Applications using Grid computing infrastructure usually require resources allocation to satisfy their Quality of Service (QoS) requirements. Given that the Grid infrastructure is a set of computing resources geographically distributed, the support of Grid applications requires the allocation of computing resources and bandwidth to enable communication among these resources. The objective is to accommodate as many applications as possible while still satisfying their requirements. Ideally, we would like to accommodate a given Grid application using a set of computing resources (e.g., one server) that are not geographically distributed (e.g., in the same LAN); however, this is not always possible. Indeed, to increase the probability of accommodating Grid applications, we may need to use computing resources scattered all over the network; in this case, bandwidth allocation is required to enable communication among these resources. In this paper, we propose an optimization model that enables the “simultaneous” allocation of computing resources and bandwidth for Grid application while maximizing the number of Grid applications being accommodated. A heuristic is proposed to solve the model with an acceptable response time; simulations show that the proposed approach outperforms existing classical approaches.

**Keywords**: Bandwidth, Grid, Quality of Service, Integer programming, Heuristics

I. INTRODUCTION

Grid computing provides a global-scale distributed computing infrastructure for executing scientific and business applications [2, 5]. As Grid computing becomes more popular, the size and the diversity in terms of resources and applications increase as well. In fact, many applications require the allocation of more than one type of resources to execute properly [4, 8, 10, 12].

Considerable research efforts have been dedicated to the allocation of resources to support agreed-upon QoS (Quality of Service) [1, 3]. In [6], the authors present a performance tuning system that applies real-time adaptive control techniques to dynamically adapt to changing application resource demands and system resources availability. The tuning in this context helps avoiding terminations of applications/services due to quality degradation. The authors in [9] present a mechanism supporting open reservations to deal with the dynamic Grid and to provide a practical solution to enforce agreed-upon QoS. In [7], the authors propose an approach that calls for splitting up processes into separated jobs and then balancing them to nodes instead of balancing the load in the Grid by process migration or by moving an entire process to a less loaded node. Load balancing helps shortening the execution time; however, extra resources utilization in a specific period could lead to an increase in the rejection ratio during that period.

The authors in [1] propose a Grid QoS Management framework that includes activities to manage QoS, such as QoS specification, selection and allocation of resources according to QoS requirements, monitoring to keep track of resources availability, etc. In particular, they propose an adaptation algorithm that reserves resources for three types of services (‘guaranteed’, ‘best effort’, and an ‘adaptive’); the algorithm tries to satisfy each new request by adjusting resources reservations between the three types of services (e.g., reducing the amount of resources reserved for a “best service” request to accommodate a “guaranteed” service request). The adjustment helps avoiding the underutilization of Grid resources and maximizing the provider’s revenues.

The authors in [15] propose a model for the resource matching problem and solve it, as an on-line optimization problem, using mixed integer programming methods; they describe how the resource dependencies, capacity requirements and constraints can be modeled. The objective is to maximize throughput, and thus minimize the blocking probability; they use CPLEX (exact solution method) to solve the model. Using an exact method resolution could take an “infinite” response time for realistic large size problems. In [16], the authors present an adaptive resource provisioning scheme that optimizes the resources utilization while satisfying the required QoS. More specifically, it minimizes the request blocking probability and, thus, maximizes the revenues of the infrastructure provider. However, the proposed scheme can be used only for the provisioning of a single type of resources.

In this paper, we propose a provisioning scheme for Grid applications/services that minimizes the blocking probability (and thus increase the provider’s revenues); it enables the “joint” optimal allocation of two types of dependent resources, namely computing resources and bandwidth. Ideally, we would like to accommodate a given Grid application (that consists of a set of processes) using a set of computing resources (e.g., servers) that are not geographically distributed (e.g., in the same LAN); in this case, there is no need for bandwidth allocation. However, this is not always possible. Indeed, to increase the probability of accommodating Grid applications, we may need to use computing resources scattered all over the network; in this case, bandwidth allocation is required to
support communication among the application processes. To support optimal “joint” allocation of computing resources and bandwidth, we propose an optimization model (formulated as an Integer Program) that minimizes the use of bandwidth (thus, it maximizes the revenues of the provider) and maximizes the throughput (thus, it minimizes the blocking probability) at the same time. The optimization problem being NP-hard, a heuristic is then proposed to solve it, thus providing an acceptable response time for realistic large size instances.

The remainder of the paper is organized as follows. Section 2 presents the optimization problem formulation. Section 3 describes the solution method. Section 4 presents simulations and analysis. Section 5 presents the model formulation with timing parameters. Finally, Section 6 concludes the paper.

II. OPTIMIZATION PROBLEM FORMULATION

Integer programming (IP) is a technique for maximizing or minimizing the value of an objective function subject to constraints, where the objective function and constraints are functions defined over a set of decision variables that are restricted to taking integer values. In the following, we propose an IP model for tackling the Grid resource reservation problem discussed in the introduction.

Let us first define the data (inputs) and the variables required by the model before stating its objective and constraints.

Data:
1. \( R \): the set of (provisioning) requests \( r \).
2. \( P' \): the set of processes included in request \( r \).
3. \( P = \bigcup P' \): the set of all processes in the system.
4. \( D \): the set of domains \( s \); a domain consists of a set processors that do not need WAN bandwidth for communication among themselves (e.g., processors in a LAN). Two processors belonging to different domains require bandwidth for communication.
5. \( M_s \): the set of processors belonging to domain \( s \).
6. \( M = \bigcup M_s \) for \( s \in D \).
7. \( E \): the set of edges \( e \) that link adjacent domains.
8. \( L^s_t \): the set of paths \( l \) between domains \( s \) and \( t \).
9. \( L = \bigcup L^s_t \) for \( s, t \in D \).
10. \( L_e \): the set of paths \( l \) to which edge \( e \) belongs.
11. \( w_l \): the cost of using path \( l \).
12. \( b_{ij} \): the bandwidth required when processes \( i \) and \( j \) of request \( r \) are assigned to different domains.
13. \( k^m \): the capacity of processor \( m \in M \).
14. \( k_e \): the capacity of edge \( e \in E \).
15. \( c_i \): the amount of computing capacity required by process \( i \).

Variables:
1. \( z_{ijr}^l = 1 \), if path \( l \) is used for traffic between processes \( i \) and \( j \) of request \( r \);
2. \( x_{ir}^m \): binary variable that indicates whether process \( i \) of request \( r \) is assigned to processor \( m \);
   \( x_{ir}^m = 1 \), if process \( i \) is assigned to \( m \), and 0 otherwise.
3. \( y_{ir}^s \): binary variable that indicates whether process \( i \) of request \( r \) is assigned to domain \( s \);
   \( y_{ir}^s = 1 \), if process \( i \) is assigned to \( s \), and 0 otherwise.
4. \( v_l \): the amount of bandwidth carried by path \( l \).

Objective function:

\[
\text{Min } \sum_{l \in L} W_l \ v_l
\]  

Constraints:
1. \( y_{ir}^s = \sum_{m \in M_r} x_{ir}^m \), \( s \in D, i \in P', r \in R \).
2. \( \sum_{s \in D} y_{ir}^s = 1 \), \( i \in P', r \in R \).
3. \( \sum_{l \in L^s_t} z_{ijr}^l \geq y_{ir}^s + y_{jr}^t - 1 \), \( i, j \in P', r \in R, s, t \in D \).
4. \( v_l = \sum_{r \in R} \sum_{i, j \in P'} b_{ij} \ z_{ijr}^l \), \( l \in L \).
5. \( \sum_{l \in L_e} v_l \leq k_e \), \( e \in E \).
6. \( \sum_{r \in R} \sum_{j \in P'} c_i \ x_{ir}^m \leq k^m \), \( m \in M \).
7. \( z_{ijr}^l \in \{0, 1\} \), \( i, j \in P', r \in R, l \in L \).
8. \( x_{ir}^m \in \{0, 1\} \), \( i \in P', r \in R, m \in M \).
9. \( y_{ir}^s \in \{0, 1\} \), \( i \in P', r \in R, s \in D \).
10. \( 0 \leq v_l \), \( l \in L \).

The objective function (1) represents the cost of bandwidth utilization, which should be minimized when the requests are served. Constraint (2) ensures that each process \( i \) of request \( r \) is assigned to one processor \( m \) of domain \( s \) when it is assigned to this domain. Constraint (3) ensures that each process \( i \) of request \( r \) is assigned to one domain \( s \). Constraint (4) that a path is assigned to connect domains \( s \) and \( t \) on which different processes \( i \) and \( j \) of any given request \( r \) are assigned. Constraint (5) computes the overall bandwidth requirement for paths. Constraint (6) ensures that the capacity of each edge \( e \) is not exceeded. Constraint (7) ensures that the capacity of each processor \( m \) is not exceeded. Constraints (8.a, 8.b, 8.c, 8.d) define the variables of the formulation.

III. PROBLEM SOLUTION

The goal is to have an optimal solution whenever requests arrive. The exact optimal solution of the problem can be computed using appropriate operations research tools (e.g.,
waiting list \textit{WAITLIST}. If the function returns \textit{TRUE}, then two input parameters: the current incomplete solution \textit{S*} and \textit{WAITLIST}:

The first function, named \texttt{complete_initial_solution}, has two input parameters: the current incomplete solution \textit{S} and \textit{WAITLIST}. If the function returns \textit{TRUE}, then an initial solution is found; otherwise, no solution exists.

\begin{verbatim}
Boolean complete_initial_solution(S, WAITLIST)
Begin
   all_inserted=true; /*Boolean variable used to show whether a solution is found or not*/
   While (still processes in WAITLIST and all_inserted=true)  
      { - select the process \textit{ps} with the largest CPU in WAITLIST
        - inserted=false; /*Boolean variable used to know whether the current selected process can be assigned*/
          - j=0; /*integer */
          - while (inserted==false and \textit{j} < number of processors)
            - selected=true; /*the process \textit{ps} is assigned to processor \textit{pr}*/
              - Update \textit{S*};
              - Remove process \textit{ps} from WAITLIST;
            Else
               { \textit{j}++; /*we pass to another processor and try again*/
               }
          if (inserted==false /*the process could not be assigned*/)
            all_inserted=false /*solution is not found */
      If (all_inserted=true)
        { /*initial solution \textit{S*} is found */
          Return (true)}
      Else
        { /*initial solution cannot be found */
          Return (false)
      End complete_initial_solution
\end{verbatim}

\begin{verbatim}
Boolean allocate_ps_in_pr(int ps, int pr)  
Begin
   { a- check whether the requested CPU amount of process \textit{ps} is smaller than the available CPU amount in processor \textit{pr}.
     b- if yes: check if \textit{needed_bandwidth(ps)==true}. /*it checks whether there is enough available bandwidth to assign to support communication with processes already assigned; see below for details*/
       If yes: return (true)
       Else: return (false)
     c- else: \textit{stop==false} /*Boolean variable used to stop the loop*/
          while (still processes in processor \textit{pr} and \textit{stop==false})
            {if (move_one_proccess_from_pr==true) /* tries to move a process from processor \textit{pr} to another one in order to make room for the process we want to assign; see below for details*/
              { stop=true }
            }
            if stop==true : repeat from a;
            else : return (false);
   }
\end{verbatim}
End allocate_ps_in_pr

Boolean move_one_process_from_pr(int pr)
Begin
  { a. Select a process pm, assigned to pr, for moving it;
    b. if the process found
      c. find a processor to where it can be moved /* a
         processor with sufficient available CPU to support
         pm */
      d. if processor found
        (i). Check whether the needed bandwidth can be
            reserved.
        (ii). If yes return (true)
        (iii). else find another processor and repeat from d.
      e. else select another process ‘pm’ and repeat from b
    f. else return (false).
  }
End move_one_process_from_pr

Boolean needed_bandwidth(int ps)
Begin
  { a-determine the bandwidth required to enable communication
      between process ps and its associated processes [p1, ..pk]
      (ps, p1,..., pk belong to the same provisioning request) that
      are assigned to different domains /* ps, p1, ..pk belong to
      the same provisioning request*/
    b-for each pair (ps, pi) where 1<i<=k, select the path, with
      the most available bandwidth, between the domain where
      pi is assigned and the domain where ps is considered for
      assignment.
    c-if for each pair (ps, pi), the selected path can accommodate
      the requested bandwidth between ps and pi, then make the
      corresponding bandwidth reservation and return true.
    d-otherwise, return false.
  }
End needed_bandwidth

Figure 1. Phase 2: pseudo-code

B. IMPROVING THE INITIAL SOLUTION: Third phase

A Tabu Search heuristic is used in this phase to improve
the initial solution found in step 2. First proposed by Glover
[13] in 1986, Tabu Search can be seen as an extension of
classical local search techniques. Its main feature is the use
of memory structures that are used to overcome local optima
and to guide the exploration of the solution space. Cycling is
prevented by forbidding the reversal of previous moves by
making these “tabu” (hence the name of the method).
Information on these tabus is recorded in short-term
memories called the tabu lists (see [14] for more details).

A key ingredient of any local search procedure is the
definition of the modifications that can be performed on the
current solution at any iteration. Since this defines which
solutions are adjacent to the current solution, it is called the
neighborhood structure of the search procedure.

In our problem, since our objective is to minimize the
bandwidth cost we need to move processes from one
domain to another to reduce the amount of bandwidth used
to communicate between processes of the requests. The
neighborhood structure is defined around domains and aims
at exchanging as many processes as possible between the
processors of a pair of domains. For each pair of domains (s,
t) of M, we try to exchange as many processes as possible
between each pair of processors belonging respectively to
those two domains. We repeat this procedure for all domain
pairs (s, t) and select the pair (s*, t*) that yields the greatest
improvement in the objective. The exchange of processes
between the processes of s* and t* is then performed and
leads to the new current solution. This process is repeated
until a (predefined) maximum number of movements is
reached. See the pseudo-code of Figure 2 for more details.

Optimize()
Begin
{int mvt=1, max_mvt /*max movement*/, nb_domain
/*number of domains*/;
int i, j; /* indexes associated to domains*/
int i_save, j_save;/* save domains’ indexes that give less
cost*/
float temp_current_value = total_cost; /*total_cost contains
the cost of the current solution*/
bool improve=false;/* Boolean variable allowing to
know whether the current solution is improved*/
float new_cost;
while (mvt<=max_mvt && improve==false)
{ (i==0; i<nb_domain; i++)
  for (j=i+1;j<nb_domain; j++)
    { new_cost= exchange(i,j,'A'); /*audit procedure call*/
      if (new_cost < total_cost )
        { i_save=i;
          j_save=j;
          temp_current_value=new_cost;
        }
    }
}
if (total_cost > temp_current_value)
{ total_cost=exchange(i_save,j_save,'U');
  mvt++;)
else
  { improve=true;
  }
}
End optimize

float exchange(int i, int j, char mode) /*function that tries
to exchange processes between domains i and j*/
float new_cost=M; // M is a big number
Begin
/*mode='A' : audit mode, mode='U' : update mode*/
int t, p;
for (t=0; t< domains[i].nbr_processors; t++)
{ a. Select processor p from domain j that verifies the 3
   following conditions:
     o Processor[p].used_capacity <=
       Processor[t].total_capacity
     o Processor[t].used_capacity <=
       Processor[p].total_capacity
     o Processor p is not already participated in exchange

**SIMULATIONS AND ANALYSIS**

In this section, we evaluate and compare a number of schemes, via simulations, in terms of (average) cost and rejection ratios for satisfying a set of provisioning requests. More specifically, we define 4 schemes: (1) Classic scheme: allocate first, the requested amount of CPU and then the requested amount of bandwidth for traffic between processes; (2) Greedy scheme: determine a solution using a greedy algorithm (first phase, Section III-A); (3) GI scheme: determine a solution using a greedy algorithm combined with an insertion procedure (Section III-A); and (4) GIT scheme: combination of GI and Tabu Search (Section III-B).

The simulations are performed by varying the number of domains, the number of processors in each domain and the network topology that links domains; we vary also the number of requests submitted in a batch, the number of processes and bandwidth required between processes inside each request. Tables 1-3 show the parameters used in our simulations. In Tables 1-2, we show the parameters of three simulation setups. Table 1 displays the details of the system under simulation; for example, the first setup consists of 4 domains, 16 processors with 70 of capacity each, 30 paths and 6 edges with 35 of capacity each. Table 2 shows, for the first setup, that 30 requests with a total of 90 processes are handled at a time, and the total number of communications between processes is 90 (in average 3 processes per request and each process, in a request, needs to communicate with the other 2 processes). Table 3 shows the range of values of CPU per process, bandwidth between any 2 processes of the same request, and the cost of using a specific path. Specific values are generated using uniform random generator.

**Table 1.** Mesh setups

<table>
<thead>
<tr>
<th>Setup</th>
<th>Domain</th>
<th>Prs</th>
<th>P_cap</th>
<th>Path</th>
<th>Edgs</th>
<th>E_cap</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>16</td>
<td>70</td>
<td>30</td>
<td>6</td>
<td>35</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>32</td>
<td>70</td>
<td>86</td>
<td>9</td>
<td>50</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>64</td>
<td>70</td>
<td>240</td>
<td>19</td>
<td>50</td>
</tr>
</tbody>
</table>

**Table 2.** Request setups

<table>
<thead>
<tr>
<th>Setup</th>
<th>Request</th>
<th>Process</th>
<th>Traffic</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>30</td>
<td>90</td>
<td>90</td>
</tr>
<tr>
<td>2</td>
<td>120</td>
<td>360</td>
<td>360</td>
</tr>
<tr>
<td>3</td>
<td>240</td>
<td>720</td>
<td>720</td>
</tr>
</tbody>
</table>

**Table 3.** Ranges of values

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Range of values</th>
<th>Distribution function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU (1)</td>
<td>[4 .. 20]</td>
<td>C++ uniform random function</td>
</tr>
</tbody>
</table>

We implemented the schemes using C++ running on Linux. For the four schemes, we run the simulations 20 times and we compute the average values observed for:

1. the response time for computing a solution,
2. the cost of the solution,
3. the rejection ratio.

We now discuss the results for each of these measures.

**A. RESPONSE TIME**

Figure 3 shows that all schemes give a good response time, in milliseconds. Even if GIT scheme has the longest response time, it still does not exceed 200 ms in all cases.

**B. UTILISATION COST**

Figure 2 shows that GIT is the least costly scheme for accommodating the set of provisioned requests. More specifically, under GIT, the cost is 18% lower than for scheme 1, 26% lower than for scheme 3, and 16% lower than for scheme 2.

**C. REJECTION RATIOS**

Figure 5 shows that on average 10% of requests are rejected when using scheme 1, 4% are rejected when using scheme 2, and only 2% when using scheme 3 or GIT.

**D. ANALYSIS**

We conclude that GIT is suitable for resource provisioning in Grid applications/services and all other distributed applications that require bandwidth and CPU resources. In the following, we summarize our findings related to the proposed solution.
• The response time is smaller than 1 second (Figure 3).
• Compared to the three other schemes described in section IV, GIT produces the smallest utilization cost while satisfying the requirements of the incoming requests.
• The rejection ratio, which presents a critical factor for users, is the smallest for GIT.

V. PROBLEM WITH TIMING PARAMETERS

Our approach can be easily adapted if we take into account timing parameters: start time and end time for each request (or each process of a request). The goal is to have an optimal solution based on the available resources at a specific time; the resources allocated for the requests already being served are not considered in the optimization. The objective function with the timing parameters can be formulated as below:

\[
\text{Min } \sum_{r \in R} \left( t_{\text{end},r} - t_{\text{start},r} \right) \sum_{l \in L} w_i v_{lr}
\]

Subject to the same constraints as in Section II, where

- \( t_{\text{end},r} \) denotes the ending time of request \( r \);
- \( t_{\text{start},r} \) denotes the starting time of request \( r \);
- \( v_{lr} \) denotes the amount of bandwidth used by request \( r \), on path \( l \), at time \( t_{\text{start},r} \).

The solution heuristic GIT is as in Section 3 but with the introduction of the time parameters in the objective.

VI. CONCLUSION

In this paper, we have proposed an approach that enables the provisioning of bandwidth and CPU resources for Grid applications and services. The basic idea behind our proposal is to allocate as many processes as possible of the same request to the same domain; the goal is to minimize the use of bandwidth and eventually reduce request failures due to network failures.

More specifically, we developed an optimization model using integer programming. Then, we defined a solution heuristic (GIT) to resolve the model with an acceptable response time, minimum cost, and low rejection ratio. The simulation results show that GIT gives a good response time in millisecond and outperforms classical schemes in terms of cost and rejection ratio. With respect to the proposed tabu search algorithm, other neighbourhood structures could be easily implemented, e.g., performing the exchange at the level of processes between two domains instead of exchange at the level of processors between domains.

REFERENCES


Figure 5. GIT Vs. other schemes: Rejection ratio

Percentage (%)

<table>
<thead>
<tr>
<th>Scheme</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheme 1</td>
<td>10</td>
</tr>
<tr>
<td>Scheme 2</td>
<td>8</td>
</tr>
<tr>
<td>Scheme 3</td>
<td>6</td>
</tr>
<tr>
<td>GIT</td>
<td>5</td>
</tr>
</tbody>
</table>
