Dewi Jones

Self-archive of publications

Title:	A hardware scheduler for parallel processing in control applications
Authors:	T P Crummey, D I Jones, P J Fleming & W P Marnane
Published in:	Proc. IEE Conference CONTROL '94, Warwick UK, IET Conference Pub 389, pp 1098 - 1103
	DOI: 10.1049/cp:19940289
Version:	This version is the authors' post-print (final draft post-refereeing)
Copyright:	

A hardware scheduler for parallel processing in control applications.

TP Crummey[1], DI Jones[1], PJ Fleming[2] and WP Marnane[3]

Dr D I Jones
GWEFR Cyf
Pant Hywel
Penisarwaun
Gwynedd LL55 3PG
United Kingdom
Tel: +44 (0)1286 400250
E-mail: dewi@gwefr.co.uk

1. University of Wales, Bangor, UK.

2. University of Sheffield, UK.

3. University College Cork, Eire.

Abstract:

One approach to catering for the higher computational demands of modern digital control systems is to use parallel processing (Fleming [1]). Despite the many examples available which use this technique (lEE Proceedings [2]), it cannot be claimed that it is the natural choice of the control engineer for implementation. The premise for the work described in this paper is that parallel processing must be as transparent and convenient to the designer as a single processor solution, if it is to become an acceptable option. Problems such as dead-lock, livelock, communication delays and network topologies contribute to the difficulty of parallel programming and should, as far as possible, be hidden from the user. The well-known processor farm paradigm is a step towards this goal because of its dynamic scheduling properties. However, control system applications generally involve processing relatively small amounts of data in short periods of time with a definite deadline which is more difficult to achieve than high average throughput on a large job. This paper describes a specialised hardware scheduler for a processor farm which minimises the overhead of scheduling multiple tasks to multiple processors. It also describes the harness software which allows the algorithm to be separately partitioned into tasks and incorporated as linked modules.

Introduction.

One of the most difficult aspects of parallel processing, when compared to sequential processing, is how to distribute the computational load across the available processors. In solving this problem, the choice of algorithm, the choice of processing topology, the relative computation and communication capabilities of the processor array and the partitioning of the algorithm into tasks and the scheduling of these tasks all have to be considered. This adds a second tier of design complexity and it is necessary to reduce these design overheads if parallel processing is to be accepted as a viable alternative to ever faster single processor solutions.

Sarkar details these problems in (Sarkar [3]) under the headings of:

i) Identifying parallelism in the algorithm; The algorithm to be computed must offer potential for parallelisation. This is not always the case, and indeed, many algorithms may need to be recast to allow them to be computed in parallel (Gaston and Irwin [4]).

ii) Partitioning the program into parallel tasks; Once it is possible to identify parallelism, it is important to partition the program so that the granularity of the tasks is suitable for the target processing array. This requires knowledge of the relative communication and computation capabilities of the processing array for the optimum task size to be chosen.

iii) Scheduling the tasks onto the available processors; Once partitioning has occurred, it is necessary to allocate the tasks to the target processors so that their dependencies are satisfied and the processors are utilised efficiently.

A processing farm assists with the third part of the parallel programming task. A master processor allocates tasks comprising the algorithm to slave workers, scheduling tasks according to a graph of dependencies. This dynamic scheduling dispenses with the need to determine a task schedule a priori and, furthermore, there is no need to explicitly code the algorithm for parallel execution as the tasks can be executed serially or in parallel as allowed by the available hardware. However, the engineer still needs to determine the dependency graph and decide how the algorithm may be partitioned into tasks.

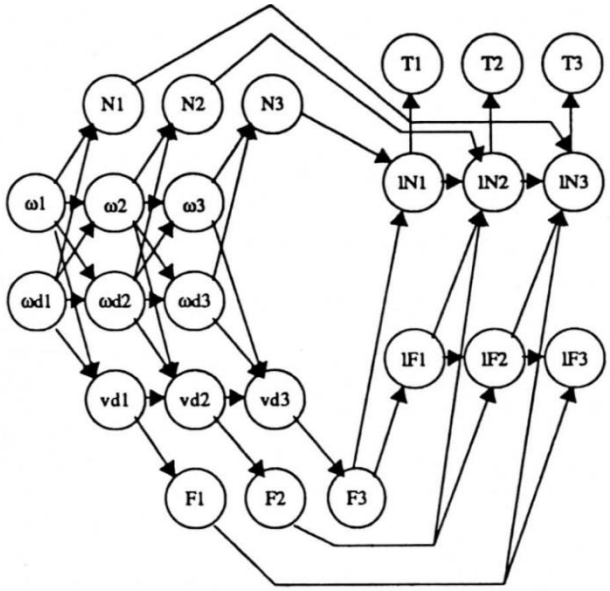
In the particular class of problems of interest here (Control algorithms), the scale of the overall computation required is quite small, but has to be completed in a short, fixed time interval. This combination of small tasks and short execution intervals causes the scheduling and communication overheads to become a significant part of the overall execution time. Indeed these overheads can dwarf the time required to compute the algorithm itself.

Previous work on this problem (Fonseca *et a1* [5]) showed that a transputer processing farm used as a general purpose execution platform had significant overheads associated with software scheduling. In addition, the communication overheads were analysed and the conclusion was that if the scheduling overheads could be reduced, this would have a beneficial impact on the over- all algorithm execution times. To this end, a partial implementation of a hardware scheduler was undertaken which showed that the overheads could be reduced substantially. The current work has taken a different approach in that it generalises the work in [5] and also simplifies the hardware required. Care has been taken to ensure that the proposed solutions are scalable and applicable to a range of problems.

Problem representation and example application

In order to implement an algorithm in parallel it is useful to represent the program as an acyclic directed task graph. The example application considered in this paper is the computation of the inverse dynamics of a 3 link robot manipulator using the Newton-Euler formulation (Craig [6]), and the task graph for this algorithm is shown in Figure 1. It is necessary to complete this computation within the sampling interval of the control loop, typically a few milliseconds. As Figure 1

shows, the velocities and accelerations of the robot links are calculated first, followed by the link to link moments and forces and finally the torques which must be applied at each joint to sustain the demanded acceleration.



ω - angular velocity	N - moment
ω_d - angular acceleration	lF - link-to-link force
v_d - linear velocity	lN - link-to-link moment
F - force	T - torque

Figure 1 Task graph for 3 link NE

This task graph is not unique. In this case, the partitioning simply consists of designating each expression in the algorithm to be a task, which gives a rough balancing of task size but cannot be considered optimal in any sense. The work described here does not directly address partitioning - the user must decide on the 'atoms' of computation which compose the task graph. Our concern is dynamically scheduling the chosen task set efficiently and transparently.

This particular example application was chosen for several reasons:

i) A good body of literature already existed on the application of parallel processing to this problem (Kasahara and Narita [7]) and it has become something of a bench- mark in parallel processing for robotics. This work uses the fully general inverse dynamics equations for a six link manipulator (as opposed to a version applied to a particular robot which is often simpler) and also allows the dynamic scheduling of the component tasks.

ii) The problem has a good mix of serial and parallel sections.

iii) The computation is quite small, but yet must be calculated in a short time period. This stresses the fine grain capabilities of the processing system.

iv) It enables us to directly compare results from previous work (Jones and Entwistle [8]).

Other, larger, applications have been run using the original processor farm such as configuration space generation (Solano and Jones [9]) and path planning (Doyle and Jones [l0]). The latter application particularly, due to its variable task sizes, may benefit from the dynamic task scheduling.

The processor farm

The Original Farm

The original processor farm [5] has the architecture outlined in Figure 2.

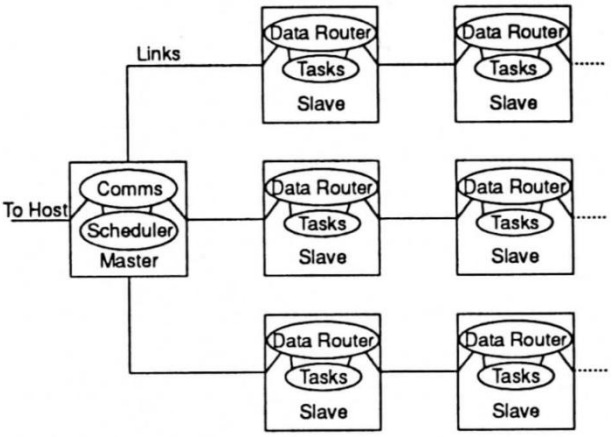


Figure 2

The master processor is connected to the host computer and to three other chains of slave processors using links. The master processor runs scheduling and communication processes and it allocates the tasks to be executed to the slaves. The slaves run the task execution software and data routing processes to allow data to pass through processors up and down the chains (see Figure 3). As tasks are executed, results are passed back to the master via any intervening processors. Once the master

receives data from completed tasks, it searches the schedule for more executable tasks and constructs data packets which are sent to free slaves to initiate further tasks. Note that a copy of each task resides on every processor.

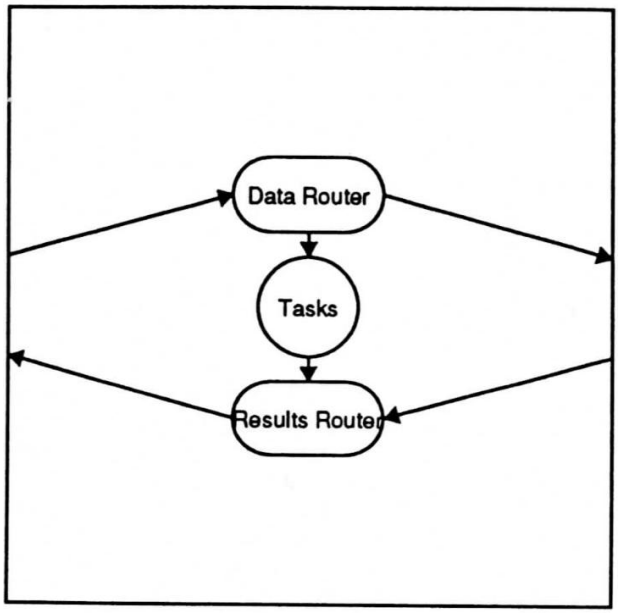


Figure 3 Original worker architecture

The Scheduler

The action of the scheduler resembles that of a cyclic task despatcher in an operating system. The task graph representation is easily converted to a square dependency matrix where each row and corresponding column is associated with an individual task. Where a '1' is entered in the matrix, the task corresponding to that row depends on the task specified by the intersecting column. The scheduler maintains a vector of finished tasks and the search for an executable task consists of comparing this vector with each row of the matrix as illustrated in Figure 4. In the diagram, a small set of eight tasks is shown with the initial task being task 1 (as it has no dependencies). On the initial comparison between the finished tasks vector and the matrix, task 1 can be executed. When task 1 has been executed, the left most bit is set in the vector and another comparison performed. This time task 5 can be computed. A full schedule could be: 1, 5, 3 & 7, 4, 6, 8, 2. The potential for parallelism here is that both tasks 3 and 7 can be executed simultaneously.

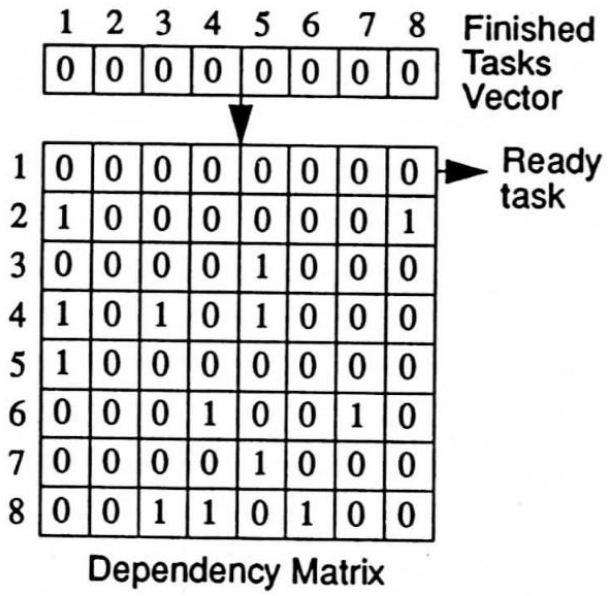


Figure 4 Scheduler Action

At this point. the scheduler would put both tasks 3 & 7 on a stack and both would be allocated as quickly as possible as processors became available. The comparison function for selecting tasks that are ready can be described as: 'If all the bits set in a row of a matrix are set in the vector, then the task is ready'. This is equivalent to a Boolean function of $\bar{A}$ + B, where A is a row in the matrix and B is the finished tasks vector.

The Communications Structure

The original processor farm was highly specific to the example application of robot inverse dynamics and depended in particular on a fixed communications structure and protocol (Entwistle [10]). It was therefore necessary to generalise the inter-processor communications software to accommodate data packets of different sizes and data types.

Furthermore, the original processor farm software passed all the results generated by the slaves back to the central scheduler for re-issue to other processors. This scheme has two disadvantages:

i) It increases the through routing requirement along the chains of processors.

ii) It makes the scheduler significantly more complex due to the need to implement and manage a data memory. This would be easy to

do in software, but would be much more difficult in hardware.

In fact, the scheduling function is independent of the data passing between tasks and it is superfluous to manage it in the scheduler. The communications software was therefore modified as shown in Figure 5. It was hoped to offset the extra communications traffic by using hitherto unused links.

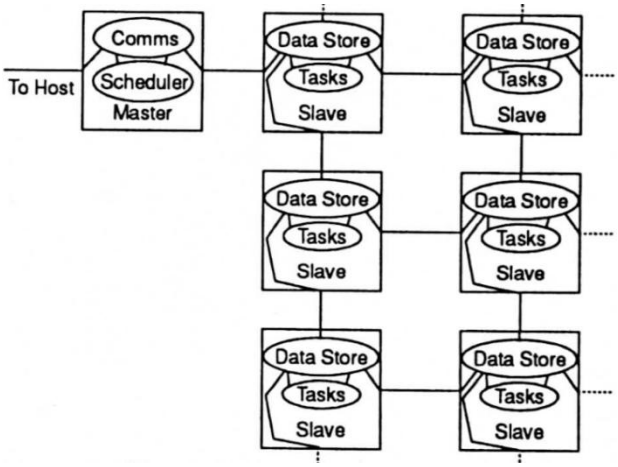


Figure 5 New Farm Architecture

Thus, the master processor merely links onto one of the comers of a fully connected mesh. The master again runs the scheduler and the external communications process. Slaves run the task execution software and also a more complex data handling process. The master still requests slaves to execute particular tasks. However the slaves, instead of passing processed data back to the master for subsequent retransmission to another slave, now broadcast the data to all nodes in the array. This modification means that the scheduler is much simplified since it does not have to process and store algorithm dependent data.

Unfortunately, the increased complexity of the new software architecture increases the communication overhead at each slave processor, (see Figure 6) but this was considered to be an acceptable penalty when traded off against the decreased scheduler complexity.

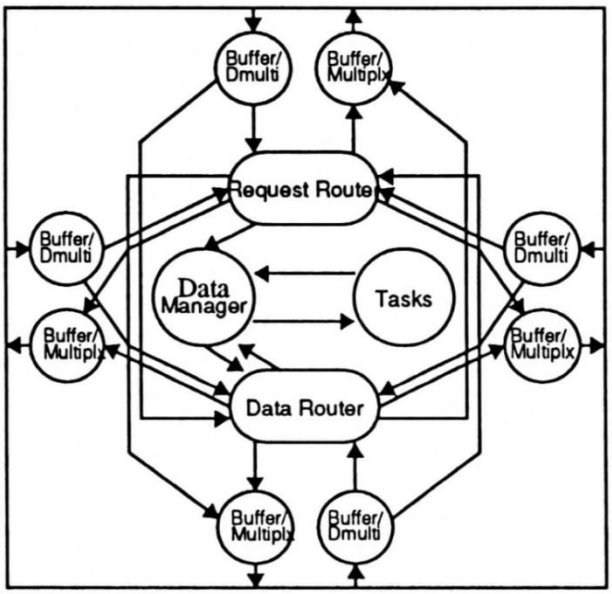


Figure 6 New Worker Architecture

A hardware scheduler

The architecture of the hardware scheduler is shown in Figure 7.

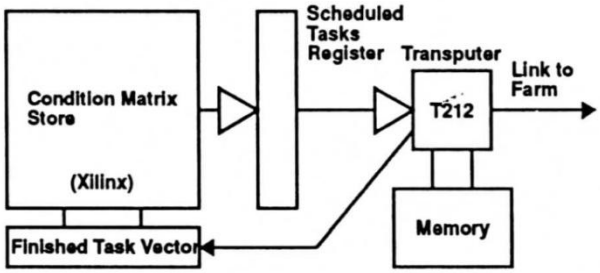


Figure 7 Architecture of Hardware Scheduler

It is implemented on an electrically programmable logic device from Xilinx which allowed several different designs to be tried. It also was flexible enough to allow a scalable hardware solution which was important since the size of the condition matrix depended directly on the number of tasks to be processed and a non scalable solution would restrict the range of applications. The current design allows any number of Xilinx devices to be cascaded

This implementation is superior to the original version because the comparison between the rows of the matrix and the finished tasks vector occurs in parallel.

The Xilinx IC contains a number of configurable logic blocks (CLBs) which can be programmed to perform a storage function and a combinatorial logic function. The logic required to implement the condition matrix looks as shown in Figure 8. Two Match cells can be implemented in each Xilinx CLB leading to a rectangular array of match cells in each square Xilinx chip. Since a condition matrix has to be square, to make best use of the Xilinx ICs the numbers required are 2, 8, 18, for increasing numbers of tasks.

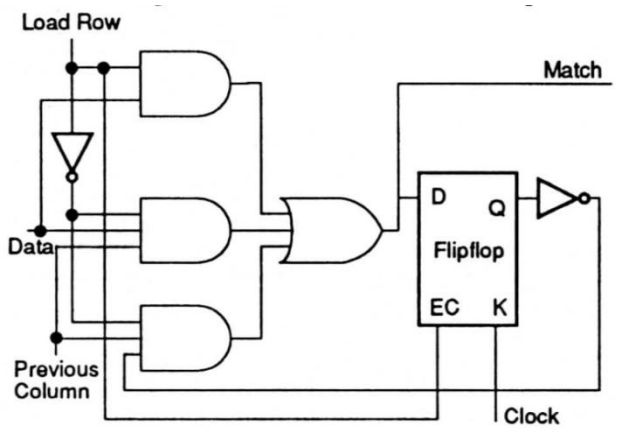


Figure 8 Match Cell Logic

A proof of concept demonstrator of the match circuit has been constructed using a Xilinx 3020 device which contains a square array of 8×8 CLBs. For the final version, we are using the 3195 device which contains a rectangular array of 16x20 CLBs. For this version, the spare logic cells will be used to implement encoders and decoders to minimise the pins required to interface with the transputer.

The fact that the search is taking place in hardware means the comparisons are much faster and leads to a considerable speed up.

Performance comparisons

As has been mentioned previously, the original farm software was faster than the new configuration for software operation. It is obvious from Figures 3 and 6 in general terms why this is, but detailed analysis of the operation of the software is being undertaken to ascertain the exact contribution each process makes to the overall processing time. A comparison of times for the two software architectures is shown in Table 1.

Farm Size	Old Farm	New Farm
1×1	37696µs	n/a
1×2	26880µs	61440µs
1×3	23232µs	45824µs
1×4	22528µs	42048µs
1×5	22080µs	39232µs

Table 1

A simple analysis of times between sending a task request out from the root processor and receiving the results shows that there are some inefficiencies to be removed (see Figure 9). This shows the 5 processor case (4 workers and the master) and similar charts for 5 workers show a considerable reduction in processor utilisation and an overall longer execution time.

The two processor case (one worker and the master) shows that the time contributed to the processing by the master processor is 15ms. Of this, 9.4ms is due solely to the searching of the condition matrix. This is the amount of time we can reduce using the hardware scheduler.

Timings of the Xilinx chip indicate that for the demonstrator version of the hardware scheduler, the time between presentation of the finished tasks vector and the generation of ready tasks output is of the order of 150 nanoseconds per chip. This means that cascading chips together would linearly increase the delay in generating the output. A second design with a different method of wiring CLBs together offers a much faster time of 46 nanoseconds with only wire delays for additional devices leading to negligible extra delay in the generation of the output.

These times are negligible when compared to the software case of 9.5ms. Therefore we see that if the hardware scheduler operates as designed, a speed up of 24% on the 5 processor farm case can be achieved.

Conclusions

This paper has shown that a hardware scheduler can remove the overhead inherent in the dynamic scheduling of a task set. In the design of this system we have taken significant steps towards generalising the application of the concept and we anticipate being able to reduce scheduler overhead. The increased communications overhead due to the new communications software means that in practice, this system will be restricted to a coarser granularity of task than is ideally the case. Certainly it is not suitable for the application example considered here.

Communications overhead could also be reduced by introducing extra hardware. The capability required is to allow every worker node to broadcast to all the other nodes. In principle this could be achieved by means of broadcast links similar to that described by Burnett *et al* [12]. It is not clear at this stage how much extra hardware would be required to implement this or the likely effect on performance.

Alternatively, the T9000 will eliminate the inefficiencies due to communication overheads associated with link multiplexing and buffering which will become much faster with hardware virtual channels and wormhole routing. The proposed scheduling device will still offer advantages over software scheduling since it is 2 to 3 orders of magnitude faster.

It is also intended to use the Inquest profiler and debugger to analyse in detail the performance of the processor farm and this will give the opportunity to manually optimize the task partitions. It may be desirable to divide some large tasks into smaller ones, or to combine a number of small interdependent tasks into one larger one to minimise communications overhead.

References

[1] Fleming, PJ., Garcia Nocetti, D.P., 1992, *Parallel Processing in Digital Control*, Springer Verlag, New York.

[2] Irwin, GW, Fleming, PJ (eds), (1990), IEE Proc D 137(4), Special Issue on Parallel Processing for real-time control.

[3] Sarkar, V., (1989), *Partitioning and Scheduling Parallel Programs for Multiprocessors*, Pitman, London.

[4] Gaston, F.M.V., and Irwin, G.W., *Systolic Kalman filtering: an overview*, (1990), Proc IEE D 137(4), pp 235-244.

[5] da Fonseca, P.N.P., Entwistle, P.M., Jones, D.I., (1990), *A Transputer Based Processor Farm for Real-time Control Applications*, in "Applications of Transputers - 2", pp. 140-147, Pritchard. DJ., Scott, C.J., (Eds) IOS Press.

[6] Craig, J.J., (1986), *Introduction to Robotics: Mechanics and control*, Addison-Wesley.

[7] Kasahara, H, Narita, S, (1985), *Parallel processing of robot arm control computation on a multi-microprocessor system*, Trans IEEE RA-l(2), 104-113.

[8] Jones, D.I., and Entwistle, P.M., (1988), *Parallel Computation of an Algorithm in Robotic Control*, Proc. I.E.E. International Conference Control '88, University of Oxford, April 1988, pp. 438-443.

[10] Doyle, A.B., Jones, D.I., (1993), *A Parallel Method of Robot Path Planning using a Transputer Network*, Proc. The Irish Colloquium on DSP and Control, University College, Dublin, pp. 35-42.

[11] Entwistle, P.M., *Parallel Processing for Real Time Control*, (1990), Ph.D. Thesis, University of Wales, Bangor.

[12] Burnett, G., Morris, P., Patterson, A., Powlesland, I., Roelofs, B., (1993), *A Flexible High Speed Distributed Control System for Aircraft Testing*, in Grebe, R., *et al* Eds. Transputer Applications and Systems 93 IOS Press, pp 198-219.

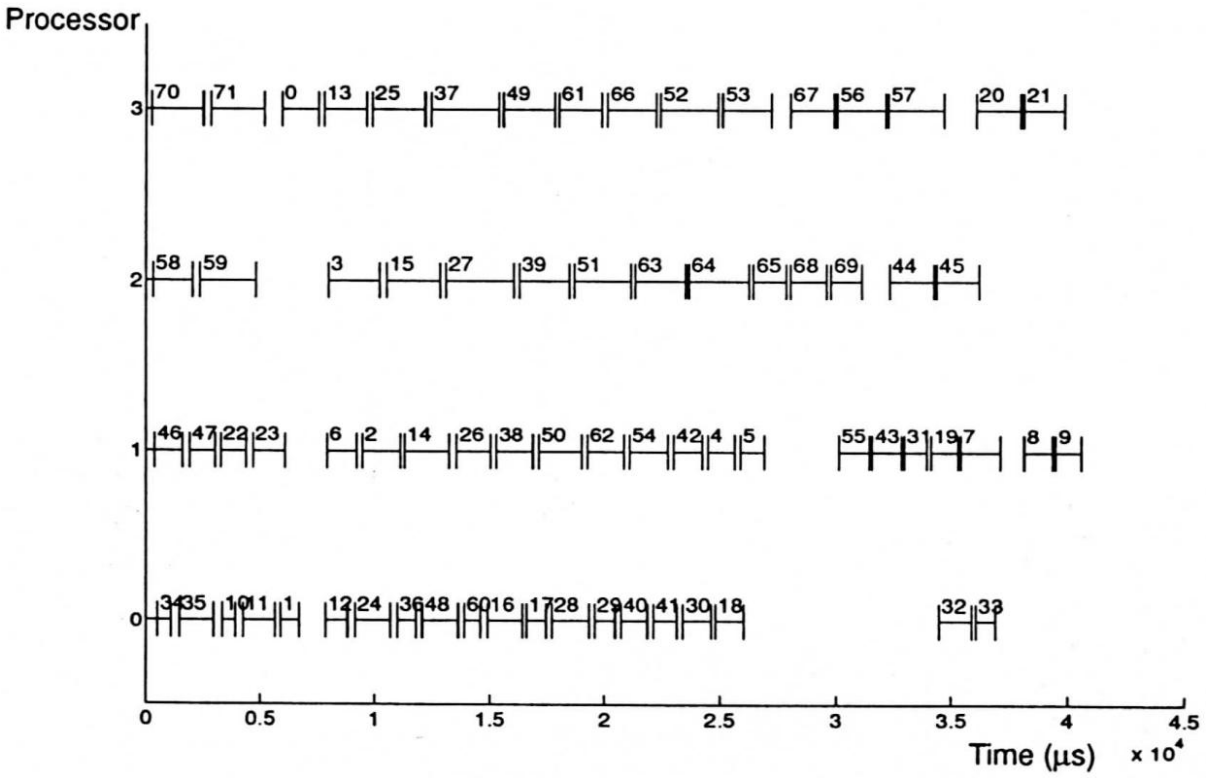


Figure 9 Gantt chart of 4 worker processors.