $\mathbf{n} \sim \mathcal{N}(0, \sigma)$. The expression $H^{\mathrm{T}}N^{-1}H$, which is involved in the normal equation, is automatically reduced to a single convolution operator, whose Fourier-transformed kernel is the module of the initial Fourier-transformed kernel scaled by $1/\sigma^2$.

[1] Laboratoire AIM-Paris-Saclay, CEA/DSM/Irfu, CNRS, Université Paris Diderot, Saclay, 91191, Gif-sur-Yvette, France

- AdditionOperator, MultiplicationOperator, CompositionOperator
- BlockRowOperator, BlockColumnOperator, BlockDiagonalOperator
- BlockSliceOperator
- SumOperator, ProductOperator, MinOperator, MaxOperator (for reductions)
- FftOperator, ConvolutionOperator, ConvolutionTruncatedExponentialOperator
- WaveletOperator
- MPIDistributionIdentityOperator, MPIDistributionLocalOperator, MPIScatterOperator
- IdentityOperator, ZeroOperator, HomothetyOperator, ConstantOperator
- DiagonalOperator, MaskOperator
- TridiagonalOperator, BandOperator, SymmetricBandOperator
- EigendecompositionOperator (Lanczos algorithm)
- PackOperator, UnpackOperator
- RollOperator, ShiftOperator
- DiscreteDifferenceOperator
- IntegrationTrapezeWeightOperator
- NumexprOperator, DiagonalNumexprOperator, DiagonalNumexprNonSeparableOperator
- RoundOperator (6 methods)
- ClipOperator, MaximumOperator, MinimumOperator
- HardThresholdingOperator, SoftThresholdingOperator

Table 1. List of operators available in PyOperators 0.7. More operators related to data acquisition can be found in the package PySimulators.

3 Block operators

Operators can be combined to form block row, column and diagonal operators. By writing H from Eq. 1.1 as a block column matrix and N^{-1} as a block diagonal matrix:

$$H = \begin{pmatrix} H_1 \\ \vdots \\ H_n \end{pmatrix} \quad \text{and} \quad N^{-1} = \bigoplus_{i=1}^{n} N_i^{-1},$$

the product of Operators $H^{\mathrm{T}} N^{-1} H$ is automatically simplified to the expression

$$\sum_{i=1}^{n} H_i^{\mathrm{T}} N_i^{-1} H_i, \tag{3.1}$$

which handles arrays of much lesser size, better suited to take advantage of the memory cache. Currently, two schemes are supported: partitioning by stacking along a new array dimension or by chunk along an existing one.

4 Memory manager

High-level languages such as Python come with a garbage collector which operates behind the scenes with the consequence that in some cases (objects with cycling references, holding a reference to an array), temporaries may reside in memory for a period of time longer than necessary. Equally important, the sequence of freeing a buffer and soon after reallocating another one of the same size does not imply that the same memory will be used and it will result in cache line flushes. So, when manipulating very large arrays, one should be sparing about temporaries allocation, even more for supercomputers with relatively limited memory per node. This is the reason why the pyoperators package uses a lightweight memory manager designed to maximise the cache locality by reusing the memory buffers. This memory manager can serve buffers to the operators according to their specific requirements, such as contiguity or alignment (crucial for the FFTW library and other libraries that leverage on the SSE and AVX instruction sets). By relying on this memory manager, we ensure that the amount of memory used in iterative algorithms does not vary from one iteration to the next one.

5 Chaining of operations

5.1 Inplace operations

The *inplace* flag indicates that an operator can handle input and output pointing to the same memory location. This property is useful to avoid temporaries. In order to minimise the memory-cache transfers during a composition, an algorithm has been put in place to determine the intermediate variables to be extracted from the memory manager. This algorithm depends on:

- the parity of the number of out-of-place operators
- whether or not the input and the output of the composition point the same memory location (in-place or out-of-place composition)
- the size of the output, and if it is large enough to be reused in the intermediate computations.

As an example, let's consider the composition of an in-place operator `IN` by an out-of-place operator `OUT`. The out-of-place composition `(IN * OUT)(x, out=y)` requires an intermediate variable only if the size of `OUT`'s output is larger than that of variable `y`. Otherwise, `y`'s buffer is used as output for the `OUT` operator and the application of the operator `IN` is performed in-place on `y`. Concerning the in-place composition `(IN * OUT)(x, out=x)`, it is not possible to avoid the use of a temporary variable, since a buffer different from the `x` variable is required for the `OUT` operator. The application of the operator `IN` is performed out-of-place because its output has to be the `x` variable.

5.2 Inplace reductions

An operator that can do in-place reductions is an operator that can add, element-wise multiply (or else) its output to its output argument. This property can be used to further remove the need of intermediate variables. First, let's compute `(o1 + o2)(x, out=y)` assuming that the operator `o2` cannot do in-place reductions:

- the variable `y` is used as `o1`'s output,
- a temporary variable is retrieved from the memory manager and is used as `o2`'s output,
- that is then added to the `y` variable.

If we now assume that `o2` can do in-place reductions, the intermediate variable is not required anymore:

- the variable `y` is used as `o1`'s output
- `o2` updates `y` in-place.

The possibility of updating directly an operator's output also leads to a better use of the memory cache, since the temporary could otherwise flush it. Such operators are common in map-making applications where H_i^{T} from Eq. 3.1 are backprojections.

- Conjugate Gradient
- Preconditioned conjugate gradient
- Non-linear preconditioned conjugate gradient
- Variational bayesian double loop inference algorithm (Seeger et al. 2010)
- FISTA proximal gradient algorithm (Beck & Teboulle 2009)

Table 2. List of solvers available in PyOperators 0.7. Solvers from scipy.optimize and scipy.sparse.linalg can also seamlessly be used with Operators.

6 Solvers

PyOperators features solvers that can handle MPI-distributed unknowns (Table 2). These solvers are written using the flexible IterativeAlgorithm class, which in addition to managing buffer handling from one iteration to another, can be interrupted and restarted or continued in a clean state.

$$\begin{array}{r c}
 & \overbrace{\hphantom{1 \;\; \ddots \;\; 0}}^{\text{all processes}} \\
\text{process } 1 \rightarrow & \begin{pmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{pmatrix} \\
\text{process } 2 \rightarrow & \begin{pmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{pmatrix} \\
\vdots & \vdots \\
\text{process } n \rightarrow & \begin{pmatrix} 1 & & 0 \\ & \ddots & \\ 0 & & 1 \end{pmatrix}
\end{array}$$

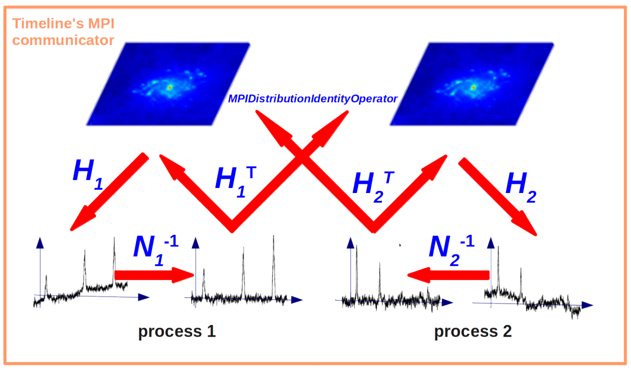

Fig. 1. Non-distributed unknown, distributed data. **Left:** Matrix representation of the operator D as MPIDistribution-IdentityOperator. **Right:** Map-making example showing the computation of $D^{\mathrm{T}}H^{\mathrm{T}}N^{-1}HD$ by two MPI processes.

$$\begin{array}{r cccc}
 & \text{process } 1 & \text{process } 2 & & \text{process } n \\
 & \downarrow & \downarrow & & \downarrow \\
\text{process } 1 \rightarrow & P_{1,1} & P_{1,2} & \cdots & P_{1,n} \\
\text{process } 2 \rightarrow & P_{2,1} & P_{2,2} & \cdots & P_{2,n} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
\text{process } n \rightarrow & P_{n,1} & P_{n,2} & \cdots & P_{n,n}
\end{array}$$

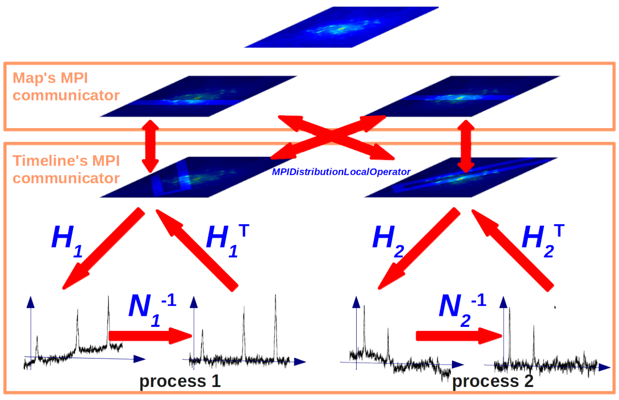

Fig. 2. Distributed unknown, distributed data. **Left:** Matrix representation of the operator D as MPIDistribution-LocalOperator. **Right:** Map-making example showing the computation of $D^{\mathrm{T}}H^{\mathrm{T}}N^{-1}HD$ by two MPI processes.

7 Parallelism

This package aims at being run on a single multicore CPU interactively (shared memory) and on clusters of CPUs (distributed memory). The first use relies on OpenMP's parallelism and the second one on MPI's. Calls to MPI routines have been encapsulated into specific operators, so that in a distributed environment, Eq.1.1 becomes:

$$\mathbf{y} = HD\mathbf{x} + \mathbf{n} \tag{7.1}$$

where $H = H_1 \oplus H_2 \oplus \ldots \oplus H_n$ is a block diagonal operator whose blocks are handled by n MPI processes, D is the MPI distribution model, y is the distributed data and x the unknown, which can be distributed or not. Note that in a non-distributed environment, D is the identity matrix, so switching from a non-distributed environment where most of the development takes place to a distributed one where actual computations are performed is straightforward.

7.1 Non-distributed unknown, distributed data

In this scheme, the unknown is global and replicated over the nodes. In Eq. 7.1, D is an MPIDistributionIdentityOperator: a block column operator whose blocks are the identity and for which each block output is handled by an MPI process. The actual MPI all-reduction is performed by D^{T}, a block row operator whose blocks are the identity, showing up in the normal equation $D^{\mathrm{T}}H^{\mathrm{T}}N^{-1}HDx = D^{\mathrm{T}}H^{\mathrm{T}}N^{-1}y$. Fig. 1 shows D as a matrix (left) and synthesizes how the computation of $D^{\mathrm{T}}H^{\mathrm{T}}N^{-1}HD$ is performed by two MPI processes (right). Since x is the same across all the processes, the solver doesn't need any MPI communication to compute the next iteration, so that all scipy's solvers can be readily used.

7.2 Distributed unknown, distributed data

The operator MPIDistributionLocalOperator handles an input partitioned into sections across the nodes and returns to each node the packed values of the global input according to the node's global mask. The matrix representation of such operator is shown in Fig. 2 (left). The block $P_{i,j}$ is a coordinate-selection matrix, which projects the input section handled by process j onto the process i according to i's mask. When the unknown is distributed, some amount of communication is required by the solver, generally when computing dot products, and scipy solvers cannot be used.

8 Conclusions

Coupled with the package PySimulators, the PyOperators package has proven to be an efficient way to construct astronomical instruments models used to simulate data acquisition, but we stress that these tools are very general and their scope goes beyond astronomical instrumentation. The details of this aspect will be treated in a subsequent paper. This package has also already been used to process amongst the largest *Herschel*/PACS observations and the results will be the subject of a third paper.

Future ideas of development include writing a PETSc interface, offloading computation to the GPU, automatic differentiation to obtain the gradient and hessian of an operator and inclusion of rules to factorise expressions. Owing to their open-source nature, the scientific community is welcome to provide a feedback and partipate in bettering these tools.

PC and NB thank the ASTRONET consortium for funding this work through the TAMASIS and CSH projects. The HPC resources were provided by GENCI-[CCRT] grant 2011-046661 and 2012-046661. M. Sauvage and F. Bournaud are thanked for their support.

References

Beck, A. & Teboulle, M. 2009, SIAM J. Imaging Sciences, Vol. 2, No. 1, p. 183

Seeger, M., Nickisch, H, Pohmann, R. et al. 2010, Magnetic Resonance in Medicine, 63, 116