SPMD Image Processing on Beowulf Clusters: Directives and Libraries

Paulo Oliveira and Hans du Buf
Vision Laboratory, University of Algarve
Campus de Gambelas – FCT, Faro, Portugal
dubuf@ualg.pt

Abstract

Most image processing algorithms can be parallelized by splitting parallel loops and by using very few communication patterns. Code parallelization using MPI still involves much programming overheads. In order to reduce these overheads, we first developed a small SPMD library (SPMDlib) on top of MPI. The programmer can use the library routines themselves, because they are easy to learn and to apply, even without knowing MPI. However, in order to increase user-friendliness, we also develop a small set of parallelization and communication directives/pragmas (SPMDdir), together with a parser that converts these into library calls. SPMDdir will be used to develop a new version of SPMDlib. This new version will contain much less but generic routines that can be optimized for different network topologies. Extensions for Fortran 90/95 and C will be discussed, as well as communication optimizations.

1. Introduction

Parallel processing has become a well-established area in computer science, although most researchers never had any opportunity to "play" with the necessary technology. Nowadays every laboratory, however small, can install a Beowulf cluster that consists of normal, cheap PCs and some interconnect, at least 100 Mb/s ethernet and one switch.

Installing clusters, Linux and MPI is rather trivial [6], but using MPI is not because of its complexity and huge functionality [4]. Programmers familiar with SGI's directive `C$DOACROSS` or the new standard OpenMP (`C$OMP`, see `www.openmp.org`) on expensive SMP and NUMA architectures find MPI programming rather archaic. Especially in image processing, where many parallel loops can be split (which is `C$DOACROSS`'s only function), but supplemented with a very few communication patterns, the simplicity of using a few parallelization and communication directives can lead to a tremendous increase in programming efficiency.

In our view, the most important keyword concerning parallel programming is *simplicity*. Image processing is an ideal area for applying the SPMD model. Arrays can easily be distributed over participating nodes and DO loops on the nodes can be adjusted such that each one does its share of the work. There are a few complications, like spatial neighborhood operations (e.g. a convolution) in which each node needs some data from its neighbors, but these can be easily solved. Although using a small subset of MPI or PVM, or the easier BSP model, is not problematic, the programmer is always obliged to do bookkeeping in terms of addresses and numbers of bytes (reals, integers). As will be shown, it is straightforward to hide all this from the programmer. Our approach is much easier than other efforts towards parallel image processing, it provides a high-level solution, and it can be applied on any system with MPI support. In addition, we aim at developing a tool which is (a) low-cost in terms of price and usage, (b) very easy to use, and (c) which preserves code portability. The latter points imply that the same program can be executed on one CPU (no communications at all), on SMP systems (shared memory with very efficient "communications" using `C$DOACROSS` or `C$OMP`) or on clusters (distributed memory with very expensive communications using our `C$SPMD`), depending on the context.

We will mention a few other approaches to parallelizing image processing. Lückenhaus and Eckstein [2] apply an agent-based approach with the thread concept on Sun SMP systems. Using task graphs, three processing models can be realized: data parallel (e.g. pixels), task parallel (independent tasks like serial code blocks) and pipelining (e.g. applying the same processing steps to an image sequence). Squyres et al. [5] describe the PIPT or Parallel Image Processing Toolkit. Instead of explicitly parallelizing each routine, a small set of basic operations is provided such that users can easily define a window operator, for example, which will be "embedded" in a parallelized routine. Gennart and Hersch [1] describe a Computer-Aided Parallelization (CAP) tool. CAP is a precompiler that generates C++ source code. It enables programmers to specify at a high

level of abstraction threads by using the split-merge concept. Seinstra et al. [3] use the C++ function template mechanism to create generic algorithms on the basis of many basic and pre-defined functions like add, sub, mult and pow that the user should include in function calls.

One common aspect of these efforts is to hide parallelization details from the programmer: *automatic* parallelization still is one of the Holy Grails of computer science. However, *efficient* parallelization, especially on distributed-memory systems, requires that the programmer must have complete control over all communications in order to minimize communication overheads: each *millisecond* spent with communications implies a loss of about 6 MFLOPs on a cluster of 8 AMD Athlon XPs (1.5 GHz). This is the reason that we extend parallelization directives with a few communication directives: we aim at both programming and execution efficiency.

Below we will first discuss our SPMD initiative, and then explain the directive set, the parser, and the first library implementation. We will conclude this paper by describing new library implementations and some concepts for Fortran 90/95 and C, including communication optimizations.

2. The SPMD initiative

The main goal is to develop a parser or preprocessor that translates parallelization and communication directives (SPMDdir) into library calls (SPMDlib). The limited functionality and simplicity of SPMDlib leads to a very small set of directives: the one-to-one mapping and modularity of both library and parser has two important advantages: (1) in contrast to the other efforts mentioned in the Introduction, the development will not cost many person-years, and (2) future extensions can be easily integrated. Additional advantages are: (3) although the programmer should know about communication costs and parallelization granularity, he/she must not necessarily know MPI; (4) once that we have a working parser version, we can optimize the library and even the parser itself. Like BERT (`www.plogic.com`), the parser could be made intelligent in order to instruct the programmer about e.g. the number of nodes to use.

The simplicity of our approach implies a restricted functionality. However, most image processing operations can be parallelized, including histogramming, classification, connected component labeling, segmentation, iterative spatial filterings and even the application of filter banks in the frequency domain for texture analysis. Advanced features allow dynamic loop scheduling for load balancing and even pipelining. In the ideal case one node (called root) will read an image file into an array. It will then distribute equal array parts to other nodes, and all nodes, including root, will process their array parts. After all processing has been completed, the other nodes will send their parts to root, which will assemble them and then write an output file.

The underlying idea of SPMDlib is to parallelize loops over arrays, i.e. 2D image arrays, but 1D and 3D arrays, or even nD arrays, can be processed in the same way. SPMD implies that all participating nodes are executing the same program, but each node has an identifier and knows the total number of nodes ($0 \leq$ myid $\leq$ nprocs $- 1$). Also knowing the loop size, each node can easily compute its part, i.e. `mystart` and `myend`. The only restriction of SPMDlib is that (in Fortran) parallel loops over arrays must start with 1 and have a stride of 1 (in the case of nested loops over multidimensional arrays only the *outer* loop over the *last* dimension must be parallelized). The implicit array partitioning allows to hide all bookkeeping from the programmer, and in most applications all nodes will equally share the work load. When the outer loop/array size is not a multiple of `nprocs`, for example a large prime number, there are several solutions: (a) let the last node do less work, (b) transpose the array, maybe the other dimension is more suitable, (c) extend the array with a few blank lines, or (d) use for this loop fewer nodes, i.e. put to "sleep" temporarily some nodes. Below we include a solution for dynamic loop scheduling (balancing), but this will involve more communications and, as for most of the other solutions, will be less suitable for fine-grain parallelization. The simplest and most efficient solution is to split the work equally using the outer loops over the last array dimensions, and to reduce communications to an absolute minimum.

Apart from pipelining, we only use a virtual star topology in which "root" distributes and collects data. Because it may be neccessary to distribute only parts of arrays or entire arrays, both possibilities are included in *root-to-all* communications. However, only array parts are involved in *all-to-root* communications (an *all-to-root* of entire arrays is called an array reduction; see below). In addition, when all nodes have processed their array parts, it may be necessary that all nodes need the entire array: *all-to-all* communications can be more efficient[1] than combining an *all-to-root* of array parts with a *root-to-all* of the entire array. Special versions of *root-to-all* and *all-to-all* communications are provided for spatial filterings and other neighborhood operations, i.e. to provide extra neighborhoods to nodes or only exchange neighborhoods between nodes.

Normal reductions concern scalars: maximum, sum, etc. Here there are two options: *reduction-on-root* (only root needs the global) and *reduction-on-all* (all nodes need the global). In addition to scalar reductions there are array reductions: it may be necessary to combine entire arrays, e.g. to sum elementwise in histogramming or to combine arrays using logical operators AND or OR.

[1] Efficiency also depends on the network usage: multicast routing is ideal for root-to-all communications!

3. SPMD directive set

In this Section we explain the main directive clauses in relation to the SPMDlib calls; the latter will be explained in the following section. Directives for Fortran start in the first column and have the form

```
C$SPMD clause[, clause] ...
```

in which the clauses are executed from left to right (depending on the context). Clauses can be continued on multiple lines, but each line starts with `C$SPMD` and clauses are separated by comma-blank. `C$SPMD` directives should not be "mixed" with `C$DOACROSS` or `C$OMP` directives; although these are allowed, they will be ignored. The clauses and their meaning are (advanced features are marked by an *; these will not be discussed in this paper):

```
INIT          initialize SPMDlib and MPI
END           finalize SPMDlib and MPI
BARRIER       node synchronization
RONLY         root only does
TIME, TTIME   time and total time
PARBLOCK      parallel code blocks (*)
DOPAR         parallel DO loop
DODYN[...]    dynamic DO, chunk size
PIPE[...]     node pipelining (*)
R2A[list]     root-to-all
A2R[list]     all-to-root
A2A[list]     all-to-all
RoR[list]     reduction-on-root
RoA[list]     reduction-on-all
```

in which `list` is a list of scalars and array names (only names!); the latter will be indicated by `<an>` below. The following abbreviations and combinations are used in some clauses:

```
TCPUS=n       total number nodes (*)
NCPUS=n       nodes to be used (*)
E:<an>        entire array
P:<an>        parts of array
P(X=n):<an>   exchange n elements/
              lines/planes 1/2/3D arrays
P(XC=n):<an>  cyclic exchange
SUM:variable  type of scalar or array
              reduction: MAX,MIN,AND...
IN=variable   input scalar or array
              in reduction
OUT=variable  output scalar or array
              in reduction
```

INIT and END should be used in the main program or in one subroutine. All other clauses, except for DODYN and PIPE, can be used in other subroutines, provided that all nodes call these.

3.1. Brief explanation

SPMDlib consists of a few subroutines for initialization (`spmd_init`), synchronization (`spmd_barrier`) and for splitting loops and arrays (`spmd_split`). Most routines are for communications. There also is an include file (`spmd.h`) with predefined and prefixed variables and a common block (in Fortran) to pass parameters to communication routines. Below follows a brief description of most directives in relation to the library.

INIT invokes `spmd_init(ierr)` that provides predefined scalars `spmd_myid`, `spmd_nprocs`, `spmd_mystart`, `spmd_mypart` and `spmd_myend` (all integers), as well as prefixed scalars for timing purposes (normal reals). Initializes `spmd_nprocs` and `spmd_myid` (`0` to `spmd_nprocs-1`). Root will be number 0. The programmer is free to use the predefined scalars mentioned above, although code portability will be lost.

END invokes `spmd_end(ierr)`. Must be included before stop/end.

BARRIER invokes `spmd_barrier(ierr)` and must be used where an explicit node synchronization is required, for example when root only (see RONLY below) needs a significant CPU time. Note that all communications and reductions cause an implicit node synchronization because of the synchronous `mpi_send` and `mpi_recv` used in the library. If BARRIER is used together with a closing RONLY, the program will not be deadlocked because it will be executed after the RONLY code (the parser will detect a BARRIER inside two RONLYs and will print a warning).

RONLY is a switch to define regions in which Root-ONLY is active, for example to prepare or output data (read/write a file, print); the other nodes will skip this region and continue the execution of the subsequent code until they encounter a barrier or a communication clause. Must be inserted before and after the code lines (or only once for timing, see TIME! below), and invoke only an `if-endif`:

```
C$SPMD RONLY
      call readfile(...,array,dim)
      call normalize(...,array,dim)
C$SPMD RONLY, R2A[P:array]
```

becomes

```
      if (spmd_myid.eq.0) then
      call readfile(...,array,dim)
      call normalize(...,array,dim)
      end if
      call spmd_split(dim)
      call spmd_r1DPtoall(array,dim)
```

This example also shows a *root-to-all* clause `R2A` and the associated routine `spmd_r1DPtoall` in the case of a real,

1D array declared with dimension `dim`, the `P` meaning that root must distribute array parts. The parser will also insert the `spmd_split` call if (a) this was not done before or (b) the dimension was different. All communication and reduction routines synchronize all nodes, hence no barrier is necessary. If the CPU time is large and the region not immediately followed by e.g. an R2A directive, the second RONLY can be combined with BARRIER.

TIME and **TTIME** are for timing purposes. TTIME is for larger code blocks that contain TIME directives. Both must be combined with RONLY. The timer starts or stops on the RONLY line that contains (T)TIME, i.e. the work to be done by root only can be excluded or included. If an RONLY does not involve other work for root, then RONLY, TIME! or RONLY, TTIME! must be used. Both must be used pairwise, i.e. they imply start-stop, and both return predefined normal reals `spmd_time` and `spmd_ttime` that the programmer can use. A future version may include an additional clause for telling the parser what code to include in order to print a message. For example:

```
C$SPMD RONLY, TIME!
      code block to be timed
C$SPMD RONLY, TIME!,
C$SPMD PRINT[Code block took:,spmd_time]
```

would be translated into

```
      if (spmd_myid.eq.0) call spmd_rtime
      code block to be timed
      if (spmd_myid.eq.0) then
      call spmd_rtime
      print*,'Code block took:',spmd_time
      end if
```

which could include some arithmatic to convert seconds to e.g. milliseconds.

DOPAR must be inserted directly before a parallel DO loop, not counting empty or comment lines (e.g. `C$OMP`). Must always be the outer loop over 2D/3D arrays because it invokes `spmd_split(dim)`, in which `dim` is the outer loop/array dimension, to compute `spmd_mystart`, `spmd_mypart` and `spmd_myend` on all nodes. The parser will change the loop `do i=1,dim` to `do i=spmd_mystart,spmd_myend` and the three parameters (start,part,end) will be passed (via common block) to communication routines involving array parts. Note that root (node number 0) equally contributes and will also process array parts, i.e. the first parts. Since DOPAR implies a static and equal loop scheduling, the user can put a BARRIER after the loop if the loads are not balanced (not necessary if the loop is immediately followed by a communication or reduction). The routine `spmd_split(dim)` will check that `dim` is a multiple of `spmd_nprocs`, and will print a warning message if this is not the case (i.e. at runtime; the parser can also be instructed—using TCPUS—to check this).

DODYN[chunk,array] is for dynamic scheduling of an outer DO loop, in which each node will start with a "chunk" and, after sending its chunk of data to root, will receive a request for doing another chunk, until the loop has been completed. Because of the irregular communication pattern, root will only distribute chunks (start numbers of the loop) and collect results (corresponding array chunks); all non-root nodes must have all necessary data. The chunk size depends on the load imbalance, and should be smaller than the (outer) loop dimension divided by the number of non-root nodes. A smaller chunk size implies more communication calls, but each with less data. The parser will put all necessary code lines and library calls before and after the loop. A typical application is Direct Volume Rendering in which the root node has an OpenGL window for displaying a rotating object (in this case we use the frontend of the cluster as root, *not* using `mpirun -nolocal`). Note: DODYN has not yet been implemented in the library, but on the basis of our experience we know that its implementation is rather straightforward using a few routines and predefined integers in the `spmd.h` include file.

3.2. Communications and reductions

All communications are done by **R2A** (root-to-all), **A2R** (all-to-root) and **A2A** (all-to-all), indicating scalars, entire arrays (E:) or parts of arrays (P:), but without any types and array dimensions. Example:

```
C$SPMD R2A(val1,val2;P:array1)
```

Already implemented are integer, real, double and complex types for 1D and 2D arrays. The following restrictions and extensions apply:

1. Array names in R2A, A2R and A2A must be preceded by E: (Entire) or P: (Parts), like P:arrayname. This is not necessary in RoR and RoA reductions.
2. By default arrays are split into equal parts, the number of which corresponds to the number of nodes specified on the command line (e.g. `mpirun -np 8`).
3. The variable list is a list of scalars and arrays separated by commas and semicolons like: scal1,scal2;P:arr1;E:arr2,arr3
4. Array communications: R2A can be P or E. A2R must be P. A2A with E implies that all nodes having only one part will receive all parts (the entire array).
5. Spatial neighborhood operations: in the case of R2A and A2A, P can be combined with (X=n), X meaning eXchange, or (XC=n), XC meaning eXchange Cyclic. Example: R2A[P(XC=3):arr1;P:arr2], where instead of "3" we can use an integer scalar. R2A will distribute the normal array parts of `arr1` plus the neighboring elements/lines for a first iteration; A2A will only exchange neighboring elements/lines necessary for multiple iterations of window operations.

Scalar reductions can be done using only two clauses: (1) **RoR** (Reduction-on-Root), where only root gets a global value; the other nodes will continue with their local values. (2) **RoA** (Reduction-on-All), where all nodes will continue with the global value. Reduction types: we will implement at least SUM, MAX, MIN, AND and OR. Allowed data types for SUM, MAX and MIN are integer, real and double; logical operations should use integers with values 0 and 1.

The same clauses and types are used for entire **array reductions**, which work elementwise. The only complication occurs when all nodes are using the same working arrays (or scalars) and the result should be stored in another array (or scalar). This can be done by using for example RoR(SUM:IN=arr1,OUT=arr2), in which arr1 and arr2 have the same type and dimensions (could be scalars of the same type). In the case of e.g. a SUM reduction, both RoR and RoA, the user should initialize arr2 to zero on root (using RONLY).

4. The dumb parser and existing SPMDlib

The parser, written in C, has the following main tasks: (1) to detect and check all directives in a source program, (2) to detect data types and array dimensions of scalars and arrays included in directives, (3) to keep track of parallel loop and array dimensions for splitting them, (4) to change parallel DO loops and (5) to insert the correct SPMDlib calls. Only in the case of the advanced clauses DODYN and PIPE it will significantly change the source code, but the parsed code will stay legible because of extra comment lines that indicate the actions taken. Furthermore, all SPMDlib routines and special scalars/arrays are prefixed (`spmd_`) in order to avoid conflicts. Here we describe the existing library, i.e. the routines already implemented in Fortran (C is under construction). Advanced features for C and Fortran 90/95, as well as the new library to be implemented, will be described in a following section.

We note that the library was first developed for being used by the programmer. Because of the user-friendliness, this implies that the routines must be named such that it is easy to select the right ones for different data types and actions, and that there are many different routines. For example, R2A communications involve 4 data types, 1D and 2D arrays, entire arrays and array parts, plus special routines for extending array parts with additional elements/lines for neighborhood operations. This means that there are 24 R2A routines. However, since MPI is not concerned about data types, i.e. only the number of bytes for each data type is important, routines for integers call the routines for reals, and those for complex call the real*8 ones. All this is not necessary in developing the new library (see next section), because the new routines are not meant to be used by the programmer—only the directives must be used. In addition, the existing library was built taking into account the most important features that we needed. A future version should be more versatile, such that most basic operations are covered, such as all possible reduction types.

The main program and all subroutines that use SPMDlib calls need special declarations and a common block in the `spmd.h` include file[2]. Most important are the integers `spmd_myid`, `spmd_nprocs`, `spmd_mystart`, `spmd_mypart` and `spmd_myend`.

The routine `spmd_init(ierror)` initializes MPI as well as a few variables, notably `spmd_myid` and `spmd_nprocs`. The routine `spmd_end` quits MPI. The routine `spmd_split(dim)` is used for static loop splitting and array partitioning; it computes `spmd_mystart, spmd_mypart` and `spmd_myend` for changing loops and for use in the communication routines for array parts.

All communication routines have the following naming convention:
`spmd_[ty][nD][P/E][func](vars)`
in which:
`[ty] = i/r/d/c` mean integer, real, real*8 (double) and complex*8 data type,
`[nD] = 1D/2D` mean arrays with one or two dimensions (3D to be done),
`[P/E]` mean only Parts of an array or an Entire array,
`[func]`: toall, toroot, sumonroot, syncarray, filtX and toallX, and
`(vars)`: the name of one array and its dimensions.
Notes: (1) Psyncarray implements A2A with the E option; (2) PtoallX implements R2A with P(X=n), XC not yet being implemented[3]; (3) PfiltX implements A2A with P(X=n), see the same footnote; (4) Esumonroot is an example of an array reduction; all other types can be written in one hour or so.

The routines for scalar reductions do only the communications, i.e. a local maximum or sum needs to be done with a split loop after which globals are determined from the locals with one (or more) reduction call. Naming convention:
`RR` means Reduction-on-Root (only root gets global maximum (for example)),
`RA` means Reduction-on-All (global maximum goes to all).
Already implemented are:
`m` means max and min, and
`s` means sum and sum-of-squared, for integer, real and real*8 data types. Examples:
`spmd_iRAm(max,min), spmd_dRRs(sum1,sum2)`
These routines are not yet linked to directives, but were developed for image processing: (1) image normalization

[2] See `http://w3.ualg.pt/~dubuf/spmd.html`

[3] Is cyclic neighborhood processing important? Top image lines have no correlation with bottom lines. We never use this!

using the maximum and minimum values, and (2) for computing the mean and variance in one (split) image loop (well, `RRs` can do two independent sum reductions).

Note: sending one or two scalars takes the same time. Hence, if we want only a maximum (minimum), we put a dummy min (max) with an arbitrary value. The parser can also do this.

5. Status, other languages, etc.

After parallelizing a few programs with the existing routines we know that the SPMDdir/lib concept works, that there are no or few flaws in the design, and that a simple parser could be beneficial for many programmers. The parser has now highest priority and the standard Fortran version will be ready by summer 2003. By then we will also have selected an implementation solution for the new libraries in Fortran and C (the directives, called pragmas in C, will be the same as in Fortran).

The three main differences in the C version will be: (a) no common block can be used, hence all pre-defined spmd variables need to be global, (b) dynamic memory allocation must be included, with all its implications for splitting loops and communications, and (c) loops and their nesting differ from Fortran. It is likely that only one pointer structure will be supported, such as the one used by MPI.

For Fortran 90/95 we will also consider dynamic memory allocation. The question is whether a parser can always detect the position where an array size dimension is valid during runtime. If this causes a problem, we could include an additional directive to assist the parser (the same problem may occur in C). Fortran 90/95 also has compact array syntax. The parser should be capable to translate array syntax to be parallelized into (nested) DO loops, but this will impose restrictions on the use of the syntax and the arrays.

The main difference between the existing library and the new ones for the different languages will be the number of routines. Instead of creating many different routines for different data types and 1D and 2D (even 3D) arrays, a few generic routines can be shared. Even P- and E-type communications, the P-type with or without the X (eXchange) option, can be done with three routines for R2A, A2R and A2A traffic. Likewise, scalar and array reductions can be implemented by means of a few generic routines that must be called with a parameter that specifies the type of reduction.

At the moment separate communications for scalars and arrays are done by calling separately the necessary library routines. This involves much MPI overheads, especially for small messages where the "elapsed" time does not depend on the number of bytes (340 μs for up to about 500 bytes using normal 100 Mb/s ethernet). For small messages the latency is important, whereas for large ones the bandwidth counts (we measured 85.8 Mb/s). Hence, one would think that "packing" scalars and arrays into one "payload" will reduce the overheads. This is probably true, because packing and unpacking involves copying from one position in memory to another position, which can be done at 83 Mbyte/s (Pentium III @ 450 MHz), a factor of 8 times faster than using ethernet. However, the final payload will still require a latency of 340 μs and the effective bandwidth will be 85.8 Mb/s. Although the directive set allows to combine different scalar and array communications at one point in a program, it should be emphasized, again and again, that communications must be restricted to an absolute minimum. As a consequence, only the different scalars as specified by one directive might be combined by packing them together.

The use of a few generic routines enables the development of specific, optimized libraries for different cluster configurations and network topologies. For example, if a system (Linux kernel) uses multicast ethernet routing, the difference between R2A communications with E and P options might be very small, because root will output all packets only once. If this is the case, there is no need for specific R2A communications, and more effort could be spent in optimizing the A2R and A2A communications.

6. Communication optimizations

For the first versions the communication efficiency is less important; we aim at something which is more or less complete and which works. The root-to-all and all-to-root communications are all based on the same concept: after testing `spmd_myid` root will execute a loop over all non-root nodes to send or receive data and the non-root nodes will only receive or send data (virtual star topology). This solution is simple but it implies a lot of idle time, i.e. time that nodes are waiting to be served.

Let us compare some timings, assuming a cluster with 8 CPUs and expressing timings in units T which corresponds to the time necessary to send 1/8th of an array (in the case of big arrays we measured 85.8 Mb/s of the theoretical 100 Mb/s). Let us also assume that we start a timer before a "toall" call, and stop the timer after a "toroot" call. Between the two calls each node must process data, which takes NT seconds, say. This is illustrated in Figure 1 with $N = 10$.

(1) Serial execution takes $8 \times N \times T$ seconds. (2) Parallel execution takes $N \times T + 2 \times 7 \times T = (14+N)T$ seconds: this includes sending twice 7/8th of the array. The speedup will be $8N/(14 + N) = 8/(1 + 14/N)$. Obviously, the bigger N, or the workload, the better the speedup. (3) Root will be completely busy sending and receiving data, whereas other nodes will be receiving or sending 1/8th of the array. These are not idle times. Figure 1 shows that the total idle time ($\odot$ symbols) will be $2 \times 2 \times 3 \times 7 = 84T$, which is an average of $10.5T$ per node. If we take out the root node, because it

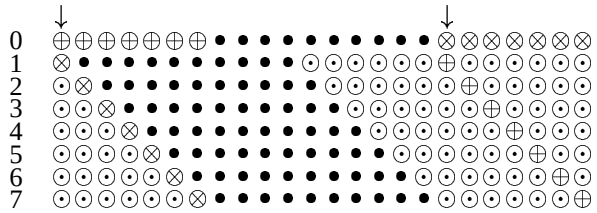

Figure 1. Node activity between a "toall" and a "toroot" call, marked by the vertical arrows, in the case of eight CPUs, assuming that there is no work before and after the calls. Symbols: $\oplus$ = send data; $\otimes$ = receive data; $\bullet$ = work; $\odot$ = idle (waiting).

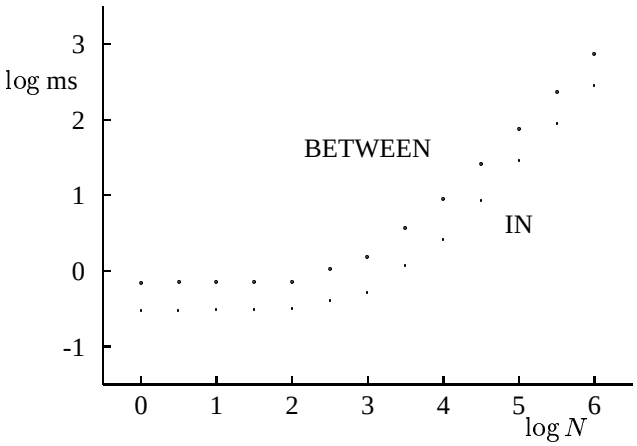

Figure 2. Ping-pong times in log(ms) as a function of log(N) in REALs on a dual-CPU cluster with fast ethernet. IN refers to communications between two CPUs inside one node. Single-trip times are half the ping-pong times.

will be completely busy with communications, the average even becomes $12T$. In words: all nodes, except root, will be idle 12 times the time it takes to transmit 1/8th of the array. The above calculation is an approximation assuming that there is no parallelism in the communications (synchronous `mpi_send` and `mpi_recv`). In practice, however, there may be substantial parallelism in the communications, e.g. during the reconstruction of array parts from ethernet packets.

In order to reduce idle times, we can adopt two strategies: (A) Instead of attributing root an equal share of the workload, its workload could be reduced, for example halved (in Fig. 1 there would only be 5 $\bullet$s in row number 0). This solution involves another array partitioning and loop splitting, it will still be static, and it can be easily implemented. However, in the ideal case, when root once distributes data and once collects data, and the bulk of processing does not involve massive communications, root will not equally contribute to the workload. (B) As for the DODYN directive, it would be better to reserve root for only doing communications and simple, fast array processing where required. In this case the workload will be evenly distributed over the $(N-1)$ other CPUs. This solution is ideal for clusters that are operated through a frontend, and it also can be implemented very easily.

Before we installed our new cluster consisting of eight Athlon single-CPU XP nodes, we experimented with a small cluster with dual-Pentium III nodes. The communications between the CPUs in one node (IN in Fig. 2) are faster than those between CPUs in different nodes (we measured a factor of 2.66, i.e. instead of 10.7 Mbyte/s (85.8 Mb/s) we measured 28 Mbyte/s). But 28 Mbyte/s is much slower than copying an array in memory (82 Mbyte/s). The reason for this difference is hidden in the TCP/IP stacks: an `mpi_send` goes down the stack, until the detection of the loopback, i.e. packets are not going out on the wire, and then up again to the matching `mpi_recv`.

There are two solutions for speeding up the communications: (a) replacing the interconnect by a more powerful one and (b) short-cutting the TCP/IP stacks. We studied (a) by replacing the ethernet by SCSI. This was indeed faster, but not much because the SCSI protocol does not have a bus master (this was solved by circulating a "token"). Hence, IP-over-SCSI[4] needs an alternative, preferably parallel interconnect with bus mastering. The second (b) problem, i.e. MPI-over-XXX has not yet been studied, but it will involve the use of direct routing tables, such that the CPUs in dual-CPU nodes know their "neighbors." In this case the entire TCP/IP stacks can be avoided by copying in memory (82 Mbyte/s), but not when doing communications between different nodes (10.7 Mbyte/s, unless another interconnect (XXX) is being used).

Apart from dual-CPU nodes being more efficient in terms of space and maintenance, they can be more efficient in communications because the underlying routines can be rewritten to exploit parallelism. Here we study the updating of an array of 10^6 REALs on 4 CPUs in 2 nodes: an A2A communication in which each CPU has 1/4th of the array but each CPU needs the entire array. Hence, we will need at least the transmissions of quarters (1/4) in an A2R routine and the entire array (1/1) in a R2A routine.

Table 1 gives single-trip times in ms for the payload sizes in REALs that we will need in order to compute communication times for $N = 10^6$ REALs. We can see that all numbers scale well with the size, i.e. sending the entire ar-

[4] See `http://ipoverscsi.sourceforge.net`

Size	IN	BETWEEN
1/1	142	373
3/4	106	280
1/2	72	187
1/4	36	94
1/8	18	47

Table 1. Single-trip times in ms as a function of the array fraction for 10^6 REALs. IN means inside a dual-CPU node, BETWEEN is between nodes.

ray takes four times the time of sending a quarter. This also follows from Fig. 2: the linearity of the curves down to 10^3 REALs, below which the communication overheads become more significant. This means that the speedup factors given below are valid until arrays of about 4000 bytes.
Scenario S1: CPUs 1, 2 and 3 send their 1/4 to 0 (A2R), after which root sends 1/1 to all others (R2A). All communications in this case are sequential. This implies for the A2R one IN of 36 ms plus two BETWEENs of 94 ms, a total of 224 ms. The R2A takes one IN of 142 plus two BETWEENs of 373, a total of 888 ms. Hence, S1 costs 1,112 ms.
Scenario S2: In the A2R CPU 1 sends 1/4 to 0 and, at the same time, 3 sends 1/4 to 2, after which 0 and 2 have each a different 1/2 of the array. This costs only one IN of 36 ms. Then 2 sends 1/2 to 0, one BETWEEN of 187. The A2R costs 223, the same as in the first scenario. We now apply the same parallelism in the R2A: first 0 sends 1/1 to 2 (one BETWEEN of 373), and then 0 and 2 send 1/1 simultaneously to 1 and 3 (one IN of 142). The R2A costs 515 ms, hence S2 costs only 738 ms.
Scenario S3: We can make one optimized A2A routine in which root does not collect and redistribute. In the first pass (a) 0 sends 1/4 to 1 and simultaneously 3 sends 1/4 to 2, which costs one IN of 36 ms. After this 1 and 2 have each 1/2. In the second pass (b) 1 sends 1/2 to 2 and 2 sends 1/2 to 1, which costs two BETWEENs of 187 or 374. Now 1 and 2 have both 1/1, but 0 and 3 need only 3/4. The third pass (c) costs only one IN of 106. We see that S3 only costs 516 ms, i.e. we have gained more than a factor of 2 relative to the first scenario.

Table 2 summarizes the results (4 CPUs) and compares these with times on single-CPU systems as well as a smaller (2 CPU) and larger (8 CPU) system. As expected, the times on dual-CPU systems are always less than those on single-CPU systems because of the smaller IN times.

Scalar RoA reductions can also be optimized. From Fig. 2 we see that the single-trip IN time is 0.15 ms and the BETWEEN time 0.35 ms for small messages. Table 3 summarizes results for single- and dual-CPU systems, by using similar scenarios.

	2 CPUs		4 CPUs		8 CPUs	
	single	dual	single	dual	single	dual
S1	374	144	1401	1112	2940	2680
S2			1027	738	1447	1187
S3			748	516	937	706

Table 2. A2A distribution times (ms) of an array of 10^6 REALs on single- and dual-CPU clusters according to three scenarios.

	2 CPUs		4 CPUs		8 CPUs	
	single	dual	single	dual	single	dual
S1	0.7	0.3	2.1	1.7	4.9	4.5
S2			1.4	1.0	2.1	1.7

Table 3. RoA reduction times (ms) on single- and dual-CPU clusters for two scenarios.

Our results show that it is possible to reduce communication times, not only by using dual-CPU nodes but also by using single-CPU nodes. After all, with ever increasing CPU performance we also need to squeeze all out of our interconnect: every millisecond counts!

Acknowledgments This work was partially supported by the FCT Programa Operacional Sociedade de Informação (POSI) in the framework QCA III.

References

[1] B. Gennart and R. Hersch. Computer-aided synthesis of parallel image processing applications. *Parallel and distributed methods for image processing III, Proc. SPIE 3817*, pages 48–61, 1999.
[2] M. Lueckenhaus and W. Eckstein. A thread concept for automatic task parallelization in image analysis. *Parallel and distributed methods for image processing II, Proc. SPIE 3452*, pages 34–44, 1998.
[3] F. Seinstra, D. Koelma, and J. Geusebroek. A software architecture for user transparent parallel image processing on mimd computers. *Proc. 7th Int. Euro-Par Conf., Springer LNCS 2150*, pages 653–662, 2001.
[4] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI: The complete reference*. MIT Press, 1996.
[5] J. Squyres, A. Lumsdaine, and R. Stevenson. A toolkit for parallel image processing. *Parallel and distributed methods for image processing II, Proc. SPIE 3452*, pages 69–80, 1998.
[6] T. Sterling, J. Salmon, D. Becker, and D. Savarese. *How to build a Beowulf*. MIT Press, 1999.