TR-532

Parallel Design Rule checking using Bitmaps

by

D. Dure

February, 1990

Parallel Design Rule Checking using Bitmaps

Daniel Dure - ICOT - fourth laboratory

June 11, 1989

Abstract

Circuit logic and layout design tools now available to circuit engineers have reduced the number of manual operations and, consequently, the probability of errors. Nevertheless, in an industrial environment, it remains necessary to check the observance of layout design rules within large scale designs. This operation requires a large amount of computing and memory resources that conventional mainframe computers can barely provide. We investigate here an implementation of such a rule checker, on the parallel machine developed at ICOT, and demonstrate that a linear speed-up of our program is to be gained as the number of processor increases.

1. Introduction

Design rule checking consists of applying a set of geometrical rules to a tentative design layout, in order to verify its consistency with regard to the technological rules which characterize a given process. In the sequel, we will refer this operation as DRC.

DRC operations usually take place at 3 specific moments, in the course of circuit conception :

- when building elementary blocks from the scratch, e.g standard cells. This operation is considered as one of the most difficult during the design, as it requires good knowledge of process details, and is likely to greatly affect performances of any circuit based on constructed primitives. DRC is necessary at this point because creation of basic cells remains essentially a manual process.
- When assembling several blocks together, by abutment or by inserting metal or polysilicon connecting paths. A this process is nowadays mostly automatized, DRC will be performed in rare occasions, when complicated assembly have been done, involving partial overlap of cells with merging of some conductive paths or body ties. As a matter of fact, it is always difficult to guarantee that libraries support this kind of operation.
- When a layout is received from a third party at the fab., and when elementary care is taken. Actually, this third case covers any messy situation in which one is not sure of the quality of a design, either because primitive conception tools have been used, or because a bug is suspected, or because of company policies, etc.

As reasonably fast computers, with high definition display devices, have become common, graphic layout editors took over more conventional methods such as paper or symbolic design (although the latter has still several applications). As well, it has appeared that the most convenient data structures to handle interactive layout design could be used also to perform DRC, in an interactive manner. A typical example of such tools with design and DRC capabilities is the MAGIC design editor, which is included in the Berkeley CAD tools package [1]. There, incremental DRC is performed over a part of the design, in a recursive fashion, and corrections are interactively suggested to the designer. Very unfortunately, such interactive tools are extensively using memory; they are unsuited to fast scanning of the whole circuit. Hence, DRC of an entire circuit, in a flat fashion, is mostly unfeasible. Of course this kind of operation does not occur frequently, as set out in the third point above, but it is nevertheless necessary in an industrial environment, with speed and accuracy as high as possible.

To this end, we developed a method which takes a direction quite opposite to usual algorithms, which are based on refined representations of layout objects in the plane : the entire design is mapped to a rectangular bitmap, each point of definition in the design corresponding with a bit in the bitmap[1]. As several layers usually occur in designs, each layer is mapped to a different bitmap. The next step is then to make some operations on these bitmaps, in order to check the correctness of the design. It has appeared that most rules found in the fab. literature could be expressed in terms of bit-wise logic operations, such as and, or, xor, shift, etc... These operations are very simple and for a normal circuit density the DRC of a circuit with such a *brute force* method has been found to have time complexity comparable to the one required by more sophisticated methods. But as data are spread over a bitmap, it becomes very easy to divide them over a network of processors, in order to perform DRC in a parallel fashion, thus gaining speed.

This note is concerned with the implementation and early results of an experiment based on this idea of parallel bitmap processing. The next section describes the operations which are necessary for bitmaps; section 3 gives some clues about the implementation on a Multi-PSI machine, using the KL1 programming language; section 4 presents some experimental figures of performance and section 5 is devoted to conclusion.

[1]Of course we assume that the design contains only polygons with faces at right angles. For DRC of designs with general shapes, see [3]. There is no reasonable way in this case but algorithmic geometry.

This program is an extension of a DEA experiment, lead in the LIE of the Ecole Normale Supérieure under tutelage of Jean-Marc Frailong and Jean-Gastinel, in the early 1986.

2. Basic primitives

It is good first to recall the main types of rules found in DRC specifications. They can be roughly divided into two categories: proximity rules and inclusion rules, with a possible mixture of them.

Proximity rules prescribe a minimum or a maximum distance between certain categories of object. Typically, they state that two metal wires should be separated by a given distance, or that the edges of a given wire should be separated by a minimum thickness distance, etc. Inclusion rules state that some object should be within some other type of object, with a minimum «security» border. As an example, in a p-well CMOS process, PMOS transistor should be within a well, with some margin within active zones and the well borders.

We shall now show, for a limited sample of such rules, how we can use bitmap operations. We will introduce new operations as they become necessary.

- For the first example, we consider the inclusion of every active zones in some well. Active zones can be obtained by the intersection of diffusion and polysilicon, say layer 0 and 1 (in a technology without buried contact). Then, to get faulty zones, we have to make a bit to bit «and» between the layout of active zones and the complement of the layout of wells. If we assume that the layout of wells lies in bitmap 2, we get faulty locations with the following sequence of instructions :

```
newregister              % allocate a new register (3)
equal   3 0              % load diffusion into 3
and     3 1              % and with polysilicon
andnot  3 2              % and with the complement of wells
```

  We used here our first instructions. A summary of the command language lies in next section. `newregister` allocates a bitmap with the same area as the largest of all bitmaps amongst all layers. `equal 3 0` copies the bitmap from layer 0 (diffusion) into layer 3. Then, the `and` operation keeps the intersection between diffusion and polysilicon. We note at this point that all active zones of the circuit are in register 3. It may be interesting to copy this register into an intermediary register, to avoid recomputation of active zones in the sequel of DRC. Eventually, with the `andnot` operation, we select amongst active zones only the ones which are not included in some well.

- The second example illustrates how proximity rules can be realized. A typical example is that metal line (assumed to be in layer 0) should be at least 3 units wide and separated by 4 units. Before we actually give the small program necessary to check this rule, we must explain what is «resizing» of polygons in a bitmap.

  Roughly, to resize a bitmap of n unit is to make all its objects «grow» by n unit. This can be expressed very simply in terms of bit-wise *shift* and bit-wise *or* operations: resize(R,n) = R1 or (R1 < n) or (R1 > n), where R1 = R or (R ↑ n) or (R ↓ n), where <, >, ↑, ↓ represent respectively shift of the bitmap to left, right, upwards and downwards, and n is the number of time the shift occurs. We assume that, at the limits of the bitmap, zero's are injected during the shift operations. Conversely, to resize a bitmap of -n unit, that is to say, to shrink it, can be represented as resize(R,-n) = R1 and (R1 < n) and (R1 > n), where R1 = R and (R ↑ n) and (R ↓ n).

  Now, let's think about the meaning of a negative resize followed by a positive resize, of the same number of units. We see on the following figure that such an operation will leave the bitmaps contents unchanged, provided that any object in the bitmap has a width greater than twice the resize parameter:

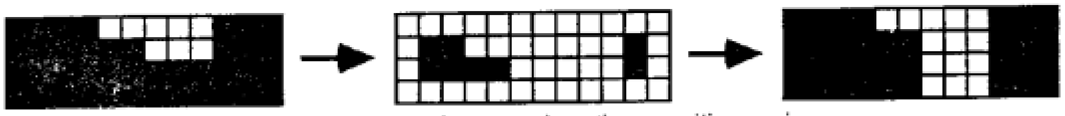

  Figure 1: effect of a negative, then positive resize.

  Conversely, what will happen if we enlarge first and then shrink of the same number of units? This time, the bitmap is left unchanged provided that all parallel extern faces of objects are separated by at the least 2n units:

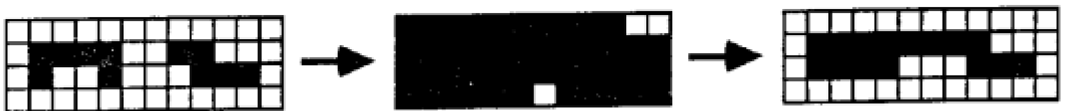

  Figure 2 : effect of a positive, then negative resize.

  This properties are quite promising, for our goal is to check proximity of edges and objects. But they allow us to check only even distances. Of course, we could double the scale of the bitmap in order to use only even coordinates, but that would cause a waste of memory. Anyway, this inconvenience of previous properties can be avoided if we use what we call «half-resize» operations. The idea is to operate resize operations in only one direction for each axis. For example, resize only to the left and to the bottom. More formally, we can express this as hresize(R,n) = R1 or (R1 < n), where R1 = R or (R ↓ n). Conversely, shrinkage can be expressed as follows: hresize(R,-n) = R1 and (R1 > n), where R1 = R and (R ↑ n).

  We can now give the little program which checks that all metal lines are at the least 3 units wide:

```
newregister                       % allocate a new register (1)
equal  1 0                        % load metal into 1
lsize  1 -3                       % shrink of 3 units
lsize  1 3                        % enlarge of 3 units
xor    1 0                        % difference with original metal
```

Note that the instruction `lsize` above is in fact the «half resize» we described before. There is also a `rsize`, which is the symmetric operation, from the viewpoint of resize directions. The composition of these two operations is equivalent to the anisotropic resize which we first introduced thereupon.

Quite similarly, we can check that metal lines are separated by 4 units:

```
newregister                       % allocate a new register (1)
equal  1 0                        % load metal into 1
lsize  1 4                        % enlarge of 4 units
lsize  1 -4                       % shrink of 4 units
xor    1 0                        % difference with original metal
```

We could give a list of the most common design rules found in fab. literature, but this is not our purpose here. If our reader wants more evidence of the validity of the method, he can have a look in [2]. For the pleasure of it, we just give the program which checks the two following nasty rules: The grid tail should overlap diffusion by 1 unit (see fig. 3), then any active zone should be within 30 units of a body tail.

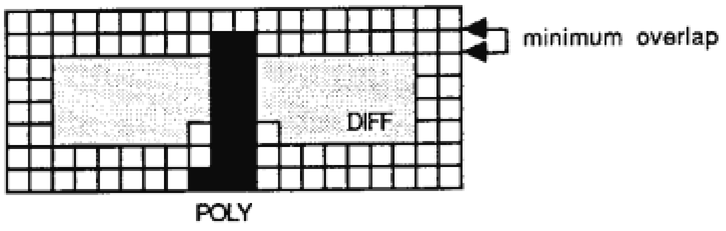

Figure 3 : overlapping grid rule.

To check this rule, we just need to compare the result of growing diffusion of one unit and then cut this with the polysilicon, to the result obtained by applying the cut before growing. Any bit of difference would be placed next to a common edge of polysilicon and diffusion, where overlap should have been. Since such a common edge could be on any side of a polygon, we have to do anisotropic resize:

```
newregister                       % allocate a new register (2)
newregister                       % allocate a new register (3)
equal  2 0                        % load diffusion into 2
lsize  2 1                        % resize left and top
rsize  2 1                        % resize right and bottom
andnot 2 1                        % cut with polysilicon
equal  3 0                        % load diffusion into 3
andnot 3 1                        % cut with polysilicon
lsize  3 1                        % resize left and top
rsize  3 1                        % resize right and bottom
xor    2 3                        % difference between the two resize ops.
```

For the second problem, the basic idea is to identify body ties, and then to resize them by 30 units. This creates objects which should cover any active zone. Here also, we have to use anisotropic resize. We assume in the following that n diffusion is in layer 0, p diff is in layer 1, polysilicon is in layer 2 and p well is in layer 3:

```
newregister                       % allocate a new register (4)
newregister                       % allocate a new register (5)
equal  4 0                        % load diffusion into 4
lsize  4 30                       % resize left and top
rsize  4 30                       % resize right and bottom
equal  5 1                        % load complementary diff into 5
and    5 2                        % and with polysilicon
andnot 5 4                        % out of bounds active zones
and    5 3                        % select correct polarity, using well.
```

As we can see, this method is quite easy to use, but requires some care while programming rules. This can be compared to assembly code generation for a high level language. It is clear that it would be nice to define a high level language to express rules in a more user friendly fashion, and then to compile these «programs» into executable bitmap operations. This would also allow data flow analysis of the program, and possibly save time : some operations over initial bitmaps are repeated several times. Of course the number of bitmap registers we can use is limited. «Register allocation» techniques used in the craft of compilers find an application here. Speed gain can be hoped all the more than, as we will see in section 4, the memory which can be devoted to the storage of bitmaps is quite huge. Hence, the number of available registers is also large.

3. Parallel implementation

Parallelism can be seek for in several ways, in order to speed up the solution of the previous problem. We first have to imagine how the overall DRC process is organized. This is illustrated in figure 4. We see there that several important operations, such as mapping of polygons to bitmaps and extraction of errors, have to be taken into account. Although there are occurring less frequently than the bitmap operations, they handle a high number of objects and can become a bottleneck for the whole DRC process.

The first stage of circuit analysis is to find the necessary size of the enclosing bitmap. In the current implementation, this is done after that all polygons have been read from a file. This is clearly a mistake, as all objects are stored in memory before their mapping into bitmaps can start. In future version, a single read of the file should be done previously to polygon splitting.

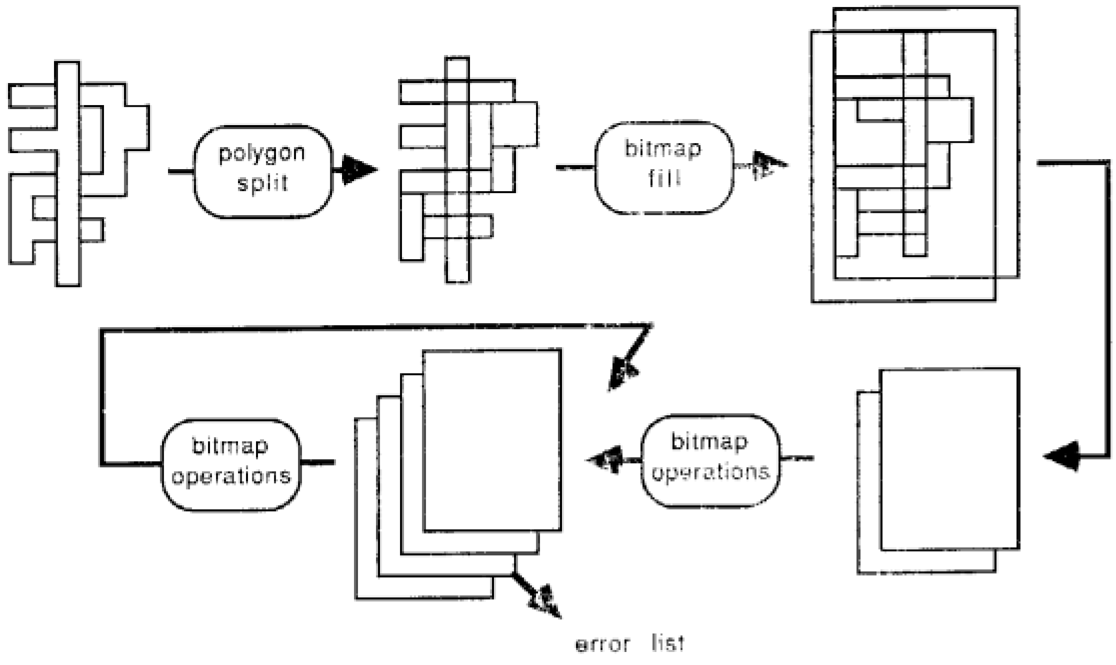

Figure 4 : the overall DRC process.

Then, polygon splitting itself can be divided among all the processors which are available in the Multi-PSI. In the current implementation, this is not the case either, due to lack of time. Basically, the idea would be to send polygons in a stream, which goes through all processors. Then each processor picks polygons in this stream, according to the following rule : if there are P processors, the nth polygon is picked by processor p if n equals p modulo P. As each polygon is split into squares, we merge all squares into a single stream. Checking the above conditions is implemented very simply by using a counter on each processor. The corresponding organization is pictured in figure 5.

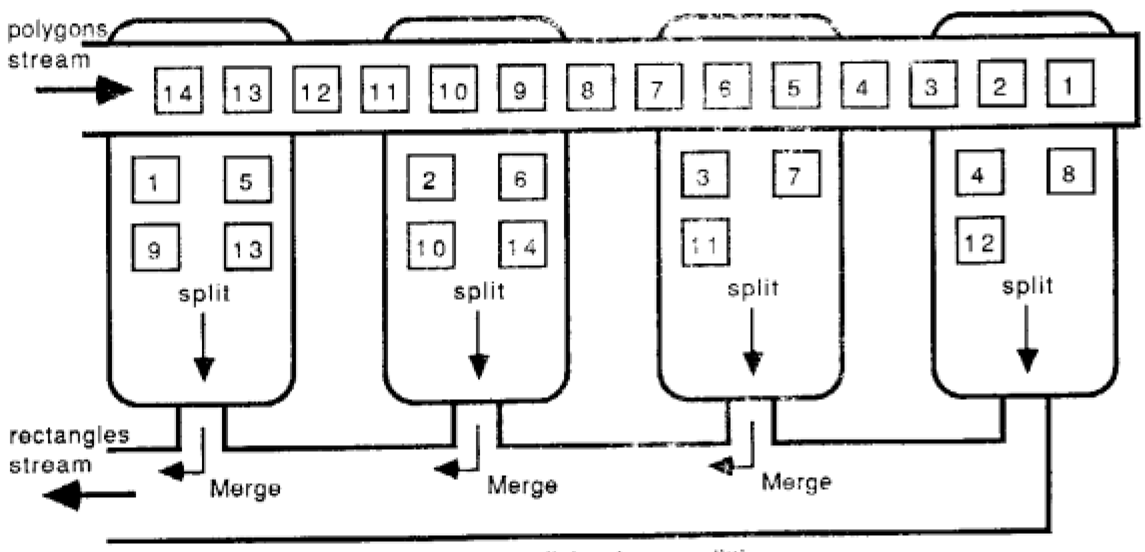

Figure 5 : parallel polygon splitting.

A minor optimization of this methods can be done, to avoid unnecessary duplication of polygons which have been split : if a processor has split a polygon, it does not put it back into the output stream. The test used to check whether the nth

polygon should be treated by processor p out of P is a little bit different : n should be equal to 1 modulo (P - p + 1). This allows a twofold decrease of the communication cost.

Now comes the parallel usage of the produced rectangles. Before explaining this point, we need to see how bitmaps themselves are divided amongst processors.

We haven't given yet a complete list of available primitives, but basically their complexity grows like the area of the bitmaps they work upon. Hence, a natural way to speed up operations is to divide area between processors. In order to distribute computation load equally between processors, and thus maximize efficiency of parallel operations, we have to divide area as uniformly as possible between processors. Also, to keep simple operations on each processor, it is more reasonable to divide the initial bitmap into rectangular bitmaps.

Dividing initial bitmap is a nice idea but some problems are then introduced. The first problem is that errors have to be extracted in several bitmaps, and then merged. This is not a big problem, as the number of such errors is usually low, compared to the number of polygons in the circuit. In the current implementation the fusion of error squares found in all the sub-bitmaps of the error bitmap is not performed, but a classical 2 dimensional sort like [4] can be used. The second problem is that during resize operations, zero bits will be injected on the sides of bitmaps. One solution would be to exchange values of border bits between processors, but that would require some communication and require synchronization. Another possibility is to duplicate some of the data between bitmaps. This is what we chose to do. If the maximum resize parameter is N, during the whole DRC, then it is necessary to duplicate N bits between borders of sub-bitmaps, in each direction. An example of the resulting division is shown in figure 6. We assume in this figure that the initial bitmap is 8x8, to be divided between 4 processors, while the highest resize magnitude is 1.

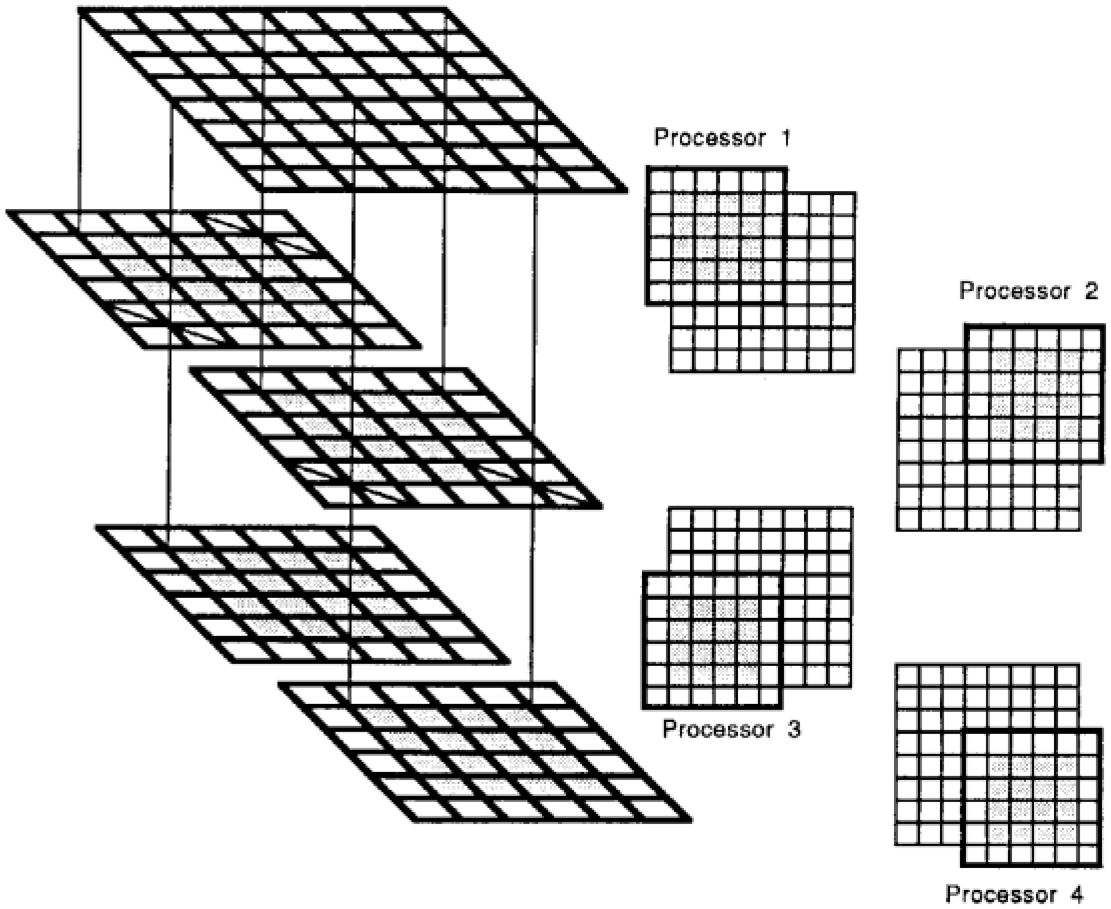

Figure 6 : overlapping of sub-bitmaps.

The overlap of some data implies that when building original bitmap from circuit description, some parts of the polygons are found at the same time in 2 bitmaps (or 4 bitmaps for the corner bits). In the sequel, the width of overlap will be called the *margin*. As we said that complexity of basic bitmap operations is proportional to the size of bitmaps, we see that our interest is to find a division which makes bitmap areas equal, but also which minimizes the perimeter of rectangles : If we assume we have a division of initial area A into P identical rectangles with perimeter λ, for a margin M, global time complexity is :

$$C = \frac{A}{P} + \lambda M + 4M^2 + P\alpha$$

where α is the small constant representing the cost of synchronizing processors. Of course, the above figure decreases when rectangle shapes become close to squares. The lower bound is :

$$C = \frac{A}{P} + 4M\sqrt{\frac{A}{P}} + 4M^2 + P\alpha$$

Refined methods [5] to get near optimal divisions exist. We chose a quite simple one, based on the observation that usual circuits have a square shape, whereas we store 32 bits in a single word, thus introducing a large unbalance between the horizontal and vertical lengths of the original bitmap.

For more than 32 processors, the following algorithm is used : we first decompose P into its prime factors $p_{m>i\geq 0}$, sorted in decreasing order. Then, starting with the initial dimensions x_0, y_0 of the overall bitmap, we apply the following rule until all primes p_i are used :

$$\text{if } y_n \geq x_n,\ y_{n+1} = \frac{y_n}{p_n} \text{ and } x_{n+1} = x_n$$

$$\text{if } y_n < x_n,\ x_{n+1} = \frac{x_n}{p_n} \text{ and } y_{n+1} = y_n$$

Once all the factors of P have been used, x_m and y_m are the sizes of bitmaps on each processor. Since the original bitmap dimensions may not be multiples of x_m and y_m, an eventual remainder of division is scattered amongst the first processors. That way, the difference in size between bitmaps on all processors is bounded by 1, in each direction, which is negligible.

Although this method is very simple, «tolerable» behaviors have been observed, as the number of processor was never a big prime number. If this should occur, complexity would become, for an originally square bitmap :

$$C \equiv \frac{A}{P} + 2M\left(\frac{\sqrt{A}}{P} + \sqrt{A}\right) + 4M^2 + P\alpha$$

If we neglect the communication cost, we get an asymptotic bound which is in a ratio $\sqrt{A}$ to 2M (usually several hundreds) with the previous lower bound. For values of P under 10^3, the bad decomposition of the initial bitmap does not account for more than one third of the computation time, for usual values of A and M.

Now, our problem becomes to know how we are going to transform polygons into data in bitmaps. It was suggested above to split first polygons into rectangles. This is quite intuitive; refer to [2] for details of the algorithm. Then, we have to distribute rectangles over the «good» bitmaps. We may think of some clever process to send a rectangle only to the processor which is in charge of the corresponding bitmap. Unfortunately, as a rectangle may pertain to several bitmaps, this turns out to be difficult, and would demand multiple access to several processors at the same time, which is heavy to manage. So, we chose a very noddy approach again : all processors are traversed by a single stream, in which all rectangles are sent. If a rectangle is at least partly included in the bitmap managed by some processor, the latter makes a copy of the rectangle and puts it into the bitmap. In any case, the rectangle is sent to the next processor. This method is very similar to the one we plan to use for polygons. It is also possible to perform an optimization which diminishes the number of communicated rectangles of 20%: if a rectangle is completely included in some processor bitmap, with no bit in the overlap, this rectangle is of no use to other bitmaps. In current implementation, this optimization is not performed.

Another point is to make uniform the flow of rectangles through processors. We chose to wait before transferring a rectangle, for completion of the previous modification of the bitmap. This is useful to avoid an accumulation of data at several places at the same time in the network. On the other hand, our method bounds the speed of the whole process to the speed of the slowest processor. This is not a big problem as subsequent operations will anyhow not start before all processor have finished. In future version, a buffer should be used.

Now that we have introduced the essence of the method, we give a summary of the DRC command language, in its current state. In the following, arguments enclosed between <> are meta-arguments, while [ ] indicate optional arguments.

<u>Pimos command</u>

There are 4 ways to call the program, to allow debugging :

```
drc:drc(<command file> [, <debug file>] [,processor])
```

`<command file>` is an atom which contains the name of the text file containing DRC commands. `<debug file>` is a character string with the name of a file in which timing measurements and reduction counts will be stored. This argument is optional. `processor` is an integer indicating on which processor the master process should be started. By default, processor 0 is used. The number of the processor connected to the front-end PSI should be put there. This is an optional argument.

<u>Command File</u>

Basically, each command is a vector. The first element of the vector is an atom which identifies the command itself, while remaining elements are arguments of the command. In the command file a vector should be enclosed between { } [1], with a period at the end of the command. Comments are allowed, provided that they start with a %. If you dare use Emacs, don't forget to put a new-line character at the end of the file.

[1]In the following, curve braces will be used to represent vectors, to make the syntax lighter.

There is no need to insert a special command at the end of the command file, but the `stop()` command will stop the parsing if it does not match with a previous `if`.

General Commands

`run(<file>)`

This command makes the DRC program start interpreting commands found in the file identified by the atom `<file>`. It works recursively and within conditional blocks. When the file execution is terminated, execution is resumed in the calling context. This command may be convenient to divide DRC into several files, possibly written by different people. If some syntax error is found in the file, no message is displayed and, in the worst case, the KL1 built-in parser crashes, usually printing some mysterious message... This will not be corrected in the future, I think.

`window(<name>)`

This command is quite similar to the previous one, in the sense that it creates a new context from which commands are read. But this time, instead of a file, we get commands from the user, via a PIMOS window. The atom `<name>` is used to give a name to the window. If there is a syntax error, a message is output on the window. But the KL1 parser may still crash...

`help()`

This command displays help on the current user window. If this command is found in a file, nothing happens. Help is rather concise and this command does not support any argument. I am afraid this will not be improved.

`echo(<data>)`

This command displays `<data>` on the system console. This data should not contain any undefined value, but can be of any type or complexity. There is no limit as to the depth of structures or any other stupidity. This command can be useful to monitor activity of the DRC process. It should be kept in mind, though, that displaying on the console is ludicrously slow.

Register management

`put(<margin>,<proc>,<file>)`

This command is used to transform a file holding polygons into a set of bitmaps. The format of the file is very simple each polygon pertains to some layer, starting from 0, which is represented as an integer followed by a period. Then follows the polygon itself, as a list of point coordinates, followed by a period. The points should be taken on the contour of the polygon, in clockwise or counterclockwise order. The following is an example of what could be such a file :

```
0.                          % the following polygon is in layer 0
[0,1,0,1,1,1,0,1].          % a square between (0,0) and (1,1)
```

No special command has to be inserted to mark the end of the file. Just don't leave half of a list, or the parser will bomb out. Comments are allowed.

The `<Margin>` parameter is an integer which should be set to the maximum number of units used in any resize in the DRC. If 0 is set, there is no overlap between bitmaps. The `<Proc>` parameter is the number of processors to use for bitmap operations. You should know that at least one processor is used just to parse commands and files. So, if you have a 16 processors system, this parameter should not exceed 15. Eventually, `<file>` is an atom identifying the file where polygons are sheltered.

In the current implementation, there is no way to keep the registers known previously to this command. In other words this command restarts everything from the fresh.

`newregister()`

This command allocates a new register. If it is the first issued after a put, the number of this register is the number of the last layer found in the circuit file plus one. As more registers are created, number is increased. Keep in mind that the reading context created when reading a command file or opening a window is not related to the register space. The register space does not work like a stack. That may be improved in future versions; otherwise, nesting command files is not very usual. Also, that would make the task of an hypothetical compiler more easy.

As to the number of available registers, the figure depends on the size of the bitmaps, depending itself on the circuit. Read next section for more details about this. As there is currently no way to know the available memory in the system, if too many registers are allocated, memory shortage will occur during garbage collection. In other words, the system crashes.

`get(<layer>,<file>)`

This commands extracts squares from the layer specified by the integer argument `<layer>`, and puts them into the file specified by the atom `<file>`. Squares should be merged to produce polygons. I don't think this will be improved as it is not an absolute necessity. The format of the produced file is compatible with the `put` command.

As our careful reader may have guessed, extraction is not performed over the entire area of each sub-bitmap. The margin is not scanned, as its contents are not consistent after resize operations. A feature of this operation is that the bitmap is cleared, except margin. So, for future use this bitmap should not be considered as blank. It has to be initialized via one of the commands `load`, `clear`, `black` or `equal`.

```
dump([<file>])
```

This command produces a picture, using 0 and 1, of all sub-bitmaps currently present in memory. Dump is done in the current command window, if command is issued from it. If this command is interpreted from a file, nothing is displayed. If the optional atomic argument `<file>` is present, dump is done in the file it identifies.

This command was used during debug and it is terribly slow, because rows are dumped character by character, using the *putt* command of Pimos file device.

```
save(<layer>,<file>)
```

This command saves the contents of the layer specified by the integer `<layer>` in the file specified by the atom `<file>`. I did my best, but this commands remains slow. That will be improved in the next version. The contents of the original layer are not changed by this command.

```
load(<layer>,<file>)
```

Conversely, this commands loads a bitmap from a file. Dimensions of the current set of bitmap should be the same, and margin as well. Practically, the two previous commands should be used only within the DRC of the same circuit. Currently, if there is a difference in size, DRC bombs out. In future version, discrepancies will be checked, but not supported.

Let's note that the two previous commands are useful when there is a shortage of register, in order to avoid recomputation of intermediary registers. But considering the sloppiness of save and load operations, I don't think it has much interest, even with few processors.

<u>conditional commands</u>

As to the matter of error output, it is not useful to output an empty error file. Also, for some complicated rules, it may be possible to abort operations if no object is present on the bitmap. The following commands are provided for the purpose of such optimizations.

```
if(<layer>)
...
stop()
```

This command has a single argument, an integer, which specifies which layer is checked : if any bit is set to one in any of the sub-bitmaps, the commands within the block are interpreted. Note that the margin is not taken into account during the check. Only the zone which will contain relevant data after a resize is checked.

The beginning of the block is marked by `if(...)` and the end of the block is marked by `stop()`. Several conditions can be nested. Since there is a complete equivalence between an end of file and the `stop()` command, there will be no syntax error in case of unbalance. In future version, some warning will be output on the console.

```
ifnot(<Layer>)
...
stop()
```

This command works in a similar way to the previous one. Following block is interpreted if the layer is blank, besides margins.

There should be a conditional command which also checks the margin part of a bitmap, and some «else» construct. That will be improved in the next version.

<u>One-operand bitmap operations</u>

```
clear(<layer>)
```

This command clears the layer specified by the integer argument `<layer>`, margins included.

```
black(<layer>)
```

This command sets all bits of the layer specified by the integer argument `<layer>` to one, margins included.

```
neg(<layer>)
```

This command inverts all bits of the layer specified by the integer argument `<layer>` to one, margins included. This operation is equivalent to an *exclusive or* of the specified layer with a blackened layer.

```
rsize(<layer>,<units>)
```

This operation performs a resize of all objects in the layer specified by the integer argument `<layer>`. Resize is done from the top and right edges of polygons. For the matter of operations on the margins, the sub-bitmaps are virtually surrounded by zero. Hence, resize operations do not produce meaningful results on the margins, as we already know it.

```
lsize(<layer>,<units>)
```

This operation performs a resize of all objects in the layer specified by the integer argument `<layer>`. Resize is done from the bottom and left edges of polygons. Besides, it works in a similar way to the previous command.

Two-operands bitmap operations

The destination register is the first one from the left; the second one is not modified by the following operations. Also, all parts of the destination register are affected similarly by the following commands, even the margins. For all the remaining commands, destination and source registers should be different, or deadlock will occur. That will be checked, but not supported, in the future version.

```
equal(<layer1>,<layer2>)
```

This copies the contents of the layer specified by the integer argument `<layer2>` into the layer specified by the integer argument `<layer1>`.

```
and(<layer1>,<layer2>)
```

This writes into the layer specified by the integer argument `<layer1>`, the result of a bit-to-bit *and* between contents of the layer specified by the integer argument `<layer2>` and the contents of the layer specified by the integer argument `<layer1>`.

```
or(<layer1>,<layer2>)
```

This writes into the layer specified by the integer argument `<layer1>`, the result of a bit-to-bit *or* between contents of the layer specified by the integer argument `<layer2>` and the contents of the layer specified by the integer argument `<layer1>`.

```
xor(<layer1>,<layer2>)
```

This writes into the layer specified by the integer argument `<layer1>`, the result of a bit-to-bit *exclusive or* between contents of the layer specified by the integer argument `<layer2>` and the contents of the layer specified by the integer argument `<layer1>`.

```
andnot(<layer1>,<layer2>)
```

This writes into the layer specified by the integer argument `<layer1>`, the result of a bit-to-bit *and* between the bit to bit 1's complement of the contents of the layer specified by the integer argument `<layer2>` and the contents of the layer specified by the integer argument `<layer1>`.

4. Experimental measurements

Experimental measurements have been conducted with two purposes in mind : to check the number of reductions done in the program and to measure the execution time. Knowing how many reductions are performed allows some verification of the global behavior of the program, while time measurements are related to the behavior of the slowest of all processors. As in our case all processors are doing more or less the same thing, I confess that making the difference is somewhat Byzantine...

Anyway. For the first experiment, we used an input file with only two squares of one bit in area, with a distance which was varying from 128x128 to 2048x2048. As we store 32 bits in a word, the bitmap sizes were varying from 4x128 to 64x2048. Considering this unbalance, as we used from one to fifteen processors, bitmap division was always done according to the vertical axis.

Experiment itself was conducted over a set of 12 instructions. More precisely one instruction of each type in the above list (`newregister`, `if` and file operations excluded). This set was executed once, then twice in a row, and the resulting mean execution time and reduction count were extracted. The error resulting from this set of measure is a systematic error (100 ms), while the average error was 15 ms. A special purpose set of routines was developed at this occasion, as the system listener does not give meaningful results and tends to perturb the program.

It is worth to note that the first version of the program was naively written, launching many goals in parallel in the same processor. The main consequence of this was an breadth-first exploration of the program execution tree and a quick exhaustion of memory. Inserting internal synchronization has lead to a threefold increase of program speed and a normal memory usage.

For our first experiment, we were basically interested in a measure of the speed up when the number of processor was increasing. The time complexity we expect is of the form :

$$T \approx (\frac{s^2}{32P} + sM(\frac{2}{P} + \frac{1}{16}) + \frac{M^2}{8})\beta + P\alpha$$

So, at the first order, when there is no margin, we expect the time to be proportional to the inverse of processors and to the square of s, the base of the bitmap dimension. This is demonstrated in figures 7 and 8.

In figure 7, we extracted the slope of the curve, for each bitmap size. These slopes have been reported in figure 8, where we can check that they match the parable found for a single processor. According to these figures, $\beta \approx 0.5$ ms.

It is also possible to measure the communication cost α, if some care is taken. Figure 9 represents the time complexity of 12 bitmaps operations for a 128x128 bitmap. We can see that the time is actually the sum of an hyperbole and a linear function of the number of processors. The coefficient of the hyperbole, 341 ms, is not totally matching the previous formula. In fact, some terms which are small but linear according to the bitmap base size have been omitted in the complexity formula. We can nevertheless consider that $\alpha \approx 20$ ms.

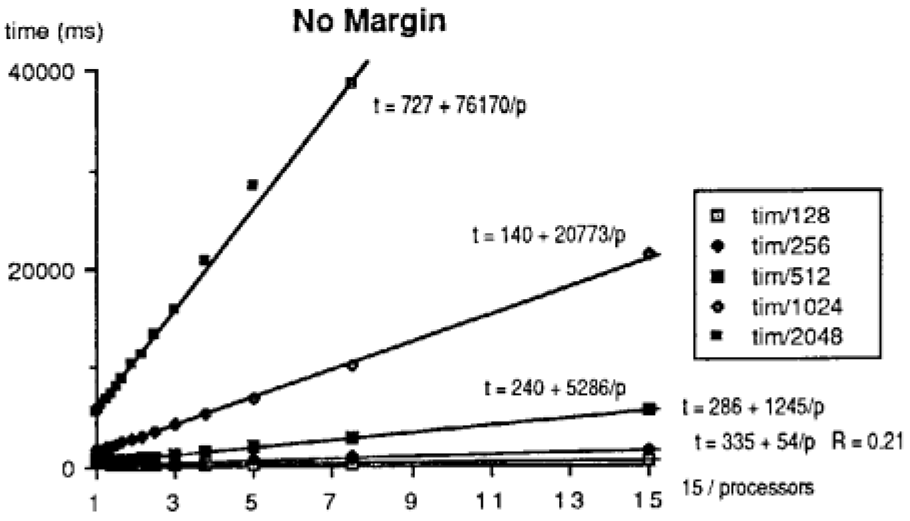

Figure 7 : execution time according to the inverse of the number of processors

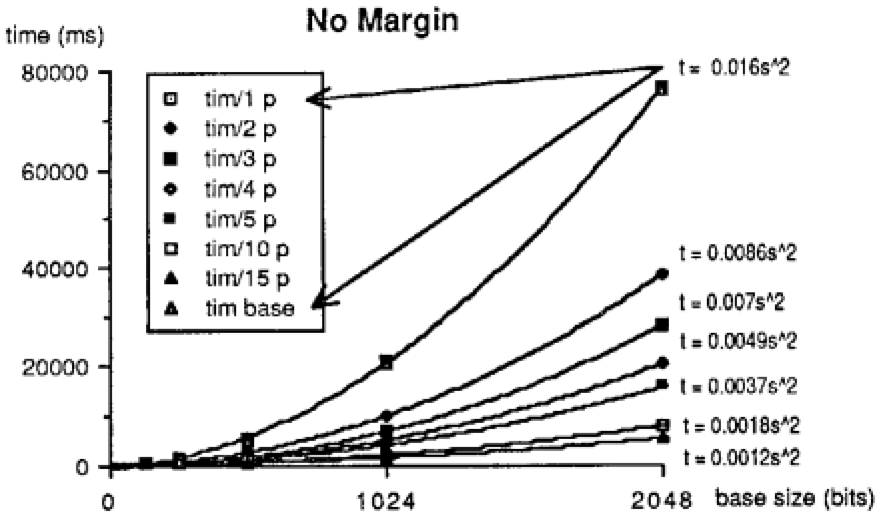

Figure 8 : execution time according to bitmap base

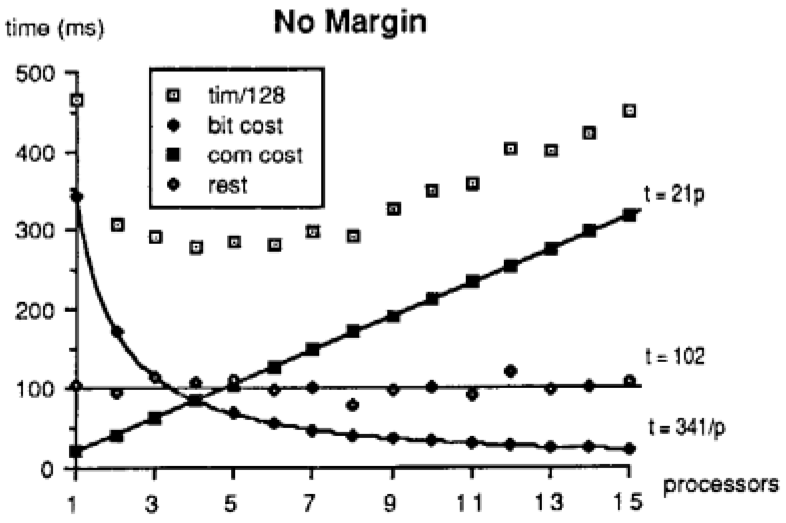

Figure 9 : set out of communication cost, for a 128x128 bitmap

It is interesting to check what happens if we use some margin. Figure 10 depicts execution time against the inverse of the number of processors, for a 32 bits margin.

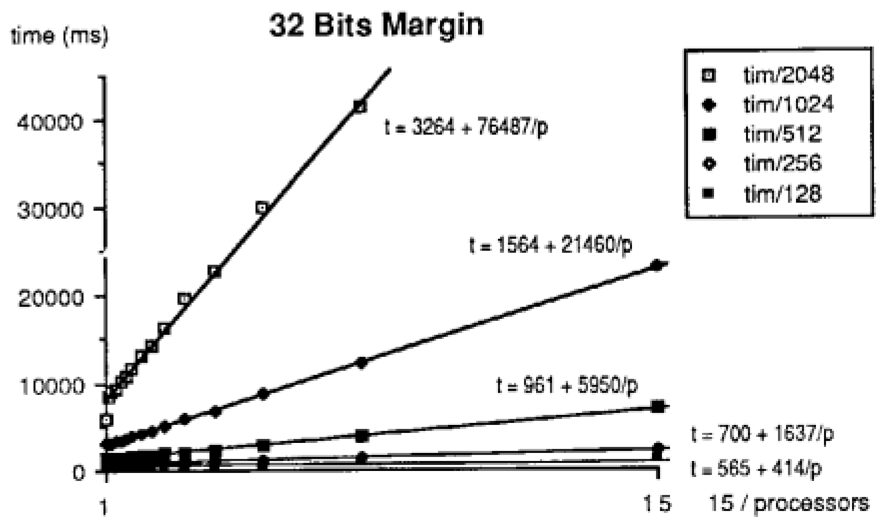

Figure 10 : execution time according to the inverse of the number of processors

We can confront the slope coefficients and base of the lines pictured in the previous figure to the analytical complexity. Figure 11 holds the comparison.

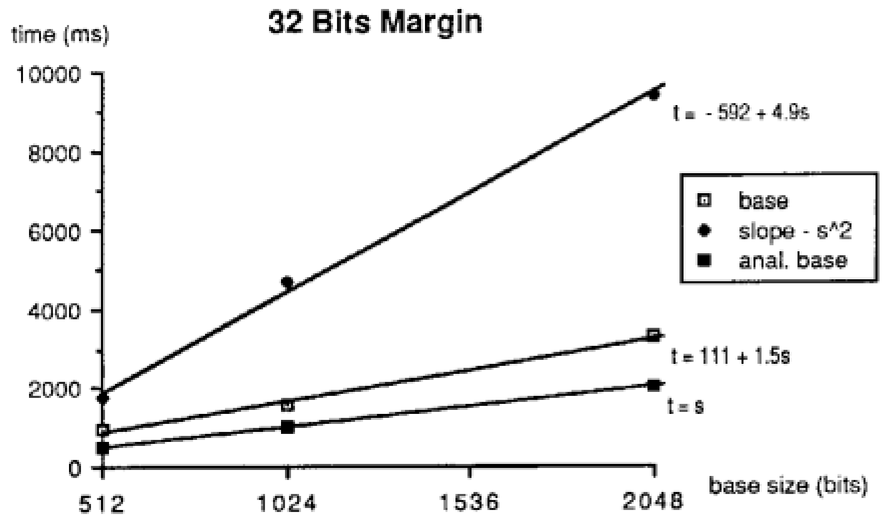

Figure 11 : modifications of slope and base coefficients of time brought by the margin

We find again a 30% discrepancy between analytical base and measured base. This is due to some operations which require a small time, which grows linearly with the base size. They have not been included in the complexity formula. As to the slope of time curve, we subtracted the quadratic figure $0.5s^2$ before plotting it in figure 11. This curve is much lower than it should be : there is a theoretical ratio of 32 between the base and the slop coefficients. General shape is satisfying, but more experiments should be conducted, with different values of the margin parameter.

From this experiment, we can infer some limits on performance, for real data. Let's consider a rather large circuit, say 1cm^2, described by using 1 μ units. Typically, we will perform about 2000 bitmap operations for an average set of DRC rules, over 10 bitmaps of 10^4 by 10^4 bits, with a 32 bits margin. The formula giving time complexity of bitmap operations is the following:

$$T = \frac{\beta}{32}\left(\frac{s}{\sqrt{P}} + 2M\right)^2 + \alpha P$$

This value reaches a minimum T_m for some value P_m of P. For $s^2 \gg P_m M^2$, an approximation of P_m is given by the following formula :

$$P_m \approx s\,\gamma^{-0.5}\,(1 + M\,s^{-0.5}\,\gamma^{-0.75}), \text{ with } \gamma = 32\,\frac{\alpha}{\beta}$$

This approximation is valid if $s\gamma^{0.5} \gg M^2$. It can be verified that in our case,with $\gamma \approx 1250$, $s = 10^4$, $M = 32$, this condition is broadly satisfied. So, we find $P_m \approx 283$. An approximation of the time limit itself is then given by the following formula :

$$T_m \approx 0.25\,(2\alpha\beta)^{0.5}\ s$$

For 2000 operations, we find that about 30 minutes are necessary to perform DRC.

Let's now compute the number of available registers : we need 10^8 bits for a register (10 are already occupied by the original circuit). With 283 processors, this figure is decreased down to 45 KBytes per processor. Since 40 MBytes are nearly available on each processor, a few hundreds registers can be allocated without much problem. This may substantially decrease the number of necessary operations, if intermediary values are kept.

To be honest, in the current version, vectors are used instead of strings in order to store bitmap. The main consequence of this is a waste of memory. The above figure has to be multiplied by 2 or 3 to reflect the exact memory cost. Still, that leaves us with a comfortable number of registers, if a clever compilation is done. Besides memory waste, vectors also cause a degradation of the performance of the program. A factor of 2 or 3 can still be gained in execution speed. Furthermore, the next version of the parallel machine, PIM, uses integrated technology instead of gate technology and should bring a fivefold increase of execution speed. All in all, measured time complexity can be decreased by a factor of 10.

The goal of the second set of experiments was to measure the time necessary to transform a set of polygons into bitmaps. We only checked that this time was decreasing linearly with the number of processors. For the experiment, we used a file with 100 identical squares. The size of squares and the number of processors changed during the experiment. Figure 12 depicts the results.

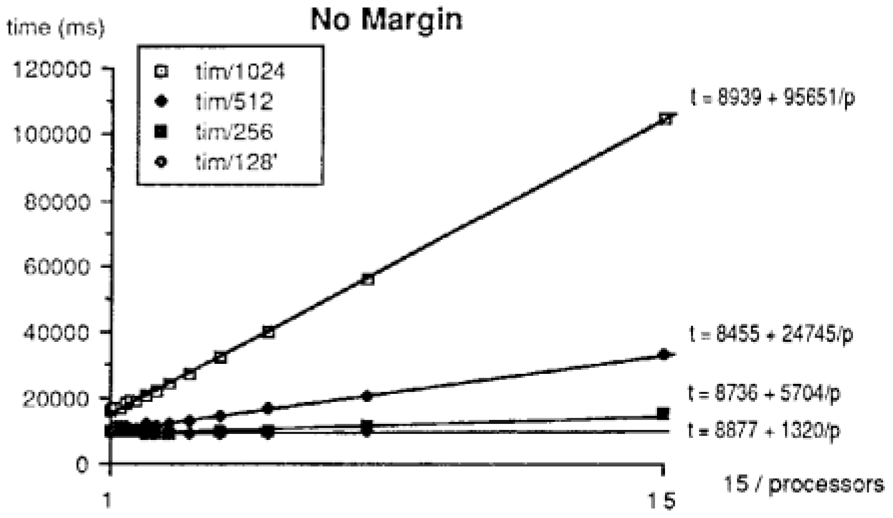

Figure 12 : time of circuit analysis versus the number of processors.

Experimental conditions are not quite satisfying : the first point is that all squares are identical and cover the whole area. Therefore, pipe-line effect is maximal. On the other hand, as the squares are comparatively large, pipe-line is long to initiate. If we compare the above complexity figure to the one for an average bitmap operation, we find that putting a square in a bitmap is roughly twice faster, say it requires a time $\delta = 8\ \mu$sec per unit of area.

The constant factor above takes into account polygon splitting (which is fast in our case) and file operations. A constant time of nearly 3 seconds has been experimentally observed to be necessary to open and close a file on the current version of the system. Therefore, we can consider that square splitting takes about $\sigma = 50$ ms. In the future version, to treat N polygons with E vertical edges and area A, on P processors, will require a time T, roughly described by the following formula :

$$T \approx \frac{N}{P}\left(\left(\frac{\sigma E^2}{2} + \delta A\right) + \alpha P\right.$$

We can take $N \approx 10^5$, $E \approx 10$, $A \approx 1000$ and $P = 500$. Then, $T \approx 500$ seconds, i.e one third of the time required to perform operations themselves. Unlike during bitmap operations, inter-processor communications don't become the bottleneck quickly, as basic operations keep a high cost.

5. Conclusion

In this note, we presented the main algorithm used in our DRC program and some early experimental results. This program is still in an immature state. The next version, if ever scheduled, should bring more performances and user facilities. Nevertheless, we showed a possible usage of a parallel machine to treat a problem which is currently handled by sequential computers.

It is true that most operations performed by our program could be performed on a SIMD machine, such as the Connection Machine™. But we observed that a non negligible time, about one third, is spent to do initial circuit analysis, whereas this task is not easy to divide on a SIMD machine. A linear speed-up is expected on a MIMD machine such as the Multi-PSI.

I would like to thank Mr Koichi Kimura and Dr Kazuo Taki for their kind help, and Ms Sungsoon Kim for tweaking complexity minimum with me.

6. References

[1] ??, «Berkeley's CAD tools», user manual, 198?, pp ??.

[2] D. Dure, «Eléments pour une CAO de circuits VLSI,» rapport de D.E.A., Université d'Orsay, 1986.

[3] C. R. Bonapace and C-Y. Lo, «Larc2: A Space-Efficient Design Rule Checker,» *proceedings of ICCAD*, 1987, pp. 298-301.

[4] L. Pierre, ECLTRI+FUSION, *A Private Collection of Pascal Programs*, Ecole Normale Supérieure, 1985.

[5] T. Y. Kong, D. M. Mounti and A. W. Roscoe, «The Decomposition of a Rectangle into Rectangles of Minimal Perimeter,» *SIAM journ. of Computers*, 17(6), 1988, pp. 1215-1231.