A Parallel Router for Printed Circuit Boards

D. Abramson †
J. Freidin §

† Division of Information Technology
C.S.I.R.O.
§ c/o Department of Communication and Electrical Engineering
Royal Melbourne Institute of Technology
P.O. Box 2476V
Melbourne 3001
Australia.

ABSTRACT:
This report describes a parallel router for printed circuit boards. Two different parallelisation strategies were employed; one decomposed the board into independent regions and then routed the regions concurrently; the other parallelised the wave-front algorithm which is used to find the shortest path between points. Whilst the first strategy is attractive because it requires very little synchronisation, it achieves poor performance even on boards in which most of the wires are independent. The second strategy is fairly efficient, but performs better on long wires than short independent ones. A combination of the two strategies is efficient for boards containing many independent short wires as well as some long wires which span the board. The paper supports the claims with experimental results taken from an Encore Multimax multi processor.

1. INTRODUCTION

Routing the tracks on a printed circuit board or integrated circuit can be extremely expensive, and typically consumes many hours of processor time. Many attempts have been made to speed the process by using special purpose routine engines [7,8]. More recently, there has been interest in how to route circuits on a parallel processor [3, 4, 5, 6, 9, 10, 11, 12, 13, 14]. This paper considers the use of commercial parallel processors to implement a parallel router for printed circuit boards. Routing involves finding a path from a source position to a destination position without crossing wires which have already been placed. Many different algorithms have been devised over the years [1]. The scheme described in this papers is based on a 'maze' router algorithm. This scheme was chosen because it is very general, and also lends itself to parallelisation.

The paper starts with a discussion of a general maze routing method called Lee's algorithm [2]. Two techniques for building a parallel router are then proposed, and some experimental results given. A number of problems with the model are shown, and some solutions given.

2. ROUTING USING LEE'S ALGORITHM

There are many different routing algorithms, and a good summary can be found in [1]. The oldest and most general technique uses a wavefront, which is advanced from the source to the destination. First described by Lee in [2], this scheme finds the shortest path between any two points, and is based on a fixed grid of cells. Cells either contains a value indicating that the board position is either occupied by a hole, circuit track, or part of an advancing wave. The cell corresponding to the source location is initialised with a low score value, e.g. 1, and is then added to a current wave front list. Then all of the cells surrounding the current wave front are set to the score 2, and are added to the wave front list. The initial cell is subsequently removed from the wave front list. This process continues until the wave either meets the target destination, or entirely fills the board. If the destination is not touched, then there is no path from the source to the destination. If the wave meets the target, then a path is traced by tracing a path of decending cell value from the destination until the source is reached. Figure 1 shows the wave advancement and path being retraced. This basic technique is enhanced to cater for practical considerations in the routing of printed circuit boards and integrated circuits.

Lee's algorithm only specifies how to route an individual wire. In order to route an entire board, wires must be chosen sequentially and processed individually. The order in which wires are chosen can effect the wiring density dramatically; a very common heuristic is to route the *short* wires before the *long* wires. This scheme tends to minimise the length of the resulting wiring track as well as minimising congestion, although some optimisations are possible. The routing algorithm described in the next section effectively routes the short wires first, and thus provides a reasonable wiring order.

The number of cells covered by Lee's algorithm can be greatly reduced by placing a frame around the source and destination points, as shown in Figure 2. This frame restricts the wavefront to the cells that are most likely to contain the path. If the target cannot be reached with the frame in place, then it can be removed and the route retried. The routing algorithm described in the next section effectively places a frame around the source and destination points, although some additional framing is used to further restrict the routing area.

The simple algorithm described above only advances the cell values by 1 each time the wave front moves forward. By weighting the increments, it is possible to favor particular paths. For example, it may be preferable to align tracks on one layer vertically, whilst aligning them horizontally on another layer. This may be achieved by using a different increment depending on whether the wave is advancing vertically or horizontally and which layer it is on. The scheme can also be used to force tracks to cluster themselves together, rather than breaking up free space.

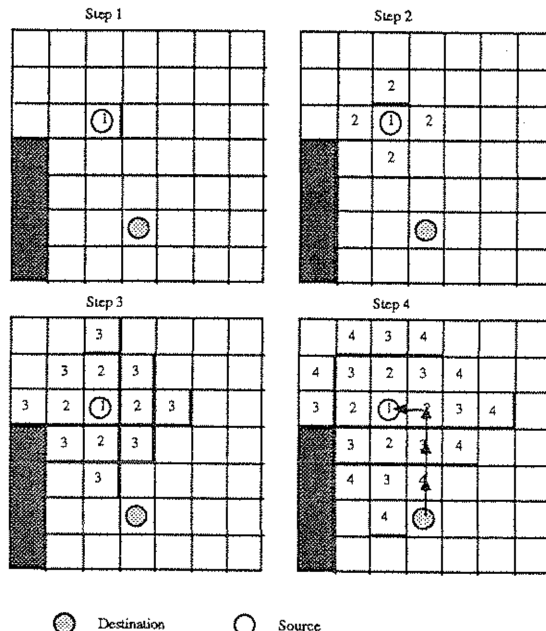


Figure 1 - Basic Lee's Algorithm

0073-1129/91/0000/0164$01.00 © 1991 IEEE

The scheme operates in the same way as the basic algorithm, except that the wave cannot stop advancing immediately after it reaches the target. Instead, it must continue until no score in the wave front is less than the score recorded at the target. Once this is the case, there can be no cheaper path, and the search can cease. Also, as the wave advances, it can overwrite a cell if it has a lower score than the one already recorded. Another difference in this scheme is that when the path is retraced, the algorithm must actually follow the steepest path down hill, rather than any path down hill. Later in the paper we will consider the effect that overwriting cells has on one of the parallelisation strategies.

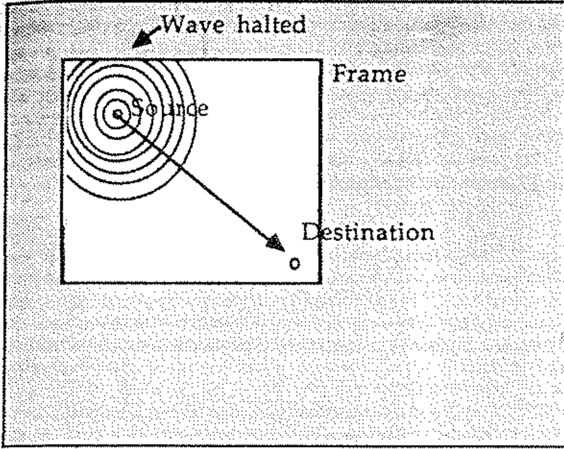


Figure 2 - Framing to reduce area covered

Lee's algorithm can easily be applied to multi-layer circuits by advancing the wave through via holes between layers. When the path is retraced, it may then pass between layers, avoiding congestion on any single layer. Vias can typically only be placed in certain grid points, thus the wave can only move between layers at these defined points. Clearance must also be left between a via hole and surrounding tracks. In order to minimise the number of vias created, a weight is usually applied to the score value as it moves between layers.

A problem with allowing an arbitrary number of layers is that the storage is greatly increased, and also the wave must travel much further than in the single layer case. A solution to this problem is to restrict the wave so that it can only move between two layers, and then to build a multi-layer board from pairs of layers. Thus, all possible wires are first created on two layers, after which two new layers are created, and the previous two saved. The advantage of this scheme is that the router need only manage two layers at a time. Vias created on previous pairs must be carried forward as new layer pairs are created. The disadvantage of these scheme is that it is not possible to river route through more than two layers. This scheme has been implemented in the parallel router, and seems to provide an effective solution to restricting the storage required whilst providing a multilayer capacity.

The biggest disadvantage of Lee's algorithm is that it is extremely time consuming. The number of cells filled is proportional to the square of the distance between the wire end points, although it can be reduced substantially by using framing.

3. A PARALLEL ROUTER - ATTEMPT 1

The basic parallel algorithm attempts to detect wires that are *unrelated* and route then concurrently. For example, wires that are at different ends of a board are likely to be completely independent, and thus could be routed at the same time. These wires are grouped by recursively dividing the board into regions, and passing the wires completely contained in a region to the router separately. Wires which are contained in more than one region are held until all those at lower levels have either been successful routed, or could not be routed because of congestion. This basic algorithm is illustrated in Figure 3, and described in Figure 4.

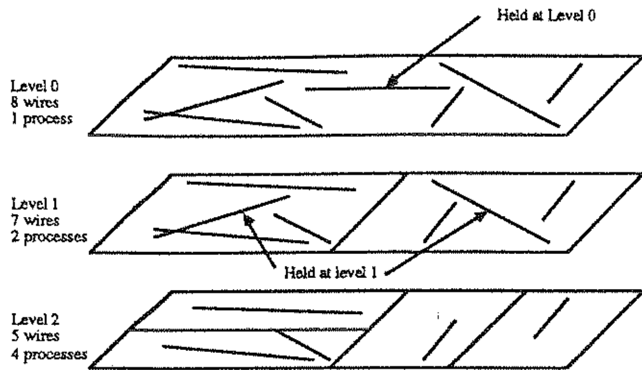


Figure 3 - example of recursive splitting

At each stage, the router calls itself, and splits the region it has been passed into two sections. It created three wire lists; two for those wires completely contained in each region, and one for those wires which cannot be partitioned. This process continues until either there are no wires left to route, or some preset maximum depth has been reached. After the routine has called itself recursively, it proceeds to route the wires it had held onto, and also attempts to route those which could not be connected by the recursive call. Thus, each call to the router accepts a wire list, and returns a wire list containing the wires it could not connect.

This process continues until it returns to the root of the call tree. If all of the wires have been connected then the router terminated, however, if there are wires left then it starts the process over again with two new layers. It is worth noting that this recursive procedure effectively sorts and routes the wires by length, because the shortest wires are passed down to the lower call levels, and are thus routed first. The scheme also implements framing, because a wire may not be routed outside its wiring area. Further framing is used to restrict the number of cels covered within a wiring area.

```
procedure route (          input   board, wiresleft, bounds;
                                   output  board, wiresleft)
begin
        if keepsplitting(...) then
                begin
                        newbounds := computemidpoints(bounds)
                        newwires(1..3) := splitwires(wiresleft);
                        for corner in 1,2 do
                                fork(route(board,newwires [corner],
                                                    newbounds[corner]));
                        mergewires(wiresleft,newwires);
                end;
        connect_the_wires(board,wiresleft, bounds);
end;
```

Figure 4 - routing algorithm

In itself, this recursive algorithm in sequential. However, by replacing the recursive call with a recursive *parallel* procedure call, each section can be routed concurrently. There is no need for any communication between the processes after the procedure call because they are all operating in separate areas of the board. If the algorithm is being executed on a shared memory multiprocessor, the board should be placed in shared memory to avoid copying it between calls. The reference parameters used in the call must also be placed in shared memory so that the results can be returned to the calling processs. If the machine is a distributed message passing machine, then the wire lists and board contents must be transmitted to the processor executing the called code.

3.2 Choosing the direction to split

The router splits the board space into two regions each time it calls itself. The simplest strategy is two split the space into two equal regions. Even with this simple case, it must choose whether to split vertically or horizontally, and when to stop splitting. The split direction should be chosen to maximise the concurrency which can be extracted. A suitable measure which can be used to determine which way to split is calculated by counting the number of wires which would be present in each partition of a vertical split (vertical 1 & 2), and the number in each partition of a horizontal split (horizontal1 & 2). The following section of code determines which split

direction should be used at each stage, and is aimed at maximising the available concurrency.

```
numhor := (horizontallist1 + horizontallist2);
numvert := (verticallist1 + verticallist2);
diffhor := greater(abs(horizontallist1 - horizontallist2),1);
diffvert := greater(abs(verticallist1 - verticallist2),1);
if      ((numhor * numhor) / (diffhor * diffhor)) >
        ((numvert * numvert) / (diffvert * diffvert)) then
                        direction := horizontal
        else
                        direction := vertical;
```

The splitting process can be terminated in one of two ways. First, if there are no wires, or less than some minimum preset amount left. Second, if some preset maximum depth is reached (e.g. stop after 3 levels of call). The number of wires required for the minimum value depends on the cost of the parallel procedure call. The value must be set so that it is worth performing the call. The maximum call level value is used to limit the maximum number of processes that are active at any time. Creating more processes than the number of available physical processors can be useful for controlling load balancing, but in general serves no real purpose.

3.3 Double recursion

The heuristic developed in the last section determines the best direction to split. However, this does not mean that the other direction is not worth considering. It is possible to try the other split direction once the recursive call has returned in the hope that there may be more potential concurrency. The code can split in the other direction either passing the wires from those which were held, or can include the wires which could not be routed by the first split.

3.4 Some results

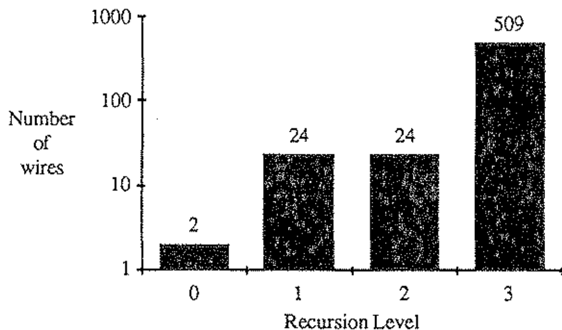
The algorithm described in the previous sections was tested with a circuit board containing 52 integrated circuits, 559 wires on a board measuring 5 by 8 inches. Table 1 below shows the execution time for varying number of processors

#Levels of recursion	Processes	Time	Speedup
0	1	180 mins	1
1	2	145 mins	1.24
2	4	120 mins	1.5
3	8	110 mins	1.6

Table 1 - Splitting results

The traces in Graph 1 show the effect of increasing the number of processors on the runs. An activity trace shows the number of processors running at any point of the computation. The progress trace shows the number of wires routed against time.

Graph 2 below is a chart showing the number of wires routed at each level in the recursion.



Graph 2 - Number of wires routed at each level

3.5 Summary of recursive splitting

It can be seen from the results presented in the previous section that simply splitting a circuit into independent regions is not sufficient to achieve good speedup. The problem occurs because the longest wires are held on until the fewest processors are available. As stated earlier, the time taken to route a wire using Lee's algorithm is proportional to the square of the euclidean distance between the end points. Because these are routed with fewer processors than the shorter wires, the process becomes dominated by the longer wires, and the resulting speedup is very poor. The phenomena is illustrated by the progress traces; increasing the number of CPU's only increases the initial slope of the curve. Similarly, the activity traces show the number of processes falling off exponentially with time. Thus, it can be seen that even though most of the wires (509 out of 559) are routed at level 3, with 8 processes, the computation is still dominated by the longer wires. However, the splitting algorithm performs very well if restricted to processing short independent wires as shown by the progress trace for 8 CPUS. This property will be exploited later when the recursive algorithm is combined with the parallel wave front algorithm.

4. A PARALLEL ROUTER - ATTEMPT 2

The second strategy used for parallelisation of the routing problem was to attempt to parallelise the wave front algorithm itself. Figure 5 illustrates how the wave front list is maintained as the wave is advanced from a source to a destination. Each time a cell is removed from the head of the wave front, and processed, a number of successor cells are placed on the tail of the list. In this example, cells v, w, x, y and z are currently on the list. When cell z is processed, it produces three new cells, z1, z2 and z3, which are appended to the tail of the list. The scheme used to parallelise this task assigns more than one process to advancing the wave. Two strategies were considered; one using a number of asynchronous waves moving from the start, and the other using a number of processes serving the one wave front list. Since the former requires very little synchronisation once the processes are started, it was expected that this would yield very high performance. The latter scheme requires the many processes to synchronise when they remove cells from the list, and should yield a poorer performance.

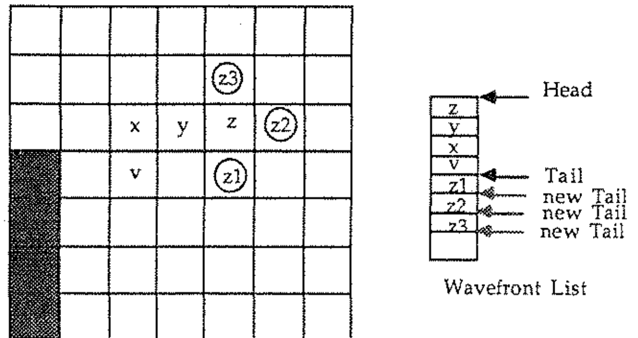
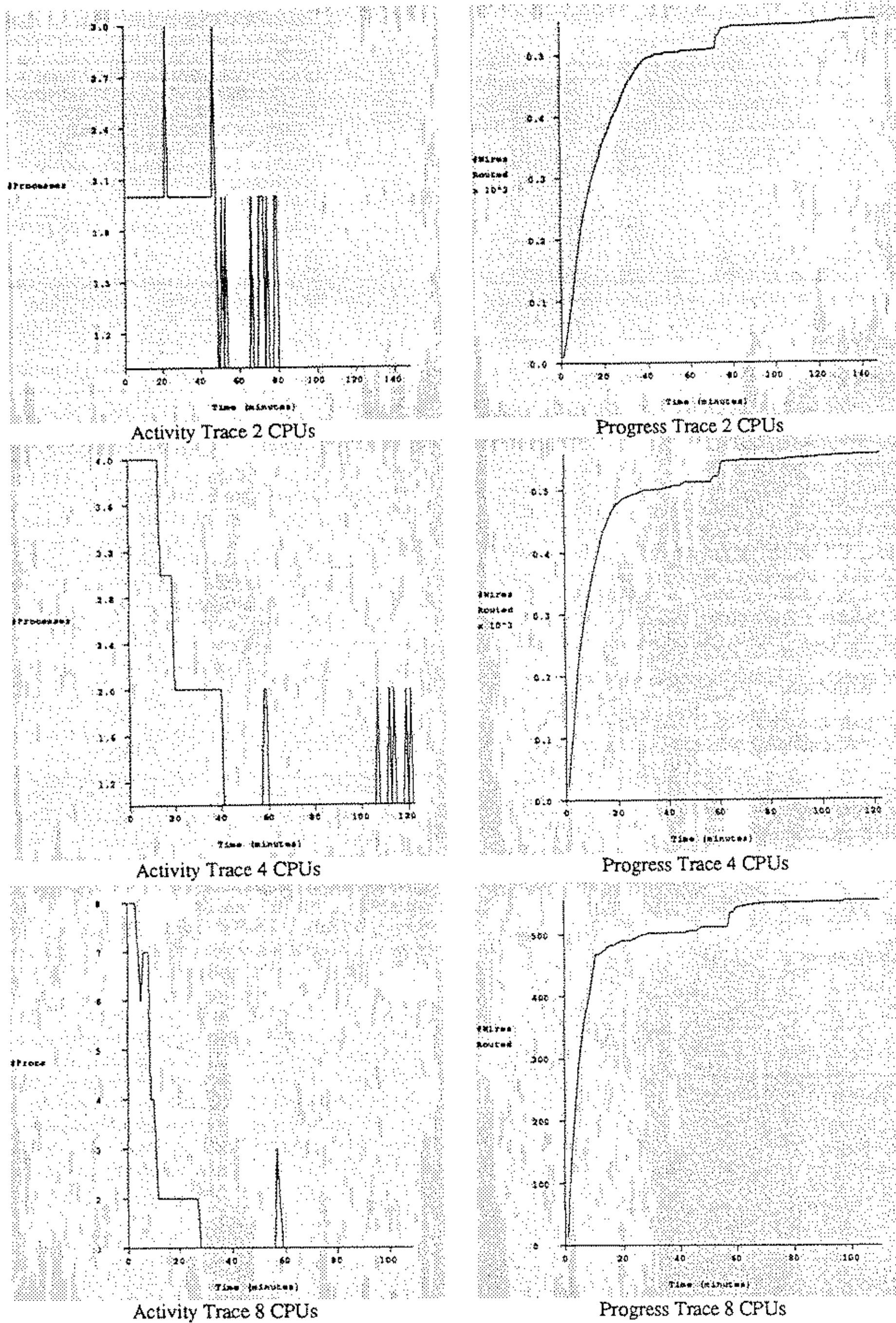


Figure 5 - wave front list management

4.1 Parallel wavefront - Asynchronous Advance with separate waves

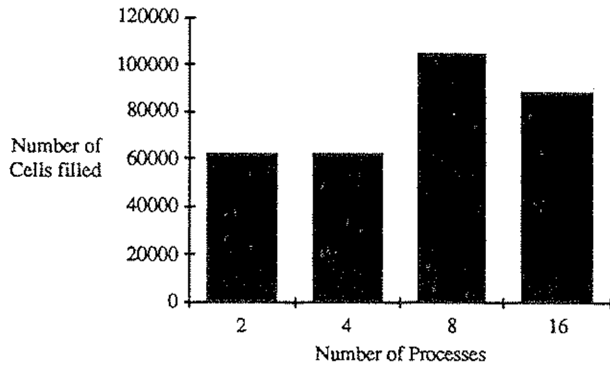
Figure 6 below shows how the wave can be considered as a number of independent waves, each advancing from the source. The algorithm begins by forking a number of processes, and passing them each a separate starting point. Once created, each process then advances its wave using its own wave front list independently of the others. When two process attempt to fill the same cell of the board because of the cell overwriting technique, some simple locking on the cell is performed. The process with the lowest cell value is allowed to claim the cell, and place it on its own wave front list. Rather than keep a lock word for every cell, the board is divided into regions, which contain many cells. Since the lock is held for only a very short time (just time for a cell read/write), there is very little interference between the processes.

Because this algorithm uses so little locking, we expected it to perform well. However, the speedup was extremely poor. The poor speedup is caused by two phenomena. First, when waves are advanced independently, they contend for cells on the boudaries of the waves. The statistics shown below in Graph 3 indicate that the number of cells filled is about the same



Graph 1 - Results of Recursive splitting algorithm

for 2 and 4 processes, but rises dramatically for 8 processes. Thus, in general, as more processes are added, the number of cells written increases, and the amount of work is increased. Consequently, the scheme yields very poor speedup. A number of experiments were performed to determine if the amount of cell overwriting could be reduced. One scheme advanced the wave from the source point sequentially, until the wave front was well clear of the source. Then, then wave front list was sorted into independent regions, and a number of new parallel waves started. This scheme still failed to reduce the amount of cell overwriting.



Graph 3 - Number of cells filled for multiple workers

However, the chart in Graph 3 also shows that the amount of work performed at 16 processes starts dropping again. Thus, one would expect the speedup to improve at higher number of processes. A problem with the asynchronous waves was that it was very difficult to create a number of processes and then balance the work load across them. For example, the framing scheme which was used to limit the number of cells visited, means that there were fewer cells to fill in the direction of the frame boundary. Thus, the processes which were responsible for these regions terminated long before the other processes which were heading in the direction of the destination. A number of heuristics failed to balance the load across the processors. The chart shown in Figure 7 shows the wide disparity in load.

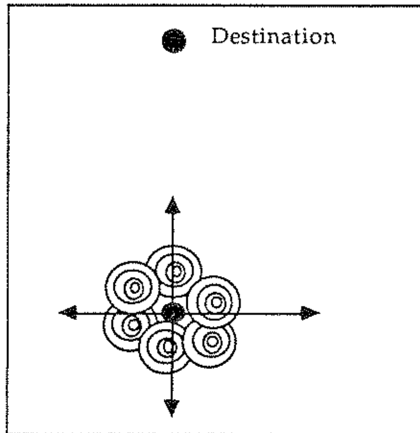


Figure 6 - Parallel waves

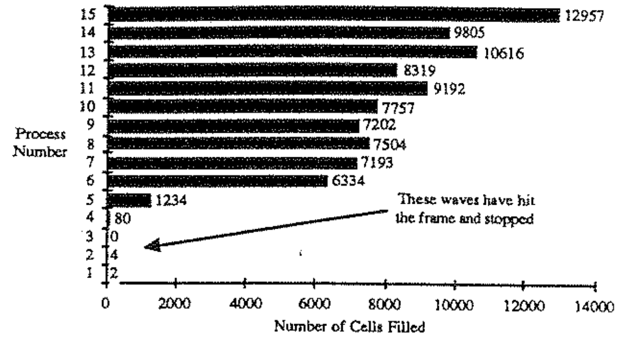


Figure 7 - load distribution in parallel waves

4.2 Parallel wavefront - One wavefront list

The other parallelisation strategy used assigned multiple processes to the task of advancing the wave, but used one shared wave front list. Each process synchronised when removing cells from the head of the list, and when adding cells to the tail. However, the computations performed on the cell were executed concurrently. This scheme out performed the multiple waves algorithm because there was no additional contention for the cells. The scheme basically behaves as though there is one worker serving the wave front list operating at a higher throughput. Locking code is still required to prevent more than one process from modifying the cell values simultaneously. The same locking scheme as in the multiple wave algorithm is used.

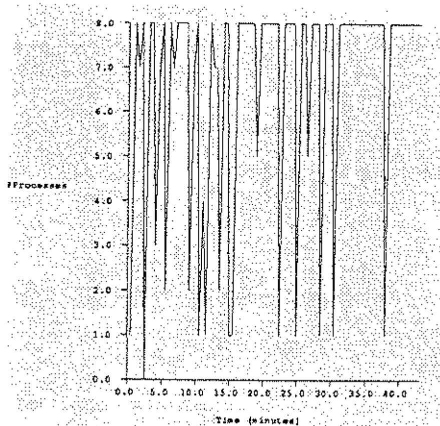
Below in Table 2 are the execution times for varying number of processes, working on the 559 wire problem. The activity and progress graphs are shown for 8 CPUS in Graph 4.

#Levels of recursion	Processes	Time	Speedup
0	1	286 mins	1
0	2	165 mins	1.7
0	4	81 mins	3.5
0	8	44 mins	6.5

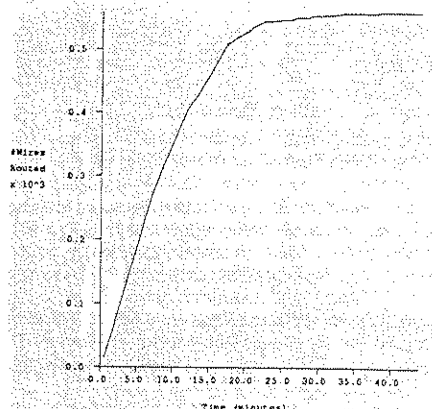
Table 2 - Parallel Wavefront algorithm

4.4 Summary of parallel wavefront

The experiments have shown that although the asynchronous scheme of advancing waves seems attractive because of the lack of a shared wave front list (which can become a bottleneck), the algorithm performs so much extra work that it is inferior to using one shared wave front list with multiple servers. The parallel wavefront algorithm performs better on long wires than short ones because it spends significant time advancing the wave front, and overcomes the overheads of starting the servers and combining the results.



Activity Trace 8 CPUs Parallel Wave



Progress Trace 8 CPUs Parallel Wave

Graph 4 - Results from parallel wave algorithm

5. COMBINATION OF TWO SCHEMES

The two algorithms which were used each have their strengths. The recursive splitting algorithm is very efficient at processing a large number of independent wires. The parallel wave algorithm is efficient at routing long wires. Some experiments were performed on using a combination of the two parallelisation strategies. In the combined scheme the recursive splitter is used for the initial short independent wires, but the parallel wave front algorithm is used for the later longer wires. Table 3 shows the times for the combined router. Graph 5 shows the activity and progress traces for the combined router with 8 CPU's.

#Levels of recursion	Processes	Time	Speedup
0	1	not avail yet	-
1	2	120 mins	1
2	4	63 mins	1.9
3	8	37 mins	3.2

Table 3 - Combined parallel wave and splitting results
Note: The times for 1 process are not yet available, so speedup has been quoted over the 2 process execution

6. CONCLUSION

This paper has presented the design of a parallel router for printed circuit boards, as well as some experimental evidence from a commercial shared memory multiprocessor. The results are interesting because they highlight some techniques which originally appeared promising as being inefficient. The recursive splitting algorithm performed poorly because the small number of wires which were held until the end required the greatest resource to route them quickly. The results of the recursive splitting algorithm can be carried over to many other problems. The use of asynchronous wave was surprising because they should have produced a better speedup than synchronous waves that share a single list. In this case the algorithm created much more work, and thus did not speed up as expected.

The work described in this paper is similar to much of the work that has already been reported in parallel routing. However, some of the conclusions are different. Most of the published material addresses routing of integrated circuits which use row based layouts. In these schemes, a global routing phase is executed in which global wiring channels are established before the detailed channel routing is performed. This scheme works well for standard cell, gate array and sea-of-gates technology. In this style of global routing, a hierarchical scheme, as described in [3], [12] and [14], works quite well. The reason is that the time spent on each wire does not increase dramatically as the hierarchy returns to the route. For example, in the test case used in this paper, only 2 wires were actually routed at the top layer. Thus, even though there is only one processor available for the top layer, the speedup is still quite good.

Even if detailed routing is performed, the use of a row-based scheme can dramatically assist a parallelisation strategy. For example, Rose [4] need only consider two bend routes for each subnet. The maze algorithm we chose can generate many bends and route around many other prelayed tracks. Thus, it is very difficult to lay down more than one track at a time when they could use the same routing cells.

However, the router described in this paper was targeted for printed circuit boards, in which the chips are often placed by hand, and do not necessarily conform to row-based routing schemes. The reason for the emphasis on circuit boards rather than integrated circuits is that the former is still the dominant form of product developed by Australian industry. Given that we applied a maze router for the detailed routing of wires, the time to route wires increased dramatically as their length increased. Thus, our hierarchical router performed very badly, unless restricted to short independent wires.

The second parallelisation strategy we adopted is similar to the scheme used in the hardware based routers [7,8]. These machines basically ignore wire-level parallelism and exploit the low level concurrency inherent in the maze routing algorithm. We observed quite good speedup in this algorithm, but its efficiency is lowered by the software overheads in extracting such low level concurrency. Thus, this parallelisation is better for long wires in which the overheads can be absorbed.

The combination works quite well for both long and short wires when channel routing schemes are not appropriate.

ACKNOWLEDGEMENTS

The Parallel Systems Architecture Project is a joint project between the Commonwealth Scientific and Industrial Scientific Organisation (CSIRO) and the Royal Melbourne Institute of Technology (RMIT). Thanks go to the members of the Parallel Systems Architecture Project for their helpful discussions.

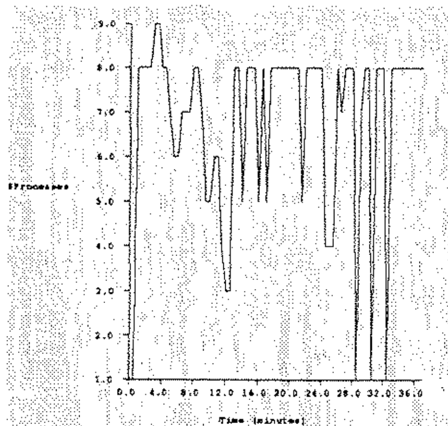
REFERENCES

[1] IEEE Transactions on Computer-Aided Design of Integrated circuits and Systems, October 1983, CAD-2, Number 4.

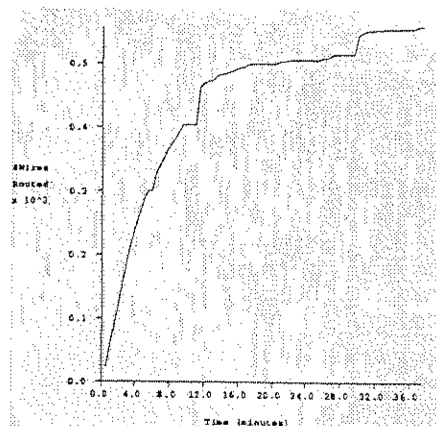
[2] Lee, An Algorithm for Path Connection and its Applications, IRE Transactions on Electronic Computers, Vol EC-10, No 3, pp 346 - 352, September 1961.

[3] M. Burstein and R. Pelavin, "Hierarchical Channel Router", Proc 20th Design Automation Conference, pp 591-597, June 1983.

[4] J. Rose, "LocusRoute: A Parallel Global Router for Standard Cells", Proc 25th Design Automation Conference, pp 189-195, June 1988



Activity Trace 8 CPUs Parallel Wave &Split



Progress Trace 8 CPUs Parallel Wave &Split

Graph 5 - Combined splitting and parallel waves algorithm

[5] O.A. Olukotun and T. Mudge, "Preliminary Investigation into Parallel Routing on a Hypercube Computer", Proc 24th Design Automation Conference, pp 814-820, June 1987.

[6] Y. Won and S. Sahni, "Maze Routing on a Hypercube Multiprocessor Computer", Proc. Int. Conference on Parallel Processing, pp 630-637, August 1987.

[7] R. Nair, S. Hong, S. Liles and R. Villani, "Global Wiring on a Wire Routing Machine", Proc 19th Design Automation Conference, pp 224-231, June 1982.

[8] T. Watanbe, H. Kitazawa and Y. Sugiyama, "A Parallel Adaptable Routing Algorithm and its Implementation on a Two-Dimensional Array Processor", IEEE Transactions Computer-Aided Design, vol CAD-6, No 2, pp241-250, March 1987.

[9] K. Lee and C. Sechen, "A New Global Router for Row-Based Layout", Proc. Int. Conference Computer-Aided Design, pp 180-183, Nov 1988.

[10] J. Cong and B. Preas, "A New Algorithm for Standard Cell Global Routing", Proc Int. Conference Computer-Aided Design, pp 176-179, Nov 1988.

[11] M. Marek-Sadowska, "Global Router for Gate-Array", Proc Int. Conference Computer Design, pp 332-337, Oct. 1874.

[12] W. Luk, D. Tang and C. Wong, "Hierarchical Global Wiring for Custom Chip Design", Proc 23rd Design Automation Conference", pp 481-489, June 1986.

[13] P. Tzeng and C. Sequin, "Codar: A Congestion-Directed General Area Router", Proc. Int. Conference Computer-Aided Design, pp 30-33, Nov 1988.

[14] R. Brouwer and P. Banerjee, "PHIGURE: A Parallel Hierarchical Global Router", Technical Report, Centre for Reliable and High Performance Computing, University of Illinois at Urbana-Champaign,