THE PROCESSOR WORKING SET AND STATIC ALLOCATION POLICY USING PWS

MK Mushtaq, 17XJ1A0526

ABSTRACT

The paper 'The Processor Working Set and Its Use in Scheduling Multiprocessor Systems' aims at two conclusions, after doing relevant research and experimentation. First sites on Processor Working Set or PWS is a good and robust metric to decide how much parallelizable is a program. Second talks about Static Processor Allocation Strategies and how its performance varies when the strategies are based upon PWS. It turns out that processor allocation based on PWS gives better through-put characteristics than resorting to conventional allocation strategies. The paper also proposes an allocation strategy using PWS at low loads and when the load increases it attunes itself and breaks down the job into fragments and also explains how at maximum load the policy allocates one-processor-per-job.

INTRODUCTION

PWS is the number of processors required for a program or a job to achieve better execution time. It is used as a metric to characterize the workload of the multi-processor. One may wonder how this value can be obtained, if looked at, intuitively, the rate of efficacy or effectiveness of the program with respect to the number of processors becomes zero, then the number of processors at this instant gives the PWS value. The rate of change of efficacy value becomes zero for different number of processors for different jobs. For deeper understanding efficacy let us define it mathematically, and later bring up a clearer notion of PWS from there. If p denotes the number of processors then,

Tagg (p) = Tcomp (p) + Tcomm (p); where **Tagg(p)** quantifies the total time signature where **Tcomp(p)** denotes total computation time and **Tcomm(p)** denotes total communication time among processors.

Speed up is defined by the time taken by one processor to execute a program by time taken by p processors to execute the same program. If time taken by p processors is less than time taken by one processor than the speed will be higher (greater than 1), similarly, it will be less then 1 if time taken by p processors is more than time taken by one processor. The speed up S(p) is given as,

S(p) = T(1)/T(p)

Similarly, the notion of efficiency can be brought up to be as speed up per processor, therefore efficiency is

E(p) = S(p)/p

Cost incorporated with increase in processors should also be kept in mind, it should be kept as minimum as possible. If more speed up is achieved with a smaller number of processors then the costs incurred are less, similarly if less speed is achieved with a greater number of processors than the costs incurred are greater. Finally, let us discuss about the term for which all of the above terms have been discussed, efficacy **(N(p)),** put in simple terms, the effectiveness. Efficacy is defined as the speed up achieved per cost incurred. If more speed is achieved with less cost then it is more effective, similarly if less speed up is achieved with more costs then it is less effective

N(p) = S(p)/C(p) = (S(p)^2) / p

So, PWS is the number of processors in use when efficacy gets maximum. Using concepts of calculus, the number of processors that were in use to maximum efficacy can be found by,

d(N(p)) / dp = 0, when the efficacy becomes maximum another thing that we get to observe is **d(S(p)) / dp =0.5*(S(p))/p**, the rate of change of speed up becomes half of the efficiency (not to be confused with efficacy). One can also find PWS from the execution time and efficiency profile, the number of processors that were in use at the ‘knee’ of the plot, determines the PWS value. The terms efficacy and speed up will be frequently used as we go along with our discussion, while the rest terms may not be used but are very essential in understanding the fundamental meaning of efficacy.

PROBLEM STATEMENT

One of the most desired traits expected of any computer machine when assigned a job to it is to get the job done in as minimum time as possible. One possible solution is to incorporate more resources and get the job done. But this solution is not always feasible, doing so may cost more than the benefits gained from the completion of the job itself. One plausible approach which is also feasible, is to parallelize the job or the program. But before the parallelizing the program, one needs to be aware whether the program is parallelizable or not, if yes then to what degree is it parallelizable? So, to characterize the parallelizability of a program a notion called PWS or Processor Working Set has been developed. However, knowing how much the program is parallelizable is not enough, because wrong allocation policy of job to processors may not be effective, worse, it may perform sequentially and thus underutilizing the available resources. Hence, using the PWS found out earlier an allocation policy has been proposed.

METHOD OF APPROACH

There are two objectives of this paper, one is two show how robust and accurate PWS is and other is to propose a static allocation policy, conclusions of which are made after relevant experimentation.

1. Simulations across algorithms

Let us discuss the method of approach for the first goal. Primarily 4 different algorithms – Matrix Multiplication, Quick Sort, Odd-Even Transposition Sort and Enumeration Sort - were studied. Each algorithm was examined with different topologies which include square mesh topology, tree topology and ring topology among others. Topology is nothing but the organization or architecture followed the nodes in a network. Topology effects the communication time due to the differences present among the architectures in the network. The experiments were carried out on a transputer-based multiprocessor machine comprises 16 IMS T800 transputers with 32 bit 10 MIPS processors, Different topologies were tried on the algorithms and the speed up, efficacy and execution time signature plots were plotted and were compared so as to see which one has performed well and what number of processors were required to achieve minimum efficacy.

1.1) Matrix Multiplication:

Matrix multiplication was primarily carried out in two schemes, first being block decomposition scheme and second being row-by-column scheme. In block decomposition the matrices recursively get divided into submatrices and the multiplication and addition operations are carried out by the processors simultaneously, the topology followed here is square mesh topology, where every node is connected to every other node. The row-by-column method was implemented by using both mesh and tree topologies. In tree topology nodes are connected in a tree fashion. The size (24*24) of matrix was same for both the block-decomposition and row-by-column scheme. For both the schemes the Tagg(p) decreases up to certain extent and start to increase thereafter, this is because as the processors increase the communication overhead also increases and it dominates the computation time and also, as observed from the graph's plots, the computation time saturates after certain number of processors.

Finally, the efficacy function behavior of tree and mesh topology are similar (but not same) can be observed with subtle differences however, the PWS values are different for them. Therefore, these small nuances developed by change in topologies are captured by PWS and thus justifying is robustness.

1.2) Quick Sort:

Quick sort is a sorting algorithm, where a pivot element is chosen and the numbers greater than the pivot are brought together and the numbers smaller than the pivot are brought together.

Then the same procedure is recursively called on these segregated sub lists until the size of the sub list becomes one. So, if assignment of these sub lists can be done then parallelism can be achieved. But the partitioning phase is sequential and impedes the execution of the program. This has also been observed in the efficacy graph where the graph only decreases monotonically implying that maximum efficacy is obtained when there is only one processor used and if only one processor is used than it is a highly sequential program and not parallelizable.

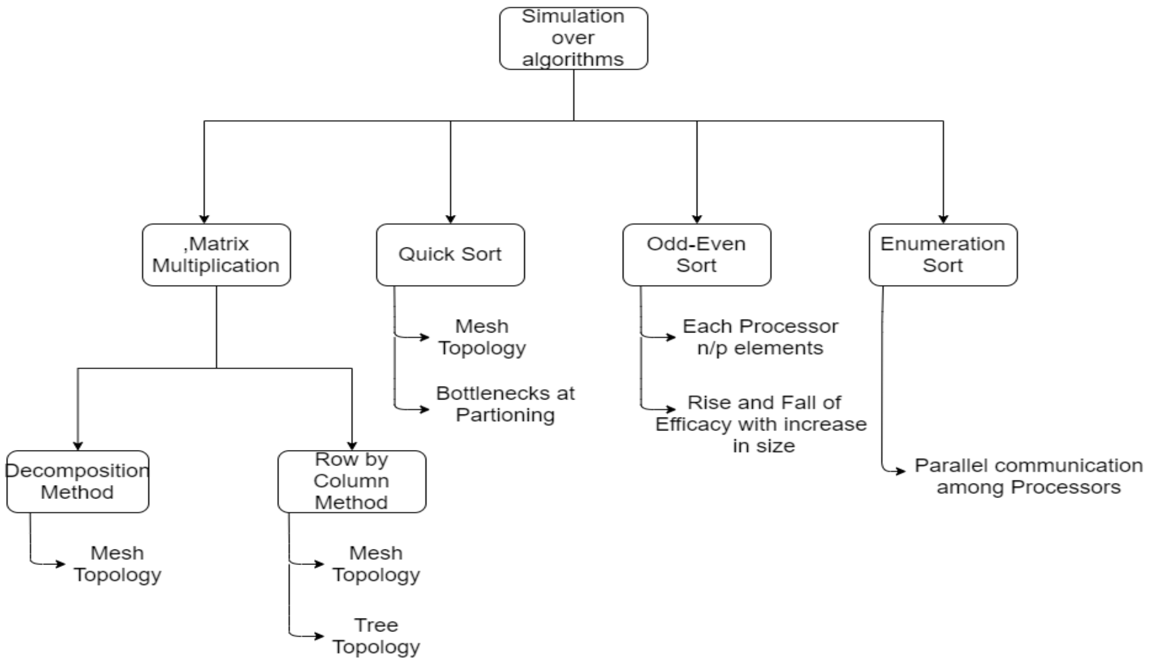
Simulation over algorithms
,Matrix Multiplication
Quick Sort
Odd-Even Sort
Enumeration Sort
Mesh Topology
Bottlenecks at Partioning
Each Processor n/p elements
Rise and Fall of Efficacy with increase in size
Parallel communication among Processors
Decomposition Method
Row by Column Method
Mesh Topology
Mesh Topology
Tree Topology

Figure 1

Figure 1 depicts about the experiment details and highlights of each diagram.

1.3) <u>Odd-Even Transposition Sort:</u>

Odd-even transposition sort works similarly to bubble sort except it does little differently. It comprises of 2 phases odd-phase and even-phase, in odd-phase the bubble sort is applied on the odd indices i.e., elements in the odd positions are compared to its adjacent element and if it the odd positioned element is greater than even position then they are swapped else they are left as is. Similarly, in even phase the same procedure is repeated but this time for even indices. The way they are distributed among the processors is by their size, if the input size is n and there are p processors then each processor gets n/p elements to process on and later the processors communicate their sorted set of numbers. The efficacy plot was plotted by varying the input size ranging from 32 to 8192, the computation time had a higher say than the communication time leading to better efficacy. The computation time decreased much more with increase in the number of processors.

1.4) <u>Enumeration Sort:</u>

Enumeration sort is sorting technique that determines the position of an element by finding out all the elements less than it, it does so by comparing all the elements in the list. Here, in the experiment the numbers are divided into sets and assigned to processors, after each processor does the enumeration sort then processor p1 compares it with processor p2, later p1, p2 compare elements parallelly with process p3 and so on and so forth. The simulation was done on different sizes of the input lists and the efficacy graph was plotted and observed that smaller input sizes suffered communication overheads leading to poor efficacy values and lager input sizes had less computation times leading to better and good efficacy values.

Post the simulations done on the 4 algorithms, three prominent observations were noticed, first PWS does not give minimum execution time but maximum speed-up does and allocating PWS number of processors maximizes the power. Second, even though the efficacy behavior is similar for different topologies for the same scheme, the PWS values are different. The difference is due to the communication overheads caused by the different topologies and PWS captures the communication overheads (by giving different PWS values) due to differences in topologies. And at last, the input size varies the efficacy behavior as observed in odd even transposition sort and enumeration sort. For smaller input size the efficacy values fall quicker with increase in processors due to communication costs and with increase in size the efficacy value increases up to certain extent and falls down thereafter.

2. Static Processor Allocation Strategies

Allocating jobs to number of processors accurately can yield good performance in terms of speed up or efficacy, therefore strategies /schemes have been discussed and their performance is compared to find out the better among them. Before the discussion about these policies begin it is important to get acquainted with the jargon associated with the policies and have some pre-requisites to better understand them. First requisite is first the job is chosen and then the number of processors required for the job is chosen accordingly. Once the number of processors is fixed it remains constant throughout until the completion of the job and hence its name static allocation. The jobs arrive with time values taken from Poisson distribution with a mean time interval of Lambda, also there is a window size for the waiting jobs, remember window size of 1 implies it follows First Cum First Serve policy. Now let us dive into the different policies that have been examined in the experiment carried out.

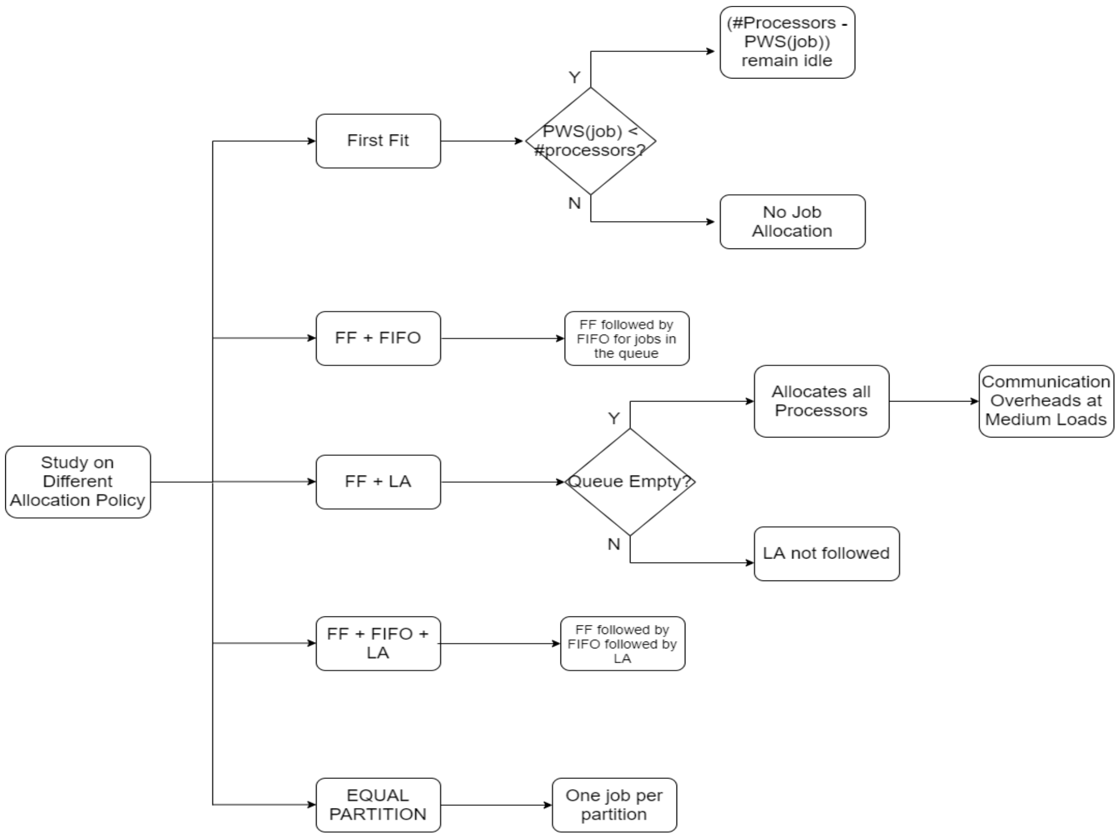
Study on Different Allocation Policy
First Fit
PWS(job) < #processors?
Y
(#Processors - PWS(job)) remain idle
N
No Job Allocation
FF + FIFO
FF followed by FIFO for jobs in the queue
FF + LA
Queue Empty?
Y
Allocates all Processors
Communication Overheads at Medium Loads
N
LA not followed
FF + FIFO + LA
FF followed by FIFO followed by LA
EQUAL PARTITION
One job per partition

Figure 2

Figure 2 illustrates working of 5 different policies that have been examined in the experiment. The policies are discussed in detail below:

2.1) First Fit:

In first fit allocation policy, among the jobs waiting in the window that job is chosen for which the PWS is less than the available number of processors. If there are more than one job that satisfy this criterion than the first one encountered is chosen. But this is not a wise technique for allocating the processors because the rest of the processors (available processors minus PWS of the job) will remain idle and the processors would be under-utilized.

2.2) First Fit with First in First Out:

The second policy in our discussion includes FF + FIFO, here once the allocation is done using FF the jobs waiting in the queue will be processed in accordance with FIFO policy.

2.3) First Fit and Largest Available:

Similar to the above one, this policy first assigns the job by looking at its PWS and the available number of processors, it then looks if there are any jobs left in the queue, if there are none it allocates all the processors to the current job.

2.4) First Fit, FIFO and Largest Available:

This policy is culmination of all the three schemes – FF, FIFO and LA. After FF and FIFO are done, it looks for jobs in the queue, if they are no jobs left then all the processors are assigned to the job.

2.5) Equal Partition:

In equal partition, the processors are divided into batches of equal number of processors and the experiment was carried. No, matter what the PWS value is each partition gets a job.

+ The FF and FF + LA policy doesn't seem to work well for the reason that the jobs in the queue would be simply waiting. The LA policy alone also doesn't benefit because the efficacy value falls down due to the increase in communication among the processors. Some schemes like Best Fit have been explored where in the PWS of the job exactly matches with the available number of processors but it has been observed that it doesn't do better than First Fit.

Results:

FF has been observed under various window sizes, only when window size is 1 the response times increases significantly with increase in arrival rate. Window sizes from 2 onwards do not show significant difference and the has small positive slope. Next, all the policy's performances were observed with increase in arrival rate, at low loads all the policies performed more or less the same way. At average loads FF + FIFO performed the best because it is a work conserving policy unlike FF where the rest processors are left idle, FF + FIFO + LA performs next to FF + FIFO this is because there are communication costs incurred if we make use of all the processors. Finally, at high loads FF + FIFO and FF + FIFO +LA will perform similarly because the number of times LA would be invoked will reduce with increasing arrival rate of jobs.

Here, in equal partitioning, with increase in loads the number of processors that will be assigned to the job will decrease meaning the policies will adaptively fragment the processors for the job at heavy load. Finally, we can say a small window size is good enough for an allocating policy, out of all discussed policies FF with FIFO performs best although FF, FIFO and LA do not perform bad but there are communication overheads due to LA at medium loads.

CONCLUSION

Putting it all together, we have observed the working of four different algorithms – matrix multiplication, quick-sort, odd-even transposition sort and enumeration sort - in parallel and observed how they performed on different input sizes. We could observe over different topologies, the efficacy looked similar but had different PWS values, these differences have been captured by PWS which can be used to characterize the parallelizability of the program. Further, we also discussed about few static allocation policies and compared its performance as well over different loads, out of which First Fit with First in First Out policy performed the best. However, there are some shortcomings, the overhead generated when simultaneous processors are working is not taken into account and the experiments made above only were had uniform memory access, non-uniform memory access should also be considered. One more possible are of exploration could be trying various other topologies and not limiting to only few topologies, the above experiment did not try out all the topologies to every algorithm, trying different topologies to every algorithm may give further insights.