Grid Scheduling using 2-Phase Prediction (2PP) of CPU Power

Nguyen The Loc[†], Said Elnaffar[††], Takuya Katayama[†], and Ho Tu Bao[†]
[†]*Japan Advanced Institute of Science and Technology, 1-1 Asahidai, Nomi, Ishikawa Japan*
[††]*College of IT, UAE University, Al-Ain, UAE*
[†]*{ lnguyen, katayama, bao}@jaist.ac.jp;* [††]*elnaffar@uaeu.ac.ae*

Abstract

Divisible workloads are that kind of workloads that can be partitioned by the scheduler into arbitrary 'chunks'. The problem of scheduling divisible loads has been defined for a long time, however, handful solutions have been proposed. Furthermore, almost all proposed approaches attempt to perform scheduling in a dedicated environment (i.e., for processing local tasks only) such as a LAN, whereas scheduling in non-dedicated environments (i.e., for processing local and external tasks) such as Grids remains an open problem. In Grids, the incessant variation of workstation's power is the chief difficulty in planning how to split and distribute workloads to these workstations. This paper presents a new strategy, called 2-Phase Prediction (2PP) for CPU power. By integrating this strategy and the UMR algorithm, a static scheduling algorithm that is designed for dedicated environments, we develop a new dynamic scheduling algorithm suitable for non-dedicated environment. Our experimental results show that our algorithm is superior to the UMR as the former is able to adapt to the dynamicity of Grid workers.

1. Introduction

A Divisible Load [1] is the load that can be arbitrarily partitioned into any number of fractions. It is typically encountered in many domains of science and technology such as protein sequence analysis, simulation of cellular micro physiology, parallel and distributed image processing, video processing, and multimedia [2]. The loads of these applications are inherently colossal such that more than one worker is needed to handle it. The profusion of workers in a distributed computing environment such as the Grid [2] makes the latter a promising platform for processing divisible loads. As usual, this raises the question concerning the problem of scheduling, that is, how to divide a workload that resides at a computer (the master) into chunks and how to assign these chunks to other Grid computers (workers) so that the execution time (makespan) is minimal.

Abundant scheduling approaches and algorithms have been proposed, however, almost all of them assume that computational resources at workers are dedicated. This assumption renders these algorithms impractical in distributed environments such as Grids where computational resources are expected to serve local tasks, which have the higher priority, in addition to the Grid tasks. The purpose of our research is to develop an efficient multi round scheduling algorithm for non-dedicated dynamic environments such as Grids.

The contributions of our paper can be summarized as follows:

- Building a model to represent worker's activities with respect to processing local and external Grid tasks.
- Developing a new strategy for predicting the computing power of a worker, i.e., the portion of the original CPU power that can be donated to Grid applications.
- Proposing a new dynamic scheduling algorithm by incorporating the performance model and the prediction method into the UMR algorithm [3], which is originally a static scheduling algorithm.

The rest of the paper is organized as follows. Section 2 reviews some of the static and dynamic scheduling algorithms. Section 3 briefly describes the heterogeneous platform on which our algorithms operate and present an execution model for local and Grid tasks. Section 4 presents our CPU power prediction strategy and explains how to incorporate it into the scheduling algorithm. Section 5 describes the simulation experiments we have conducted in order to evaluate the proposed algorithms. Section 6 concludes our paper.

2. Related Work

Divisible loads scheduling is based on the Divisible Load Theory [1]. The goal of load scheduling is to minimize the overall execution time (or makespan) by finding an optimal strategy to split the total loads into many chunks as well as finding a reasonable order of workers to receive chunks from the master. The first multi-round algorithm, which was introduced by Bharadwaj as a Multi-Installment algorithm (MI) [1], optimizes the makespan by exploiting the overlap between computation and communication processes.

Beaumont [4] proposes another multi-round scheduling algorithm that fixes the execution time throughout any round. Yang et al. extend the MI algorithm by making it more realistic as they factor in the computation and communication latencies. Their UMR (Uniform Multi Round) algorithm [3] is ultimately based on the premise of making the total time of data transfer and execution the same in each round for each worker. This assumption enables them to analyze the constraints and determine the near-optimal number of rounds as well as the size of chunks in each round. Our work in this research extends the UMR algorithm and augments its scheduling mechanism.

The above described algorithms are deemed static because they assume that the full computational capacity of workers is constantly available and can be readily tapped into, which makes them impractical for dynamic environments such as the Grid. Workers hooked to the Grid are supposed to handle locally arriving tasks, first, and donate their unused time to the external Grid tasks. As a result, any scheduling that assumes guaranteed CPU capacity of a worker is deemed implausible in this dynamic environment.

The RUMR [5] is a step towards algorithm dynamicity as it shows tolerance towards errors in predicting the available CPU power using the Factoring method. However, all of the RUMR parameters are determined once before the RUMR starts and retained fixed afterwards, which makes the RUMR a non-adaptive scheduling algorithm. Apparently, dynamic algorithms are more appropriate for Grids. In [6] we use Mixed-Tendency Based prediction strategy to estimate workers' power. However, the Mixed-Tendency Based strategy [7] considers the aggregate execution of applications, while our computation model discussed in this paper is based on the M/M/1 model [8] that ultimately takes into consideration the distinction between local and Grid task. This should present a more realistic model.

3. Grid Computation Model

3.1. Heterogeneous Platform

Let us consider a computation Grid in which a master process has access to N computing workers. We assume that the master uses its network connection in a sequential fashion. i.e., it does not send chunks to some workers simultaneously. Workers can receive data from network and perform computation simultaneously. The following notation will be used throughout this paper:

- W_{total}: the total amount of workload.
- M: the number of scheduling rounds.
- $chunk_{ji}$: the fraction of total workload that the master delivers to worker i in round j (i = 1,...,N ; j = 1,...,M).
- S_i: computation speed of worker i (flop/s).
- B_i: the data transfer rate of the connection link between the master and worker i (flop/s).
- ES_i: estimated speed of worker i for Grid tasks on the next round (see Section 4).
- $round_j$: the fraction of workload dispatched during round j.
- $Tcomp_{ji}$: computation time required for worker i to process $chunk_{ji}$.

3.2. Non-Dedicated Model

During the execution of a Grid task, some local tasks may arrive causing to interrupt the execution of the lower priority Grid tasks. The arrival of the local tasks of worker i is assumed to follow a Poisson distribution with arrival rate λ_i, their execution process follows an exponential distribution with service rate μ_i and the local task process in the worker is an M/M/1 queuing system [8]. The execution time $Tcomp_{ji}$ of $chunk_{ji}$ on worker i can be expressed as:

$$Tcomp_{ji} = X_1 + Y_1 + X_2 + Y_2 + \ldots + X_{NL} + Y_{NL}$$

where

- NL: the number of local tasks which arrive during the execution of $chunk_{ji}$
- Y_k: execution time of the local task k (k = 1,2,...,NL)
- X_k: execution time of k^{th} section of $chunk_{ji}$. We have:

$$X_1 + X_2 + \ldots + X_{NL} = chunk_{ji} / S_i$$

From the M/M/1 queuing theory [8] we have:

$$E(NL) = \frac{\lambda_i chunk_{ji}}{S_i}; E(Y_k) = \frac{1}{\mu_i - \lambda_i}$$

Because NL and Y_k are independent random variables (k = 1,2,...,NL) we derive

$$E\left(Tcomp_{ji}\right) = E\left(Tcomp_{ji} \mid NL\right) = \sum_{k=1}^{NL} X_k + $$

$$+ \sum_{k=1}^{NL} E\left(Y_k\right) = \frac{chunk_{ji}}{S_i} + E\left(NL\right)E\left(Y_k\right) = \frac{chunk_{ji}}{S_i\left(1-\rho_i\right)}$$

where $\rho_i = \lambda_i/\mu_i$, which represents the CPU Utilization. ρ_i, λ_i, μ_i are representative on the long run but cannot be used to estimate the imminent execution time that will take place on a given worker. Therefore, we introduce the adaptive factor δ_i, which represents the credibility of performance prediction for worker i and it is initialized to 1 at the beginning of the scheduling process (i.e., in the first round). At the end of each round, δ_i is updated as follows: $\delta_i = FS_i / ES_i$ where FS_i denotes the factually measured available CPU power. Now the expected value of the execution time of $chunk_{ji}$ is

$$\frac{chunk_{ji} \times \delta_i}{S_i\left(1-\rho_i\right)}$$

Since the actual power of workers available to the Grid tasks varies over time, we have to predict how δ_i changes, as explained next.

4. The 2PP-Based Scheduling Algorithm

Our scheduling algorithm consists of two components: the 2-Phase Prediction (2PP) strategy and the UMR static scheduling algorithm. Before any scheduling round commences, the 2PP strategy is invoked to estimate the available CPU power (ES_i) at each worker. In light of the CPU power estimation the UMR splits and dispatches the appropriate load chunks at each round.

4.1. The 2-Phase Prediction (2PP) Strategy

For the sake of readability, we drop the use of the subscript i that refers to worker i in this section. In order to estimate the next δ for a particular worker, we consider the historically measured time series $c_1, c_2,...,c_n$. Data point c_t is value of δ at time t. This time series of δ is sampled at some frequency (e.g., 0.1 Hz) during the execution of a round. However, we are interested in estimating δ for the upcoming round, not for the upcoming time tick. Therefore, we need to compress the original time series into interval time series by aggregating the former as follows: If we denote D as the aggregation degree, where

D = execution time of a round × frequency of original time series

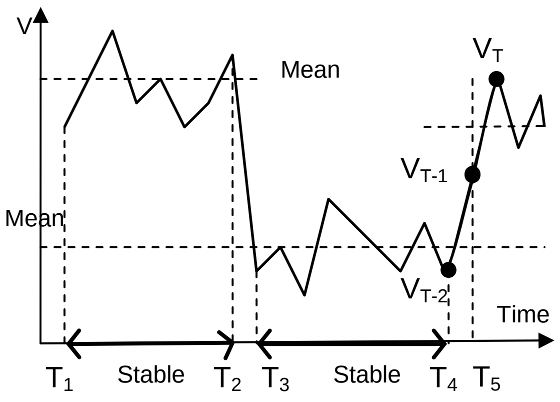

Fig. 1. 2-Phase Prediction (2PP) Strategy

Then the interval time series $V_1, V_2, \ldots, V_k$ ($k = \lceil n/D \rceil$ can be calculated as follows:

$$V_r = \frac{\sum_{j=1}^{D} \delta_{n-(k-r+1)D+j}}{D} \qquad r = 1,2,\ldots,k$$

Each value V_r is the average value of the adaptive factor δ over a round. The 2PP strategy operates on this V_r time series in order to predict V_{k+1} of the next round. Since δ plays the role of a smoothing factor that progressively adjusts the estimated CPU power available, we should expect that its interval average, V_r, should oscillate between some periods of stability and others of conversion as shown in Figure 1. In the stable stage, the available CPU power exhibit less variation as it approaches some constant. The time intervals (T_1, T_2) and (T_3, T_4) are examples of the stable stage. In the conversion stage, the available CPU power tends to experience major changes due to increase or decrease in the arrival rate of local tasks. The time intervals (T_2, T_3) and (T_4, T_5) represent conversion stages. Toggling between different stages can be identified by comparing the current absolute deviation $\left| V_T \text{ - } Mean \right|$ with a threshold value *threshold*. Algorithm 1 outlines the 2PP strategy where:

- V_T : the value of current data point.
- V_{T-1}: the value of last data point.
- V_{T+1}: the estimated value of the next data point.
- *Mean*: the mean value of data points in current stage.
- *T*: current time point
- *H*: the starting point of current stage

The procedure *UpdateMean()* simply adjusts the mean as follows:

$$Mean = \frac{V_H + V_{H+1} + \ldots + V_T}{T - H + 1}$$

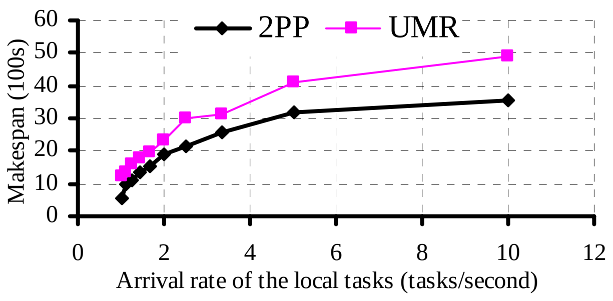

Fig. 2. Performance comparison under configuration I

The procedure *UpdateThreshold()* updates the threshold as follows: if L denotes the number of historical thresholds, and $|V_T - Mean|$ denotes the current threshold value, then the updated threshold is:

$$threshold = \frac{L \times threshold + |V_T - Mean|}{L+1}$$

The predicted value of V_{T+1} is used as an estimate for the adaptive factor, δ, for the upcoming round. Subsequently, we can compute the average speed, ES_i, of worker i on the next round as follows:

$$ES_i = S_i(1-\rho_i)/\delta_i$$

Algorithm 1: 2PP Strategy

```
Begin
  $CurrentStage$ = “stable”; $Threshold = 2(V_2 - V_1)$;
  Repeat
    if $CurrentStage$ == “stable”
      if $|V_T - Mean| > threshold$
        begin   // Conversion stage is starting
          UpdateThreshold();
          $CurrentStage$ = “conversion”;
          $V_{T+1} = 2.V_T - V_{T-1}$;
        end
      else      // Stable state, continue
        begin
          UpdateMean(); $V_{T+1} = 2.Mean - V_T$;
        end
    else        // CurrentStage == “Conversion”
      if $(V_T - V_{T-1}) \times (V_{T-1} - V_{T-2}) < 0$
        begin // Stable state is starting
          CurrentStage = “stable”; $H = T-1$;
          UpdateMean(); $V_{T+1} = 2.V_T - V_{T-1}$;
        end
      else      // Conversion, continue
        $V_{T+1} = 2.V_T - V_{T-1}$;
  Until all of $W_{total}$ is processed;
End
```

4.2. Task Scheduling

Our scheduling relies on the UMR's [3] scheduling method. As worker N processes chunk j, the master sends (N-1) chunks to the (N-1) remaining workers. To maximize bandwidth utilization, the master must finish sending the last $chunk_{j+1,N}$ of $round_j$ to the last worker N before worker N finishes processing $chunk_{j+1,N}$. Algorithm 2 outlines our scheduling algorithm. Initially, ES_i is computed using the 2PP method, and $round_0$, $chunk_{0i}$ are computed using the UMR's initialization procedure. The chunks of the first round get delivered. The algorithm keeps running until no workload is remaining. In each iteration, our algorithm uses the 2PP strategy to estimate the ES_i for each worker before the start of each round. It then uses the UMR's scheduling method to compute $round_j$ and $chunk_{ji}$ in light of ES_i.

Algorithm 2: Proposed Scheduling Algorithm

```
Begin
  compute $\{ES_i\}$, $M$, $round_0$, $\{chunk_{0i}\}$ $(i = 1,2,...,N)$
  $W_{remains} = W_{total} - round_0$;
  deliver $\{chunk_{0i}\}$ to {worker $i$} $(i = 1,2,...,N)$
  Repeat        //Processing on round j
    collect items of the series C in the last round
    use 2-PP to derive $\{\delta_i\}$ $(i = 1,2,...,N)$
    compute $\{round_j, chunk_{ji}, ES_i\}$ $(i = 1,2,...,N)$
    if $(round_j > W_{remains})$ $round_j = W_{remains}$;
    $W_{remains} = W_{remains} - round_j$;
    deliver $\{chunk_{ji}\}$ to {worker $i$} $(i = 1,2,...,N)$
  Until $W_{remains}$ == 0
End
```

Table I. Parameters of Configuration I

Worker	CPUpower (Mflop/s)	Bandwidth (Mflop/s)	cLat (s)	nLat (s)
iRMX	100	500	0.1	0.053
Kunig	80	423	0.1	0.053
Bosqet	80	408	0.1	0.053
Soucy	30	169	0.1	0.053
Brown	30	153	0.1	0.053
Steph	30	28	0.1	0.053
Robert	20	57	0.1	0.053
Sirois	50	45	0.1	0.053
Moniq	40	24	0.1	0.053
Jacks	40	49	0.1	0.053

5. Experiments

In order to evaluate the new algorithm, we developed a simulator using the SIMGRID [9] toolkit, which is specially designed for building simulations for various scheduling algorithms in parallel and distributed environments. We compare the

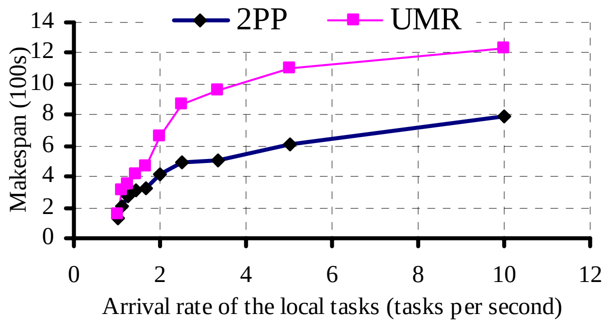

Fig. 3. Performance comparison under configuration II

performance of our algorithm with the original UMR algorithm using two experimental configurations. Under configuration I, we have the following setup:

- Workers: we use 10 workers whose system properties are shown in Table 1.
- Total load (W_{total}): 1000 (Mflops).
- The average processing time of local tasks: 20 (s).

One of the chief differences between the UMR algorithm and ours is the ability of the latter to scheduling load chunks in light of the estimated CPU power for each worker. Hence, and in order to quiz the performance of the two algorithms, we intensified the arrival rate of local tasks on the strongest worker, iRMX, by ten times more than any other worker. As a result, this worker ends up, practically, being the weakest worker with respect to the available CPU power for the Grid tasks. Unlike our 2PP-based algorithm, the UMR does not recognize this fact as it assumes that iRMX continually offers all of its capacity to the Grid tasks. Therefore, the UMR mistakenly keeps sending the bigger chunks of workload to iRMX, which leads to performance deterioration. Figure 2 shows the performance of the UMR vs. our 2PP-based algorithm under different arrival rate of local tasks. The 2PP algorithm keeps outperforming the UMR with respect to the task makespan. Similarly, we experiment with configuration II that has the following setup:

- Number of workers *N*: 30.
- The average power of worker: 60 (Mflop/s).
- The average bandwidth: 50 (Mflop/s).
- Total load (W_{total}): 5000 (Mflop).
- Computation and communication latencies: 0.1 (s)
- The average processing time of local tasks: 40 (s).

As we do in the first configuration setup, we exposed the top 10% of the workers to a higher arrival rate of local tasks. Again, as shown in Figure 3, the 2PP outperforms the UMR as the latter is not aware of the run-time availability of the actual CPU power of workers.

6. Conclusion

In this paper, we presented a dynamic scheduling algorithm that is built on top of the static UMR algorithm after augmenting it with our 2PP strategy for CPU power. We discussed the task execution model that takes into account processing local and Grid tasks on each worker. We used this model to perform near future forecasting for the available CPU power at each worker machine. Based on the estimated run-time computational power available, we decide on how to distribute workload chunks. The superior results that our algorithm exhibit suggest that the 2PP-based algorithm is adaptive and more suitable for dynamic, non-dedicated environments such as the Grid.

Acknowledgement

Our research is conducted as a program for the ”Fostering Talent in Emergent Research Fields" in Special Coordination Funds for promoting Science and Technology by Ministry of Education, Culture, Sports, Science and Technology, Japan. Our research has also been supported by research grant #02-06-9-11/06 from the Scientific Research Council of the UAE University.

References

[1] V. Bharadwaj, D.Ghose, V.Mani, and T. G. Robertazzi, *Scheduling Divisible Loads in Parallel and Distributed Systems*, IEEE Computer Society Press, 1996.

[2] I. Foster and C. Kesselman, *Grid2: Blueprint for a New Computing Infrastructure*, second ed. San Francisco, Morgan Kaufmann Publisher, 2003.

[3] Y. Yang, K.V. Raart, and H. Casanova, “Multiround Algorithms for Scheduling Divisible Loads”, *IEEE Transaction on Parallel and Distributed Systems*, Nov. 2005, Vol. 16.

[4] O. Beaumont, A. Legrand, and Y. Robert, “Scheduling Divisible Workloads on Heterogeneous Platforms”, Parallel Computing, Sep. 2003, Vol. 9.

[5] Y. Yang and H. Casanova, “RUMR: Robust Scheduling for Divisible Workloads”, *HPDC'03* Seattle, USA, 2003.

[6] N.T. Loc, S. Elnaffar, T. Katayama, and H.T. Bao, “A Scheduling Method for Divisible Workload Problem in Grid Environments”, *PDCAT 05*, Dec. 2005, Dalian, China.

[7] L. Yang, J. Schopf and I. Foster, “Homeostatic and Tendency-based CPU Load Predictions”, *IPDPS'03*, Nice, France, Apr. 2003.

[8] A. Papoulis and S. U. Pillai, *Probbility, Random Variables and Stochastic Processes*, McGraw-Hill 2002.

[9] A. Legrand, L. Marchal, and H. Casanova, “Scheduling Distributed Applications: the SimGrid Simulation Framework”, *CCGrid'03*, Japan, 12-15 May 2003.