Stencil Skeleton Evaluation

Francesco Piccinno <stack.box@gmail.com>

July 12, 2011

Contents

List of Figures

List of Tables

List of listings

Abstract

The aim of the report is to briefly introduce the concept of data-parallel paradigm and to study and evaluate the implementation of a stencil skeleton, a particular parallel computation in which data dependence pattern is implemented by the mean of inter-worker communications.

1 Introduction

A data parallel computation, which can be defined both on streams and or single data value, consists in the partition of possibly large data structures to different processing entities which are able to compute a function on the assigned partition in parallel. This parallel pattern bases its functionalities on function replication. As a consequence, the knowledge of the sequential computation form is necessary both for function replication and data partitioning. This is the main reason for the parallel program design complexity. Latency and memory capacity are optimized compared to other stream parallel paradigms like farms, though a potential load unbalance exists.

To better understand, let us consider a data parallel computation operating on a bi-dimensional matrix as depicted in Figure 1. The matrix is partitioned in blocks of $g = \frac{M^2}{n}$ elements. Successively each block is transmitted to each worker through a *scatter* operation. Then all workers apply the same function F to the corresponding partition in parallel. At the end, the result may be collected through a *gather* operation.

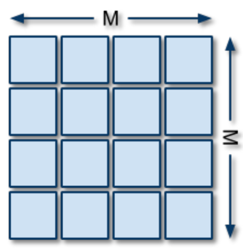


Figure 1: Data parallel partition scheme

Map
: When all the workers are *fully independent* and operate on their own partition, without any cooperation

Stencil-based
: When a workers operate in parallel and *cooperate* through data exchanges. In this case data dependencies are implemented by inter-worker communications.

The form of a stencil may be *statically* predictable or *dynamically* exploited according to the current data values. In case of static stencil we have two other possibilities: a *fixed* stencil where the communication scheme is fixed throughout the computation, and a *variable* stencil where the communication scheme varies from a computation step to the next one, although the overall behavior is statically predictable.

In this report we will concentrate our attention to *static fixed stencils* applied on bi-dimensional matrix.

2 Abstract implementation

In this section we briefly discuss the general idea of our implementation leaving the implementation details out of this section. The interested reader can find a more detailed analysis of the real implementation in "Concrete implementation" section.

The main idea we followed is very simple. Since we are dealing with a data-parallel pattern, some kind of partition of the input problem is required to properly solve the problem. The schema can be synthesized in four steps:

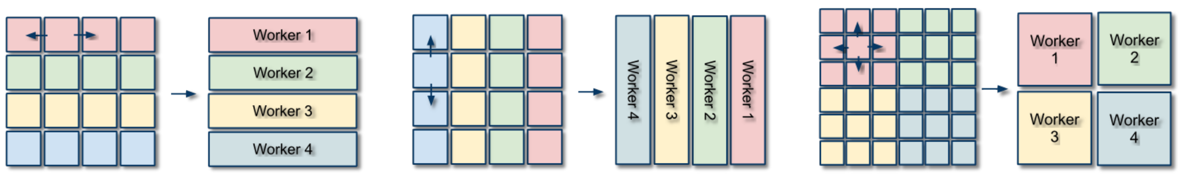
1. The input matrix is loaded by the collector and equally split among the available workers

2. Each worker communicate its contour segments needed to the other neighbors

3. After having received the necessary data from the neighbors, the computation on the given partition is started in parallel on each worker

4. The resulting partition is sent back toward a collector, which is in charge of collating partial results, thus providing the final result.

As you might imagine, the central point in this scheme which can introduce substantial optimization is the partition phase. This can be implemented in different ways on the basis on the input offsets.



(a) Partition by row (b) Partition by column (c) Partition by blocks

Figure 2: Different partition schemes

In our implementation, the partition phase tries to derive an optimal partition by trying to minimize the communication between workers. We have isolated three different situations depicted, which are depicted in Figure 2:

Row partition Whenever the input offsets set contains dependencies that are row dependent only, a row partition scheme is derived. This means that all the elements in the input offset set are of the type *(0,x)*. In this case no communication at all is required between workers, therefore a full *map* parallelism is possible if each worker has an entire row assigned.

Column partition This is the mirror of the *row partition* case and it is verified whenever a given element depends only on elements which are on the same column. In this situation the input offset set contains only elements of the type *(x, 0)*. No communications is required, on the condition that at each worker is assigned an entire column of the input matrix.

Block partition Whenever a *row* neither a *column partition* is applicable, a block partition is derived. In this case the input offset set contains "heterogeneous" points, so no kind of trivial derivation is possible. In our case a minimum rectangle called "Elementary Rectangle" is derived. This rectangle is evaluated by taking as reference point a single element and deriving the smallest rectangle that encompass all data dependencies needed to evaluate the function in that specific point.

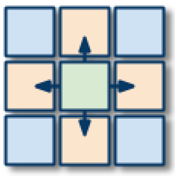
If we are in presence of a *Block partition* situation another optimization is still possible. Indeed, we can try to still partition the matrix by "row-blocks" or "column-blocks" by vertically or horizontally stacking single *elementary rectangles* thus achieving a more optimized partition. In fact this situation is able to avoid communications between "elementary" workers that are still present in a situation like the one depicted in Figure 2(c).

3 Cost model: completion time

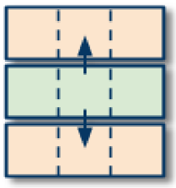
Since the cost model specifically depends on the offsets passed as input to the stencil skeleton, we have decided to derive a general approximated cost model which tries to collect the most configurations still maintaining a reasonable precision in the evaluation. Before reaching the final formulation, we explain the steps the brought us to the successful derivation.

In our case the starting matrix is read from the disk and then partitioned in some way among n workers through the use of a remote communication which has a cost of $T_{scatter}(n, g)$, where g is the block size assigned to each worker ($g = \frac{M^2}{n}$, assuming "balanced" partitions).

Then each worker after having received its own partition, is in charge to communicate to all the neighbors the necessary contour partition. Of course, this communication has an associated cost, which has to be evaluated but varies according to the specific input offsets passed to the skeleton. To model this part of the formula we have to introduce a parameter which is rc (real communications), which represents the number of communications we have to make with the neighbors. To better understand the situation, let's assume to have as input offsets: *(0, -1), (-1, 0), (0, 1) (1, 0)*



(a) Elementary partition



(b) Partition by row

Figure 3: Two ways of partitioning the same matrix

The situation is the one depicted in Figure 3(a). In this case we are assuming the "elementary" partition 1 by 1 where each worker has responsibility for just 1 matrix element. In this case before starting the computation the worker has to communicate respectively to the upper, next, lower and previous neighbor its data partition, assuming the "owners-compute-rule" to be satisfied. At the end the parameter rc will be 4. This has to be multiplied by the $T_{comm}(P)$ where P represent the size of the data to be transmitted to the neighbors. In this specific case this is equal to 1, but in other cases this may vary based on the size of the partition and the offsets, as you will see. In fact, by keeping the same input offsets and by only changing the size of the sub-matrix partition assigned to each worker the things changes, as presented in Figure 3(b). In this specific case, not only the P parameter changes and becomes 3 but also the rc changes to 2.

Then a calculation phase is started in parallel by each worker on its specific partition which takes a total time of $g \cdot T_f$, where T_f is the time needed to calculate the function among all inputs (in our case 5 considering the central value) while g is the size of the partition assigned to each worker or better the number of items a given worker has assigned inside the partition.

At the end of the calculation, a second phase is started which consists in transferring back the computed partition to the root node ($T_{gather}(n, g)$). This can be evaluated by simply taking into account the double communication at the first phase. Thus the resulting formula is:

$$2T_{scatter}(n, g) + rc \cdot T_{comm}(P) + g \cdot T_f$$

Function	M	T_{calc} (sec)	T_f (sec)
`min`	3000	61.923	$6.880{\cdot}10^{-06}$
	4000	95.880	$5.992{\cdot}10^{-06}$
	5000	146.599	$5.864{\cdot}10^{-06}$
	6000	206.726	$5.742{\cdot}10^{-06}$
`linsolve`	3000	711.255	$7.903{\cdot}10^{-05}$
	4000	1274.833	$7.968{\cdot}10^{-05}$

Table 1: Calculation time timings

As a measure of comparison, the sequential algorithm cost model is given by the last part of the previous formulation, considering that the partition in this case is the entire input matrix: $M^2 \cdot T_f$.

4 Performance measurements and tests

In this section we present performance measurements we have collected during our experiments. For this scope we have decided to test two different functions with different complexity. The first one is `min` which applied to various elements returns the minimum among them. The latter is `linsolve` a simple function which tries to solve a system of linear algebra equations. The only assumption we made during the experiment was to have as input offsets the following set: *(0, -1), (-1, 0), (0, 1) (1, 0)*.

To better understand the final goal of the `linsolve` function take in consideration the scheme below, which assumes "e" as the target element to be computed:

$$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix} \rightarrow \begin{cases} bx + hy = e \\ dx + fy = e \end{cases} \rightarrow result = \frac{x+y}{2}$$

Regarding the ideal statistics we have used some instrumentation inside the code to evaluate the calculation time $T_{calc} = M^2 \cdot T_f$ and used it to derive the correct T_f value. Results collected and derived are presented in Table 1.

Since the implementation we have provided makes use of MPI message passing library which does not guarantee any constant behaviour for the setup "socket" procedure, we preferred to avoid to take in consideration the T_{setup} parameter which should be paid once for every communication as $T_{comm}(L) = T_{setup} + L \cdot T_{transm}$ formula suggests. By the way this is not a wrong assumption, because in MPI after the setup procedure all peers involved in the MPI "world" are free to communicate each other without paying any more the setup procedure. So the setup is made once and for all, thus ignoring this parameter, by placing instrumentation code in proper position, does not impact in any way our study case.

The evaluation of T_{transm} parameter was simply evaluated by taking a statistic mean on top of the scattering time of the matrix registered by our instrumentation code. The resulting value results to be equal to $3.437 \cdot 10^{-07} \frac{item}{sec}$.

After having set up all the parameters we collected all the needed timings to derive the proper statistics. Since we were dealing with a data-parallel skeleton, we have focused our attention on the following parameters:

Completion time	Time needed to complete the computation on the given input	T_c
Efficiency	Provides a measure of how close is the effective performance with respect to the ideal one.	$\epsilon(n) = \frac{T_{c-id}^{(n)}}{T_c^{(n)}}$
Speedup	Provides a measure of the relative speed of the parallel computation with respect to the sequential one.	$s(n) = \frac{T_{c-id}^{(1)}}{T_c^{(n)}} = \epsilon(n) \cdot n$
Scalability	The ratio between parallel execution time with parallelism degree equal t 1 and the parallel execution with parallelism degree equal to n	$scalab(n) = \frac{T_c^{(1)}}{T_c^{(n)}}$

4.1 Statistics for min function

In this section we present the statistics we have collected regarding the `min` function. Since no interesting differences were reported for the mid-sized cases, we decided to just attach the statistics regarding the limiting cases. The former when the array size is equal to 3000 by 3000 elements. The latter where the array is composed of 6000 by 6000 elements.

A compare between completion time of the two limit cases is presented in Figure 4. As you can imagine, in this situation a solution of this kind does not offer any kind of advantages, since the time taken by the function to compute a given element is very low compared to the effort taken by the skeleton to distribute data and work among workers through the network. Therefore the transmission overhead highly impacts on the possibility to scale up.

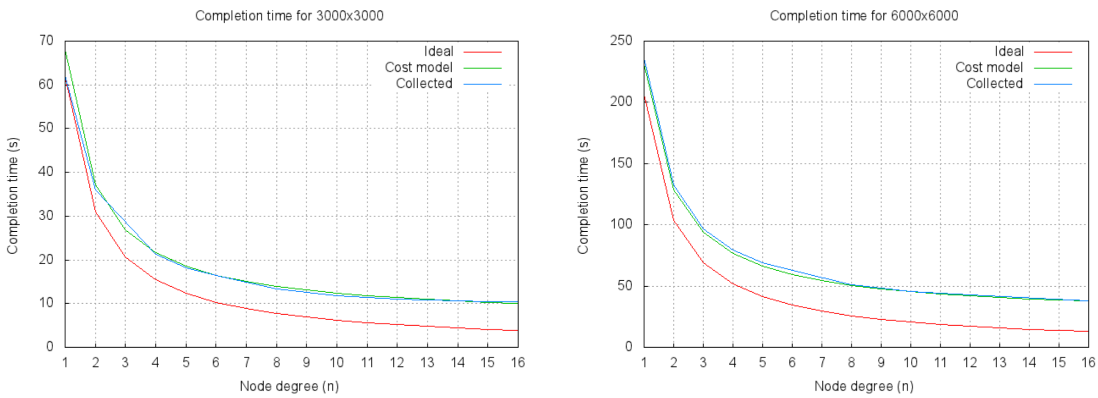


Figure 4: `min` function: Completion time

The thing is more evident if you take as reference Figure 5 which presents statistics regarding speedup, efficiency and scalability. As you can see, for the first case, with the maximum number of processing elements we can achieve a speedup equals to 6, which is quite low for the employed resources. By taking in consideration the second case, an efficiency of less than 35% produces a speedup of ≈ 5.5 for 16 processing elements.

4.2 Statistics for linsolve function

Regarding the parallelization of `linsolve` function, we got interesting results which demonstrate the goodness of our implementation. Figure 6 shows the completion time graph for the two runs. As you can see, the cost model elaborated really resemble in a very precise way the behaviour

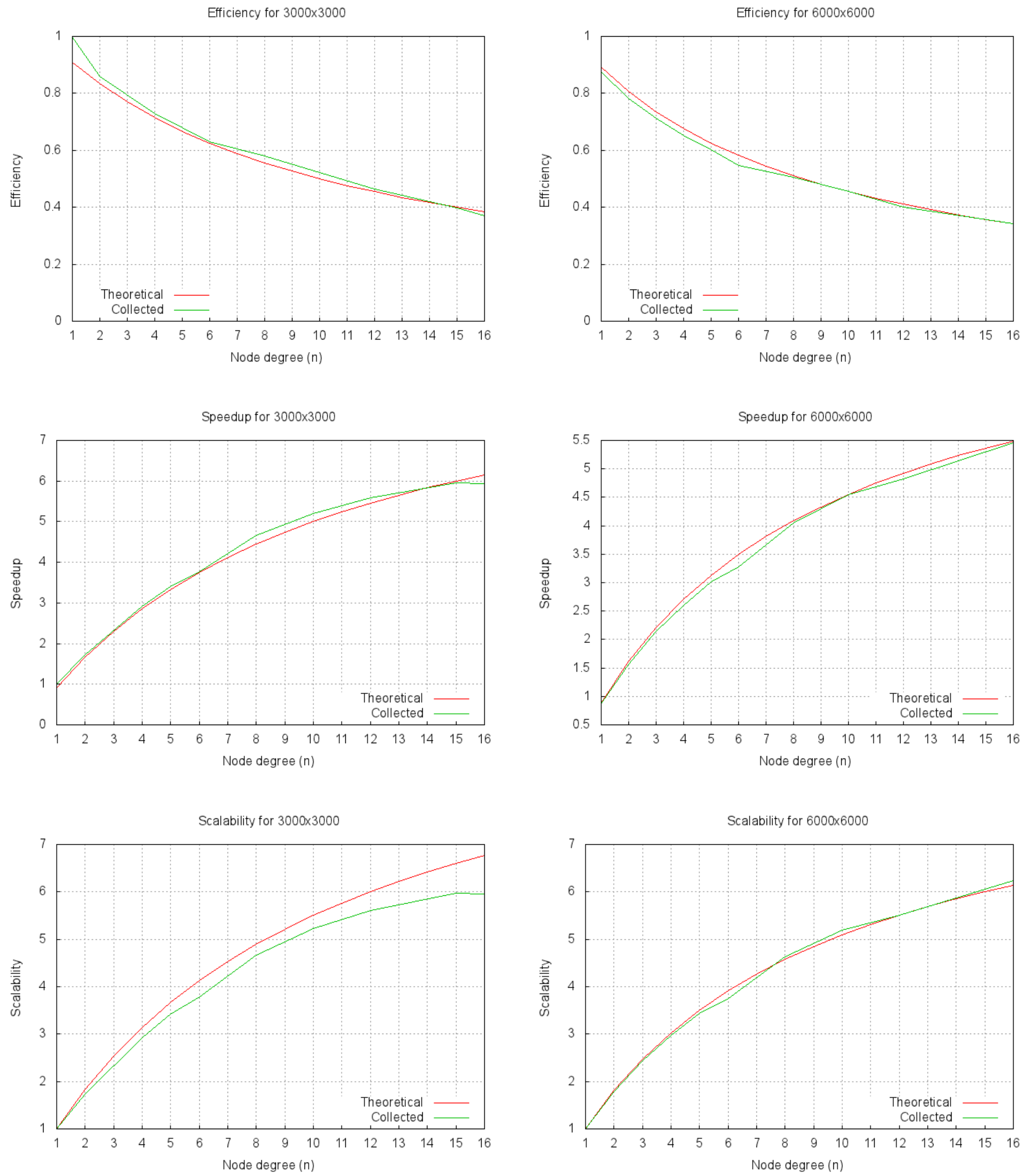


Figure 5: `min` function: Efficiency, Speedup and Scalability

of the program. Indeed all the three lines almost describe the same decay function. This is because our computation is time is very large compared to the other overhead we incur on when we split the matrix through the network of multi-cores.

On the other hand, Figure 7 shows a comparison between *speedup* and *efficiency* statistics. As you may notice, there is the presence of differences between the real and the predicted situation. Indeed this is not strange at all. The behavior is mostly caused by the semantic of `linsolve` function itself. In fact, the function tends to generate load unbalancing among workers, thus resulting in a slight increase of completion time and therefore in fluctuations between theoretical efficiency and the real one. This is because some partitions might have a major number of solvable systems while the others might have less or at least simpler one.

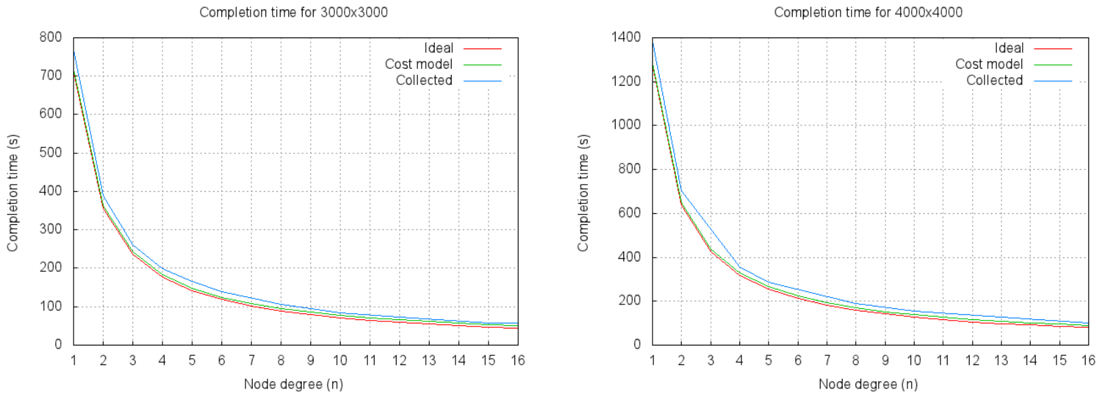


Figure 6: `linsolve` function: Completion time

Inputs	`min`	`linsolve`
3000	0.2143	1.296
4000	0.1156	1.101

Table 2: Average of standard deviation between workers calculation time

Anyhow, the final results demonstrate how much the parameter T_f impacts on the speedup.

This factor was further investigated and studied, by taking the standard deviation on the registered time interval taken by each worker to compute the given partition thus revealing some kind of asymmetry between the `min` and `linsolve` functions. As reference take in consideration Table 2. From a rapid look you can notice a magnitude of order of difference between the two functions. This means that the T_f parameter we take as reference is not fixed but may vary depending on the input values and should have taken as an average case measure. By the way, although slightly imprecise, the curve still describe with a given approximation the real situation.

5 Concrete implementation

Regarding the concrete implementation part, we decided to use Python[1] programming language jointly with `mpi4py`[2] library which provides a wrapper to Message Passing Interface API specification.

Since Python (actually CPython the official Python implementation) does not provide an efficient implementation of threads due to GIL[3] (Global Interpreter Lock) we decided to just use process as method for exploiting cluster of multi-cores resources. Regarding communications everything is delegated to the library which is in charge of introducing optimization whenever two workers are allocated on the same machine and they need to exchange data.

We will start our analysis by taking a look to the `run.py` (Listing 1) which is responsible to set up everything and start the real computation. It accepts as parameter two parameters. The first one may be set to "seq" to spawn the sequential version of the program or to "par" for the parallel version. The second parameter instead just pass the matrix file where the function should be applied to. The program accepts an optional third parameter which gives indication

[1] `http://python.org`
[2] `http://mpy4py.scipy.org`
[3] `http://wiki.python.org/moin/GlobalInterpreterLock`

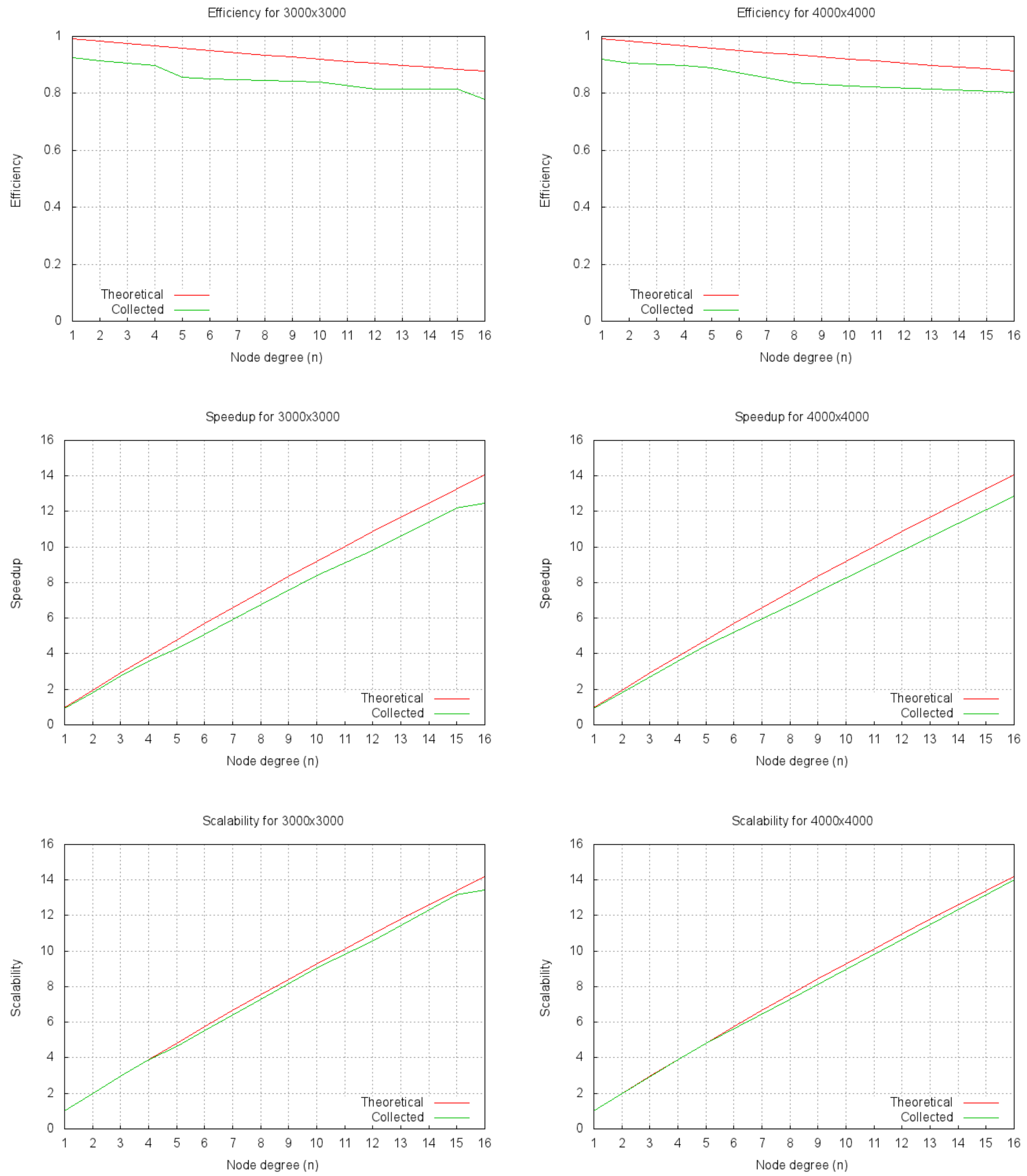


Figure 7: `linsolve` function: Efficiency, Speedup and Scalability

about the partition scheme. If it is not present the software will try do some calculation to derive an optimal partition scheme by following the procedure we have described in the "Abstract implementation" section.

As you can see in case of parallel version is required to be run, the run function is called which is in charge of setting up the master (`main` function) and the workers (`slave` function).

Listing 1: Spawner: `run.py`

```
import sys
from mpi4py import MPI
from main import main, sequential
from slave import slave
```

```python

def run(matrix_file, rows=None, cols=None):
    rank = MPI.COMM_WORLD.Get_rank()

    if rank == 0:
        main(matrix_file, rows, cols)
    else:
        slave()

if __name__ == "__main__":
    if len(sys.argv) < 3:
        print "Usage: %s <par|seq> <matrix-file> [rowsxcols]" % sys.argv[0]
        sys.exit(0)

    if sys.argv[1] == 'seq':
        sequential(sys.argv[2])
    elif sys.argv[1] == 'par':
        if len(sys.argv) == 4:
            params = map(int, sys.argv[3].split('x'))
        else:
            params = (None, None)

        run(sys.argv[2], *params)
```

The worker logic is presented in Listing 2. The worker just waits to receive its partition and then create a `StencilWorker` instance and starts the computation.

Listing 2: Worker logic: `slave.py`

```python
#!/usr/bin/env python
from mpi4py import MPI
from stencil import StencilWorker

def slave():
    worker = StencilWorker(*MPI.COMM_WORLD.scatter(root=0))
    worker.start()
```

Listing 3 shows the two function which are called by the `run.py` module. The first function is the one responsible for the parallel execution of the program, while the second is meant for sequential execution. Instrumentation code is also present to take into account loading time of the matrix, and completion time.

Listing 3: Master logic: `main.py`

```python
import cPickle as pickle
import time

from mpi4py import MPI
from stencil import Stencil
from functional import offsets

def main(matrix_file, rows=None, cols=None):
    start = time.time()
    matrix = pickle.load(open(matrix_file, "r"))
    print "%.10f seconds to load the matrix" % (time.time() - start)

    start = time.time()
    st = Stencil(offsets)
    st.apply(matrix, rows, cols)
    print "%.10f seconds to apply on %d processors" % \
            (time.time() - start, MPI.COMM_WORLD.Get_size() - 1)

def sequential(matrix_file):
    print "Trying sequential version"
    start = time.time()
    matrix = pickle.load(open(matrix_file, "r"))
    print "%.10f seconds to load the matrix" % (time.time() - start)
```

```

    st = Stencil(offsets)
    start = time.time()
    st.seq_apply(matrix)
    print "%.10f seconds to apply sequentially" % \
            (time.time() - start)
    #matrix.dump()
```

Now we will go deeper in the analysis of the stencil skeleton by introducing Listing 4. The main class in this case is the `Stencil` whose constructor takes as parameter the input offset set. Immediately after the `analyze_offsets` method is called which is in charge of extracting communication data dependencies and to convert them into elementary vectors tuples. This is needed whenever a dependency of the type *(x,y)* with $x \neq 0$ and $y \neq 0$ is present. In this specific case all the input offsets are analyzed to derive a tuple like *(max_x, max_y)*, for the composed directions (*eg.* up-left, down-right, ...).

The `apply` function is the most important function in this class. It is in charge of deriving a proper partition scheme in case (if required, by calling the homonym method) and to scatter the matrix among the workers. Method `fix_data_dep` is called before the actual scattering phase takes place. This is necessary to avoid bogus function evaluation in various corner cases, such the one depicted in Figure 8.

Figure 8: Bogus data derivation

In this situation, a partition 2×2 is derived while the input offset set is equal to *(-1, -1)*. Therefore although not only the up-left neighbor is involved but partially also the left and the upper neighbors. This corner case is verified whenever the composed direction tuple (x, y) is lesser than the partition dimension ($x < height \vee y < width$). If this condition is met, `fix_data_dep` function is in charge of adding redundant communication to the pattern (in this case the communication with the up and left neighbors).

After this check a phase of set-up is made, to interconnect the workers by the mean of `autoconnect` method, which is called twice first to connect left-, right-, up- and down- neighbors each other and then for the remaining up-right, up-left, down-right and down-left neighbors.

Then the real scattering phase is done and after the gathering procedure followed by a reconstruction phase. A final barrier is then used to synchronize all the remote entities and conclude the execution.

Regarding the class `StencilWorker` which is in charge of modeling the behaviour of the workers, no further comment is needed. The class first sends the contour part of the partition to the other workers, then it waits to receive the parts which are sent by the other neighbors. After reception the calculation phase is executed by making use of the `Puzzle` class. The final step is taking part in the global gathering phase were each worker sends back to the master (node with rank 0) its own computed part.

Listing 4: Stencil skeleton implementation: `stencil.py`

```
import time
```

```python
import numpy
import logging

from mpi4py import MPI

from matrix import Matrix
from puzzle import Puzzle
from functional import function

from communicator.enum import *
from communicator.mpi import Communicator

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
name = MPI.Get_processor_name()

logging.basicConfig()

log = logging.getLogger("stencil")
log.setLevel(logging.INFO)

class StencilWorker(object):
    _id = 0

    def __init__(self, offsets, data_segments, partition, pwidth, pheight, \
                 connections=None):
        self.wid = StencilWorker._id
        self.offsets = offsets
        self.data_segments = data_segments
        self.partition = partition

        self.col, self.row = 0, 0
        self.width, self.height = self.partition.cols, self.partition.rows
        self.pwidth, self.pheight = pwidth, pheight

        self.conns = connections

        if rank == 0:
            log.debug("Worker %d has %s" % (StencilWorker._id, self.partition))
            StencilWorker._id += 1
        else:
            self.comm = Communicator(self)

    def autoconnect(self, wdict, rows=None, cols=None):
        if self.conns:
            if self.conns.up:
                self.conns.up_left = self.conns.up.conns.left
                self.conns.up_right = self.conns.up.conns.right
            if self.conns.down:
                self.conns.down_left = self.conns.down.conns.left
                self.conns.down_right = self.conns.down.conns.right

            return

        tot = int(rows * cols)

        i_col = self.wid % cols
        i_row = int(self.wid / cols)

        start = cols * i_row

        # Here we calculate our position in the general matrix
        self.col, self.row = i_col * self.height, i_row * self.width

        links = []

        for wid in ((self.wid + 1) % cols + start,
                    (self.wid - 1) % cols + start,
                    (self.wid - cols) % tot,
                    (self.wid + cols) % tot):
```

```python

            if wid == self.wid:
                links.append(None)
            else:
                links.append(wdict[wid])

        self.conns = Connections(*links)

    def __repr__(self):
        return 'Worker(%d [%d %d %d %d])' % (self.wid, self.row, self.col,
                self.width, self.height)

    def start(self):
        log.info("Worker started on processor %d %s" % (rank, name))

        # First we send all the required partitions to the neighbors and then
        # receive all the segments and reconstruct a local matrix where the
        # function will be evaluated.

        puzzle = Puzzle(self.partition)

        for lbl, enum in zip(LABELS, ORDERED):
            self.comm.send(getattr(self.conns, lbl), enum)

        for lbl, enum in zip(RLABELS, REVERSED):
            m = self.comm.receive(getattr(self.conns, lbl), enum)
            if m: puzzle.add_piece(m, enum)

        start = time.time()
        puzzle.apply(self.offsets, function)
        log.info("Worker %d: %.10f seconds to compute the partition" % \
                 (rank - 1, time.time() - start))

        log.info("Sending back the computed sub-partition from %d" % rank)

        start = time.time()
        comm.gather(self.partition.matrix, root=0)
        log.info("Worker %d: %.10f seconds to send back the partition" % \
                 (rank - 1, time.time() - start))

        comm.Barrier()

class Stencil(object):
    def __init__(self, offsets):
        self.offsets = offsets
        self.data_segments = None
        self.analyze_offsets()

    def analyze_offsets(self):
        """
        This function is in charge of extracting the proper sub-partitions that
        must be transmitted to the other workers in a stencil fashion
        """
        extract = lambda l, rev: sorted(filter(l, self.offsets), reverse=rev)

        right = extract(lambda x: x[0] == 0 and x[1] > 0, True)
        left  = extract(lambda x: x[0] == 0 and x[1] < 0, False)
        up    = extract(lambda x: x[0] < 0 and x[1] == 0, False)
        down  = extract(lambda x: x[0] > 0 and x[1] == 0, True)

        # Sort the first key which is the row so order is False since we are
        # interested in far jumps which are < 0
        up_left  = extract(lambda x: x[0] < 0 and x[1] < 0, False)

        if up_left:
            max_row = up_left[0][0]
            max_col = sorted(up_left, key=lambda x: x[1], reverse=False)[0][1]
            tgt_up_left = (max_row, max_col)
        else:
            tgt_up_left = None
```

```
        up_right = extract(lambda x: x[0] < 0 and x[1] > 0, False)

        if up_right:
            max_row = up_right[0][0]
            max_col = sorted(up_right, key=lambda x: x[1], reverse=True)[0][1]
            tgt_up_right = (max_row, max_col)
        else:
            tgt_up_right = None

        # Sort the first key which is the row so order is True since we are
        # interested in far jumps which are > 0
        down_left = extract(lambda x: x[0] > 0 and x[1] < 0, True)

        if down_left:
            max_row = down_left[0][0]
            max_col = sorted(down_left, key=lambda x: x[1], reverse=False)[0][1]
            tgt_down_left = (max_row, max_col)
        else:
            tgt_down_left = None

        down_right = extract(lambda x: x[0] > 0 and x[1] > 0, True)

        if down_right:
            max_row = down_left[0][0]
            max_col = sorted(down_left, key=lambda x: x[1], reverse=True)[0][1]
            tgt_down_right = (max_row, max_col)
        else:
            tgt_down_right = None

        tgt_right = right and right[0] or None
        tgt_left  = left and left[0] or None
        tgt_up    = up and up[0] or None
        tgt_down  = down and down[0] or None

        log.debug("List of possible targets:")
        log.debug("Left: %s Right: %s" % (tgt_left, tgt_right))
        log.debug("Up   : %s Down : %s" % (tgt_up, tgt_down))

        log.debug("Up-left  : %s Up-right : %s" % \
                  (tgt_up_left, tgt_up_right))
        log.debug("Down-left  : %s Down-right : %s" % \
                  (tgt_down_left, tgt_down_right))

        # Now we try to assign correct mapping. Please beware that if you
        # change the enumeration order you also need to change the order of
        # this assignment.

        self.data_segments = [
            tgt_left,
            tgt_right,
            tgt_down,
            tgt_up,
            tgt_down_right,
            tgt_up_left,
            tgt_down_left,
            tgt_up_right,
        ]

    def fix_data_deps(self, prow, pcol):
        """
        Try to fix data dependencies to avoid bogus data
        @param prow is the number of rows each worker has assigned to
        @param pcol is the number of cols each worker has assigned to
        """
        def adjust(corner1, corner2, direction, check_idx, set_idx):
            targets = filter(lambda x: x,
                    [self.data_segments[REVERSED[corner1]],
                     self.data_segments[REVERSED[corner2]]])

```

```
            if not targets: return

            targets.sort(reverse=True, key=lambda x: abs(x[check_idx]))
            rect = targets[0]
            checker = (prow, pcol)

            # If our partition to transmit is smaller than the partition size
            # assigned it means that we are loosing part of the block needed to
            # the computation.
            if rect and abs(rect[check_idx]) < checker[check_idx]:
                target = self.data_segments[REVERSED[direction]]

                # If it is so we have to evaluate the maximum number of items
                # we have other dimension.
                if (target and abs(target[set_idx]) < abs(rect[set_idx])) or \
                   not target:

                    if check_idx == 0: out = (0, rect[set_idx])
                    else:              out = (rect[set_idx], 0)

                    self.data_segments[REVERSED[direction]] = out
                    log.info("Fixing data dependencies by adding %s in %s " \
                             "direction" % (str(out), LABELS[direction]))

        adjust(UP_LEFT, DOWN_LEFT, LEFT, 0, 1)
        adjust(UP_RIGHT, DOWN_RIGHT, RIGHT, 0, 1)

        adjust(UP_LEFT, UP_RIGHT, UP, 1, 0)
        adjust(DOWN_LEFT, DOWN_RIGHT, DOWN, 1, 0)

    def apply(self, matrix, rows=None, cols=None):
        nw = comm.Get_size() - 1

        if rows is not None and cols is not None:
            log.info("Skipping auto-derivation")
            rw = (matrix.cols / cols) * (matrix.rows / rows)
        else:
            rows, cols, rw = matrix.derive_partition(self.offsets, nw)

        self.fix_data_deps(rows, cols)

        wdict = {}
        workers = []

        for _ in range(rw):
            # Just assign an empty invalid partition the correct partition will
            # be derived when needed.
            worker = StencilWorker(self.offsets,
                                   self.data_segments,
                                   Matrix.from_list(rows, cols, None),
                                   matrix.cols, matrix.rows)
            wdict[worker.wid] = worker
            workers.append(worker)

        log.info("After partition => Rows: %d Cols: %d" % \
                 (matrix.rows/rows, matrix.cols/cols))

        for worker in workers:
            worker.autoconnect(wdict, matrix.rows / rows, matrix.cols / cols)
        for worker in workers:
            worker.autoconnect(wdict)
        for worker in workers:
            worker.conns.convert_to_id()

        data = [None,]

        for worker in workers:
            partition = matrix.partition(rows, cols, worker.wid)
            data.append((worker.offsets, worker.data_segments, partition, \
                         worker.pwidth, worker.pheight, worker.conns))
```

```python

        start = time.time()
        comm.scatter(data, root=0)
        log.info("%.10f seconds to scatter the matrix" % \
                 (time.time() - start))

        matrix = self.reconstruct(matrix.cols/cols)

        log.info("Terminated.")
        comm.Barrier()

        #print "Result is"
        #print matrix

    def reconstruct(self, col_stop):
        start = time.time()
        data = comm.gather(None, root=0)
        log.info("%.10f seconds to gather the results" % \
                 (time.time() - start))

        start = time.time()

        rows = []
        row, col = 0, 0
        curr_row = None

        for partition in data[1:]:
            # Creates rows then merge back
            if curr_row is None:
                curr_row = partition
            else:
                curr_row = numpy.column_stack((curr_row, partition))

            col += 1

            if col == col_stop:
                col = 0
                row += 1

                rows.append(curr_row)
                curr_row = None

        matrix = None

        for row in rows:
            if matrix is None:
                matrix = row
            else:
                matrix = numpy.vstack((matrix, row))

        log.info("%.10f seconds to reconstruct the result" % \
                 (time.time() - start))
        return matrix

    def seq_apply(self, matrix):
        old = matrix.clone()
        rows, cols = matrix.rows, matrix.cols

        for i in range(matrix.rows):
            for j in range(matrix.cols):
                val = old.matrix[i][j]

                for (x, y) in self.offsets:
                    val = function(val,
                       old.matrix[(i + x) % rows][(j + y) % cols]
                    )

                matrix.matrix[i][j] = val
```

Listing 5 presents the class which is used by the `StencilWorker` to re-arrange all contour partitions received by the neighbors in a new sub-matrix, which will then be used to execute the evaluation of the functional code. We make use of utilities functions such as `vstack` and `hstack` which are provided by the `numpy`[4] library, respectively to vertically or horizontally stack small matrices into bigger ones.

Listing 5: Stencil skeleton implementation: `puzzle.py`

```
import time
import logging

from numpy import zeros, vstack, hstack
from matrix import Matrix
from communicator.enum import UP, DOWN, LEFT, RIGHT, \
                              DOWN_LEFT, DOWN_RIGHT, \
                              UP_LEFT, UP_RIGHT

logging.basicConfig()

log = logging.getLogger("puzzle")
log.setLevel(logging.INFO)

class Puzzle(object):
    def __init__(self, center):
        self.center = center
        self.pieces = [None, ] * 8

        self.max_up    = 0
        self.max_down  = 0
        self.max_left  = 0
        self.max_right = 0

    def add_piece(self, piece, position):
        if position in (UP, UP_LEFT, UP_RIGHT):
            self.max_up = max(self.max_up, piece.rows)
        if position in (DOWN, DOWN_LEFT, DOWN_RIGHT):
            self.max_down = max(self.max_down, piece.rows)
        if position in (LEFT, DOWN_LEFT, UP_LEFT):
            self.max_left = max(self.max_left, piece.cols)
        if position in (RIGHT, DOWN_RIGHT, UP_RIGHT):
            self.max_right = max(self.max_right, piece.cols)

        self.pieces[position] = piece

    def pad(self, part, where, dtype):
        if part is None:
            if where == RIGHT and self.max_right > 0:
                return zeros((1, self.max_right))
            if where == LEFT  and self.max_left > 0:
                return zeros((1, self.max_left))
            if where == UP    and self.max_up > 0:
                return zeros((self.max_up, 1))
            if where == DOWN  and self.max_down > 0:
                return zeros((self.max_down, 1))

            return None

        if isinstance(part, Matrix): m = part.matrix
        else:                        m = part

        if where == RIGHT and m.shape[1] < self.max_right:
            pad = zeros((m.shape[0], self.max_right - m.shape[1]), dtype)
            return hstack((m, pad))

        elif where == LEFT and m.shape[1] < self.max_left:
            pad = zeros((m.shape[0], self.max_left - m.shape[1]), dtype)
```

[4] http://numpy.scipy.org

```
            return hstack((pad, m))

        elif where == UP and m.shape[0] < self.max_up:
            pad = zeros((self.max_up - m.shape[0], m.shape[1]), dtype)
            return vstack((pad, m))

        elif where == DOWN and m.shape[0] < self.max_down:
            pad = zeros((self.max_down - m.shape[0], m.shape[1]), dtype)
            return vstack((m, pad))

        return m

    def apply(self, offsets, function):
        start = time.time()
        matrix = self.center.matrix.copy()

        up = self.pad(self.pieces[UP], UP, matrix.dtype)
        down = self.pad(self.pieces[DOWN], DOWN, matrix.dtype)

        is_empty = lambda x: x is not None

        matrix = vstack(filter(is_empty, (up, matrix, down)))

        left  = self.pad(self.pieces[LEFT], LEFT, matrix.dtype)
        right = self.pad(self.pieces[RIGHT], RIGHT, matrix.dtype)

        upl   = self.pad(self.pad(self.pieces[UP_LEFT], UP, matrix.dtype),
                         LEFT, matrix.dtype)
        downl = self.pad(self.pad(self.pieces[DOWN_LEFT], DOWN, matrix.dtype),
                         LEFT, matrix.dtype)
        upr   = self.pad(self.pad(self.pieces[UP_RIGHT], UP, matrix.dtype),
                         RIGHT, matrix.dtype)
        downr = self.pad(self.pad(self.pieces[DOWN_RIGHT], DOWN, matrix.dtype),
                         RIGHT, matrix.dtype)

        # Hacky fix
        if self.max_up == 0:    upl, upr     = None, None
        if self.max_down == 0:  downl, downr = None, None
        if self.max_left == 0:  downl, upl   = None, None
        if self.max_right == 0: downr, upr   = None, None

        coll = filter(is_empty, (upl, left, downl))

        if coll: left = vstack(coll)
        else:    left = None

        coll = filter(is_empty, (upr, right, downr))

        if coll: right = vstack(coll)
        else:    right = None

        matrix = hstack(filter(is_empty, (left, matrix, right)))

        log.info("%.10f seconds to create auxiliary matrix" % \
                 (time.time() - start))

        dest = self.center
        rows, cols = matrix.shape
        idisp, jdisp = self.max_up, self.max_left

        for i in range(self.max_up, self.max_up + self.center.rows):
            for j in range(self.max_left, self.max_left + self.center.cols):
                val = matrix[i][j]

                for (x, y) in offsets:
                    val = function(val, matrix[(i + x) % rows][(j + y) % cols])

                dest.matrix[i - idisp][j - jdisp] = val
```

To finish our analysis we present the communicator class which is responsible to exchange data between workers whenever there is the necessity to do it. Listing 6 presents the `Communicator` class, which provides communication by means of MPI API specification. The only method which deserves an explanation is the `send` method. It is in charge, by looking in the `data_segment` structure, to literary extract the needed sub-matrix from the central one. This is done by the means of the `extract` method provided by the high-level `Matrix` class.

Listing 6: Communication: `communicator/mpi.py`

```python
import logging
from mpi4py import MPI
from enum import REVERSED, LABELS

logging.basicConfig()

log = logging.getLogger("comm-mpi")
log.setLevel(logging.INFO)

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

class Communicator(object):
    def __init__(self, parent):
        self.parent = parent

    def send(self, remote, direction):
        rect = self.parent.data_segments[direction]

        if rect is None or remote == None or remote == rank - 1:
            return

        if rect[0] > 0:
            row_stop = rect[0]
            row_start = 0
        elif rect[0] < 0:
            row_stop = self.parent.height
            row_start = self.parent.height - abs(rect[0])
        else:
            row_stop = self.parent.height
            row_start = 0

        if rect[1] > 0:
            col_stop = rect[1]
            col_start = 0
        elif rect[1] < 0:
            col_stop = self.parent.width
            col_start = self.parent.width - abs(rect[1])
        else:
            col_stop = self.parent.width
            col_start = 0

        m = self.parent.partition.extract(row_start, row_stop, \
                                          col_start, col_stop)

        log.debug("%d --> send to  : %s (direction %s)" % \
                  (rank - 1, str(remote), LABELS[direction]))

        comm.send(m, dest=remote + 1, tag=direction)
        log.debug("%d --> send to  : %s DONE" % (rank - 1, str(remote)))

    def receive(self, remote, direction):
        rect = self.parent.data_segments[REVERSED[direction]]

        if rect is None or remote is None or remote == rank - 1:
            return

        log.debug("%d <-- recv from: %s (direction %s)" % \
                  (rank - 1, str(remote), LABELS[direction]))
```

```

        data = comm.recv(source=remote + 1, tag=REVERSED[direction])

        log.debug("%d <-- recv from: %s (direction %s) DONE" % \
                  (rank - 1, str(remote), LABELS[direction]))

        return data
```

Listing 7 is presented just to complete the global vision. It just contains various constants and enumerations which are used in various part of the code, and the `Connections` class which is just a class placeholder that is used to store information about the connections between workers (neighbor information).

Listing 7: Communication constants: `communicator/enum.py`

```python
RIGHT,      \
LEFT,       \
UP,         \
DOWN,       \
UP_LEFT,    \
DOWN_RIGHT, \
UP_RIGHT,   \
DOWN_LEFT = range(8)

LABELS = 'right left up down up_left down_right up_right down_left'
RLABELS = 'left right down up down_right up_left down_left up_right'

LABELS = LABELS.split(' ')
RLABELS = RLABELS.split(' ')
ORDERED = range(8)
REVERSED = [LEFT, RIGHT, DOWN, UP, DOWN_RIGHT, UP_LEFT, DOWN_LEFT, UP_RIGHT]

class Connections(object):
    def __init__(self, right, left, up, down):
        self.right = right
        self.left = left
        self.up = up
        self.down = down

        self.up_left = None
        self.up_right = None
        self.down_left = None
        self.down_right = None

    def __repr__(self):
        return "right=%s, left=%s, up=%s, down=%s," \
                "ul=%s, ur=%s, dl=%s, dr=%s" % \
            (self.right, self.left, self.up, self.down,
             self.up_left, self.up_right, self.down_left, self.down_right)

    def convert_to_id(self):
        for lbl in LABELS:
            obj = getattr(self, lbl)

            if obj is None:
                setattr(self, lbl, None)
                continue

            wid = obj.wid

            setattr(self, lbl, wid)
```

The last file is the higher level encapsulation of the `numpy.array` data structure. The class `Matrix` is shown in Listing 8. Besides typical methods, such as `get set` or `random`, it provides also a `derive_partition` method which applies the same reasoning we have presented in Section 2 to derive an optimal partitioning scheme and `extract` which is used by the `Puzzle` class as mentioned before.

Listing 8: Matrix class: `matrix.py`

```
import itertools
import logging
import numpy
import copy

logging.basicConfig()

log = logging.getLogger("matrix")
log.setLevel(logging.INFO)

class Matrix(object):
    """
    The text below is considered a test case.

    >>> m = Matrix.from_string(6, 5, "2 3 4 5 6 5 6 8 9 1 4 4 6 7 3 6 6 3 2 3 0 4 6 3 0 3 5 1 6 99")
    >>> row, col, proc = m.derive_partition([(0, 1), (0, 2)], 100)
    >>> (row, col, proc)
    (1, 2, 15)
    >>> row, col, proc = m.derive_partition([(1, 0), (3, 0)], 100)
    >>> (row, col, proc)
    (3, 1, 10)
    >>>

    Test the limit
    >>> row, col, proc = m.derive_partition([(1, 0), (3, 0)], 8)
    >>> (row, col, proc)
    (3, 2, 5)

    >>> m = Matrix.from_string(50, 40, ' '.join([i for i in map(lambda x: str(x), range(50 *40))]))
    >>> m.derive_partition([(-3, -2), (2, 2)], 10)
    (5, 40, 10)
    >>> m.derive_partition([(-1, -1), (2, 2), (0, -1)], 111)
    (5, 4, 100)
    """

    def __init__(self, rows, cols, contents=None):
        self.rows, self.cols = rows, cols

        if contents is not None:
            self.matrix = contents
        else:
            self.matrix = numpy.empty((rows, cols))

    def clone(self):
        return copy.deepcopy(self)

    def dump(self):
        print str(self.matrix)

    @staticmethod
    def from_string(rows, cols, contents):
        mtx = Matrix(rows, cols)
        elems = itertools.imap(int,
                               contents.replace("\n", " ").strip().split(" "))
        try:
            for i in range(rows):
                for j in range(cols):
                    mtx.matrix[i][j] = elems.next()
        except StopIteration:
            pass

        return mtx

    @staticmethod
```

```
    def from_list(rows, cols, contents):
        return Matrix(rows, cols, contents)

    @staticmethod
    def random(rows, cols):
        contents = numpy.random.randint(0, 9, (rows, cols))
        return Matrix(rows, cols, contents)

    def derive_partition(self, offsets, nproc):
        """
        This method should derive a proper partition scheme starting from the
        offsets you pass in input

        @param offsets model the dependencies you need in order to properly
                       evaluate your function on the matrix.
        @param nproc number of processors you have available
        @return a tuple (rows, cols, rw) that can be used to derive the correct
                partition
        """

        def get_min_step(lst, idx):
            if not lst: return 1
            return lst[0][idx]

        row_dependent = sorted([x for x in offsets if x[0] != 0],
                               key=lambda x: abs(x[0]), reverse=True)
        col_dependent = sorted([x for x in offsets if x[1] != 0],
                               key=lambda x: abs(x[1]), reverse=True)

        # Now we need to get the maximum displacement element, so we
        # sort by using abs()

        def trivial_sol(depends, is_col=0):
            aref, bref = self.cols, self.rows
            if is_col == 0: aref, bref = bref, aref

            astep = get_min_step(depends, is_col)
            num = aref / astep

            # Adaption in the other direction is not needed since we
            # start from the barely minimum

            while num > nproc:
                if astep * 2 > self.cols:
                    break

                astep *= 2
                num = int(aref / astep)

            bstep = max(int(bref / (nproc - num)), 1)
            eproc = int((aref * bref) / (bstep * astep))

            while eproc > nproc:
                bstep *= 2
                eproc = int((aref * bref) / (bstep * astep))

            return (is_col == 0) and (astep, bstep, eproc) \
                                 or (bstep, astep, eproc)


        # If we do not depend on rows in any way we can fully exploit
        # the parallelism pattern and do a partition by rows.

        if not row_dependent:
            return trivial_sol(col_dependent, 1)
        elif not col_dependent:
            return trivial_sol(row_dependent, 0)

        log.debug("Non-trivial case evaluation triggered")

```

```
        # Now for non-trivial partition we derive the biggest square
        # that our offsets can derive.

        extract = lambda l, k, rev: sorted(filter(l, offsets), reverse=rev)

        # Sort row descending
        targets = extract(lambda x: x[0] > 0,
                          lambda x: x[0], True)

        height = targets and targets[0][0] or 0

        targets = extract(lambda x: x[0] < 0,
                          lambda x: x[0], False)

        height += targets and abs(targets[0][0]) or 0

        if height > 0:
            height += 1

        # Sort row descending
        targets = extract(lambda x: x[1] > 0,
                          lambda x: x[1], True)

        width = targets and targets[0][1] or 0

        targets = extract(lambda x: x[1] < 0,
                          lambda x: x[1], False)

        width += targets and abs(targets[0][1]) or 0

        if width > 0:
            width += 1

        log.debug("Possible partition individuated %dx%d" % (height, width))

        # Now we try to figure out how many workers we can spawn

        if self.cols % width != 0:
            width = max(1, self.cols / nproc)

        if self.rows % height != 0:
            height = max(1, self.rows / nproc)

        def throttle(is_width):
            aparam, bparam = width, height
            if not is_width: aparam, bparam = bparam, aparam

            astep = aparam
            elems = self.rows * self.cols

            while (elems / (aparam * bparam)) > nproc:
                aparam += astep

                if (elems / (aparam * bparam)) < nproc:
                    break

            return aparam

        log.debug("Trying to fit %d processors" % nproc)

        if width <= height:
            width = throttle(True)
            height = throttle(False)
        else:
            height = throttle(False)
            width = throttle(True)

        log.debug("Possible partition individuated %dx%d" % (height, width))

        eproc = int((self.cols * self.rows) / (height * width))
```

```python

        return (height, width, eproc)

    def partition(self, rows, cols, idx):
        col_idx = (idx * cols)
        row_idx = int(col_idx / self.cols) * rows
        col_idx %= self.cols
        return self.extract(row_idx, rows + row_idx, col_idx, cols + col_idx)

    def extract(self, row_start, row_stop, col_start, col_stop):
        return Matrix(row_stop - row_start, col_stop - col_start,
                      self.matrix[row_start:row_stop,col_start:col_stop])

    def get(self, i, j):      return self.matrix[i][j]
    def set(self, i, j, val): self.matrix[i][j] = val

if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Regarding the functional code this is simply stored in the `functional.py` file presented in Listing 9. In the file the `offsets` variable and a function named `function` is exported to the other modules.

Listing 9: Functional code: `functional.py`

```python
import numpy

offsets = ((-1, 0), (0, -1), (0, 1), (1, 0))

def variance(a, b):
    if isinstance(a, int):
        n, mean, m2 = 0, 0, 0
    elif isinstance(a, list):
        n, mean, m2 = a

    n += 1
    delta = b - mean
    mean += delta / n
    m2 = m2 + delta * (b - mean)

    if n == 4:
        return m2/(n - 1)
    else:
        return [n, mean, m2]

function = variance

def linsolve(a, b):
    """
    Here we try to interpret the matrix as follows
      2
    1 5 3
      4

    5 will be our a parameter. The aim of the function is to find a solution to
    the given algebraic system:
     1x + 3y = 5
     2x + 4y = 5

    Then the mean value is calculated and put in the center.
    """
    if isinstance(a, int):
        return [a, b]
    elif isinstance(a, list):
        a.append(b)

        if len(a) == 5:
            equations = numpy.array(((a[1], a[4]),
```

```
                                         (a[2], a[3])))
            given     = numpy.array((a[0], a[0]))
            try:
                return numpy.linalg.solve(equations, given).mean()
            except Exception:
                return 0
        return a


#offsets = ((-2, -1), (-2, 0), (-2, -2))
function = min
```

6 Build instructions

In order to successfully run the source code you need at least CPython 2.5.x. For getting a usable build of `mpi4py` package you need a compiler and the development file of the CPython. You can easily install them by typing:

```
$ apt-get install python python-dev gcc
```

To avoid installing the extension system wide, we have used `virtualenv`[5] to create an isolated Python environment.

```
$ wget http://pypi.python.org/packages/source/v/virtualenv/virtualenv-1.6.1.tar.gz
$ tar xfz virtualenv-1.6.1.tar.gz
$ python virtualenv-1.6.1/virtualenv.py mpi
New python executable in mpi/bin/python
Installing setuptools...........done.
Installing pip...............done.
$ . mpi/bin/activate
(mpi) $
```

After creating the environment you can install the MPI python wrapper. In our test-bed we were using machines with `MPICH-1`[6] implementation of the 1.0 MPI API specification.

In our case since MPIv1.0 is not well supported we have to do some tricks in order to get a working build. We have to thank Lisandro Dalcin the author of the library for his precious support.

```
(mpi) $ http://mpi4py.googlecode.com/files/mpi4py-1.2.2.tar.gz
(mpi) $ tar xfz mpi4py-1.2.2.tar.gz && cd mpi4py-1.2.2

# This is needed to fix bogus error message in the finalization state.
# Take a look to http://code.google.com/p/mpi4py/issues/detail?id=20 for
# more information
(mpi) $ wget http://mpi4py.googlecode.com/svn/trunk/src/python.c -O src/python.c
(mpi) $ export MPICC=/usr/bin/mpicc.mpich
(mpi) $ export MPICH_USE_SHLIB=yes
(mpi) $ python setup.py install
(mpi) $ python setup.py build_exe
(mpi) $ python setup.py install_exe
```

[5] http://pypi.python.org/pypi/virtualenv

[6] http://www.mcs.anl.gov/research/projects/mpich2/

At the end of this phase you should have a `python2.5-mpi` executable installed under `mpi/bin/` directory. Now to run a specific test you have first to create a random matrix. You have to change the current directory and make it point to `skippy/src`. Then simply spawn a python interpreter and type:

```
Python 2.5.2 (r252:60911, Jan  4 2009, 17:40:26)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import matrix
>>> import cPickle # For serialization
>>> cPickle.dump(matrix.Matrix.random(3000, 3000), open("/tmp/matrix-3000-3000.mtx", "w"))
>>> quit()
```

This should create a randomly filled matrix 3000 by 3000 in your `/tmp` directory. Now to run the program on different machines you have to write down a *machinefile* which contains the specification about the machines to use in your network. For our tests we have used the following file:

```
axth1.cli.di.unipi.it:3
axth3.cli.di.unipi.it:2
axth4.cli.di.unipi.it:2
axth6.cli.di.unipi.it:2
axth7.cli.di.unipi.it:2
axth8.cli.di.unipi.it:2
axth10.cli.di.unipi.it:2
axth15.cli.di.unipi.it:2
axth29.cli.di.unipi.it:2
axth34.cli.di.unipi.it:2
```

The syntax is pretty straightforward, first the machine host-name or IP address, followed by colon and then by the number of slots to reserve on that specific machine. Than to run the test you can simply type:

```
(mpi) $ mpirun -machinefile multicores -np 3 `pwd`/mpi/bin/python2.5-mpi \
        run.py par /tmp/matrix-3000-3000.mtx 3000x1500
3.8472688198 seconds to load the matrix
INFO:stencil:Skipping auto-derivation
INFO:stencil:After partition => Rows: 1 Cols: 2
INFO:stencil:Worker started on processor 1 axth15.cli.di.unipi.it
INFO:stencil:3.3783619404 seconds to scatter the matrix
INFO:stencil:Worker started on processor 2 axth27.cli.di.unipi.it
INFO:puzzle:0.0981070995 seconds to create auxiliary matrix
INFO:puzzle:0.1032311916 seconds to create auxiliary matrix
INFO:stencil:Worker 0: 27.8885238171 seconds to compute the partition
INFO:stencil:Sending back the computed sub-partition from 1
INFO:stencil:Worker 1: 29.1980512142 seconds to compute the partition
INFO:stencil:Sending back the computed sub-partition from 2
INFO:stencil:Worker 0: 2.8959200382 seconds to send back the partition
INFO:stencil:Worker 1: 3.1485359669 seconds to send back the partition
INFO:stencil:32.5039210320 seconds to gather the results
```

```
INFO:stencil:0.1175770760 seconds to reconstruct the result
INFO:stencil:Terminated.
36.0049271584 seconds to apply on 2 processors
```