ABSTRACT OF PhD THESIS

Reactive Scheduling of DAG Applications on Heterogeneous and Dynamic Distributed Computing Systems

Mapeo de Aplicaciones Paralelas tipo DAG en Sistemas Distribuidos Heterogéneos y Dinámicos

Graduated: Jesús Israel Hernández Hernández
Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh, UK.
j.i.hernandez@sms.ed.ac.uk
Graduated in December 4th, 2008

Supervisor: Murray Cole
Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh, UK.
mic@inf.ed.ac.uk

Abstract
Emerging computational platforms enable a set of geographically distributed computers with different capabilities to be linked together and used in a coordinated fashion to solve a parallel application at the same time. Effective scheduling mechanisms are essential to exploit the tremendous potential of computational resources offered by such platforms. We consider the problem of scheduling parallel applications which are often abstracted as directed acyclic graphs (DAGs), in which vertices represent application tasks and edges represent data dependencies between tasks. The core scheduling issues are that the availability and performance of resources, which are already by their nature heterogeneous, can be expected to vary dynamically, even during the course of an execution. This thesis summary presents the main results of the Global Task Positioning (GTP) mapping method, which is based on the cyclic use of a static mapping method over time. We place strong emphasis in three key aspects, which we believe are central to address the dynamic nature of the problem: reactivity, data-aware components and fault tolerance.

Keywords: Parallel processing, heterogeneous computing, task scheduling, DAG scheduling, fault tolerance.

1 Introduction

Shared Heterogeneous Computing Systems (SHCS) are a natural result of the advances in network technology, in which it became possible for geographically distributed computers with different capabilities to efficiently communicate and therefore collaborate in a coordinated fashion to solve a wide range of applications [Foster et al.,
We consider the DAG scheduling problem on SHCS. The core scheduling issues are that the availability and performance of resources, which are already by their nature heterogeneous, can be expected to vary dynamically, even during the course of an execution. Parallel applications can be represented as directed acyclic graphs (DAG), in which vertices represent application tasks and edges represent data dependencies between tasks. The DAG scheduling problem aims to map each task of the DAG onto selected candidate resources in a way which minimizes the resulting schedule length (makespan) while satisfying the task precedence constraints. Since this scheduling problem is NP-complete in its general forms [(Gary et al., 1979), (Papadimitriou and Steiglitz, 1998)], a vast number of heuristics have been proposed in the literature. However, most approaches were designed for homogeneous environments, assuming that the processors have the same capabilities [(Kwok, 1999)]. Some other approaches were designed for particular heterogeneous environments, assuming that heterogeneous resources are dedicated and unchanged over time [(Gerasoulis et al., 1992), (Topcuoglu, 2002), (Shi and Dongarra, 2006), (Sih and Lee, 1993)]. Few algorithms can be found addressing the heterogeneous and changeable nature of SHCS. Such heuristics can be divided into two main categories. One approach proposes to schedule all tasks at run-time, as they become available [(Pegasus, 2003), (Deelman et al., 2003), (Maheswaran and Siegel, 1998)]. The other approach, which we follow, is related with cyclic use (rescheduling) of a mapping method over time [(Hernandez and Cole, 2007a), (Hernandez and Cole, 2007b), (Hernandez and Cole, 2007c), (Zhao and Sakellariou, 2004)]. In this thesis, we propose the Global Task Positioning (GTP), a reactive scheduling system to map DAG applications on SHCS. We place strong emphasis in three key aspects, which we believe are central to address the heterogeneous and dynamic nature of the problem: reactivity, data-aware components and fault tolerance. Thus, the description of our research is divided into three main parts. The first part defines the Global Task Positioning (GTP) scheduling system, a list-scheduling heuristic based model, which addresses the problem by allowing rescheduling and migration of the tasks of an executing application, in response to significant variations in resource characteristics. Our overall system is sketched in Fig.1, in which ITG represents the task graph, STG contains dynamic information concerning the progress of the application, and GRP contains dynamic information concerning the performance of the processors. We will define these structures more formally in Section 2. We consider that an initial task schedule is generated by a standard static mapping method and launched to SHCS. Then, our model considers the cyclic use (rescheduling) of a static mapping method over time. The notion behind this approach is to refine an initial schedule over time, taking into account the most recent performance information of the resources and the progress of the application. The second part, based on observations of previous results for GTP, proposes a new version of the model called GTP/c, in which re-use of information is introduced, to improve the utilization of resources and to minimize the impact of the migration cost on the application makespan. Finally, the third part explores the case of extreme variation of dynamic resources (i.e., processor failure), for situations in which the availability of computational resources cannot be guaranteed. To address this, we propose a rewinding mechanism, an event-driven process executed when a failure is detected at some rescheduling point. The mechanism rewinds the progress of the application to a previous state, thereby preserving the execution despite the failed processor(s). We show how to integrate the rewinding mechanism within GTP and GTP/c producing the new versions GTP/r and GTP/c/r respectively.

The remainder of this article is organized as follows. In the next section, the DAG scheduling problem and some related terminology are defined. In Section 3, we present the GTP system for scheduling DAG applications on SHCS. Then, Section 4 presents the GTP/c system, an extended version of GTP in which re-use of information is introduced, to improve the utilization of the computational resources and to minimize the impact of the migration cost on the application makespan. Section 5 describes the rewinding mechanism to preserve the execution of the application despite the presence of processor failure, resulting in the new versions GTP/r and GTP/c/r, for GTP and GTP/c respectively. The evaluation of GTP, GTP/c, GTP/r and GTP/c/r is conducted by simulation, since this allows us to generate repeatable patterns of resource performance variation. In Section 6 we describe the simulation framework in which we conduct our experiments. In Section 7 we present the assessment of our experimental results. Finally, Section 8 discusses future work and concludes the article.
2 Description of the GTP System

The Global Task Positioning (GTP) system consists of a DAG application, a target Shared Heterogeneous Computing Systems (SHCS) and a reactive heuristic scheduling mechanism to map the tasks composing the DAG application on SHCS. The term \textit{Global} denotes the coordinated collaborative environment of shared resources potentially located at global scale, made possible by advances in network technology.

2.1 Definition of the SHCS

To represent Shared Resource Pools (SRP), we will use graphs $SRP ::= (P, L, \text{avail}, \text{bandwidth})$ where $P$ is the set of available processors in the system, $P \ni \{1 \leq i \leq |P|\}$. $L$ is the set of communication links connecting pairs of distinct processors, $L \ni \{1 \leq i \leq |L|\}$ such that $l(pm, pn) \in L$ denotes an undirected communication link between $pm$ and $pn$. We assume that the intra-processor communication cost ($pm = pn$) is negligible. We assume that the processors are fully connected. Our dynamic scheduling decisions will be based upon the latest available resource performance information (as returned by standard Grid monitoring tools such as NWS\cite{(NWS, 2002)} or Globus MDS\cite{(MDS, 2000)}). Thus, at time $t$ we assume knowledge of $\text{avail}^t : P \rightarrow [0..1]$, capturing the availability of each CPU and $\text{bandwidth}^t : L \rightarrow \text{Float}$ capturing the available bandwidth on each link. We note that the models $GTP$ and $GTP/c$ described in Section 3 and Section 4 respectively, assume $|\text{avail}^t| > 0$ in resources. The rewinding mechanism described in Section 5 addresses the case of extreme variations when the variability is equal to zero ($\text{avail}^t = 0$).

![Fig.1. The GTP System](image)

2.2 Definition of the Input Task Graph (ITG)

Static information about the DAG application (see Fig.1.) is represented by an \textit{input task graph} $ITG :: (V, E, \text{data}, W)$. $V$ is the set of tasks, $V \ni \{1 \leq i \leq |V|\}$. $E \subseteq V \times V$ is the set of directed edges connecting pairs of distinct tasks, $E \ni \{1 \leq i \leq |E|\}$, where $e(v_i, v_j) \in E$ denotes a precedence constraint and data transfer from task $v_i$ to task $v_j$. This implies that $v_j$ cannot start execution until $v_i$ finishes and sends its results to $v_j$. For future convenience, we define the notation

\begin{align*}
\text{GTP} &\quad \text{Rescheduling} \\
\text{GTP/c} &\quad \text{Rescheduling/c} \\
\text{ITG} &\quad \text{Input Task Graph} \\
\text{ITG} &\quad \text{Input Task Graph} \\
\text{SRP} &\quad \text{Shared Resource Pool} \\
\text{STG} &\quad \text{Static Task Graph} \\
\text{SHCS} &\quad \text{Shared Heterogeneous Computing Systems} \\
\text{NWS} &\quad \text{Network Weather Service} \\
\text{MDS} &\quad \text{Metadata Service}
\end{align*}
We use 3.1 Setting Task Ranks intend to adapt this mapping strategy for SHCS. dedicated heterogeneous computing systems). Thus, with the advent of emerging technologies such as SHCS, we scheduling strategy when the computational platform evolves (i.e., from homogeneous computing systems to time). Our interest in the list scheduling approach is the evolvement process observed in the literature for this sequence will be assigned onto that processor which a rank (priority) is assigned to each task and the STG. We assume that information about data transfer sizes and task computation times are provided in standard units, compatible with those of our bandwidth and computational performance measures. We use data :: \( V \times V \to \text{Int} \) to describe the size of data transfers, such that data\((i,j)\) denotes the amount of data to be transferred from \( v_i \) to \( v_j \).

Remembering that our processors are heterogeneous, we represent computation times with \( W :: V \times P \to \text{Int} \), where \( W(i,m) \) denotes the execution time in standard units of task \( v_i \) on processor \( p_m \), when working at full availability (i.e., availability \( 1 \) in terms of function avail).

### 2.3 Definition of the Situated Task Graph (STG)

Just as we maintain dynamic information avail and bandwidth on the SRP, so we must maintain additional dynamic information on the progress of the DAG execution. We model this by augmenting the static ITG, to form a Situated Task Graph STG. This includes information on current schedule of tasks, partial completion of tasks and partial completion of communications. This is necessary, together with monitored information on the availability of processors and links, to allow us to iteratively compute improved schedules, taking into account migration costs and resource availability changes. A key new concept is that of the placed task. A task is said to become placed on a processor once it has begun to gather its input data on that processor. A task which has merely been assigned to some processor by the current schedule is said to be non-placed. The distinction is important because of its impact on migration costs associated with data retransmission. The decision to migrate a non-placed task will incur no migration cost because retransmission of data is not needed. We define \( STG :: (V,E,\text{data},W,\Pi,\kappa',\kappa') \), where the first four components are taken directly from the corresponding ITG. We use \( \Pi :: V \to P^+ \) to represent placement information. \( P^+ \) represents \( P \) augmented with the special value NONE. For placed tasks \( v_i \), \( \Pi(v_i) \) indicates the corresponding processor. For non-placed tasks \( v_i \), \( \Pi(v_i) = \text{NONE} \). A placed task remains placed until migrated or until the whole application terminates, because even after task completion we will later need to retrieve (or re-retrieve in the case of migration) its results. We assume that information concerning the progress of computations and communications is made available by monitoring tools at each rescheduling point. We use \( \kappa :: V \to [0..1] \) to capture the proportion of a task’s computation which has been completed, and similarly, \( \kappa' :: E \to [0..1] \) to capture the proportion of a data transfer which has been completed. The initial \( STG \) is effectively just the \( ITG \) with all completions equal to zero and all task placements set to NONE.

### 3 The GTP Scheduling Method

GTP is based on the list scheduling approach which basically consist of two phases: The task prioritization phase in which a rank (priority) is assigned to each task and the candidate processor selection phase in which each task in the sequence will be assigned onto that processor which optimizes a predefined cost function (i.e., the earliest finish time). Our interest in the list scheduling approach is the evolvement process observed in the literature for this scheduling strategy when the computational platform evolves (i.e., from homogeneous computing systems to dedicated heterogeneous computing systems). Thus, with the advent of emerging technologies such as SHCS, we intend to adapt this mapping strategy for SHCS.

#### 3.1 Setting Task Ranks

We use \( Ru(v_i) \) (also known as blevel), which is an upward rank computed from the exit node to \( v_i \) and defined as the length of the critical path from \( v_i \) to an exit node. \( Ru(v_i) \) is computed recursively as,

\[
Ru(v_i) = W + \max_{v_j \in \text{Succ}(v_i)}(\text{data}(v_i,v_j)+Ru(v_j))
\]  

(1)
where $W_i$ is the average execution cost of task $v_i$ across all processors and it is defined by,

$$W_i = \frac{\sum_{m=1}^{P} W(v_i, p_m)}{|P|}$$  \hspace{1cm} (2)

Notice that the computation weight of a node is approximated by the average of its weights across all processors, following the approach of [(Topcuoglu, 2002)].

### 3.2 Costing of Candidate Schedules

Our cost prediction approach is based upon redefinition of concepts drawn from the literature [(Kwok, 1999; Topcuoglu, 2002)], together with some additional operations required by the dynamic nature of SHCS.

#### 3.2.1 Estimating Communication Cost

During (re)scheduling at time $t$, we need to predict how much time will be required to transfer data for various candidate assignments of tasks to processors. In general, this will depend upon the processors involved and any existing partial completion of the transfers. The estimated communication cost in standard-units to transfer data associated with an edge $e(v_i, v_j)$ from $p_m$ (processor assigned to $v_i$) to $p_n$ (processor assigned to $v_j$) is defined by,

$$C(v_i, p_m, v_j, p_n) = \text{StartUp} + \frac{\text{data}(v_i, v_j)}{\text{bandwidth}(p_m, p_n)}$$  \hspace{1cm} (3)

$\text{StartUp}$ is the system dependent fixed time taken between initiating a request for data and the beginning of the data transfer, and is therefore only applicable to transfers which have not already begun. $\text{data}(v_i, v_j)$ denotes the remaining volume of data to transmit from task $v_i$ to task $v_j$ at time $t$ and is computed as,

$$\text{data}(v_i, v_j) = \text{data}(v_i, v_j) \times (1 - \kappa(v_i, v_j))$$  \hspace{1cm} (4)

#### 3.2.2 Estimating Computation Cost

In estimating the value of candidate schedules we need to predict the time at which some task could begin execution on some processor and the time at which that execution will finish. These times depend upon the availability of the processors (which may have other tasks to complete first) and the availability of input data (which may have to be transferred from other processors). We must first define two mutually referential quantities. $\text{EST}(v_i, p_m)$ is the Estimated Start Time of task $v_i$ on processor $p_m$ where the estimate is made at time $t$. For tasks which have already begun (or even completed) on $p_m$ at $t$, $\text{EST}$ will be $t$ (the effect of already completed work will be allowed for in $\text{EFT}$).

$$\text{EST}(v_i, p_m) = t, \hspace{0.5cm} \text{if} \hspace{0.5cm} \mu(v_i) = p_m \text{ and } k'(v_i) > 0$$  \hspace{1cm} (5)

For other tasks it will be determined by the need for predecessors of $v_i$ to complete and send their data to $p_m$.

$$\text{EST}(v_i, p_m) = \max \{PA'(p_m), DA'(v_i) \}$$  \hspace{1cm} (6)
where \( PA'(p_m) \) is a function which returns the time at which the processor becomes available, having completed other tasks. We notice that our model uses a non-insertion approach to fill the available capacity of processors, therefore the function will return the latest estimated finish time among tasks already assigned to \( p_m \).

\[
PA'(p_m) = \max_{v_i \in \text{Pred}(v_i)} \{EFT(v_i, p_m)\} 
\]  

(7)

Meanwhile, \( DA'(v_i) \) is the estimated earliest time at which data from a predecessor task \( v_j \) (mapped on \( \mu(v_j) \)) will be available at \( p_m \).

\[
DA'(v_i) = \max_{v_j\in\text{Pred}(v_i)} \{EFT(v_j, p_k) + C(v_j, p_k, v_i, p_m)\} 
\]  

(8)

The max block in Equation (8) returns the estimated time of arrival of all data needed to execute task \( v_i \) onto processor \( p_m \). This is calculated by considering the evolving status of each \( v_j \in \text{Pred}(v_i) \). Similarly, \( EFT(v_i, p_m) \) is the Estimated Finished Time of task \( v_i \) on processor \( p_m \). For already completed tasks (at \( t \)) we have

\[
EFT'(v_i, p_m) = t, \text{ if } k'(v_i) = 1 
\]  

(9)

For other tasks it will be determined by the quantity of work outstanding and the availability of \( p_m \).

\[
EFT'(v_i, p_m) = EST'(v_i, p_m) + W'(v_i, p_m) 
\]  

(10)

where \( W'(v_i, p_m) \) denotes the amount of work still to completed for task \( v_i \) on processor \( p_m \), defined by

\[
W'(v_i, p_m) = \frac{W(v_i, p_m) \ast (1-k'(v_i))}{\text{avail}(p_m)} 
\]  

(11)

As with communication cost prediction, migrated tasks must be costed for a restart from scratch (i.e., we reset \( k'(v_i, v_j) = 0 \)). The discrepancy between real and predicted times is incorporated into our rescheduling as a result of the difference between actual completion information (\( k',k'' \)) returned by monitoring, and that which would have been expected at the preceding rescheduling point. Thus, the overall objective of minimising the real makespan of the DAG application is achieved by minimising iteratively the estimated makespan.

### 3.3 The Task Migration Model in GTP

A placed task \( v_i \) is migrated when it has been rescheduled onto a processor other than \( \Pi(v_i) \). In our costings, we adopt a pessimistic model, in which the migrated task must be restarted from the very beginning, including regathering all inputs from its predecessors. This is illustrated in Fig.2a with a hypothetical case. We have that at \( RP_i \), the tasks \( v_1 \) and \( v_2 \) were executed at \( p_1 \) and \( p_3 \) respectively and task \( v_3 \) was scheduled to be executed at \( p_2 \) after receive the data required. However it has so far just received data from \( v_1 \). By considering the current status of both resources and application, the model reschedules the application and \( v_3 \) is migrated from \( p_3 \) to \( p_2 \), expecting to be executed at some point before \( RP_{v_3} \). Thus, data from \( \text{Pred}(v_i) \) must be totally retransmitted to \( p_2 \). At \( RP_{v_3} \) we have the same situation. The requirement was partially fulfilled as just \( v_2 \) successfully transmitted the needed data to \( v_3 \). Again, after updating both the performance of resources and progress of tasks, the model reschedules the application and \( v_3 \) is migrated back to processor \( p_4 \), expecting to be executed before \( RP_{v_3} \). Now, at some point before \( RP_{v_3} \), we observe that \( v_3 \) is finally executed after receiving the required data from \( \text{Pred}(v_3) \). We notice that task \( v_1 \) sent the data twice to the same processor \( p_2 \) as a result of the pessimistic model used by \( GTP \). Obviously, in more complex DAGs, this will tend to increase the overhead cost and the makespan of the application. In Section 4 we consider a more sophisticated method to improve the utilization of resources and minimize the impact of the migration cost on makespan, by exploiting copies of results.
3.4 Time Complexity Analysis for the GTP model

The time complexity analysis is centered in the cyclic use of the static mapping method part which involves two main phases: the computation of task ranks and the costing of candidate schedules. The computation of task ranks traverses the graph upward from the exit nodes which can be done in $O(e + v)$. Then, we have the sorting of the list of tasks by rank (priorities) which takes $O(v \times \log v)$. The costing of candidate schedules which selects a task $v_i$ from the list and maps each task onto that processors offering the minimum earliest finish time, takes $O(e \times p)$ for $e$ edges and $p$ processors for each cycle. For a dense graph when the number of edges $e$ is proportional to $O(v^2)$, the time complexity for the costing of candidate schedule is of the order of $O(v^2 \times p)$. Thus the time complexity for the GTP model for each cycle is of the order of $O(v^2 \times p)$.

4 Description of the GTP/c Model

In GTP, a migrated task had to be restarted from the very beginning, including regathering all inputs directly from its predecessors. Obviously this may negatively affects the makespan of the application. We observed from experiments with GTP that as a consequence of the adaptive nature of the model, some results of some completed tasks transmitted to succeeding tasks, which later on migrate to another processor, can be reused after subsequent migrations as possible sources of its required data. To exploit this observation, we extended the GTP model by adding a Copying Maintenance function, resulting in a new version, the Global Task Positioning with copying facilities (GTP/c) system. The GTP/c considers the maintenance of a collection of reusable copies of the results of completed tasks. This information is maintained in the STG structure which, as before, contains the dynamic information related to the progress of the application.

4.1 Extension of the Situated Task Graph with Copying (STG/c)

We extend the definition of the Situated Task Graph (STG) structure defined in Section 2.3 as $STG/c: (V,E,data,W,\Pi,\kappa,\kappa',\Omega)$, where the first seven components are taken directly from the previous definition of
The key new concept is that of reusable copy. A data transfer for a particular edge \( e(i, j) \) is said to become reusable on a processor once it has been totally transmitted \((\kappa'(e(i, j)) = 1)\) from \( \Pi(v_i) \) to \( \Pi(v_j) \). It is reusable because if during the process, \( v_j \) migrates to a different processor, the copy may be used as source in subsequent scheduling decisions. The copy will remain reusable until \( v_j \) finishes execution. The adaptive nature of our model allows multiple reusable copies for a particular \( e(i, j) \), since task \( v_j \) can migrate at each rescheduling point, if the benefits are substantial. We expect that reusable copies will help to minimize the impact of migration on makespan by avoiding unnecessary data transfer between tasks and exploiting the network links which offers the minimum data transfer cost according to the latest performance resource information. To do this, we need to keep information about every reusable copy generated at time \( t \) in our model. We use \( \Omega' : E \rightarrow P(P) \) to describe the subset of \( P \) where copies of the given (edge) data are available at time \( t \).

### 4.2 Costing of Candidate Schedules

As before, we need to obtain the costing of candidate schedules, now considering that the results (copies) of some tasks can be stored in others sites and reused in subsequent migration of placed tasks.

#### 4.2.1 Estimating the Communication Cost

In the same manner as GTP, during (re)scheduling at time \( t \), we need to predict how much time will be required to transfer data, now considering that the data for a particular edge may have several copies distributed on several sites, for various candidate assignments of tasks to processors. In general, this will depend upon the latest performance information of the link (bandwidth) associated with the processors involved, the location of the reusable copies generated and any previous partial completion of the transfers. We retain equations from Section 3.2.1 for the GTP model, to estimate the communication cost in standard units. The Copying Management (CM) function defined in equation (12), will return the minimum data transfer cost for data associated with \( e(i, j) \) to \( \mu'(v_j) \). Thus, for a particular \( e(i, j) \), CM evaluates the locations (processors) for each reusable copy in \( \Omega'(e(i, j)) \) and together with the latest bandwidth of the links involved, returns the minimum data transfer cost to \( \mu'(v_j) \).

\[
CM(v_i, v_j) = \min_{\mu' \in \Omega'(e(i, j))} \{C(v_i, p, v_j, \mu(v_j))\} \tag{12}
\]

#### 4.2.2 Estimating Computation Cost

We retain definitions from Section 3.2.2 for the GTP model to predict the time at which some task could begin execution on some processor. However, in such prediction we must now include the existent copies which will certainly affect the beginning execution of tasks. Thus, we have redefined the equation (8) such that, now the new equation (13) will compute the estimated earliest time at which data from a predecessor task \( v_j \) (mapped on \( \mu'(v_j) \) and any available copies of their results) will be available at \( p_m \).

\[
DA'(v_i) = \max_{\nu \in \mathcal{P}(v_i)} \{EFT(v_j, p_k) + CM(v_j, v_i)\} \tag{13}
\]

In the same manner, we need to predict the time at which that execution will finish. For this, we retain the equation (9), (10) and (11) for GTP. As before, migrated tasks must be costed for a restart from scratch (i.e., \( k'(v_i, v_j) = 0 \)) and GTP/c ignores possible contention in communication by assuming an infinite number of links from \( p_m \) to \( p_n \).

### 4.3 The Task Migration Model in GTP/c

We recall that GTP uses a pessimistic model, in which the migrated task must be restarted from the very beginning, including regathering all inputs directly from the predecessors (see Fig.2a). Now, with GTP/c we have that in an execution with relatively frequent migration, it may be that, over time, the results of some task have been copied to several other nodes, and so a subsequent migrated task may have several possible sources for each of its inputs. Some of these copies may now be more quickly accessible than the original, due to dynamic variations in...
communication capabilities. The adaptive nature of the GTP/c model is illustrated in Fig. 2b where we can observe the difference of strategies used between GTP and GTP/c. In such figure, we observe that at RP_{1}, task v_3 could not be executed as v_3 only received the required data from task v_1. However, the idea behind the GTP/c model, is that we now maintain the copy of the result generated by v_1 in the system in \( \Omega(e(v_1, v_3)) \), such that it may be used as an input in future migrations for v_3. Thus, we have that at RP_{1} and after considering the latest information about both resources and progress of the application, task v_3 is migrated from p_4 to p_2 and we observe that the required data from v_1 can be transmitted from the site p_4 storing the copy or from the site p_1 where v_1 was executed. The decision to select the site from which the data will be transmitted will depend upon the prediction of the minimum estimated finish time which involves the estimated availability of the processors (which may have other tasks to complete first) and the estimated availability of input data (which may have to be transferred from other processors). Following the example, at RP_{v_3}, v_3 was not computed as it had only received data from v_2. This creates a new copy in the system and is maintained in \( \Omega_{p_4}(e(v_2, v_3)) \) for future migration for v_3. At RP_{v_3} task v_3 is now migrated from p_2 to p_4, and we observe that there are several possible sources for each preceding tasks. At the end we observe that v_3 is finally executed, using the copy \( \Omega_{p_4}(e(v_2, v_3)) \) and a direct data transfer for \( e(v_2, v_3) \).

5 Reliable DAG scheduling with Rewinding and Migration

Fault tolerance is an important issue in SHCS as the availability of shared resources can not be guaranteed [(Medeiros et al.,2003),(In et al.,2005)]. The presence of a resource failure during the DAG execution may disrupt the subsequent execution of some tasks in the DAG. The tasks expected to be disrupted when a processor \( p_m \) fails, can be grouped as a) those tasks \( v_i \) mapped to a processor other than \( p_m \), but still retrieving data from preceding tasks already executed on \( p_m \), and b) those unfinished tasks mapped to \( p_m \) which have begun to gather input data for execution. To address this, we designed the rewinding mechanism, an event-driven process executed when a failure is detected at some checkpoint (see Fig.3). The rewinding mechanism preserves the execution of the application by recomputing and migrating those tasks which will disrupt the forward execution of succeeding tasks. This section describes the rewinding mechanism and shows how to integrate it within our adaptive mapping methods, producing the new versions GTP/r and GTP/c/r respectively. We identify three main steps to consider in the integration of the rewinding mechanism into a particular reactive scheduling approach,

1. The first step is related to the integration of the rewinding mechanism with the data structures containing the information on both the performance of the processors composing the SHCS and the progress of the application (i.e., STG and SRP defined below).
2. The second step is related to the procedure of the rewinding mechanism itself, which will rewind those critical tasks associated with the failed processor which will disrupt the forward execution of succeeding tasks.
3. The last step is related to particular considerations in the dynamic scheduling strategy (i.e., copying, data replication) and deals with resetting the information maintained in the system and linked to the failed processor, to avoid inconsistencies in subsequent scheduling decisions.

The integration and performance of the rewinding mechanism into our scheduling method, is highly dependent upon the details of the scheduling strategies used, encompassing issues such as task assignments, data transfers, migration of tasks, data replication and so on.

5.1 The GTP system with rewinding (GTP/r)

The inclusion of the rewinding mechanism into GTP produces the GTP/r version, which we describe next.

5.1.1 Definition of the SHCS

We will take the same definition and assumptions from the GTP model described in Section 2.1 to represent Shared Resource Pools (SRP). As we defined, at time \( t \) we assume knowledge of \( \text{avail} : P \rightarrow [0,1] \), capturing the availability
of each CPU. We consider that a possible failure in some processor $p_m$ occurs when the latest $avail(p_m) = 0$. Then, at each rescheduling point (RP), if a failure is detected then the rewinding process will be triggered to rewind the application if necessary.

![Diagram of the rewinding mechanism](image)

Fig.3. The rewinding mechanism

5.1.2 Redefinition of the Situated Task Graph (STG)
We extend the definition of the Situated Task Graph (STG) structure defined in Section 2.3 as $STG/r :: (V,E,\text{data},W,\Pi,\kappa_c,\kappa_d,Q)$, where we now consider $Q' :: P \rightarrow P(V)$ to denote the current set of placed tasks mapped on each $p_i \in P$. Recall that a placed task remains placed until migrated or until the whole application terminates, because even after task completion we will later need to retrieve (or re-retrieve in the case of migration) its results. As before, we use $\kappa_c :: V \rightarrow [0..1]$ to capture the proportion of a task’s computation which has been completed, and similarly, $\kappa_d :: E \rightarrow [0..1]$ to capture the proportion of a data transfer which has been completed. A key new concept is that of rewinding a placed task $v_i$ which means that all the current computations and all their inputs and outputs will be initialized, giving the impression of rewinding the application to a previous state. To rewind a task $v_i$ at $t$ we must perform the following operations on $STG$.

1. $\forall v_j \in \text{SUCC}(v_i) \text{ set } \kappa_d(v_i, v_j) \text{ to } 0$
2. $\forall v_k \in \text{PRED}(v_i) \text{ set } \kappa_d(v_k, v_i) \text{ to } 0$
3. Set $\kappa_c(v_i) \text{ to } 0$
4. Set $\Pi(v_i) \text{ to } \text{NONE}$

Thus, rewinding $v_i$ gives the impression of rewinding a portion of the application to a previous state in which nothing has happened and leaving it unplaced once again.
5.1.3 Procedure of the GTP/r model

\( Q'(p_m) = \{v_0, v_1, v_2, ..., v_k\} \) contains the set of \( k \) placed tasks known at time \( t \) to be mapped onto \( p_m \), from which we will rewind those placed tasks which are expected to disrupt the forward execution of succeeding tasks. To do this, we must consider each task in \( v_i \in Q'(p_m) \). Intuitively, \( v_i \) must be rewound if either:

i. it has a successor task which has not yet received a complete copy of the result of \( v_i \), or

ii. it has a successor \( v_j \), which is also assigned to \( p_m \) and which also needs to be rewound.

The recursive form of this rule means that we must consider tasks in \( Q' \) in an order which respects a reverse topological sort (according to the precedence constraints between tasks). Thus, within \( Q'(p_m) \) we must consider exit tasks first, then their predecessors, and so on. This ordering is straightforward to maintain in an implementation because all precedence information is available. Thus, a task \( v_i \in Q'(p_m) \) must be rewound if,

1. \( \exists e(v_i, v_j) \in E : \kappa^e(v_i, v_j) < 1, \) or
2. \( \exists v_k \in SUC(v_i) : v_k \in Q(p_m) \) and \( v_k \) must be rewound.

Note the importance of maintaining information about all placed tasks in \( Q' \), including those whose completion is complete. Following the procedure, we now know that no information related to the failed processor \( p_m \) is maintained in GTP/r. Obviously, after the rewinding process, the failed processor will not be considered in the subsequent scheduling decisions, unless \( avail(p_m) > 0 \) at future \( RP' \)'s. To illustrate the rewinding mechanism for GTP, we extend the example of Fig. 2a by adding a failure in processor \( p_3 \) before finishing the execution of the DAG application at some point between \( RP_{p_3}^{+1} \) and \( RP_{p_3}^{+2} \) as shown in Fig. 3(a). We observe that the failure in \( p_3 \) will inhibit the precedence constraint satisfaction for \( e(v_2, v_3) \) as \( v_3 \) will stop retrieving the input required from \( v_2 \) to start execution. Then, the failure will be detected at \( RP_{p_2}^{+2} \) and therefore the rewinding mechanism will be triggered at this point. The rewinding mechanism must determine which placed tasks mapped to \( p_3 \) need to rewind to preserve the execution of the DAG application. At \( RP_{p_2}^{+2}, Q_{p_2}^{+2}(p_1) = \{v_1\}, Q_{p_2}^{+2}(p_2) = \{v_0, v_2\} \) and \( Q_{p_2}^{+2}(p_3) = \{v_3\} \). Then, the rewinding mechanism will evaluate in reverse order the sequence of each placed task \( v_i \in Q_{p_2}^{+2}(p_i) \). Thus, the first task to evaluate is \( v_2 \) which as we observe inhibit the precedence constraint satisfaction for \( e(v_2, v_3) \), as \( v_3 \) will stop retrieving input from \( v_3 \) executed on \( p_3 \). Then, \( v_2 \) is rewound as explained above. Now, the next task to evaluate from \( Q_{p_2}^{+2}(p_2) \) is \( v_0 \), which \( Succ(v_0) = \{v_1, v_2\} \), then for the first precedence constraint \( e(v_0, v_1) \) is satisfied as \( v_1 \) has finished its execution at \( p_1 \). However, when evaluating the second precedence constraint \( e(v_0, v_2) \) we observe that it is not satisfied as \( v_2 \) (already rewound) will not be able to retrieve their input from \( v_3 \) executed on \( p_3 \). Thus, task \( v_0 \) must also be rewound. Since, tasks \( v_0 \) and \( v_2 \) were rewound, they will be ready to be rescheduled and migrated to a different available processor, guaranteeing the data transfer of the remaining tasks and preserving the forward execution of the DAG application. Obviously the processor \( p_3 \) will not be considered for scheduling decisions. Following the steps for the rewinding mechanism, there is no additional information linked to \( p_3 \) which could lead to inconsistencies in scheduling decisions. After rewinding and rescheduling the application at \( RP_{p_2}^{+2} \), the task \( v_3 \) was finally executed at \( p_3 \) after receiving the required inputs.

5.2 The GTP/c System with Rewinding (GTP/c/r)

In the same manner we will follow the three steps defined to integrate the rewinding mechanism into the GTP/c system resulting in the GTP/c/r version.

5.2.1 Redefinition of the Situated Task Graph with Copying (STG/c)

We extend the definition of the Situated Task Graph with copying (STG/c) structure defined in Section 4.1 as \( STG/c/r :: (V,E,\text{data},W,P,\kappa,\kappa',\Omega, Q) \). In particular we remember \( \Omega::E \rightarrow \mathcal{P}(P) \) to capture information on location of
copies which can be used as source. In the same manner as GTP/r, we use $Q'$ to capture information on tasks placed on each processor.

5.2.2 Procedure of the GTP/c/r model

The placed tasks at time $t$, $v_i \in Q'(p_m)$, are evaluated in reverse topological order. The first criterion to select those tasks to be rewound is the same as GTP/r, which states that a placed task $v_i$ mapped to $p_m$ will be rewound if there exists at least a data transfer $e(v_i, v_j) \in E$ such that it is partially transmitted $k^i(v_i, v_j) < 1$. However, now we have a second criterion to be met related to the existence of possible reusable copies for a particular edge $e(v_i, v_j) \in E$, such that if there exist at least one reusable copy in a processor different than $p_m$, then it means that $v_j$ can retrieve the data from its copy despite $p_m$, and therefore rewinding is not needed. This particular feature of GTP/c is expected to reduce the overhead cost generated by the rewinding mechanism. More formally, for GTP/c/r, a task $v_i \in Q'(p_m)$ must be rewound if,

1. $\Omega(v_i) = \{p_m\}$, (this is the only copy), and either
2. $\exists (v_i, v_j) \in E : k^i(v_i, v_j) \leq 1$, or
3. $\exists v_k \in Succ(v_i) : v_k \in Q'(p_m)$ and $v_k$ must be rewound.

As before, for tasks to be rewound, we must reset elements of $k^i$, $k^c$ and $\Pi$ to reflect the rewinding. For GTP/c/r, all the copies located at the failed processor $p_m$ and maintained in STG can lead to scheduling thrashing if they are not eliminated. Thus, and following with the procedure, those copies $\Omega_e(e_{i,j}) = p_m$ must be eliminated from STG. To illustrate GTP/c/r, we will use the same case as for GTP/r with the same failure in processor $p_3$ at some point between $RP_{i+1}$ and $RP_{i+2}$. This is shown in Fig.3(b). At $RP_{i+2}$, $Q_{i+2}(p_1) = \{v_1\}$, $Q_{i+2}(p_3) = \{v_0, v_2\}$ and $Q_{i+2}(p_4) = \{v_3\}$. Then, the rewinding mechanism will evaluate in reverse topological order the sequence of each placed task $v_1 \in Q_{i+2}(p_3)$. Thus, the first task to evaluate is $v_2$ which, as we observe, inhibits the precedence constraint satisfaction for $e(v_2, v_3)$, as $v_3$ will stop retrieving input from $v_2$ executed on $p_3$. However, due to the maintenance of reusable copies for GTP/c/r, the input required by $v_3$ from $v_2$ can be retrieved from the copy generated before $RP_{i+1}$ and stored at $p_2$, satisfying the precedence constraint. Then, rewinding task $v_2$ is not needed. The next task to be evaluated is $v_0$ with $Succ(v_0) = \{v_1, v_3\}$. The first precedence constraint for $e(v_0, v_1)$ is satisfied as $v_1$ has finished execution at $p_1$. The next precedence constraint for $e(v_0, v_2)$ is considered as satisfied as $v_2$ kept its status of finished task, because it was not rewound. Thus task $v_0$ will not be rewound. Finally, since GTP/c/r maintains a collection of reusable copies some of which may be stored at $p_3$, we need to reset those copies stored at $p_3$ which could lead to inconsistency in future decisions. In this case, the copy $\Omega_{i+2}(e(v_2, v_3))$ stored at $p_3$ must be deleted from the system as it can lead to inconsistencies in the scheduling decisions in the case that task $v_0$ be migrated in the future. Thus, after the third step, the application has been rewound and its execution has been preserved despite failure of $p_3$ at $RP_{i+2}$. Completing the example, after rewinding and rescheduling the application at $RP_{i+2}$, $v_3$ was finally executed at $p_4$ after receiving the required inputs.

6 Simulation Framework

We have selected the well known HEFT [Topcuoglu, 2002] and DLS/sr against which to evaluate the performance of GTP and GTP/c. In [Zhao and Sakellariou, 2004], a selective rescheduling policy is proposed to reduce the frequency of rescheduling attempts. We use this approach to build an adaptive version (DLS/sr) of the well known Dynamic Level Scheduling (DLS)[Sih and Lee, 1993] algorithm. For our purposes, we will use the spare time between tasks as selective rescheduling policy, which denotes the maximal time that for a particular edge $e(v_i, v_j) \in
Reactive Scheduling of DAG Applications on Heterogeneous and Dynamic Distributed Computing Systems

E, task \( v_i \) can execute without affecting the start time of their successor \( v_j \). It also includes the adjacent task of \( v_i \) in the execution order of the assigned processor. Our evaluation is conducted by simulation, since this allows us to generate repeatable patterns of resource performance variation. We have used an extended version of the Simgrid simulator ([Simgrid, 2001]) for this purpose.

6.1 The Directed Acyclic Graphs (DAG)

The shape of the graphs considered in our experiments were taken from the Standard Task Graph Project [STG, 2000]. The graph size (in number of tasks) varied in the range \( \{50, 100, 300, 500, 1000\} \). For each size of DAG, we generated three different graphs with different Communication to Computation Ratio (CCR) characteristics, to test the mapping methods. The DAG’s CCR is defined as the average of all its communication costs divided by the average of all its computation costs. Thus, for each size of the task graph, we generated three different graphs for CCR equal to 0.1, 0.5 and 1.5.

6.2 The Scheduling Scenarios

We created a number of test scenarios to evaluate the performance of GTP, GTP/c, GTP/r and GTP/c/r. A scenario involves a sequence of randomly defined (but repeatable) events, each simulating a resource change in either processor or bandwidth availability. Our scenarios are distinguished by the bound placed on the maximum variation allowed in one event, expressed as a percentage of the peak performance of a resource. For example, an scenario with a bound of 30%, any one event can cause the availability of a processor to decrease to no less than 70% of its peak performance, or of a link to decrease to no less than 70% of its maximum bandwidth. We experimented with a bound ranging from 0% to 90% in increments of 10%. Our graphs embrace the whole spectrum of bounds.

6.3 Comparison Metrics for GTP and GTP/c

We use the Normalized schedule length (NSL) to compare the performance of GTP with that of GTP/c and DLS/sr. The NSL metric is defined as the ratio of the schedule length (makespan) to the sum of the computational weights along the critical path and can be computed as,

\[
NSL = \frac{\text{Makespan}}{\sum_{i \in \text{Critical Path}} W_i}
\]  

Other metrics to be used to understand the behaviour of each model are the average number of migrated tasks, the average number of remappings and the average overhead cost.

6.4 Comparison Metrics for GTP/r and GTP/c/r

In the same manner, we use the NSL metric defined in equation (14) to evaluate the performance of the rewinding mechanism integrated into the GTP (GTP/r) and GTP/c (GTP/c/r) models. Aiming to understand the behavior of such mechanisms, we will use two complementary metrics: The Rewound Tasks (RT) metric, which counts the number of placed tasks rewound to preserve the execution of the application and the Rewound Levels (RL) metric, which determines how deep the application had to be rewound after processor failure.

7 Experimental Results

In this section we show and evaluate the performance results for GTP, GTP/c, GTP/r and GTP/c/r.

7.1 Performance Evaluation of GTP and GTP/c

We report here only the results for SCE10-300 (10 processors and 300 tasks) with CCR=0.5. The case in which scenarios include 0% of variability allows us to investigate the extent which emerging discrepancies between real and predicted behavior are handled by GTP and GTP/c. Thus, for this particular scenario, we observe in the graphic showing the average NSL (Fig. 4) that as resource variability increases, increases the discrepancies between the
predicted and real estimations, being the performance of the static model HEFT more negatively affected than the reactive models. This means that GTP, GTP/c and DLS/sr, at each RP, reacted to inaccurate estimation in the previous schedule and obtained a refined schedule considering the progress of the application on unchanging environments, which increased the performance of the application compared with HEFT.

For more SHCS-like scenarios, the experimental results show that GTP/c outperforms HEFT, GTP and DLS/sr in most cases. Exceptions are limited to the use of DAGs with few tasks (mainly 50 and 100 tasks) and low variability in computational resources. GTP/c has a better performance particularly when the application becomes larger and complex (i.e., 300, 500 and 1000 tasks). This can be observed in the results for SCE10-300 (see Fig.4), where the average NSL for GTP/c when the variability is 40%, outperforms HEFT by up to 14% and as shown, it tends to considerably increase the performance as the variability increases, due to the static nature of HEFT. GTP/c outperforms GTP by up to 3% and the performance of DLS/sr by up to 7%, with increasing improvement as variability increases. We believe that this is because, the number of copies will tend to increase, and the migrated tasks will have several possible sources to retrieve the information. Thus, some reusable copies will reduce the impact of migration on makespan by avoiding unnecessary data transfer between tasks, and by exploiting the network link which offers the minimum data transfer cost according to the latest performance resource information. A natural consequence is that the number of remappings will decrease, decreasing the migrated tasks, decreasing the overhead cost and finally decreasing the makespan of the application. This can be observed in the graphs showing the average number of remappings, migrated tasks and overhead cost in Fig.4.

![Fig.4. Performance results of GTP and GTP/c](image)

7.2 Performance Evaluation of GTP/r and GTP/c/r.
The experimental results show that for most cases the performance of the rewinding mechanism for GTP/c/r outperforms GTP/r in the presence of a processor failure. This can be observed in the results for SCEr10-300 (see Fig.5), where the average NSL for GTP/c/r when the variability is 20%, outperforms GTP/r by 5% and as shown, the performance of GTP/c/r tends to considerably increase as the variability increases. Now, from the complementary
metrics, we observe in Fig. 5 that GTP/r will need up to 3% more levels rewound than GTP/c/r, and the number of tasks to be recomputed is up to 4% more than GTP/c/r, generating 3% more rewinding overhead. From this we learn that when processors fail, the strategy of using reusable copies in the GTP/c/r model, may help some remaining tasks still retrieving data from the failed processor, to retrieve data from other sites.

Fig.5. Performance results of GTP/r and GTP/c/r

8 Summary

This research work explored the DAG scheduling problem on SHCS. The core issues are that the availability and performance of resources, which are already by their nature heterogeneous, can be expected to vary dynamically, even during the course of an execution. We placed strong emphasis in three key aspects which we believe are central to address the dynamic nature of the problem: reactivity, data-aware components and fault tolerance. Thus, we presented the GTP, GTP/c, GTP/r and GTP/c/r scheduling methods. Experimental results showed that GTP/c outperformed GTP, HEFT and DLS/sr; and GTP/c/r outperformed GTP/r. Since we believe that new classes of complex DAG applications will emerge to exploit the vast number of resources offered by SHCS, our future work goes in the directions of developing new scheduling strategies to effectively address the dynamic nature of emerging global computational platforms.

References

Israel Hernandez obtained his Ph.D. degree in Informatics in 2008 from the School of Informatics, University of Edinburgh, UK and his M.Sc. in Computer Science in 1996 from Monterrey Institute of Technology (ITESM), Campus Monterrey. He has been involved in IT projects with companies such as Nissan Mexicana, Vitro Corporation and others. Since 2008, he is a professor/researcher at Polytchnic University of Victoria in Mexico. His research interests include parallel processing, heterogeneous computing, task scheduling, DAG scheduling, reactive scheduling, grid and cloud computing and fault tolerance.

Murray Cole is a member of the Institute for Computing Systems Architecture at Edinburgh University, UK, with an interest in parallel programming models, emphasising approaches which exploit “skeletons” to package well known patterns of computation and interaction as customisable frameworks. At present his efforts focus on the eSkel and Enhance projects, which investigate these ideas in the contexts of single machine parallelism and Grid computing respectively.