## APPENDIX A
## APPLICATION COMMUNICATION PATTERNS

In this Appendix, we provide several examples of communication patterns; what is shown is the number of messages exchanged between processes. The rank numbers correspond to that of `MPI_COMM_WORLD`. Figure 5 shows the pattern for the CG kernel of the NAS benchmarks, Figure 6 shows FT and Figure 7 shows LU. For Zeus/MP, Fig. 8 shows the pattern for 64 processes.

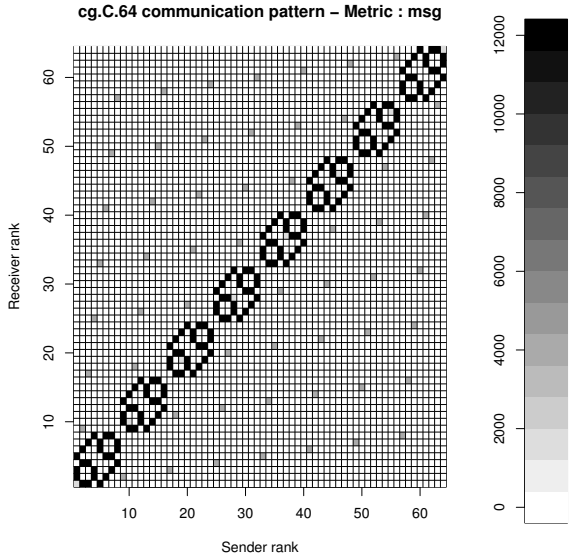



Fig. 5. Communication pattern of CG.C.64

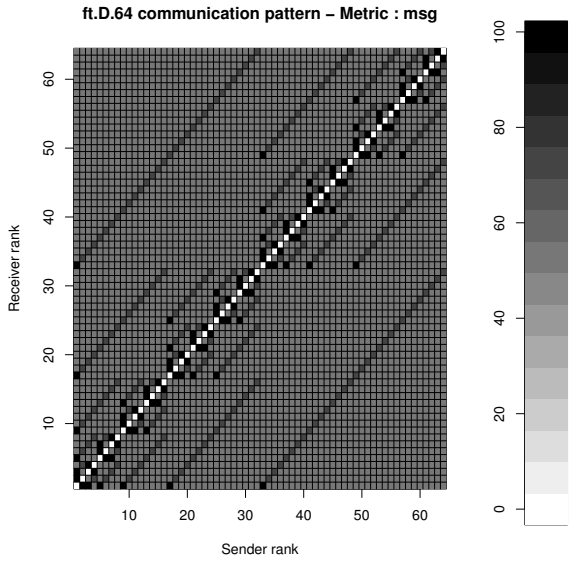



Fig. 6. Communication pattern of FT.D.64

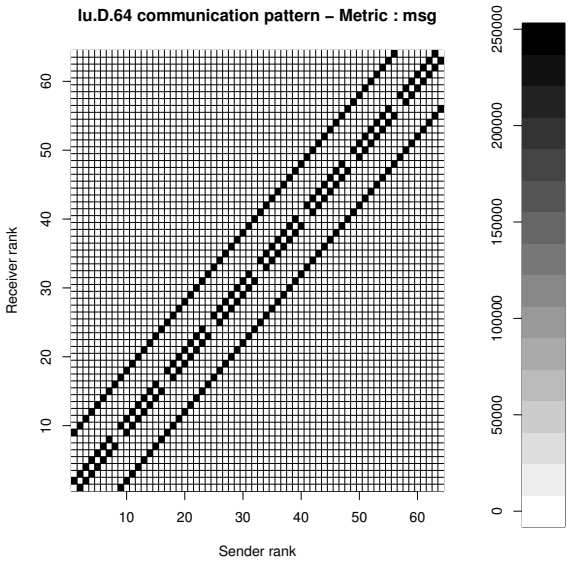



Fig. 7. Communication pattern of LU.D.64

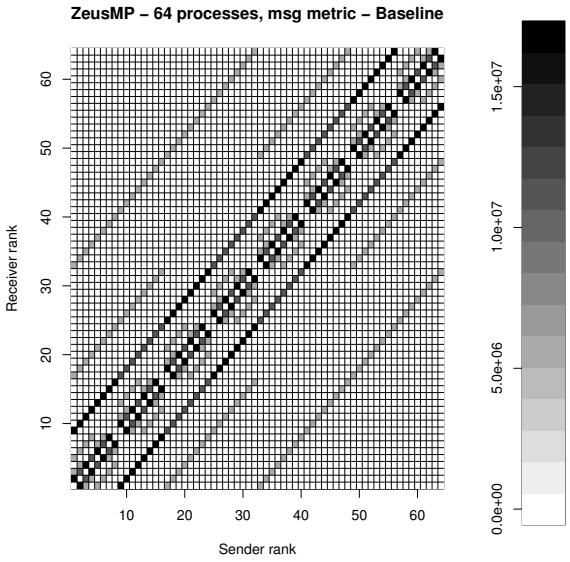



Fig. 8. Communication of Zeus/MP 64

## APPENDIX B
## PROCESS BINDING

When the processes are bound to computing units, the application is able to deliver more stable and predictable performance. Indeed, the standard deviation of the overall execution time is decreased, as shown in Table 1. When a process is not bound to a specific computing unit, the operating system scheduler may swap it to another computing unit, leading to cache misses that harm performance. However, as the scheduling of processes is not deterministic, the impact on the performance varies from one run to another. That is why the standard devi-

ation of several runs is lower when binding is enforced.

| Number of Iterations | No Binding of Processes | Binding of Processes | Improvement |
|---|---|---|---|
| 1000 | 0.077 | 0.089 | +15% |
| 2000 | 0.127 | 0.062 | -51% |
| 3000 | 0.112 | 0.097 | -13% |
| 4000 | 0.069 | 0.052 | -25% |
| 5000 | 0.289 | 0.121 | -58% |
| 10000 | 0.487 | 0.194 | -60% |
| 15000 | 0.24 | 0.154 | -36% |
| 20000 | 0.374 | 0.133 | -64% |
| 25000 | 0.597 | 0.247 | -59% |
| 30000 | 0.744 | 0.26 | -65% |
| 35000 | 0.78 | 0.3 | -61% |
| 40000 | 0.687 | 0.227 | -67% |
| 45000 | 0.776 | 0.631 | -19% |
| 50000 | 1.095 | 0.463 | -58% |

TABLE 1
Standard deviation figures for 10 runs of ZEUS-MP/2 CFD application with 64 processes (mhd blast case)

## APPENDIX C
## MODIFICATIONS TO LEGACY MPI SOURCE CODES

We modified legacy MPI applications to issue a call to the MPI 2.2 `MPI_Dist_graph_create` routine in order to create a new communicator (`comm_to_use`) in which the processes ranks are reordered. This call is made (and the reordering computed) just after the initialization step and before any application data are loaded into the MPI processes, otherwise data movements are necessary. Then, all relevant occurrences of `MPI_COMM_WORLD` are replaced by `comm_to_use` in the rest of the code. We provide as a commodity a routine (`read_pattern`) that gets the pattern from a trace file generated by a previous run of the target application[8].

### C.1 An example of C application: the IS NAS kernel

In the case of the IS NAS kernel, two new program arguments are used: the first one is the name of the pattern information file while the second argument is the metric to be used (`size`, `msg` or `avg`). Also, fourteen occurrences of `MPI_COMM_WORLD` have been replaced by `comm_to_use` in the rest of the code (file `is.c`).
Here is the complete modified code:

```
/*  Initialize MPI */
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &my_rank );
    MPI_Comm_size( MPI_COMM_WORLD, &comm_size );

    if(argc > 1){
      int  reorder = 1;

      if (my_rank == 0){
          int *sources      = NULL;
          int *degrees      = NULL;
          int *destinations = NULL;
          int *weights      = NULL;

          read_pattern(pattern_file,comm_size,
                       &destinations,&weights,
                       &sources,&degrees,metric_to_use);

          MPI_Dist_graph_create(MPI_COMM_WORLD,comm_size,
                                sources,degrees,
                                destinations,weights,
                                MPI_INFO_NULL,
                                reorder, &comm_topo);
          free(sources);
          free(degrees);
          free(destinations);
          free(weights);
        } else {
          MPI_Dist_graph_create(MPI_COMM_WORLD,0,
                                NULL,NULL,
                                NULL,NULL,
                                MPI_INFO_NULL,
                                reorder, &comm_topo);
      }

      if ( comm_topo != MPI_COMM_NULL )
         comm_to_use = comm_topo;
      else
        comm_to_use = MPI_COMM_WORLD;

      MPI_Comm_rank( comm_to_use, &my_rank);
    }
```

8. The users are supposed to provide such a pattern and feed it directly to the `MPI_Dist_graph_create` function through its `destinations`,`weights`, `sources` and `degrees` parameters.

### C.2 An example of Fortran application: ZEUS-MP/2

The ZEUS-MP/2 source code modifications follow the same scheme as the previous example:

1) A new variable `comm_to_use` has been introduced in the file `mod_files.F`.
2) The file `configure.F` has been modified to make the call to `MPI_Dist_graph_create` (as shown below).
3) MPI_COMM_WORLD occurrences have been replaced by `comm_to_use` in the following source files: `mstart.F` (seven occurrences), `rshock.F` (two occurrences), `setup.F` (14 occurrences), `marshak.F` (one occurrence), `restart.F` (two occurrences), `fftwplan.c` (two occurrences) and `fftw_ps.c` (two occurrences).

Here is an excerpt of the `configure.F` file demonstrating the use of the `MPI_Dist_graph_create` function:

```
#ifdef MPI_USED
c
c Reordering modifs
c
      character name*64, mode*1
      integer switch_comm
      integer num_degrees, numargs, newmode
      integer, allocatable, dimension(:) :: sources
      integer, allocatable, dimension(:) :: degrees
      integer, allocatable, dimension(:) :: destinations
      integer, allocatable, dimension(:) :: weights
#endif

c
c-----------------------------------------------------------
c     If parallel execution, start up MPI
c-----------------------------------------------------------
c
#ifdef MPI_USED
       call MPI_INIT( ierr )
       call MPI_COMM_RANK( MPI_COMM_WORLD, myid_w  , ierr )
       call MPI_COMM_SIZE( MPI_COMM_WORLD, nprocs_w, ierr )
c
c Reordering Modifs
c
       reorder = .true.
       switch_comm = 0

       numargs = iargc()
```

```
      if (numargs .eq. 2) then
         if(myid_w .eq. 0) then
            allocate(sources(nprocs_w))
            allocate(degrees(nprocs_w))
            num_degrees = 0
            call getarg(1, pattern_file)
            call read_pattern_fortran_get_degrees(num_degrees,
     >           pattern_file)

            allocate(destinations(num_degrees))
            allocate(weights(num_degrees))

            call getarg(2, metric_to_use)
            read( metric_to_use, '(i10)' ) newmode
            call read_pattern_fortran(nprocs_w, num_degrees,
     >           destinations(1), weights(1), sources(1),
     >           degrees(1),newmode,pattern_file)

            call MPI_DIST_GRAPH_CREATE(MPI_COMM_WORLD,
     >           nprocs_w,sources,degrees,destinations,
     >           weights, MPI_INFO_NULL,
     >           reorder,comm_topo,ierr)

            deallocate(sources,stat = ierr)
            deallocate(degrees,stat = ierr)
            deallocate(destinations,stat = ierr)
            deallocate(weights,stat = ierr )
         else
            call MPI_DIST_GRAPH_CREATE(MPI_COMM_WORLD, 0,
     >           0, 0, 0, 0,
     >           MPI_INFO_NULL,
     >           reorder,comm_topo,ierr)
         endif
         switch_comm = 1
      endif

      if(switch_comm .eq. 1) then
         comm_to_use = comm_topo
      else
         comm_to_use = MPI_COMM_WORLD
      endif

      call MPI_COMM_RANK(comm_to_use, myid_w, ierr)
      reorder = .false.
#else
      myid_w = 0
      myid   = 0
#endif /* MPI_USED */
```

# APPENDIX D
# DETAILS OF THE TREEMATCH ALGORITHM

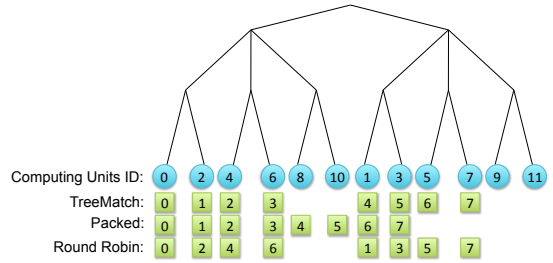
## D.1 Regular version of TREEMATCH

For TREEMATCH, we assume that the topology tree is balanced (leaves are all at the same depth) and symmetric (all the nodes of a given depth possess the same arity). Such assumptions are indeed very realistic in the case of a homogeneous parallel machine where all processors, sockets, nodes or cabinets are identical. In order to optimize the communication time of an application, the TREEMATCH algorithm will map processes to cores depending on the amount of data they exchange. The TREEMATCH algorithm is depicted in Algorithm 1, page 5.

To describe how the TREEMATCH algorithm works we will run it on the example given in Figure 9. Here, the topology is modeled by a tree of depth 4 with 12 leaves (computing units). The communication pattern between MPI processes is modeled by an $8 \times 8$ matrix (hence, we have eight processes). The algorithm processes the tree upward at depth 3. At this depth, we call the arity of the node of the next level $k$. In our case $k = 2$ and divides the order $p = 8$ of the matrix $m$. Hence,

| Proc | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1000 | 10 | 1 | 100 | 1 | 1 | 1 |
| 1 | 1000 | 0 | 1000 | 1 | 1 | 100 | 1 | 1 |
| 2 | 10 | 1000 | 0 | 1000 | 1 | 1 | 100 | 1 |
| 3 | 1 | 1 | 1000 | 0 | 1 | 1 | 1 | 100 |
| 4 | 100 | 1 | 1 | 1 | 0 | 1000 | 10 | 1 |
| 5 | 1 | 100 | 1 | 1 | 1000 | 0 | 1000 | 1 |
| 6 | 1 | 1 | 100 | 1 | 10 | 1000 | 0 | 1000 |
| 7 | 1 | 1 | 1 | 100 | 1 | 1 | 1000 | 0 |

(a) Communication Matrix




(b) Topology tree (squares represent mapped processes using different algorithms)

Fig. 9. Input example of the TREEMATCH algorithm

we directly go to line 6 where the algorithm calls the function `GroupProcesses`.

This information is given by a communication matrix, which can be determined as explained in Section 2.1.

**Function** GroupProcesses($T$,$m$,depth)

**Input**: $T$ //The topology tree
**Input**: $m$ // The communication matrix
**Input**: depth // current depth
1 $l \leftarrow$ListOfAllPossibleGroups($T$,$m$,depth)
2 $G \leftarrow$GraphOfIncompatibility($l$)
3 **return** IndependentSet($G$)

This function first builds the list of possible groups of processes. The size of the group is given by the arity $k$ of the node of the tree at the upper level (here 2). For instance, we can group process 0 with processes 1 or 2 up to 7 and process 1 with processes 2 up to 7 and so on. Formally we have $\binom{2}{8} = 28$ possible groups of processes. As we have $p = 8$ processes and we will group them by pairs (k=2), we need to find $p/k = 4$ groups that do not have processes in common. To find these groups, we will build the graph of incompatibilities between the groups (line 2). Two groups are *incompatible* if they share a process (e.g., group (2,5) is incompatible with group (5,7) as process 5 cannot be mapped at two different locations). In this graph of incompatibilities, vertices correspond to the groups and we have an edge between two vertices if the corresponding groups are incompatible. In the literature, such a graph is referred to as the complement of a Kneser Graph [41]. The set of groups we are looking for is thus an *independent set* of this graph. A valuable property of the complement of the Kneser graph is that since $k$ divides $p$, any maximal independent set is maximum and of size $p/k$. Therefore, any greedy algorithm always finds an independent set of the required size. However, all grouping of processes (i.e., independent sets) are not of equal quality. They

depend on the values of the matrix. In our example, grouping process 0 with process 5 is not a good idea as they exchange only one piece of data and if we group them we will have a lot of remaining communication to perform at the next level of the topology. To account for this, we valuate the graph with the amount of communication reduced thanks to this group. For instance, based on the matrix $m$, the sum of communication of process 0 is 1114 and of process 1 is 2104 for a total of 3218. If we group them together, we will reduce the communication volume by 2000. Hence, the valuation of the vertex corresponding to group (0,1) is 3218-2000=1218. The smaller the value, the better the grouping. Unfortunately, finding such an *independent set* of minimum weight is NP-Hard and inapproximable at a constant ratio [42]. Therefore, we use heuristics to find a "good" independent set:

- **smallest values first**: we rank vertices by smallest values first and we build a maximal independent set greedily, starting with the vertices with smallest values.
- **largest values last**: we rank vertices by smallest values first and we build a maximal independent set such that the largest index of the selected vertices is minimized.
- **largest weighted degrees first**: we rank vertices by their decreasing weighted degrees (the average weight of their neighbors) and we build a maximal independent set greedily, starting with the vertices with largest weighted degrees [42].

In our implementation we start with the first method and try to improve the solution by applying the last two. We can use a user-defined threshold value to disable the weighted degrees technique when the number of possible groups is too large.

In our case, regardless of the heuristic we use, we find the independent set of minimum weight, which is {(0,1),(2,3),(4,5),(6,7)}. This list is affected to the array group[3] in line 6 of the TREEMATCH algorithm. This means that, for instance, process 0 and process 1 will be put on leaves sharing the same predecessor.

**Function** AggregateComMatrix($m$,$g$)

```
   Input: m // The communication matrix
   Input: g // list of groups of (virtual) processes to merge
1  n ← NbGroups(g)
2  for i ← 0..(n − 1) do
3      for j ← 0..(n − 1) do
4          if i = j then
5              r[i, j] ← 0
6          else
7              r[i, j] ← Σ_{i1∈g[i]} Σ_{j1∈g[j]} m[i1, j1]
8  return r
```

Then, we build the groups at depth 2, but we need to aggregate the matrix $m$ with the remaining communication beforehand. The aggregated matrix is computed in the `AggregateComMatrix` Function. The goal is to compute the remaining communication between each group of processes. For instance, between the first group (0,1) and the second group (2,3) the amount of communication is 1012 and is put in $r[0,1]$ (see Figure 10(a)). The matrix $r$ is of size $4 \times 4$ (we have four groups) and is returned to be affected to $m$ (line 7 of the TREEMATCH algorithm). Now, the matrix $m$ corresponds to the communication pattern between groups of processes (called virtual processes) built during this step. The goal of the remaining steps of the algorithm is to group these virtual processes up to the root of the tree.

| Virt. Proc | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 0 | 1012 | 202 | 4 |
| 1 | 1012 | 0 | 4 | 202 |
| 2 | 202 | 4 | 0 | 1012 |
| 3 | 4 | 202 | 1012 | 0 |

(a) Aggregated matrix (depth 2)

| Virt. Proc | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 0 | 1012 | 202 | 4 | 0 | 0 |
| 1 | 1012 | 0 | 4 | 202 | 0 | 0 |
| 2 | 202 | 4 | 0 | 1012 | 0 | 0 |
| 3 | 4 | 202 | 1012 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |

(b) Extended matrix

| Virt. Proc | 0 | 1 |
|---|---|---|
| 0 | 0 | 412 |
| 1 | 412 | |

(c) Aggregated matrix (depth 1)

Fig. 10. Evolution of the communication matrix at different steps of the algorithm

**Function** ExtendComMatrix($T$,$m$,depth)

```
   Input: T //The topology tree
   Input: m // The communication matrix
   Input: depth // current depth
1  p ← order of m
2  k ←arity(T,depth+1)
3  return AddEmptyLinesAndCol(m,k,p)
```

The algorithm then loops and decrements depth to 2. Here, the arity at depth 1 is 3 and does not divide the order of $m$ (4). Hence, we add two artificial groups that do not communicate with any other groups. This means that we add two lines and two columns full of zeroes to the matrix $m$. The new matrix is depicted in Figure 10(b). The goal of this step is to allow more flexibility in the mapping, thus yielding a more efficient mapping.

Once this step is performed, we can group the virtual processes (group of processes built in the previous step). Here the graph modeling and the independent set heuristics lead to the following mapping: {(0,1,4),(2,3,5)}. Then we aggregate the remaining communication to obtain a $2 \times 2$ matrix (see Figure 10(c)). During the next loop (depth=1), we have only one possibility to group the virtual processes: {(0,1)}, which is affected to group[1].

The algorithm then goes to line 8. The goal of this step is to map the processes to the resources. To perform this task, we use the groups array, which describes a hierarchy of groups of processes. A traversal of

this hierarchy gives the process mapping. For instance, virtual process 0 (resp. 1) of group[1] is mapped on the left (resp. right) part of the tree. When a group corresponds to an artificial group, no processes will be mapped to the corresponding sub-tree. At the end, processes 0 to 7 are mapped to leaves (computing units) 0,2,4,6,1,3,5,7, respectively (see bottom of Figure 9(b)). This mapping is optimal. The algorithm provides an optimal solution if the communication matrix corresponds to a hierarchical communication pattern (processes can be arranged in a tree, and the closer they are in this tree the more they communicate), that can be mapped to the topology tree (such as the matrix of Figure 9(a)). In this case, optimal groups of (virtual) processes are automatically found by the independent set heuristic, as the corresponding weights of these groups are the smallest among all the groups. Moreover, thanks to the creation of artificial groups (line 5), we avoid the *Packed* mapping 0,2,4,6,8,1,3, which is worse as processes 4 and 5 communicate a lot with processes 6 and 7 and hence must be mapped to the same sub-tree. On the same figure, we can observe that the *Round Robin* mapping, which maps process $i$ on computing unit $i$, leads also to a suboptimal result.

The main cost of the algorithm is in the function `GroupProcesses`. If $k$ is the arity of the next level and $p$ is the order of the current communication matrix, the complexity of this part is proportional to the number of $k$-sized groups among a set of $p$ elements and this number is $\binom{p}{k} = O(p^k)$.

### D.2 Optimizations

| $k$ | $p^*$ | $d^*$ |
|---|---|---|
| 4 | 8 | 2 |
| 6 | 10 | 3 |
| 8 | 12 | 4 |
| 9 | 14 | 3 |
| 10 | 14 | 5 |
| 12 | 16 | 6 |
| 14 | 18 | 7 |
| 15 | 20 | 5 |
| 16 | 20 | 8 |
| 18 | 22 | 9 |
| 20 | 24 | 10 |
| 21 | 26 | 7 |
| 22 | 26 | 11 |
| 24 | 28 | 12 |
| 25 | 32 | 5 |
| 26 | 30 | 13 |
| 27 | 32 | 9 |
| 28 | 32 | 14 |
| 30 | 34 | 15 |
| 32 | 36 | 16 |
| 33 | 38 | 11 |
| 34 | 38 | 17 |
| 35 | 42 | 7 |
| 36 | 40 | 18 |
| 38 | 42 | 19 |
| 39 | 44 | 13 |
| 40 | 44 | 20 |
| 42 | 46 | 21 |
| 44 | 48 | 22 |
| 45 | 50 | 15 |
| 46 | 50 | 23 |
| 48 | 52 | 24 |
| 49 | 58 | 7 |
| 50 | 54 | 25 |
| 51 | 56 | 17 |
| 52 | 56 | 26 |
| 54 | 58 | 27 |
| 55 | 62 | 11 |
| 56 | 60 | 28 |
| 57 | 62 | 19 |
| 58 | 62 | 29 |
| 60 | 64 | 30 |
| 62 | 66 | 31 |
| 63 | 68 | 21 |
| 64 | 68 | 32 |
| 65 | 72 | 13 |
| 66 | 70 | 33 |
| 68 | 72 | 34 |
| 69 | 74 | 23 |
| 70 | 74 | 35 |
| 72 | 76 | 36 |
| 74 | 78 | 37 |
| 75 | 80 | 25 |
| 76 | 80 | 38 |
| 77 | 86 | 11 |
| 78 | 82 | 39 |
| 80 | 84 | 40 |
| 81 | 86 | 27 |
| 82 | 86 | 41 |
| 84 | 88 | 42 |
| 85 | 92 | 17 |
| 86 | 90 | 43 |
| 87 | 92 | 29 |
| 88 | 92 | 44 |
| 90 | 94 | 45 |
| 91 | 100 | 13 |
| 92 | 96 | 46 |
| 93 | 98 | 31 |
| 94 | 98 | 47 |
| 95 | 102 | 19 |
| 96 | 100 | 48 |
| 98 | 102 | 49 |
| 99 | 104 | 33 |
| 100 | 104 | 50 |
| 102 | 106 | 51 |
| 104 | 108 | 52 |
| 105 | 110 | 35 |
| 106 | 110 | 53 |
| 108 | 112 | 54 |
| 110 | 114 | 55 |
| 111 | 116 | 37 |
| 112 | 116 | 56 |
| 114 | 118 | 57 |
| 115 | 122 | 23 |
| 116 | 120 | 58 |
| 117 | 122 | 39 |
| 118 | 122 | 59 |
| 119 | 128 | 17 |
| 120 | 124 | 60 |
| 122 | 126 | 61 |
| 123 | 128 | 41 |
| 124 | 128 | 62 |
| 125 | 132 | 25 |
| 126 | 130 | 63 |
| 128 | 132 | 64 |

TABLE 2
Table of optimal tree division. Given a node of arity $k$ it tells above which number of processors $p^*$ it is useful to divide this node into $d^*$ nodes of arity $k/d^*$. It is based on the fact that $\forall p \geq p^*, \binom{p}{k} > d^* \binom{p}{k/d^*}$.

## APPENDIX E
## BANDWIDTH MEASUREMENTS BETWEEN COMPUTING UNITS

In Figure 11 we present timings measuring the bandwidth obtained for transmitting a given number of bytes on PLAFRIM using the NetPIPE[9] tool. Three cases are shown, depending on whether the sender and the receiver share the same L3 cache, share a memory bank, or are on different nodes. We see that the bandwidth is always the highest for the first case and always the lowest for the last cases. Moreover, we see that the performance is not linear (and not affine) in any cases. Overall, these experiments strengthen the structural models exploited by TREEMATCH compared to the quantitative approach followed by other tools such as Scotch.

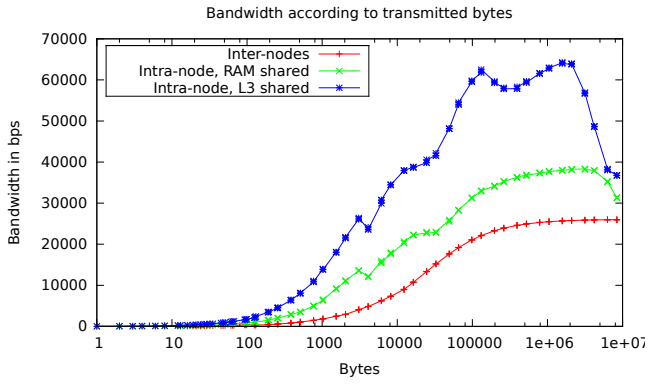



Fig. 11. ZEUS-MP/2 (metric: msg, 256 processes)

## APPENDIX F
## DISCUSSION ON THE TLEAF STRUCTURE OF SCOTCH

To represent architectures, Scotch uses several formats. When dealing with a hierarchical, tree-structured topology, Scotch proposes the *tleaf* built-in definition. A tleaf file is a single line file with the following syntax:
**tleaf** $n\ a_0\ v_0\ \ldots a_{n-1}\ v_{n-1}$

There are $n+1$ levels in the tree (numbered from 0 to $n$). The leaves are at the level $n$. $a_i$ $(0 \leq i \leq n-1)$ is the

9. http://www.scl.ameslab.gov/netpipe/

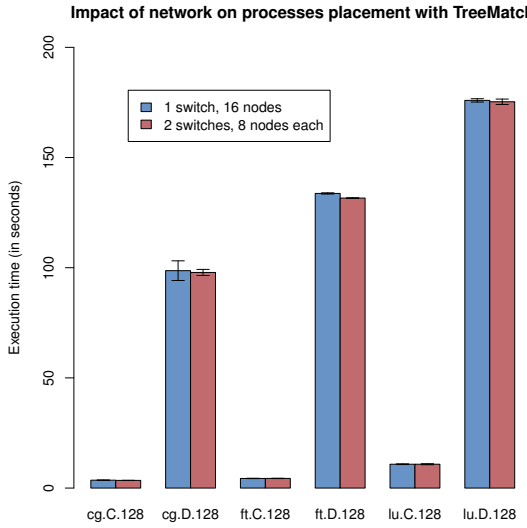



Fig. 12. Network fragmentation

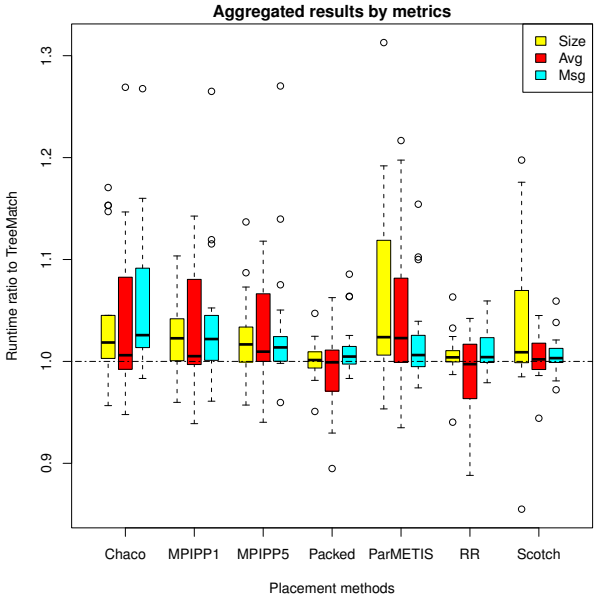



Fig. 13. Average execution time ratio between TREEMATCH and other placement methods for the NAS benchmarks. Results projected by metrics (number of messages, data size or average message size)

arity of the $i^{\text{th}}$ level and $v_i$ ($0 \leq i \leq n-1$) is the traversal cost between level $i$ and $i+1$.

In our 128 computing unit experiments, we used two tleaf structures:

- `tleaf 4 16 4 2 3 2 2 2 1` for the *scotch* case and,
- `tleaf 4 16 500 2 100 2 50 2 10` for the *scotch_w* case.

Each tleaf has the same structure: five levels with arity of intermediate nodes being 16, 2, 2 and 2. Only costs between the levels change. However, this is mainly a change in orders of magnitude. Nevertheless, the scotch_w tleaf leads to worse results than the *scotch* one for the ZEUS/MP experiments as shown in Fig. 3.

## APPENDIX G
## EXPERIMENTAL VALIDATION: EXTENDED RESULTS

### G.1 Impact of network fragmentation

In Figure 12, we present the timings of several NAS kernels on 128 computing units (16 nodes) for both classes C and D. Two cases are studied. In the first, all the nodes are on the same switch. In the second, computing nodes are dispatched evenly on two switches, incurring more costly communications. The solution computed by TREEMATCH is the same in both cases. Results show that, nevertheless, the impact on the performance for these two cases is small, meaning that here, flattening the network does not significantly hinder the result.

### G.2 NAS parallel benchmarks results

In Section 5.2.2, we show the projection of the results for the NAS kernels. In this section, we provide projections for other parameters: the various metrics (Figure 13), the various classes (Figures 15 and 14) and the various process counts considered (Figure 16).

In Figure 13, we show the ratios for the three metrics: the amount of exchanged data (*Size*), the number of messages (*Msg*), and the average size of exchanged messages (*Avg*). We see that except for the *Avg* metric for *Packed* and *RR* all the medians are above 1. For the *Size* and *Msg* metrics, almost 75% of the ratios are above one. Gains of more than 10% in execution times are commonplace while we lose more than 10% only in marginal cases.

The worst results are obtained when *RR* and *Packed* policies are compared to TREEMATCH using the *Avg* metric. The actual execution times are in general higher for this metric than for the *Size* and *Msg* ones. Indeed, *RR* and *Packed* methods do not depend on any metric as they do not use the communication matrix but only the computing units numbering. Hence, it is generally better not to use the average message size metric for computing the process placement. For this reason, we have excluded this metric from Figures 14, 15, and 16.

In Figure 14, we see that the ratio over the other metrics decreases when we increase the class (i.e., the problem size). Indeed, the tested kernels are compute-intensive ones, meaning that when increasing the input size, the computation time grows faster than the communication time. As the process placement helps in improving the communication time, the gain is in proportion less for large inputs than for small inputs. However, if we display the difference between TREEMATCH and the other methods we can see that the gain (in terms of difference) is increasing along with the input size, as

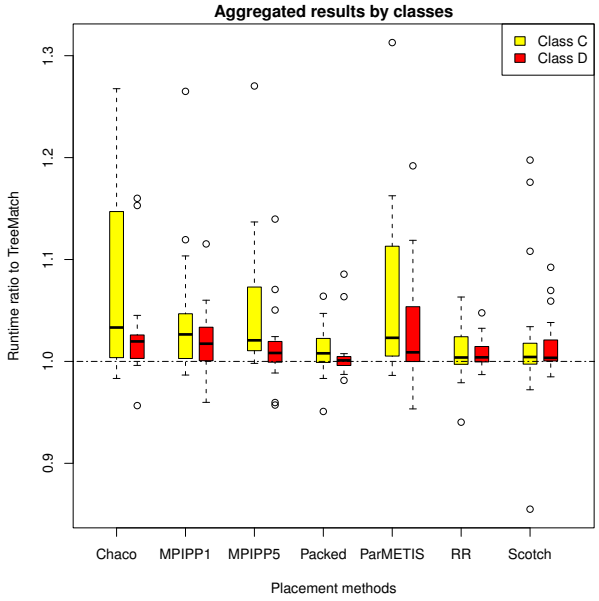



Fig. 14. Average execution time ratio between TREEMATCH and other placement methods for the NAS benchmarks. Results projected by Class (C: average problem size, D: large problem size). Metric Avg excluded

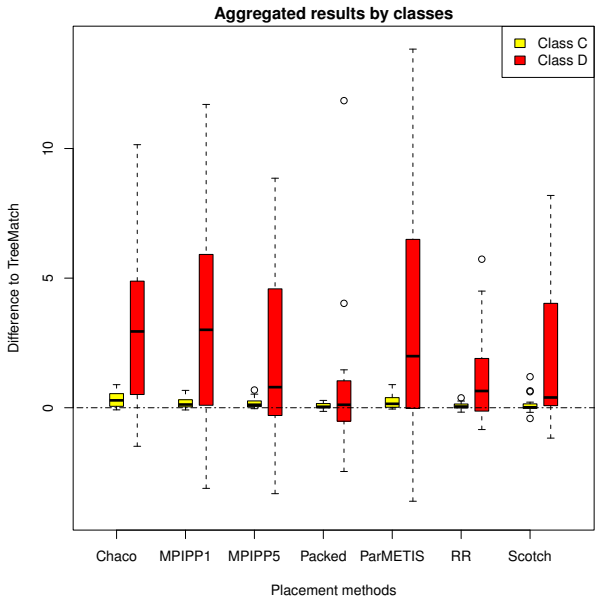



Fig. 15. Average execution time difference between TREEMATCH and other placement methods for the NAS benchmark. Results projected by Class (C: average problem size, D: large problem size). Metric Avg excluded.

shown in Figure 15.

In Figure 16, we see what happens when the number of processes increases. The main result from this figure is that the discrepancy of the ratios tends to increase along with the number of processes. Actually, most of the cases with a ratio greater than one tend to keep a ratio greater than one when we change the number

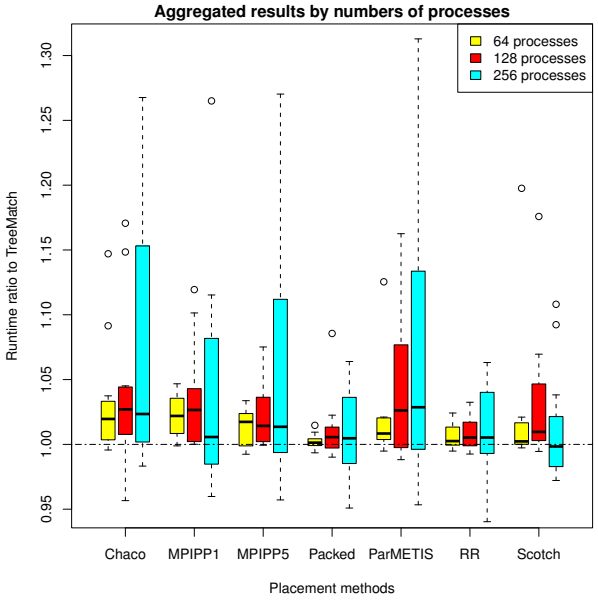



Fig. 16. Average execution time ratio between TREEMATCH and other placement methods for the NAS benchmarks. Results projected by number of processes (64, 128, 256). Metric Avg excluded.

of processes. One of the explanations for the distance to 1.0 increasing with the number of processes is that the computation to communication ratio decreases and effects due to communication are therefore more visible with higher numbers of processes.

Other noticeable results are that MPIPP1 is worse than MPIPP5 whilst Chaco is the worst method of all. Last, ParMETIS does not guarantee that the computed partitions are of equal sizes because it is not designed to produce such partitions. This would explain the poor results achieved with this partitioner.