ELECTRONIC PUBLISHING, VOL . 2(1), 25–46 (APRIL 1989)

Parallel processing in document formatting: an experiment using PIC

DAVID F. BRAILSFORD AND DAVID R. EVANS

Department of Computer Science
University of Nottingham
NOTTINGHAM NG7 2RD
UK

SUMMARY

The manipulation of text and graphics within a computer provides opportunities for the exploitation of parallel processing. It is straightforward to identify blocks of material such as complete diagrams or paragraphs of text which can be processed in parallel and which have modest requirements for synchronization and communication between the blocks. The features of a problem which lead to an elegant and efficient application of parallelism are identified, including good locality of reference, a small 'state vector' of shared global variables and a clear relationship between the material on the page and the 'cost' of processing it. This last-named attribute enables a problem to be partitioned among multiple processors by a static compile-time analysis rather than relying on costly run-time allocation strategies. The PIC program for line diagrams has been modified to allow for such a static allocation and to permit synchronization and rendezvous between multiple invocations of the program. The aim of this was to investigate whether worthwhile gains in performance would result from subdividing a diagram drawn with PIC and then processing the various portions in parallel. A series of benchmark timings is presented which show the degree of overlap obtainable in processing separate parts of a diagram together with the inherent limits to parallelism imposed by the 'atomic' entities in PIC and the inevitable communication overheads between the parallel processes. The design features of the PIC language are identified which made it suitable for these researches and we are able to draw certain general conclusions about desirable properties of text and graphic entities which are to be processed in parallel. This in turn enables us to identify design features of the underlying software which will facilitate parallel processing.

KEY WORDS Document processing Parallel processing PIC Benchmarking

INTRODUCTION

It is a widely recognised but sparsely chronicled fact that Parkinson's First Law applies just as surely in computer science as it does in economics: whatever the speed and memory capacity of one's computer it is virtually certain that the number and complexity of the programs running on it will rise so as to match its capabilities and then to outstrip them. Indeed, there have always been application areas in computing whose demand for hardware resources seems insatiable, but it is hard for inexperienced users to believe that document processing could be one of them. However, when it comes to slowing down a multi-user machine, users of UNIX systems will testify that few things can compete with the deadening impact of several dozen simultaneous invocations of the *troff* [1] text

0894–3982/89/010025–22$11.00

Received 11 October 1988
Revised 24 January 1989

formatter. The reasons for this soon become clear: not only is there considerable input/output activity when running text formatting software but also the apparently straightforward tasks of justifying text, laying out tables, placing subscripts and so on impose a sizeable computational load because of the sheer number of calculations that have to be performed.

For these reasons text and document formatting will always be a field where extra computer power is welcome. Over the years a steady fall in memory and processor costs has enabled all computer users to run larger and more computationally intensive programs simply by purchasing more powerful versions of the traditional, von Neumann, computer architecture. Like other classics of good design, this architecture seems to defy all predictions of its impending obsolescence and early demise. Instead, it remains stubbornly competitive by means of various enhancements and clever tricks, ranging from memory caches and pipeline registers to the off-loading of input/output activity onto a variety of peripheral processors.

In the long run, though, the performance of any single processor, whatever its architecture, will be limited by factors such as the physical properties and dimensions of the materials used within the chips themselves. So what should our strategy be for achieving further increases in speed, once the limits of fabrication technology have been reached? The attraction in choosing parallel processing as a solution is that, whatever the fabrication technology of a processor and whatever its performance limits, there is always the prospect of devising a suitable partitioning of a computer program among n such processors, with the hope of achieving an n-fold speed increase. Further gains in performance can then be realised, so the argument goes, by simply increasing the value of n. Of course, nothing is ever this straightforward: the interconnection topology between the processors, and its suitability for the problem in hand, is just one of the many factors which determine whether communication between the processors can be handled fast enough to prevent overall performance being degraded.

The early sections of this paper give a brief introduction to parallel processing, for those unfamiliar with the issues involved, and we point out some of the difficulties in achieving the n-fold speed increases referred to. In later sections we relate the general requirements for parallel systems to the specifics of text processing, illustrating various points with the aid of experimental results from a modified version of PIC [2] (which pre-processes line diagrams for the *troff* text formatter). Given that much of the existing literature makes great play of the need for new languages to exploit parallelism fully, it might seem eccentric to choose an existing piece of software to highlight the points we want to make. However, it is a testimony to the economy of PIC's design and implementation that, despite being a 'conventional' language with no obvious concessions to parallelism, we needed to make only a handful of small modifications to adapt the program for our investigations. In the course of describing why PIC is so suitable for our purposes we shall be able to identify some of the features that future text processing software should possess if it is to be adaptable for parallel processing.

In a paper of this sort it is not possible to review the multitude of existing software and hardware systems for parallel processing. Many novel architectures for exploiting parallelism have been proposed and built over the past twenty years, ranging from array and vector processors through to more recent dataflow, graph-reduction and connection machines. Those interested in further details will find a useful survey in [3] and some more recent introductory articles in [4].

PARALLELISM — RECOGNITION AND IMPLEMENTATION

For any problem to be readily amenable to parallel processing there are certain requirements which bear on the nature of the problem itself, the algorithms and data structures used for implementing it and the characteristics of the implementation language. Topics such as image processing and matrix manipulation possess a high degree of replication in the operations to be performed which, in turn, facilitates a partitioning of the problem into a multitude of independent, and almost identical, sub-tasks. Such problems can be mapped onto specialised array processor or systolic array computers, leading to speed increases of a thousandfold or more. On the other hand there are problems which are almost completely serial, both at a macroscopic and a microscopic level, with each stage of the problem being required to take place in a strictly pre-determined sequence. Between these two extremes lie most of the problems encountered in computer science applications (including the investigations presented in this paper). In these cases, parts of the program are capable of being executed in parallel but there are various intervening serial 'bottlenecks' where a rendezvous of preceding parallel sections has to take place.

Granularity of parallelism

In deciding how to map a problem on to some set of available processors, one of the key factors is the 'granularity' of parallelism that is to be sought. We can illustrate this in the familiar context of conventional programs by examining various choices for the indivisible unit of parallel programming. If this unit is the whole program itself, then we are talking about a system which can run several programs concurrently or pseudo-concurrently — which is exactly what modern multi-user operating systems can do very effectively. The reason for their success is that the degree of interaction and communication between the programs is usually quite small and so any synchronization requirements between the programs can be handled by the kernel of the operating system.

At a finer level of subdivision it often proves useful, inside a given program, to allow multiple invocations of procedures (or groups of procedures) to be initiated as separate concurrent tasks. Such a facility is sometimes called *multi-threading* and it mimics, on a smaller scale, some of the features of a multi-process operating system. This particular form of multi-tasking is supported by several operating systems and finds widespread use in a range of real-time software applications. The only extra complication of subdividing a program in this way is the potential problem when the various processes attempt to write to some shared area of memory at the same time. Variables which are local to procedures cause no difficulties — their various instances are kept separate on some form of multiple stack mechanism provided by the multi-tasking environment — but unconstrained access to *global* variables could lead to these variables being over-written by the various tasks in arbitrary ways at arbitrary times. A system of controlled access to all global entities has to be devised, and the classic paper by Dijkstra [5] introduced the concept of *semaphores* for just this purpose.

At a yet more microscopic level we might ask if it would be worthwhile, inside a procedure, to perform all its assignment statements, conditional clauses and so on in parallel? Might we go even further and suggest that operands in expressions could be fetched and evaluated in parallel? In proposing this, we begin to recognise that operand

values represent the ultimate atomic components of conventional programs and that we are coming very close to the maximal theoretical parallelism attainable from that sort of system.

At such a stage it is evident that once there are enough processing elements to satisfy the multitude of individual calculations in a maximally parallel system there is absolutely no point in adding any more processing power. What tends to happen, anyway, long before maximal parallelism is reached, is that the communication overheads between the various small parallel sub-tasks start to outweigh any gain in efficiency due to the notional extra parallelism achieved. Beyond this optimum point performance will not just level off as the number of processors is increased, it may actually start to decline, as the communication overheads begin to swamp the computation. The problems arising from unconstrained fine-grain parallelism — very often at just this level of arithmetic sub-expressions — are well illustrated in some dataflow machines [6], where a huge pool of partly executed sub-expressions can accumulate, giving rise to serious problems both in storage and in dynamically routing them to the available processors for execution.

To relate these factors to text processing, we see at once that there is a wide variety of logical and physical entities (sentences, paragraphs, chapters, diagrams and so on) which would be good candidates for the atomic blocks of any parallel system. The existence of abstract entities in the Office Document Architecture (ODA) model which make it suitable for the application of parallelism has been pointed out in a recent paper by Brown [7]. On the other hand, if we go to extreme levels of granularity, for example by placing individual text characters on the page using one process per character, we would be likely to encounter all the communication overheads already described.

Language issues

When using parallel processing systems we seek to express a problem at a level of granularity which affords an allocation among n processors, so as to minimise the communication overhead and to maximise the amount of computation that can truly proceed in parallel. We also need an implementation language which limits, or abolishes, the side-effects which arise when one or more processes have access to some common global variable without taking care to synchronize that access. In other words we need a formulation which minimises the 'state vector' of global entities and the communication overheads between processes, while still providing some form of inter-process synchronization, possibly in the form of semaphores.

Conventional languages are notorious for allowing users to have almost unrestricted access to global variables, and if this access is abused it can lead to a host of unlooked-for side-effects and bugs [8] For this reason such languages seem at first to be totally unsuitable for parallel execution; this has led many workers to pursue parallel programming systems within the context of functional or logic languages, where it is claimed that complete freedom from side-effects can be obtained because destructive assignment to global variables is not a part of the language semantics. Approaches of this sort have led to the development of graph reduction machines for functional and logic languages, exemplified by hardware such as ALICE [9] but, here again, the problem of communication overheads can still arise because of the fine-grain packet-based nature of the system.

Fortunately, it is not essential to embrace functional programming in order to enjoy

the benefits of parallelism. Progress is possible with conventional languages provided that the state vector of global variables can be kept small and that all changes to these variables can easily be detected. Synchronization is needed whenever the global variables are altered, but, if these alterations can be identified as the code is being compiled, then the appropriate inter-communication and synchronization can be arranged as part of the static allocation of the problem among the processors.

Once the issue of access to global variables has been sorted out we then require that communication overheads be minimised. In this respect it helps greatly if the separate portions of the problem have good *locality*; which is to say that the proportion of intra-process to inter-process activity is as high as possible [10].

Static *vs* dynamic allocation of parallel tasks

At first sight the most efficient usage of a parallel system would occur if one had a dynamic allocation of tasks to processors. However, experience with such systems shows that careful design is needed if the overheads of dynamic scheduling and allocation are not to counteract any gains from the elimination of idle time among the processors. These communication penalties can become very serious when a dynamic allocation algorithm, striving for maximal parallelism, attempts to send small packets of information to a highly distributed system.

Things become simpler if it is possible to allocate work to the available processors during the compilation or pre-processing of the program text, so that all of them will be kept busy during the total time taken to process the problem. The ease with which such a static allocation can be achieved depends not only on the nature of the task, but also on the kind of programming system that is employed to solve it. For all the freedom from side-effects enjoyed by functional languages, or by suitably constrained versions of logic languages, it is often difficult to express a problem in the particular style required and even more difficult to predict, however approximately, what the run-time behaviour will actually be. In this regard the various components of document processing score well; the languages used to process documents are usually of conventional design and it is broadly the case that the amount of processing power needed for a block of text, say, will be proportional to the length of that block. It is also easy to detect more complex material, e.g. equations and tables, which will lead to a heavier processing load. In the case of line diagrams, which are the focus of our studies here, it is straightforward to estimate the computational overhead for processing elements such as boxes, lines, circles, splines etc. and to make a sensible compile-time guess as to the amount of work to be allocated to each processor.

PARALLEL PROCESSING WITH PIC

If a document contains several line diagrams then the size, coordinate system and so on for each of them will usually be independent of all the others, except perhaps for constraints during the final pagination process which might require two related diagrams to appear on the same page. There are clear-cut advantages, therefore, in striving for inter-diagram parallelism, where each diagram is processed separately but in parallel with all other diagrams. All of these remarks apply with equal force to suitably-sized units of plain text. It seems reasonable, for example, that paragraphs could be processed

in parallel but the gains in doing this are clearer to see in a system such as *troff*, where paragraphs are almost totally independent of one another, than it would be in TEX [11], where the more sophisticated vertical justification algorithms might lead to heavy communication traffic before paragraphs could be placed correctly.

In the present paper we wish to delve even deeper and to ask whether further worthwhile gains in processing speed could be achieved from a finer-grained parallelism. For line diagrams do we gain anything by subdividing each diagram into portions, then processing all these portions in parallel, and then arranging a suitable synchronization and rendezvous once all the segments of the picture are complete? Would there be enough innate parallelism within the average paragraph or the average picture to justify this extra effort?

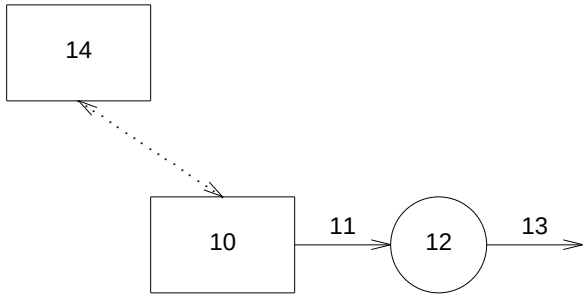
To answer these questions we looked for some existing piece of software which could be modified, with little effort, to allow for a simulation of a parallel system. As we have seen, one of the major requirements is that the set of global variables allowed by the software should be manageable in number and it was for this reason that we chose the PIC compiler for line diagrams as our model system.

In order to carry out our experiments it was necessary to modify PIC a little to allow processing to take place on a simulated parallel system. Before giving details of these modifications, and the parallel algorithm used, it is necessary to explain, for those unfamiliar with PIC, that it is a piece of software where one *describes* the picture to be drawn using a set of graphic primitives such as `line`, `circle` and so on, together with positioning information and details of any text to be inserted in or around the graphic elements.

For example the PIC source code:

```
arrow <- left "13" above
circle "12"
arrow <- "11" above
box "10"
line dotted <-> from last box .n up boxht left boxwid
box "14" with .s at end of last line
```

produces the picture



A few points can be noted at once from this example. Firstly, the description is almost self-explanatory — particularly after comparing the source text with the final picture — and global variables such as `boxht`, and `boxwid` appear from time to time.

Above all, the descriptive style encourages use of *relative* positioning information in the neighbourhood of objects ('`above`', '`at end of last line`', and so on, or by the use of `.c` to signify the centre, and `.n`, `.s`, etc., to denote the various compass points around the periphery of an object).

The PIC compiler builds up a data structure in the form of a linked list, as the PIC source code is parsed, with each item on this list representing one of the objects (`box`, `line`, `circle`, etc.) in the description. The coordinates of each object are centred at some reasonable place on the object (e.g. the position coordinates for a circle are those of its centre) and, as the picture is built up, the coordinates of each object are calculated on a relative basis, by a form of 'dead reckoning', given that the first object described in the PIC source is placed by default at the origin. It has to be decided exactly where this local origin will be placed on the page so that the picture will fit, bearing in mind that parts of the picture will lie above, below, to the left and so on of this arbitrary starting point; moreover any request to draw huge objects may lead to the whole picture needing to be shrunk in size. To keep track of all this information PIC updates its notion of the overall 'bounding box' for the diagram as each new element is encountered. Once the picture is complete, the linked-list is traversed and the output phase is entered; every element on the linked list is mapped into one or more drawing commands for later processing by the *troff* package but the relative coordinates now undergo a linear transformation, guided by PIC's knowledge of the bounding box, so that the drawing commands passed to *troff* incorporate correct absolute coordinates.

In our parallel scheme using PIC we allocate the source code onto a set of *streams*[12] Each of these streams is a process running a modified PIC compiler and the whole system runs pseudo-concurrently, almost as classic coroutines. It is necessary to extend the PIC language slightly by adding synchronization primitives between the streams called `send` and `recv`. These operations are inserted into the source coding by a pre-processor and they behave like a combined semaphore and message-passing mechanism, with the appropriate source or destination stream being one of the arguments to the `send` or `recv` call. Synchronization between the streams ensures that the various portions of the diagram join up correctly and that the overall bounding box of the picture is correct. Note that we do not attempt to subdivide the PIC program itself into routines running in parallel, nor do we address the possibilities for fine-grained parallelism in the process of imaging lines and points on raster devices. Indeed the atomic objects within PIC such as `line`, `circle` and `box` continue to be treated as indivisible within our scheme; there is no attempt, for example, to further subdivide boxes into line segments or lines into points.

Parallel simulation implementation

The parallel simulation (see figure 1) works by first pre-processing the picture description, to assess the amount of work to be done, using a 'weighted object' algorithm to be described later. All the PIC statements are passed on, unchanged, to one or other of the streams which are running the parallel version of PIC. From time to time the pre-processor inserts into its output various `send` and `recv` primitives, to effect the necessary inter-stream communication, and these primitives are recognised and acted upon by the modified version of PIC running on all the streams.

The message passed via the `send` and `recv` primitives is a set of coordinates,

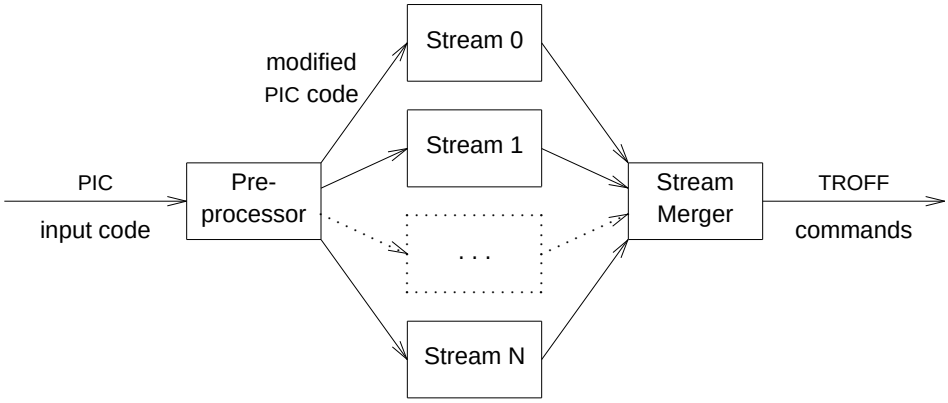


Figure 1. Simulation implementation

denoting either a synchronization point (which will enable two distinct portions of the diagram to join up correctly) or the corners of the bounding box for some particular stream. These primitives are implemented in our simulation by sending the message to a file (whose name is generated from the parameters passed to `send`), and then receiving the message, on another stream, from the same file. The format of calls to `send` and `recv` is:

```
send(s1, p1, s2, p2, position, type)
recv(s1, p1, s2, p2, type)
```

where:

`s1` = source stream number (coordinates are being sent from this stream)
`p1` = unique place number in source stream
`s2` = destination stream number (this stream receives the coordinate information)
`p2` = unique place number in destination stream
`position` = coordinates to be sent (or 0 if none)
`type` = 0 for synchronization, 1 for bounding box.
(N.B. the place numbers are used to create unique file names for message passing.)

Both the `send`s and the `recv`s are queued internally if the message to be sent or received is not yet available. This means that the `recv` primitive will not block unless there is no other work to be done on the receiving stream, and the message has still not arrived when the internal queue of pending synchronization requests comes to be processed.

A final synchronization is required, once all the streams have parsed their PIC code, so that they may agree where the origin is and the size of the global bounding box. (During "parallel processing" each stream has its own *local* coordinate system, hence the need for synchronization.) Just before the final synchronization and rendezvous there may be idle time on some of the streams, either because of unbalanced workloads or because a particular stream is now blocked on a `recv` and is waiting for a `send` to be executed elsewhere.

In addition to participating as a normal PIC stream, stream 0 is nominated to take the governing rôle for the rendezvous, and broadcasts the final bounding box to all streams. Each stream then has to *adjust* its coordinates so that they become relative to the finally agreed origin. The output phases on all streams can then run completely in parallel with all output being collected by the 'stream merger', which passes a single stream to the usual *troff* pipeline. The 'stream merger' also preserves the correct ordering of output by collecting output from `stream 0` to `stream N` in that order, so that any position-dependent commands embedded in the PIC source-code should still work as expected.

The streams themselves are simulated by having the parser fork separate UNIX processes (as described in [13]), each running the parallel PIC implementation. These separate invocations of PIC compete for processing time, and the total CPU time in each process is monitored using the `gprof` performance profiling software available under UNIX.

To illustrate the method let us take a rather more elaborate version of our previous diagram and see how we might allocate PIC source code to four parallel streams, as indicated by the bold headings. Note that the only syntactic additions to the PIC language are the `send` and `recv` primitives which effect the requisite synchronization between sending and receiving streams.

Stream 0

```
reset
box "1"
arrow "2" above
circle "3"
arrow "4" above
ellipse "5"
arc -> cw
send(0,0,1,6,Here,0)
recv(1,27,0,28,1)
send(0,29,1,30,0,1)
recv(2,31,0,32,1)
send(0,33,2,34,0,1)
recv(3,35,0,36,1)
send(0,37,3,38,0,1)
```

Stream 1

```
down
recv(0,0,1,6,0)
box "6"
arrow <-> "7 " rjust
box "14"
move to last box .w
arrow <- left "13" above
circle "12"
send(1,0,2,14,Here,0)
send(1,27,0,28,0,1)
recv(0,29,1,30,1)
```

Stream 2

```
left
recv(1,0,2,14,0)
arrow <- "11" above
box "10"
line dotted <-> from last box .n up boxht * 2 left boxwid
move down boxht * 5 / 2 then right boxwid / 2
left
arrow <- "9" above
box "8"
send(2,0,3,22,Here,0)
send(2,31,0,32,0,1)
recv(0,33,2,34,1)
```

Stream 3

```
left
recv(2,0,3,22,0)
move right boxwid * 5 / 2 + linewid * 3 + circlerad * 2
then down boxht / 2
spline -> from Here then down .2 then left .5 down .5
box wid boxwid * 3 / 2 "15"
send(3,35,0,36,0,1)
recv(0,37,3,38,1)
```

Figure 2 shows the picture generated by this source text when it is executed on our simulated parallel system. To make clear what is going on we have added the extra embellishments of the bold labels '$\mathbf{S_0}$', ... , '$\mathbf{S_3}$' (to show where the streams begin), and the black bullets ,●, to show the synchronization points between adjacent streams. In what follows we shall refer to this picture as the 'simple diagram'.

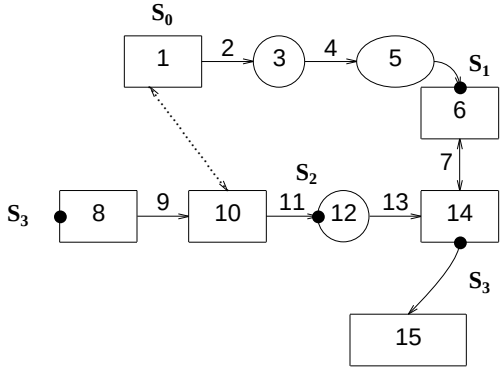


Figure 2. Simple diagram, with bullets showing synchronization points

More details of the algorithm for allocating source code to streams are given in the next section.

STREAM LAYOUT ALGORITHM

When allocating PIC commands to streams there are two requirements that the layout algorithm must fulfill in order to achieve a good overlap of processing:

(i) It must endeavour to partition the *total* amount of work evenly amongst the streams, and give enough work to each, so that the communication overhead does not overwhelm the benefits gained by exploiting any parallelism. If the workloads on each stream are similar then the majority of processing should occur in parallel. Also, any necessary synchronization at the end of this processing should then occur at about the same time, so that no stream will be idle for very long.

(ii) The second requirement is that the work assigned to any particular stream be as independent as possible of any other stream. This will minimise the inter-stream communication and so reduce the communication overheads.

A rough and ready algorithm for allocating the PIC input code would be to assign each stream the same number of objects.[1] In practice this simple algorithm is unlikely to produce streams with well-balanced work loads, since different PIC entities require quite different amounts of processing. For example, each separate object has its own bounding box, which has to be calculated in order that the stream may eventually calculate a bounding box for its own portion of the total picture. If the object is a line or circle this computation is trivial, but for an arc or spline there may be many expensive floating point calculations to be made.

A better algorithm involves taking into account the varying amounts of work involved in processing different objects. We have adopted this 'weighted object' algorithm in our modified PIC parser, each object being assigned a weight corresponding to the amount of processing that it requires. Each stream is then allocated objects with about the same total weight, so as to achieve a balance of work between the streams and to minimise the 'idle' time before the final synchronization (when the global bounding box is computed).

The weights for the objects are shown in table 1. Their choice involves some measure of guesswork (and, with hindsight, it would have been better to increase the weighting given to arcs) but the exact values are not too important given that the algorithm only tries to bring the stream workloads into approximate balance.

Table 1. Object weights

Spline	8	*Arc*	1
Box	4	*Line*	1
*Arrow**	3	*Move*	1
Circle	1	*Text*	1
Ellipse	1		

*An arrow's weight varies depending on how many lines are used to draw its head (the default is two).

The assignment of PIC objects to streams is now straightforward. The source code is allocated to streams more or less as it is encountered, but with the proviso that boxes, lines and so on are logical entities and so should remain intact. Conditional statements,

[1] In PIC, objects are the basic picture units such as boxes, circles, lines, splines and text strings but there is also a higher level object called a *block*, which is essentially a collection of objects in an independent sub-picture.

macros and loops should also be kept as single units, otherwise there will be a need for more inter-stream communication, and this is to be avoided where possible. Some care is needed **not** to take a new stream immediately before a phrase such as '`circle at end of last line`' which would at that stage refer back to some unknown object on a previous stream.

Another problem is to keep track of the 'global state' between streams and, in particular, the default sizes for the various objects. For instance, if stream 0 were to change the default size for a circle, then it would need to inform all the streams that came after it of what the new default should be. Mercifully, the global state in PIC is fairly small, with only eighteen global variables, and it is entirely feasible to look out for changes to any of these without slowing down the stream allocation algorithm too much.

Although PIC has a macro facility it has only a limited notion of local variables and restricted scopes. This can lead to problems, in our parallel version, when users declare their own variables. All such variables have global scope but, fortunately, alterations to them can usually be detected at compile time and the appropriate information may then be passed on to other streams. However, if these variables are declared inside macros, which might not even be called, it is not known before run time what the value will be. An obvious way round this problem would be to communicate the value of the variable at run time, using the `send` and `recv` mechanism just described. For the moment, though, we note that the pre-processor has not been programmed to detect assignment statements involving variables within macros, and any such alterations to the values of variables in these circumstances is likely to cause total havoc.

BENCHMARK RESULTS

It would be unfortunate if the constraints just described demanded an awkward style of PIC programming before the advantages of parallelism could be realised. In case it might be thought that our examples in this section have been doctored to steer clear of any problems, we can reveal that some of our results use an unmodified test set of some 35 pictures, which have been gathered together from various sources by Kernighan.

The results we shall present were obtained by running both the parallel and the ordinary implementations of PIC, under UNIX, on a Sun 3/160 fileserver. Although the results are reasonably representative of what one might expect to achieve on a true multi-processor parallel machine, we shall see that the synchronization process, and the consequent passing of information, are subject to the overheads inherent in the UNIX system supporting the simulation.

A small test example

Figure 3 shows a graph of the total time taken to draw the simple diagram (figure 2), first of all with conventional 'single stream' PIC and then by allocating the source code to four streams in the manner just described. It is apparent in figure 3 that a good overlap of activity has been achieved between the four streams and also that the various overheads of synchronization, idle time and so on are far from negligible.

In the four-stream case we see the overheads which can arise in parallel processing schemes of any sort. The time periods labelled 'i' are the idle times, which occur when a stream is waiting for information from one or more of the other streams. The

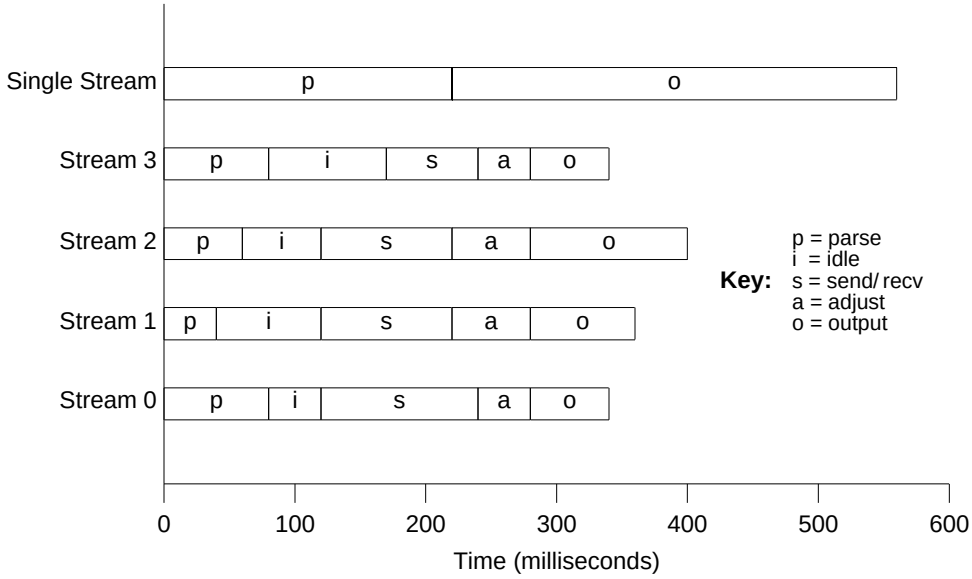


Figure 3. Overlap of processing for the simple diagram in figure 2

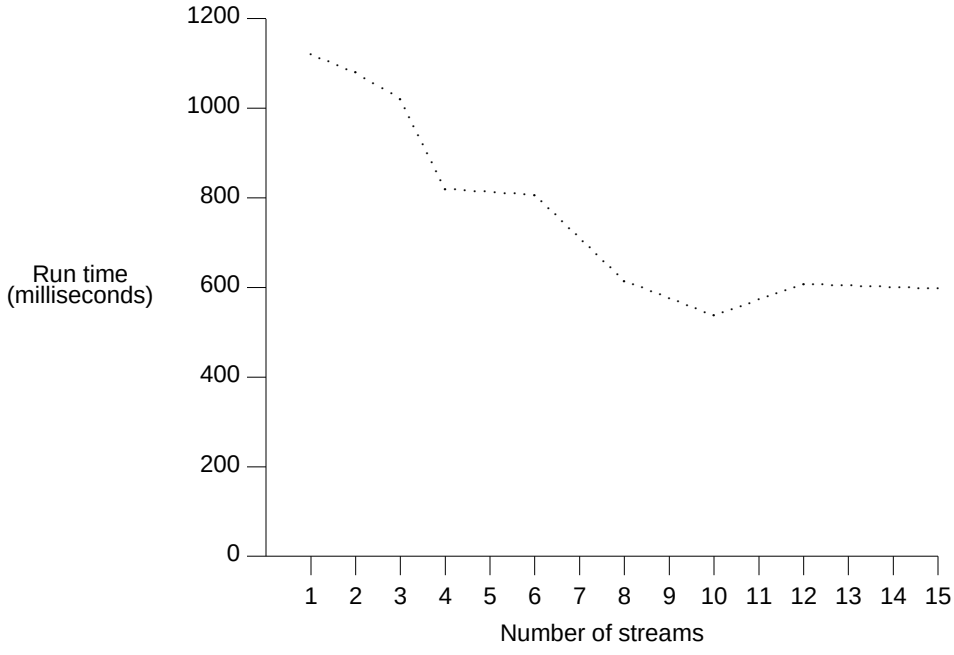


Figure 4. Variation of run time for the simple diagram as number of streams increases

implementation of `send` and `recv`, and their use at synchronization points, does not contribute strongly to this idle time because, as already explained, it has been arranged that the processes executing these primitives will very seldom become blocked. The idle time, then, arises largely from the various processes waiting to participate in the final rendezvous for establishing the global bounding box. The portion of time marked '`s`' is the overhead involved in the `send`/`recv` mechanism, which is in turn dominated by the system call overheads of UNIX. (These system calls are for opening, reading, writing and scanning the files where the messages are passed.) The adjust time, '`a`', is the time taken for each stream to submit the coordinates of its own bounding box to stream 0 and for that stream to calculate the global bounding box and to broadcast it. The output time, '`o`' is the time taken to traverse the linked list on the given stream, after the global bounding box is known, and to output the appropriate *troff* drawing primitives for that portion of the diagram.

The execution time of the four-stream case is essentially that of its longest running stream (Stream 2) and the ratio of this time to that achieved by unmodified PIC gives a rather disappointing speed-up factor of 1.4. Even if the various overheads shown in figure 3 were to be ignored it would not speed up execution by more than a factor of 3 so the obvious inference is that we might be close to the performance limit (i.e. there is not enough material in the picture to make extra processors worthwhile). Fortunately it proved easy to verify this hunch by simply observing the variation of the speed-up factor as the same PIC code was allocated to an increasing number of streams. To do this the modified compiler was parameterised with an integer, n, which stipulated the number of streams to be used for code allocation. For the simple diagram it was then possible to plot the total execution time as a function of the number of streams used (see figure 4). It will be seen that the resulting curve is not smooth; this arises because the amount of PIC code on each stream decreases as the number of streams increases and the processing time per object becomes comparable with the total processing time per stream. What we then observe is a 'quantum effect', where large variations in a stream's processing time can occur depending on whether the indivisible processing time for a given object is allocated to this particular stream rather than to either of its neighbours.

However, even allowing for the unevenness of the curve, it is clear that performance levels off after about 8 streams with the run time having been reduced by a factor of 2 compared to the single stream case. This seems to be the limit of parallelism for this particular diagram.

A larger example

The results obtained with the small diagram in the previous section prompt us to try a larger diagram, to verify that more encouraging performance gains would be obtained when more material is available for processing in parallel. Our tests used the diagram shown in figure 5, with the PIC code being allocated to 10 streams. Once again the bold labels '$\mathbf{S_0}$', ..., '$\mathbf{S_9}$' show where the streams begin, as allocated by the weighted objects algorithm, and the bullets indicate the synchronization points between adjacent streams. We shall refer to this picture from now on as the 'complex diagram'.

Although there is not the space here to reproduce the source code for this diagram,

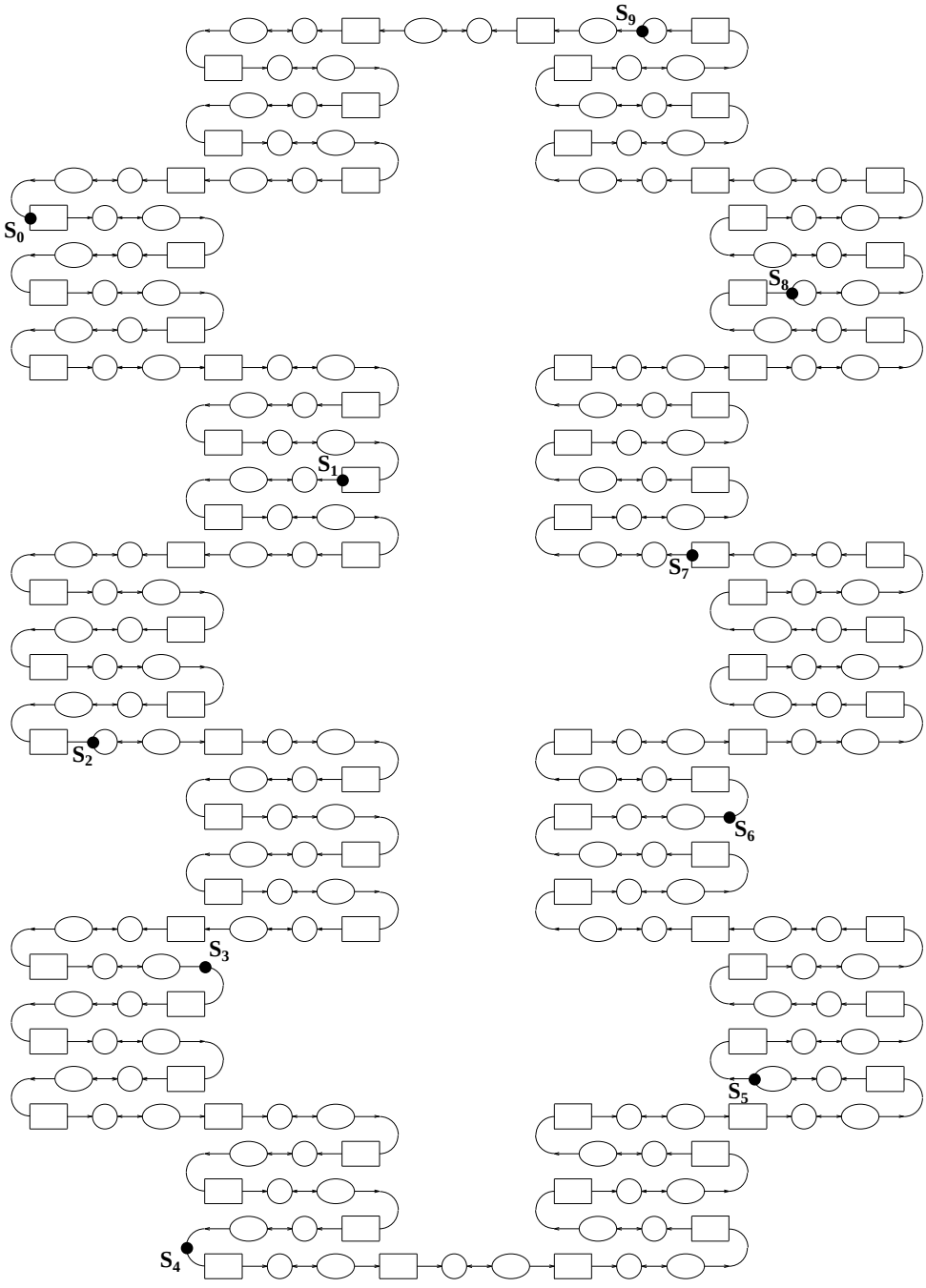


Figure 5. Complex diagram, with bullets showing synchronization points

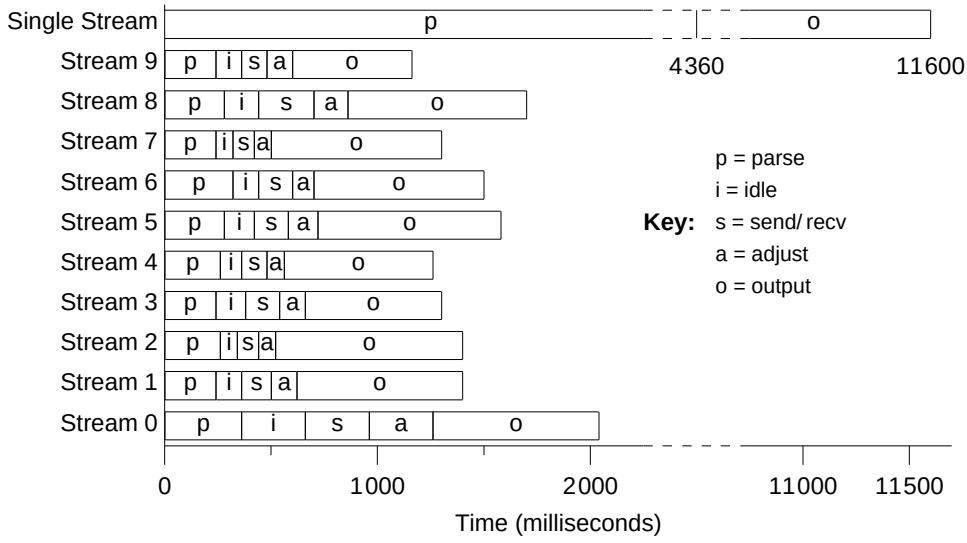
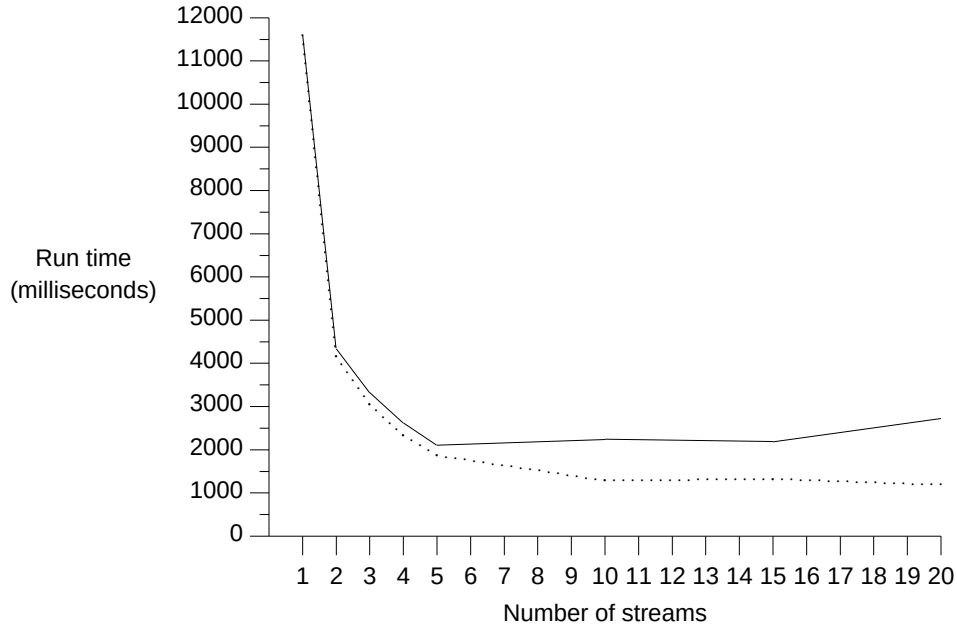


Figure 6. Overlap of processing for the complex diagram



Key: ——— variation of processing time for complex diagram in figure 5.
····· variation of processing time for complex diagram, ignoring communication overheads.

Figure 7. Variation of run time with number of streams for the complex diagram

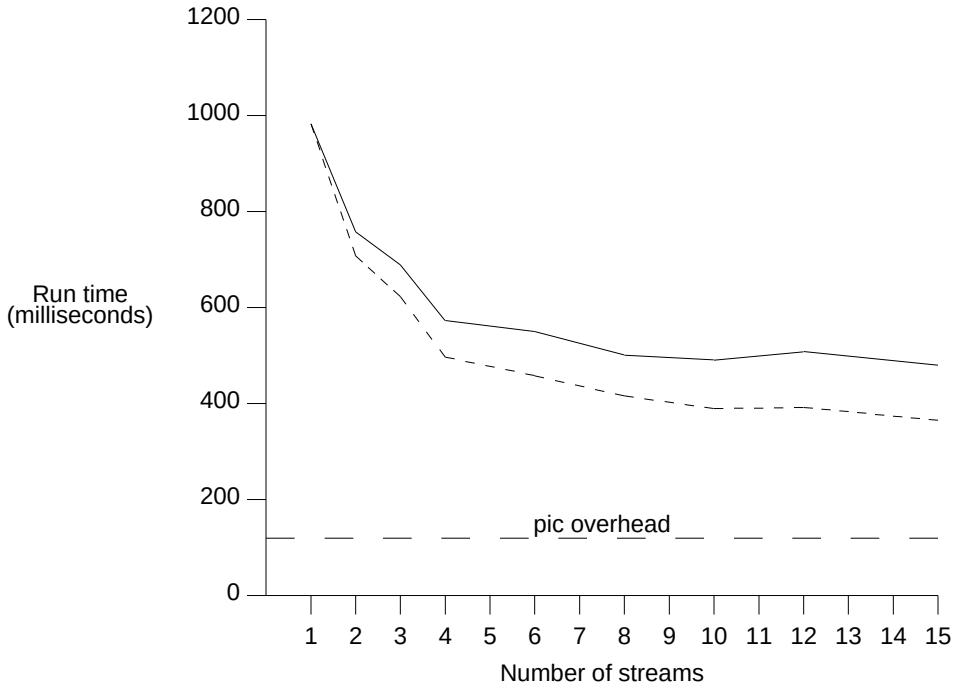
figure 6 shows the overlap of activity during processing. It can be seen that the total processing time on each stream is about the same, showing that the code allocation algorithm has done a reasonable job in apportioning work between the streams.

The speed-up factor is now about 5 but we notice, once again, that system calls contribute in large measure to the communications overheads. Fortunately the time spent in system calls can be determined from the `gprof` output and can be subtracted from the times for each corresponding stream. When this has been done we find no change in the *relative* magnitudes of the various run times. Stream 0 still takes the longest time to complete but its overall run-time is reduced to 1500 ms, giving a speed-up factor of 7.7, which is near to the ten-fold effect that might be expected from the number of streams in use. Obviously, on a truly parallel machine the speed-up factor would lie somewhere between these two numbers, depending largely on the extent to which communications overheads can be minimised and the workloads on the streams kept in balance. But whether one takes 5 or 7.7 as being the performance index does not matter too much; the more interesting question is how near we are to the performance limit, as determined by the nature of the atomic objects in PIC and the number of these objects that are available for parallel processing.

Figure 7 shows how the overall processing time for this complex picture varies with number of streams that are brought into use. Two plots are shown — one with system call overheads included, and the other where they have been subtracted out. Although the overall shape of these two curves is fairly similar it is interesting to note that, with system calls included, the performance actually deteriorates from about 6 streams onwards whereas, when this overhead is ignored, the curve merely flattens out. What we are witnessing is the classic behaviour of parallel systems whenever communications overheads reach the same order of magnitude as the computational load on the individual processors. The flattening out of the dotted curve (i.e. after due allowance has been made for system calls) can be ascribed to at least two factors: firstly, it is not possible to balance the workloads so perfectly that idle times are totally eliminated and furthermore, for every new stream introduced, more information needs to be communicated to and from stream 0 during adjust time.

Results from the PIC test set

Having established the magnitude of the performance improvements obtainable, with the simple diagram and the complex one, it seemed a useful next step to investigate the average behaviour over a set of diagrams. The PIC test set has already been mentioned; the diagrams in it range from straightforward demonstrations of the basic capabilities of `line`, `circle` and so on, through to more complex examples, which have been included on the grounds of aesthetic appeal or because they exercise some more advanced feature of PIC. The test set as a whole was benchmarked in much the same way as the two individual examples already considered, with the source code being allocated to a steadily increasing number of streams and the total run-time being measured in each case. We cannot disguise our satisfaction (though tinged perhaps with just a little surprise) in reporting that none of the diagrams caused difficulties for the stream allocation algorithm. Furthermore, the collected and merged output from the streams always succeeded in drawing the correct pictures, regardless of the number of streams in use. It would appear that PIC programmers (or, at any rate, those whose



Key: ——— mean run time per diagram for the PIC test set.
– – – run-time without system call overheads.

Figure 8. Variation of average run time per diagram for the PIC test set as the number of streams increases

diagrams appear in the test set) write their code in a straightforward fashion, with the lexical ordering of the program being indicative of the actual execution path, and with little or no recourse to obscure side-effects on global variables.

From the performance viewpoint, figure 8 shows the variation of the mean processing time per picture, for the PIC test set, as the number of streams is increased. The average run-time is in the region of a few hundred milliseconds and a speed increase of about 2 is obtained by the time 5 streams are in use, with little improvement thereafter. Since these figures are quite close to those obtained when processing the simple diagram, we conclude that the average picture complexity is closer to the simple diagram than to the complex one. The reassuring point here is that the model system seems to be well-conditioned in giving reproducible and plausible results for a wide variety of diagrams provided they have roughly the same number of atomic PIC objects.

As a final point of interest we should note that the ultimate performance limit, in our scheme, is determined not so much by running out of material for processing in parallel as by the initialisation overhead for setting up the PIC parser, on all the streams, before any processing begins. This overhead is the same on all streams, of course, and it amounts to some 120 ms on the computer used for this simulation. We show this initialisation time as a horizontal line on figure 8, though we could equally well have included it on all our earlier performance graphs. The reason for waiting until this late

stage before drawing attention to it is that this particular overhead is fixed in magnitude and is consequently of greatest importance for precisely the 'typical' simple diagrams we have just been discussing, where the run-time for the picture is small. We note that this factor alone limits the maximum possible speed-up for a typical diagram to a factor of about 8.

CONCLUSIONS

It has been fascinating to see how our streams model for PIC shows all the familiar characteristics of larger parallel systems, but the figures we have obtained for gains in performance yield no major surprises.

It was always apparent that massive parallelism does not exist within line diagrams — or at any rate not at the level of abstraction at which PIC is processing them. (Indeed, for the case where a document contains many diagrams, it could be argued that processing time could be reduced far more effectively by ignoring parallel processing altogether and by developing some tool similar to the UNIX *make* which processes only those diagrams which have altered since some previous run.)

Judging from the results obtained with the simple diagram, and from the mean processing time per diagram in the PIC test set, it seems that our exploitation of intra-diagram parallelism leads to a halving of processing time for the sort of diagrams that users actually put into their documents. The flattening out of the performance curves beyond a certain number of streams tends to show that we are running out of parallelism (at the granularity level of the atomic objects in PIC) rather than being hampered by any lack of parallel processing resources. We could justifiably conclude that extracting this intra-diagram parallelism is not worth the effort, and certainly not in cases where diagrams constitute only a small part of the document.

This judgement would be balanced rather differently if a document contained a large number of diagrams such as the one in figure 5. If such diagrams were to be generated descriptively, perhaps with the use of macros, then our methods could process them very effectively and a ten-fold speed-up in processing time would certainly seem to be worth pursuing. However, it might seem far more appealing to produce such complex pictures with interactive software, even though the final output code could still be PIC, to allow it to be easily transported from document to document. It turns out that analysis of the PIC code we have been using, as opposed to that which might be generated by an interactive system, throws more light on the future prospects for parallelism in document processing than can be gained merely from our figures of performance gains. These matters we shall now examine in a little more detail.

Absolute and relative coordinates

There are many interactive software systems for workstations which enable diagrams to be drawn by choosing from a palette of object shapes, followed by placement and sizing of these shapes. The connections between these objects can be made with lines, arcs and so on. The output from such programs may take on a variety of forms, but some of them do indeed generate PIC as their output. One of the earliest of these, and typical of the *genre*, is Sally Browning's CIP [14].

The nub of the problem with any interactive scheme is that it is extraordinarily hard to

generate the same kind of PIC code that would be produced by a human actually *describing* the picture. The ability, in interactive usage, for placing objects with arbitrary spatial relationships to one another, leads to a heavy reliance on absolute rather than relative coordinates. A few moments' reflection should convince us that parallel processing with our streams model would be straightforward if all the PIC code used correct absolute coordinates. It would still be advisable for the streams to rendezvous, in order to establish that the bounding box would fit on the physical page, but there would be no need for any inter-stream synchronization, because each object could be independently and correctly placed with code such as:

```
line from (0.0, 0.0) to (1.0, 0.0)
box width 1i with .c at (1.5, 0.0)
```

The PIC compiler can map objects having absolute coordinates into suitable output codes very easily indeed, without any need to take the coordinates of other objects into account, and this means that the streams can run truly independently and at full speed (a luxury which is normally only achieved in the output phase for the examples we have presented). However, the penalty we pay for tying any system to absolute coordinates at an early stage is that it becomes inflexible when one wishes to modify the diagram. Moreover, the logical connection between the various portions of the diagram ('this line touches the next box at its west point') is completely lost, which makes any scaling or adjustment of composite objects extremely difficult. In the end one reaches the weary conclusion that it is too much effort to adjust, by hand, any PIC source which is littered with absolute coordinates. Instead one has to go back to the interactive system that generated the PIC before any modifications can be made.

By contrast, all of our examples have used relative coordinates almost exclusively, and this is the way that programmers would naturally generate a PIC diagram. Most objects are logically and physically linked to some other and our lines and boxes can now be seen to join up explicitly, provided we describe things in relative language such as:

```
line right 1i
box width 1i with .w at last line.end
```

The small penalty we pay for this, in our streams model, is the communication traffic incurred when the distinct portions of the diagram link up at the synchronization points. What we gain is the ability to edit the PIC and to alter the relationship of one object to another without compromising the ability to process in parallel. Interestingly, the worst of all worlds occurs when the PIC code has some arbitrary mixture of absolute and relative positioning commands, for then the communications traffic for adjusting local coordinate schemes at synchronization points, and for agreeing absolute bounding boxes, reaches major proportions.

The division between absolute and relative systems is not confined to the domain of Cartesian coordinates, nor to the drawing of diagrams. It affects all areas of document processing and all areas of computing. In the end all systems have to become 'absolute', if only because the hardware works that way at its lowest level, but the lesson learned from many years of computer development is that it pays to delay any absolute bindings for as long as possible. For example, it is perfectly feasible to run multi-process operating systems where every program is bound to absolute addresses in memory. The

illusion of parallelism is certainly maintained, but the system is incredibly unwieldy if it is desired to add more memory or if a process has to be moved for any reason. However, if one is prepared to pay the small penalty incurred with memory management hardware, where addresses are kept in relative form until the last possible moment, then one still retains the ability to do parallel processing but now with a greatly enhanced facility for moving processes around and for creating new ones. Or again, in text processing, any WYSIWYG system which takes the physical objects of line and page as its unit of processing may well be adaptable to some sort of parallel processing, but it will usually be much less flexible than systems which place words, sentences and paragraphs in a relative way, leaving details of line breaks and pagination until the last possible moment.

Language considerations

Although we were able to say that our choice of PIC diagrams was not constrained by the fact that they were to be processed in parallel, we certainly cannot make the same assertion about the choice of PIC itself for our model system. Having made some unkind remarks about functional languages in the earlier sections we must now come clean and admit that the reason for our success with PIC lies in the fact that it is designed on 'functional' lines. The vast majority of PIC source code uses combinations of atomic objects such as `line`, `circle`, and `box` in a declarative style. These objects interface with one another at a logical and physical level by the use of attributes such as `above`, `.c` and so on. For convenience alone there is a small set of global quantities and users are allowed to define their own variables[2] but the style of PIC is not dominated by these factors.

It is now easy to understand why we chose diagram processing as our starting point; the very nature of PIC made it easy to use and adapt. If we had chosen to delve into text processing at the same level of detail we would have to face the fact that the widely available tools, such as *troff* and TEX have a much larger state-vector of global variables than PIC does. Even worse, their design and operation at the basic level depends on making detailed and obscure changes to that vector in order to effect changes to point size, font, line length, character slant and so on. To be fair, both of these systems are capable of being used in a more abstract manner by means of macro packages but it is still the case that a host of side-effects can be caused, outside of the macros, which have a dramatic effect on the document and which would be hard to administer in any parallel scheme.

Further work

Our researches continue on the more general theme of processing blocks of text in parallel. We remain convinced that a small number of global entities (point size, font, leading, line length and so on) needs to be maintained but the style of the underlying language also needs to be more 'functional' to allow for processing in parallel. Questions arise also as to the scale of the optimum granularity. Should this be at the sentence, paragraph or subsection level? We hope to report our results in a future paper.

[2] Though it is an intriguing observation that much of the usage of this facility is to attach names to user-defined *constants* rather than for recomputation and re-assignment of new variable values.

ACKNOWLEDGEMENTS

Thanks are due to Brian Kernighan for supplying us with the PIC test set. Our referees suggested numerous improvements both to the overall style of the paper and also to the detailed exposition of the streams mechanism.

REFERENCES

1. J. F. Ossanna, 'NROFF/TROFF User's Manual', Bell Laboratories: Computing Science Technical Report No. 54 (April 1977).
2. Brian W. Kernighan, 'PIC — A Graphics language for Typesetting. Revised User Manual', Bell Laboratories: Computing Science Technical Report No. 116, (December 1984).
3. Philip C. Treleaven, David R. Brownbridge, and Richard P. Hopkins, 'Data-Driven and Demand-Driven Computer Architecture', *ACM Computing Surveys*, **14** (1), (1982).
4. Scientific American, *Trends in Computing*, **1**, (October 1988).
5. E. W. Dijkstra, 'Co-operating Sequential Processes', in *Programming Languages*, ed. F. Genuys, Academic Press, New York, pp. 43–112, (1968).
6. J. R. Gurd, C. C. Kirkham, and I. Watson, 'The Manchester Prototype Dataflow Computer', *Communications of the ACM*, **28** (1), pp. 34–52 (1985).
7. H. Brown, 'Parallel Processing and Document Layout', *Electronic Publishing — Origination, Dissemination and Design*, **1** (2), pp. 97–104 (1988).
8. W. Wulf and Mary Shaw, 'Global Variable Considered Harmful', *SIGPLAN Notices* pp. 28–34 (February 1973).
9. J. Darlington and M. J. Reeve, 'ALICE: A Multiprocessor Reduction Machine for Applicative Languages', *Proc. ACM/MIT Conference on Functional Languages and Computer Architecture* (1981).
10. M. Tokoro, J. R. Jagganathan, and H. Sunahara, 'On the Working Set concept for Dataflow machines', *Proc 10th Annual Symposium on Computer Architecture* pp. 90–97 (1983).
11. D. E. Knuth, *TEX and METAFONT: New Directions in Typesetting*, Digital Press and the American Mathematical Society, Bedford Mass. and Providence R.I., 1979.
12. D. F. Brailsford and R. J. Duckworth, 'The MUSE Machine — an Architecture for Structured Data Flow Computation', *New Generation Computing*, **3** (2), pp. 181–195 (1985).
13. Brian W. Kernighan and Rob Pike, *The UNIX Programming Environment*, Prentice-Hall, 1984.
14. Sally A. Browning, 'CIP User's Manual', AT&T Bell Laboratories Internal Memorandum (March 1982).