Task Packing to Maximize CPU Utilization

Report submitted in fulfillment of the requirements
for the Algorithms project of

Fourth Semester

By
Kshitij Ramesh Tadkase
And
Nandika Rohilla

Under the guidance of
Dr. Ravi Shankar Singh

Department of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY (BHU) VARANASI
Varanasi 221005 India

Acknowledgements

We would like to express our sincere gratitude to our supervisor Dr. Ravi Shankar Singh and Mr. Navin Mani Upadhyay for their constant guidance and support during the project work.

Place: IIT (BHU) Varanasi
Date: 30/04/2020

Kshitij Ramesh Tadkase and Nandika Rohilla
18075029 18075040

ABSTRACT

Although an application is balanced, due to external factors load imbalance can occur in a parallel computing system.
In this report, we aim to improve the CPU scheduling process by carrying out **Task Packing** using **dynamic programming** approach. This is a NP hard problem and thus we have devised an exponential time solution using dynamic programming.
Our main aim is to reduce the number of CPUs used for scheduling of the processes keeping execution time same.
The aim is to reduce the idle cycles for the CPUs such that they can be put to some other use.

Contents

1. Introduction

1.1 Load imbalance in parallel computing:

Load imbalance is the uneven distribution of work across resources. Load imbalance can impact scalability. [1]
It is a source of efficiency loss in parallel computing systems. Load imbalance results in loss of resources due to wasting of CPU cycles in idle state of processes, also it reduces the performance capability of an application by inefficient use of the High Performance Computing (HPC) system. [2]
With the ever-growing energy requirements of the High Performance Computing systems, it is now crucial to save the wasted resources to the farthest level possible.
Due to the complexity of the problem and the variety of data / scenarios it is presented, load imbalance does not have any standard solution.
Due to the unavailability of a standard solution, programmers tend to tune their schedulers by hand by redistributing data or by applying some load balancing method to assist their program code. This process is monotonous and boring also.

1.2 Objective:

In this project we aim to reduce the load imbalance in parallel systems that results in the wastage of resources in the form of wasted CPU cycles in Idle state. Multiple tasks arrive and are allocated to the CPUs refer Fig.1 ,[2] among these tasks are different requirements of CPU time , so after the usage of computation power the smaller tasks are required to wait for the slowest task to reach synchronization point to move ahead and allocate the CPUs to next batch of tasks.

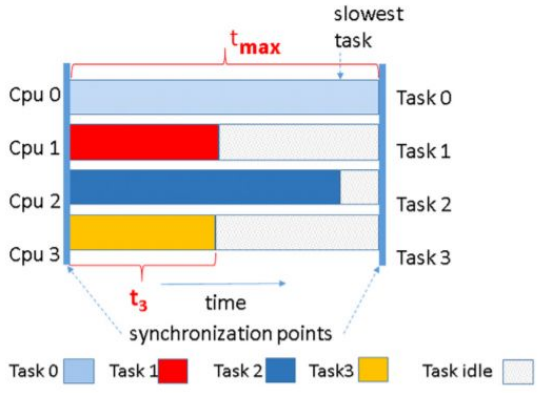

Fig. 1. Unbalanced parallel task execution between two synchronization points.

In this case, Task 0 is the slowest task, so other tasks are required to wait in idle mode. Here our job proposes that by packing tasks 1 and 3 together, we can free a CPU (thus saving crucial

computation power) and this does not increase the time too, since the tasks are scheduled to wait till the next synchronization point. We used the Subset Sum algorithm to carry out the task packing to maximize CPU utilization. [2]

1.3 Dynamic programming approach:

We observe that the problem of assigning tasks to CPU has Optimal Substructure and it involves solving subproblems at each step. This gives the idea of using dynamic programming in the task packing algorithm for assigning a minimum number of CPUs and hence reducing idle cycles of the CPUs in the system. This results in maximum usage of the available resources and getting free CPUs which can then be used to do other tasks.
We have used Bitmasks and Dynamic Programming to get the subset sum and arrange the processes in a sequence which gives the optimal scheduling. [3]

1.4 Our Contribution

We have implemented the subset sum algorithm to carry out CPU scheduling to reduce the number of CPUs. Apart from this, we have tried to reduce power consumption in the project by optimizing the code to take minimum time to run.
We have modified the basic subset sum algorithm [4] alongwith using bitmasks to reduce the runtime of the code and provide results with minimum time keeping in mind the main aim to reduce idle cycles of CPU and using minimum CPUs.

2. Problem Statement

2.1 Task and defining the problem:

The task is to carry out CPU scheduling and create a subset sum invoker for the Task Packing. The aim is to assign a minimum number of CPUs to the process and hence reducing the number of idle cycles of CPUs all the while not affecting the process time limits.
Given the number of processes and the computation time requirements of each process (excluding the synchronization time). A subset sum algorithm is used along with bitmask in the approach to schedule the processes on the CPUs and output the minimum number of CPUs used and the indexes of the processes on each CPU.

We propose to use a scheduling strategy to reallocate tasks in CPUs in such a way that **n tasks**, each one with time duration **t_i** and being **t_max** the largest task duration, can be bounded to a CPU if they satisfy the condition: $\sum t_i \leq t_{max}$ (for i=0 to n) .

This task packing ensures on one hand that a given CPU only has useful cycles, and on the other hand, that idle cycles are concentrated so that some CPUs can be freed from executing the application, therefore minimizing the number of CPUs used by the application.[2]

After getting a sequence of processes/ jobs to be executed, we can allot CPUs to the processes using a simple greedy approach according to the sum of their time durations **t_i** such that,

let, the number of processes on any CPU = k
each process has a time duration, let it be t_i for i=1 to k (processes are indexed from 1)
we have max time duration = t_max
so, sum of time duration of processes on the CPU will be less than or equal to t_max.

3. Our Approach

3.1 Implementation details:

Experimental Setup:
First, we take inputs for the scheduler. The number of processes are stored. The processes with their index and iteration times are stored in a process array in the form of pairs.

```
pair<int,int> process[no_of_proc];   //this pair array stores the indices and times of processesT
```

We will compare CPU usage assuming the condition when each process is given a unique CPU for execution.

3.1.1 Algorithm for scheduling:

1. Sort the processes according to their iteration times in ascending order.
2. Store the time taken by maximum time taking process as max_sized_proc. This will be the upper bound of the sum of times of process in our subset. This will also be the upper limit on the total time of processes on a single CPU.
3. The Dynamic Programming algorithm is implemented by function subsetSum which is explained in section 3.1.2.
4. After step 3 we will have a sequence of processes which need to be carried out through a parent array (this is a helper array to store sequence of processes). We store it in a vector named arr.
5. Now, we have our maximum limit previously stored in step 2.
6. Now, we will just use the maximum limit and assign processes to CPUs greedily such that no CPU is left idle if it can still accommodate some process hence ensuring the minimum number of CPU as shown in Fig. 2.

```
sort(process,process+no_of_proc);          //processses are sorted and the max iter time is stored.
int max_sized_proc=process[no_of_proc-1].first;
int max_limit=(1<<no_of_proc)-1;

 //function for getting sequence through which the processes will be scheduled
subsetSum(max_limit, no_of_proc, process, max_sized_proc);

int ini=max_limit;
vector<int>arr;                           //has the sequence of processes
while (ini>0)
{
  arr.push_back(parent[ini]);
  ini^=(1<<parent[ini]);
}
reverse(arr.begin(),arr.end());
vector<vector<int> >cpu;                  //for storing the CPU which is alloted to the processes.
int s=0;
vector<int>temp;
for(auto i:arr)
{
  if(s+process[i].first>max_sized_proc)
  {
    cpu.push_back(temp);
    temp.clear();
    temp.push_back(process[i].second);
    s=process[i].first;
  }
  else
  {
    s+=process[i].first;
    temp.push_back(process[i].second);
  }
}
cpu.push_back(temp);
```

Fig .2 Shows how the main scheduling works.

3.1.2 Algorithm for Dynamic Programming:

1. The array dp stores pairs in which dp[i] = {x,y} stores: x number of CPUs are used till first i processes and they have y time left to complete.
2. Now we start filling the dp in a bottom-up manner. Here the set bits in binary representation of i gives the CPUs which are alloted some process as shown in Fig . 3.
3. We simultaneously store the indexes of the process with their previous process through the helper array i.e. parent array. This parent array gives the sequence in which the processes are carried out.

```
void subsetSum(int max_limit, int no_of_proc, pair<int,int>process[], int max_sized_proc)
{
  dp[0]={0,0};
  for(int i=0;i<=max_limit;i++)
  {
    for(int j=0;j<no_of_proc;j++)
    {
      if((i&(1<<j))==0)
      {
        int z=dp[i].second+process[j].first;
        int cnt=dp[i].first;
        if(z>max_sized_proc)
        {
          cnt++;
          z=process[j].first;
        }
        else if(z==max_sized_proc)
        {
          cnt++;
          z=0;
        }
        if(dp[i|(1<<j)].first==cnt)
        {
          if(z<dp[i|(1<<j)].second)
          {
            parent[i|(1<<j)]=j;
            dp[i|(1<<j)]=min(dp[i|(1<<j)],{cnt,z});
          }
        }
        else if(dp[i|(1<<j)].first>cnt)
        {
          parent[i|(1<<j)]=j;
          dp[i|(1<<j)]={cnt,z};
        }
      }
    }
  }
}
```

Fig. 3 shows the working of subset sum function

3.2 Execution Framework:

Data is collected and the code is executed on a Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz system with 2TB main memory and the results are analysed on **Qtiplot**.

4. Results and Evaluations

We plotted the time taken to schedule the processes against the number of processes and the graph obtained is shown in Fig. 4. This shows that as the number of processes to be scheduled increases, the program takes more time to allot CPUs.
But this algorithm is quite fast when the number of processes is around 30.

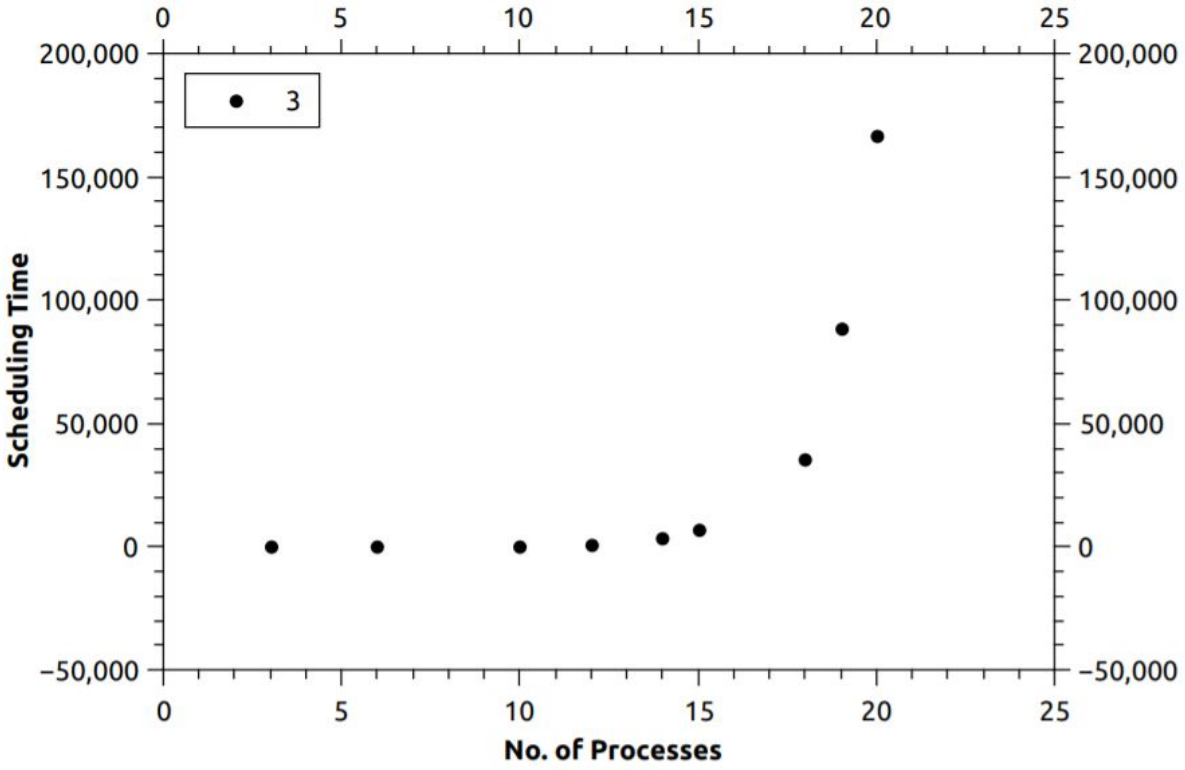

Fig. 4 graph showing scheduling time with number of processes.

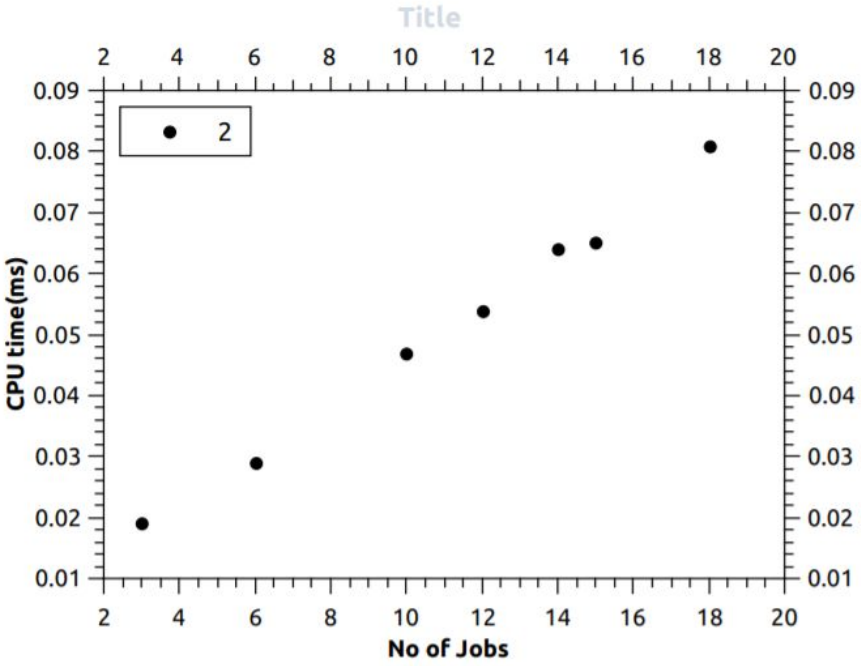

Fig. 5 graph shows how CPU time increases with increase in the number of processes.

Next, to check power consumption on the CPU, we plotted number of processes to be scheduled by time it takes on the CPU i.e. CPU time.
We used Intel's Vtune Profiler for calculating the CPU time used by our program and then used the data obtained to plot the graph using Qtiplot. The graph is shown in Fig. 5

5. Conclusions and Discussions

The current work aims to maximize CPU utilization by packing the given tasks over minimum possible CPUs by the calling of Subset Sum Algorithm to reduce the load imbalance in parallel computing systems.
The Subset Sum Algorithm in use here is NP-complete, so the solution occurs in polynomial time and isn't the most efficient , as seen with results , it loses the value of CPU utilization due to increase in scheduling times as the number of tasks rise above 30.
Future work will be oriented towards finding the optimal solution keeping scheduling time in mind while not compromising the primary aim.

References

1. https://www.sciencedirect.com/topics/computer-science/load-imbalance
2. Gladys Utrera, Montse Farreras, Jordi Fornes. "Task Packing: Efficient task scheduling in unbalanced parallel programs to maximize CPU utilization", Journal of Parallel and Distributed Computing 134 (2019) 37–49.
3. https://www.hackerearth.com/practice/algorithms/dynamic-programming/bit-masking/tutorial/
4. E. Horowitz, S. Sahni, Computing partitions with applications to the knapsack problem, J. ACM 21 (2) (1974) 277–292