A Dynamic Scheduling Algorithm for Divisible Loads in Grid Environments

Nguyen The Loc
Hanoi National University of Education, Hanoi, VietNam
Email: locnt@hnue.edu.vn

Said Elnaffar
College of IT, UAE University, Al-Ain, UAE
Email: elnaffar@uaeu.ac.ae

***Abstract*—Divisible loads are those workloads that can be partitioned by a scheduler into any arbitrary chunks. The problem of scheduling divisible loads has been defined for a long time, however, a handful of solutions have been proposed. Furthermore, almost all proposed approaches attempt to perform scheduling in dedicated environments such as LANs, whereas scheduling in non-dedicated environments such as Grids remains an open problem. In Grids, the incessant variation of a worker's computing power is a chief difficulty of splitting and distributing workloads to Grid workers efficiently. In this paper, we first introduce a computation model that explains the impact of local (internal) tasks and Grid (external) tasks that arrive at a given worker. This model helps estimate the available computing power of a worker under the fluctuation of the number of local and Grid applications. Based on this model, we propose the CPU power prediction strategy. Additionally, we build a new dynamic scheduling algorithm by incorporating the prediction strategy into a static scheduling algorithm. Lastly we demonstrate that the proposed dynamic algorithm is superior to the existing dynamic and static algorithms by a comprehensive set of simulations.**

***Index Terms*—CPU power prediction, divisible loads, Grid scheduling.**

I. 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 them. 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 begs the typical scheduling question of how to divide a workload that resides at a computer (*master*) into chunks and how to assign these chunks to other Grid computers (*workers*) so that the execution time (*makespan*) is minimal.

Numerous scheduling approaches and algorithms have been proposed, however, the majority of them assume that the computational resources at workers are dedicated. This assumption renders these algorithms impractical in distributed environments such as the Grid 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 computation model that explains the performance of the worker under the impact of processing local applications as well as Grid tasks.
- Developing a new strategy, 2PP (Two Phase Prediction), for predicting the computing power of a worker, i.e., the fraction of the original CPU power that can be donated to the incoming Grid applications.
- Proposing a new dynamic scheduling algorithm by incorporating the prediction strategy 2PP into the MRRS (Multi-round Scheduling with Resource Selection) algorithm [3, 4], which is originally a static scheduling algorithm.

The rest of the paper is organized as follows. Section II reviews some of the static and dynamic scheduling algorithms. In Section III, after defining the scheduling problem in non-dedicated environments we present a performance model for the computations that take place at workers. This model helps estimate the computing power of a worker under the fluctuation of local applications vs. Grid tasks. Section IV explains how our CPU power prediction strategy, 2PP, is built on top of this worker computation model. Section V reviews the

Based on "Grid Scheduling Using 2-Phase Prediction (2PP) of CPU Power", by Nguyen The Loc, Said Elnaffar, Takuya Katayama, and Ho Tu Bao, which appeared in the Proceedings of the IIT'06, IEEE Communication Society Press, IEEE Catalogue Number 06EX1543C, ISBN 1-4244-0674-9, Dubai, UAE (November 19-21, 2006), © 2006 IEEE.

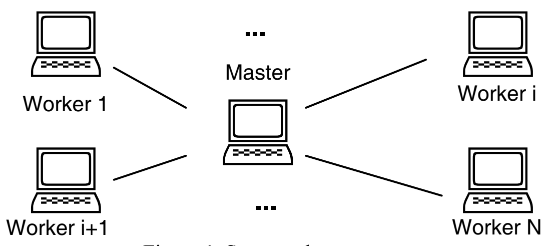

Figure 1. Star-topology

MRRS static algorithm and explains how to integrate it with 2PP in order to build our proposed dynamic scheduling algorithm. Section VI describes the experiments we have conducted in order to evaluate our work. Section VII concludes the paper.

II. Related Work

Most of the studies that focus on scheduling divisible loads are based on the Divisible Load Theory [1]. The goal of load scheduling is to minimize the overall execution time (hereafter called *makespan*) by finding an optimal strategy of splitting the original load received by the master computer into a number of chunks as well as distributing these chunks to the workers in the right order.

The first scheduling algorithm, named MI (Multi-Installment) [1], optimizes the makespan by exploiting the overlap between computation and communication processes. Beaumont [5] proposes another multi-round scheduling algorithm that fixes the execution time throughout each round. Yang et al. extend the MI algorithm by making it more realistic by factoring in the computation and communication latencies. Their UMR (Uniform Multi Round) algorithm [6] 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. Based on the theoretical analysis as well as simulation results [4], UMR exhibits the best performance among its family of algorithms.

The MRRS (Multi-round Scheduling with Resource Selection) algorithm [3,4] extends the UMR by considering the network bandwidth and latency in addition to the computation capacity of workers. Furthermore, the MRRS is the first scheduling algorithm for divisible loads that is featured with a resource selection policy that finds the best subset of available computers.

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 [7] algorithm is a step towards 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 remain fixed afterwards, which makes RUMR a non-adaptive scheduling algorithm. Apparently, dynamic algorithms are more appropriate for Grids.

To the best of our knowledge, the algorithm discussed in [8] is the first dynamic scheduling algorithm for divisible loads in non-dedicated environments. It employs the tendency-based prediction strategy described in [9,10] in order to be adaptive to the Grid. In this paper, we introduce a new dynamic algorithm, named 2PP, for which the theortical analysis and the experimental results show that it outperforms the previous static and dynamic algorithms.

III. Grid Computation Model

A. Heterogeneous Configuration

We adopt the same computation model used in [1,5,6,7] where a *master* computer is connected to *n* worker computers in a star-topology network.

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 [1]. The following notations will be used throughout this paper:

- W_i: worker i
- L_{total}: the total amount of workload that resides at the master.
- m: the number of scheduling rounds.
- $chunk_{ji}$: the fraction of total workload that the master delivers to W_i in round j ($i = 1,...,n$; $j = 1,...,m$).
- S_i: computation speed of W_i.
- $cLat_i$: the fixed overhead time needed by W_i to start computation
- $nLat_i$: the overhead time incurred by the master to initiate a data transfer to W_i.
- B_i: the data transfer rate of the connection link between the master and W_i.
- ES_i: estimated speed of worker i for Grid tasks on the next round.
- $round_j$: the fraction of workload dispatched during round j.
- $Tcomp_{ji}$: computation time required for W_i to process $chunk_{ji}$
- $Tcomm_{j,i}$: communication time required for the master to send $chunk_{ji}$ to W_i

B. Problem Statement

The task scheduling problem in non-dedicated environments can be defined as follows. If we have:

- A total amount of divisible load L_{total} that resides at the master.

- A non-dedicated computational platform consists of the master and n workers connected with each other by a star-topology network (Fig. 1).
- And dynamic availability of CPU capacity, i.e. CPU power S_i of worker i varies over time ($\forall i = 1,2,...,n$), which was not the case in previous studies [1,5,6],

Our ultimate question is: given the above platform settings, in what proportion should the workload W_{total} be split up among the heterogeneous, dynamic workers so that the overall execution time is minimum?

Formally, we need to minimize the following objective function:

$$\max_{i=1,2,...n}\left(\sum_{k=1}^{i} Tcomm_{1,k} + \sum_{j=1}^{m} Tcomp_{j,i}\right) \rightarrow \min$$

where the expression between brackets is the total running time, that is, the sum of waiting time, communication time and computation time of worker W_i.

C. Non-Dedicated Platform

We use an M/M/1 queuing system [11] to model the activities that take place at the worker machine. Local and Grid tasks arrive at workers in order to be processed (Fig. 2). If a Grid task cannot be served upon arrival, it joins the service queue whose capacity is assumed to be unlimited. This queuing system has the following characteristic:

- *The input process*. The arriving tasks consist of Grid tasks and local tasks. Grid tasks are the $chunk_{ji}$ portions of total load L_{total}, which are dispatched by the master. The local tasks are tasks that are produced by local applications (e.g. desktop applications) at the worker. The arrival of the local tasks at W_i is assumed to follow a Poisson distribution with an arrival rate λ_i and their service demands follow an exponential distribution with a service rate μ_i.
- *The service mechanism*. During the execution of a Grid task on a certain worker, some local tasks may arrive causing to interrupt the execution of the lower priority Grid tasks. We consider the execution of the local tasks as preemptive, i.e. a local task must be executed until completion once it gets started. The local tasks are processed on a first-come-first-served basis.
- *The worker's capacity*. From the Grid tasks' point of view, the state of a worker alternates between unavailable and available depending on whether the worker is busy with a local task or not, respectively. As stated earlier, S_i denotes the maximum computing power of worker W_i that can be donated to Grid tasks when the worker is absolutely available.

The execution time $Tcomp_{ji}$ of $chunk_{ji}$ on worker W_i can be expressed as:

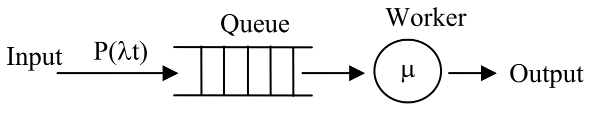

Figure 2. M/M/1 queue

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

where

- NL: the number of local tasks that 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:

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

From the M/M/1 queuing theory [11] we have:

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

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

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

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

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 associated with 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

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

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

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

Our scheduling algorithm consists of two components: the 2-Phase Prediction (2PP) strategy and the MRRS-based 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 MRRS splits and dispatches the appropriate load chunks at each round.

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 the 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

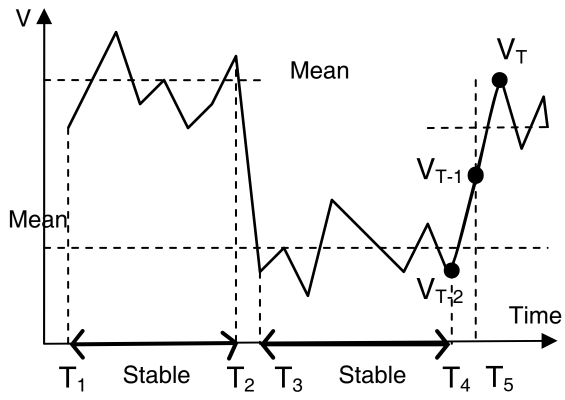

Figure 3. 2-Phase Prediction (2PP) Strategy

$$D = \text{execution time of a round} \times \text{frequency of original time series}$$

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, will oscillate between some periods of stability and others of conversion as shown in Fig. 3. During the stability 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 stages. During the conversion stage, the available CPU power tends to experience major changes due to an increase or decrease in the arrival rate of local tasks. The time intervals (T_2, T_3) and (T_4, T_5) are instances of conversion stages. Toggling between different stages can be detected by comparing the current absolute deviation $|V_T - Mean|$ 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:

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

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 ($V_T - V_{T-1}$)×($V_{T-1}$-$V_{T-2}$) <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
```

V. MRRS Scheduling

We sketch here the static scheduling algorithm MRRS and refer the reader to [3,4] for more information and the detailed derivations.

A. Induction Relation for Chunk Sizes

Fig. 4 depicts how the MRRS algorithm distributes work chunks to workers. At time T_1, the master starts sending *round*$_{j+1}$ amount of load to all workers and the last worker W_n starts working on chunk *j* concurrently. To fully utilize the network bandwidth, the dispatching of the master and the computation of W_n should finish at the same time T_2:

$$\sum_{i=1}^{n}\left(nLat_i + \frac{chunk_{j+1,i}}{B_i}\right) = \frac{chunk_{j,n}}{S_n} + cLat_n$$

If we replace $chunk_{j+1,i}$ and $chunk_{j,n}$ by their expression we derive:

$$round_{j+1} = round_j \times \theta + \mu \qquad (1)$$

where

$$\theta = B_n \Bigg/ \left((B_n + S_n)\sum_{i=1}^{n}\frac{S_i}{B_i + S_i}\right)$$

$$\mu = \left(\frac{\beta_n}{S_n} + cLat_n - \sum_{i=1}^{n}\left(nLat_i + \frac{\beta_i}{B_i}\right)\right) \Bigg/ \sum_{i=1}^{n}\frac{\alpha_i}{B_i}$$

From the induction equation (1) we can compute:

$$round_j = \theta^j(round_0 - \eta) + \eta \qquad (2)$$

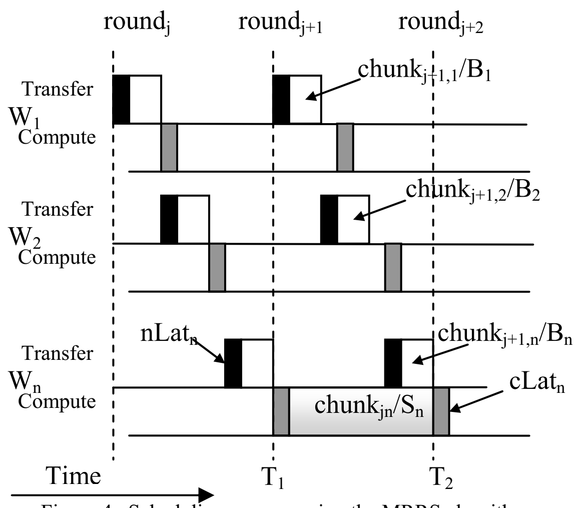

Figure 4. Scheduling process using the MRRS algorithm

where $$\eta = \frac{\beta_n + cLat_n - \sum_{i=1}^{n}\left(nLat_i + \frac{\beta_i}{B_i}\right)}{\sum_{i=1}^{n}\frac{\alpha_i}{B_i} - \frac{\alpha_n}{S_n}}$$

B. Determining the Parameters of the Initial Round

In this section we compute the optimal number of rounds, *m*, and the size of the initial load fragment that should be distributed to workers in the first round, *round$_0$*. Let *F(m,round$_0$)* denote the makespan:

$$F(m, round_0) =$$
$$= round_0\left(\sum_{i=1}^{n}\frac{\alpha_i}{B_i} + \frac{\alpha_n(1-\theta^m)}{S_n(1-\theta)}\right) + \sum_{i=1}^{n}\left(\frac{\beta_j}{B_i} + nLat_i\right)$$
$$+ m\left(cLat_n + \frac{\alpha_n\eta + \beta_n}{S_n}\right) - \frac{\alpha_n\eta(1-\theta^m)}{1-\theta}$$

Our objective is to minimize the makespan *F(m,round$_0$)* , subject to:

$$G(m, round_0) = m\eta + (round_0 - \eta)\frac{1-\theta^m}{1-\theta} - L_{total} = 0 \quad (3)$$

This constrained optimization problem can be solved by the Lagrangian method [12]. After solving this equation system we obtain *m*. Using (3) one can then compute *round$_0$*. At last, using (2) and (1) we will obtain the value of *round$_j$* and *chunk$_{j,i}$* respectively (*i=1..n, j=1..m*).

C. Worker Selection Policy

Let *V* denote the original set of *N* available workers (|*V*|=*N*). In this subsection we explain our resource selection policy that aims at finding the best subset *V** (*V**⊆*V*, |*V**|=*n*) that minimizes the makespan.

Policy I (θ>1)

When θ>1 we get

$$makespan_{MRRS}(V^*) = \frac{L_{total} \times B_n}{(B_n + S_n)\sum_{i\in V^*}\frac{B_iS_i}{B_i + S_i}} + C$$

where *C* is a constant $C = \sum_{i\in V^*} nLat_i + m.cLat_n$

we can see that under this policy, *V** is the subset that maximizes the objective function ψ

$$\psi(V^*) = \sum_{i\in V^*}\frac{B_iS_i}{B_i + S_i}$$

subject to θ>1 or

$$\sum_{i\in V^*}\frac{S_i}{B_i + S_i} < \frac{B_n}{B_n + S_n}$$

One can observe that this is a Binary Knapsack [13] problem that can be solved using the Horowitz-Sahni algorithm [13].

Policy II (θ≤1)

When θ<1, we have to find out the subset *V** such that minimizes the objective function π()

$$\pi(V^*) = \sum_{i\in V^*}\frac{S_i}{B_i + S_i} \Big/ \sum_{i\in V^*}\frac{B_iS_i}{B_i + S_i}$$

subject to θ<1 or $\sum_{i\in V^*}\frac{S_i}{B_i + S_i} > \frac{B_n}{B_n + S_n}$

Similarly, when θ=1 we have to find out the subset V* such that minimizes the objective function π()

subject to θ=1 or $\sum_{i\in V^*}\frac{S_i}{B_i + S_i} = \frac{B_n}{B_n + S_n}$

It can be seen that, this is an Integer Nonlinear Optimization [13] problem. In [3,4] we have designed a Branch and Bound algorithm, called OSS, to solve it. Next, we shed light on some details germane to the worker selection algorithm OSS.

To begin with, let us denote by Ω$_V$ the set of subset of *V*: Ω$_V$ = {*X*: *X*⊆*V*}.

Lemma 1. Consider the following function:

Lower: $\Omega_V \to R$

$$X \mapsto \frac{1}{B_k} : B_k = \max\{B_i : W_i \in X\}$$

Lower() is a lower bound of function π(), i.e. *Lower*(*X*)≤π(*X*) (∀*X*⊆*V*)

Proof. Assume that *X* = {*W$_1$*, *W$_2$*, ... *W$_r$*}. We have:

$$\forall i : B_k > B_i \Rightarrow B_k\sum_{i=1}^{r}\frac{S_i}{B_i + S_i} > \sum_{i=1}^{r}\frac{B_iS_i}{B_i + S_i}$$

$$\Rightarrow \frac{\frac{S_k}{B_k + S_k}}{\frac{B_kS_k}{B_k + S_k}} \le \frac{\sum_{i=1}^{r}\frac{S_i}{B_i + S_i}}{\sum_{i=1}^{r}\frac{B_iS_i}{B_i + S_i}} \Rightarrow L(X) \le \pi(X) \quad \square$$

Let us denote:

- $\hat{u} = (\hat{u}_1, \hat{u}_2, \dots \hat{u}_n)$: the current solution, $\hat{u}_i \in \{0,1\}$. $\hat{u}_i = 1$ if worker i is selected (W_i belongs to V^*), otherwise $\hat{u}_i = 0$.
- $u = (u_1, u_2, \dots u_n)$: the best solution so far, $u_i \in \{0,1\}$
- $\hat{a}$: the value of the current solution, i.e., the value of $\pi()$ associated with the subset $\hat{u}$
- a : the value of the best solution so far, i.e., the value of $\pi()$ associated with the subset u

```
Function Numerator ($\hat{u}$) /* return $\pi()$ at the subset of V
that correspond with $\hat{u}$ */
  Begin
      $y$:=0;
      for $i$:=1 to $n$ do if ($\hat{u}_i$ =1) then $y := y + S_i / (B_i+S_i)$;
      return ($y$);
  End;
Function pi ($\hat{u}$)   /* return $\pi()$ at the subset of V that
correspond with $\hat{u}$ */
  Begin
      $y$:=0;
      for $i$:=1 to $n$ do if ($\hat{u}_i$ =1) then $y := y + B_i S_i / (B_i+S_i)$;
      return (Numerator($\hat{u}$) /$y$);
  End;

Procedure OSS ($V$)    /* Input: $V$; Output: $u$, $a$, $V^*$ */
Begin
    $a := \infty$ ; $\hat{a}$ := 0; $j$:=1;
    While (true) Do
    Begin
      1.  /* estimate the lower bound */
        find $B_{\max} = \max\{B_{j+1}, B_{j+2}, \dots, B_n\}$;
        Lower:=1/$B_{max}$;
        if $a \le \hat{a}$ + Lower then go to 2;
      /* forward */
        $\hat{u}_j$:=1; $\hat{a}$:= pi($\hat{u}$) ; $j$:=$j$+1;
        if $j \le n$ then go to 1.
        if Numerator ($\hat{u}$) > $Z$ then
      /* update the best solution */
        begin $u := \hat{u}$ ; $a := \hat{a}$; end;
      /* remove worker j from the current solution */
        $\hat{u}_n$ :=0; $\hat{a}$:=pi($\hat{u}$);
      2.  /* back track */
        find $i$=max{$k$ | $k$<$j$ and $\hat{u}_k$ =1}
        if no such i then return;
        $\hat{u}_i$ :=0; $j$:=$i$+1; go to 2 ;
    End
End
```

In the OSS algorithm, a *forward move* consists of inserting the next worker W_j into the current solution $\hat{u}$. A *backtracking move* consists of removing the last inserted worker from the current solution. After a *backtracking move*, the lower bound *Lower()* corresponding to the current solution is computed and compared with the value of the best solution so far, a, in order to check whether a further forward move would lead to a better one. If that is the case, a new forward move is performed, otherwise a backtracking move follows. When the last worker W_n has been considered, the best solution is updated. The algorithm terminates when no further backtracking can be performed.

D. The 2PP-based Dynamic Scheduling Algorithm

By integrating the prediction strategy (2PP) with the static scheduling algorithm MRRS we can have the 2PP-based dynamic scheduling algorithm as outlined next:

Algorithm 2: Proposed Scheduling Algorithm 2PP

```
Begin
  Use OSS to select the set of workers $V^*$;
  j:=0;
  Use MRRS to compute {$chunk_{0i}$}
  Deliver {$chunk_{0i}$} to {$W_i$ : $W_i \in V^*$}
  Repeat          //Processing on round j
    j:=j+1;
    Use OSS to select the set of workers V*;
    Use 2PP to estimate {$ES_i$: $W_i \in V^*$}
    Use MRRS to compute {$chunk_{0i}$}
    Deliver {$chunk_{0i}$} to {$W_i$ : $W_i \in V^*$}
  Until $L_{total}$ is finished
End
```

Algorithm 2 shows that we initially use OSS strategy to find out the best subset V*. Subsequently, $round_0$ and $chunk_{0i}$ are computed using the MRRS's initialization procedure, then the chunks of the first round get delivered. The algorithm keeps running until no workload is remaining. The first step of each iteration is to examine the optimality of the workers subset using the OSS method. Next, the 2PP prediction is used to estimate the ES_i for each worker before the start of each round. At last, we use the MRRS scheduling method to compute $round_j$ and $chunk_{ji}$ in light of ES_i.

VI. EVALUATION

A. The 2PP-based Scheduling Algorithm vs. the Static Algorithms

As discussed in Section II, the UMR is deemed to be one of the best static scheduling algorithms. Therefore, we choose to compare the performance of the UMR with the performance of our algorithm. To begin with, and using theoretical proofs, we show in [3,4] that 2PP outperforms UMR in all cases.

Here, we show that 2PP is better than UMR experimentally. For this purpose, we developed a simulator using the SIMGRID [14] toolkit, which has been used for building simulations for various scheduling algorithms, such as UMR and LP, in parallel and distributed environments. We compare the performance of 2PP with UMR using two experimental configurations. Configuration I has the following setup:

- Number of workers: 10
- The average power of a worker: 40 Mflop/s
- Total load (L_{total}): 1 Gflops.
- The average service demand of a local task: 20s.

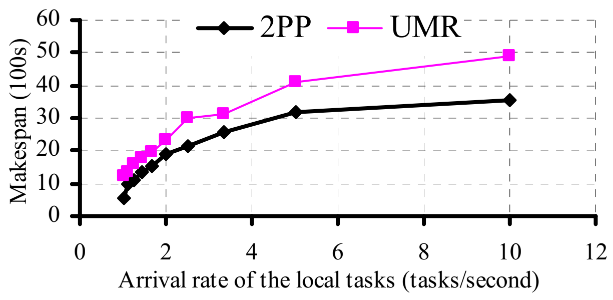

Figure 5. Performance of 2PP vs. UMR under configuration I

One of the chief differences between 2PP and UMR 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 stress test 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 becomes practically the weakest worker with respect to the available CPU power that can be granted to the foreign Grid tasks. Unlike 2PP, 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. Fig. 5 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: 90.
- The average power of a worker: 60 Mflop/s.
- Total load (L_{total}): 2 Gflop.
- The average service demand of a local task: 40s.

Similar to what we did in configuration I, we exposed the top 10% of the workers in configuration II to a higher arrival rate of local tasks. Again, as shown in Fig. 6, 2PP outperforms UMR as the latter is not aware of the run-time availability of the actual CPU power of workers.

B. The 2PP-based Scheduling Algorithm vs. the Dynamic Algorithms

As discussed earlier, the DSA algorithm [8] seems to be the only dynamic scheduling algorithm for divisible loads that we are aware of its existence. Therefore, we

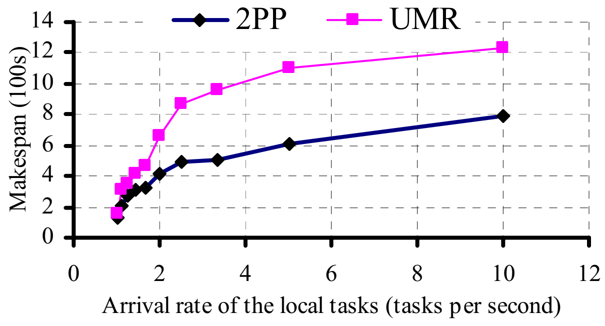

Figure 6. Performance of 2PP vs. UMR under configuration II

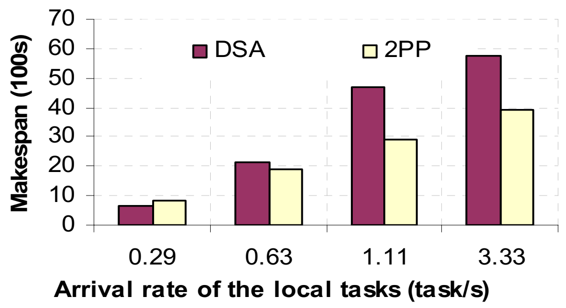

Figure 7. Performance of 2PP vs. DSA

compare the performance of our 2PP-based algorithm with the DSA algorithm under the following experimental configuration:

- Number of workers: 50.
- The average power of a worker: 20 to 60 Mflops/s.
- Task Ratio: Grid task's size/Local task's size (see Table 1).

Fig. 7 contrasts the makespans of the 2PP algorithm vs. the DSA. From these results we can make the following remarks:

- With a low arrival rate of local tasks, DSA is faster than 2PP. However, when the arrival rate exceeds a certain threshold (about 0.5 tasks/s in our experiments), 2PP outperforms DSA.
- The makespan deviation between 2PP and DSA increases proportionally to the increase in the arrival rate of local tasks.

Consequently, we may conjecture that 2PP performs better than other Grid schedulers especially when the local applications at a worker machine compete with the incoming Grid tasks.

TABLE I.
PERFORMANCE OF 2PP VS. DSA

Arrival rate of local task	Makespan of DSA (100s)	Makespan of 2PP (100s)	Grid task's size/local task's size
3.33	57.62	39.13	1.6
1.11	47.13	29.1	1.94
0.63	21.34	18.76	2.38
0.29	6.37	8.21	3.33

VII. CONCLUSION

In this paper, we presented a dynamic scheduling algorithm that is built on top of the static MRRS algorithm after augmenting it with our 2PP strategy for CPU power. We discussed the task execution model that takes into account processing local as well as Grid tasks at workers. We used this model to perform short term forecasting of 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 exhibits suggest that the 2PP-based algorithm is adaptive and more suitable for dynamic, non-dedicated environments such as the Grid.

ACKNOWLEDGEMENT

Our research is supported by the "Fostering Talent in Emergent Research Fields" program sponsored by the Ministry of Education, Culture, Sports, Science and Technology, Japan. This work has been also funded by research grant #02-06-9-11/06 from the Scientific Research Council of the UAE University (UAE), and partially by project #2-002-06 from the Ministry of Science and Technology of Vietnam.

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] T.L. Nguyen, S. Elnaffar, T. Katayama, and T.B. Ho, "MRRS: A More Efficient Algorithm for Scheduling Divisible Loads of Grid Applications", *IEEE/ACM International Conference on Signal-Image Technology and Internet-based Systems (SITIS'06)*, Dec. 2006, Tuynidia.

[4] T.L. Nguyen, S. Elnaffar, T. Katayama, and T.B. Ho," UMR2: A Better and More Realistic Scheduling Algorithm for the Grid, *International Conference on Parallel and Distributed Computing and Systems (PDCS'06)*, Texas, USA, pp. 432-437, ISBN: 0-88986-638-4, 2006.

[5] O. Beaumont, A. Legrand, and Y. Robert, "Scheduling Divisible Workloads on Heterogeneous Platforms", *Parallel Computing*, Sep. 2003, Vol. 9.

[6] 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.

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

[8] T.L. Nguyen, S. Elnaffar, T. Katayama, and H.T. Bao, "A Scheduling Method for Divisible Workload Problem in Grid Environments", *PDCAT'05*, Dec. 2005, Dalian, China.

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

[10] L. Yang, J. Schopf, and I. Foster, "Conservative Scheduling: Using Predicted Variance to Improve Scheduling Decision in Dynamic Environments", *SuperComputing*, Nov. 2003.

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

[12] D. P. Bertsekas, *Constrained Optimization and Lagrange Multiplier Methods*, Belmont, Mass.: Athena Scientific, 1996.

[13] S. Martello and P. Toth, *Knapsack problems : algorithms and computer implementations*, Chichester, West Sussex, England : Wiley, 1990.

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

Dr. Nguyen The Loc received his Ph.D. in 2007 from the Graduate School of Information Science, Japan Advanced Institute of Science and Technology (JAIST). He got his B.S. and M.S. degrees from the Faculty of Information Technology, Ha Noi University of Technology (Ha Noi, Viet Nam) in 1998 and 2001, respectively. Presently, he is an Assistant Professor at the Faculty of Information Technology, Hanoi National University of Education (HNUE) where his research interests focus on Grid Scheduling Problems, Parallel and Distributed Computing.

Dr. Said Elnaffar received his Ph.D. in Computer Science from Queen's University (ON, Canada) in October, 2004. He got his M.Sc. in computer science from Queen's University in 1999. He worked as an Adjunct Assistant Professor in the School of Computing at Queen's University (September-December 2004). Presently, he is an Assistant Professor at the College of Information Technology, UAE University (UAE). His research interests include self-managing systems, Grid systems, and web services. He had several research collaborations with leading industrial corporations such as IBM. He received several awards from different industrial and governmental research agencies.