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

Jesus Israel Hernandez

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2008

Abstract

Emerging technologies enable a set of distributed resources across a network to be linked together and used in a coordinated fashion to solve a particular parallel application at the same time. Such applications are often abstracted as directed acyclic graphs (DAGs), in which vertices represent application tasks and edges represent data dependencies between tasks. Effective scheduling mechanisms for DAG applications are essential to exploit the tremendous potential of computational resources. 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. In this thesis, we first consider the problem of scheduling DAG task graphs onto heterogeneous resources with changeable capabilities. We propose a list-scheduling heuristic approach, the Global Task Positioning *GTP* scheduling method, which addresses the problem by allowing rescheduling and migration of tasks in response to significant variations in resource characteristics. We observed from experiments with *GTP* 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 sites, 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. To exploit this observation, we extended our model with a *Copying Management* (CM) function, resulting in a new version, the Global Task Positioning with copying facilities (GTP/c) system. The idea is to reuse such copies, in subsequent migration of placed tasks, in order to reduce the impact of migration cost on makespan. Finally, we believe that fault tolerance is an important issue in heterogeneous and dynamic computational environments as the availability of resources cannot be guaranteed. To address the problem of processor failure, we propose a rewinding mechanism which rewinds the progress of the application to a previous state, thereby preserving the execution in spite of the failed processor(s). We evaluate our mechanisms through simulation, since this allow us to generate repeatable patterns of resource performance variation. We use a standard benchmark set of DAGs, comparing performance against that of competing algorithms from the scheduling literature.

Acknowledgements

First I would like to express my deep gratitude to my supervisor Dr. Murray Cole. His patience, always appropriate advice and support, often beyond duty, have been invaluable to shape my research skills. I would like to express my appreciation to Dr. Jane Hillston and Dr. Mike O'Boyle for providing valuable comments and suggestions on drafts of this research; and to the members of the Institute for Computing and Systems Architecture (ICSA) for the vibrant research environment.

I would like to give particular thanks to Dr. Rizos Sakellarious and Dr. Marcelo Cintra for their valuable advice and discussion about the details of the research.

My sincerest thanks go to all the members of my family for their unconditional support and encouragement during every stage of my life. In the same manner, I thank my mother and brothers in law for their support and prayers.

I would like to thanks the fellowship of Duncan Street Baptist Church for all their love and support to my wife and to me, especially to Jaqui, Ruth, Harry and Brenda, Frank and Peggy, Ira and Isobel, David and Esther, Robert and Ruth, Tom, Allan, Evelyn, Mike, John and Betty. I thank the fellows from the Ramblers Group, especially to the leaders, Jim, Douglas and Shirley, for the opportunity to enjoy the amazing Scottish landscapes. God bless you all. Finally, I gracefully acknowledge the moral and financial support of CONACYT.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Jesus Israel Hernandez*)

I dedicate this thesis to my two women: *La Miss Gloria*, my mother and *Liz*, my wife. My deepest thank for your love, patience, prayers and care, which made this endeavor a reality.

Table of Contents

List of Figures

Chapter 1

Introduction

Emerging computational platforms enable a set of heterogeneous and non-dedicated resources distributed across a network to be linked together and used in a coordinated fashion to solve a particular problem at the same time. We consider the problem of scheduling parallel applications, represented by directed acyclic graphs (DAGs), onto heterogeneous and shared computational resources, in a way that minimises the resulting schedule length (makespan) of the application. The core issues are that the availability and performance of the resources, which are already by their nature heterogeneous, can be expected to vary *dynamically*, even during the course of an execution. This thesis is motivated by the fact that the DAG scheduling problem is NP-complete in its general forms. A vast number of heuristics have been proposed in the literature. However, most of the heuristics were designed for homogeneous environments composed by processors with the same computational capabilities. Some heuristics were designed for heterogeneous environments composed by processors with different computational capabilities, but assuming that such capabilities are dedicated and unchanging over time. New efforts are required to develop scheduling mechanisms to address the heterogeneous and dynamic nature of emerging global computational platforms.

1.1 Contribution

In this research, we place strong emphasis in four key aspects, which we believe are central when designing mapping methods for heterogeneous and dynamic distributed computing systems: reactivity, data-aware components, data transfer flow and fault tolerance. The contributions of this work are summarized as follows:

1. We propose a list-scheduling heuristic approach, the Global Task Positioning *GTP* mapping method, which addresses the DAG scheduling problem for heterogeneous and dynamic computational environments with a cyclic use of a static mapping method. The term *Global* denotes the coordinated collaborative environment of resources located potentially at global scale, made possible by advances in network technology. Our method allows rescheduling and migration of tasks when this helps to minimize makespan. Thus, at each rescheduling point the objective is to obtain an improved task schedule which minimises the anticipated makespan, considering the current status of both application and computational resources.

2. We observed from experiments with GTP that due to their task migration policy, the results of some tasks may have been copied to several other sites, 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. To exploit this observation, we extended the GTP model by including a Copying Management function, resulting in the GTP/c model. We demonstrate that reusing such copies will help to reduce the impact of migration on makespan by avoiding unnecessary data transfer between tasks.

3. The relationship between the DAG application (defined by the owner of the DAG) and the mapping method (defined by the owner of the method) is not fully explored. Most mapping methods focus on scheduling strategies which use the shape and static information of the DAG, just as a reference to map tasks onto processors. They do not consider the mechanism through which communication of task results is actually achieved. We have found that ignoring this issue may negatively affect the performance of the application. We observed two main models to allow the transfer of data among tasks, the *PUSH model* and the *PULL model*. In the PUSH model as soon as a task finishes execution, the data results are pushed to its successors for execution. In the PULL model, the data results are pulled from predecessors as soon as a task is mapped on a particular processor. We conducted some experiments in which we show that the final makespan of the application can be affected depending on the data transfer model used to execute the DAG application.

4. Fault tolerance is an important issue in computational environments where re-

sources are heterogeneous, non-dedicated and distributed, as the availability of processors cannot be guaranteed. Effective DAG scheduling methods must include fault tolerant mechanisms to preserve the execution of the application, despite the presence of a processor failure. To address this, we propose a rewinding mechanism, an event-driven process executed when a failure is detected at some rescheduling point. The rewinding mechanism seeks to preserve the execution of the application by recomputing and migrating those tasks which will disrupt the forward execution of succeeding tasks. The mechanism rewinds the progress of the application to a previous state, thereby preserving the execution despite the failed processor(s).

1.2 Outline of the dissertation

This dissertation is structured as follows. Chapter 2 provides a review of relevant literature about the DAG scheduling problem. We describe the elements and evolution of the DAG scheduling problem from homogeneous environments to emerging global computational environments composed by heterogeneous and non-dedicated computational resources. In Chapter 3, we present the GTP reactive method for scheduling DAG applications on heterogeneous resources with changeable capabilities over time. Then, we present the GTP/c reactive method, 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. Finally, we propose the rewinding mechanism to preserve the execution of the application despite the presence of processor failure, increasing the reliability of our dynamic scheduling methods GTP and GTP/c. The evaluation of our reactive mapping methods is conducted by simulation, since this allows us to generate repeatable patterns of resource performance variation. In Chapter 4 we describe all the elements contained in the simulation framework in which we conduct our experiments. We describe the source and characteristics of the input task graphs used in the evaluation. Then, we describe the distinctive characteristics of our scenarios under which the mapping methods are evaluated. At the end of this chapter, we describe the adaptive version of the Simgrid software, which allows us to manage dynamic events in simulating variations in the performance of resources. In Chapter 5 we present the assessment of our experimental results. By using defined metrics, we use the $HEFT$ static method and DLS/sr, an adaptive version of the dynamic level scheduling (DLS) static method for heterogeneous and dynamic com-

putational environments, to benchmark and evaluate the performance of our proposed scheduling methods *GTP* and *GTP/c*. In the same manner, we include the rewinding mechanism into our scheduling methods *GTP* and GTP/c, and based on defined metrics, we evaluate their performance. Finally, Chapter 6 discusses future work and concludes the dissertation.

Chapter 2

Review of Literature

Grid computing is an emerging technology distinguished by integrating large-scale, geographically distributed and heterogeneous computational resources with different administrative domains. Although this is a relatively simple concept, achieving it has been a major challenge in Computer Science. Our research work has been inspired by Grid systems [(Schopf, 2004), (Foster et al., 2001), (Foster et al., 2002), (Foster et al., 2003b), (Foster et al., 2003a)]. However, it should be noted that there are many practical obstacles which would make it difficult to apply our work directly to real, current Grid systems, including the independence of local schedulers, the problems (or even impossibility) of trying to arrange co-scheduling across domains and other administrative matters. Thus, while we hope that our more abstract results may be of interest to Grid-like systems of the future, we expect that they may be of more immediate interest to the more open, heterogeneous distributed systems available within organizations and their related domains.

Our research work focuses on the scheduling mechanisms to address the DAG scheduling problem on heterogeneous and dynamic distributed computing systems. 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. Since this scheduling problem is NP-complete in its general forms [(Gary and Johnson, 1979), (Papadimitriou and Steiglitz, 1998)], a 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 and Ahmad, 1999a), (McCreary et al., 1994)]. Some other approaches were designed for particular heterogeneous environments, assuming that heterogeneous resources are dedicated and unchanged over time [(Topcuoglu, 2002), (Shi and Dongarra,

2006), (Sih and Lee, 1993)]. In this chapter we make a literature review embracing the different elements involving the DAG scheduling problem. We start this chapter by giving a brief overview of heterogeneous computing. Next we describe the task scheduling process on heterogeneous and dynamic distributed computing systems, followed by the different scheduling architectures observed in the literature. Then, we describe the different parallel application classes that we can find in the literature and we continue by reviewing a pair of issues needed to address the DAG scheduling problem on heterogeneous resources with changeable capabilities, the task mapping strategies and the mapping method operation mode. We continue by exploring particular issues related to dynamic heterogeneous environments, such as fault tolerance and data-aware scheduling, to introduce part of our research work. We finish this chapter by describing some simulation toolkits from the literature, to build and test mapping methods.

2.1 Heterogeneous Computing

Initially, homogeneous computational environments (i.e., parallel computers) were distinguished by the capability to execute multiple tasks in parallel on dedicated processors with the same capabilities connected by local interconnection networks. This allowed the creation of problem partitioning techniques to solve large problems, which usually did not fit on a single processor or could not be solved in a reasonable time. We will refer to such partitioned problems as parallel applications. The objective of parallel computing was to exploit the parallelism of the computational platform to execute parallel applications, which could help to solve large problems in reasonable time. To achieve this, two major problems had to be addressed: the first concerns partitioning the problem into a set of smaller tasks and the second concerns the mechanism used to schedule the tasks onto processors [(Bokhari, 1981), (Girkar and Polychronopoulos, 1987)].

Heterogeneous computing is a natural result of the advances in network technology, in which it became possible for distributed computers with different capabilities to efficiently communicate and therefore collaborate to solve parallel applications at the same time. Heterogeneous computing can be seen as a special form of parallel and distributed computing. Parallel computing on homogeneous environments is distinguished by containing all the elements required by the parallel application at a single machine or parallel computer. In heterogenous computing platforms, the elements required by the parallel application are dispersed among distributed resources. Un-

like other distributed applications, heterogeneous computing requires, in a coordinated fashion, direct access to the main components (i.e., cpu, data, memory, etc.) of the computational resources to solve the parallel application.

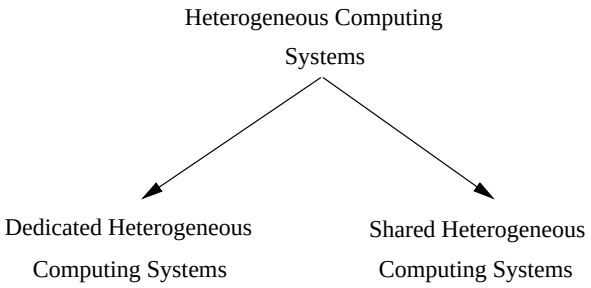


Figure 2.1: Heterogeneous Computing Systems

As shown in Figure 2.1, we identify two classes of Heterogeneous Computing Systems (HCS): dedicated heterogeneous computing systems (DHCS) and shared heterogeneous computing systems (SHCS). DHCS describes heterogeneous systems where resources are tightly controlled, and may therefore be dedicated to a particular application. Parallel machines, clusters or networks of workstations with the ability to provide dedicated, exclusive scheduling illustrate this class. SHCS describes heterogeneous systems in which there is weaker, less restrictive or less coordinated control and resources may therefore be shared with other unknown applications. For example, networks of workstations without access control, or alternatively, collections of more widely distributed systems with their own globally inaccessible local scheduling policies fall into this category. We focus on the challenge of scheduling DAG applications on SHCS systems, dealing with the definition and development of mapping mechanisms which must consider the resulting dynamic nature of the heterogeneous resources.

2.2 Scheduling DAGs on SHCS

A key challenge in SHCS, is to define the scheduling mechanisms that enable a set of heterogeneous resources with changeable capabilities across a network to be linked together and used in a coordinated fashion to solve a particular parallel application. Due to the dynamically shared nature of SHCS resources, Figure 2.2 distinguishes two classes of scheduling approaches: an *application level* scheduling and a *resource level* scheduling. *Application level* scheduling is also known in the literature as global

scheduling [(Casavant and Kuhl, 1988)] or meta-scheduling [(GridWay, 2002)]. *Resource level* scheduling is referred to as local scheduling [(Casavant and Kuhl, 1988)]. Our research work concerns an application level scheduling and we will refer to it during this work as global scheduling.

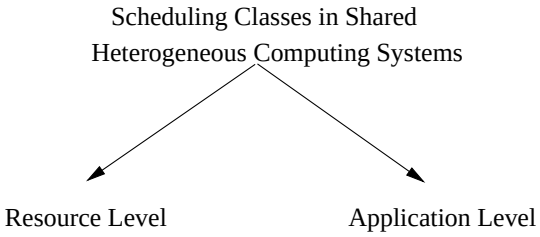


Figure 2.2: Scheduling Classes in SHCS

Global scheduling can be defined as the process of making scheduling decisions over distributed resources with different and changeable capabilities over time. There are a pair of challenges that global scheduling mechanisms must face: *heterogeneity* of resources, which results in different capabilities for task processing and the *dynamic nature* of resources which may vary their performance (i.e., availability and bandwidth) over time, even during the execution of a particular application. Variations in the availability of processors may come from the autonomy of processors to follow local policies and competition by other applications for resources. Variations in network bandwidth may come from the traffic on network links. Thus, effective mapping methods for SHCS must include mechanisms to address the dynamic nature of resources and local schedulers.
The global scheduling process consists of several steps over time. The first three steps are required to map tasks onto heterogeneous resources. The fourth step concerns a time dimension, in which the first three steps are iterated in response to significant variations in resource characteristics (see Figure 3.1).

1. *Resource Pool Definition* is the process of collecting (discovering) information concerning the resources available at some point of time. Information is a critical resource, and gathering this information is a vital activity. In the literature, we can find monitoring tools used in the grid context, such as the Network Weather Service [(NWS, 2002)] or Globus Monitoring and Discovery System [(MDS, 2000)].

2. *Task Mapping Strategies* aim to assign the tasks onto selected candidate resources according to some objective function. To achieve this, the current up-

dated information about both the progress of the application and the performance of resources must be available. This step is highly dependent upon the details of the scheduling method used.

3. *Task Execution Process* in which the tasks are submitted to the selected candidate resources to be executed.

4. *Reactivity* concerns the dynamic nature of resources , expressing the need for an iterative use of the first three steps to adapt the application to the dynamic nature of the computational platform.

2.3 The Scheduling Architectures

As shown in Figure 2.3, scheduling architectures can be classified into a three category taxonomy: centralized, hierarchical and decentralized.

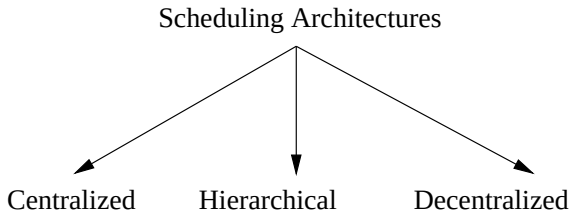


Figure 2.3: Taxonomy of the Scheduling Architectures

1. *A centralized* mechanism involves one centralized scheduler in the execution of the application [(Pegasus, 2003)]. The scheduler maintains all the dynamic information concerning both the progress of the application (tasks and data transfers) and the performance of computational resources. As we will detail in the next chapter, our proposed model (see Figure 3.1) is based on this approach [(Hernandez and Cole, 2007a)]. We note that by maintaining complete knowledge about the application, it is possible to design and include complementary modules (i.e., fault tolerant mechanisms) to support the endeavor of the scheduler [(Hernandez and Cole, 2007b)].

2. *A hierarchical* mechanism is composed of a central scheduler interacting with multiple lower-level schedulers. The central scheduler is responsible for controlling the execution of the application and assigning a portion of the application to

each of the lower-level schedulers [(Thanalapati and Dandamudi, 2001), (Senger et al., 2006), (Cao et al., 2003)].

3. *A decentralized* mechanism allows the tasks composing the parallel application to be scheduled by multiple schedulers. Thus, each scheduler maintains the information relating to the set of tasks assigned to it. The scheduling decisions are made by each scheduler over a portion of the application [(Ranganathan and Foster, 2004), (Lima et al., 1999), (Arora et al., 2002)].

2.4 Applications Class Taxonomy

Parallel applications, as shown in Figure 2.4, can be represented by two main classes: the first class consists of independent tasks and the second class consists of task graphs, for which two main sub-groups are distinguished: parallel applications represented by directed acyclic graphs (DAGs) and other graphs which may contain cycles and/or be undirected, we will refer to such graphs as Non-DAGs. The scheduling mechanism will be highly dependent on the class of the application.

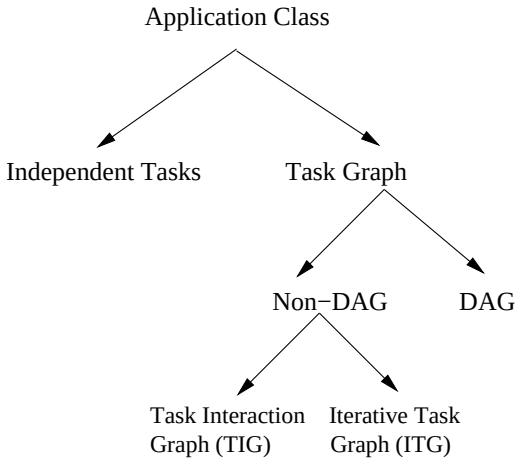


Figure 2.4: Applications Class Taxonomy

2.4.1 Independent Tasks

Parallel applications in this category, are partitioned into a relatively large number of mutually independent tasks. This means that they can be executed in any order. Applications such as master-slave [(Beaumont et al., 2005)] or parameter sweep [(Buyya et al., 2005)] are part of this category. In the literature we can find a vast number

of strategies to schedule applications composed of independent tasks onto processors. Next, we describe some of the well known strategies for this category [(Xhafa and Barolli, 2007), (Braun et al., 2001)].

MET (Minimum Execution Time) [(Amstrong et al., 1998)] is also known in the literature as LBA(Limited Base Assignment). It assigns each task to the processor which allow the smallest execution time for the task. This method is motivated by giving each task the most suitable processor, ignoring its availability. In dynamic heterogeneous environments where resources are shared and non-dedicated, this method could lead to load imbalance among processors.

MCT (Minimum Completion Time) [(Freund et al., 1998)] assigns a task to the processor which allows the minimum completion time. To achieve this, it must consider the availability of processors to compute the estimated completion time.

OLB (Opportunistic Load Balancing) [(Freund et al., 1998)] assigns a task to the processor having the earliest idle machine without considering the execution time of the task on that processor. The notion behind this method is that it tries to keep the processors as loaded as possible. Since this method does not consider the execution times, it can affect the performance of the application.

Min-Min [(Amstrong et al., 1998), (Braun et al., 2001), (Ibarra and Kim, 1977)] consists of two steps. In the first step, it computes the MCT value for each task on each available processor for it. In the second step, the algorithm selects the task with the minimum MCT value and assigns it to the corresponding processor. This is done iteratively until all the tasks have been scheduled. Intuitively, at each iteration, the makespan increases the least possible (shorter tasks first), expecting to obtain a reduced final makespan. However, it is not effective in terms of exploiting the concurrency. For instance, by executing first the shorter tasks, there could be longer tasks which will wait until all the shorter tasks scheduled first in the processor, finish execution, even if there is another processor available with no more tasks to execute.

Max-Min [(Amstrong et al., 1998), (Braun et al., 2001), (Ibarra and Kim, 1977)] is similar to the Min-Min algorithm in the first step. In the second step, the difference is that Max-Min will select the task with the maximum MCT value and assign it to the corresponding processor. In the same manner, this is done

until all the tasks have been allocated. Intuitively, by executing longer tasks first, there could be shorter tasks which can be executed concurrently with longer tasks on other resources, exploiting more effectively the concurrency and expecting to be reflected in better performance.

Sufferage [(Maheswaran et al., 1999)] is based on the idea that better mappings can be generated by assigning a processor to a task that would ”suffer” most in terms of expected completion time if that particular processor is not assigned to it. The sufferage value for each task is defined as the difference between its second-best MCT and the best MCT. Thus, a task having a relatively high sufferage value suggests that if it is not assigned to the processor with the best MCT, it may have a bad performance, as the second-best MCT value is far from the best MCT.

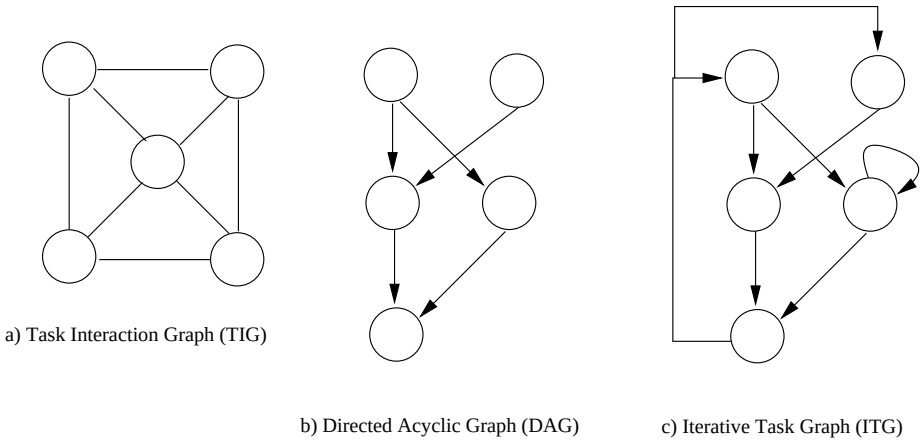


Figure 2.5: Task Graphs

2.4.2 Task Graphs

In this class, we can distinguish a pair of groups: The DAG graph and Non-DAG graph.

2.4.2.1 The Directed Acyclic Graph (DAG)

Applications in this category can be abstracted by directed acyclic graphs (DAGs), in which vertices represent application tasks and edges represent data dependencies between tasks (see Figure 2.5(b)). In the literature we can find a vast number of scheduling algorithms to schedule DAGs onto processors. The objective of such algorithms is to map tasks to processors in a way which minimises the resulting schedule length (makespan) while satisfying the task precedence constraints. Since this

problem is NP-complete, a number of heuristics have been proposed in the literature [(Kwok and Ahmad, 1999a)]. In the literature, [(Kwok and Ahmad, 1999a)] presents a taxonomy of DAG scheduling algorithms grouping common assumptions considered in the algorithms such as the task graph structure, computation costs, data transfer costs, task duplication, number of processors and connectivity among processors. However the algorithms considered are mainly designed for homogeneous environments. [(Casavant and Kuhl, 1988)] presents a hierarchical taxonomy for heuristics scheduling methods in general-purpose distributed computing systems. By combining hierarchical characteristics with more general flat characteristics, it differentiates a wide range of scheduling algorithms. Other taxonomies can be found in [(Braun et al., 1998), (Yu and Buyya, 2005)].Although the DAG scheduling problem has been explored by many researchers, most of the algorithms were designed for homogeneous environments, assuming that the processors have the same capabilities [(Kwok and Ahmad, 1999a), (McCreary et al., 1994)]. Some other algorithms were designed for particular heterogeneous environments, assuming that heterogeneous resources with different capabilities are dedicated and unchanged over time [(Topcuoglu, 2002), (Shi and Dongarra, 2006), (Sih and Lee, 1993)]. Few algorithms can be found addressing the characteristics of heterogeneous resources with changeable capabilities [(Zhao and Sakellariou, 2004b), (Hernandez and Cole, 2007a), (Deelman et al., 2003)]. In Section 2.5, we describe a taxonomy for task mapping strategies addressing the DAG scheduling problem.

2.4.2.2 Non-DAG Graph

Applications in this category are represented by non-directed acyclic graphs (Non-DAGs). In the literature we can distinguish a pair of different classes of Non-DAG applications: The first class is based on the *Task Interaction Graphs (TIGs)* [(Hui and Chanson, 1997)], an undirected graph as shown in Figure 2.5(a), where edges represent interactions between tasks. A TIG was conceived as a graph that divides a program into maximal sequential regions connected by undirected edges to denote the interaction between tasks. In [(Bokhari, 1981), (Hui and Chanson, 1997)], scheduling algorithms are proposed to schedule TIGs onto resources. The objective of the scheduling algorithms is to minimize the maximum processor workload, which is defined for each processor as the total cost due to computation and communication of all the tasks mapped to it. The second class is based on the *Iterative Task Graphs (ITGs)*, used in many scientific problems, which capture the pattern of recurrency at both task and application level,

as shown in Figure 2.5(c). An ITG can be directed or undirected. The problem of scheduling ITGs is also known as loop scheduling. In [(Pam, 1988), (Yang and Fu, 1997), (Gasperoni and Schwiegelshohn, 1992)] can be found scheduling mechanisms for ITGs.

2.5 Task Mapping Heuristic Strategies

The task scheduling problem is in its general form NP-complete, therefore it is not possible to find an optimal solution in polynomial-time unless $P = NP$ [(Kwok and Ahmad, 1999a), (Kwok and Ahmad, 1997)]. An *Optimal* assignment denotes that based on some objective function, the mapping method obtains the best solution (schedule) for the problem [(Papadimitriou and Steiglitz, 1998)]. In the literature, there are only three special cases for which it is possible to obtain an optimal solution in polynomial time. The first case is related to scheduling tree-structured task graphs with uniform computation costs on an arbitrary number of processors [(Hu, 1961)]. The second case is related to scheduling arbitrary task graphs with uniform computation costs on a two-processor system [(Coffman and Graham, 1972)]. The third case involves scheduling an interval-ordered task graph with uniform node weights to an arbitrary number of processors [(Papadimitriou and Yannakakis, 1979)]. In all cases the communication cost among tasks is ignored. Since any credible formalisation of the scheduling problem is NP-complete, some researchers focus on finding suboptimal solutions (*heuristics*) to address the intractability of the problem, which usually obtain a good solution in an acceptably short time. Heuristics can be seen as informed methods, which exploit efficiently the knowledge about the system to obtain a solution. In the literature, most of the mapping methods have been developed for homogeneous computing environments (HCE) [(Kwok and Ahmad, 1999a), (Gerasoulis and Yang, 1993)]. However, these approaches are not easily applicable to heterogeneous environments, which initially were DHCS, as they do not include mechanisms to properly map tasks on heterogeneous resources. Thus, most of the scheduling approaches for heterogeneous computing systems were developed for DHCS, with the common assumptions that heterogeneous resources are dedicated and unchanging over time [(Ercegovac, 1998), (Leangsuksun and Potter, 1993), (Eshaghian and Wu, 1997), (Eshaghian, 1993), (Yang et al., 1993)]. As shown in Figure 2.6, heuristics can be grouped in four main categories: approximate, clustering, task duplication and list scheduling.

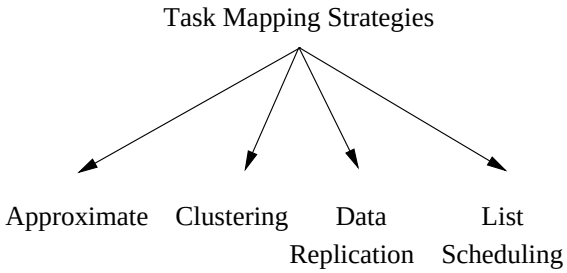


Figure 2.6: Taxonomy of Task Mapping Strategies

1. *Approximate* algorithms usually search for a solution which is not the optimal, but it is considered to be near the optimal value in the solution-space. They are also known as iterative algorithms, because they usually explore (iterate) several candidate solutions in the solution-space before finding a satisfactory solution according to some objective function. Depending on the size of the problem, the process to obtain a satisfactory solution may take a considerable time [(Sait and Youssef, 1999), (Abraham et al., 2000), (Nath, 1997), (Spooner et al., 2003)].

2. *List scheduling* based algorithms basically consist of two phases: The *task prioritization* phase in which a rank (priority) is assigned to each task, such that if we construct an ordered list of tasks in decreasing order of priority, then we obtain a predicted sequence of tasks execution. 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). The notion behind the approach is that the tasks with higher priority will be executed first, expecting to improve the performance. These algorithms tend to perform local optimization by assigning one after another each task onto the suitable resources which minimizes some objective function. Typically, the task prioritization process is based on static information about the application (weights of nodes and edges). A pair of well known attributes are used to set the task priorities, the *t-level* (top level) attribute and *b-level* (bottom level) [(Adam et al., 1974b), (Gerasoulis and Yang, 1992)]. The *t-level* of a node v_i is defined as the length of a longest path from an entry node to v_i (excluding v_i). The length of a path is determined by the sum of all the node and edge weights (execution and communication costs) along the path. The *b-level* of a node v_i is defined as the the length of a longest path from v_i to an exit node. The $critical path$ of a DAG is the longest path in the DAG. The nature of the *b-level* and *t-level* attributes

allows identification of the nodes on the critical path(s). This can be done by computing for each task, the static attribute $CP = t\text{-}level + b\text{-}level$, then those tasks on the critical path(s) will have the highest value of CP. The algorithms in this category tend to provide good solutions (quality of schedules) in a short time [(Kwok and Ahmad, 1999b)]. The dynamic models proposed in our research work are based on this category of heuristics. Next we describe some of the well known list scheduling algorithms.

- The *Dynamic Level Scheduling (DLS)* algorithm [(Sih and Lee, 1993)] was proposed for both homogeneous and heterogeneous environment. The DLS algorithm for homogeneous environments, determines the task priorities by computing a dynamic attribute called *dynamic level (DL)*. The *DL* of a task-processor pair is denoted as $DL(v_i, p_j)$, and reflects how well v_i and p_j are matched. The *DL* is determined by two terms. The first term is the Static Level (SL) of the task which in this case is equal to the *b-level* attribute and the term $max(t, DA(v_i, p_j))$ which denotes the time at which v_i can start execution (ST) on p_j, as it receives the last data transfer from their predecessors. Thus, *DL* is determined by $SL(v_i) - ST(v_i, p_j)$. At each step, the algorithm computes the *DL* for each ready task on every candidate processor. The task-processor pair which gives the largest value of *DL* among all other pairs is selected for scheduling. This process is repeated until all the tasks are scheduled. At this point it is assumed that all processors are homogeneous. Thus, the static level $SL(v_i)$ loses its meaning when the processors are heterogeneous. The authors adapted the DLS algorithm to consider heterogeneous processors by modifying the definition of *DL*. A key new term $\Delta(v_i, p_j) = E*(v_i) - E(v_i, p_j)$ is added to the expression of *DL*, denoting the varying processing costs. $E*(v_i)$ is the median of execution times of v_i over all processors and $E*(v_i, p_j)$ denotes the cost of executing v_i on p_j. A large positive $\Delta(v_i, p_j)$ indicates that p_j executed v_i more rapidly than most processors, while a large negative $\Delta(v_i, p_j)$ indicates the opposite.

  We notice that the DLS algorithm is one of the earliest algorithms to consider heterogeneous processors. Other recent algorithms tend to include static information about the heterogeneous processors when defining the static level attribute (i.e., *b-level*) [(Topcuoglu, 2002)].

- The *Earliest Start Time (EST)* algorithm was proposed for homogeneous processors [(Graham, 1969)]. The notion behind the algorithm is to start executing a task as early as possible. This version assumes that communication costs are zero. The algorithm maintains a list of ready tasks. At each step, the algorithm determines the predicted start time of the task on each processor. The computation of this time depends upon the availability of the processors and the predicted finish time of the predecessors. Then, the task is mapped onto that processor which allows the minimum earliest start time.
- The *Earliest Completion Time (ECT)* algorithm seeks to execute a task as soon as possible [(Wang and Cheng, 1992)]. It considers homogeneous processors and assumes that communication costs are zero. At each step, the algorithm determines the predicted completion time of the task on each processor. The computation of this time depends upon the predicted start time of the task plus the expected execution time of the task on the processor. Thus, the task is mapped onto the processor which allows the minimum earliest completion time.
- The *Heterogeneous Earliest Finish Time (HEFT)* algorithm [(Topcuoglu, 2002)] is a natural evolution of the *ECT* algorithm to heterogeneous environments, since using *ECT* with heterogeneous processors might lead to poor predictions. HEFT has two major phases: a *task prioritizing phase* for computing the priorities of all tasks and a *candidate processor selection phase* for selecting the tasks in the order of their priorities and scheduling each selected task onto that processor which allows the task's earliest finish time. The major adaptation to consider heterogeneous processors was in the *task prioritizing phase*. HEFT considers static knowledge about the heterogeneous processors by maintaining for each task, the computation cost of the task on each heterogeneous processor. Obviously in homogeneous environments, the computation cost is the same for all the processors. This knowledge is used to determine the computation weight of a node, which now is part of the formula to compute the rank of the task. In HEFT the computation weight of a node is approximated by the average of its weights across all processors. In [(Zhao and Sakellariou, 2004a)], it is shown that there are different schemes for computing the weights of a task and depending on the scheme used, the makespan of the application may

be affected.

3. *Clustering* based heuristics consist of two phases: the clustering-phase in which tasks are grouped into clusters and the mapping-phase in which the clusters are mapped onto processors [(Gerasoulis and Yang, 1992), (Gerasoulis and Yang, 1993)]. The clustering-phase is as follows. Initially, each task is considered a cluster. Then, two clusters are merged if the merging helps to optimize an objective function (i.e., reduce the completion time). The merging process continues until no more merging is possible. The notion behind the method is that by grouping tasks into the same cluster, it is possible to reduce the amount of communication among the tasks. Thus, the tasks grouped into the same cluster are allocated to the same processor. An important characteristic of this heuristic is that it allows scheduling decisions based on global aspects (e.g., critical path), which it is believed could derive better solutions. Then, the mapping phase will allocate the tasks of each cluster onto the same processor. For the case in which the number of clusters created is greater than the number of processors, usually there is an extra merge process in which a further merge is performed with the considered clusters [(Eshaghian and Shaaban, 1994)]. Next, we discuss some of the well known clustering algorithms.

   - *The Edge-Zeroing (EZ)* algorithm [(Sarkar, 1989)] selects clusters for merging base on the edge weights. This algorithm computes the $b-level$ value for each task and creates sorted list of edges in a descending order of edge weights. Thus, at each step it selects the largest edge weight and zeros the edge weight if the completion time (CP) is not increased. When two cluster are merged then all the edges involving these two clusters are zeroed. The ordering of tasks within a particular cluster is based on their $b-level$ value.

   - *The Linear Clustering (LC)* algorithm [(Kim and Browne, 1988)] considers the critical path (CP) to merge tasks into a single cluster. The algorithm first determines the set of tasks forming the CP, then such tasks are merged into a single cluster, zeroing all the edges and removing all the edges incident to the critical path (decoupling the cluster). Obviously by decoupling the cluster formed by the tasks in the CP, the DAG will have a new CP. This process is repeated until all the tasks are clustered.

- *The Dominant Sequence Clustering (DSC)* algorithm [(Yang and Gerasoulis, 1994)] introduces the Dominant Sequence (DS) of a DAG as the length of the critical path during the process. This means that this algorithm attempts to reduce the DS by clustering tasks. Obviously at the beginning of the process the length of the *CP* is equal to the length of the DS. Initially, the *t-level* attribute is computed for each task and an ordered list of tasks in decreasing order of *t-level* is created. Then, for each task in the list, the algorithm is able to distinguish those tasks which are part of the DS and those which are not. Thus, if the task selected is part of the DS then it merges the task with one of its parents if such a merge reduces the length of the DS by zeroing the edge, otherwise the task is considered as a new cluster. This means that the decisions are made based on the global impact on the expected execution time. If the task is not part of the DS, then it merges the task with one of its parents if such a merge reduces the *t-level* value of the task.

4. *Task Duplication based heuristics (TDB)* considers the replication of tasks as a strategy to reduce the schedule length (makespan) [(Papadimitriou and Yannakanis, 1990), (Kruatrachue and Lewis, 1988), (Bansal et al., 2003), (Ahmad and Kowk, 1998)]. The notion behind this method is to use resource idle-times to replicate parent tasks to reduce the waiting time of dependant tasks. Basically, the key aspect in this strategy is to identify those critical tasks to replicate. A pair of strategies can be distinguished: The first strategy considers replication of some parent tasks based on a particular criterion. The second strategy considers the replication of all the possible parent tasks. Next we describe some of the well known task duplication algorithms.

   - The *Duplication Scheduling Heuristic (DSH)* algorithm [(Kruatrachue and Lewis, 1988)] uses the idea of list scheduling algorithms combined with duplication of tasks to reduce the makespan. As in list scheduling algorithms, it creates a task list sorted by the static attribute $b-level$. Then, it selects a task from the list and it predicts the start time of the task on a particular processor as follows: first compute the start time of the task on the processor. Next, it evaluates if duplicating the predecessors can reduce the predicted start time of the task. The duplication process tries to find an idle-time slot of the processor and it will insert the duplicated predecessor tasks

until either such a slot is not available or the start time of the task can not be improved further. The process is repeated until all the tasks have been allocated onto processors. DSH duplicates tasks where necessary, avoiding redundant duplications, to reduce the overall communication delay.

- The *Bottom-Up Top-Down Duplication Heuristic (BTDH)* algorithm [(Chung and Ranka, 1992)] is an extension of the DSH algorithm. The major improvement is that BTDH attempts to duplicate the predecessors onto the processor assigned to their succeeding task even if the idle time-slot is filled up and it also ignores the effect of increasing the start time when duplicating predecessors. The notion behind the algorithm is that the start time may eventually be reduced by duplicating all the necessary predecessors.
- The *Critical Path Fast Duplication (CPFD)* [(Ahmad and Kowk, 1998)] is based on the notion that an accurate identification of the important tasks for duplication may lead to obtain short schedules. It classifies the nodes in a DAG into three categories in the order of decreasing importance: Critical Path Nodes(CPN), In-Branch Nodes(IBN) and Out-Branch Nodes(OBN). The authors believe that the most important nodes are on the critical path (CP), as CP is the longest path of the task graph and, therefore, the finish times of CP nodes (CPNs) bound the final schedule length. An In-Branch Node (IBN) is a node from which there is a path reaching a CPN. An Out-Branch Node (OBN) is considered the least important of the nodes, being neither a CPN nor an IBN. The procedure contains three main parts. First, it creates a priority list called the CP-Dominant Sequence containing in the first instance all the tasks on the CP and all the OBNs are appended to the sequence respecting the precedence constraints. Second, for each task in the list, it determines the earliest start time of the task on each candidate processor and selects that processor which allows the minimum earliest start time. Third, a minimization-process tries to minimize the start time of the task by considering duplicating each possible predecessor (starting from the predecessor whose message arrives last and so on) in the earliest idle-time slot of the selected processor. If duplicating a particular predecessor is successful then the start time of the task will be reduced and the process will try to further minimize, by considering duplicating the next predecessor. If the duplication is not successful then the minimization-process stops. The second and third steps continue until there are no more

tasks in the list.

These algorithms show the close relationship between the parallel application, target platform and scheduling mechanism. We observe that as the computational environment evolved from homogenous to heterogeneous environments, which initially were dedicated heterogenous computing systems (DHCS), the mapping strategies were adapted with particular mechanisms to properly achieve the task mapping on heterogeneous resources.
The advent of emerging technologies such as SHCS will allow scientists and engineers to build distributed applications to exploit resources at global scale. However, the mapping strategies for DHCS, are not capable of addressing the dynamic nature of SHCS, as they just use static information of the computational resources to make scheduling decisions before the execution, ignoring that resources may change dynamically over time. We will consider this issue in the next section.

2.6 Task Mapping Operation Modes

In Section 2.5, we described that some mapping methods addressing the heterogeneity of the computational resources, are not capable of addressing the dynamic nature of emerging computational platforms such as SHCS, as they assume that resources are dedicated and unchanging over time. In the literature, few heuristic mapping methods have been developed for SHCS [(Maheswaran and Siegel, 1998), (Zhao and Sakellariou, 2004b), (Hernandez and Cole, 2007a), (Hernandez and Cole, 2007c)]. They include particular considerations of the mode in which the mapping method would operate. As shown in Figure 2.7, the mode in which strategy mapping methods are implemented, can be classified as either static or reactive mode.

1. In a *Static mode*, all the static information related to the application and computational resources is assumed to be available before the execution. Thus, an initial static schedule is generated by a particular mapping strategy, launched to the target architecture and maintained during the execution of the application. A pair of assumptions are distinguished: the first assumption is related to the accurate knowledge about both the DAG application and the computational system (i.e., task computation times, bandwidth, data dependencies and communication times among tasks). The second assumption states that the resources are dedicated and the fluctuations in the variability of resources are minimum.

Scheduling algorithms designed for *homogeneous* environments (e.g., parallel computers) or some others designed for *heterogeneous* environments with a tight degree of control over resources such as DHCS, fit in this mode. We will refer to mapping methods operating in static mode as static mapping methods.

2. *Reactive mode* is based on the notion that maintaining an initial static schedule in computational environments where the performance of resources may vary over time, even during the execution of the application, may affect the predictions and eventually, the performance of the application. Thus, mapping strategies operating in reactive mode seek to incorporate dynamic information into the scheduling decisions. To achieve this, reactive approaches constantly monitor the state of both the progress of the application and performance of resources before taking scheduling or rescheduling decisions. We will refer to mapping methods operating in reactive mode as reactive mapping methods. The reactive mode can be classified into either *rescheduling* or *run-time scheduling* schemes. The *rescheduling* scheme, is related with cyclic use of a mapping method over time. The notion behind the rescheduling mechanism 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. Based on the criterion on which the application is rescheduled, two different approaches are observable: remapping points and events. Remapping (or rescheduling) points set over time will determine the moment at which the application must be rescheduled. An important issue is to optimize the cost of the remapping points. Using many rescheduling points may incur in a high overhead cost, while using fewer rescheduling points may result in an inadequate reaction to the problem. The other approach is related to rescheduling the application based on the detection of predefined events [(Huedo et al., 2004), (Yu and Shi, 2007), (Yu and Shi, 2004)]. [(Hernandez and Cole, 2007a)] presents an approach which includes the cycle use of a mapping method with fixed-period rescheduling point. The details will be presented in chapter 3.

   In [(Zhao and Sakellariou, 2004b)] a rescheduling policy is proposed which attempts to reschedule the application at a few selected points during execution, expecting to reduce the overhead cost generated by rescheduling the application. To achieve this, the approach evaluates two different metrics: the spare time and the slack of a node. The spare time denotes the maximal time that a particular

predecessor node can execute without affecting the start time of some of its dependent nodes that are either connected by an edge in the DAG or are adjacent in the execution order of the assigned processor. The slack of a node is defined as the minimum spare time on any path from this node to the exit node of the DAG. This is the maximum delay that can be tolerated in the execution time of the node without affecting the overall schedule length (makespan). For instance, if the slack of a node is zero, then it means that the node is critical and any delay in the execution of this node will affect the makespan of the application.
Another approach in this category can be found in [(Spooner et al., 2005)] where an iterative invocation of a genetic algorithm is proposed, considering migration of tasks when a defined performance contract is not achieved.
Our research is focused on task mapping heuristics operating in reactive mode. Our model allows rescheduling of an executing application in response to significant variations in resource characteristics, to efficiently execute parallel applications on SHCS.

In the *run-time scheduling*, rather than generating a refined schedule over time, the mapping strategy operates in a manner that progressively schedule blocks of tasks over time. It uses the run-time information that becomes available from the execution of previous blocks of tasks to make scheduling predictions for subsequent blocks of tasks. The process continues until all the blocks of tasks have been executed.
The *Just In-time approach* is proposed in the Pegasus project [(Deelman et al., 2004)]. They propose to schedule all tasks at run-time, as they become available. To achieve this, they designed a mechanism (the partitioner) that partitions the abstract workflow (DAG) into smaller partial workflows. The dependencies between the partial workflows reflect the original dependencies between the tasks of the abstract workflow. Once the partitioning is performed, Pegasus maps and submits the partial workflows to the dynamic system as follows: If there is a dependency between two partial workflows, Pegasus is made to wait (by [(DAGman, 2002)]) to map the dependent workflow until the preceding workflow has finished executing.
Another approach is proposed in [(Maheswaran and Siegel, 1998)], where a hybrid remapper is presented to dynamically schedule DAG applications. It assumes that an initial schedule is provided as an input. The hybrid remapper

executes in two phases: in the first phase, prior to the execution, it partitions the DAG into blocks (the levels of the DAG) such that the subtasks within a block (level) are independent. The second phase of the hybrid remapper, executed during application run-time, involves the execution of tasks proceeding from the top block (the highest level) to the bottom block. Thus, it uses the run-time information that becomes available from the execution of previous blocks of tasks to make scheduling predictions for subsequent blocks of tasks and eventually remap the remaining blocks of tasks. This approach is similar to the just-in-time approach, the difference is that the remapper allows the execution of several blocks in an overlapped fashion.

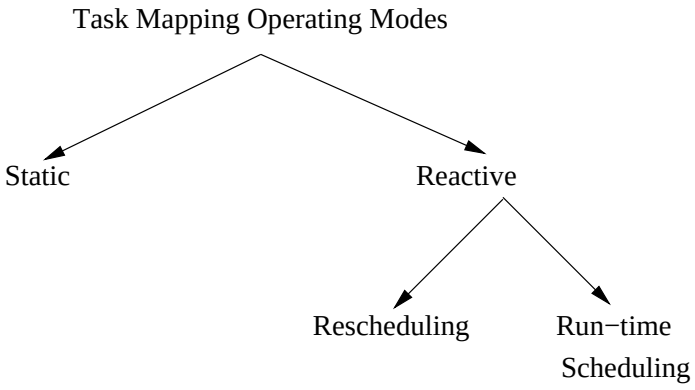


Figure 2.7: Taxonomy of Task Mapping Operation Modes

2.7 Data Awareness Taxonomy

In this section we describe another dimension of the mapping methods for SHCS. It concerns the treatment of the results of completed tasks, which can be an important issue if the applications are large and complex. Thus, as shown in Figure 2.8, mapping methods for SHCS can be classified in two main categories: data-aware and data unaware.

The *data-aware approach* includes mechanisms to consider results of completed tasks (i.e., output files) over execution. In [(Hernandez and Cole, 2007c)], the GTP/c model is presented. This proposes the reuse of data for migrated tasks in order to reduce the impact of migration cost on makespan. This may be relevant in applications with a relatively high number of tasks and data transfers (i.e., data-intensive applications). Details of the GTP/c model will be presented in chapter 3. We notice that this

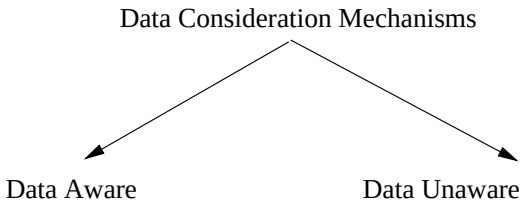


Figure 2.8: Data Awareness Taxonomy

approach assumes data-storage of sufficient size on each processor to perform all the data transfers among tasks. A complementary work can be found in [(Ramakrishnan et al., 2007)], which considers data-storage constraints when scheduling data intensive applications. Their approach is two-fold: they minimize the amount of space a workflow requires during execution by removing results of completed tasks (i.e., output files) at runtime when they are no longer required and they schedule the workflows in a way that assures that the amount of data required and generated by the workflow fits onto the individual processors. Most of the approaches in the literature are *data-unaware*, and do not take into account data-storage constraints.

2.8 Taxonomy of Fault Tolerance Mechanisms

Although, the reactive scheduling strategies described in Section 2.6 react in response to significant variations in resource characteristics, they are not necessarily able to react to processor failure during execution. Fault tolerance is an important issue in SHCS as the availability of resources cannot be guaranteed. Some work has been conducted to design fault tolerant mechanisms for DAG applications [(Medeiros et al., 2003)] to preserve the execution of the application despite the presence of a processor failure. In Figure 2.9, we show a fault tolerance mechanisms taxonomy, similar to that proposed in [(Hwang and Kesselman, 2003)]. Fault tolerant mechanisms can be classified in two major categories according to the level. The first category is at *task level*, in which just the knowledge about the task (i.e., processor assigned) is used to redefine just the status of a particular failed task. The second category is at *application level* in which more knowledge (i.e., status of predecessors and successors) is required to redefine the whole status of the application in order to address the failure.

The *task level* category groups several strategies such as retry, alternate resource, checkpoint/restart and task duplication. The retry approach simply considers a number of tries to execute the same task on the same resource after detecting the failure

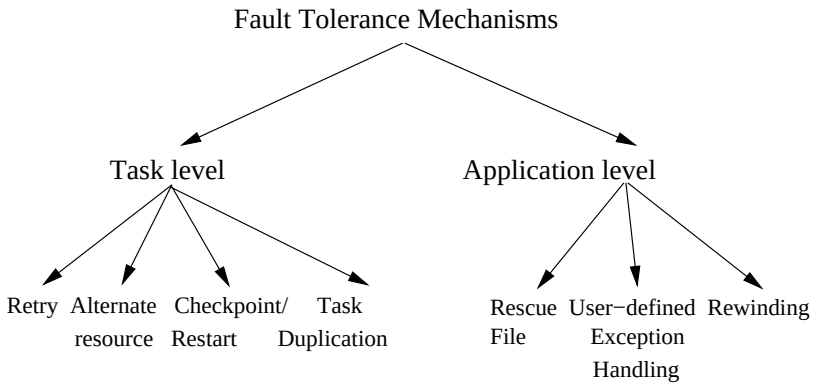


Figure 2.9: Fault Tolerance Mechanisms Taxonomy

[(DAGman, 2002), (Taverna, 2004)]. The checkpoint/restart approach usually saves the computation state over time, such that it migrates the saved work of failed tasks to other processors, so that the tasks can resume execution from the failure point [(Jalote, 1994), (Condor, 2001)]. The task duplication approach selects tasks for duplication, hoping that one of the replicated tasks will finish successfully in case of failure [(Abawajy, 2004), (In et al., 2005)].

The *application level* category groups several strategies: Rescue file, Redundancy, User-defined Exception Handling and Rewinding. The rescue file mechanism is proposed in [(DAGman, 2002)]. Such a mechanism consists of the resubmission of uncompleted portions of a DAG when one or more tasks resulted in failure. If any task in the DAG fails, the remainder of the DAG is continued until no more forward progress can be made due to the DAG dependencies. At this point, DAGMan produces a file called a Rescue DAG (input file), containing information about the progress of the DAG (unfinished and successfully finished tasks). Then, using this Rescue DAG as input file, the unfinished tasks are resubmitted. The User-defined Exception Handling is proposed in [(Hwang and Kesselman, 2003)] to allow users to give a special treatment to a specific failure of a particular task. A rewinding mechanism is proposed in [(Hernandez and Cole, 2007b)] to address a processor failure. The rewinding mechanism seeks to preserve the execution of the application by recomputing and migrating those tasks which will disrupt the forward execution of succeeding tasks. The mechanism rewinds the progress of the application to a previous state, thereby preserving the execution despite the failed processor(s). Details of the mechanism will be presented in Section 3.5.

2.9 Global Scheduling Simulators

In the literature, we find simulators which allow building and testing mapping methods for distributed environments. The major objective of the simulators is to provide a framework to model, evaluate and compare scheduling strategies in a repeatable and configurable environment.

- *Simgrid* [(Simgrid, 2001), (Legrand et al., 2003)] is a discrete-event simulation toolkit. It provides a set of core abstractions and functionalities that can be used to build simulators for specific application domains and/or computing environment topologies. In Simgrid, resources are modelled by their latency and service rate. Simgrid provides mechanisms to model performance characteristics either as constants or from traces. This means that the latency and service rate of each resource can be modelled by a vector of time-stamped values (or traces). Traces allow the simulation of arbitrary performance fluctuations in computational resources. The user is responsible for scheduling computations and communications in the right order (in the case of DAGs) on the right resources.

- *GridSim* [(GridSim, 2002)], is an object-oriented toolkit implemented in Java for resource modelling and scheduling simulation. As Simgrid, it is a discrete-event simulation toolkit, which allows us to investigate and model scheduling mechanisms in SHCS. It can be used to simulate application schedulers for single or multiple administrative domains distributed computing systems such as clusters and networks of workstations. GridSim simulates time and space-shared resources with different capabilities, time zones and configurations. One main difference is that GridSim incorporates economic issues, where the implementation of mapping methods includes deadline and budget constraints in the scheduling decisions.

- *GangSim Simulator* [(Dumitrescu and Foster, 2005)] is the result of the enhancement of the Ganglia Monitoring Toolkit [(Massie et al., 2004)]. It was designed to support studies of scheduling strategies in grid environments, which comprise potentially large number of resources, resource owners and virtual organizations (VOs). It allows us to model not only sites but also components of virtual organizations such as users and planners. GangSim mainly focus on exploring the interactions between local and VOs resource allocation policies. GangSim simulates a policy-driven management infrastructure in which policies concerning the

allocation of resources within VOs and the allocation of resources across VOs at individual sites interact to determine the ultimate allocation of individual computational resources. In addition, GangSim permits parallel processing and can combine simulated components with instances of a Ganglia Monitoring Toolkit running on real resources.

- *OptorSim Simulator* [(Bell et al., 2003a)] was developed as part of the European DataGrid project [(Project, 2004)]. OptorSim is a grid simulator distinguished by including data replication strategies. Thus, OptorSim allows to investigate scheduling algorithms to ensure effective usage of resources and replication algorithms which involve the creation and management of data replicas at different sites, in order to optimize the data access time. The scheduling algorithms are focused on reducing the cost needed to run a job, including the following approaches: Random (a site is chosen randomly), Access Cost (cost is the time needed to access all the files needed for the job), Queue Size (cost is the number of jobs in the queue at that site) and Queue Access Cost (the combined access cost for every job in the queue, plus the current job). The replication algorithms are mainly divided in three replication strategies. In the first strategy, the non-replication option is available. In the second strategy, it always replicates when a file is requested for processing, deleting existing files if necessary. The third strategy involves an economic model in which sites ”buy” and ”sell” files using an auction mechanism [(Bell et al., 2003b)] .

Other simulators include DAGsim [(Jarry et al., 2000)],Bricks [(Takefusa et al., 1999)] and Microgrid [(Song et al., 2000)]. The DAGsim project is implemented on top of Simgrid, and is focused to implement and evaluate DAG mapping methods in several simulation scenarios. The Bricks project mainly focuses on simulating resource allocation strategies and policies for global computing systems. In the bricks context, global computing systems are composed by client-server systems that provide remote access to scientific libraries and packages running on high performance computing. Unlike other simulators, Microgrid aims to allow grid researchers to evaluate and execute their applications on a virtual grid emulated environment. Microgrid supports the execution of applications on emulated grid resources, which use the Globus toolkit [(Foster and Kesselman, 1997)].

2.10 Summary

In this chapter we have presented a literature review related to our research. It embraced the different elements involved the DAG scheduling problem. We reviewed the heuristic mapping strategies that can be found in the literature. Next, we described the operation modes in which mapping strategies are implemented, being the reactive mode used to address the task mapping on heterogeneous and dynamic resources. We described particular issues related to dynamic heterogeneous environments, such as fault tolerance and data-aware scheduling. We finished this chapter by describing some simulation toolkits from the literature, to build and test mapping methods.
In the work presented here, we investigate the problem of scheduling DAG applications on shared heterogeneous computing systems (SHCS), which according to our classification in Figure 2.1, is an emergent class of heterogeneous computing systems, distinguished by the heterogeneous and dynamic nature of the computational resources. Classifications in the Figures 2.2 and 2.3 indicate that we follow an application level scheduling model operating under a centralized architecture scheme. In terms of our classification in Figure 2.5, we propose a reactive mapping method, which considers the cyclic (rescheduling) use of a mapping method over time. Our task mapping model is based on the list scheduling approach, one of the groups in which heuristic mapping methods can be classified according to the taxonomy showed in Figure 2.6. 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. Additionally, list scheduling algorithms usually generate good solutions in a reasonable amount of time. Our reactive approach is based on the use of remapping points to reschedule the application. Unlike other reactive approaches (i.e., run-time scheduling), we believe that this scheme can help not only to react to dynamic changes in resources, but to inaccurate predictions from previous schedules. An extension in our proposed model is related to the data consideration mechanisms (see the classification in Figure 2.8) into the scheduling decisions to improve the utilization of resources. We consider data reuse of the results of completed tasks over the execution of the application, which can be an important issue if we consider the data-storage constraints. Finally, we propose a fault tolerant mechanism at the application level (see the classification in Figure 2.9), to preserve the execution of the DAG application,

despite the presence of a processor failure. Unlike other fault tolerant approaches, our model is distinguished by considering recomputation and migration of tasks which had already finished, but which will disrupt the forward execution of succeeding tasks. For instance, those preceding tasks already executed on the failed processor still transmit data to succeeding tasks.

Chapter 3

The Global Task Positioning (GTP) Models

Our research considers the problem of scheduling parallel applications, represented by directed acyclic graphs (DAGs), 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. This chapter describes our proposed model to address the dynamically heterogeneous nature of SHCS. The description 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. The term *Global* denotes the coordinated collaborative environment of shared resources potentially located at global scale, made possible by advances in network technology. 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. Effective DAG mapping methods for SHCS must include fault tolerant mechanisms to preserve the execution of DAG applications despite the presence of a processor failure. To address this, we designed the rewinding mechanism, which preserves the execution of the application by recomputing and migrating those tasks which will disrupt the forward execution of succeeding tasks.

3.1 Description of the GTP Model

Our overall system is sketched in Figure 3.1, where ITG represents the task graph, STG contains dynamic information concerning the progress of the tasks, and SRP contains dynamic information concerning the current performance of the shared resources in SHCS. We will define these structures more formally during the content of this chapter. Our model assumes that the initial task graph is given as the ITG structure, together with the initial model of resources as the *SRP* structure. An initial schedule μ^0 is generated by a standard static mapping method (e.g., HEFT). After this, the initial task graph is copied into the STG structure and the initial schedule is launched to SHCS. Our model addresses the dynamic nature of SHCS with cyclic use of a mapping method to react to dynamic changes in resources. We will refer to each cycle as a *rescheduling point* (RP). We consider a fixed frequency rescheduling cycle during execution. The selection of the rescheduling cycle frequency is discussed in Section 4.2. At each rescheduling point the dynamic information describing the progress of the application (tasks and data transfers) is updated in STG. In the same manner, the latest information about resource performance is updated in *SRP*. Then, given the latest information about both resources and application, our model generates a refined schedule μ^t, aiming to minimize the estimated makespan of the application. It considers migration of tasks when the cost of the migration itself is outweighed by the global time saved due to execution at the new site. The cyclic process continues until the application finishes execution.

3.1.1 Definition of the SHCS

To represent Shared Resource Pools (SRP) (see Figure 3.2(c)), we will use graphs *SRP* :: $(P, L,$avail,bandwidth$)$ where P is the set of available processors in the system, $p_i (1 \leq i \leq |P|)$. We assume data storage of sufficient size on each $p_i \in P$ to perform all the data transfers among tasks. L is the set of communication links connecting pairs of distinct processors, $l_i (1 \leq i \leq |L|)$ such that $l(p_m, p_n) \in L$ denotes an undirected communication link between p_m and p_n. We assume that the intra-processor communication cost $(p_m = p_n)$ 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[(NWS, 2002)] or Globus MDS[(MDS, 2000)]). Thus, at time t we assume knowledge of $avail^t :: P \rightarrow [0..1]$, capturing the availability of each CPU and

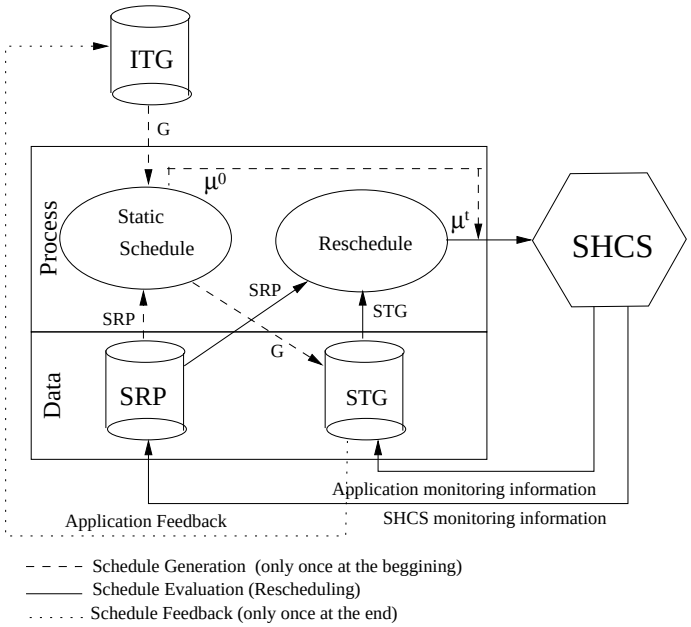


Figure 3.1: The Global Task Positioning (GTP) mapping method

$bandwidth^t :: L \rightarrow Float$ capturing the available bandwidth on each link. We note that the models GTP and GTP/c described in Section 3.2 and Section 3.3 respectively, assume $|avail^t > 0|$ in resources. The rewinding mechanism described in Section 3.5 addresses the case of extreme variations when the variability is equal to zero ($avail^t = 0$).

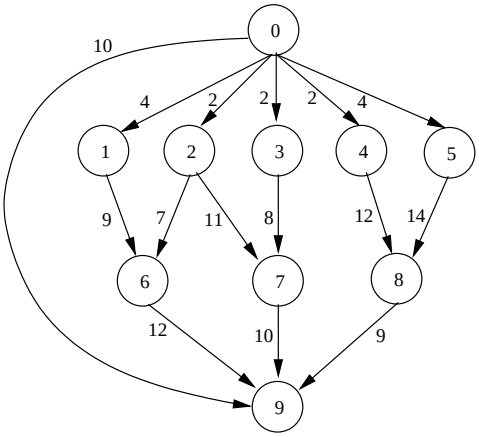
3.1.2 Definition of the Input Task Graph (ITG)

Static information about the DAG application (see Figure 3.2(a)) is represented by an *input task graph* $ITG :: (V, E, data, W)$. V is the set of tasks, $v_i (1 \leq i \leq |V|)$. $E \subseteq V \times V$ is the set of directed edges connecting pairs of distinct tasks, $e_i (1 \leq i \leq |E|)$, where $e(v_i, v_j) \in E$ denotes a precedence constraint and data transfer from task v_i to task v_j. An edge $e(v_i, v_j) \in E$ 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 $Pred(v_i)$ to denote the subset of tasks which directly precede v_i and $Succ(v_i)$ to denote the subset of tasks which directly succeed v_i. Those tasks v_i such that $|Pred(v_i)| = 0$ are called entry tasks and $|Succ(v_i)| = 0$ are called exit tasks. 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 $data :: V \times V \rightarrow Int$ 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 (see Figure 3.2(b)) with $W :: V \times P \rightarrow Int$, 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*). In practice this information may be difficult to obtain, and concrete realizations of such systems may have to rely upon programmer estimates, information from previous runs and other ad-hoc methods. For applications which are executed repeatedly, information to initialize W could be maintained from one run to the next. We will factor in the effect of dynamically affected processor availabilities and link bandwidths during execution. For future convenience, since given they will be traversed in a top-down fashion, we use $Level(v_i)$ to denote how deep in terms of number of edges, a task v_i is from the entry node. In the same manner we can compute the inter-dependencies level $IDL(e(v_i,v_j))$ for a particular $e(v_i,v_j) \in E$, which denotes how deep in terms of number of edges, a task v_i is from the task v_j. We notice that the impact of the mechanism through which communication of task results is achieved, is not fully explored. In terms of the communication model among tasks, we observe two main models to allow the transfer of data among tasks, the *PUSH model* and the *PULL model*. In the PUSH model, as soon as a task finishes execution, it pushes the data result to its successors to be executed. In the PULL model, as soon as a task is mapped on a particular processor, it requests to pull the data needed from its predecessors. We will show in Section 5.7 that ignoring this issue may negatively affect the performance of the application.

3.1.3 Definition of the Situated Task Graph (STG)

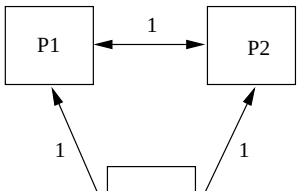
Just as we maintain dynamic information *avail* and *bandwidth* on the *SRP*, so we must maintain additional dynamic information on the progress of the DAG execution. We model this by augmenting the static ITG, to form a *Situated Task Graph* STG. This includes information on current schedule of tasks, 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



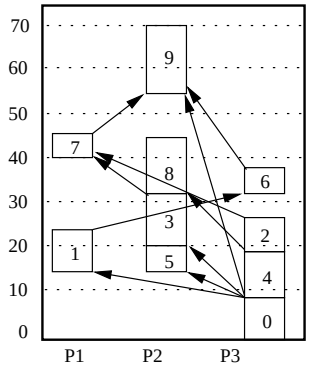
a) DAG application

Task	P1	P2	P3	$\overline{Wi,j}$	Ru
0	12	13	9	11	75
1	10	15	11	12	58
2	13	10	8	10	54
3	15	11	13	13	54
4	9	12	10	10	59
5	8	7	12	9	60
6	8	10	7	8	37
7	5	8	7	6	33
8	11	13	10	11	37
9	20	16	17	17	17

b) Computation times of tasks on PE's



c) SHCS architecture



d) Initial Schedule (HEFT)

Figure 3.2: Example of the elements required by the GTP model

incur no migration cost because retransmission of data is not needed.

We define $STG :: (V, E, data, W, \Pi, \kappa^c \kappa^d)$, where the first four components are taken directly from the corresponding ITG. We use $\Pi :: V \rightarrow P^+$ 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, $\Pi(v_i) = NONE$. 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 mechanisms at each rescheduling point. 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. The initial STG is effectively just the ITG with all completions equal to zero and all task placements set to NONE.

3.2 The *GTP* Scheduling Method

Before describing the details of the *GTP* scheduling algorithm, we describe the process of setting the tasks ranks, the migration model and the costing of candidate schedules.

3.2.1 Setting Task Ranks

As described in Section 2.5, list-scheduling heuristic approaches maintain a list of unfinished tasks ordered by a *rank*(priority), which is computed statically. Such list denotes a predicted sequence of tasks execution. There are several methods to statically set the ranks of tasks for a heterogeneous environment [(Kwok and Ahmad, 1999a)]. We use $R_u(v_i)$ (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. $R_u(v_i)$ is computed recursively as,

$$R_u(v_i) = \overline{W_i} + max_{v_j \in Succ(v_i)}(data(v_i, v_j) + R_u(v_j)) \tag{3.1}$$

where $\overline{W_i}$ is the average execution cost of task v_i across all processors and it is defined by,

$$\overline{W_i} = \frac{(\Sigma_{m=1}^{|p|} W(v_i, p_m))}{|P|} \tag{3.2}$$

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)]. There are other schemes to approximate the weight for nodes and edges of the task graph. In [(Zhao and Sakellariou, 2004a)], an experimental investigation is conducted into the rank function of the HEFT algorithm, the same that we use in equation 3.2. They experiment with different schemes (e.g., mean value, worst value) for computing these weights. They show that the predicted makespan of the schedule created may be affected significantly by the scheme used.

Thus, following the example for the DAG shown in Figure 3.2(a), the computed values of W_i and R_u for each task are displayed in Figure 3.2(b). For instance, for task V_2, $\overline{W(v_2)} = (13+10+8)/3 = 10$ and $R_u(v_2) = 10 + max(7 + R_u(v_6), 11 + R_u(v_7)) = 54$. Once the set of ranks R_u are computed for all tasks, then the list of unfinished tasks ordered by R_u is determined. For the DAG shown in Figure 3.2(a), the list is $\{v_0, v_5, v_4, v_1, v_2, v_3, v_6, v_8, v_7, v_9\}$. Since $R_u(v_i)$ is an upward rank computed from the

exits nodes to the entry nodes and it considers the edges in the computation, for a particular precedence constraint $e(v_i, v_j)$, $R_u(v_i) > R_u(v_j)$. Furthermore, its value remains the same over time despite the varying status of each $v_j \in Pred(v_i)$. This means that task ranks are computed just once during the cyclic process.

An interesting issue in this section is related to the fact that a particular task v_i may be faster than some other task v_j on some processor p_x, but slower than the same task on another processor p_y. This is related to the heterogeneity of resources in SHCS. For example, this can be seen for the DAG shown in Figure 3.2(a), for tasks v_0 and v_5. One reason to explain this, is that in the real world, there are some tasks which are more suitable for a particular processor, for instance a task involving data parallelism is more suitable for vector machines, however the same task executed on network of workstations may be not so efficient. This is reflected in the computation costs.

3.2.2 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 Figure 3.3 with a hypothetical case. At RP_n, 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_4 after receiving 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_4 to p_2, expecting to be executed at some point before RP_{n+1}. Thus, data from $Pred(v_3)$ must be totally retransmitted to p_2. At RP_{n+1} 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 $n+2$. Now, at $n+2$, we observe that v_3 is finally executed after receiving the required data from $Pred(v_3)$. We notice that task v_1 sent the data twice to the same processor p_4 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 3.3 we will 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.

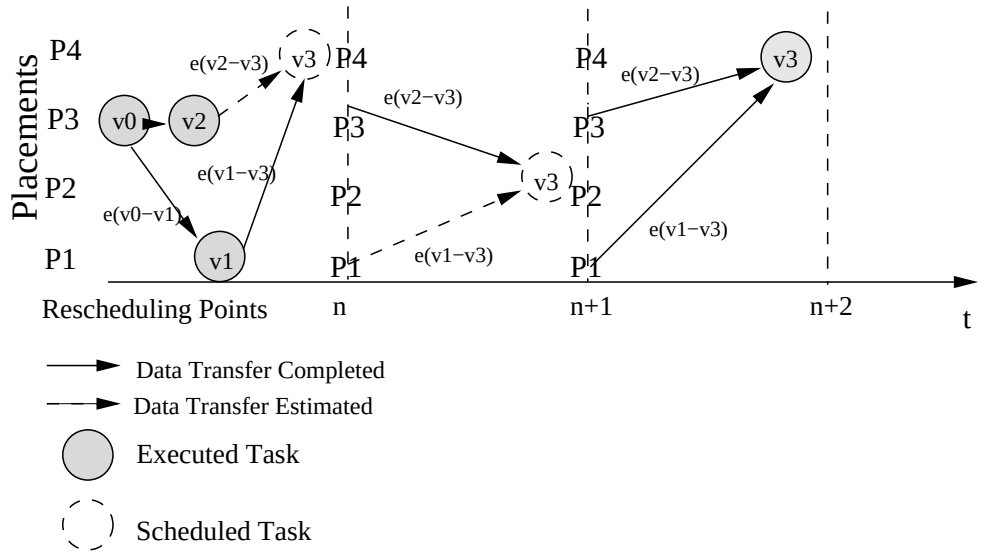


Figure 3.3: The Migration Model in GTP

3.2.3 Costing of Candidate Schedules

Our cost prediction approach is based upon redefinition of concepts drawn from the standard scheduling literature [(Kwok and Ahmad, 1999a), (Topcuoglu, 2002)], together with some additional operations required by the dynamically heterogeneous nature of our target system.

3.2.3.1 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,

$$C^t(v_i, p_m, v_j, p_n) = StartUp + \frac{data^t(v_i, v_j)}{bandwidth^t(p_m, p_n)} \tag{3.3}$$

StartUp is the system dependent fixed time t_s 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 (including the migrated tasks as we explained in Section 3.2.2),

$$StartUp = t_s, \ \texttt{if} \begin{cases} (\kappa^d(v_i, v_j) = 0 \ and \ p_m \neq p_n) \ \texttt{or} \\ (\Pi(v_i) \neq p_m) \ \texttt{or} \\ (\Pi(v_j) \neq p_n) \end{cases} \tag{3.4}$$

being 0 otherwise. Equation 3.4 captures the situation in which a non-zero start-up cost is counted: either the transfer has not yet begun, or the source and/or destination are not those previously associated with the transfer (i.e., a migration has occurred). $data^t(v_i, v_j)$ denotes the *remaining volume* of data to transmit from task v_i to task v_j at time t and is computed as

$$data^t(v_i, v_j) = data(v_i, v_j) * (1 - \kappa^d(v_i, v_j)) \tag{3.5}$$

3.2.3.2 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. $EST^t(v_i, p_m)$ 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, EST will be t (the effect of already completed work will be allowed for in EFT).

$$EST^t(v_i, p_m) \quad = \quad t, \ \texttt{if} \begin{cases} \mu^t(v_i) = p_m \ \texttt{and} \\ \kappa^c(v_i) > 0, \end{cases} \tag{3.6}$$

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

$$EST^t(v_i, p_m) = max\{PA^t(p_m), DA^t(v_i)\} \tag{3.7}$$

where $PA^t(p_m)$ 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.

$$PA^t(p_m) = max_{\{v_i \mid (\mu^t(v_i)=p_m)\}}\{EFT^t(v_i,p_m)\} \tag{3.8}$$

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

$$DA^t(v_i) = max_{v_j \in Pred(v_i)}\{EFT^t(v_j,p_k) + C^t(v_j,p_k,v_i,p_m)\} \tag{3.9}$$

The max block in equation 3.9 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 $v_j \in Pred(v_i)$. Similarly, $EFT^t(v_i,p_m)$ is the *Estimated Finished Time* of the computation of task v_i on processor p_m. For already completed tasks (at t) we will have

$$EFT^t(v_i,p_m) = t, \texttt{if } \kappa^c(v_i) = 1, \tag{3.10}$$

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

$$EFT^t(v_i,p_m) = EST^t(v_i,p_m) + W^t(v_i,p_m) \tag{3.11}$$

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

$$W^t(v_i,p_m) = \frac{W(v_i,p_m) * (1-\kappa^c(v_i))}{avail^t(p_m)} \tag{3.12}$$

As with communication cost prediction, migrated tasks must be costed for a restart from scratch (i.e., we reset $\kappa^d(v_i,v_j) = 0$). We note that our model ignores possible contention in communication by effectively assuming an infinite number of links from p_m to p_n (the assumption is implicit in the *max* in equation 3.9). We are aware that this assumption may affect the predictions of the schedules generated by our models. In the literature we find some scheduling methods [(Sinnen et al., 2006), (Sinnen and Sousa, 2005), (Agarwal et al., 2006)] which consider traffic contention in their scheduling decisions. The discrepancy between real and predicted times is incorporated into our rescheduling as a result of the difference between actual completion information (κ^c, κ^d) returned by monitoring, and that which would have been expected at the preceding RP. Thus, the overall objective of minimising the real makespan of the DAG application is achieved by minimising iteratively the estimated makespan.

3.2.4 Scheduling in the GTP Model

The procedure of the GTP model is as follows. The GTP model has three main parts: the first part concerns *the generation of the initial schedule*, given the initial task graph

in ITG and the initial resource performance information in *SRP*. We consider that the initial schedule is generated by a standard static mapping method and launched to the SHCS. The second part concerns *the cyclic use of the mapping method* which has two principal phases. The first phase is *the computation of task ranks* (priorities) for each task. This phase is performed just once before starting the execution of the application. After this, in keeping with the principles of list-scheduling, we maintain a list of unfinished tasks ordered by *rank* and updated at each RP (removing finished tasks). The rank determines the order in which tasks are assigned to processors. The second phase is *the costing of candidate schedules* which selects each unfinished task from the list according to its rank (priorities). Then, for each such task v_i, *GTP* computes the estimated *Earliest Finished Time* (EFT) value for scheduling v_i to all processors $p_i \in P$ and remaps v_i onto that processor which offers the smallest EFT. The refined task schedule generated at time t is represented as a function $\mu^t : V \rightarrow P$ such that $\mu^t(v_i) = p_m$ denotes that task v_i is to be executed by processor p_m at t. Notice that for *placed* tasks v_i, which have not been migrated by μ^t, we will have $\mu^t(v_i) = \Pi(v_i)$. Finally, the third part concerns *updating* the latest information about both the performance of resources and the progress of the application into the *SRP* and STG structures respectively. The second and third components are iterated at RPs, in response to dynamic changes in resources. The cycle continues until the application finishes execution.
To illustrate the procedure of the *GTP* model, we will the use the task graph, the target architecture and the static information about the tasks on processors from Figure 3.2. Thus, at time $t = 0$ the information concerning the resources is initialized in SRP (see Figure 3.4(a)) and the initial information about the application in ITG is copied to STG (see Figure 3.4(b)). The next step is to generate the initial schedule, which in this case is generated by using the HEFT algorithm, as shown in Figure 3.4(c). Then, the initial schedule is launched to the SHCS. We assume a fixed RP of 14, which means that every 14 time units, a remapping of the application is considered. Note that in Section 4.2 we discuss how the fixed length of the rescheduling points was chosen.

Following the procedure, at the RP at $t = 14$, *SRP* is updated with the latest performance of resources shown in Figure 3.5(a), where we observe the resource changes in either processor availability or bandwidth, which occurred from the period of time between $t = 0$ and $t = 14$. $l1(p_1, p_3)$ varied in bandwidth (from 1 to 0.7) and processor p_1 varied in availability (from 100% to 70%) and P_2 (from 100% to 40%). The progress of the application in STG is shown in Figure 3.5(b). We observe that v_0 has finished execution, v_4 is being executed and the data transfers for $e(v0, v1)$ and $e(v0, v5)$ have

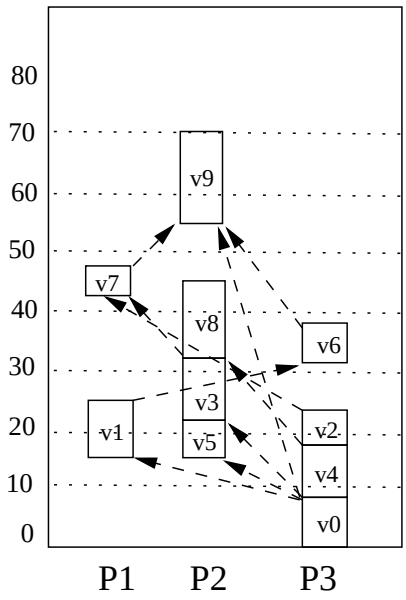
Resource	Bandwidth	Avail (%)
p1	−	100
p2	−	100
p3	−	100
L(p1−p2)	1	−
L(p1−p3)	1	−
L(p2−p3)	1	−

a) SRP structure at t = 0

Task/Comm.	PE	weight	advance
V0	3	9	0
V5	2	7	0

Comm.	link	weight	advance
data(v0−v5)	(p3−p2)	4	0

b) STG structure at t = 0



c) Gantt Chart at time = 0

Figure 3.4: Example (t=0) of the GTP System

been completed.

Then, given the progress of tasks, the list of unfinished tasks is updated and given the latest information about the performance of resources, the model reschedule the application in response to variability in resources. Thus, following the costing of candidate schedules, in which tasks are assigned to that processor which offers the minimum Earliest Finish Time, the refined scheduled is generated and shown in Figure 3.5(c) where we observe that the new estimated makespan of the application is 94.78 units of time.

We note that the task v_5 migrated from p_2 to p_1 as p_1 offered the minimum earliest finish time. This action requires the retransmission from the very beginning of data from v_0 to v_5. Obviously, the migration model will tend to generate migrations only when the benefits are substantial, and will reduce the potential for schedule thrashing. Therefore, in terms of our formalization, a migrated task becomes *non-placed* until it starts to gather inputs again. For costing purposes, this means that for a migrated task v_i, we must reset $\kappa^c(v_i) = 0$ (the computation must restart) and $\kappa^d(v_j, v_i) = 0, \forall v_j \in Pred(v_i)$ (all communications to v_i must restart).

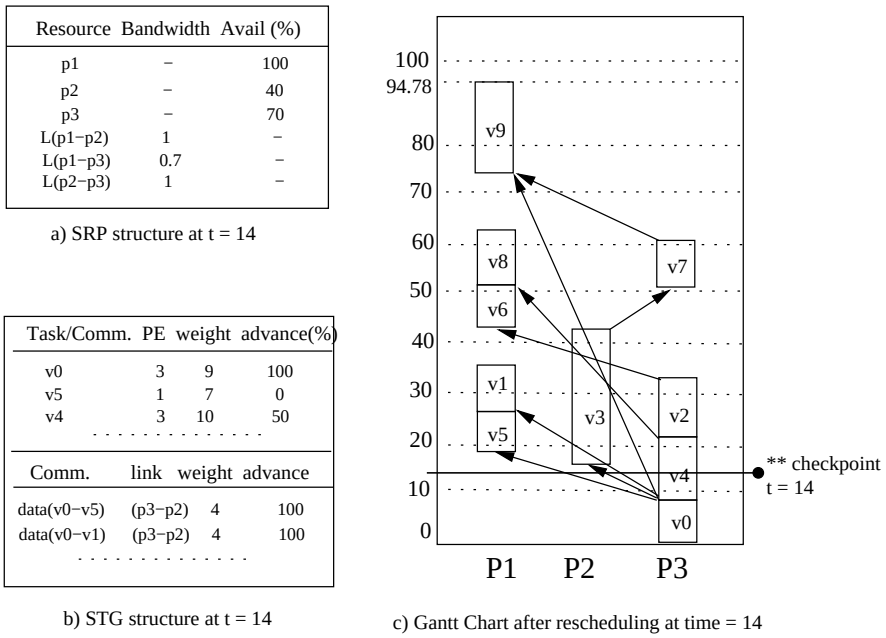


Figure 3.5: Example (t=14) of the GTP System

3.2.5 Time Complexity Analysis for the GTP model

The time complexity analysis is centered in *the cyclic use of the mapping method* part which involves two main phases: *the computation of task ranks* and *the costing of candidate schedules* (see Section 3.2.4). 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 processor 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 cyclic use of the mapping method for each cycle is of the order of $O(v^2 \times p)$. We will report on actual times for real examples in Chapter 5.

3.3 Description of the GTP/c Model

The *GTP* model described in the previous section addressed the dynamically heterogeneous nature of SHCS by allowing rescheduling and migration of tasks when this helps to minimize makespan. In *GTP*, a migrated task had to be restarted from the very be-

ginning, including regathering all inputs directly from its predecessors. Obviously this 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 overall GTP/c system is sketched in Figure 3.3 in which we consider 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.

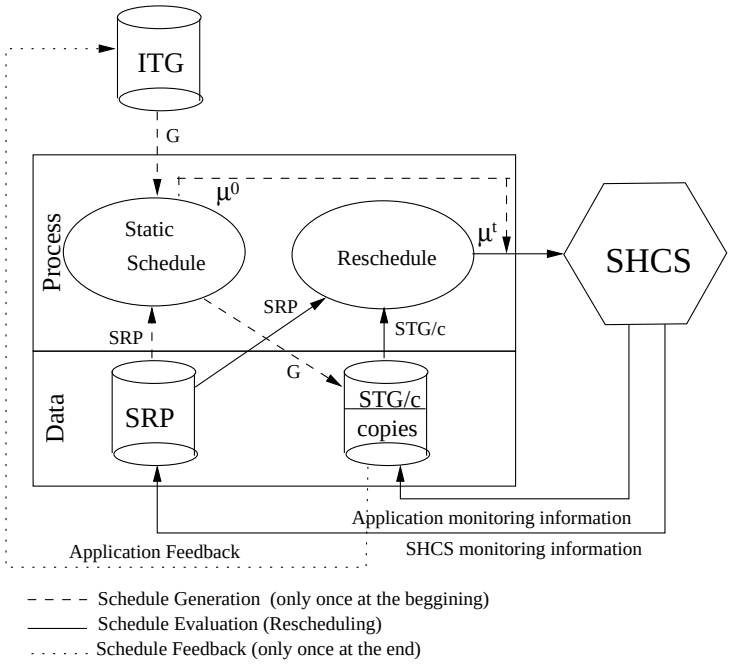


Figure 3.6: The GTP/c System

3.3.1 Definition of the SHCS

We will take the same definition and assumptions from the GTP model described in Section 3.1.1 to represent Shared Resource Pools (*SRP*) in GTP/c. We notice that GTP/c assumes sufficient storage to maintain the reusable copies on each processor. A natural danger in real environments is that when the application has a relatively high number of tasks and data transfers, the copies could overwhelm existing data storage.

A complementary work can be found in [(Ramakrishnan et al., 2007)], which considers data-storage constraints when scheduling data intensive applications. Their approach is two-fold: they minimize the amount of space a DAG application requires during execution by removing results of completed tasks (i.e., output files) at runtime when they are no longer required and they schedule the application in a way that assures that the amount of data required and generated by the application fits onto the individual processors.

3.3.2 The Situated Task Graph with Copying (STG/c)

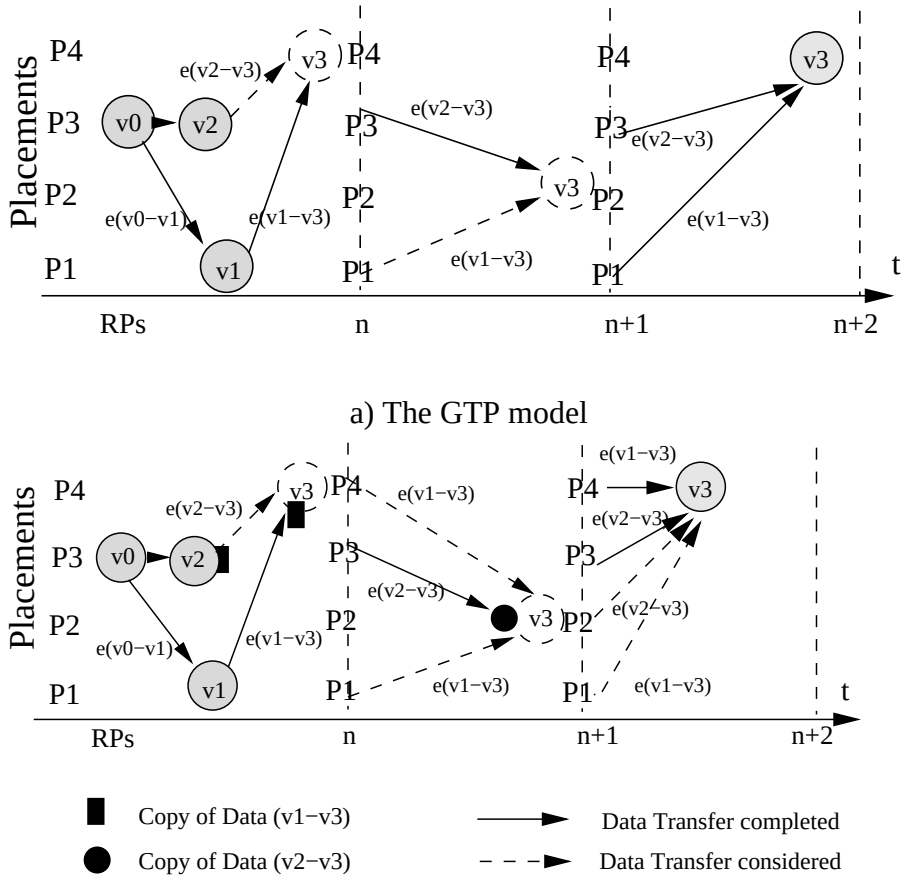
We extend the definition of the Situated Task Graph structure defined in 3.1.3 as $STG/c :: (V, E, data, W, \Pi, \kappa^c, \kappa^d, \Omega)$, where the first seven components are taken directly from the previous definition of STG. The key new concept is that of *reusable copy*. A data transfer for a particular edge $e(i, j)$ is said to become *reusable copy* on a processor once it has been totally transmitted ($\kappa^d(e(i, j)) = 1$) 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 task 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 RP, if the benefits are substantial. We hope 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 $\Omega_k :: E \rightarrow \mathcal{P}(P)$ to describe the subset of P where copies of the given (edge) data are available at time k.

3.4 The *GTP/c* Scheduling Method

In this section we describe the GTP/c method. As GTP/c is an extension of GTP, it has the same three main parts: *the generation of the initial schedule*, *the cyclic use of the mapping method* and at each cycle, *updating* the latest information about both the performance of resources and the progress of the application into the *SRP* and STG structures respectively.

3.4.1 Setting Task Ranks

We keep the same process to set the task ranks (priorities) from the GTP model described in Section 3.2.1.



a) The GTP model

b) The GTP/c model

Figure 3.7: The GTP/c Migration Model

3.4.2 The Task Migration Model in GTP/c

The adaptive nature of the GTP/c model is illustrated in Figure 3.7 where we can observe the difference of strategies used between GTP and GTP/c. In terms of our formalization a placed task v_i is migrated when it has been rescheduled onto a processor other than $\Pi(v_i)$. 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 Figure 3.7(a)). Now, with GTP/c, 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. For instance, in Figure 3.7(b), at RP_n, 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 $\Omega_n(e(v_1,v_3))$, such that it may be used as an input in future migrations for v_3. Thus, at RP_n 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_{n+1}, 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 $\Omega_{n+1}(e(v_2,v_3))$ for future migration for v_3. At RP_{n+1} task v_3 is now migrated from p_2 to p_4, and we observe that there are several possible sources for each preceding task. At the end we observe that v_3 is finally executed, using the copy $\Omega_n(e(v_1,v_3))$ and a direct data transfer for $e(v_2,v_3)$.

3.4.3 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 definitions 3.3, 3.4 and 3.5 for the GTP model, to estimate the communication cost in standard units.

The *Copying Management*(CM) function defined in equation 3.13, will return the minimum data transfer cost for data associated with $e(i,j)$ to $\mu^t(v_j)$. Thus, for a particular $e(i,j)$, CM evaluates the locations (processors) for each reusable copy in $\Omega_t(e(i,j))$ and together with the latest bandwidth of the links involved, returns the minimum data

transfer cost to $\mu^t(v_j)$.

$$CM^t(v_i, v_j) = \\ min_{p \in \Omega_t(e(i,j))}\{C^t(v_i, p, v_j, \mu(v_j))\} \tag{3.13}$$

3.4.4 Estimating Computation Cost

We retain the equations 3.6, 3.7 and 3.8 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 existing copies which will certainly affect the beginning execution of tasks. Thus, we have redefined the equation 3.9 such that, now the new equation 3.14 will compute the estimated earliest time at which data from a predecessor task v_j (mapped on $\mu^t(v_j)$ and any available copies of their results) will be available at p_m.

$$DA^t(v_i) = max_{v_j \in Pred(v_i)}\{EFT(v_j, p_k) + CM^t(v_j, v_i)\} \tag{3.14}$$

In the same manner, we need to predict the time at which that execution will finish. For this, we retain equations 3.10, 3.11 and 3.12 for GTP.
As before, migrated tasks must be costed for a restart from scratch (i.e., we reset $\kappa^d(v_i, v_j) = 0$) and GTP/c ignores possible contention in communication by assuming an infinite number of links from p_m to p_n.

3.4.5 Procedure of the GTP/c Model

The procedure of the GTP/c model is the same as that followed by the *GTP* model described in Section 3.2.4 with the introduction of copies allowing for more flexibility in scheduling.
To illustrate the procedure of the GTP/c model, we will extend the example of Section 3.2.4, in which we used the tasks graph, the target architecture and the initial static information from Figure 3.2 to follow the procedure of the *GTP* model. Thus, at time $t = 0$, the first part related with *the generation of the initial schedule* is the same as *GTP* where the initial schedule is generated by using the HEFT algorithm and shown in Figure 3.2(d), followed by their launch to the SHCS. Then, at time $t = 14$, GTP/c uses the same sequence of resource changes as that used in *GTP*, which is updated in the *SRP* structure (see Figure 3.8(a)). The progress of the application is shown in

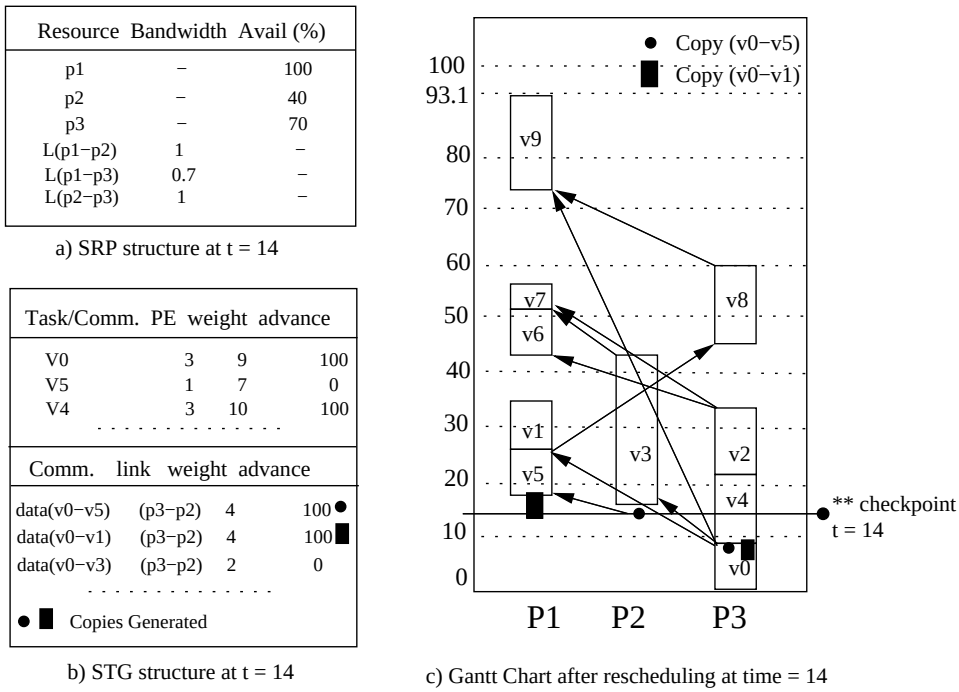


Figure 3.8: Example of The GTP/c System

Figure 3.8(b) where we observe that v_0 has finished execution, v_4 is being executed and the data transfers for $e(v_0, v_1)$ and $e(v_0, v_5)$ have been transmitted. According to our formalization, two copies have been generated at this point, one of $e(v_0, v_5)$ and one of $e(v_0, v_1)$. These copies can be used in future migrations for v_5 and v_1. Following the procedure of GTP/c, given the progress of the application and given the latest performance information about the performance of resources, GTP/c reschedules the application obtaining a new refined schedule shown in Figure 3.8(c). In this schedule we observe that task v_5 migrated from p_2 to p_1, and the model decided to use the copy of $e(v_0, v_5)$ located on p_2, as it allowed the minimum earliest finish time for task v_5, to retransmit the required data for $e(v_0, v_5)$. The benefit of such decision is reflected in the estimated makespan which is now 93.1 units of time (1.80% better than the 94.78 units of time computed for the estimated makespan for GTP shown in Figure 3.4(c)).

3.4.6 Time Complexity Analysis for the GTP/c Model

We focused on time complexity analysis of the *cyclic use of the mapping method* part which involves two main phases: *the computation of task ranks* and *the costing of candidate schedules* (see Section 3.2.4). As before, 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 $O(v \times logv)$. The costing of candidate schedules which selects a task v_i from the list and computes the earliest finish time value for scheduling v_i to all processors, for which it is considered that the results (copies) of some tasks $v_j \in Pred(v_i)$ can be stored in others sites, takes $O(e \times p)$ for e edges and p processors. 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 $O(v^2 \times p)$. Then the time complexity for the cyclic use of the mapping method for each cycle is on the order of $O(v^2 \times p)$.

3.5 Reliable DAG Scheduling with Rewinding and Migration

In the literature we can find some mapping methods to execute DAG applications on SHCS. However, most of them (including GTP and GTP/c) are not able to react to extreme variations (i.e., processor failure) in some of the processors. Effective DAG scheduling methods for SHCS must include fault tolerant mechanisms to preserve the execution of DAG applications, despite the presence of resource failure. To address this, we designed the rewinding mechanism, an event-driven process executed when a failure is detected at some checkpoint (see Figure 3.9). 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 reactive mapping methods. At the end of the section we define some metrics to evaluate the performance of such mechanism.

Fault tolerance (as reviewed in Section 2.8) is an important issue in SHCS as the availability of resources can not be guaranteed. The presence of a resource failure in a particular processor p_m during execution at time t, may disrupt the subsequent execution of other tasks. The tasks expected to be disrupted 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.

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

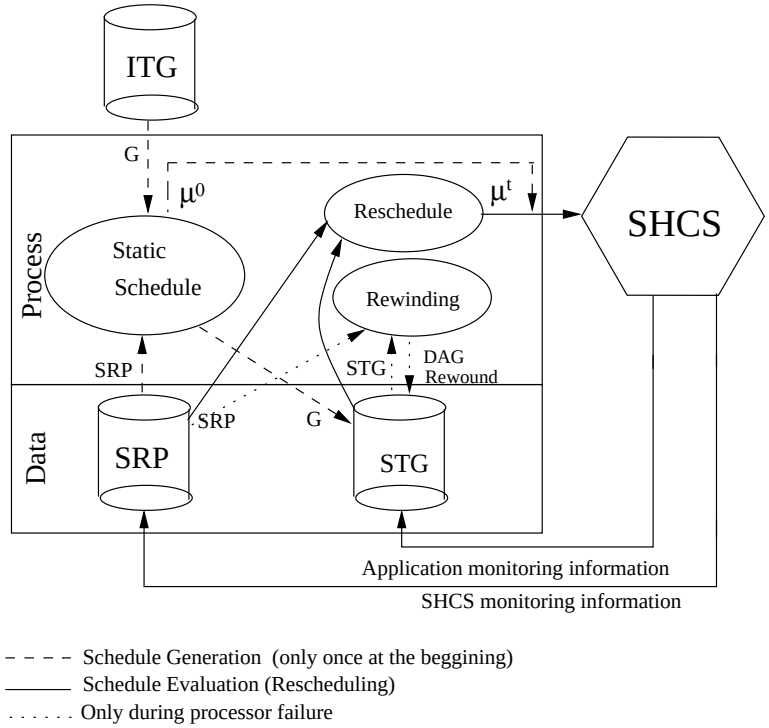


Figure 3.9: The Rewinding Mechanism

replication and so on. Thus, 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.

3.6 The GTP System with Rewinding (GTP/r)

We recall that the GTP system defined in Section 3.2 allows rescheduling and migration of tasks in response to variations in the performance of resources. The inclusion of the rewinding mechanism into GTP produces the GTP/r version.

3.6.1 Definition of the SHCS

We have to identify key information related with the processors availability used by the model. The dynamic information about resources in SHCS has been defined in Section 3.1.1, in which Shared Resource Pools (SRP) are represented with graphs $SRP :: (P, L, \text{avail}, \text{bandwidth})$ where P is the set of available processors in the system, $p_i (1 \leq i \leq |P|)$. L is the set of communication links connecting pairs of distinct processors, $l_i (1 \leq i \leq |L|)$ such that $l(m,n) \in L$ denotes a communication link between p_m and p_n. The decision to rewind the application will be based upon the latest available resource performance information (as returned by standard Grid monitoring tools such as NWS or Globus MDS). Thus, at time t we assume knowledge of $avail^t :: P \rightarrow [0..1]$, capturing the availability of each CPU. Failure in some processor p_m occurs when the latest $avail^t(p_m) = 0$. Then, at each RP, if a failure is detected then the rewinding process will be triggered to rewind the application. We note that failures in traditional distributed systems are mostly linked to physical failures which make the resources unavailable. However, in our context, in which resources are shared and autonomous, a failure embraces other situations, which affect the availability of resources. For instance, during the execution of the DAG application, we may have the case, outside our scheduler's control, in which a particular processor is assigned to another job with higher priority.

3.6.2 Definition of the Situated Task Graph (STG)

Just as we use dynamic information about resources on *SRP* to take decisions about when to rewind the application, so we must identify the dynamic information related to the progress of the DAG application to determine which tasks will be rewound. The information related to the progress of the tasks has been defined in Section 3.1.1 where we defined $STG :: (V, E, data, W, \Pi, \kappa^c \kappa^d)$. V is the set of tasks, $v_i (1 \leq i \leq |V|)$. $E \subseteq V \times V$ is the set of directed edges connecting pairs of distinct tasks, $e_i (1 \leq i \leq |E|)$, where $e(v_i, v_j) \in E$ denotes a precedence constraint and data transfer from task v_i to

task v_j. We use $data :: V \times V \rightarrow Int$ 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. $W :: V \times P \rightarrow Int$, indicates the heterogenous characteristics of the processors composing a SHCS where $W(i,m)$ denotes the execution time in standard units of task v_i on processor p_m. $\Pi :: V \rightarrow P^+$ represents 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, $\Pi(v_i) = NONE$. For future convenience, we define $Q^t :: P \rightarrow \mathcal{P}(V)$ 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 time t, we must perform the following operations on the STG data structure.

1. $\forall v_j \in SUCC(v_i)$ *set* $\kappa^d(v_i, v_j)$ *to* 0
2. $\forall v_k \in PRED(v_i)$ *set* $\kappa^d(v_k, v_i)$ *to* 0
3. *Set* $\kappa^c(v_i)$ *to* 0
4. *Set* $\Pi(v_i)$ *to* $NONE$

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.

3.6.3 The GTP System with Rewinding (GTP/r)

In this section we define the GTP system with rewinding (GTP/r) to preserve the execution of a DAG application despite the failure of a particular processor p_m during the process. $Q^t(p_m) = \{v_0, v_1, v_2, ..., v_k\}$ 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 $v_i \in Q^t(p_m)$. Intuitively, v_i must be rewound if either.

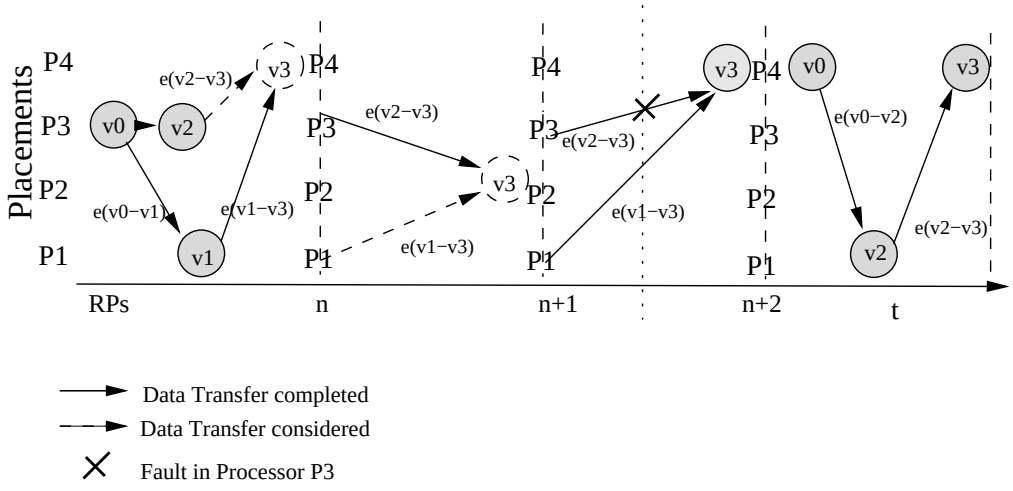


Figure 3.10: The Rewinding Mechanism for GTP

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^t in an order which respects a reverse topological sort (according to the precedence constraints between tasks). Thus, within $Q^t(p_m)$ 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 $v_i \in Q^t(p_m)$ must be rewound if,

1. $\exists e(v_i, v_j) \in E : \kappa^d(v_i, v_j) < 1$, *or*

2. $\exists v_k \in SUCC(v_i) : v_k \in Q^t(p_m)$ *and* v_k *must be rewound*

Note the importance of maintaining information about all placed tasks in Q^t, 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 $avail^t(p_m) > 0$ at future RP's.

To illustrate the rewinding mechanism, we extend the example of Figure 3.3 by adding a failure in processor $p3$ before finishing the execution of the DAG application at some

point between RP_{n+1} and RP_{n+2} as shown in Figure 3.10. 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_{n+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_{n+2}, $Q^{n+2}(p1) = \{v1\}$, $Q^{n+2}(p3) = \{v0, v2\}$ and $Q^{n+2}(p4) = \{v3\}$. Then, the rewinding mechanism will evaluate in reverse order the sequence of each placed task $v_i \in Q^{n+2}(p3)$. 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. Then, v_2 is rewound as explained above. Now, the next task to evaluate from $Q^{n+2}(p3)$ is v_0, which $Succ(v_0) = \{v1, v2\}$, then for the first precedence constraint $e(v0, v1)$ is satisfied as v_1 has finished its execution at p_1. However, when evaluating the second precedence constraint $e(v0, v2)$ we observe that it is not satisfied as v_2 (already rewound) will not be able to retrieve their input from v_0 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 inconsistences in scheduling decisions. After rewinding and rescheduling the application at RP_{n+2}, the task v_3 was finally executed at p_4 after receiving the required inputs.

3.7 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.

3.7.1 Definition of SRP and STG

Our definition of *SRP* (Shared Resource Pools) and *STG* (Situated Task Graph) are identical to those from the GTP/c system defined in Section 3.3. In particular we remember $\Omega :: E \rightarrow \mathcal{P}(P)$ to capture information on location of copies which can be used as source and Q^t captures information on tasks placed on each processor.

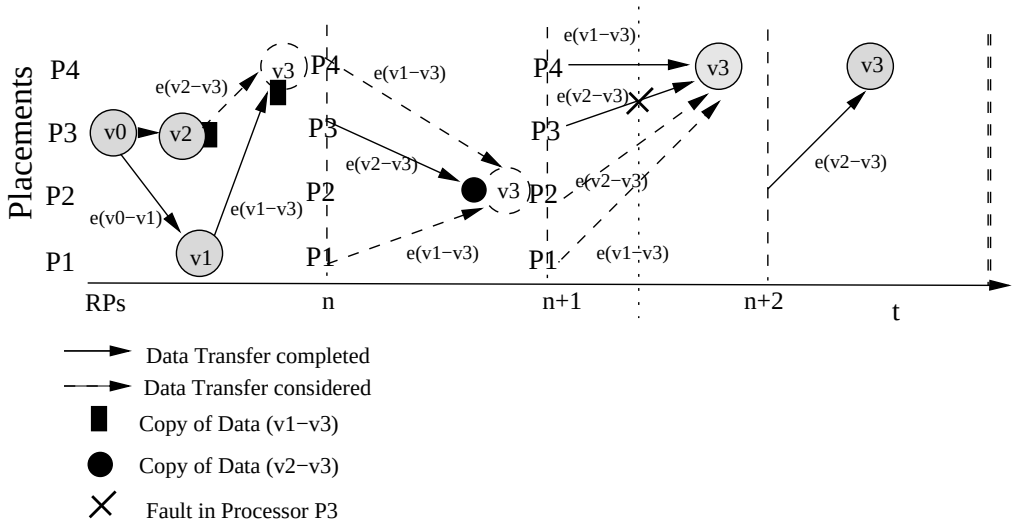


Figure 3.11: The Rewinding Mechanism for GTP/c

3.7.2 Procedure of the GTP/c/r Model

The rewinding mechanism for $GTP/c/r$ is similar to that for GTP/r. In the same manner, the placed tasks $v_i \in Q^t(p_m)$ 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 $\kappa^d(v_i, v_j) < 1$. However, now we have a second criterion to be met related to the existence of possible reusable copies in $\Omega(e_{i,j})$ 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 $v_i \in Q^t(p_m)$ must be rewound if,

1. $\Omega(v_i) = \{p_m\}$, $(this\ is\ the\ only\ copy)$, *and either*
2. $\exists (v_i, v_j) \in E : \kappa^d(v_i, v_j) \leq 1$, *or*
3. $\exists v_k \in SUCC(v_i) : v_k \in Q^t(p_m)$ *and* v_k *must be rewound*

As before, for tasks to be rewound, we must reset elements of κ^d, κ^c and Π 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 $\omega^t(e_{i,j}) = p_m$ must be eliminated from STG.
To illustrate the rewinding mechanism for $GTP/c/r$, we will use the same case as for GTP/r with the same failure in processor $p3$ at some point between RP_{n+1} and RP_{n+2}. This is shown in Figure 3.11. At RP_{n+2}, $Q^{n+2}(p1) = \{v1\}$, $Q^{n+2}(p3) = \{v0, v2\}$ and $Q^{n+2}(p4) = \{v3\}$. Then, the rewind mechanism will evaluate in reverse order the sequence of each placed task $v_i \in Q^{n+2}(p3)$. 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 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 $Succ(v_0) = \{v1, v2\}$. 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 inconsistence in future decisions. In this case, the copy $\Omega_l(v_2, v_3)$ stored at p_3 must be deleted from the system as it can lead to inconsistences in the scheduling decisions in the case that task v_3 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_{n+2}. Completing the example, after rewinding and rescheduling the application at RP_{n+2}, v_3 was finally executed at p_4 after receiving the required inputs.

3.8 Summary

In this chapter we defined the proposed reactive scheduling mechanisms to address the dynamically heterogeneous nature of SHCS. We started by defining the Global Task Positioning (GTP) scheduling system, a list-scheduling heuristic based model, which addresses the problem of heterogeneity and dynamism of SHCS by allowing rescheduling and migration of the tasks of an executing application. Next, based on observations of previous results for GTP, we defined the Global Task Positioning system with Copying facilities (GTP/c) which re-use information to improve the utilization of resources and to minimize the impact of the migration cost on the application makespan. Finally, considering that fault tolerance is an important issue in SHCS where the availability of processors cannot be guaranteed, we defined the rewinding mechanism, which pre-

serves the execution of the application, despite the presence of a processor failure. Unlike other fault tolerant approaches, our mechanism preserves the execution of the application by recomputing and migrating those tasks which will disrupt the forward execution of succeeding tasks. We showed how to integrate the rewinding mechanism into GTP and GTP/c.

Chapter 4

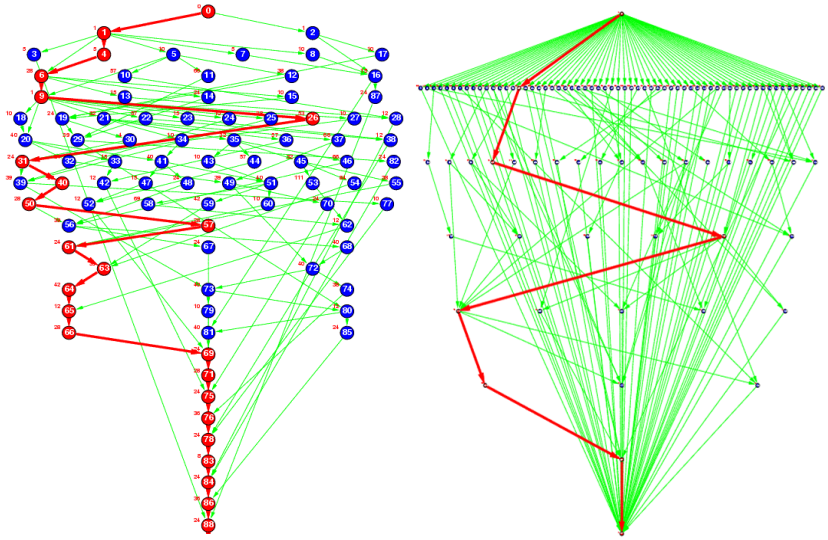
The Simulation Framework

The evaluation of our reactive scheduling mechanisms is conducted by simulation, since this allows us to generate repeatable patterns of resource performance variation. In this chapter we describe all the elements involved in the simulation framework in which we conducted the evaluation. We start describing the source and characteristics of the Input Task Graphs (ITGs) used in the evaluation. Then, remembering that SHCS is dynamic, we describe the distinguishing characteristics of our scenarios. Next, we explain the criterion used to define the fixed rescheduling points used to reschedule the application. The next elements that we describe are the metrics used to evaluate the performance of GTP, GTP/c, GTP/r and $GTP/c/r$. Finally, we describe the Simgrid software used to perform the evaluation. We describe the difficulties that Simgrid presented to manage dynamic events in simulating variations in the performance of resources. To address this problem, we designed a tracking mechanism, built on top of Simgrid, which allows changes in resource performance characteristics over time, as observable in real dynamic resources.

4.1 The Directed Acyclic Graphs (DAGs)

In this section we present the Input Task Graphs (ITGs) used to evaluate our dynamic mapping methods. In the literature, different research groups use their own methods to determine the shape and size of the DAGs used to evaluate their mapping methods [(Topcuoglu, 2002), (Zhao and Sakellariou, 2004b), (Shi and Dongarra, 2006)]. This complicates the process of benchmarking the mapping methods designed by different researchers. In [(STG, 2000)] we found the Standard Task Graph (STDGP) Project, an effort to define a set of standard DAGs for fair evaluation of mapping algorithms.

The STDGP consists of two main sets. The first set contains a small set of four DAGs modelled from actual application programs. For instance, in Figure 4.1(a) we see a task graph for Newton-Euler dynamic control calculation for the 6-degrees-of-freedom Stanford manipulator [(Kasahara and Narita, 1985)]. Figure 4.1(b) represents a task graph for a random sparse matrix solver of an electronic circuit simulation that was generated using a symbolic generation technique and the OSCAR FORTRAN compiler [(Kasahara et al., 1991)].



a)Newton−Euler graph application b) Sparse−matrix graph application

Figure 4.1: DAGs for Particular Applications

The second set, which we used, contains a considerable number of task graphs (900 graphs) generated randomly. The graph size (in number of tasks) varies between 50 and 2700. The graph shapes were determined based on four different methods [(Almeida et al., 1992), (Yang and Gerasoulis, 1994), (Adam et al., 1974a)].

Before explaining the characteristics of each method, we note that, in terms of our formalization in Section 3.1.2, a given initial DAG $ITG = (V, E, data, W)$ can be represented in two different ways (see Figure 4.2). The first approach uses an adjacency matrix, where the i^{th} node is represented as i^{th} row and i^{th} column, an edge from i^{th} to j^{th} is represented as 1 in row i and column j (no edge is represented as 0). The second approach uses adjacency lists. For DAGs (directed acyclic graphs), nodes are arranged as lists of arrays in which each node stores the succeeding nodes in the graph.

Let A denote an adjacency matrix with elements $a(i, j)$, where $0 \leq i, j \leq n+1$ denote tasks (0 is the entry dummy node and $n+1$ is the exit dummy node). Next we explain the four methods used by the STDGP project to create the DAG graphs.

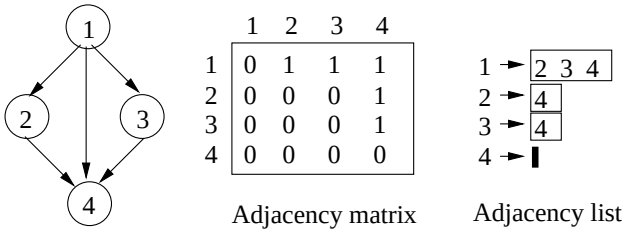


Figure 4.2: DAGs representation

1. The first method to determine the shape of the DAG is 'sameprob' [(Almeida et al., 1992)], in which the creation of an edge is determined by independent random values defined as follows,

$$P[a(i,j)=1]=\pi \; for \; 1 \leq i < j \leq N \tag{4.1}$$

$$P[a(i,j)=0]=1-\pi \; for \; 1 \leq i < j \leq N \tag{4.2}$$

$$P[a(i,j)=0]=1 \; if \; i \geq j \tag{4.3}$$

   The parameter π indicates the probability that there exists a direct dependency (edge) between task i and task j. Equation 4.1 shows that the density of precedence relations between tasks is determined by the value of π and equation 4.3 indicates that the structure of the graph is acyclic. The value of π in [(Almeida et al., 1992)] was unique, however in the STDGP project we observe that it is considered as a range of values. Figure 4.3(a) shows a random task graph with 50 tasks generated by the 'sameprob' method, where π was set in STDGP as 0.1 and 0.2. With the *sameprob* method, the number of precedence relations increases as the number of tasks increases.

2. The second method to determine the shape of the DAG is the 'samepred' method which specifies the average number of predecessors for each task. Figure 4.3(b) shows a random task graph with 50 tasks generated by the 'samepred' method, where the average number of precedence relations was specified as 3. Currently, the average number of precedence relation is set to 1,3 or 5.

3. The third method is the 'layrprob' method [(Yang and Gerasoulis, 1994)], which first randomly generates the number of levels (layers) in the task graph. Next,

the number of independent tasks in each level is randomly set. Finally, edges between tasks are connected randomly at different layers.For this particular method, the author notes the importance that the shape of the DAGs may have in the evaluation of scheduling approaches. Thus, they consider the following statistical information to create the DAGs,

(a) The range of independent tasks in each layer, which approximates the average degree of parallelism.

(b) The number of layers, and

(c) The average ratio of task weights over edge weights, which approximates the graph granularity.

In keeping with the consistency of the previous methods, the STDGP project uses the same probability π as with the 'sameprob' method to determine a direct dependency (edge) between tasks. Figure 4.3(c) shows a random task graph with 50 tasks generated by the 'layrprob' method, where the number of layers was specified as 5 and the value of π was specified as 0.2. Currently, the average number of tasks in each layer is fixed to 10, and the number of layers is calculated as $(number\ of\ tasks)/10$.

4. The last method is the 'layrpred' method which generates levels(layers) in the same manner as with 'layrprob', with the mechanism to connect edges as 'samepred'. Figure 4.3(d) shows a random task graph with 50 tasks generated by the 'layrpred' method, where the number of layers was specified as 5 and the average number of predecessors was specified as 5. Currently, the average number of tasks in each layer is fixed to 10, and the number of layers is calculated as $(number\ of\ tasks)/10$, with the average number of predecessors set to 1,3 or 5.

For our experiments we extracted from *STDGP* a sample of DAGs to be used as an input into our model. We first defined the range of the size (in number of tasks) of the DAGs to be used. The size of the DAGs is 50,100,300,500 and 1000 tasks. Then, for each size, we selected randomly (from the 900 DAGs in STDGP) up to 3 DAGs for each of the creation methods. Thus 12 DAGs were used for each size giving a total of 60 DAGs. The STDGP project makes a pair of assumptions which limit the applicability of the DAGs graphs for evaluating mapping methods for SHCS. The first

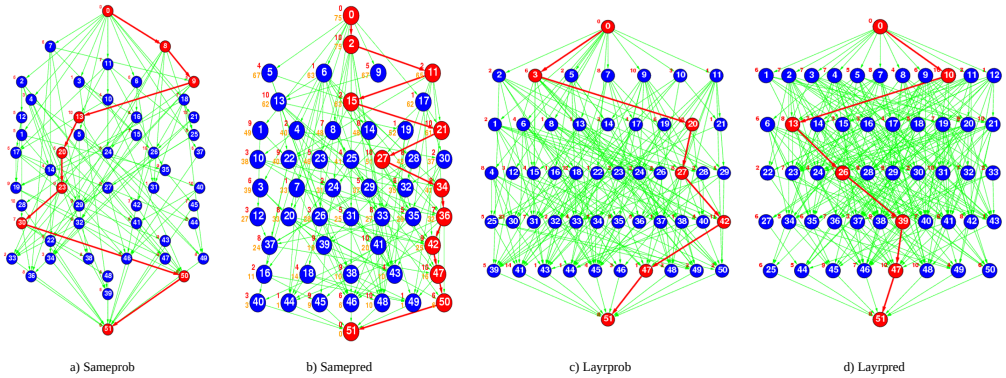


Figure 4.3: Random DAGs in the STDGP Project

assumption concerns the absence of communication cost among tasks, and the second concerns the assumption that the DAGs will be evaluated in homogeneous environments. To address this, we included a module to generate W and $data$ information to produce our ITGs. Remembering that our processors are heterogeneous, the task computation times for a particular task v_i were created for each processor using uniformly distributed random numbers from the interval [1 to 10]. 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. Notice that, due to the heterogeneous nature of the problem, the computation cost of a node is approximated by the average of its costs across all processors. Thus, for each size of the task graph, we generated three different graphs for CCR equal to 0.1,0.5 and 1.5.

4.2 Setting the Fixed Rescheduling Point

The setting of rescheduling points is an important element of cyclic mapping methods. We use a fixed length rescheduling cycle. Choosing the length of the cycle presents a trade-off. A long cycle will not properly react to dynamic changes. For instance it is important to detect a failure in some of the resources as soon as possible to reduce the impact of the failure on the makespan. Alternatively, a short cycle can increase the number of remappings and migrated tasks lengthening the makespan. Thus, in keeping with the principles of schedule feedback, we assume the availability of the most recent makespan of the application, and set the fixed-period rescheduling cycle at 10% of the value of the makespan. For new DAG applications, we use the HEFT approach

to obtain the initial predicted makespan. Calculating an optimal size for each cycle is a matter for further research. We believe that new efforts to optimize the size of the rescheduling points may improve the makespan of the application.

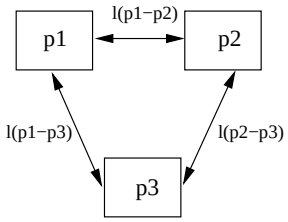
4.3 The Scheduling Scenarios

In this section we describe the characteristics of the scheduling scenarios used to evaluate the performance of our dynamic models. We recall that, in SHCS the availability and performance of computational resources can vary dynamically over time, even during the course of an execution. In order to have a more realistic environment to test our dynamic mapping methods, we included into our scenarios events which simulate a change in the performance of the resource (availability or bandwidth). Thus, considering the nature of our dynamic mapping methods, we created a pair of different groups (TE1 and TE2) of scenarios. The first group (TE1) was used to evaluate the performance of the GTP and GTP/c systems and the second group (TE2) to evaluate GTP/r and $GTP/c/r$. Both TE1 and TE2 contain the same events. The key difference is that we injected in TE2 an additional event simulating a processor failure to occur at the mid-point of the execution. Each scenario is instantiated for 5, 10 and 20 processors and assumes that processors are fully connected.

4.3.1 The Scenarios for GTP and GTP/c

For each scenario, we defined events, each simulating a resource change in either processor or bandwidth availability. Then, we set a bound placed on the maximum variation allowed in one event, expressed as a percentage of the peak performance of a resource. For example, in the 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%. We will refer to a particular scenario as $SCE(x,y,z)$, which means that it involves x processors, y tasks and z percent of variability in resources. Note that $SCE(x,y,0)$ (no dynamic resource variation) refers to a static environment, an approach used by most of the mapping methods in the literature [(Topcuoglu, 2002), (Shi and Dongarra, 2006), (Sih and Lee, 1993), (Kwok and Ahmad, 1999a)]. It allowed us to investigate the extent to which emerging discrepancies between real and predicted behavior are handled by GTP and

GTP/c.

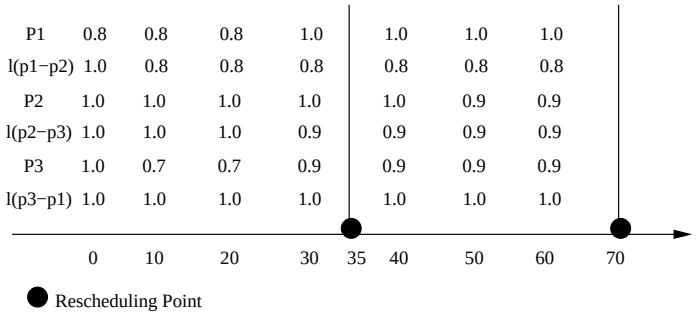


a) SHCS system

Processor	Availability	Time
P1	0.80	0
P3	0.70	10
P1	1.00	30
P3	0.90	30
P2	0.90	50

Link	Bandwidth	Time
l(p1−p2)	0.80	10
l(p2−p3)	0.90	30

b) Sequence of events with a bound of 70%



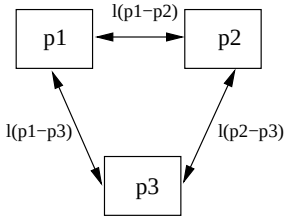
c) Gantt Chart with simulating changes in resources

Figure 4.4: The Scheduling Scenarios for GTP and GTP/c

To illustrate this, consider Figure 4.4 which shows a simulation scenario with a bound of 70% as the maximum variation allowed in one event. We observe that each event occurs at some point of time affecting the performance of the resources involved in the event. The execution time of the tasks or data transfers on the resources involved will be affected. For instance, processor p_3 decreases availability from 1 to 0.7 at $t = 10$ and such availability remains until $t = 30$ where p_3 increases availability from 0.7 to 0.9. The execution time of tasks mapped onto p_3 is affected. In the same manner the link $l(p1-p2)$ decreases in bandwidth from 1 to 0.8 at $t = 10$ and will remain so until the next event involving the link. It is important to note that while our simulation tracks resource variations as they occur (with immediate impact on task execution time), our scheduling algorithms only become aware of variations at rescheduling points, and may not even notice some short-lived variations. For example, during the rescheduling point at $t = 35$ the latest resource performance information will be updated within the model (i.e., the SRP structure). We note that at the rescheduling point at $t = 35$, the availability for p_3 is 0.9 which will be updated in GRP, but the first change at $t = 10$ where p_3 varied from 1 to 0.7 was never updated in GRP, however the simulated execution time of the task(s) being executed at that time will be correctly affected.

4.3.2 The Scenarios for GTP/r and $GTP/c/r$

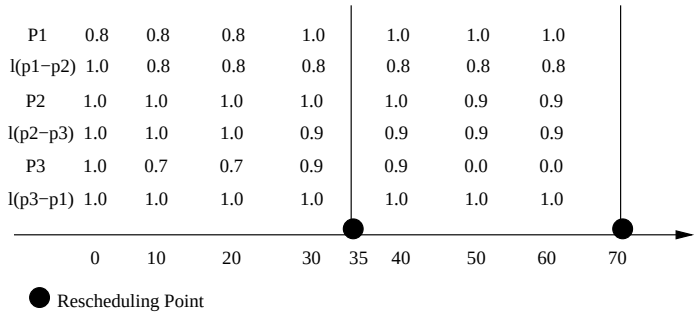
To evaluate the rewinding mechanism integrated within the GTP/r and $GTP/c/r$ systems we created a set of scenarios forming the group $TE2$. Scenarios in this group involve a similar sequence of randomly defined events as $TE1$, each simulating a resource change in either processor or bandwidth availability. The key difference is that TE2 may contain events with availability equal to zero, simulating a processor failure to occur at relatively the mid-point of the execution.



a) SHCS system

Processor	Availability	Time
P1	0.80	0
P3	0.70	10
P1	1.00	30
P3	0.90	30
P2	0.90	50
P3	0.00	50
Link	Bandwidth	Time
l(p1−p2)	0.80	10
l(p2−p3)	0.90	30

b) Sequence of events with a bound of 70%



c) Gantt Chart with simulating changes in resources

Figure 4.5: The Scheduling Scenarios for GTP/r and $GTP/c/r$

Figure 4.5 illustrates this case. It shows the same scenario with a bound of 70% as the maximum variation allowed in one event, but with an event simulating a failure (availability equal to zero) in processor p_3 at $t = 50$. This means that the tasks mapped onto p_3 will stop the execution at $t = 50$ and those tasks mapped onto other processors but still retrieving data from preceding tasks already executed at p_3, will stop retrieving data as the processor becomes unavailable. It will not be until the rescheduling point at $t = 70$ that the resource performance information (GRP) will be updated within the model and the rewinding mechanism will be triggered, after detecting this resource failure.

4.4 Comparison Metrics

In this section we describe the comparison metrics used to evaluate the performance of our dynamic models. We first describe the metrics to evaluate the GTP and GTP/c models and then we describe the metrics for GTP/r and $GTP/c/r$.

4.4.1 Comparison Metrics for GTP and GTP/c

We use the *Normalized Schedule Length (NSL)* [(Kwok and Ahmad, 1996)], also called *Schedule Length Ratio(SLR)* [(Topcuoglu, 2002)], to compare the performance of our reactive approaches GTP,GTP/c,DLS/sr and $HEFT$. The NSL metric is defined as the ratio of the schedule length (makespan) to the sum of the computational weights along the critical path. Note that, the critical path of a particular DAG graph is computed statically. This means that, even though the reactive nature (a new task graph is generated at each rescheduling cycle due to some tasks may have finished execution) of some competing methods, the NSLs obtained can be directly compared. The NSL can be computed as

$$NSL = \frac{Makespan}{\sum_{v_i \in CPath} \overline{W_i}} \tag{4.4}$$

Note that the denominator in NSL takes no account of dependencies outside the critical path. This makes it quite likely that the minimal theoretical NSL of 1 will often be impossible in practice, and that it is quite natural for NSL to grow significantly as task graphs become large and complex. We use averaged NSL over set of DAGs as a comparison metric. Our main interest in NSL (as opposed to absolute makespan) is as a means of judging the relative success of competing schedulers.
To help understand the behavior of each model, we introduce three other metrics averaged over all the graphs under consideration:

1. The number of remappings in which at least one placed task was migrated. This may differ from the total number of remappings which can be determined by dividing the makespan of the application by the fixed-size of the rescheduling point.

2. The number of migrated placed tasks over time.

3. The overhead cost incurred by the mapping method defined as the sum of the total or partial re-computation or re-transmission of data involved in the migration

of tasks. We note that the cost of rescheduling the application is not included. The fact that GTP and GTP/c allow migration of tasks, does not necessarily mean that there will be migrated tasks, but in those cases in which we have migrated tasks, they incur an overhead cost.

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

In the same manner, we use the NSL metric defined in equation 4.4 to evaluate the performance of the rewinding mechanism integrated into the reactive approaches GTP (GTP/r) and $GTP/c(GTP/c/r)$ models.

Little work has been conducted to design fault tolerant mechanisms for DAG applications. Thus, aiming to understand the behavior of such mechanisms, we will use three complementary metrics averaged over all the graphs under consideration:

1. The Rewound Tasks (RT) metric, which counts the number of placed tasks rewound to preserve the execution of the application.

2. The overhead cost incurred by the rewinding mechanism. In this part we include the amount of computation and data transfer (in units of time) which was repeated as part of the rewinding mechanism.

3. The Rewound Levels (LR) metric, which considering that the DAG graph can be divided into levels (layers), denotes the number of levels (how deep) the application had to be rewound after processor failure.

Note that the rewinding mechanism is a *responsive* approach, as it is triggered when a processor failure is detected at some rescheduling point. It is true that, injecting just one event simulating a processor failure at the mid-point of the execution may not reflect the behaviour of a real distributed system, where one or more processor failures may occur at any point of time. However, we seek to maintain the consistency in our scheduling scenarios by involving the same sequence of randomly defined (but repeatable) events (each simulating a resource change in either processor or bandwidth availability) used for evaluating GTP and GTP/c. The key difference is that we now injected an additional event simulating a processor failure to occur at the mid-point of the execution. Additionally, we make strong emphasis in the correlation between the rewinding mechanism and the mapping method (i.e., rewound tasks, rewound levels). Thus, by considering the simple case in which only one processor failure occurs

during execution, we start to explore the impact of the mapping strategies on the performance of the application when a processor failure is detected at some rescheduling point. In the literature, some research projects addressing fault tolerant mechanisms (i.e., retry, alternate resource) focus on evaluating the fault tolerant mechanism using standard metrics to produce failure cases at a certain arrival rate [(Hwang and Kesselman, 2003), (Duda, 1983), (Beguelin et al., 1997)]. Some of the standard metrics include:

1. The Mean Time Between Failures (MTBF) measures the average amount of time between failures.

2. Mean Time to Failure (MTTF), is the average time between adjacent arrivals of failures.

3. Mean Time To Repair (MTTR), is the time taken to repair a failure.

4. Probability of failure on demand (POFOD), is the possibility that the system will fail when a user requests service.

The conceptual framework within which such metrics are applicable is somewhat different to our own. Essentially, in our scenario, in scheduling terms, the extreme case is that a single surviving processor could re-execute the entire application from scratch - there is no concept, for example, of tying resources or actions to specific locations. Thus, in effect failure only occurs when the entire resource pool closes down. From another perspective, it could be argued that our approach might be very vulnerable to failure, depending upon the detailed mechanism used to gather resource information. However this presents an implementation challenge independent of the scheduling actually done with the information. We have not attempted to extend our model to encompass these more conventional aspects of fault-tolerance.

4.5 The Simgrid Software

Our evaluation is conducted by simulation, since this allows us to generate repeatable patterns of resource performance variation. In this section we describe the steps taken in our simulation to model our distinctive simulation scenarios described above, in which resources vary their performance characteristics (availability and bandwidth)

over time during the execution of some particular application. For this, we use the Simgrid (Grid Simulator) software version 2.8, described and downloadable from [(Simgrid, 2001)], as a grid simulation platform. Simgrid provides a set of core abstractions and functionalities that can be used to build simulators for specific application domains and/or computing environment topologies. In the literature, Simgrid has already been widely used by different researchers [(Hernandez and Cole, 2007a), (Legrand et al., 2003), (Beaumont et al., 2002), (Beaumont et al., 2003), (Faerman et al., 2002), (Hernandez and Cole, 2007c)]. Simgrid assumes that resources have two performance characteristics: *latency* in time units to access the resource and *service rate* which is the number of work units performed per time unit. Simgrid documentation suggests a pair of mechanisms to evaluate performance characteristics: a) as constants, in which the initial value defining the performance characteristics of the resource (availability or bandwidth) remains constant during the execution, and b) as traces (dynamic events). Traces allow us to model changes in resource performance characteristics over time, such as the ones observable for real dynamic resources and following our notion of simulation scenarios described above.

Early experiments, modelling resources with constant characteristics and static scheduling, proved successful. However, when evaluating our dynamic models according to the characteristics of our scenarios described in Section 4.3, in which we consider variations in resources characteristics over time, we note that the Simgrid mechanism using traces did not behave according to our expectations. To address this problem, we designed the tracking mechanism which was built on top of the Simgrid software, to obtain the notion of more realistic dynamic scenarios, allowing to have a sequence of events, each simulating a fluctuation in resource performance during execution.

4.6 The Tracking Mechanism

In keeping with the principles of our planned scenarios, our tracking mechanism supports a sequence of chronological events (i.e., ev_1^t, $ev_2^{t_1}$, $ev_3^{t_2}$, etc.) over time, each simulating a resource change in either processor or bandwidth availability. Our mechanism works on the principle that the problem of including traces (dynamic events) during the execution of the application, can be represented as a sequence of evaluations with constant performance characteristics. The mechanism can be seen as a lower-level cycle, iterating at each event, which we called tracking point. The mechanism manages the notion of a virtual clock during the execution, such that it is able to distinguish

the time at which each event (tracking point) occurs and then at each tracking point, reflects the change(s) in the respective resource(s), such that the subsequent execution of the remaining tasks will now reflect such changes, obviously affecting the execution time of those tasks mapped onto the processors involved in the events. Thus, the tracking mechanism gives the impression that the application is executed in more realistic scenarios. To achieve this, the tracking mechanism must perform the following actions at each tracking point,

1. Update the progress of the tasks in the STG structure.
2. Update the current schedule with the progress of tasks.
3. Update the change in the performance in ITG for each resource involved.
4. Update the virtual clock to the time at which the event occurs.

Now, after performing the above operations, the next step is to evaluate the current updated schedule until the next planned event or until the application finishes execution (i.e., no more events). The tracking mechanism seeks to guarantee that the sequence of events, each simulating a resource change in either processor or bandwidth availability, will occur at their planned time, affecting the execution time of those tasks mapped onto the respective processor. In our context, the main difference between rescheduling and tracking points, is that the iterations caused by the events (tracking points) do not perform rescheduling of the application (as in rescheduling points), they just reflect the changes in the resource characteristics such that they can be considered in the subsequent simulation.

To illustrate the tracking mechanism, consider Figure 4.6 in which we observe a sequence of chronological dynamic events (traces), each simulating a resource change (i.e., ev_1^{t1}, ev_2^{t2}, ev_3^{t3}, etc) and indicating a tracking point . We recall that the information about the initial performance (availability or bandwidth) is stored in the *GRP* structure. Then, as the execution starts, the resources take into account the initial performance. We observe that the tracking mechanism must recognize the time at which every trace (events) occurs in chronological order. Thus, following the example, the first evaluation (iteration) performed by the tracking mechanism is performed from $t = 0$ to $ev_1^{t_1}$ at which a change in the performance of p_1 occurs, then at this tracking point, the mechanism must reflect the change and perform the operations defined above so that the subsequent execution of the remaining tasks reflects the changes. Thus, we observe that this event occurs when v_1 is being executing on p_1, then after reflecting the

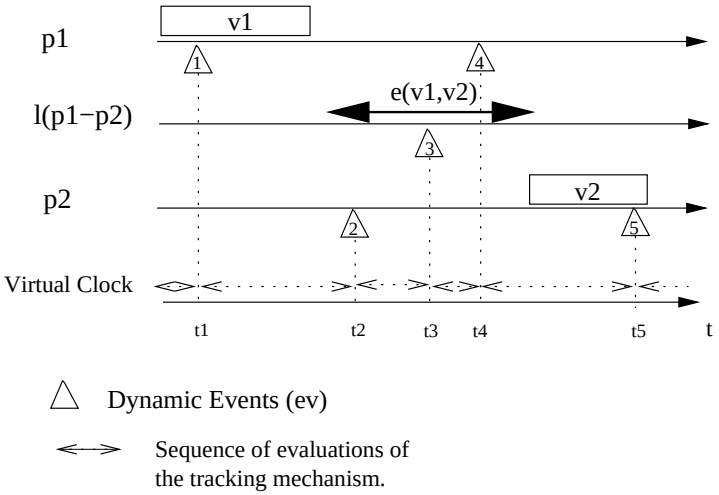


Figure 4.6: The Tracking Mechanism

change of p_1, the remaining work of v_1 on p_1 will be performed subject to the new performance of p_1. The virtual clock is updated to t_1. The performance of p_1 will remain constant until ev_4 which is the next event involving p_1. After $ev_1^{t_1}$, the next tracking point occurs at $ev_2^{t_2}$ involving the processor p_2, thus the next evaluation is performed from the current virtual time t_1 to t_2, in the same manner, the change in p_2 will be reflected in the subsequent execution following the procedure of the mechanism. Thus, the mechanism continues until the application finishes execution.

4.7 Summary

In this chapter we described all the components involved in the simulation framework. The evaluation of our reactive scheduling mechanisms is conducted by simulation, since this allows us to generate repeatable patterns of resource performance variation. We described the source and characteristics of the Input Task Graphs (ITGs) used in the evaluation. Next, considering the dynamic nature of SHCS, we described the distinguishing characteristics of our simulation scenarios. We explained the criterion used to define the fixed rescheduling points used to reschedule the application, followed by a description of the metrics used to evaluate the performance of our models. Finally, we described the extended version of the Simgrid software, which allows changes in resource performance characteristics over time, as observable in real dynamic resources.

Chapter 5

Experimental Results

In Chapter 3 we defined the GTP model, which allows rescheduling and migration of tasks in response to significant variations in resource characteristics. In the same chapter, we defined the GTP/c model, an extended version of GTP that considers the maintenance of a collection of reusable copies of the results of completed tasks to improve the utilization of resources and to minimize the impact of the migration cost on the application makespan. Finally, we defined the rewinding mechanism to preserve the execution of the application despite the presence of failure in resources. The rewinding mechanism was integrated into the GTP and GTP/c models, resulting in the extended versions GTP/r and $GTP/c/r$ respectively. In this chapter we evaluate the performance of the models by using the metrics defined for each model in Section 4.4, averaged over all the graphs under consideration. Our evaluation is conducted by simulation, since this allows us to generate repeatable patterns of resource performance variation. To achieve this, we will use a collection of DAGs and a number of test scenarios. A scenario involves a sequence of randomly defined (but repeatable) events, each simulating a resource change in either processor or bandwidth availability. We used the Simgrid software [(Simgrid, 2001)], which we have adapted to support the variability in resources, as described, in Section 4.5.

5.1 Structuring the Experimental Results

We have structured our observations based on the experimental results as shown in Figure 5.1. We start by analyzing the behavior of the static mapping methods (i.e., HEFT and DLS) evaluated on our simulation scenarios. Then, we continue with the analysis of the performance of the GTP reactive mapping method, which considers

the cyclic use of a mapping method over time, in response to variability in resources. Then, we analyze the results of GTP/c, describing the impact on makespan when it uses reusable copies for scheduling tasks onto processors,

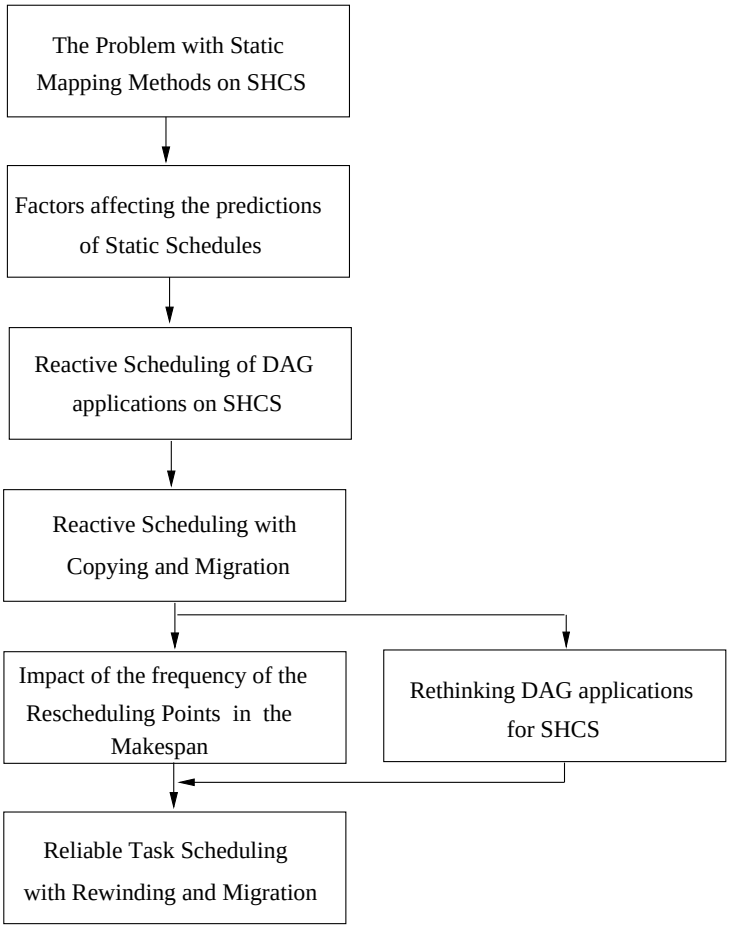


Figure 5.1: Structure of the experimental results obtained in our research

After this, we describe a pair of issues related to the performance of the reactive mapping methods: The first issue concerns the length of the rescheduling cycle. The second issue concerns the data flow mechanism among the tasks composing the application. Finally, we present the results of the rewinding mechanism included in GTP/r and $GTP/c/r$.

We recall that in SHCS the availability and performance of computational resources can vary dynamically over time, even during the course of an execution. Thus, in order to have a more realistic simulation environment, we included into our scenarios events, which simulate a change in the performance of the resource (availability or bandwidth). As described in Section 4.3, 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, in the 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 consider a bound ranging from 0% to 90% in increments of 10%. We notice that scenarios with a bound of 0% are a special case in which resources remain fully available over the execution of the application. This scenario is more suitable for static mapping methods generating static schedules.
The graphics used in this section to present the results of the evaluation are distinguished by showing the whole spectrum of bounds for each scenario. We will refer to a particular scenario as $SCE(x,y,z)$, which means that the scenario uses x processors, y tasks and z percent of variability in computational resources.

5.2 The Problem with Static Mapping Methods on *SHCS*

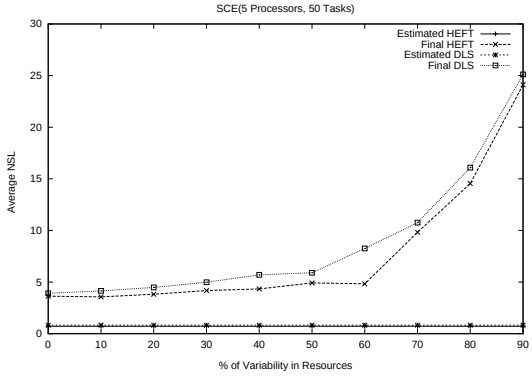
In Section 2.6, we mentioned that when evaluating a schedule of a particular DAG application on SHCS, we may consider whether to use static mapping methods or to consider reactive mapping methods such as *GTP*, which iteratively compute improved schedules over time. In this section, we intend to explore the problem and to understand the behavior of static mapping methods when they are executed on SHCS. We first start by describing the performance of static mapping methods, which obtain an initial schedule and launch the schedule onto the target system (i.e., SHCS) under the assumption that resources are dedicated and unchanging over time. To achieve this, we sought in the literature some computationally low-cost static mapping methods, which were capable of addressing heterogeneous resources and producing good solutions in a reasonable amount of time. Thus, we selected the Heterogeneous Earliest Finish Time (HEFT) [(Topcuoglu, 2002)] and the Dynamic Level Scheduling (DLS) [(Sih and Lee, 1993)]. The HEFT algorithm might be one of the most frequently referred to listing static mapping methods. For instance, it is evaluated in [(Wieczorek et al., 2005)] and compared with a genetic algorithm and a myopic algorithm. The experimental results show that HEFT outperformed the other algorithms. On the other hand, the DLS algorithm is one of the earliest algorithms to consider heterogeneous processors. It is also referred by many researchers. In [(Jarry et al., 2000)], it is evaluated and compared with the Dynamic critical-path algorithm (DCP) [(Kwok and Ahmad, 1996)]. The results show that DLS outperformed DCP when the fluctuations in the variability

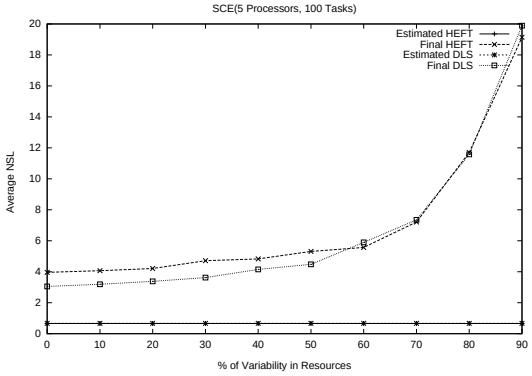
of resources increased considerably.
We proceed to evaluate both static mapping algorithms by using scheduling scenarios for DAGs with CCR = 1.5 and the average of the NSLs of all the graphs under consideration. We will refer to the difference between the average of the estimated NSLs, which consider the estimated initial makespan, and the average of the real NSLs, which consider the real final makespan, as the NSL gap.

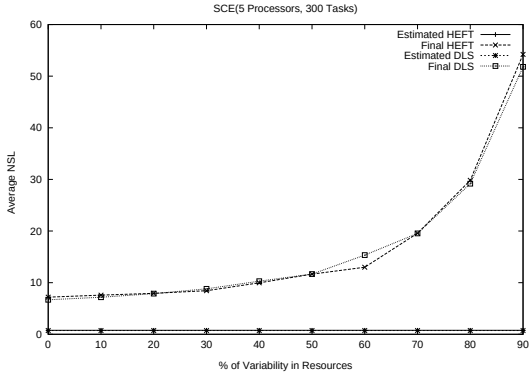
Then, to describe the experimental results concerning static mapping methods with static schedules on SHCS, we divide our observations in two main parts. The first part shows the results of both HEFT and DLS for $SCE(x,y,0)$, (0% of variability in resources, considered as suitable for static mapping methods). The second part shows the results for $SCE(x,y,z)$ where $10 \leq z \leq 90$ describes the percentage of variation among resources, as more realistic scenarios.

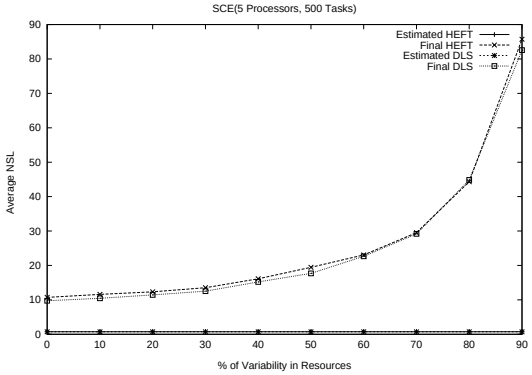
1. In Figure 5.2 we can observe the performance results for HEFT and DLS for each scenario $SCE(x,y,0)$. Our first observation is that the estimated (initial) average NSL for both is similar. In terms of the final average NSL, HEFT tends to have a better performance than DLS, particularly in those scenarios with 10 and 20 processors. For instance, in $SCE(10,1000,0)$ HEFT outperforms DLS by 11% and $SCE(20,300,0)$ by 16%. For those scenarios with 5 processors, DLS tends to outperform HEFT by up to 9%. In general terms HEFT outperforms DLS, having best performance when the number of tasks increases (500 and 1000 tasks).

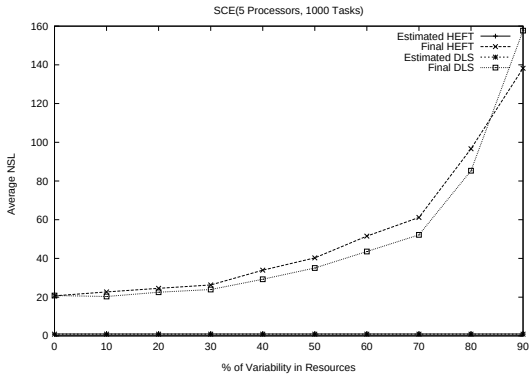
   Considering the NSL gap, this tends to be quite high, even given the static nature of resources in this sort of scenarios. For scenarios with limited number of resources (5 Processors), the NSL gap tends to be higher, being gradually incremented as the application becomes larger and complex (500 and 1000 tasks). As we increase the number of processors, the NSL gap decreases, however it maintains the gradual increase for larger DAGs. For instance, for $SCE(5,1000,0)$ the NSL gap for HEFT is up to 22 times the estimated average NSL and up to 23 times for DLS. If we increase the number of processors, then for $SCE(10,1000,0)$ the NSL gap is up to 12 times the estimated average NSL for HEFT and up to 13 times for DLS. This means that apart from the argument that resources may vary over time, static mapping methods producing static schedules are affected by some factors that may negatively affect the performance of the application, increasing the gap between the real and predicted makespan. In the next section we describe some observations about such factors.

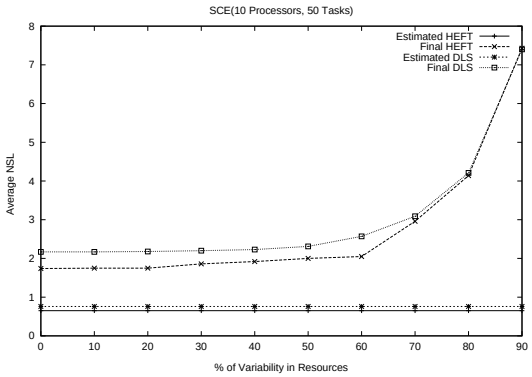

SCE(5 Processors, 50 Tasks)
Estimated HEFT
Final HEFT
Estimated DLS
Final DLS
Average NSL
% of Variability in Resources

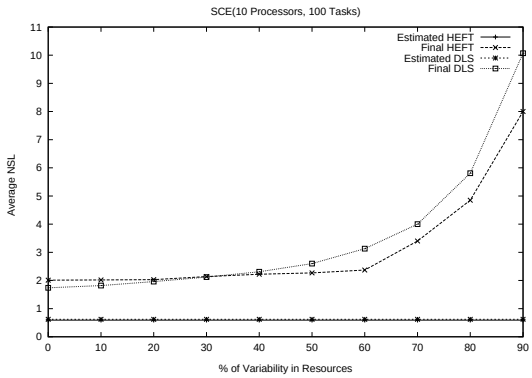

SCE(5 Processors, 100 Tasks)
Estimated HEFT
Final HEFT
Estimated DLS
Final DLS
Average NSL
% of Variability in Resources

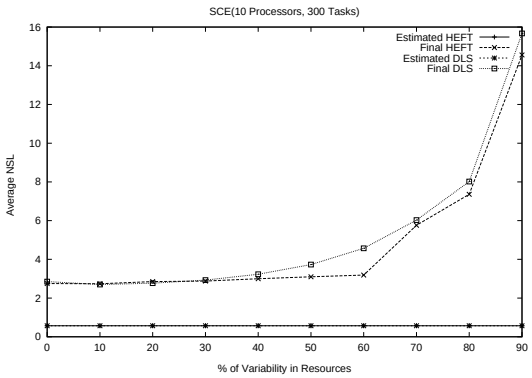

SCE(5 Processors, 300 Tasks)
Estimated HEFT
Final HEFT
Estimated DLS
Final DLS
Average NSL
% of Variability in Resources

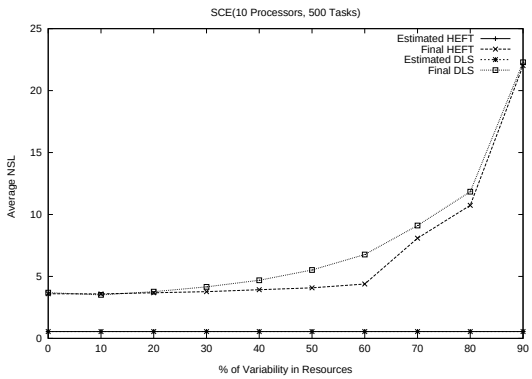

SCE(5 Processors, 500 Tasks)
Estimated HEFT
Final HEFT
Estimated DLS
Final DLS
Average NSL
% of Variability in Resources

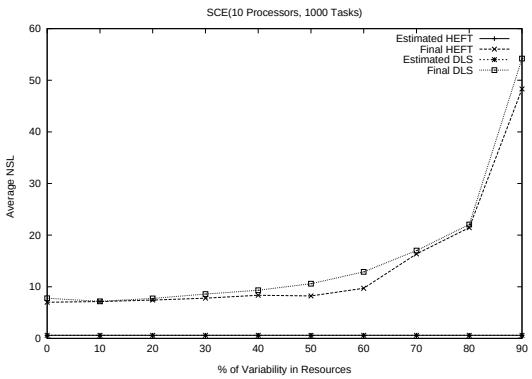

SCE(5 Processors, 1000 Tasks)
Estimated HEFT
Final HEFT
Estimated DLS
Final DLS
Average NSL
% of Variability in Resources


SCE(10 Processors, 50 Tasks)
Estimated HEFT
Final HEFT
Estimated DLS
Final DLS
Average NSL
% of Variability in Resources


SCE(10 Processors, 100 Tasks)
Estimated HEFT
Final HEFT
Estimated DLS
Final DLS
Average NSL
% of Variability in Resources


SCE(10 Processors, 300 Tasks)
Estimated HEFT
Final HEFT
Estimated DLS
Final DLS
Average NSL
% of Variability in Resources


SCE(10 Processors, 500 Tasks)
Estimated HEFT
Final HEFT
Estimated DLS
Final DLS
Average NSL
% of Variability in Resources


SCE(10 Processors, 1000 Tasks)
Estimated HEFT
Final HEFT
Estimated DLS
Final DLS
Average NSL
% of Variability in Resources

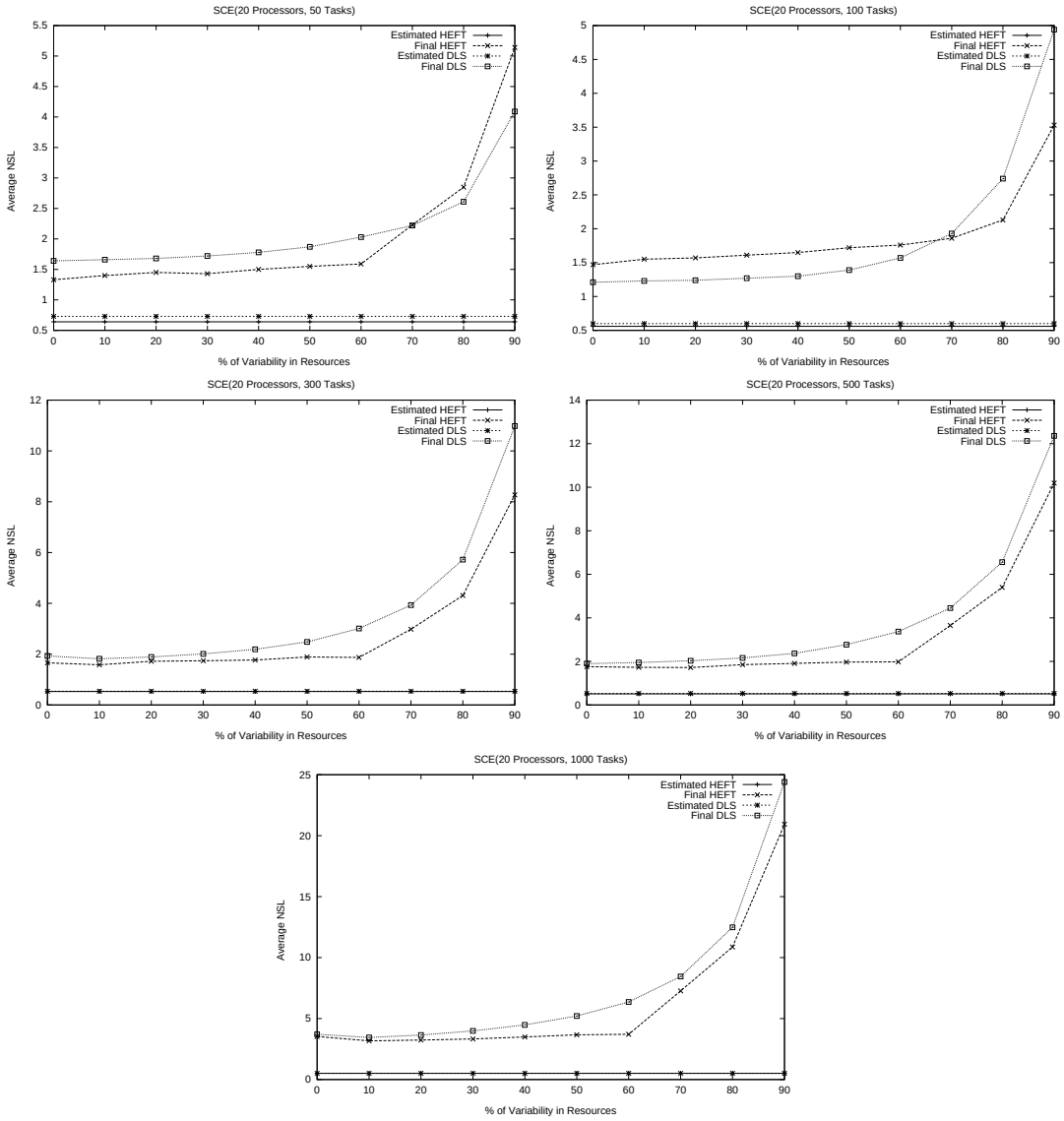


Figure 5.2: Average NSL of the static mapping methods HEFT and DLS

2. In general terms, in more realistic scenarios which include variations in resources ranging from 10% to 90%, we observe that HEFT tends to be better than DLS, in most cases up to 20% with few exceptions. We can see for each scenario in Figure 5.2 a steady increase in the final average NSL of the application, increasing the NSL gap when the variability in resources increases. This means that the initial predictions of those DAG applications executed with static schedules are affected over time by the dynamic nature of SHCS, affecting the performance of the application by increasing the final average NSL of the application.

The initial predictions made by the static mapping methods with static schedules

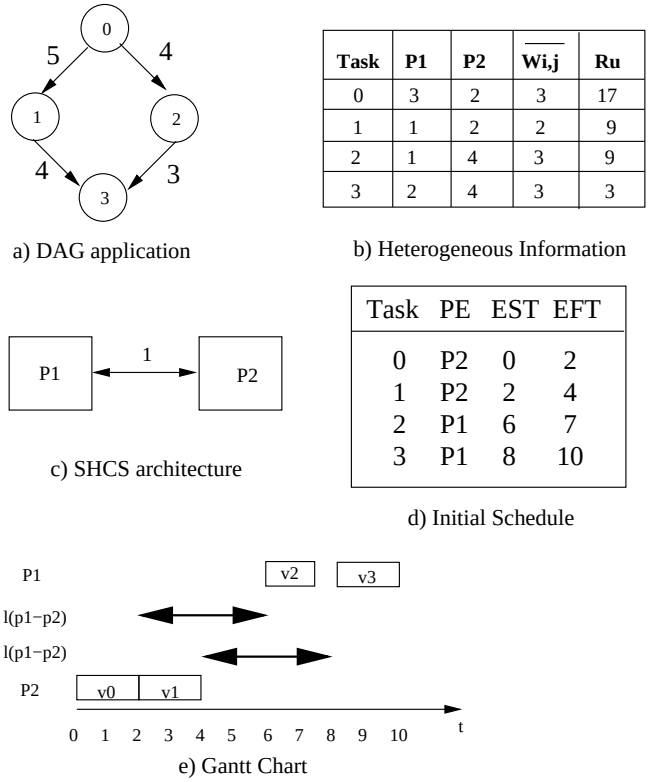
are affected over time by external and internal factors. We will refer to as external factors those factors outside the scope of the scheduling decisions but affecting the predictions of the candidate schedule. Thus, we identify two external factors: the variability in resources and the communication model among tasks (PULL and PUSH models). We found that the external factors related to the variability of resources, may negatively affect the performance of the application by increasing the final makespan of the application. In Section 5.7, we will show how the communication model may affect the application performance. We will refer to as internal factors those related with the scheduling decisions affecting the predictions reflected in the schedule to be launched to SHCS. This was observed when the static schedules were evaluated on scenarios with dedicated and unchanging resources over time.

5.3 Factors affecting the predictions of static schedules

As we described in Section 5.2, there are internal and external factors, which may affect the initial predictions of static schedules during the execution process. We demonstrated that the external factors, related to variations in resources, affect the initial predictions by increasing the final makespan of the application. In this section we enumerate some internal factors that we believe are related to the performance observed for the application using HEFT and DLS (and probably other static approaches in the literature), particularly when we conducted the experiments for perfect (z=0) resources.

We argue that the internal factors are linked to the distorted notion of SHCS used by the static methods when scheduling decisions. This means that scheduling decisions are based on the notion of unrealistic SHCS architectures when the prediction cost is computed, mainly to keep the simplicity of the models. We describe below some of the unrealistic notions considered during scheduling decisions,

1. *Ignoring traffic contention* by assuming an infinite bandwidth between a pair of processors when the prediction cost is calculated. This notion of SHCS may lead to poor estimated schedules because the predicted time to transfer data among tasks may be shorter than the real time. This is illustrated in the example of Figure 5.3 where we follow the steps of the HEFT algorithm. Consider the task graph (Figure 5.3(a)), the heterogeneous information (Figure 5.3(b)) and



Task	P1	P2	$\overline{W_{i,j}}$	Ru
0	3	2	3	17
1	1	2	2	9
2	1	4	3	9
3	2	4	3	3

Task	PE	EST	EFT
0	P2	0	2
1	P2	2	4
2	P1	6	7
3	P1	8	10

Figure 5.3: Example of a distorted notion of SHCS architectures

the SHCS architecture (Figure 5.3(c)). The order in which tasks will be mapped is based on the upward rank (R_u) showed in Figure 5.3(c) which gives the sequence $\{v_0,v_1,v_2,v_3\}$. Thus, Figure 5.3(d) shows the initial schedule generated by the HEFT algorithm. We can observe in Figure 5.3(e), the Gantt Chart generated for the initial schedule in which we observe that HEFT ignores traffic contention by assuming an infinite bandwidth for the $link(p_1, p_2)$. Thus, the estimated makespan was computed to be 10 units of time. However, by sequencing the data transfers on $link(p_1, p_2)$, it will actually take 12 units of time, 2 units of time more than the estimated makespan.

2. *Fully connected networks* assume that there always exists a link to data transfer between each pair of processors, when it can be completely different in real SHCS environments. For our purposes, we consider in our models GTP and GTP/c that the processors composing the target architecture are fully connected. However, such assumption may lead to poor predictions when the real target architecture is not fully connected. For instance, when estimating the communication cost to transfer data from a task mapped to a processor p_m and a succeeding

task mapped to a different processor p_n, such that there is not communication link between p_m and p_n.

We believe that to increase the performance of the application, new efforts are required in designing static mapping methods to address the internal factors. In [(Sinnen et al., 2006), (Sinnen and Sousa, 2005), (Agarwal et al., 2006)] it is observed that including traffic contention into the scheduling decisions, may improve the predictions of the generated schedules. Obviously, the inclusion of such techniques increases the complexity of the algorithm.
The other option, which we seek to address in this work, uses reactive mapping methods such as *GTP*, which iteratively compute improved static schedules over time and are able to react to the dynamic nature of SHCS (external factors). However, the problem of the internal factors may remain if they rely on distorted notions of SHCS.

5.4 Reactive Scheduling of DAG Applications on SHCS

First, we notice that the cost prediction for the *GTP* model is based upon redefinition of concepts drawn from the standard scheduling literature [(Kwok and Ahmad, 1999a), (Topcuoglu, 2002)], together with some additional operations required by the dynamically heterogeneous nature of our target system. Hence, *GTP* includes the same distorted notion of SHCS described in the previous Section 5.3 when the prediction cost is calculated. In this section we evaluate the performance of the *GTP* method on SHCS. To achieve this, we benchmark *GTP* against a pair of algorithms described in Chapter 2.5, the Heterogeneous Earliest Finish Time (HEFT) [(Topcuoglu, 2002)] which considers a static schedule approach and DLS/sr, a reactive mapping method described in Section 2.6. We recall that DLS/sr evaluates two different metrics for the selective rescheduling policy: the spare time and the slack of a node. For our purposes, we selected the spare time of tasks, which denotes the maximal time that a particular predecessor node can execute without affecting the start time of some of its dependent nodes that are either connected by an edge in the DAG or are adjacent in the execution order of the assigned machine. We notice that DLS/sr includes a migration model similar that of *GTP*, in which migration of tasks may be invoked when the cost of the migration itself is outweighed by the global time saved due to execution at the new site. In the same manner, a pessimistic model is considered, in which the migrated task must be restarted from the very beginning, including regathering all inputs directly

from the predecessors.

As part of the assessment, we are interested in two issues. The first issue concerns the results of the benchmark of the mapping methods and the second concerns the effect of traffic contention on scheduling decisions involving heterogeneous resources with changeable capabilities over time. To achieve this, we will use three different scheduling scenarios varying the communication to computation ratio (CCR) as 0.1, 0.5 and 1.5. The assessment of the results will be based on the metrics defined in Section 3.2. We proceed to describe the experimental results by grouping our observations in two main groups. The first group benchmarks reactive mapping methods (GTP and DLS/sr) against static mapping methods ($HEFT$). In this group, we will divide our observation in two main parts. The first part relates to scenarios which assume that resources are unchanging $SCE(x,y,0)$ and the second part to scenarios with $SCE(x,y,z)$ where $10 \leq z \leq 90$ describes the percentage of variation among resources, as more realistic scenarios. The second group describes individual results for the reactive mapping methods, GTP against DLS/sr. For convenience, this also includes results for GTP/c. We will discuss the introduction of GTP/c in a later section.

5.4.1 Scheduling Scenario for CCR = 0.1 and infinite bandwidth

We recall that the task computation times for a particular task v_i were created for each processor using uniformly distributed random numbers from the interval [1 to 10]. We start our evaluation by considering DAGs with CCR=0.1 and infinite bandwidth. Intuitively, the scheduling decisions must not be affected by the traffic contention.

1. *Static Mapping Methods against Reactive Mapping Methods.*

   - We observe in Figure 5.4 showing the average NSL, that for the case in which scenarios include 0% of variability, $HEFT$, GTP and DLS/sr present similar performance. The experimental results do not identify a clear tendency to determine the mapping method with the best performance. For this class of scenario, we observe that discrepancies between real and predicted estimations are low.
   - For more SHCS-like scenarios, we observe that the average NSL for HEFT, tends to gradually increase as the variability increases, more than GTP and DLS/sr. This is observed in those scenarios with 5 and 10 processors, when the DAGs become larger and complex (500 and 1000 tasks). This means

that the variability of resources tend to affect $HEFT$ more than GTP and DLS/sr (Figure 5.4). For instance, in scenarios with 80% of variability, the average NSL for HEFT is up to 15% higher than GTP and up to 17% higher than DLS/sr. The reactive strategy allowed GTP and DLS/sr to react more efficiently to external factors such as resource variability.

2. *Evaluation of Reactive Strategies*
   The experimental results show that GTP presents a similar performance to DLS/sr in many cases. We believe that the characteristics of this scenario contribute with more accurate predictions, as the impact of the traffic contention on the scheduling decision is practically null. It is observed in Figure 5.4 that DLS/sr outperforms GTP for scenarios with 5 processors, but GTP outperforms DLS/sr for scenarios with 20 processors. Complementary information shows that DLS/sr required a similar number of remappings than GTP (Figure 5.5) for DAGs with relatively few tasks (50 and 100 tasks). However, as the number of tasks increases, the number of remappings increases for DLS/sr. For instance, the experimental results for $SCE(10,500,40)$ indicate that DLS/sr required 5 times more remappings than GTP and for $SCE(10,1000,40)$, it required 7 times more remappings.

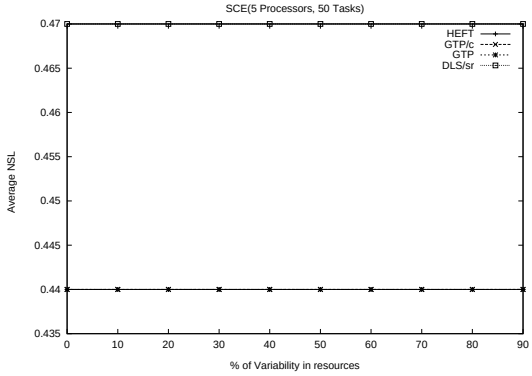
In general terms, for these particular scenarios, where the bandwidth is infinite, the problem of traffic contention tend to be null. We believe that this contributes to the more accurate estimations.

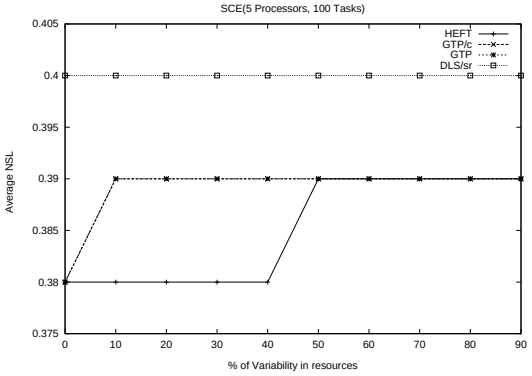
5.4.2 Scheduling Scenario with CCR = 0.5 and variable bandwidth

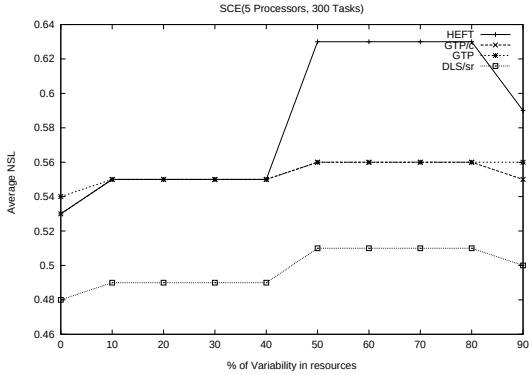
This scenario uses the same computation times, CCR = 0.5 and changing bandwidth over time, with the maximum bandwidth equal to one unit of data per unit of time. Now, by increasing the communication cost, we explore the impact on scheduling decisions when the bandwidth is finite.

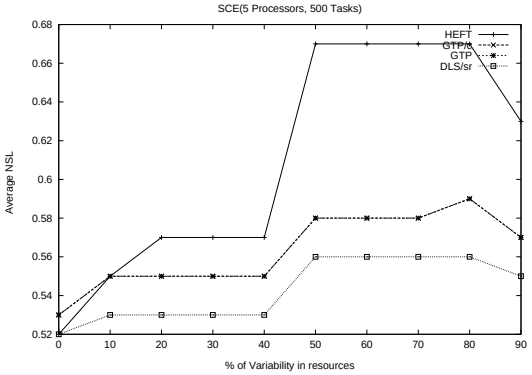
1. *Static Mapping Methods against Reactive Mapping Methods.*

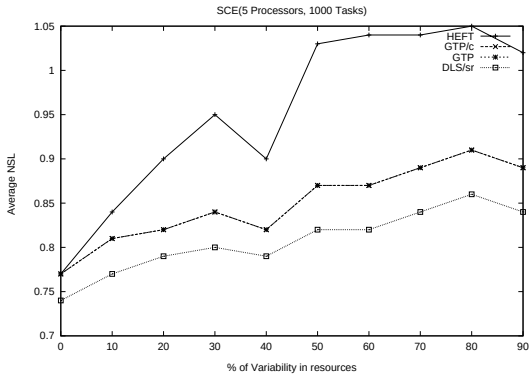
   - For scenarios with 0% of variability, we observe in Figure 5.6 that in most cases GTP tends to outperform HEFT, mainly as the number of tasks increases (300,500 or 1000 tasks). For instance, in $SCE(20,300,0)$, GTP outperforms HEFT by 5% and for $SCE(20,1000,0)$, GTP outperforms

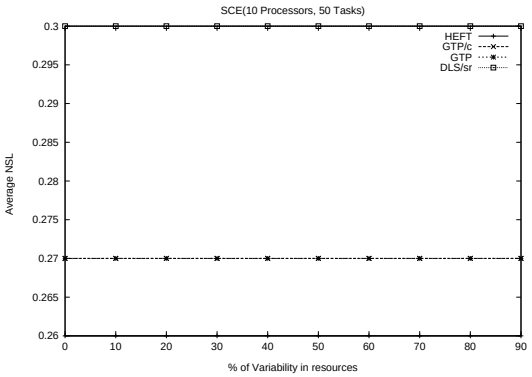

SCE(5 Processors, 50 Tasks)
Average NSL
% of Variability in resources
HEFT
GTP/c
GTP
DLS/sr

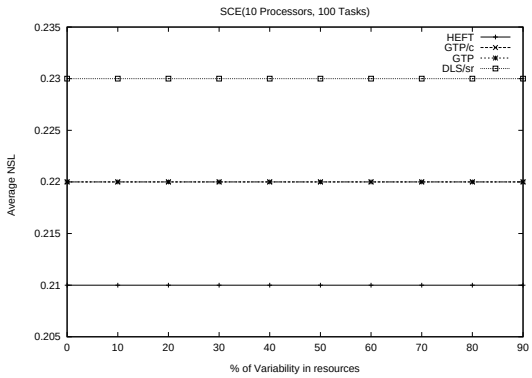

SCE(5 Processors, 100 Tasks)
Average NSL
% of Variability in resources
HEFT
GTP/c
GTP
DLS/sr

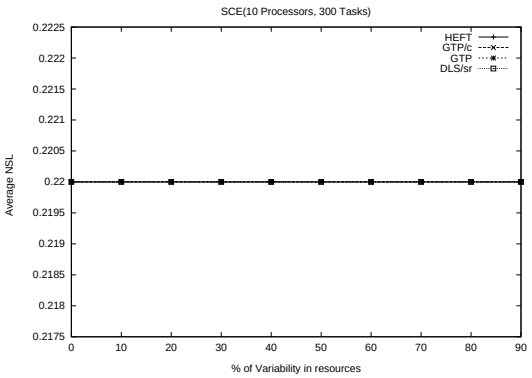

SCE(5 Processors, 300 Tasks)
Average NSL
% of Variability in resources
HEFT
GTP/c
GTP
DLS/sr

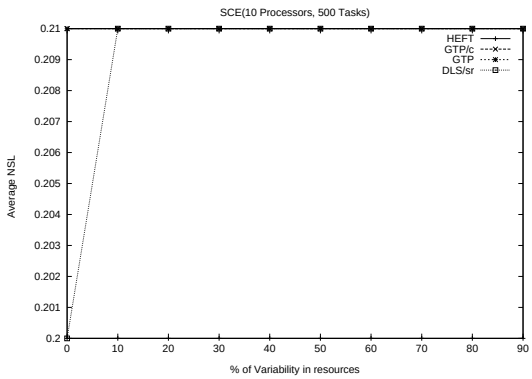

SCE(5 Processors, 500 Tasks)
Average NSL
% of Variability in resources
HEFT
GTP/c
GTP
DLS/sr

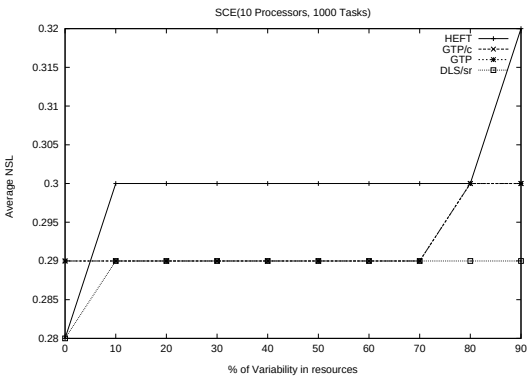

SCE(5 Processors, 1000 Tasks)
Average NSL
% of Variability in resources
HEFT
GTP/c
GTP
DLS/sr


SCE(10 Processors, 50 Tasks)
Average NSL
% of Variability in resources
HEFT
GTP/c
GTP
DLS/sr


SCE(10 Processors, 100 Tasks)
Average NSL
% of Variability in resources
HEFT
GTP/c
GTP
DLS/sr


SCE(10 Processors, 300 Tasks)
Average NSL
% of Variability in resources
HEFT
GTP/c
GTP
DLS/sr


SCE(10 Processors, 500 Tasks)
Average NSL
% of Variability in resources
HEFT
GTP/c
GTP
DLS/sr


SCE(10 Processors, 1000 Tasks)
Average NSL
% of Variability in resources
HEFT
GTP/c
GTP
DLS/sr

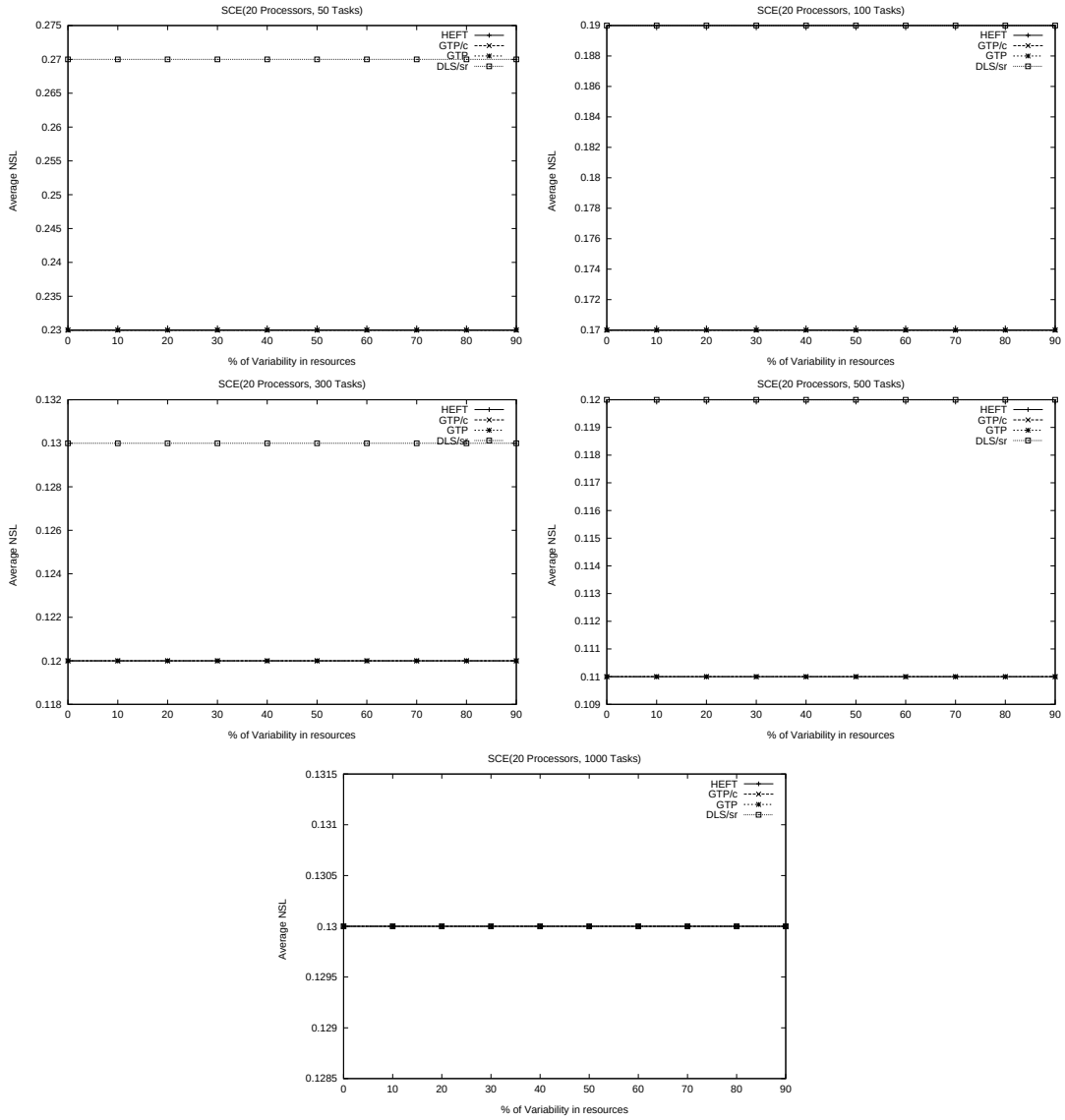
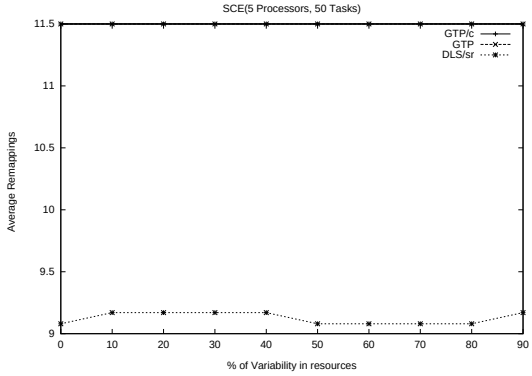
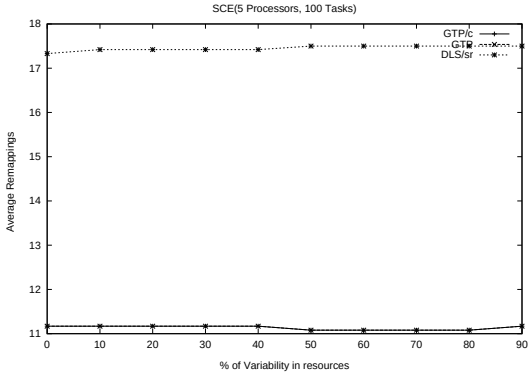


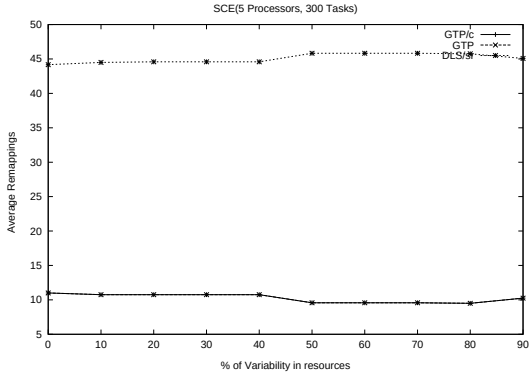
Figure 5.4: Average NSL for GTP, GTP/c and DLS/sr when CCR=0.1 and infinite bandwidth

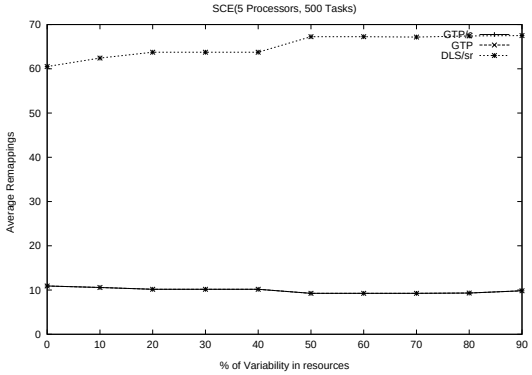
HEFT by 4%. Additionally, we observe that HEFT outperforms DLS/sr in some cases. Considering the same scenarios, in $SCE(20,300,0)$, HEFT outperforms DLS/sr by 6% and for $SCE(20,1000,0)$ by 1%.

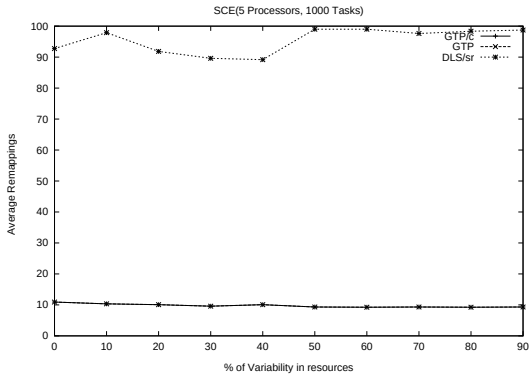
- For more SHCS-like scenarios, GTP outperforms $HEFT$ in most of the cases. DLS/sr outperforms $HEFT$ in some cases, particulary when the variability in computational resources is greater than 20%. In Figure 5.6, we can observe that the average NSL for HEFT, tends to gradually increase as the variability increases, more than the reactive mapping methods GTP and DLS/sr. This means that the internal (traffic contention) and external

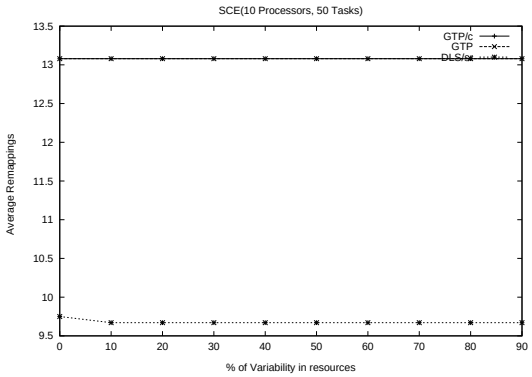

SCE(5 Processors, 50 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources

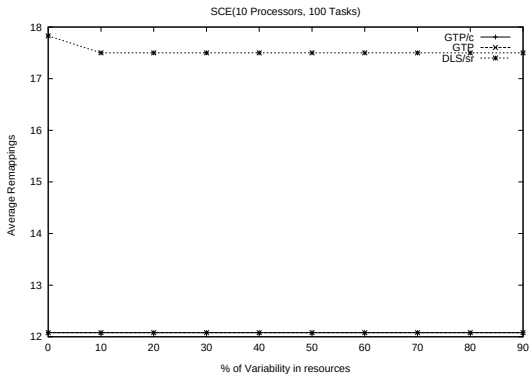

SCE(5 Processors, 100 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources

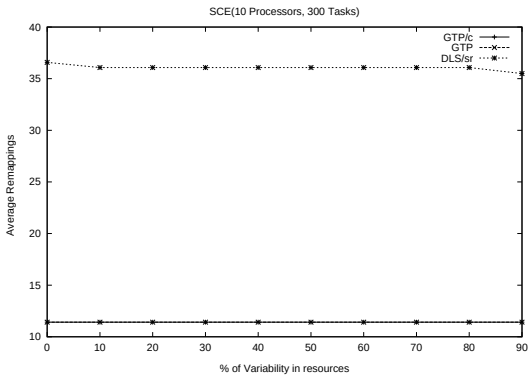

SCE(5 Processors, 300 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources

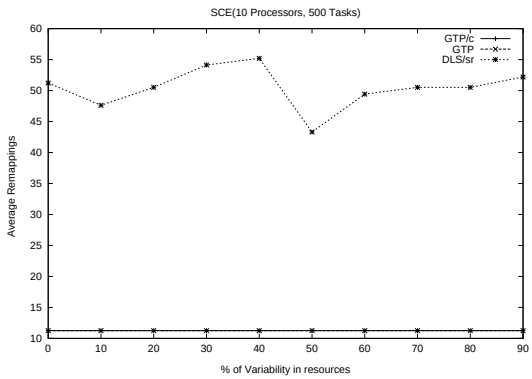

SCE(5 Processors, 500 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources

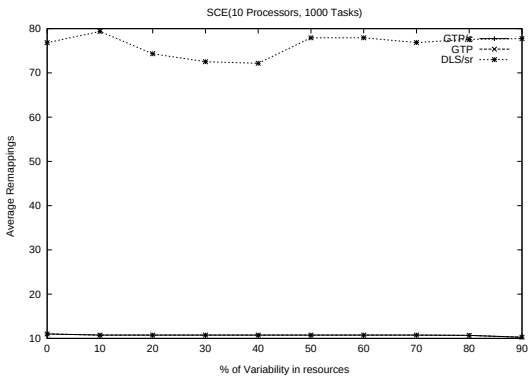

SCE(5 Processors, 1000 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources


SCE(10 Processors, 50 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources


SCE(10 Processors, 100 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources


SCE(10 Processors, 300 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources


SCE(10 Processors, 500 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources


SCE(10 Processors, 1000 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources

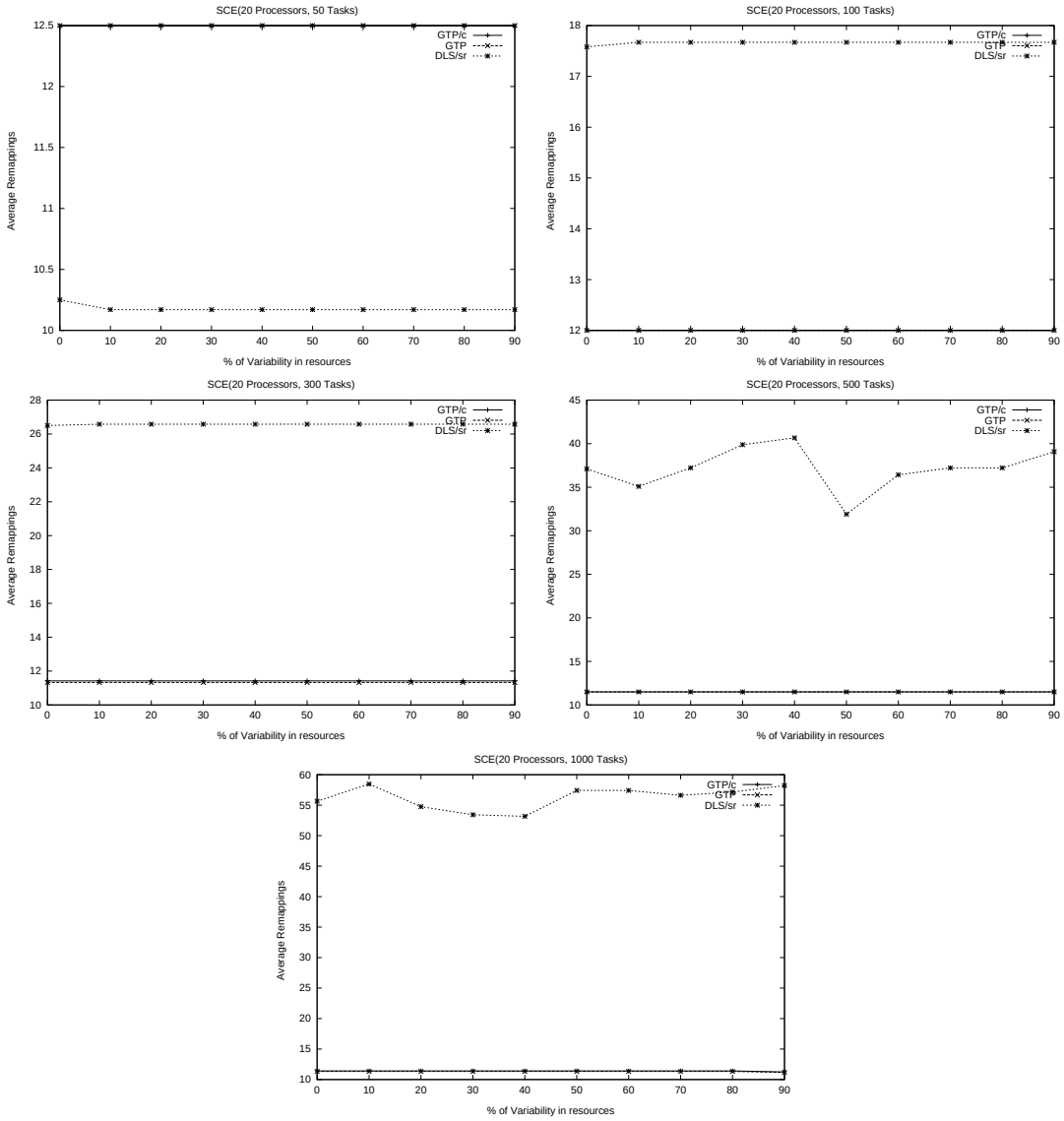


Figure 5.5: Average Remappings for GTP, GTP/c and DLS/sr when CCR=0.1 and infinite bandwidth

(resource variability) factors affect the initial estimations of the static approach $HEFT$ more than the reactive approaches GTP and DLS/sr. For instance, in scenarios with 70% of variability, the average NSL for HEFT is up to 0.53 times higher than GTP and up to 0.60 higher than DLS/sr. The reactive strategy allowed GTP and DLS/sr to refine the initial predictions considering the dynamic changes in resources over time into the scheduling decisions, addressing more efficiently the resource variability (external factors). However, the strategy of rescheduling the application tends to increase the overhead cost. This can be observed in Figure 5.9, which shows

the overhead cost.

2. *Evaluation of Reactive Strategies*
   The experimental results (Figure 5.6) show that, compared with the previous scenario where CCR=0.1 and infinite bandwidth, GTP outperforms DLS/sr in most cases. For instance, in Figure 5.10, we observe that for $SCE(5,1000,60)$, GTP outperformed DLS/sr by an average of 9% in terms of the average NSL, requiring an average of 10 remappings (see Figure 5.11) and an average of 700 migrated tasks (see Figure 5.12). On the other hand, DLS/sr required an average of 300 remappings and an average of 2000 migrated tasks, generating 4 times more overhead cost (recomputation and retransmitting) than GTP (see Figure 5.13). We believe that the performance of DLS/sr was more affected than GTP in the presence of traffic contention. In the next scenario we increase the CCR to 1.5 and maintain the bandwidth in one unit of data per unit of time, expecting to increase the traffic contention during execution.

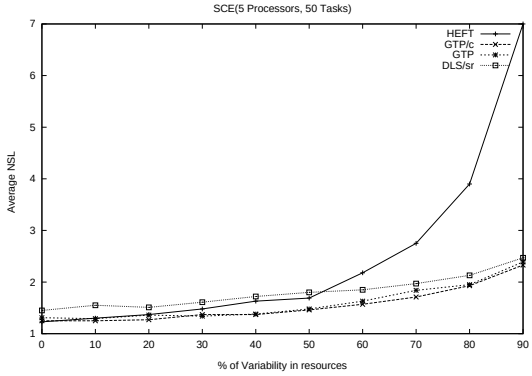
We believe that, as the traffic contention increases, the discrepancies between real and predicted estimations increase. In general terms, reactive mapping methods addressed more efficiently the internal (i.e.,traffic contention) and external (i.e.,variability) factors , showing in most cases, a better performance than the static approach.

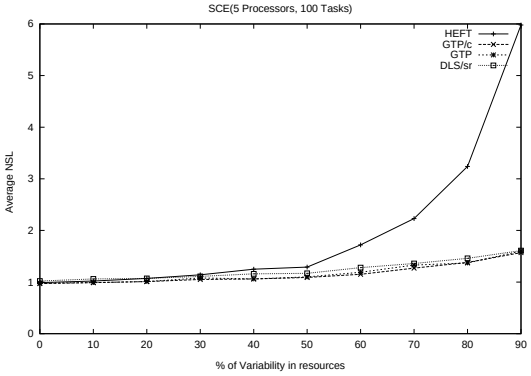
5.4.3 Scheduling Scenario with CCR = 1.5 and variable bandwidth

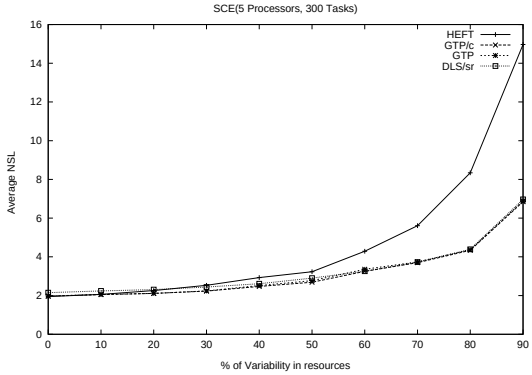
In this scenario, we keep the same computation times and the same characteristics for bandwidth. However, we increase the communication cost by considering CCR = 1.5.

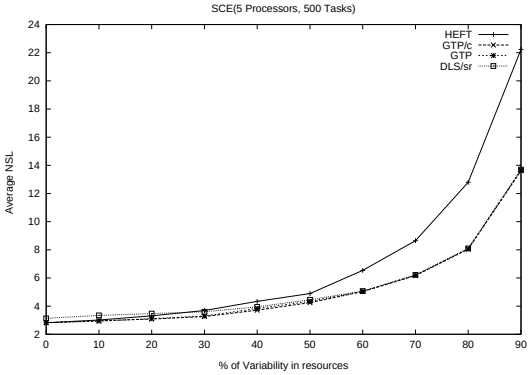
1. *Static Mapping Methods against Reactive Mapping Methods*

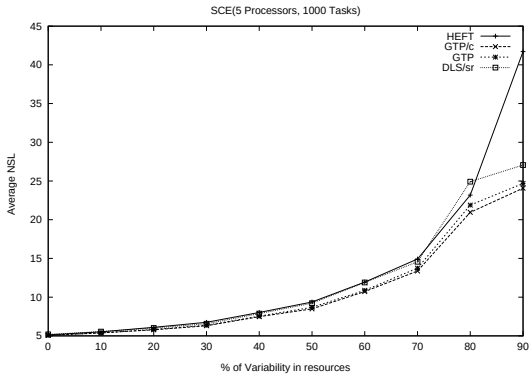
   - The case in which scenarios include 0% of variability allows us to investigate the extent which emerging discrepancies between real and predicted behavior are handled by GTP. We observe that as the communication cost increases, the traffic contention increases, increasing the discrepancies between the predicted and real estimations. Thus, in this scenario, the performance of HEFT is more negatively affected than in the previous scenarios. We observe in Figure 5.10 that in most cases GTP tends to outperform HEFT, mainly as the number of tasks increases (300,500 or 1000 tasks). For instance, in $SCE(20,500,0)$, GTP outperforms HEFT by

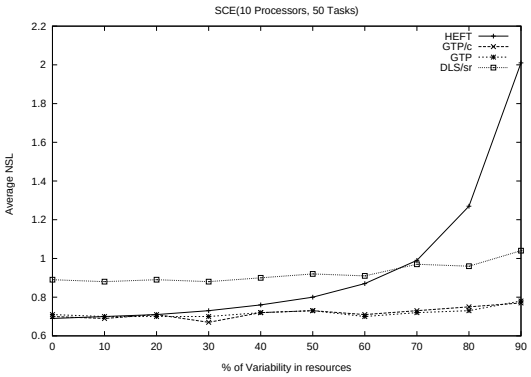

SCE(5 Processors, 50 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources

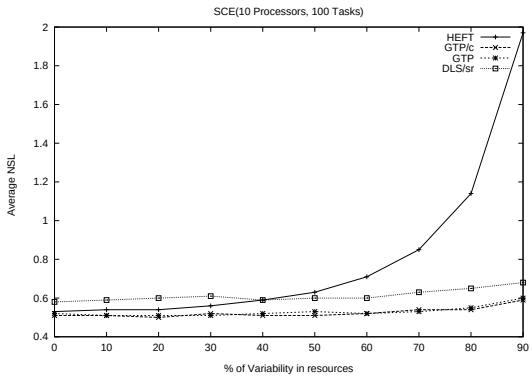

SCE(5 Processors, 100 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources

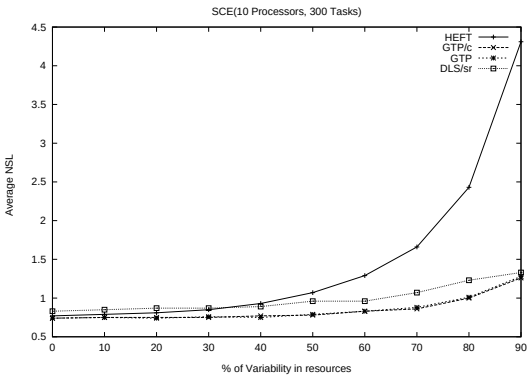

SCE(5 Processors, 300 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources

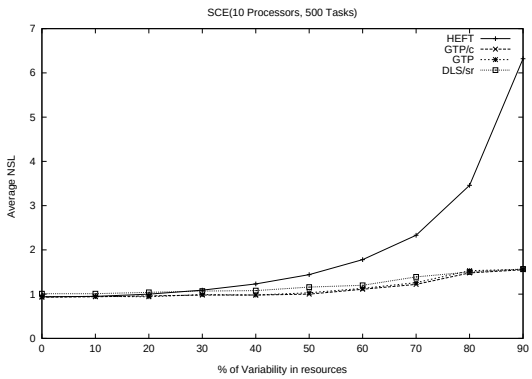

SCE(5 Processors, 500 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources

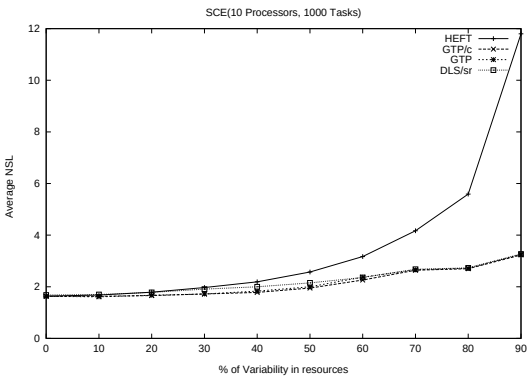

SCE(5 Processors, 1000 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources


SCE(10 Processors, 50 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources


SCE(10 Processors, 100 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources


SCE(10 Processors, 300 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources


SCE(10 Processors, 500 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources


SCE(10 Processors, 1000 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources

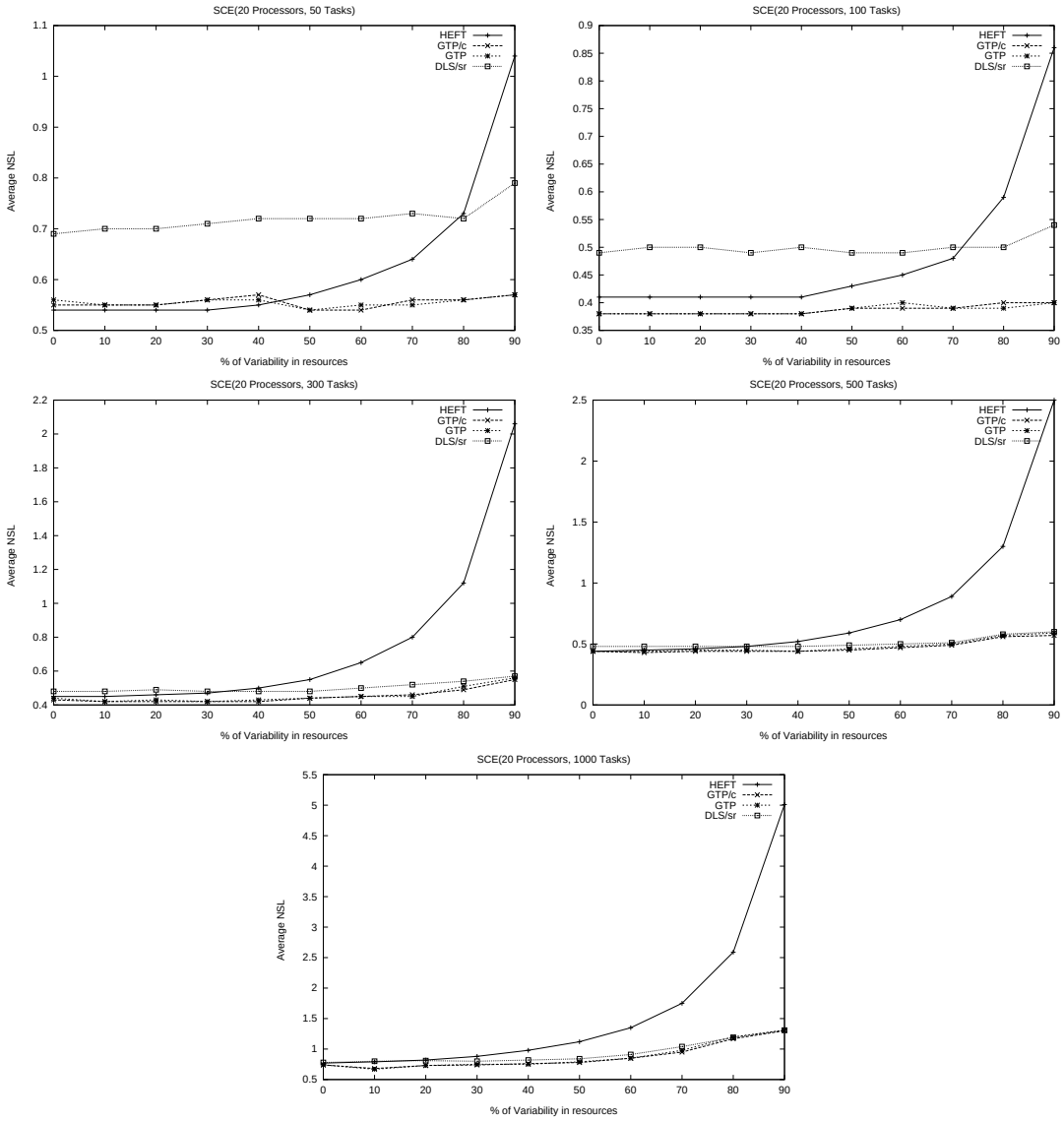
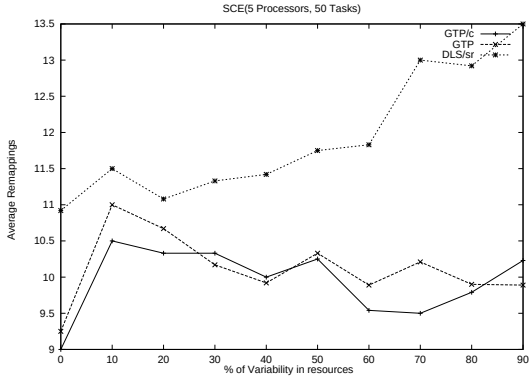
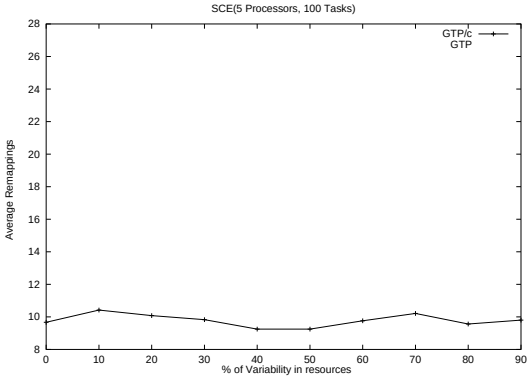
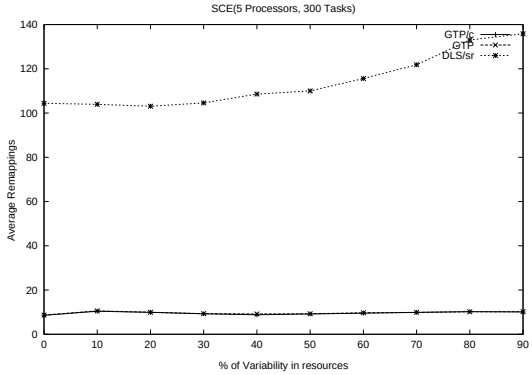


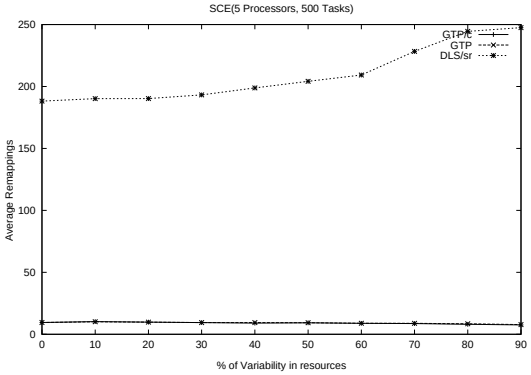
Figure 5.6: Average NSL for GTP, GTP/c and DLS/sr when CCR=0.5 and variable bandwidth

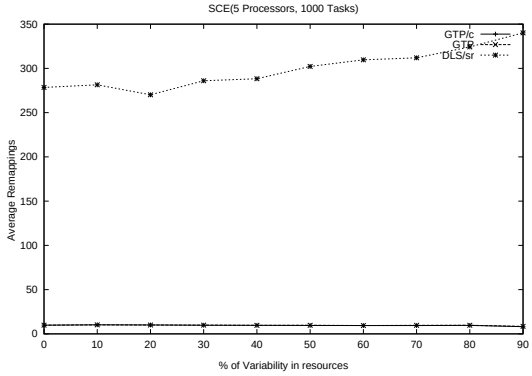
9% and for $SCE(20,1000,0)$ which increases the number of tasks, GTP outperforms HEFT by 13%. Complementary information shows that for $SCE(20,500,0)$, GTP needed an average of 9 remappings (Figure 5.11) and an average of 380 migrated tasks (Figure 5.12). In the same manner, DLS/sr tends to outperform HEFT in most cases. The best performance for DLS/sr is for $SCE(10,1000,0)$, where DLS/sr outperforms HEFT by 4%, requiring an average of 300 remappings and 1400 migrated tasks. This means that GTP and DLS/sr, at each RP, reacted to inaccurate estimation (caused mainly by the internal factors) in the previous schedule and ob-

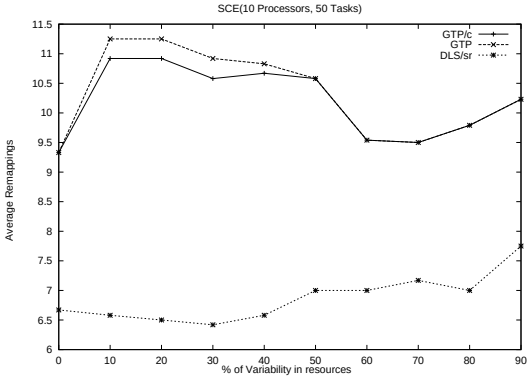

SCE(5 Processors, 50 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources

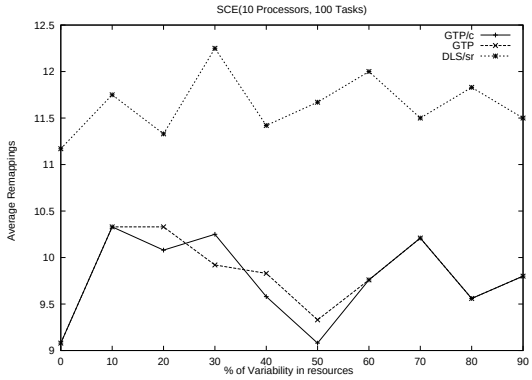

SCE(5 Processors, 100 Tasks)
GTP/c
GTP
Average Remappings
% of Variability in resources

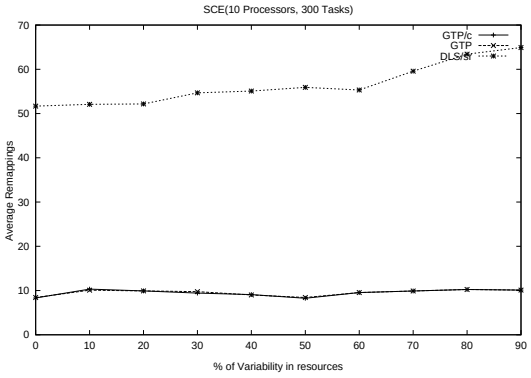

SCE(5 Processors, 300 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources

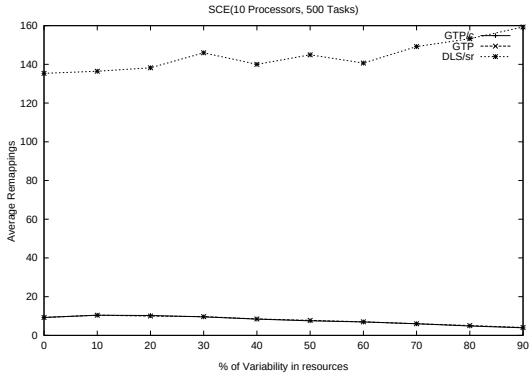

SCE(5 Processors, 500 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources

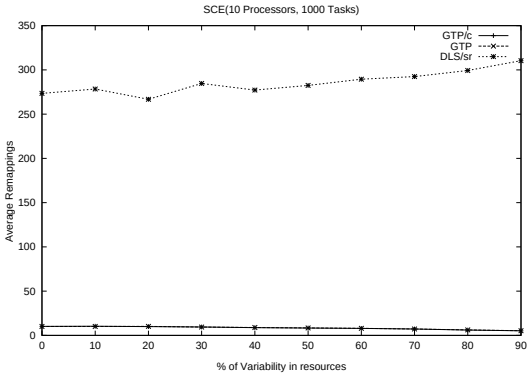

SCE(5 Processors, 1000 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources


SCE(10 Processors, 50 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources


SCE(10 Processors, 100 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources


SCE(10 Processors, 300 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources


SCE(10 Processors, 500 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources


SCE(10 Processors, 1000 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources

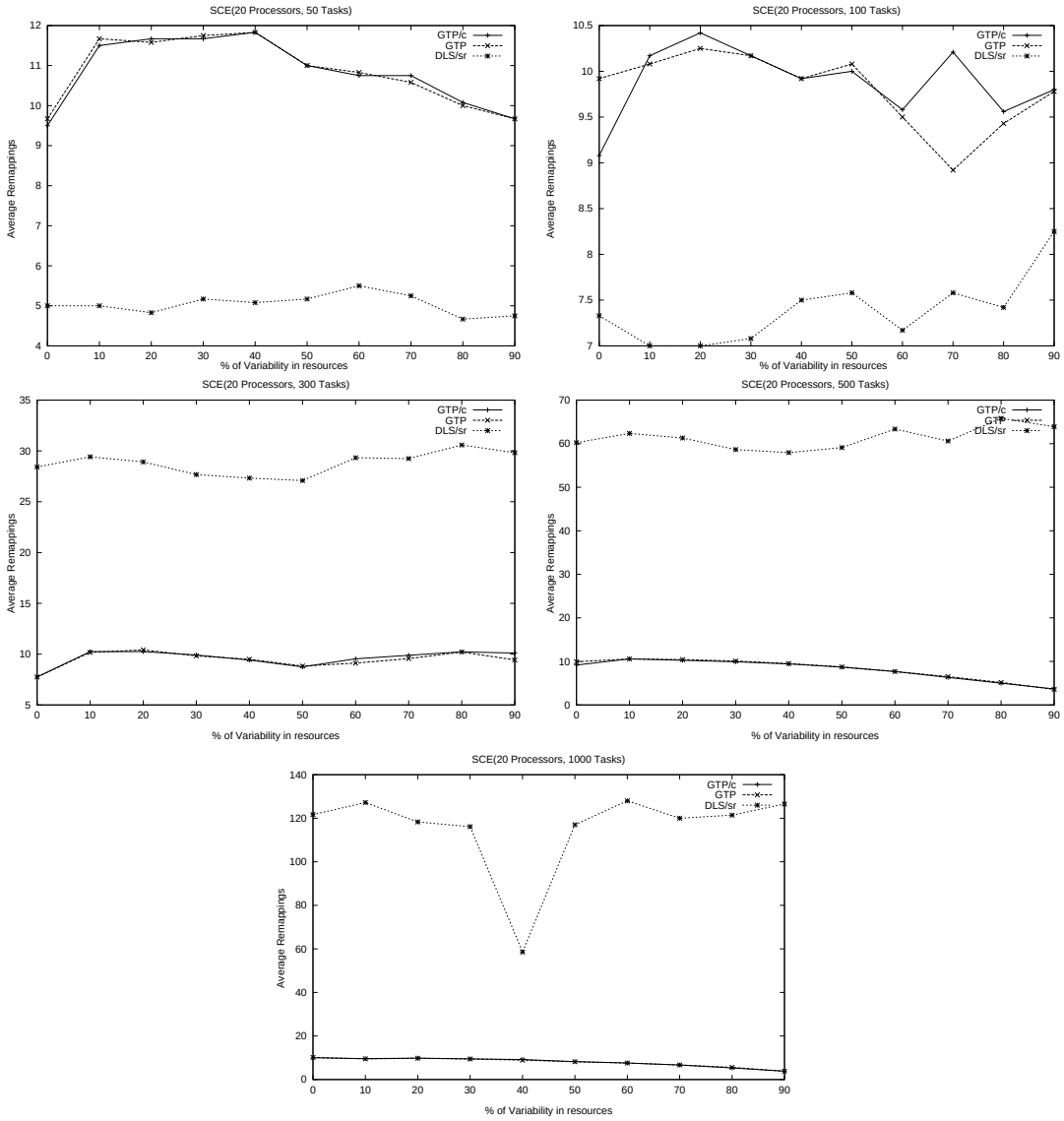
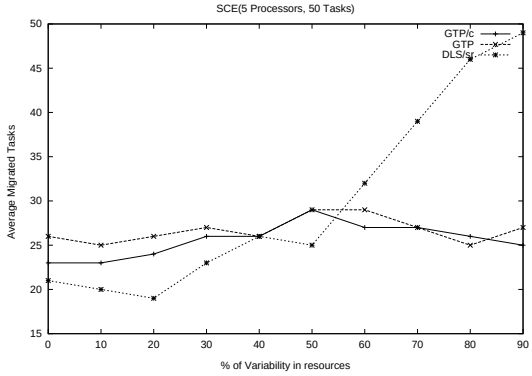
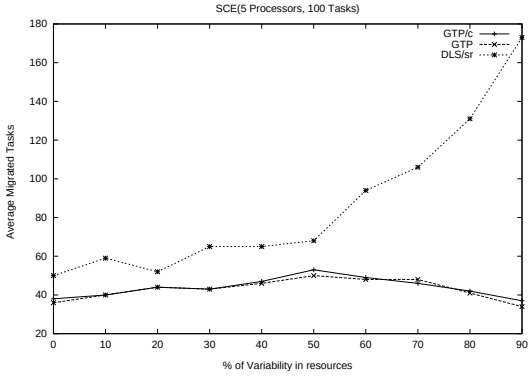


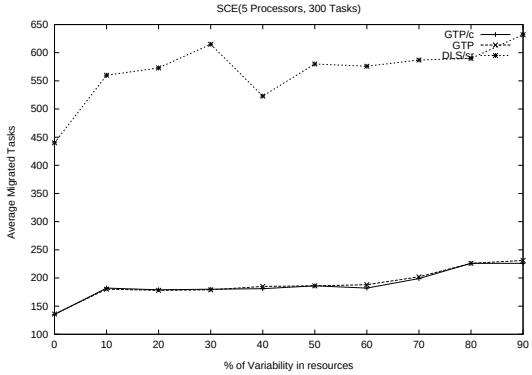
Figure 5.7: Average Remappings for GTP, GTP/c and DLS/sr when CCR=0.5 and variable bandwidth

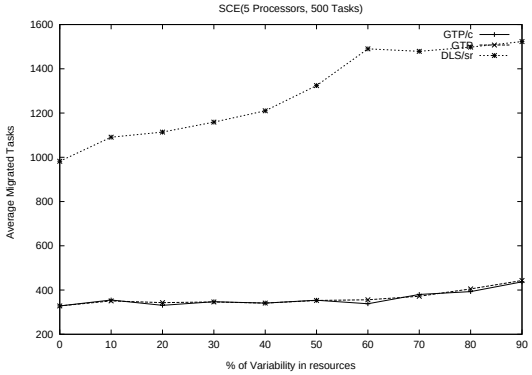
tained a refined schedule considering the progress of the application on unchanging environments, which increased the performance of the application compared with $HEFT$. Obviously, the decision to migrate a placed task will incur migration cost because retransmission of data is needed.

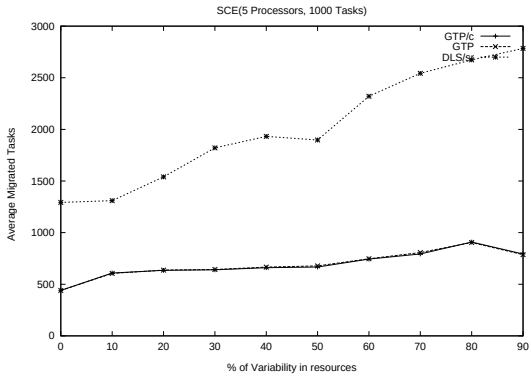
- For more SHCS-like scenarios, GTP and DLS/sr outperform $HEFT$ in most of the cases. In Figure 5.10, we can observe that for all the scenarios, the average NSL for HEFT, tends to increase considerably compared with the previous scenarios. This tendency is gradually incremented more than GTP and DLS/sr, as the variability increases. Thus, in scenarios with

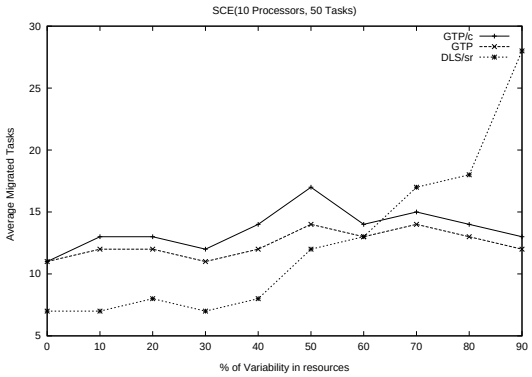

SCE(5 Processors, 50 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources

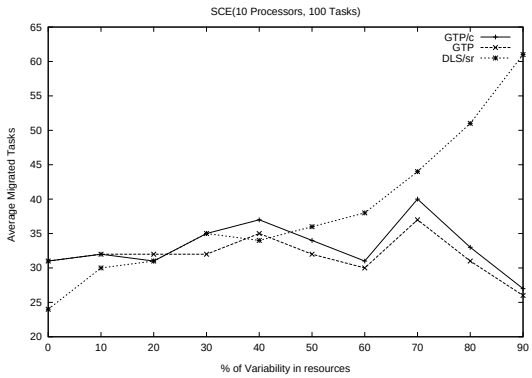

SCE(5 Processors, 100 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources

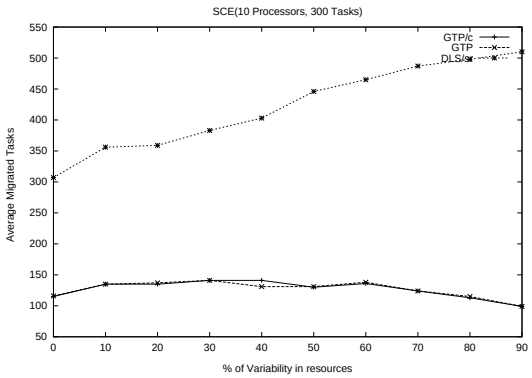

SCE(5 Processors, 300 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources

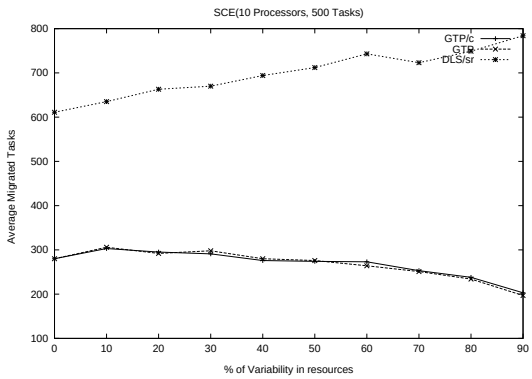

SCE(5 Processors, 500 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources

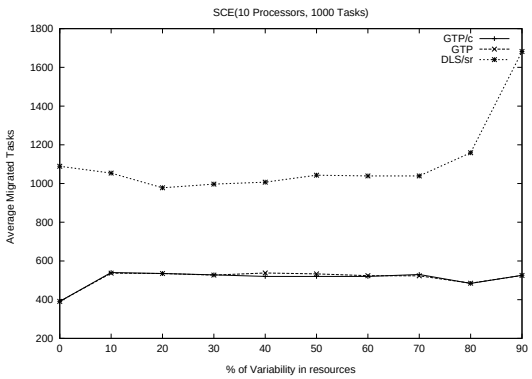

SCE(5 Processors, 1000 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources


SCE(10 Processors, 50 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources


SCE(10 Processors, 100 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources


SCE(10 Processors, 300 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources


SCE(10 Processors, 500 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources


SCE(10 Processors, 1000 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources

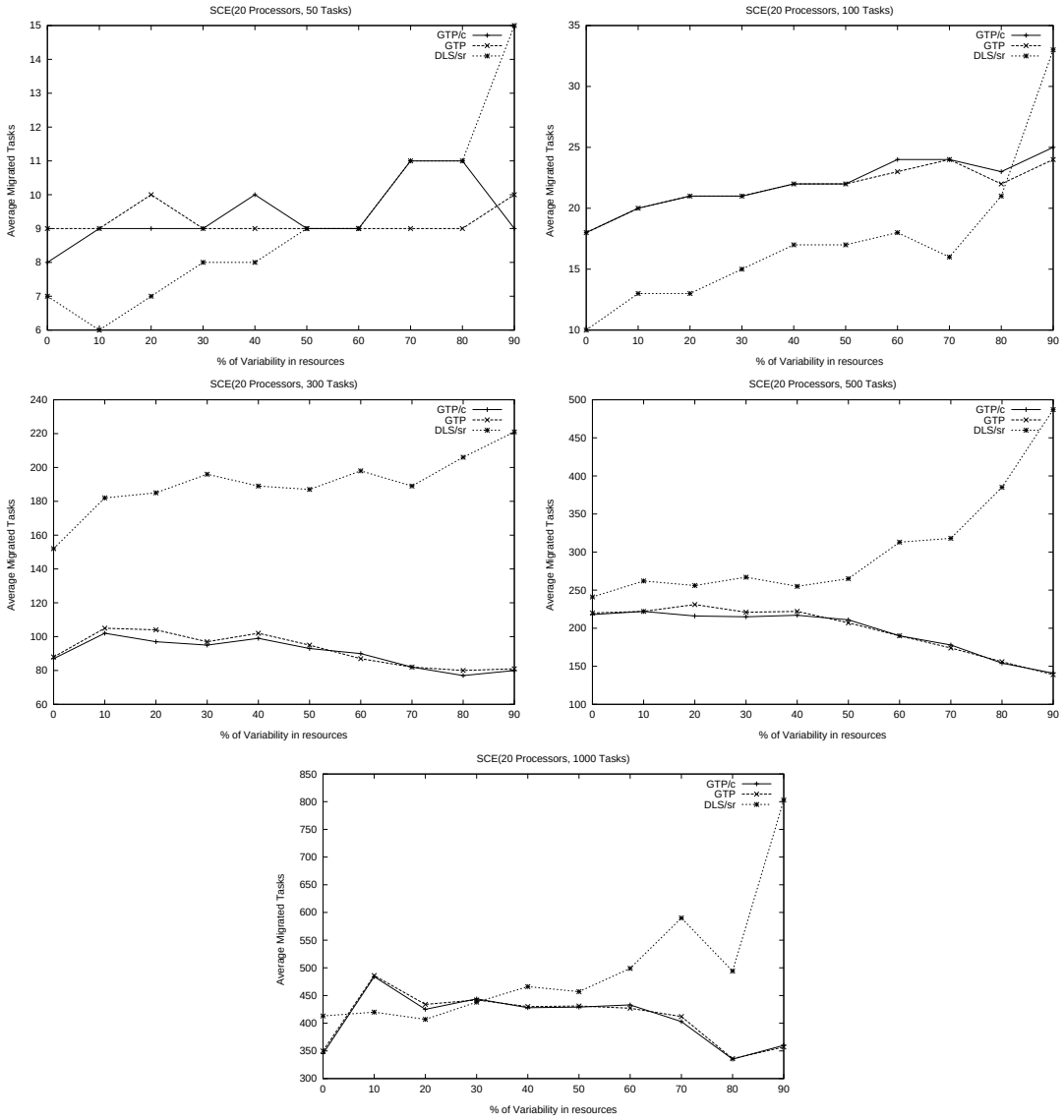
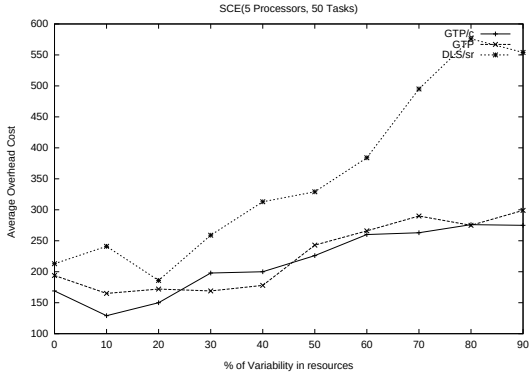


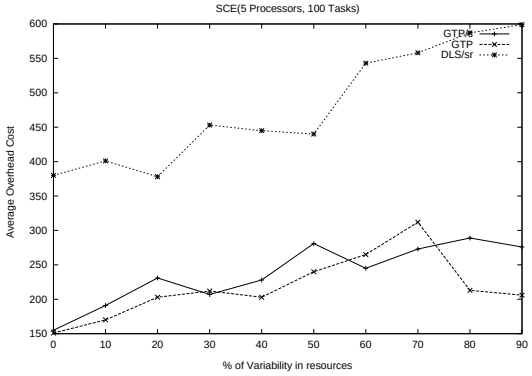
Figure 5.8: Average Migrated Tasks for GTP, GTP/c and DLS/sr when CCR=0.5 and variable bandwidth

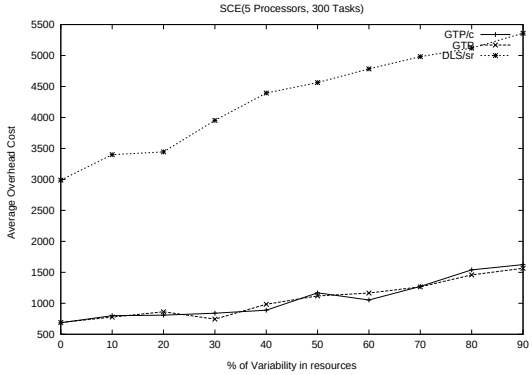
90% of variability, the NSL for HEFT is up to 2.3 times higher than GTP and up to 2.6 times higher than DLS/sr. The reactive strategy allowed GTP and DLS/sr to react more efficiently to resource variability (external factors) and traffic contention (internal factors). However, the pessimistic model used, in which the migrated task must be restarted from the very beginning, including regathering all inputs directly from the predecessors, tends to increase the overhead cost, lengthening the makespan.

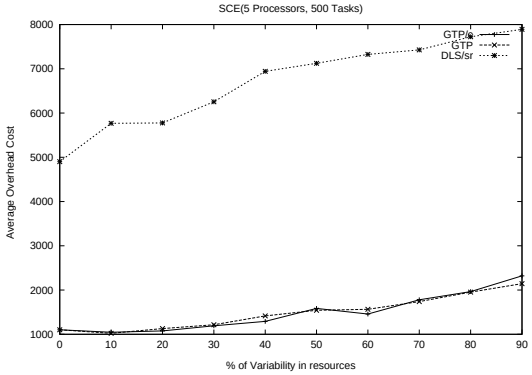
2. *Evaluation of Reactive Strategies*

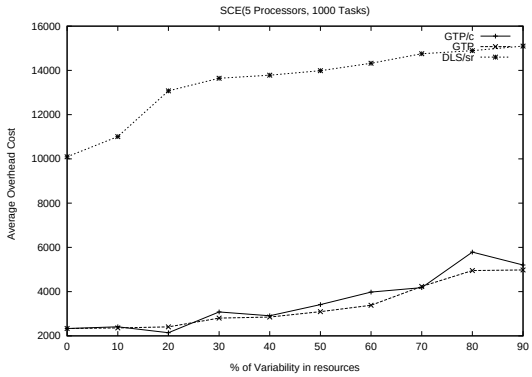
   The experimental results show that GTP outperforms DLS/sr in most cases. We

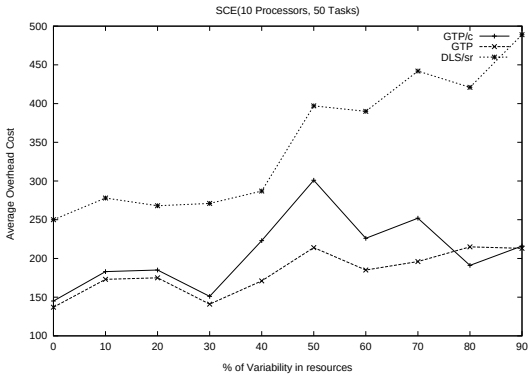

SCE(5 Processors, 50 Tasks)
Average Overhead Cost
% of Variability in resources
GTP/c
GTP
DLS/sr

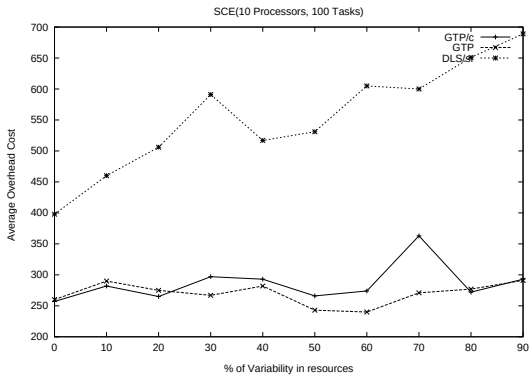

SCE(5 Processors, 100 Tasks)
Average Overhead Cost
% of Variability in resources
GTP/c
GTP
DLS/sr

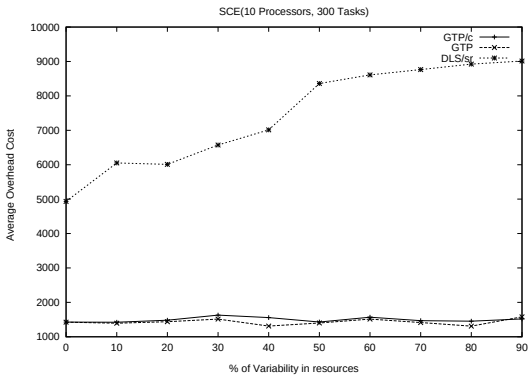

SCE(5 Processors, 300 Tasks)
Average Overhead Cost
% of Variability in resources
GTP/c
GTP
DLS/sr

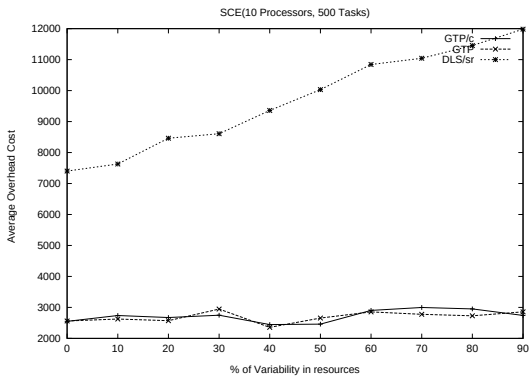

SCE(5 Processors, 500 Tasks)
Average Overhead Cost
% of Variability in resources
GTP/c
GTP
DLS/sr

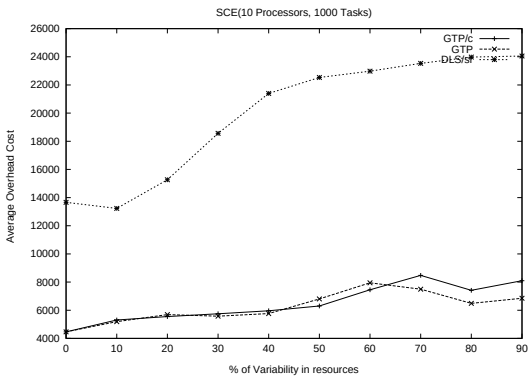

SCE(5 Processors, 1000 Tasks)
Average Overhead Cost
% of Variability in resources
GTP/c
GTP
DLS/sr


SCE(10 Processors, 50 Tasks)
Average Overhead Cost
% of Variability in resources
GTP/c
GTP
DLS/sr


SCE(10 Processors, 100 Tasks)
Average Overhead Cost
% of Variability in resources
GTP/c
GTP
DLS/sr


SCE(10 Processors, 300 Tasks)
Average Overhead Cost
% of Variability in resources
GTP/c
GTP
DLS/sr


SCE(10 Processors, 500 Tasks)
Average Overhead Cost
% of Variability in resources
GTP/c
GTP
DLS/sr


SCE(10 Processors, 1000 Tasks)
Average Overhead Cost
% of Variability in resources
GTP/c
GTP
DLS/sr

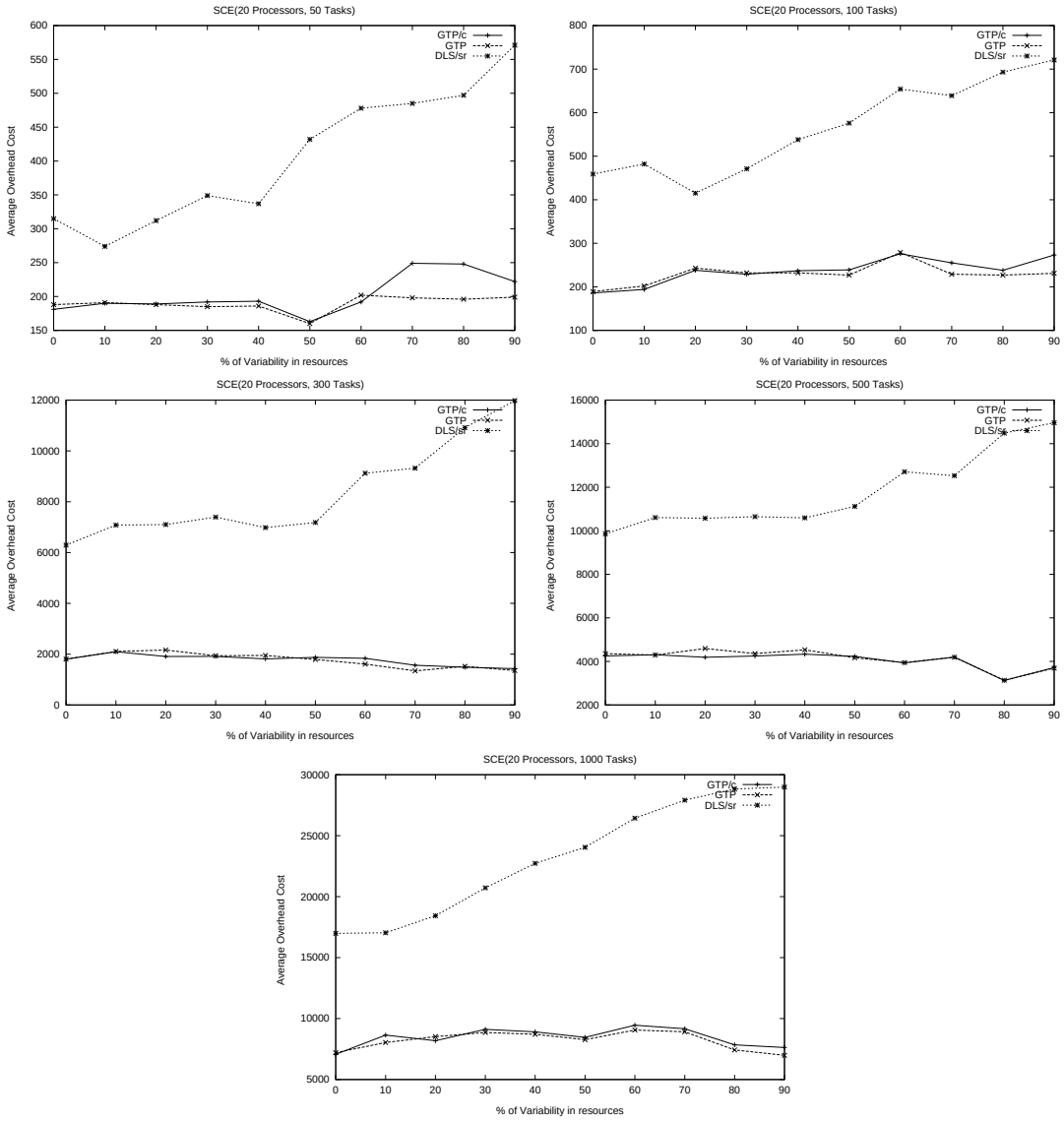
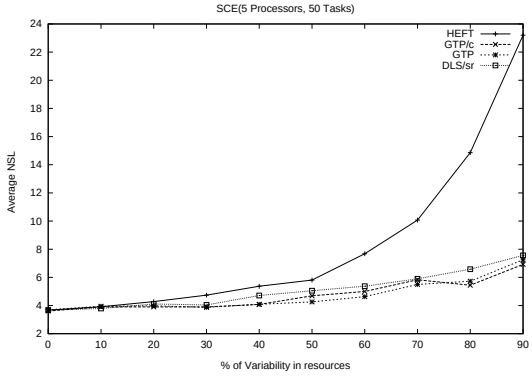


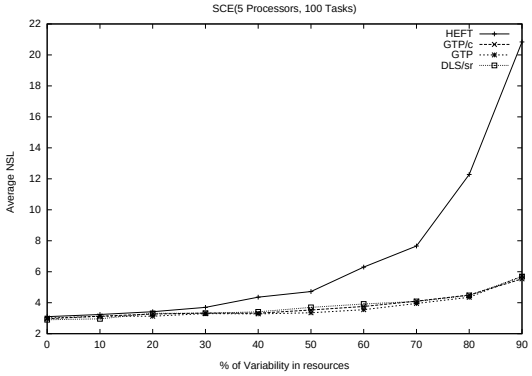
Figure 5.9: Average Overhead Cost for GTP, GTP/c and DLS/sr when CCR=0.5 and variable bandwidth

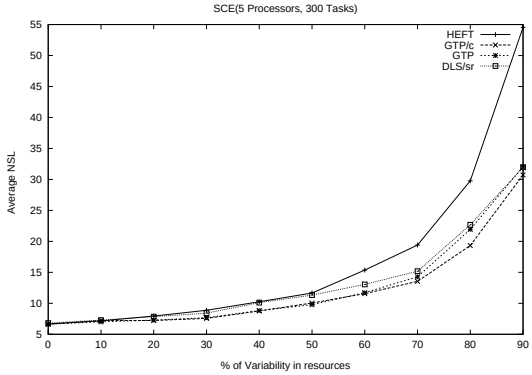
can see in Figure 5.10, that those cases in which DLS/sr outperforms GTP in terms of the average NSL, mainly involve scenarios with a low variability in resources and DAGs applications with relatively few tasks (50 and 100 tasks). For instance, $SCE(5,100,10)$ shows that the average NSL for DLS/sr is up to 7% less than the average NSL for GTP. By observing the experimental results obtained in the scheduling scenarios, we observe that GTP outperforms DLS/sr in most cases. We believe that there are two main contributing factors: a) The first factor concerns the prediction of the spare time of tasks, which may be affected by the external and internal factors described previously. For instance, ignoring

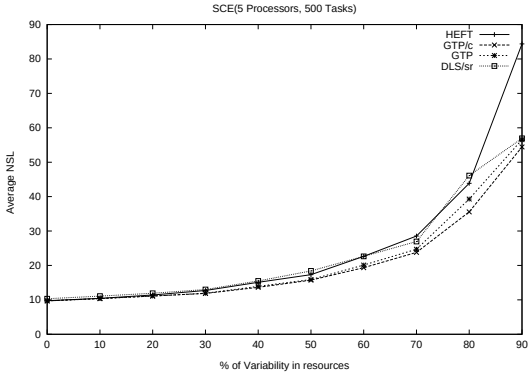
traffic contention in the prediction of the spare time of tasks. This can be seen in the first scheduling scenario where the traffic contention was practically null. In this scenario we observed that the performance of GTP and DLS/sr was similar. However, as the communication cost was gradually increased in the next scheduling scenarios, GTP outperformed DLS/sr in most cases. b) The second factor concerns the criterion to apply the selective rescheduling policy considered in our benchmark, which dictates that the spare time of tasks are evaluated when a task finishes execution. Thus, the combination of both factors will directly affect the performance of DLS/sr, which will affect the accuracy of the spare time of tasks. As a consequence, the number of rescheduling points will increase, leading to more migrated tasks, leading to a higher overhead which may affect the final makespan of the application. We can observe this, in this scenario, when the DAGs become larger and complex (300, 500 and 1000 tasks). For instance, in Figure 5.10, we observe that for $SCE(20,1000,40)$ GTP outperformed DLS/sr by an average of 13%, requiring an average of 9 remappings (see Figure 5.11) and an average of 620 migrated tasks (see Figure 5.12). On the other hand, DLS/sr required an average of 60 remappings and an average of 1400 migrated tasks generating 60% more overhead cost (recomputation and retransmitting) than GTP (see Figure 5.13). Another important issue that we will explore in the Section 5.6, is related to the frequency of the rescheduling points (RP's), which may affect the final makespan. Intuitively, many RP's will increase the overhead cost of the application, but very few RP's will allow internal and external factors to negatively impact the makespan.

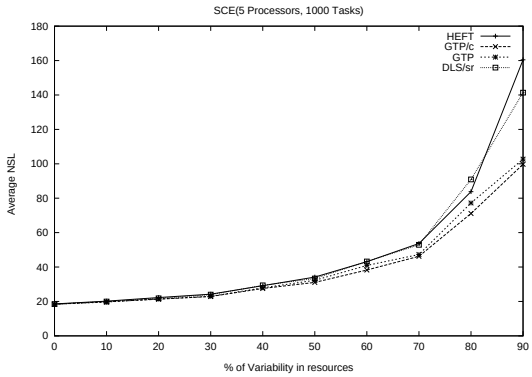
In general terms, the reactive mapping methods tend to have a better performance than those considering static schedules, in the presence of traffic contention and resource variability. We observe that reactive approaches allow the application to react to both variability in resources and inaccurate estimations from previous schedules. This means that, in some cases, reactive approaches may have a better performance than static approaches even in environments with dedicated and unchanging resources. This is mainly observed when the number of tasks and data transfers is increased.
We note that the assignment policy used in GTP, which allows a task to be mapped onto that processor offering the minimum earliest finish time, may contribute to the relatively high number of migrated tasks, as it allows migration of tasks even if the predicted time saved is small. We believe that by improving the assignment policy, the number of migrated tasks may reduce, thus increasing the application performance.

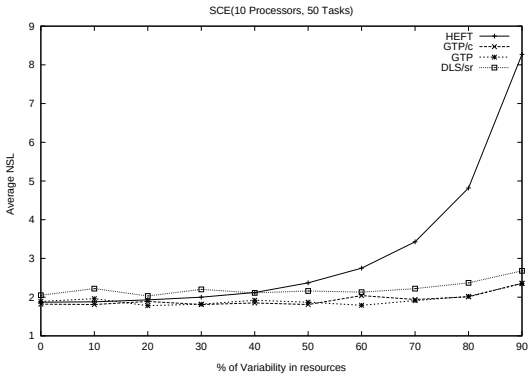

SCE(5 Processors, 50 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources

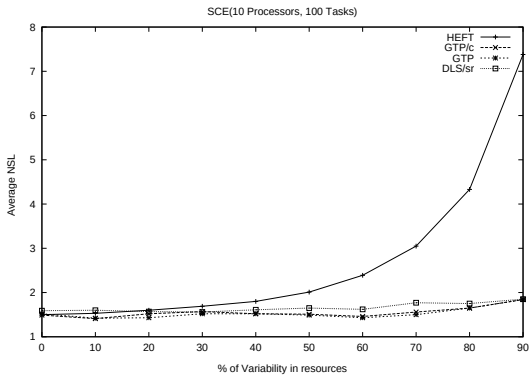

SCE(5 Processors, 100 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources

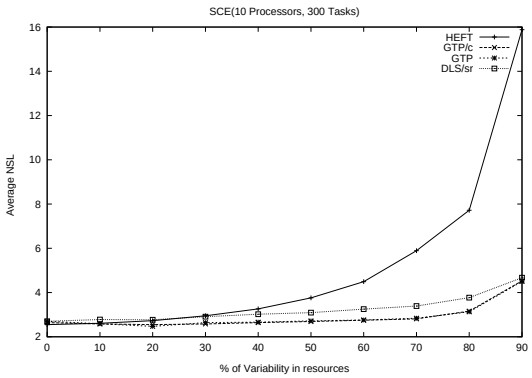

SCE(5 Processors, 300 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources

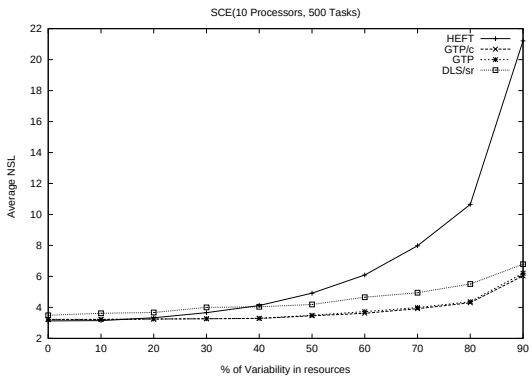

SCE(5 Processors, 500 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources

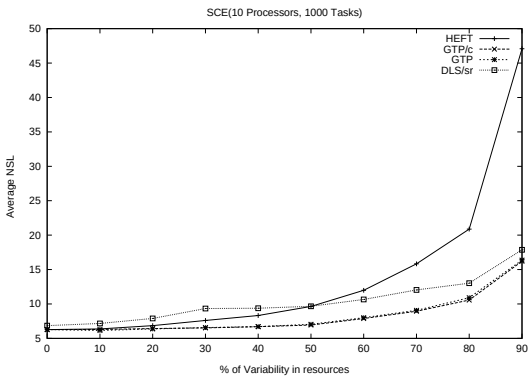

SCE(5 Processors, 1000 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources


SCE(10 Processors, 50 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources


SCE(10 Processors, 100 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources


SCE(10 Processors, 300 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources


SCE(10 Processors, 500 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources


SCE(10 Processors, 1000 Tasks)
HEFT
GTP/c
GTP
DLS/sr
Average NSL
% of Variability in resources

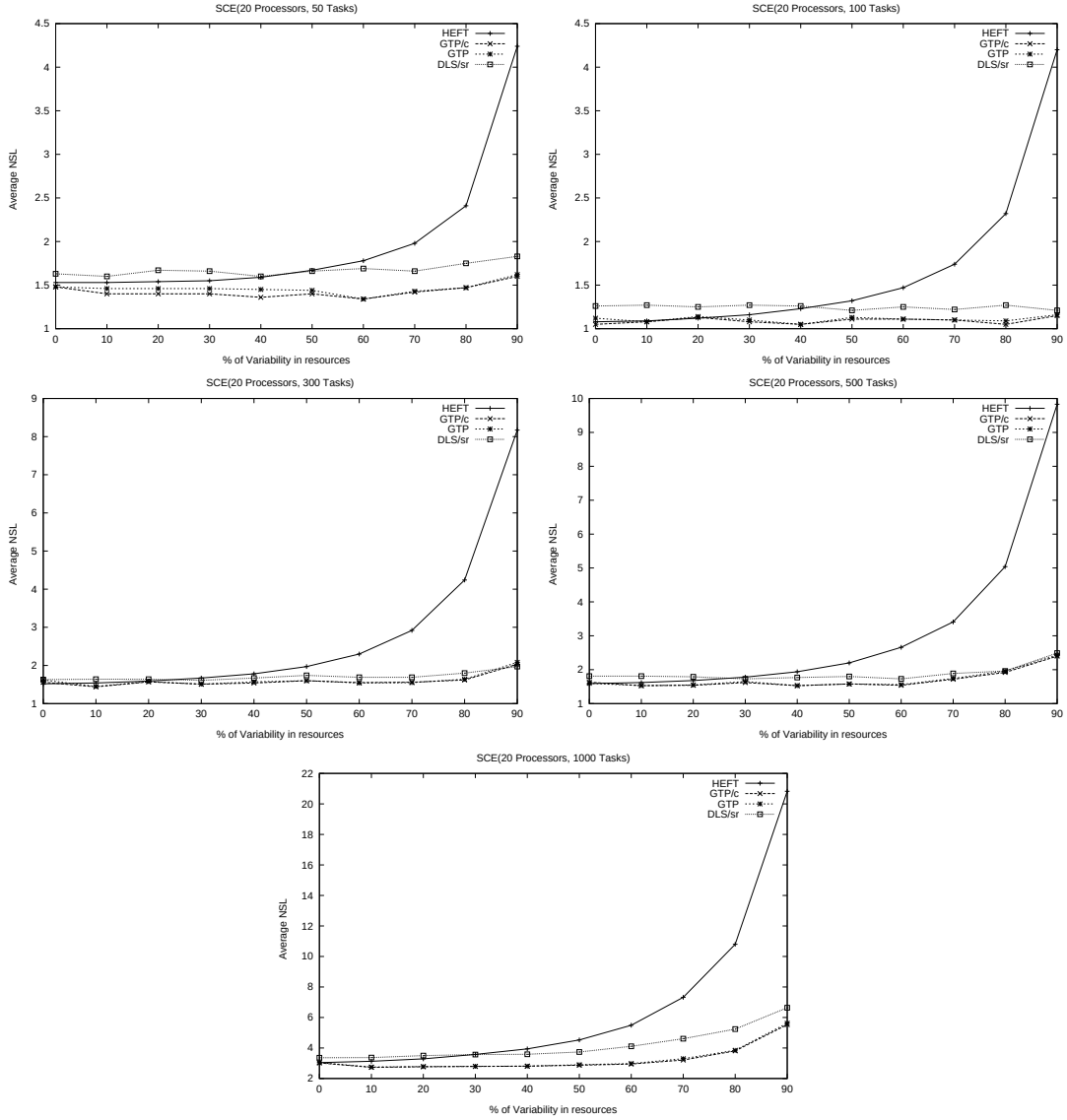
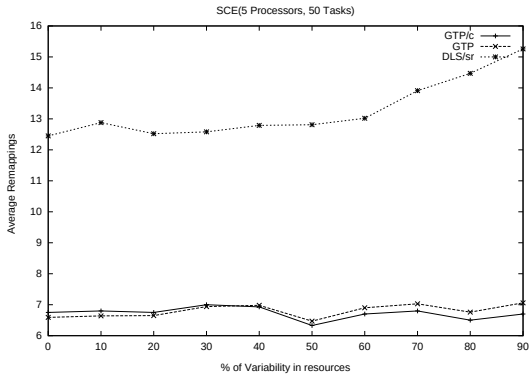
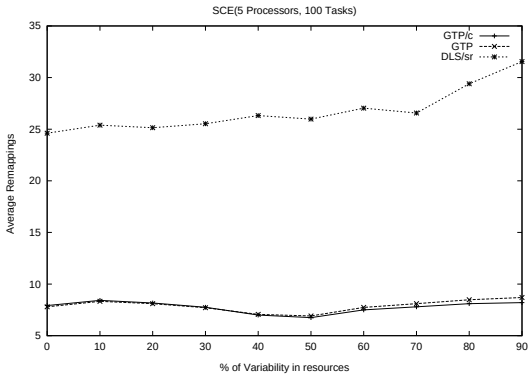


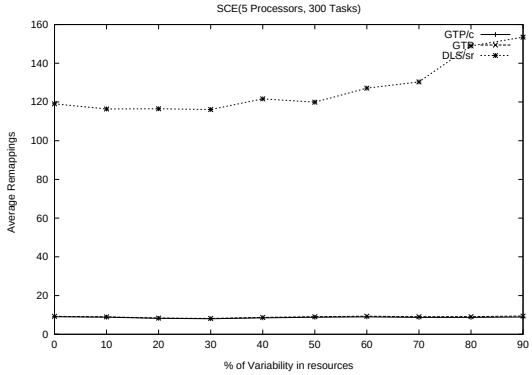
Figure 5.10: Average NSL for GTP, GTP/c and DLS/sr when CCR=1.5 and variable bandwidth

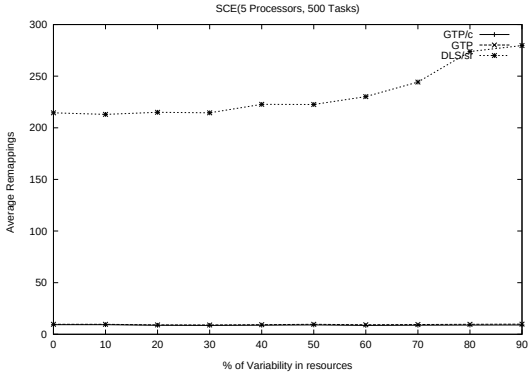
5.5 Reactive Scheduling with Copying and Migration

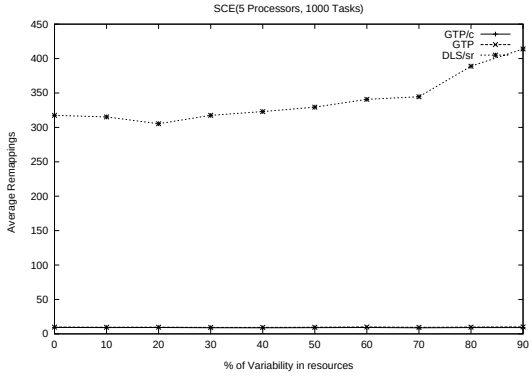
In this section we show and evaluate the performance results for the GTP/c system, an extended version of the GTP system. With GTP/c we observed that in an execution with relatively frequent migration, it may be that, over time, the results of some task have been copied to several other nodes, and so 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. Thus, we first discuss the monitoring of data flow among tasks within

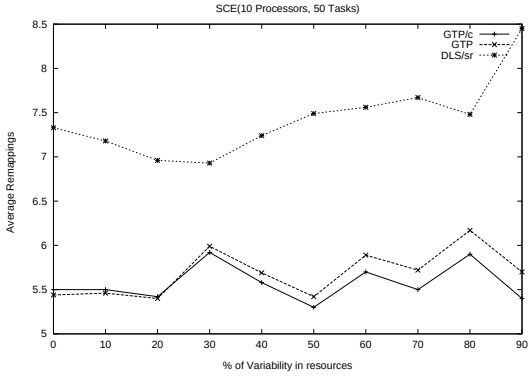

SCE(5 Processors, 50 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources

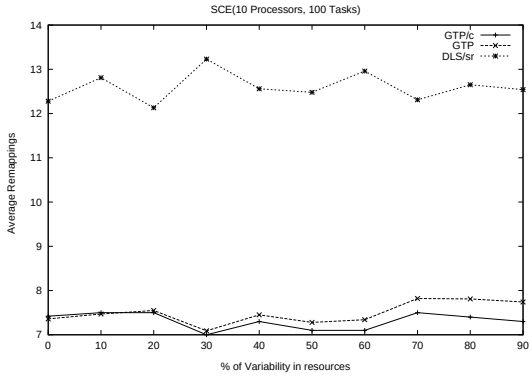

SCE(5 Processors, 100 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources

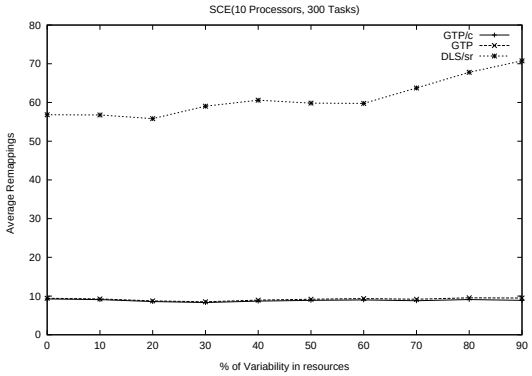

SCE(5 Processors, 300 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources

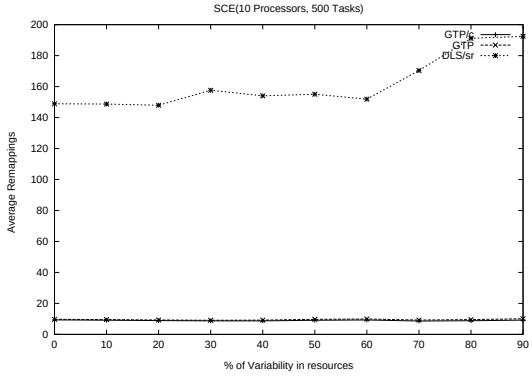

SCE(5 Processors, 500 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources

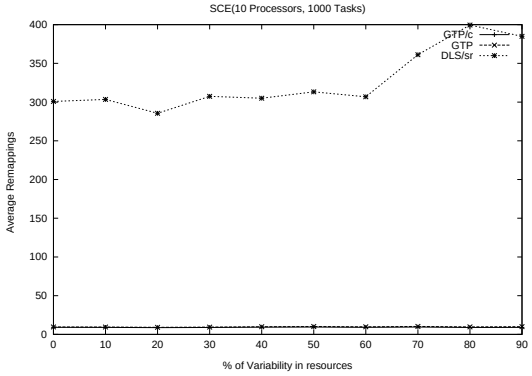

SCE(5 Processors, 1000 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources


SCE(10 Processors, 50 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources


SCE(10 Processors, 100 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources


SCE(10 Processors, 300 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources


SCE(10 Processors, 500 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources


SCE(10 Processors, 1000 Tasks)
GTP/c
GTP
DLS/sr
Average Remappings
% of Variability in resources

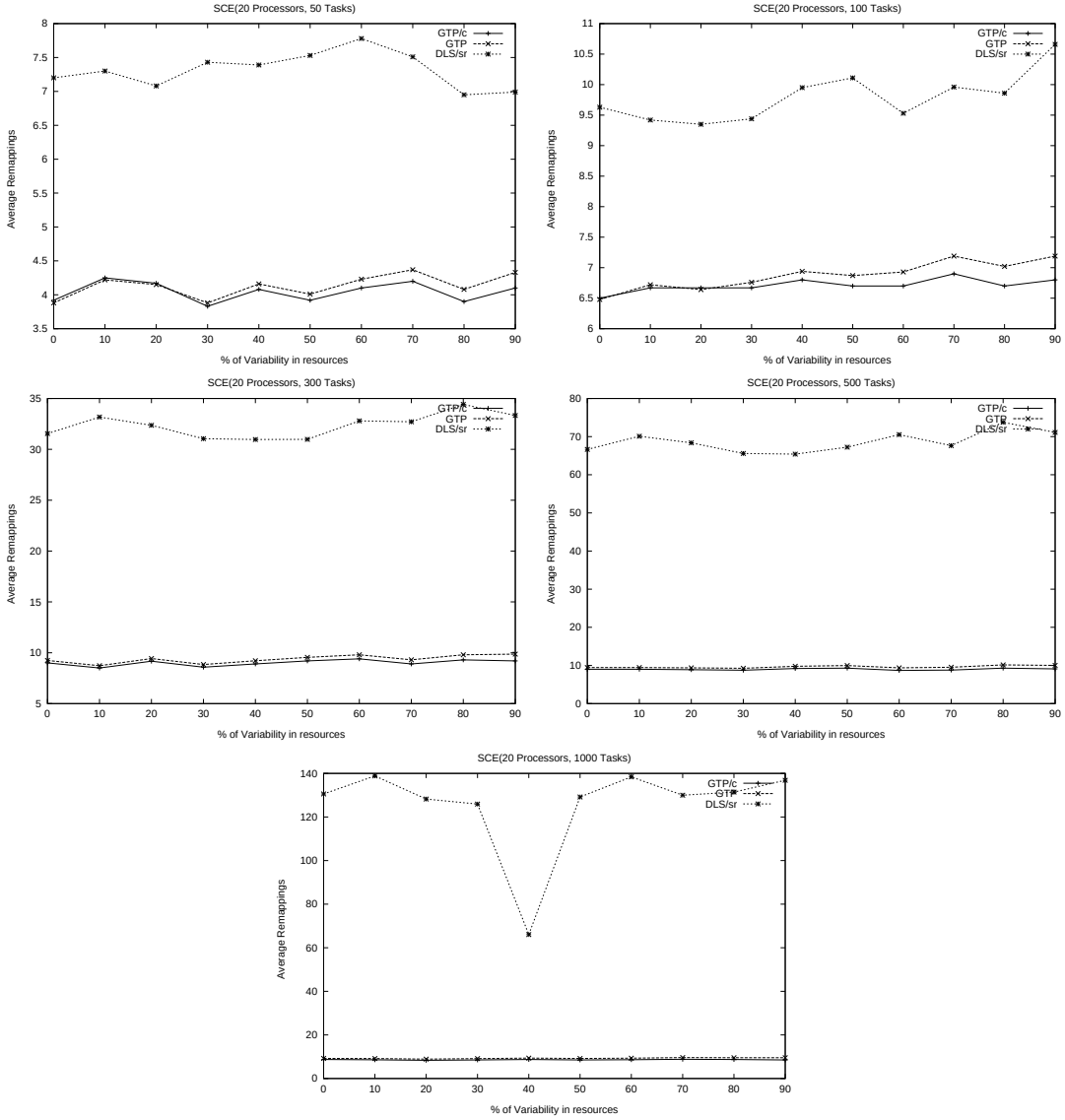
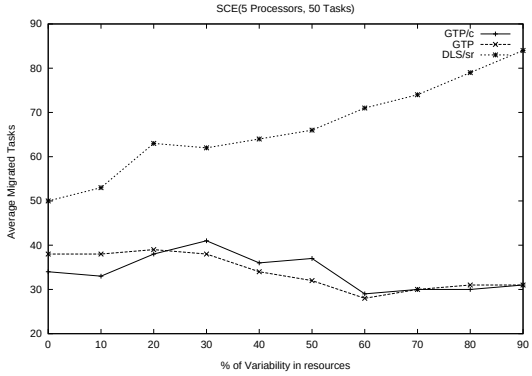
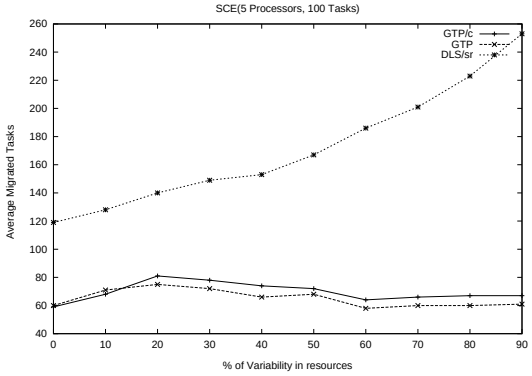
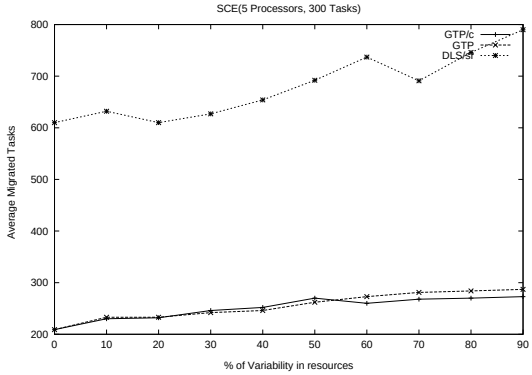


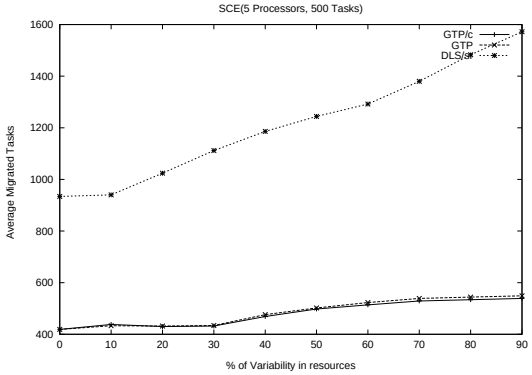
Figure 5.11: Average Remappings for GTP, GTP/c and DLS/sr when CCR=1.5 and variable bandwidth

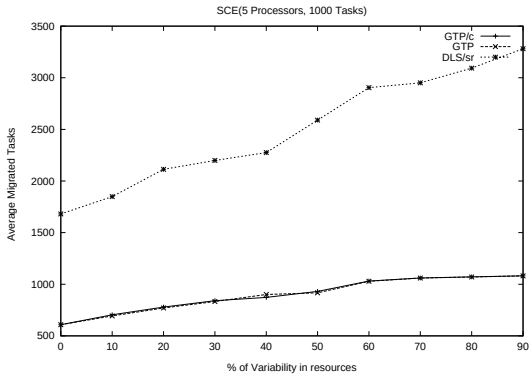
the context of GTP and GTP/c. The information monitored embraces the following variables: The *Total Data Transfers*, $|e_k(i,j)| : e_k \in E$ shows the average of the total number of expected data transfers for each DAG application considered in our experiments. The *Data Transfers Used* $|e_k(i,j)| : e_k \in E$ *and* $\kappa^d(v_i,v_j) = 1$ describes the average of the numbers of data transfers which used bandwidth capabilities to perform the data transfer. Obviously, if we subtract the *Data Transfers Used* from the *Total Data Transfers* we will obtain the average of the *Intra-processors Data Transfers*, for which the communication cost is considered negligible. The *Copies Generated* $|\Omega_k(e(v_i,v_j)) : \Omega_k \in \Omega|$ represents the average number of copies generated during ex-

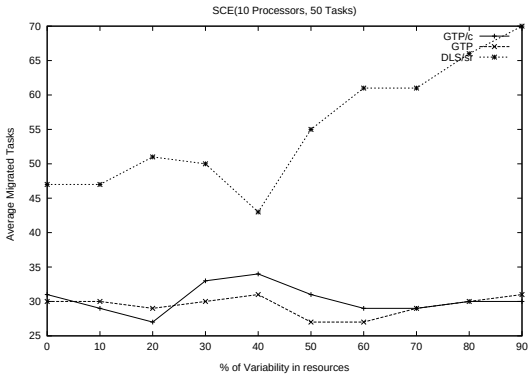

SCE(5 Processors, 50 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources

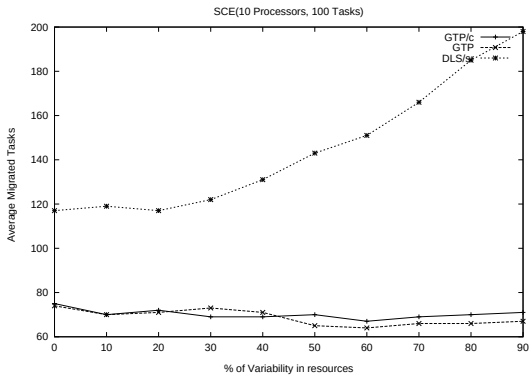

SCE(5 Processors, 100 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources

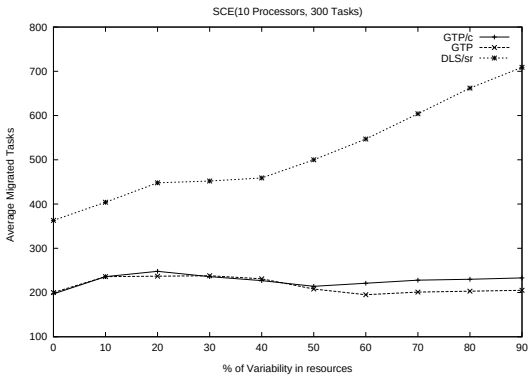

SCE(5 Processors, 300 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources

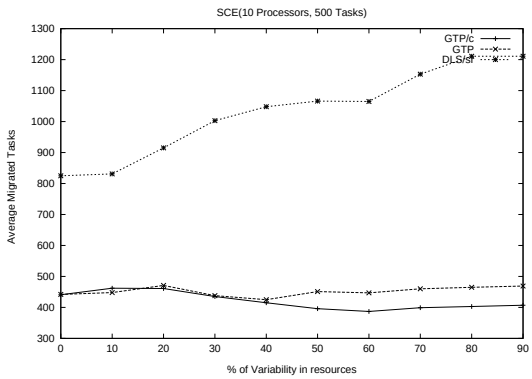

SCE(5 Processors, 500 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources

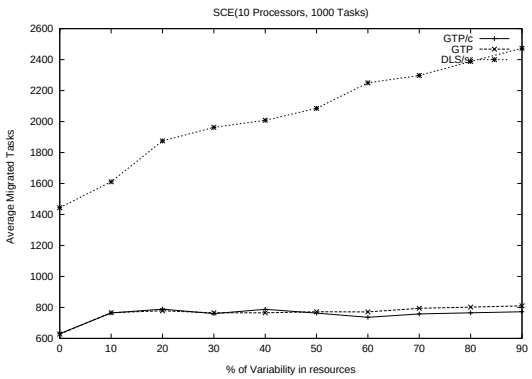

SCE(5 Processors, 1000 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources


SCE(10 Processors, 50 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources


SCE(10 Processors, 100 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources


SCE(10 Processors, 300 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources


SCE(10 Processors, 500 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources


SCE(10 Processors, 1000 Tasks)
GTP/c
GTP
DLS/sr
Average Migrated Tasks
% of Variability in resources

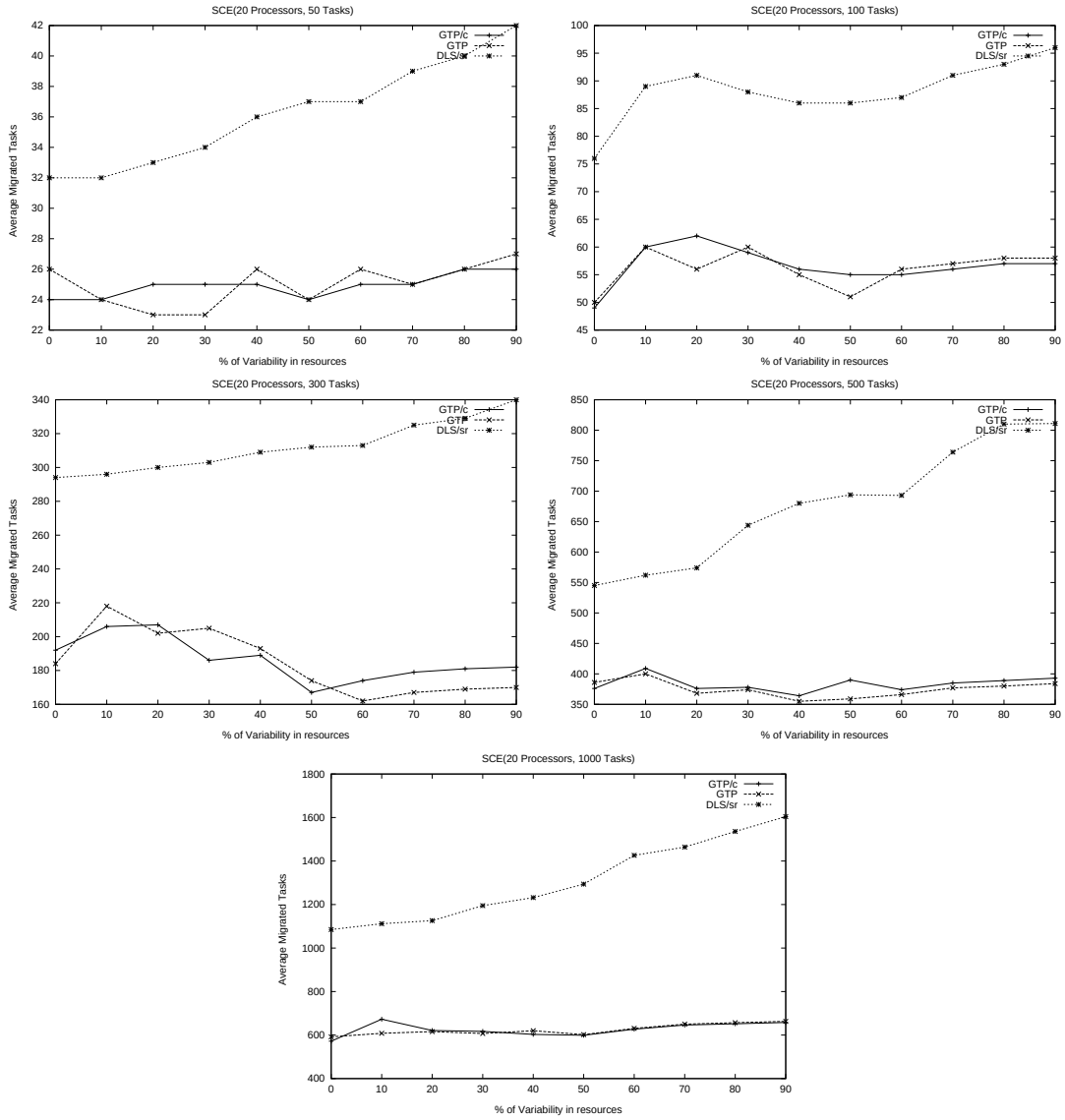
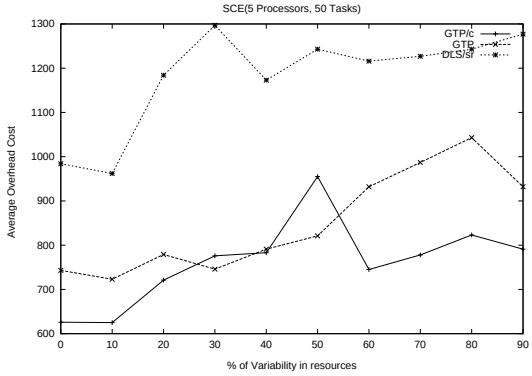
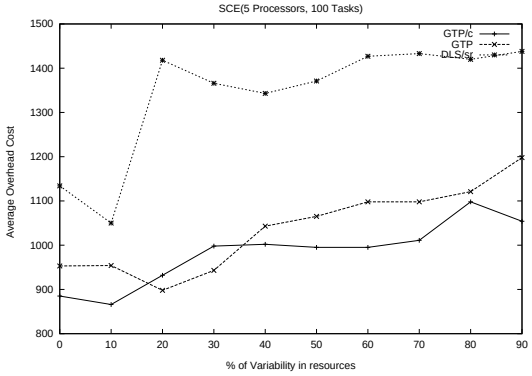


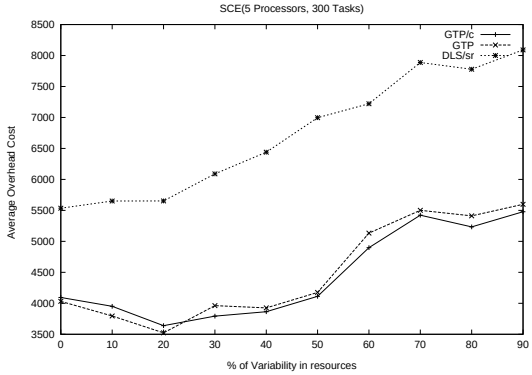
Figure 5.12: Average Migrated Tasks for GTP, GTP/c and DLS/sr when CCR=1.5 and variable bandwidth

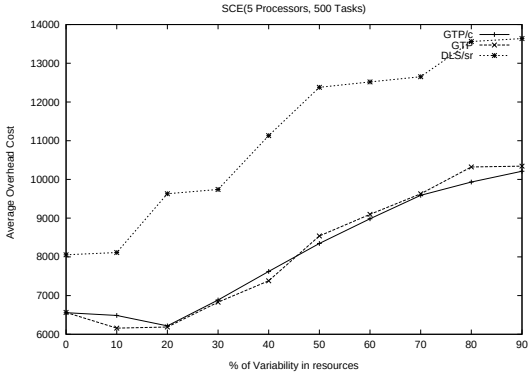
ecution as a consequence of the migration policy defined for GTP. We recall that a particular edge may have several copies. The *Copies Used* describes the average of the copies that were used directly as an input for a particular $e(i, j)$. We notice that those *Copies Used* which used bandwidth capabilities are included in the set of *Data Transfers Used*.

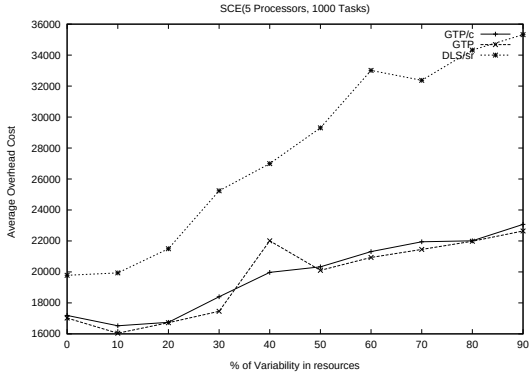
To conduct our experiments, we will use the characteristics of the second scenario, for which CCR = 0.5 and changing bandwidth over time, with the maximum bandwidth equal to one unit of data per unit of time. In Figure 5.14 we observe that a number of copies were generated during execution as a result of the migration policy defined for

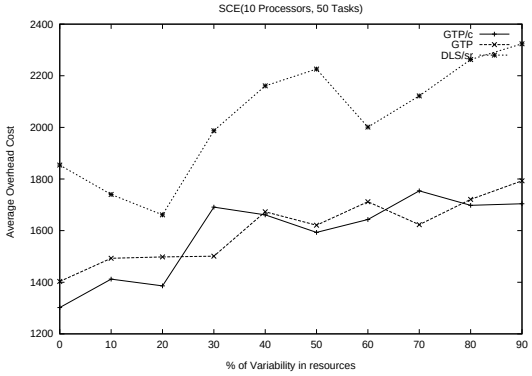

SCE(5 Processors, 50 Tasks)
GTP/c
GTP
DLS/sr
Average Overhead Cost
% of Variability in resources

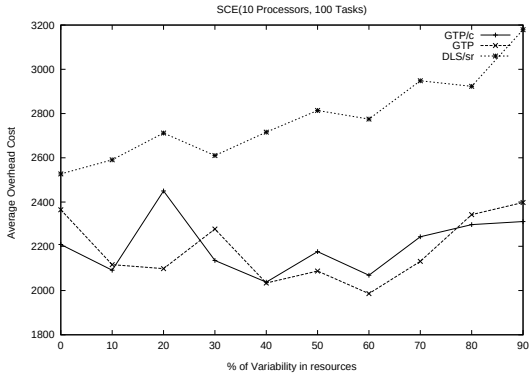

SCE(5 Processors, 100 Tasks)
GTP/c
GTP
DLS/sr
Average Overhead Cost
% of Variability in resources

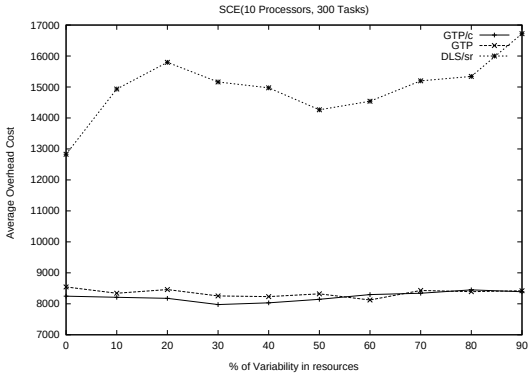

SCE(5 Processors, 300 Tasks)
GTP/c
GTP
DLS/sr
Average Overhead Cost
% of Variability in resources

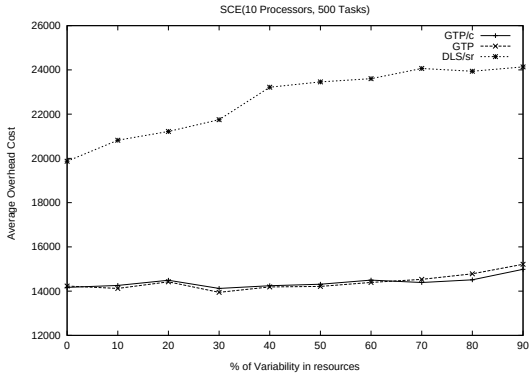

SCE(5 Processors, 500 Tasks)
GTP/c
GTP
DLS/sr
Average Overhead Cost
% of Variability in resources

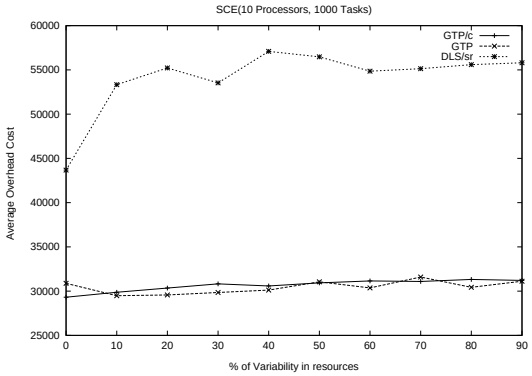

SCE(5 Processors, 1000 Tasks)
GTP/c
GTP
DLS/sr
Average Overhead Cost
% of Variability in resources


SCE(10 Processors, 50 Tasks)
GTP/c
GTP
DLS/sr
Average Overhead Cost
% of Variability in resources


SCE(10 Processors, 100 Tasks)
GTP/c
GTP
DLS/sr
Average Overhead Cost
% of Variability in resources


SCE(10 Processors, 300 Tasks)
GTP/c
GTP
DLS/sr
Average Overhead Cost
% of Variability in resources


SCE(10 Processors, 500 Tasks)
GTP/c
GTP
DLS/sr
Average Overhead Cost
% of Variability in resources


SCE(10 Processors, 1000 Tasks)
GTP/c
GTP
DLS/sr
Average Overhead Cost
% of Variability in resources

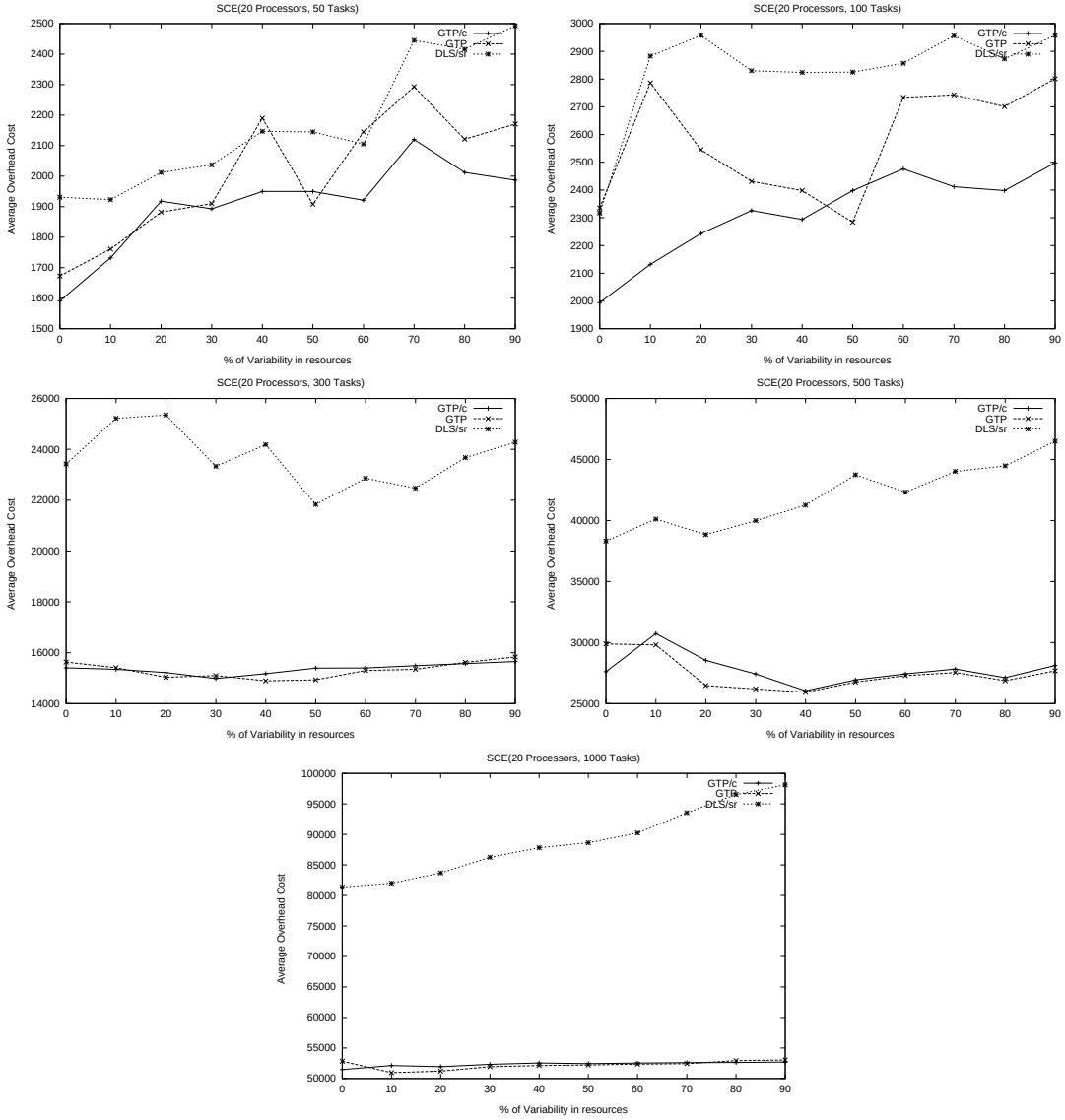
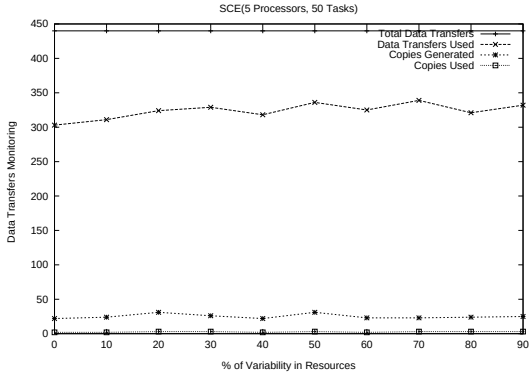
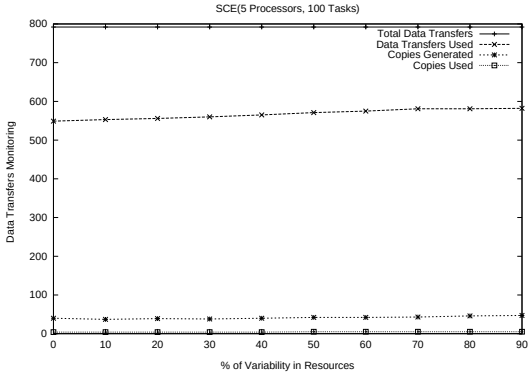
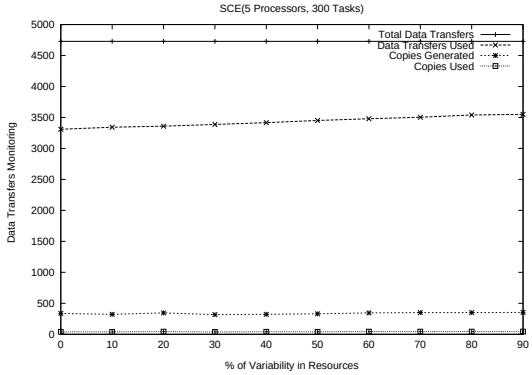


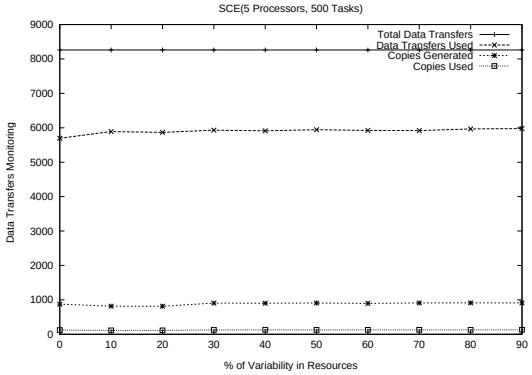
Figure 5.13: Average Overhead Cost for GTP, GTP/c and DLS/sr when CCR=1.5 and variable bandwidth

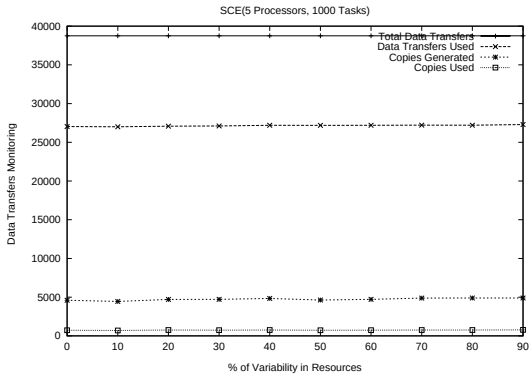
GTP. For instance, the average number of copies generated for $SCE(5,300,30)$ is 317, representing 7% of the average of the *Total Data Transfers*, and an average of 38 copies were used as direct input for some particular tasks, representing 12% of the number of the copies generated. Our experimental results show that in general terms, the average number of copies generated ranged from 5% to 20% of the average of the total data transfers and the average number of copies used ranged from 10% to 19% of the average number of copies generated. The minimum values mainly correspond to those applications with few tasks (50,100), increasing in range as the number of tasks increases (300,500,1000). Despite the low percentage in the number of copies used as

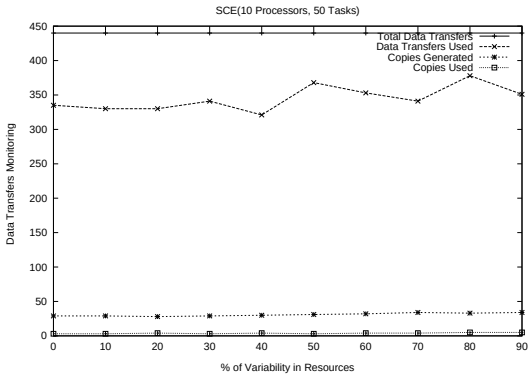

SCE(5 Processors, 50 Tasks)
Total Data Transfers
Data Transfers Used
Copies Generated
Copies Used
Data Transfers Monitoring
% of Variability in Resources

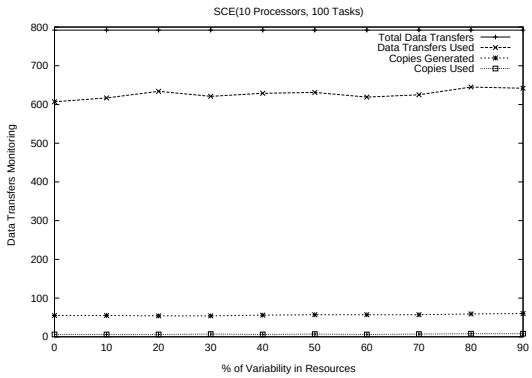

SCE(5 Processors, 100 Tasks)
Total Data Transfers
Data Transfers Used
Copies Generated
Copies Used
Data Transfers Monitoring
% of Variability in Resources

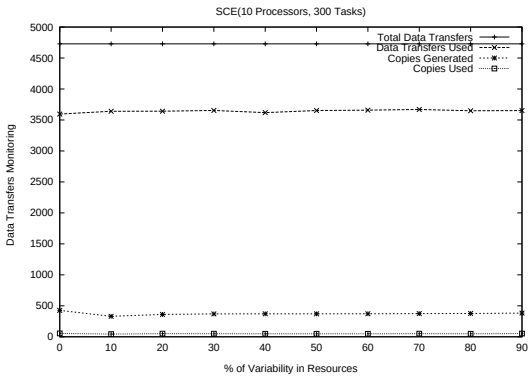

SCE(5 Processors, 300 Tasks)
Total Data Transfers
Data Transfers Used
Copies Generated
Copies Used
Data Transfers Monitoring
% of Variability in Resources

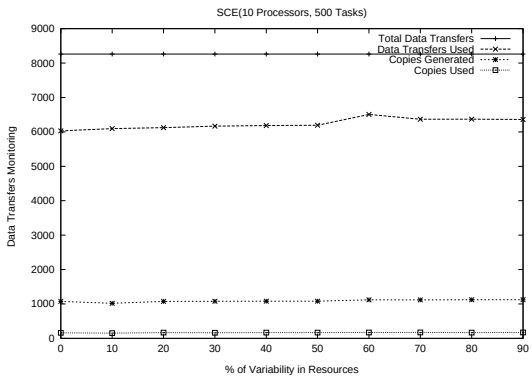

SCE(5 Processors, 500 Tasks)
Total Data Transfers
Data Transfers Used
Copies Generated
Copies Used
Data Transfers Monitoring
% of Variability in Resources

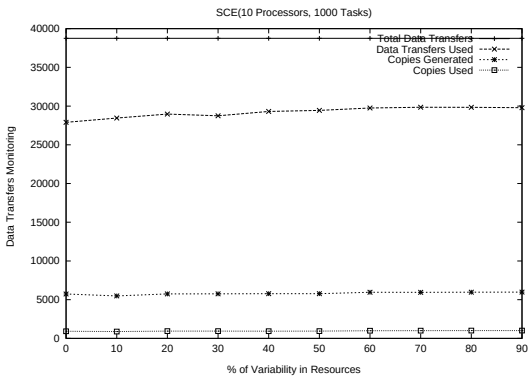

SCE(5 Processors, 1000 Tasks)
Total Data Transfers
Data Transfers Used
Copies Generated
Copies Used
Data Transfers Monitoring
% of Variability in Resources


SCE(10 Processors, 50 Tasks)
Total Data Transfers
Data Transfers Used
Copies Generated
Copies Used
Data Transfers Monitoring
% of Variability in Resources


SCE(10 Processors, 100 Tasks)
Total Data Transfers
Data Transfers Used
Copies Generated
Copies Used
Data Transfers Monitoring
% of Variability in Resources


SCE(10 Processors, 300 Tasks)
Total Data Transfers
Data Transfers Used
Copies Generated
Copies Used
Data Transfers Monitoring
% of Variability in Resources


SCE(10 Processors, 500 Tasks)
Total Data Transfers
Data Transfers Used
Copies Generated
Copies Used
Data Transfers Monitoring
% of Variability in Resources


SCE(10 Processors, 1000 Tasks)
Total Data Transfers
Data Transfers Used
Copies Generated
Copies Used
Data Transfers Monitoring
% of Variability in Resources

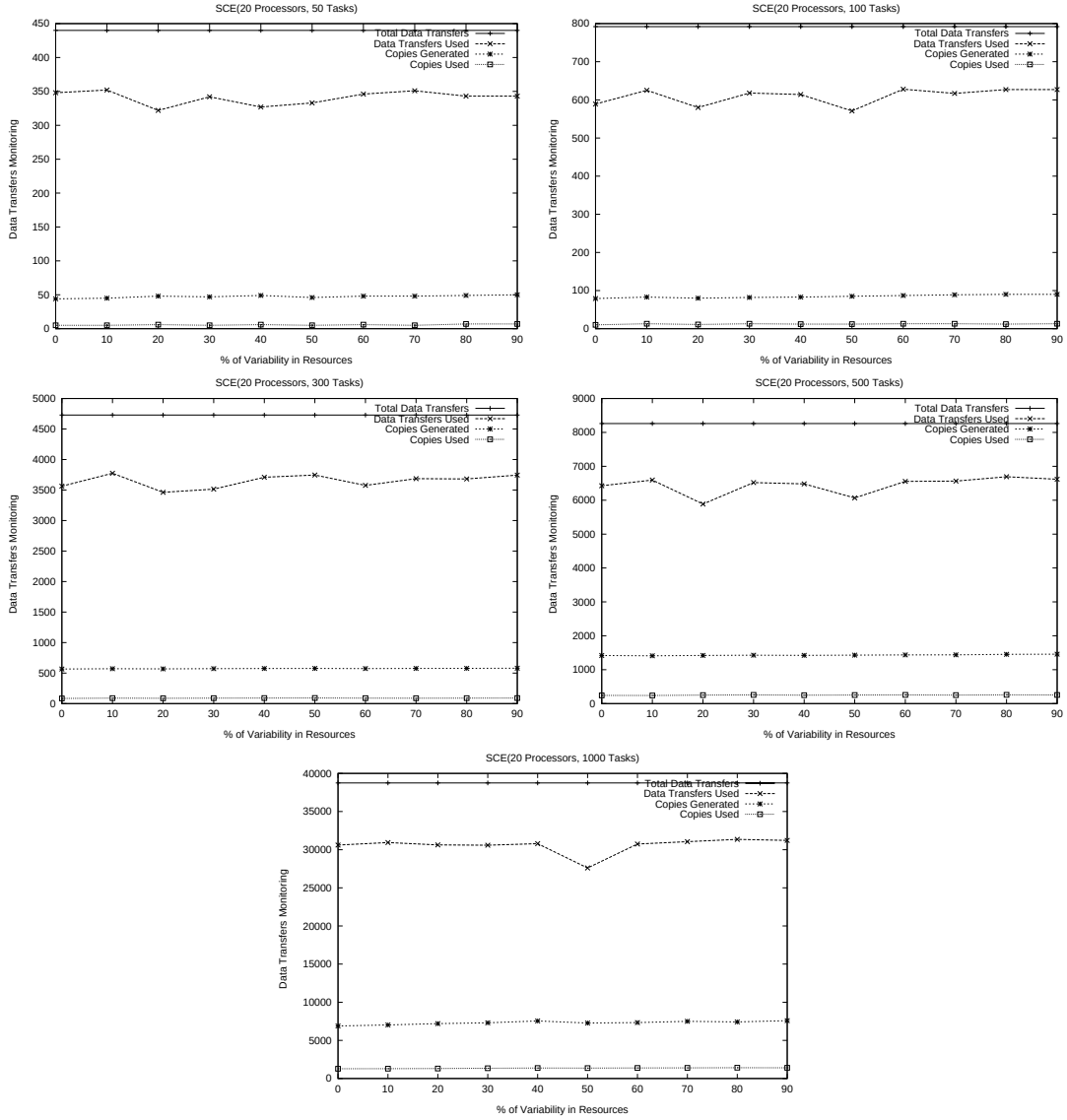


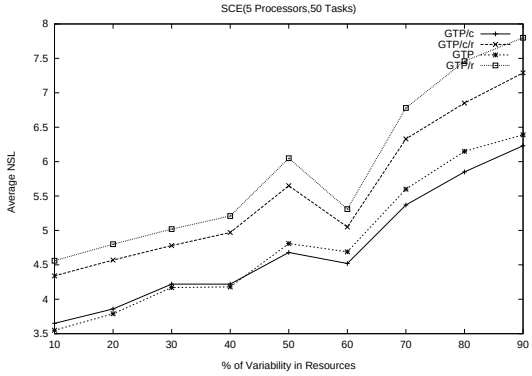
Figure 5.14: Average Data Transfers Monitoring

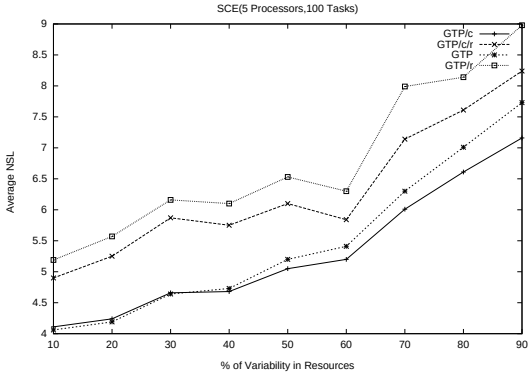
a direct input, this produces a significant improvement in the performance of the application. This can be seen in Figure 5.10, where we can see that GTP/c outperforms $HEFT$,GTP and DLS/sr in most cases. Exceptions are limited to the use of DAGs with few tasks (mainly 50 and 100 tasks) and low variability in resources. GTP/c has a better performance particularly when the application becomes larger and complex. 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. For instance,

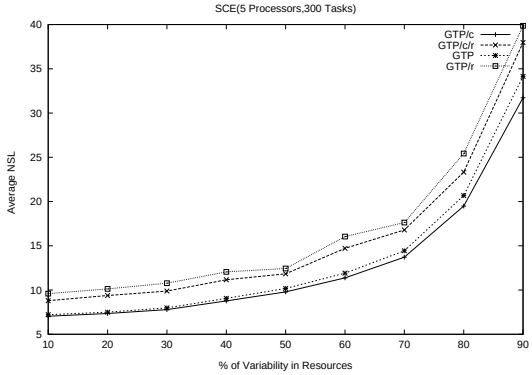
in $SCE(10,1000,30)$, the average NSL for GTP/c outperforms HEFT by up to 16%, GTP by up to 6% and DLS by up to 9%. For this scenario, we have an average of 38749 expected data transfers, 28748 of the data transfers were used to transmit data between a pair of tasks mapped on different processors, 5757 copies were generated and 940 of the copies were used as a direct input, representing 16% of the copies generated. Furthermore, we observe that in some cases, the average NLS for GTP/c is better than GTP, however the number of migrated tasks tends to be higher for GTP/c than for GTP. We believe that this is because at some point of the execution, where a particularly high number of copies has been generated, the cost of migrating a task is cheaper for GTP/c tending to increase the number of migrated tasks but not necessarily the overhead cost. This can be seen in $SCE(10,300,20)$ where the average NSL for GTP/c is better that GTP by up to 3%. However, the number of migrated tasks for GTP/c is 4% higher than GTP, but the overhead cost is 3% less than GTP.
In general terms, the cyclic use of a mapping method can generate reusable copies which can be used as direct input for some succeeding tasks. The reusable copies can reduce the impact of the overhead cost on makespan by avoiding unnecessary data transfer between tasks, and exploiting more effectively the network links. Obviously, the benefit of reusable copies can not be exploited in static schedules.

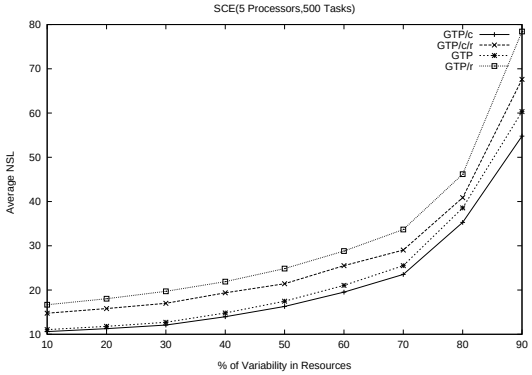
5.6 Impact of the frequency of the Rescheduling Points in the Makespan

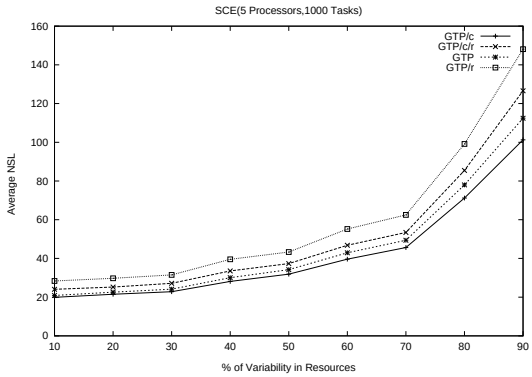
Our reactive mapping methods GTP and GTP/c, address the dynamic nature of SHCS by allowing rescheduling of an executing application in response to significant variations in resource characteristics. As we described in Section 4.2, to perform our experiments, we set a fixed-period rescheduling cycle at 10% of the value of the initial makespan, for the whole spectrum of bounds for each scenario. In this section we intend to explore the impact on makespan when the number of rescheduling points is varied. Intuitively, many RP's will increase the overhead cost of the application, but very few RP's will allow internal and external factors to negatively impact the makespan. To achieve this, we use the reactive mapping method GTP on the third scheduling scenario, which uses DAGs with CCR=1.5 and changing bandwidth over time, with the maximum bandwidth equal to one unit of data per unit of time. We evaluate GTP considering different lengths for the rescheduling point. Thus, we consider

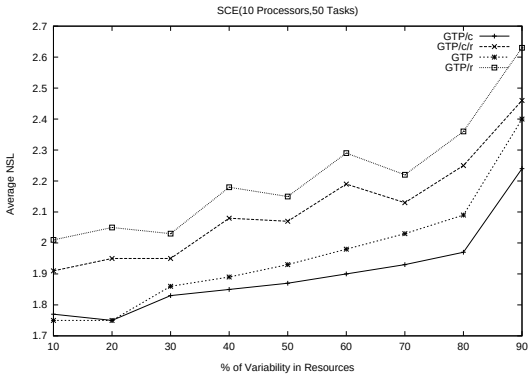

SCE(5 Processors,50 Tasks)
GTP/c
GTP/c/r
GTP
GTP/r
Average NSL
% of Variability in Resources

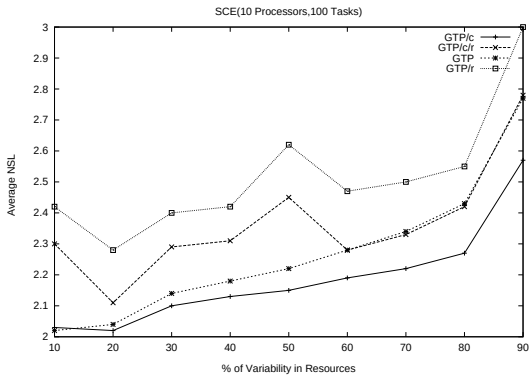

SCE(5 Processors,100 Tasks)
GTP/c
GTP/c/r
GTP
GTP/r
Average NSL
% of Variability in Resources

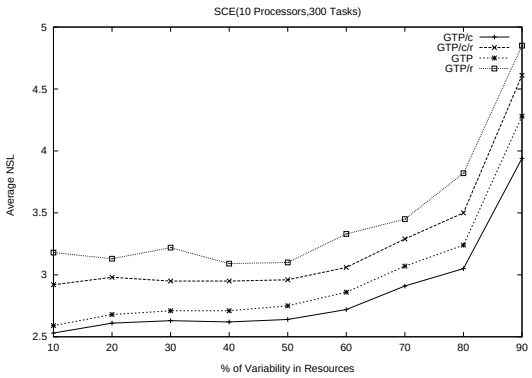

SCE(5 Processors,300 Tasks)
GTP/c
GTP/c/r
GTP
GTP/r
Average NSL
% of Variability in Resources

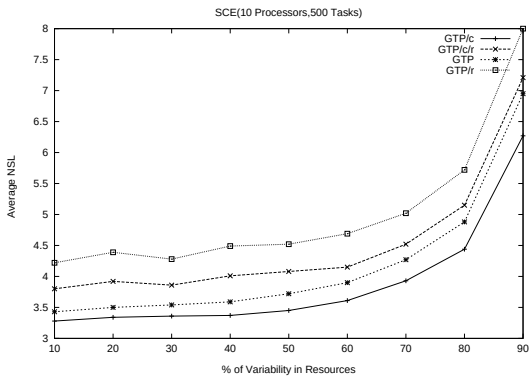

SCE(5 Processors,500 Tasks)
GTP/c
GTP/c/r
GTP
GTP/r
Average NSL
% of Variability in Resources

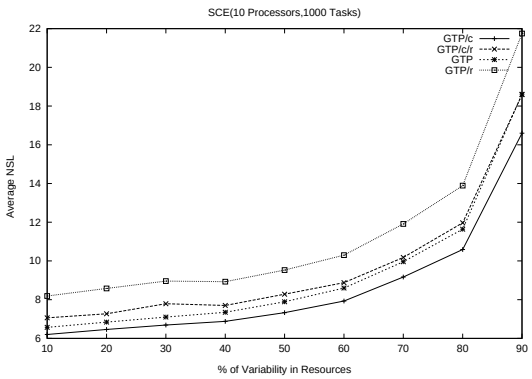

SCE(5 Processors,1000 Tasks)
GTP/c
GTP/c/r
GTP
GTP/r
Average NSL
% of Variability in Resources


SCE(10 Processors,50 Tasks)
GTP/c
GTP/c/r
GTP
GTP/r
Average NSL
% of Variability in Resources


SCE(10 Processors,100 Tasks)
GTP/c
GTP/c/r
GTP
GTP/r
Average NSL
% of Variability in Resources


SCE(10 Processors,300 Tasks)
GTP/c
GTP/c/r
GTP
GTP/r
Average NSL
% of Variability in Resources


SCE(10 Processors,500 Tasks)
GTP/c
GTP/c/r
GTP
GTP/r
Average NSL
% of Variability in Resources


SCE(10 Processors,1000 Tasks)
GTP/c
GTP/c/r
GTP
GTP/r
Average NSL
% of Variability in Resources

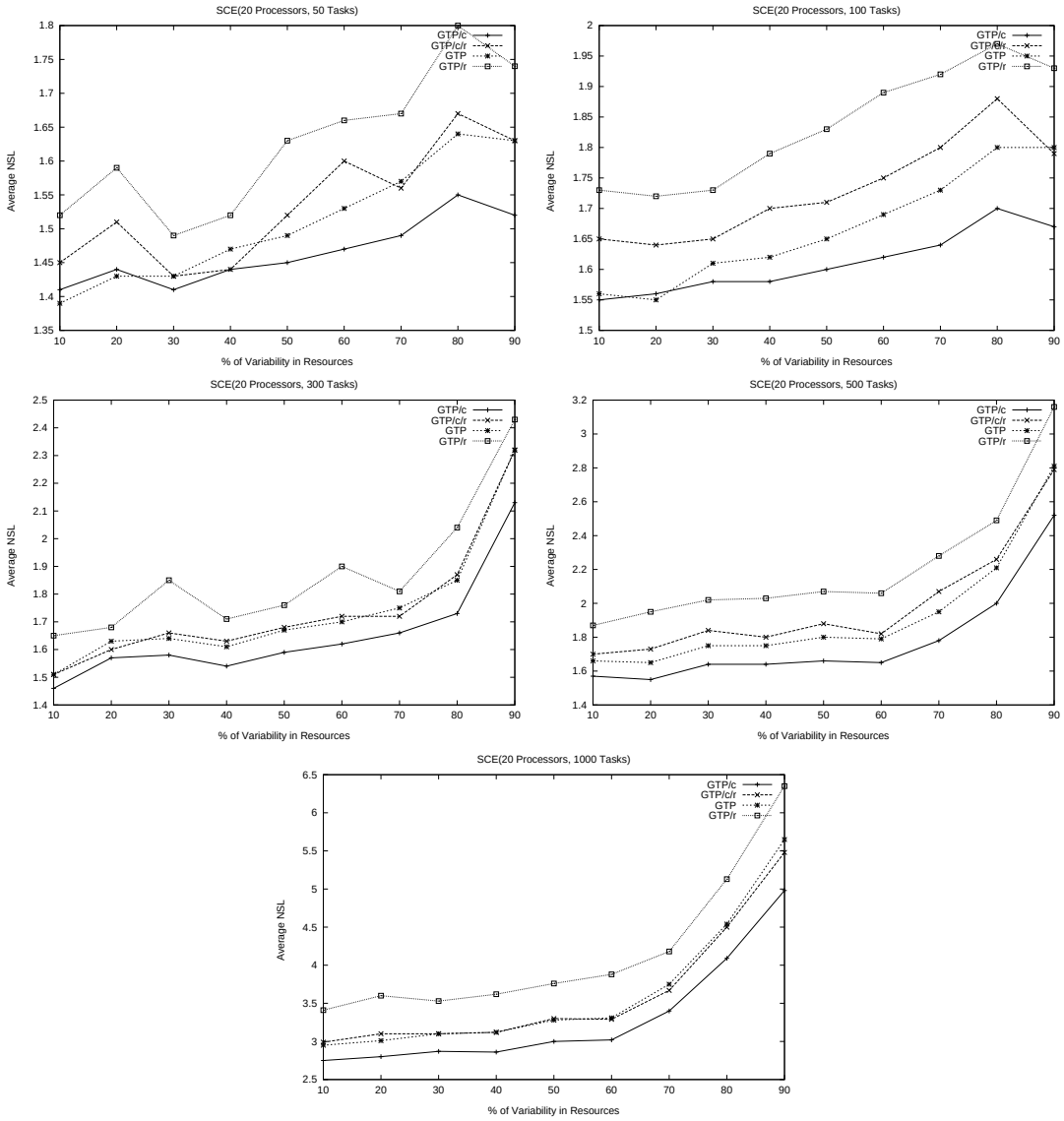
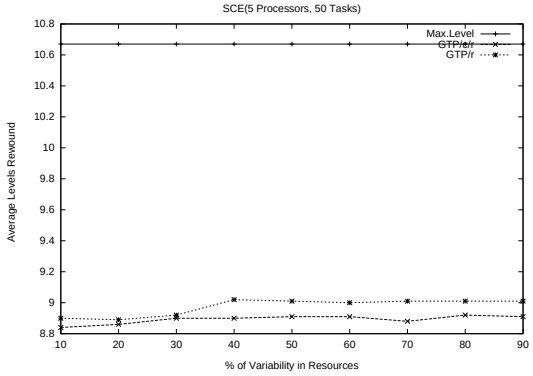
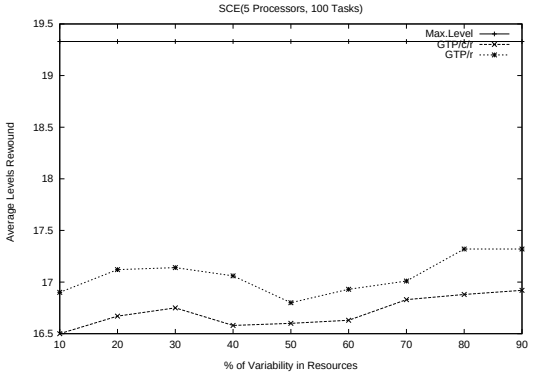


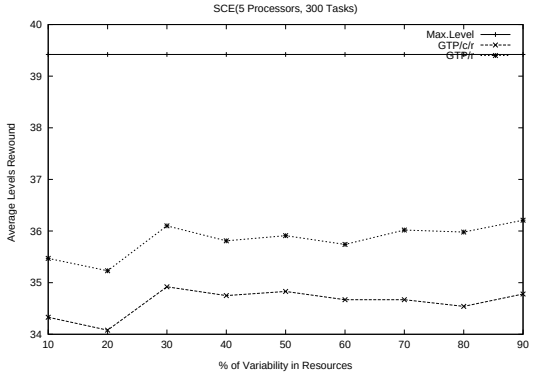
Figure 5.15: Average NSL for the GTP/r and $GTP/c/r$ System

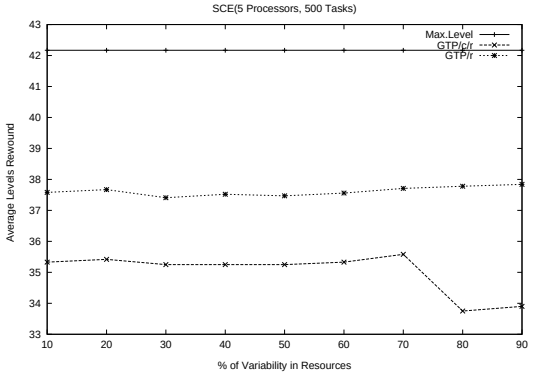
fixed-period rescheduling cycles at 1%, 3%, 10% and 30% of the value of the initial makespan.

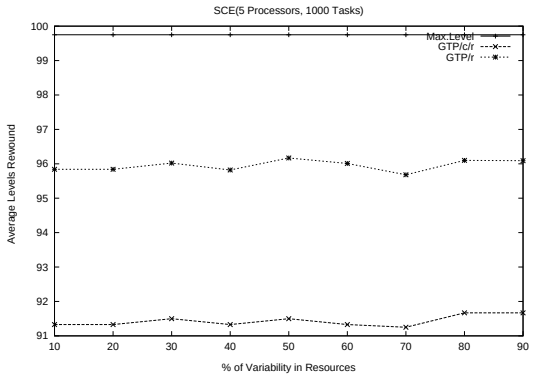
Our experimental results show that, following our strategy of setting a fixed-period rescheduling cycle for the whole spectrum of bounds for each scenario, it is not possible to distinguish a clear tendency to determine a fixed value for the rescheduling cycle. Instead, we observe that, the decision of setting the value of the length of the rescheduling point, may be linked to the variability of resources. We observe this in the three-dimensional graph in Figure 5.19, which shows for GTP, the average NSL for $SCE(x, 300, 0)$ and $SCE(x, 1000, 0)$ when the resource variability is equal to zero.

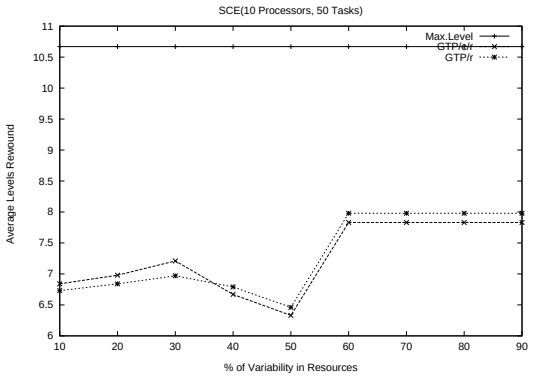

SCE(5 Processors, 50 Tasks)
Max.Level
GTP/c/r
GTP/r
Average Levels Rewound
% of Variability in Resources

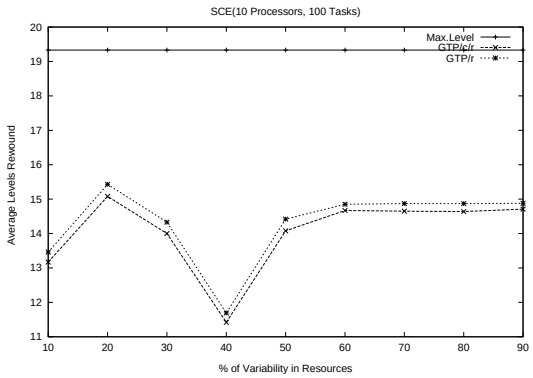

SCE(5 Processors, 100 Tasks)
Max.Level
GTP/c/r
GTP/r
Average Levels Rewound
% of Variability in Resources

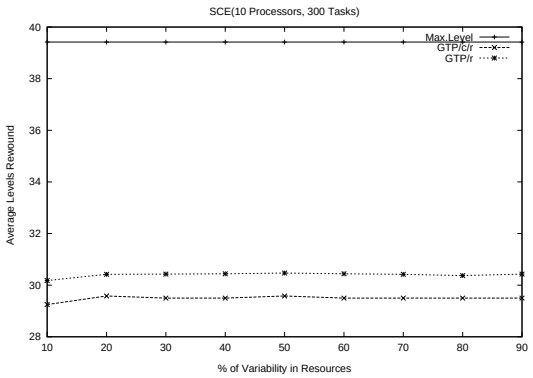

SCE(5 Processors, 300 Tasks)
Max.Level
GTP/c/r
GTP/r
Average Levels Rewound
% of Variability in Resources

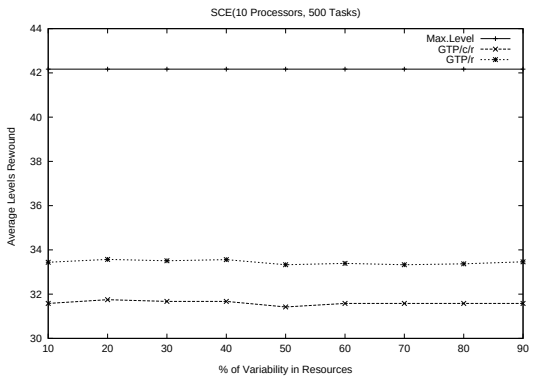

SCE(5 Processors, 500 Tasks)
Max.Level
GTP/c/r
GTP/r
Average Levels Rewound
% of Variability in Resources

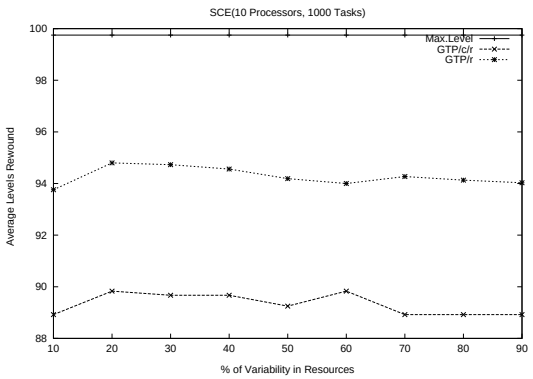

SCE(5 Processors, 1000 Tasks)
Max.Level
GTP/c/r
GTP/r
Average Levels Rewound
% of Variability in Resources


SCE(10 Processors, 50 Tasks)
Max.Level
GTP/c/r
GTP/r
Average Levels Rewound
% of Variability in Resources


SCE(10 Processors, 100 Tasks)
Max.Level
GTP/c/r
GTP/r
Average Levels Rewound
% of Variability in Resources


SCE(10 Processors, 300 Tasks)
Max.Level
GTP/c/r
GTP/r
Average Levels Rewound
% of Variability in Resources


SCE(10 Processors, 500 Tasks)
Max.Level
GTP/c/r
GTP/r
Average Levels Rewound
% of Variability in Resources


SCE(10 Processors, 1000 Tasks)
Max.Level
GTP/c/r
GTP/r
Average Levels Rewound
% of Variability in Resources

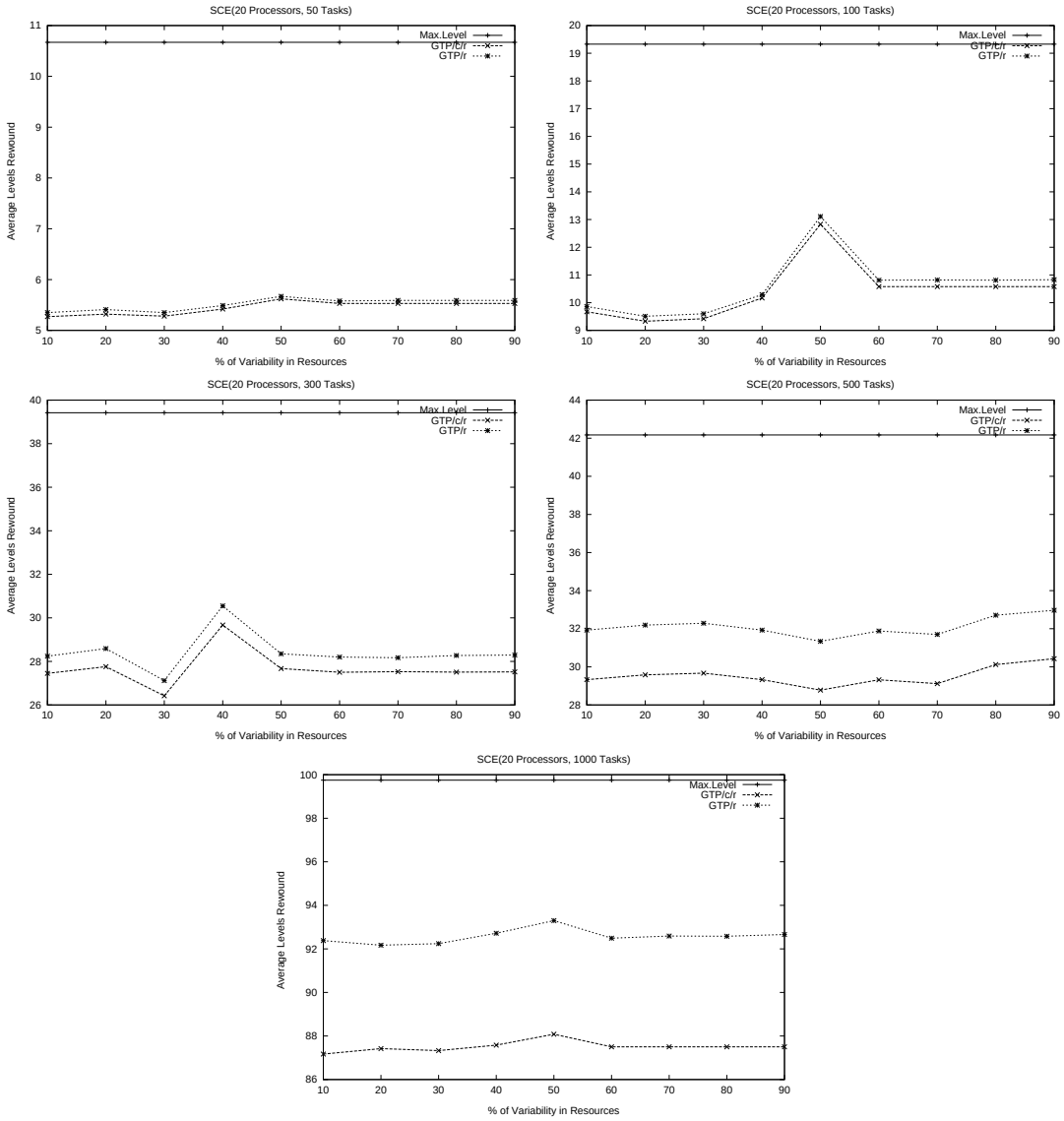
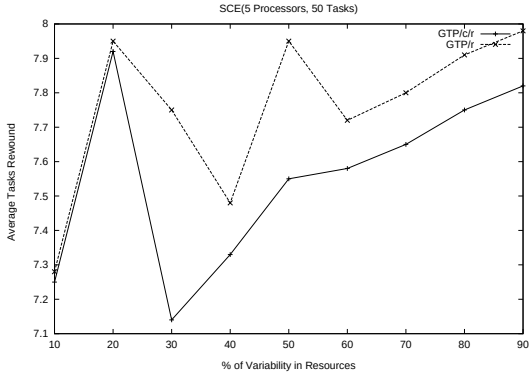
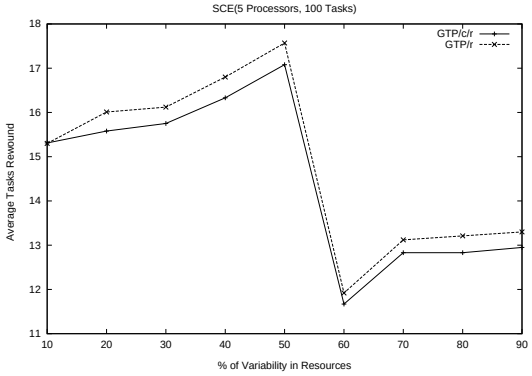
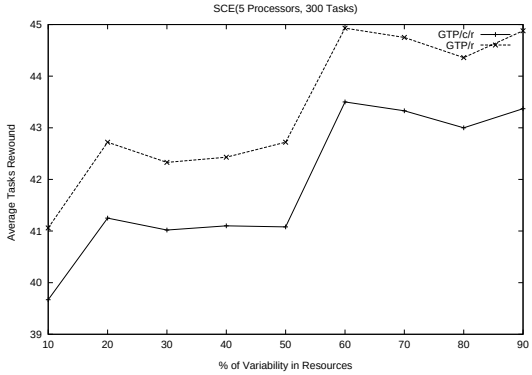


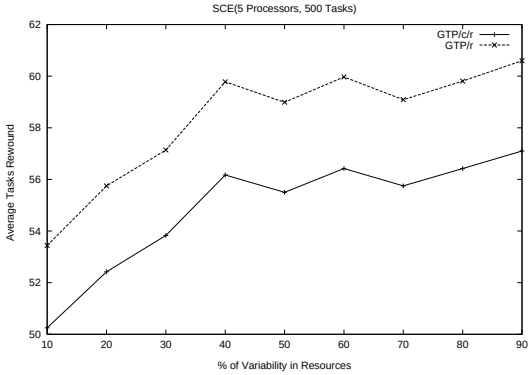
Figure 5.16: Average Levels Rewound for the GTP/r and $GTP/c/r$ System

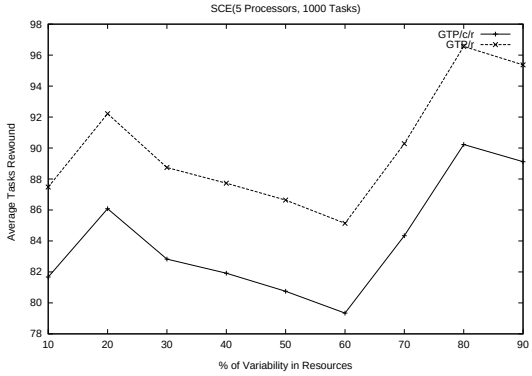
The x-axis corresponds to the different rescheduling points at 1%, 3%, 10% and 30% of the value of the initial makespan. The y-axis corresponds to the average NSL of the application and the z-axis corresponds to the number of processors (5,10 and 20 processors). Thus, we observe that as the number of rescheduling points increases (1%), the average NSL increases. For instance for $SCE(5,1000,0)$, the average NSL when the rescheduling cycle is 1% is equal to 19.91, when 3% is equal to 18.28, when 10% is equal to 18.57 and when 30% is equal to 18.39. This could mean that many short cycles may be inadequate when the fluctuations in the variability of resources are minimum. Many short cycles may increase the number of migrated tasks, therefore lengthening the makespan.

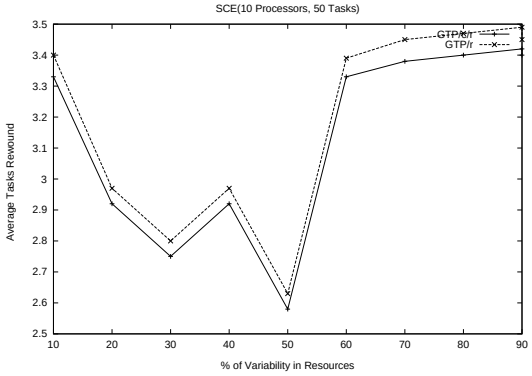

SCE(5 Processors, 50 Tasks)
GTP/c/r
GTP/r
Average Tasks Rewound
% of Variability in Resources

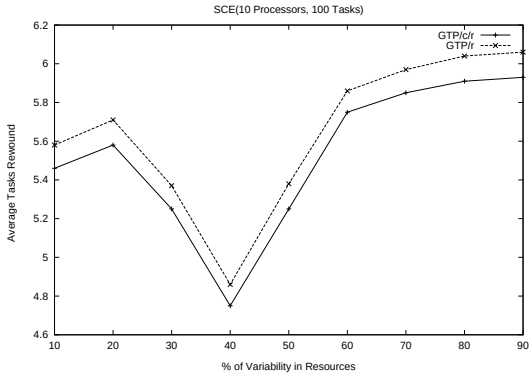

SCE(5 Processors, 100 Tasks)
GTP/c/r
GTP/r
Average Tasks Rewound
% of Variability in Resources

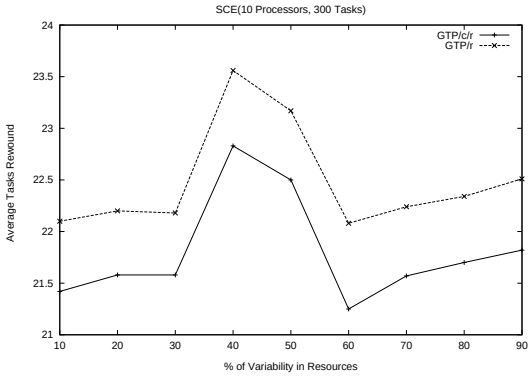

SCE(5 Processors, 300 Tasks)
GTP/c/r
GTP/r
Average Tasks Rewound
% of Variability in Resources

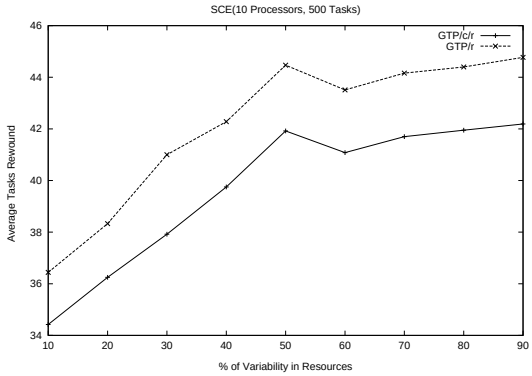

SCE(5 Processors, 500 Tasks)
GTP/c/r
GTP/r
Average Tasks Rewound
% of Variability in Resources

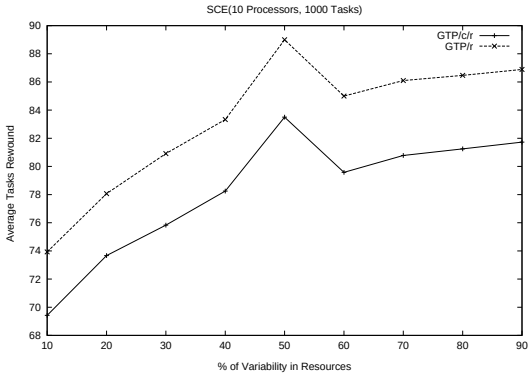

SCE(5 Processors, 1000 Tasks)
GTP/c/r
GTP/r
Average Tasks Rewound
% of Variability in Resources


SCE(10 Processors, 50 Tasks)
GTP/c/r
GTP/r
Average Tasks Rewound
% of Variability in Resources


SCE(10 Processors, 100 Tasks)
GTP/c/r
GTP/r
Average Tasks Rewound
% of Variability in Resources


SCE(10 Processors, 300 Tasks)
GTP/c/r
GTP/r
Average Tasks Rewound
% of Variability in Resources


SCE(10 Processors, 500 Tasks)
GTP/c/r
GTP/r
Average Tasks Rewound
% of Variability in Resources


SCE(10 Processors, 1000 Tasks)
GTP/c/r
GTP/r
Average Tasks Rewound
% of Variability in Resources

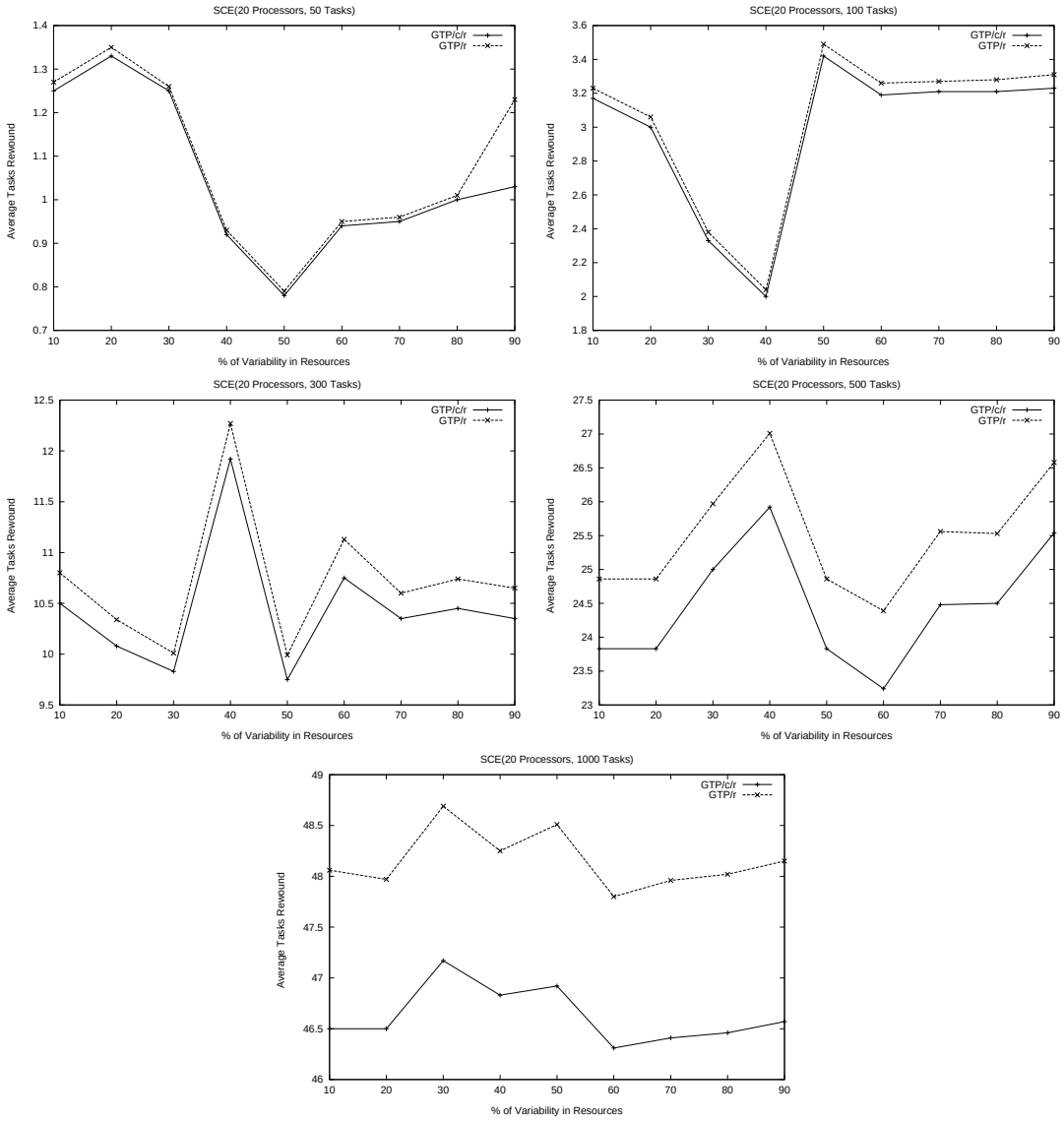
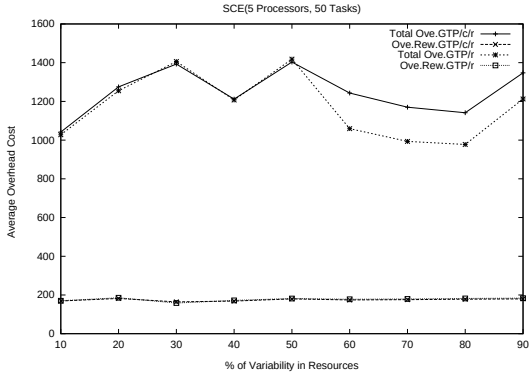
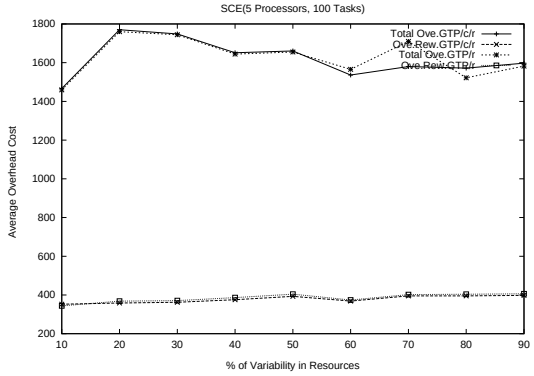
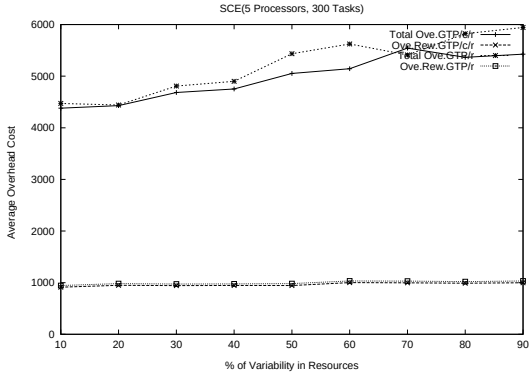


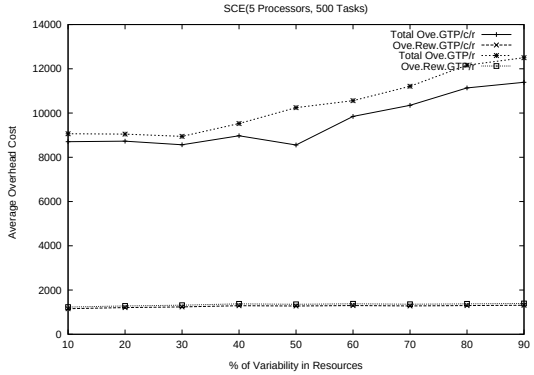
Figure 5.17: Average Tasks Rewound for the GTP/r and $GTP/c/r$

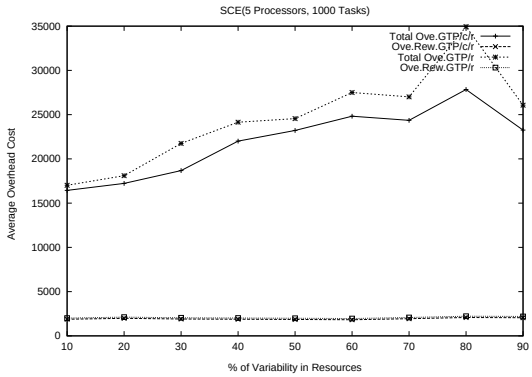
On the other hand, in the three-dimensional graph in Figure 5.20, it is shown for *GTP*, the average NSL for $SCE(5,300,z)$ and $SCE(20,1000,z)$ when the resource variability is high ($70 \leq z \leq 90$). The x-axis corresponds to the different rescheduling points at 1%, 3%, 10% and 30% of the value of the makespan. The y-axis corresponds to the average NSL of the application and the z-axis corresponds to the resource variability ranging from 70% to 90%. In this scenario, we observe that as the number of rescheduling points increases (1%), the average NSL tends to decrease compared with the other rescheduling points. For instance for $SCE(20,100,70)$, the average NSL when the rescheduling cycle is 1% is equal to 3.07, when 3% is equal to 4.55, when 10% is equal to 3.03 and when 30% is equal to 3.30. It could mean that many short cy-

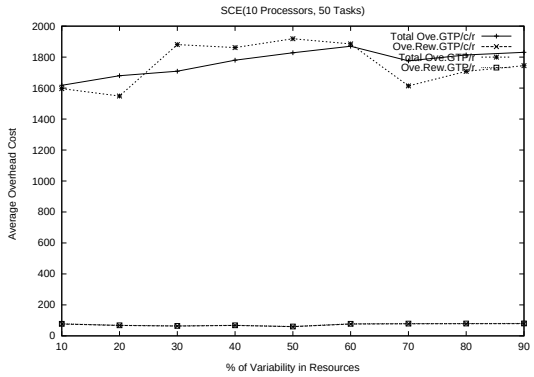

SCE(5 Processors, 50 Tasks)
Total Ove.GTP/c/r
Ove.Rew.GTP/c/r
Total Ove.GTP/r
Ove.Rew.GTP/r
Average Overhead Cost
% of Variability in Resources

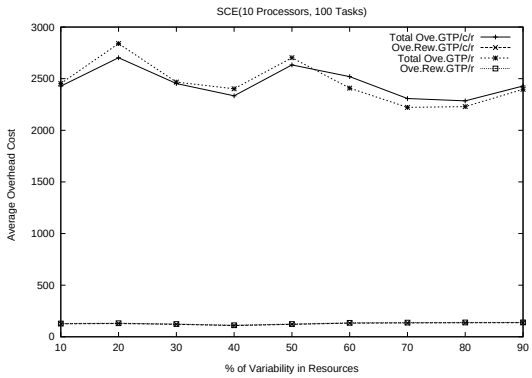

SCE(5 Processors, 100 Tasks)
Total Ove.GTP/c/r
Ove.Rew.GTP/c/r
Total Ove.GTP/r
Ove.Rew.GTP/r
Average Overhead Cost
% of Variability in Resources

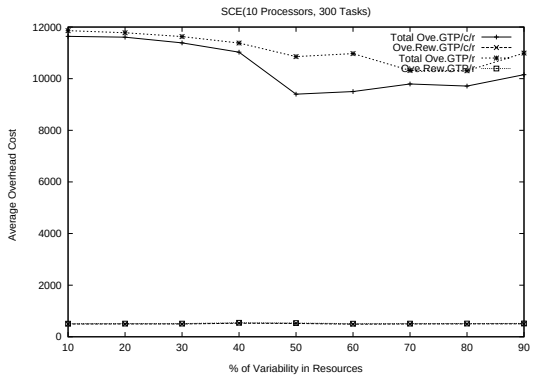

SCE(5 Processors, 300 Tasks)
Total Ove.GTP/c/r
Ove.Rew.GTP/c/r
Total Ove.GTP/r
Ove.Rew.GTP/r
Average Overhead Cost
% of Variability in Resources

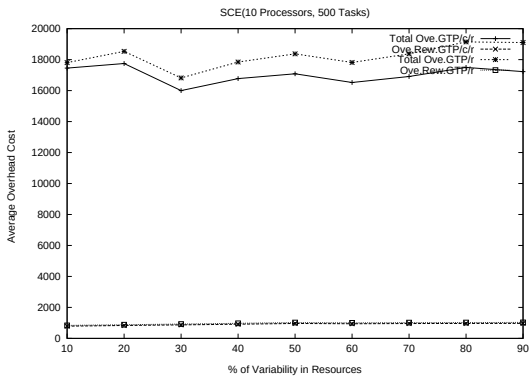

SCE(5 Processors, 500 Tasks)
Total Ove.GTP/c/r
Ove.Rew.GTP/c/r
Total Ove.GTP/r
Ove.Rew.GTP/r
Average Overhead Cost
% of Variability in Resources

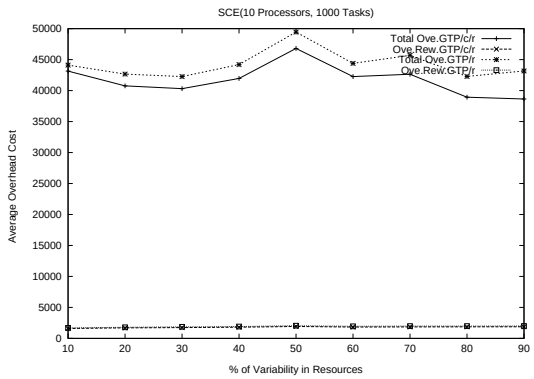

SCE(5 Processors, 1000 Tasks)
Total Ove.GTP/c/r
Ove.Rew.GTP/c/r
Total Ove.GTP/r
Ove.Rew.GTP/r
Average Overhead Cost
% of Variability in Resources


SCE(10 Processors, 50 Tasks)
Total Ove.GTP/c/r
Ove.Rew.GTP/c/r
Total Ove.GTP/r
Ove.Rew.GTP/r
Average Overhead Cost
% of Variability in Resources


SCE(10 Processors, 100 Tasks)
Total Ove.GTP/c/r
Ove.Rew.GTP/c/r
Total Ove.GTP/r
Ove.Rew.GTP/r
Average Overhead Cost
% of Variability in Resources


SCE(10 Processors, 300 Tasks)
Total Ove.GTP/c/r
Ove.Rew.GTP/c/r
Total Ove.GTP/r
Ove.Rew.GTP/r
Average Overhead Cost
% of Variability in Resources


SCE(10 Processors, 500 Tasks)
Total Ove.GTP/c/r
Ove.Rew.GTP/c/r
Total Ove.GTP/r
Ove.Rew.GTP/r
Average Overhead Cost
% of Variability in Resources


SCE(10 Processors, 1000 Tasks)
Total Ove.GTP/c/r
Ove.Rew.GTP/c/r
Total Ove.GTP/r
Ove.Rew.GTP/r
Average Overhead Cost
% of Variability in Resources

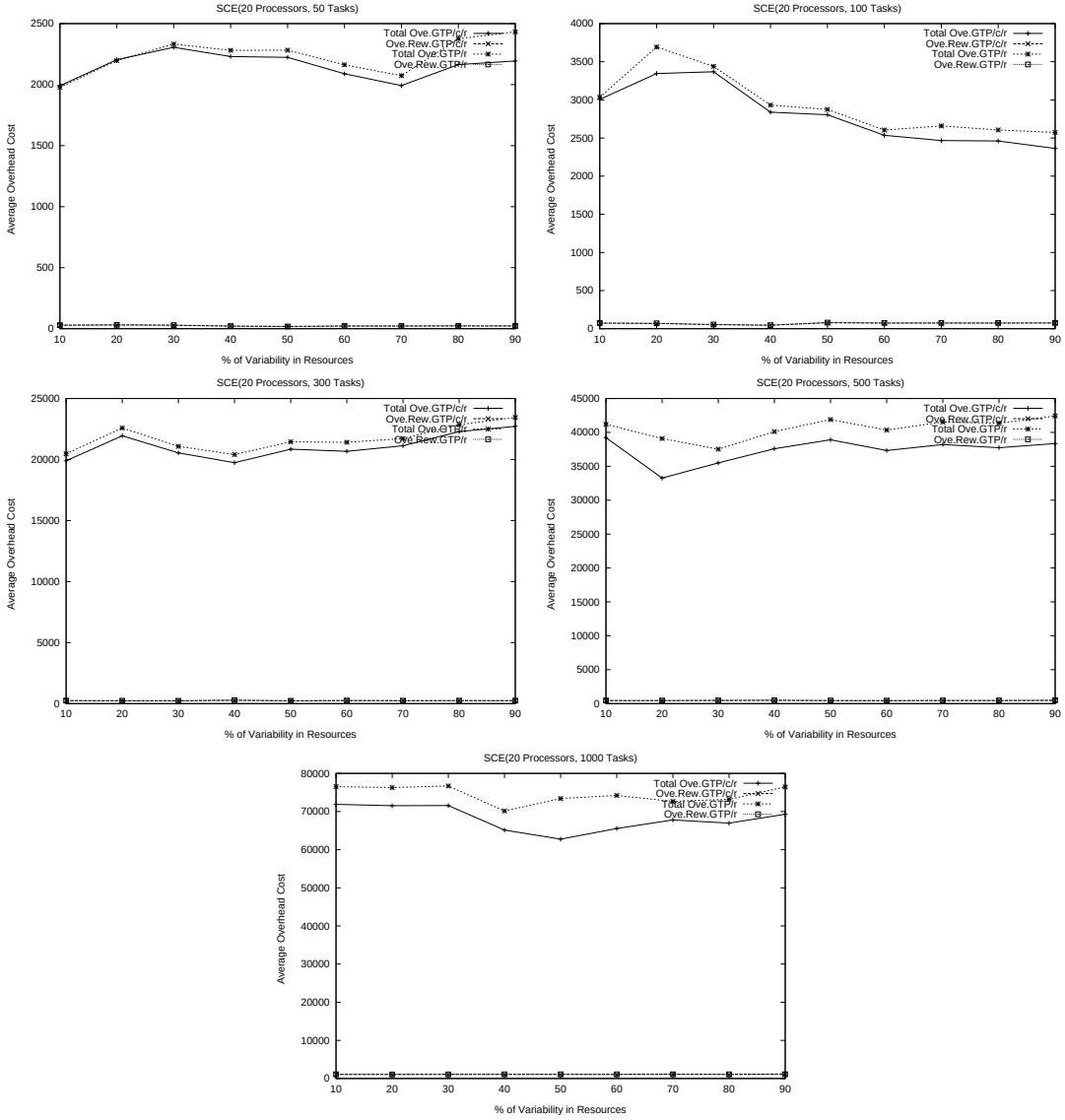


Figure 5.18: Average Overhead Cost for for the GTP/r and $GTP/c/r$

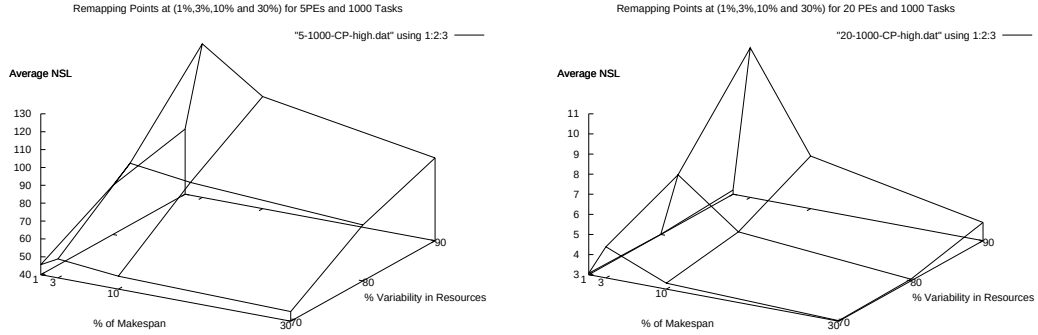


Figure 5.19: Average NSL for Scenarios with minimum variability

cles may be required to react more efficiently to resource variability. Few long cycles may not properly react to dynamic changes.

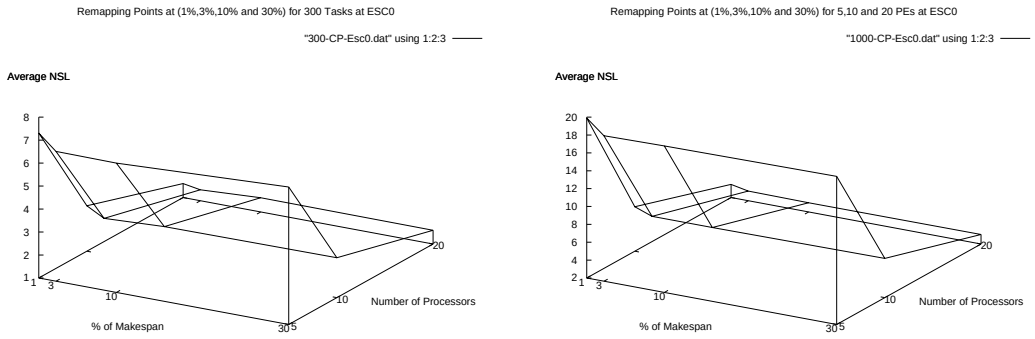


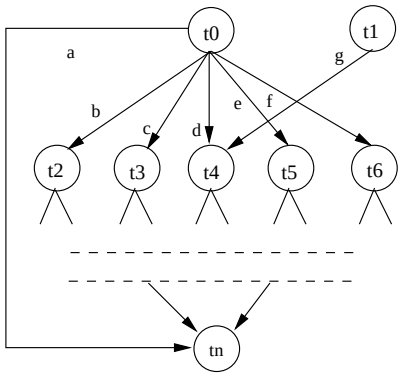
Figure 5.20: Average NSL for Scenarios with high variability

In general terms, the setting of rescheduling points is an important element of reactive mapping methods based on the cyclic use of a mapping method. Considering our experimental results, we believe that the strategy used in our experiments to evaluate the performance of the reactive mapping methods, which consider a fixed-period rescheduling cycle for the whole spectrum of bounds for each scenario, may not be adequate for extreme cases. New efforts are required to optimize the frequency of the rescheduling cycles. We believe that the observed variability of resources can be a parameter to determine the frequency of the rescheduling cycle.

5.7 Rethinking DAG Applications for SHCS

In previous literature, the relationship between the DAG application (defined by the owner of the DAG) and the scheduling mechanism (defined by the owner of the method) is not fully explored. Most mapping methods focus on scheduling strategies which use the shape and static information of the DAG. They do not consider the mechanism through which communication of task results is actually achieved. We have found that ignoring this issue may negatively affect the performance of the application. To explain this, we will use a hypothetical case shown in the Figure 5.21. Figure 5.21(a) describes a portion of some particular DAG application and Figure 5.21(b) shows the schedule generated by the HEFT algorithm for the tasks shown in the partial DAG. In such schedules we assume that the tasks are ordered according to the task ranks. In keeping with the consistency of our formal definitions, we will use $EST(v_i, p_j)$

and $EFT(v_i, p_j)$ to denote the estimated earliest start time and earliest finish time of task v_i on processor p_j respectively. In the same manner, we will use $RST(v_i, p_j)$ and $RFT(v_i, p_j)$ to denote the real start time and real finish time of task v_i on processor p_j respectively. In terms of the communication model among tasks, we observe two main models to allow the transfer of data among tasks, the *PUSH* and *PULL* models.



a) DAG application

Tasks	Processor
t0	P1
t1	P1
t4	P2
t3	P3
t2	P2
t5	P3
t6	P2
. . . .	. . . .
tn	P3

b) Schedule

Figure 5.21: DAG application

1. The PUSH model, in which as soon as a task finishes execution, it pushes the data result to its successors to be executed. For instance, by using the HEFT schedule from Figure 5.21, the computation of the estimated start time for task $t4$ mapped on p_2 is given by, $EST(t_4, p_2) = max(EFT(t_0, p_1) + C(t_0, p_1, t_4, p_2), EFT(t_1, p_1) + C(t_1, p_1, t_4, p_2))$. However, the real start time for $t4$ on p_2 when using the PUSH model is, $RST(t_4, p_2) = max(RFT(t_0, p_1) + C(t_0, p_1, t_4, p_2) + C(t_0, p_1, t_2, p_2) + C(t_0, p_1, t_6, p_2), RFT(t_1, p_1) + C(t_1, p_1, t_4, p_2))$. This means that in order to be executed, t_4 must wait until t_1 pushes the data to those successors mapped in the same processor and t_2 pushes the remaining data needed by t_4 for execution.

2. The PULL model, in which as soon as a task is mapped on a particular processor, it requests to pull the data needed from its predecessors. By using the same example, now the computation of the estimated start time for task $t4$ is given by , $EST(t_4, p_2) = max(EFT(t_0, p_1) + C(t_0, p_1, t_4, p_2), EFT(t_1, p_1) + C(t_1, p_1, t_4, p_2))$, and the real start time using the PULL model is, $RST(t4, p2) = max(RFT(t_0, p_1) + C(t_0, p_1, t_4, p_2), RFT(t_1, p_1) + C(t_1, p_1, t_4, p_2)$. This means that for this model, $t4$ will wait less time to be executed as it receives its inputs just after its predecessors $t0$ and $t1$ finish execution.

Thus, the communication model among tasks is another external factor which may impact the makespan of the application. We believe that the design of DAG applications must be reconsidered when they are executed on dynamic environments such as SHCS. The notion behind this statement is that we observed that for our DAGs , the nature of PULL models increases the performance of the application. We enumerate below some of the ways in which PULL models may enhance the performance of the application,

1. Data storage time refers to the waiting time that the data remains stored at some particular processor before being used by some task. Data with long waiting times on resources may become unavailable in case of processor failure or may affect the predictions of tasks, increasing the number of migrated tasks. We observe that the data storage time tends to be less for PULL models than PUSH models. This can be observed in Figure 5.21 where, for the PUSH model, the data transfer for the edge (t_0, t_n) is sent when t_0 finishes execution, even if t_n executes much later.

2. Pulling data may allow data transfers to arrive at the source relatively close in time such that the data storage time will tend to be short, this is important because it may help to decrease the number of migrated tasks due to the dynamic nature of resources. When pushing data, the arrival of data at the source will be more dispersed in time, as a consequence the data storage time could be longer and the number of retransmissions may increase.

3. Long data storage time will require a major amount of physical storage. We notice that neither GTP nor GTP/c make any attempt to optimize the physical data storage. Complementary work can be found in [(Ramakrishnan et al., 2007)], which considers physical data-storage constraints when scheduling data intensive applications.

In this context, we believe that for some DAGs, communication models based on the PULL model are more suitable for SHCS than the PUSH model. We observe that the PULL model requires data to be stored for less time than the PUSH model. To support our statement, we designed some experiments to understand the impact of using the PUSH or PULL model with both dynamic and static mapping approaches. We note that the experiments using the PULL model are the same experiments used

to evaluate the performance of GTP and GTP/c. We present the experimental results below.

5.7.1 Evaluating the PUSH and PULL Models for Static Mapping Methods

In this section we evaluate the PUSH and PULL models for static mapping approaches. To conduct our experiments, we will use the characteristics of the second scenario, for which CCR = 0.5 and changing bandwidth over time, with the maximum bandwidth equal to one unit of data per unit of time. We use the HEFT static mapping method, evaluated in our scenarios by using a version with the PUSH model and another version with the PULL model. In Figure 5.22 we present the results for 10 processors. The performance of the PULL model in most cases tends to be better than the PUSH model, particularly for DAGs with 500 and 1000 tasks. For instance, in $SCE(10,500,40)$, HEFT with the PULL model is up to 4.8% better than HEFT with the PUSH model and in $SCE(10,1000,30)$ the performance increases up to 5.5%.

5.7.2 Evaluating the PUSH and PULL Model for Reactive Mapping Methods

In this section we evaluate the PUSH and PULL models for the reactive scheduling mechanisms GTP, GTP/c and DLS/sr and present the results of the evaluation using our scenarios with 10 processors. We observe that the impact of using PUSH or PULL models in reactive scheduling approaches tends to be more significant.

The DLS/sr approach is the most affected by the mapping methods evaluated. For instance we observe in Figure 5.23 that for $SCE(10,1000,10)$, the average NSL when using the PUSH model is up to 8 times higher than when using the PULL model. We believe that the nature of the PUSH model combined with the internal and external factors described in Section 5.3 will tend to negatively affect the predictions of the spare time of tasks. This can be observed in Figure 5.24, where we observe that for the same $SCE(10,1000,10)$ the number of remappings increases up to 3 times, increasing by up to 5 times the number of migrated tasks (see Figure 5.25) and finally increasing the overhead cost up to 7 times (see Figure 5.26). These values tend to gradually increase as the variability increases.

The GTP model with PULL outperforms GTP with PUSH. For the same scenario

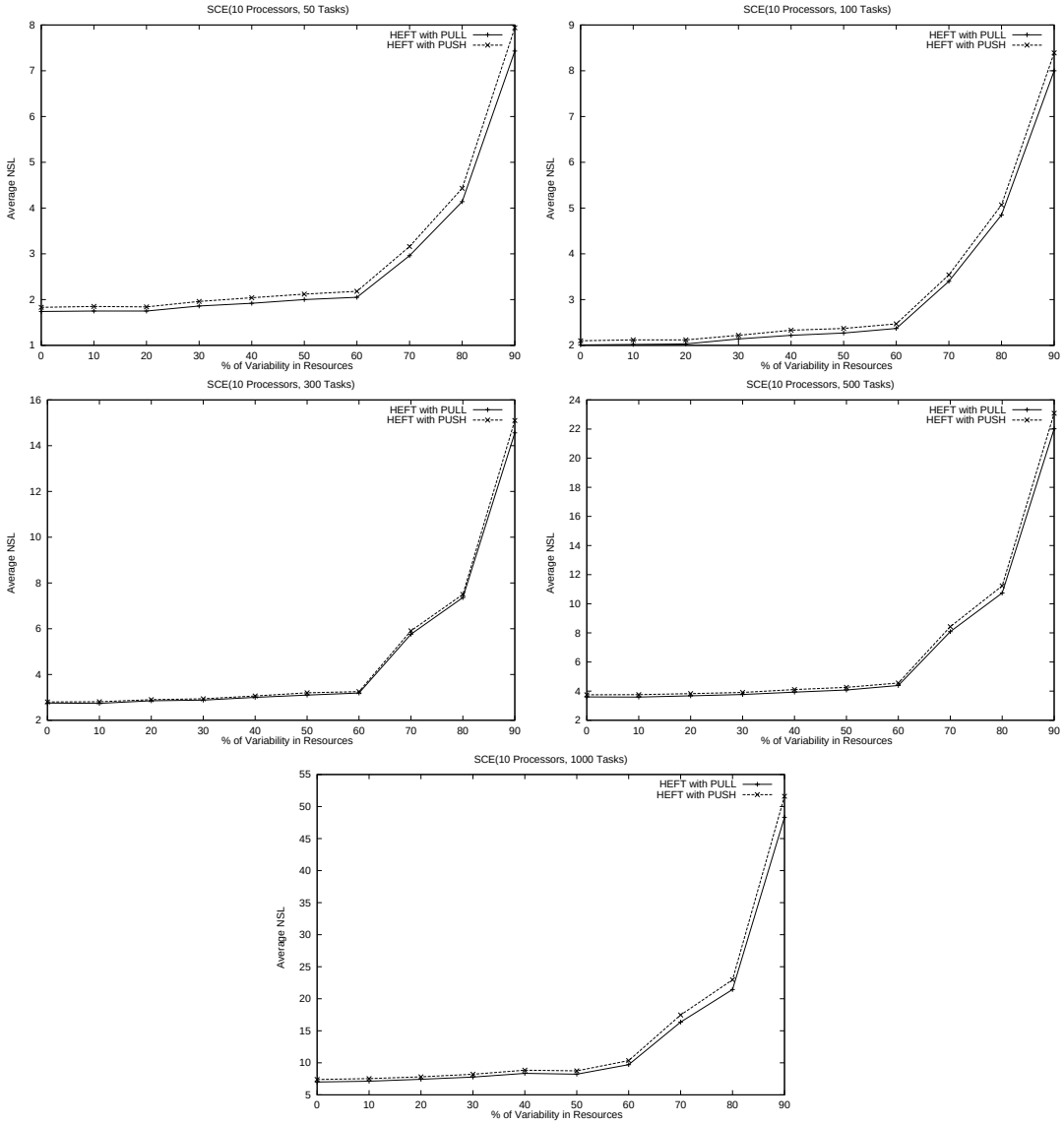


Figure 5.22: Comparison of HEFT with PUSH and PULL Models

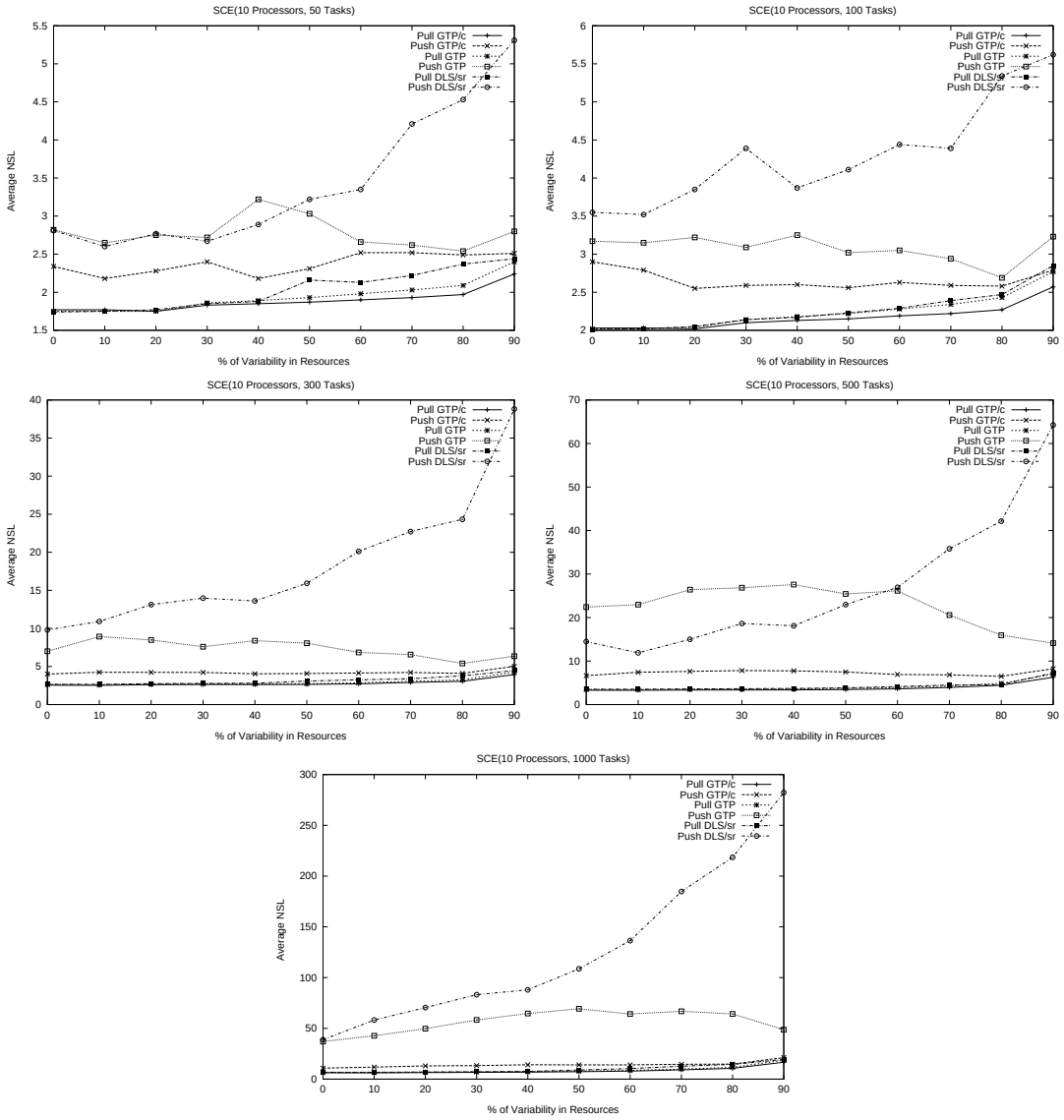


Figure 5.23: Comparison of average NSL for reactive methods with PUSH and PULL models

$SCE(10, 1000, 10)$, we observe in Figure 5.23 that the average NSL is up to 6.5 times higher for GTP with PUSH. In Figure 5.24 it is observed that GTP with PULL requires up to 6 times less remappings than GTP with PUSH. Consequently, the number of migrating tasks decreases up to 7 times and the overhead cost is up to 9 times shorter than GTP with PUSH.

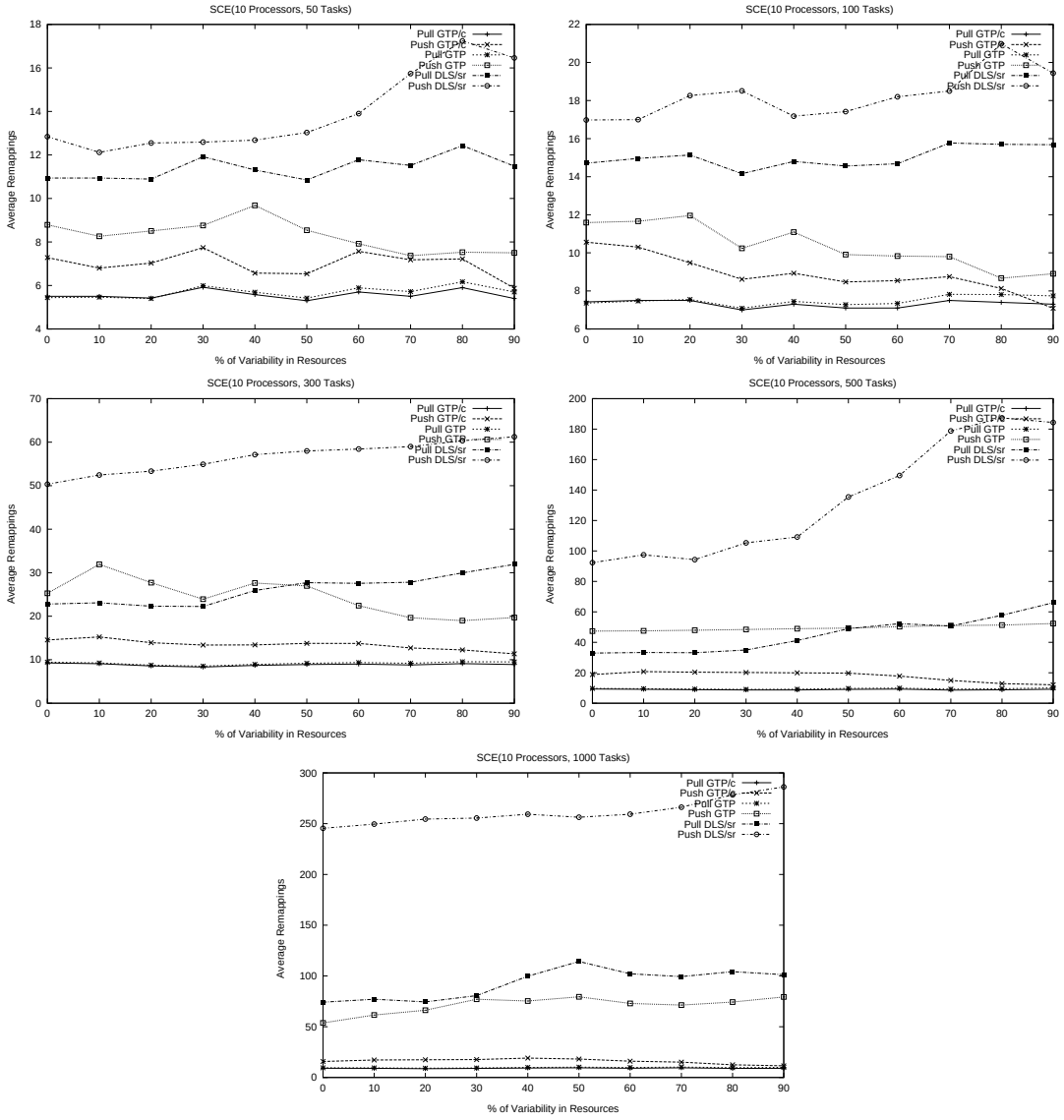


Figure 5.24: Comparison of average remappings for reactive methods with PUSH and PULL Models

GTP/c is the dynamic mapping method least affected in our evaluation. The copying facilities which allows GTP/c to reuse copies of the results of some tasks help to decrease the impact of the inaccurate predictions caused by the PUSH model. Following with the same scenario $SCE(10, 1000, 10)$ the NSL for GTP/c with PUSH is twice

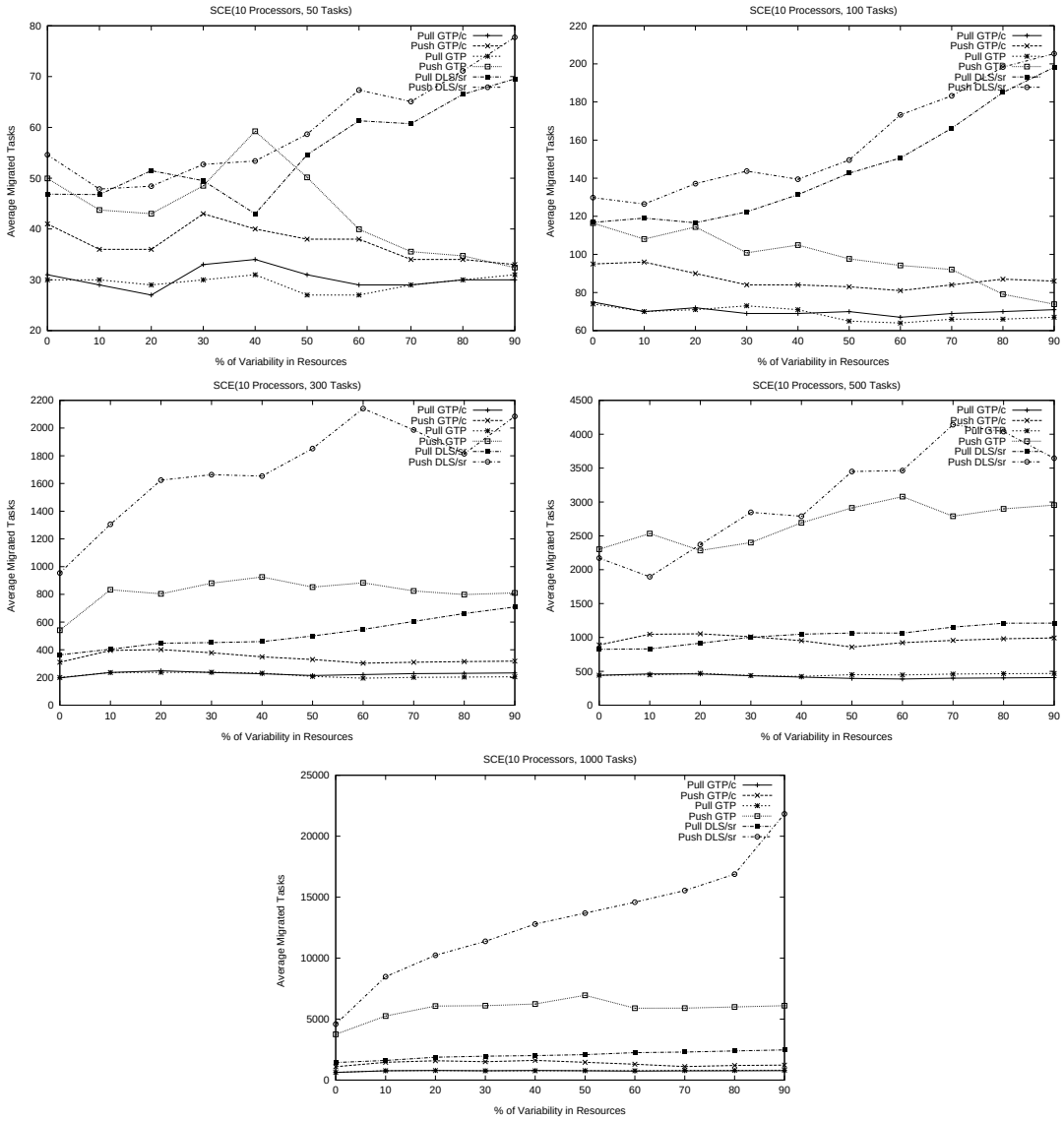


Figure 5.25: Comparison of average migrated tasks for reactive methods with PUSH and PULL Models

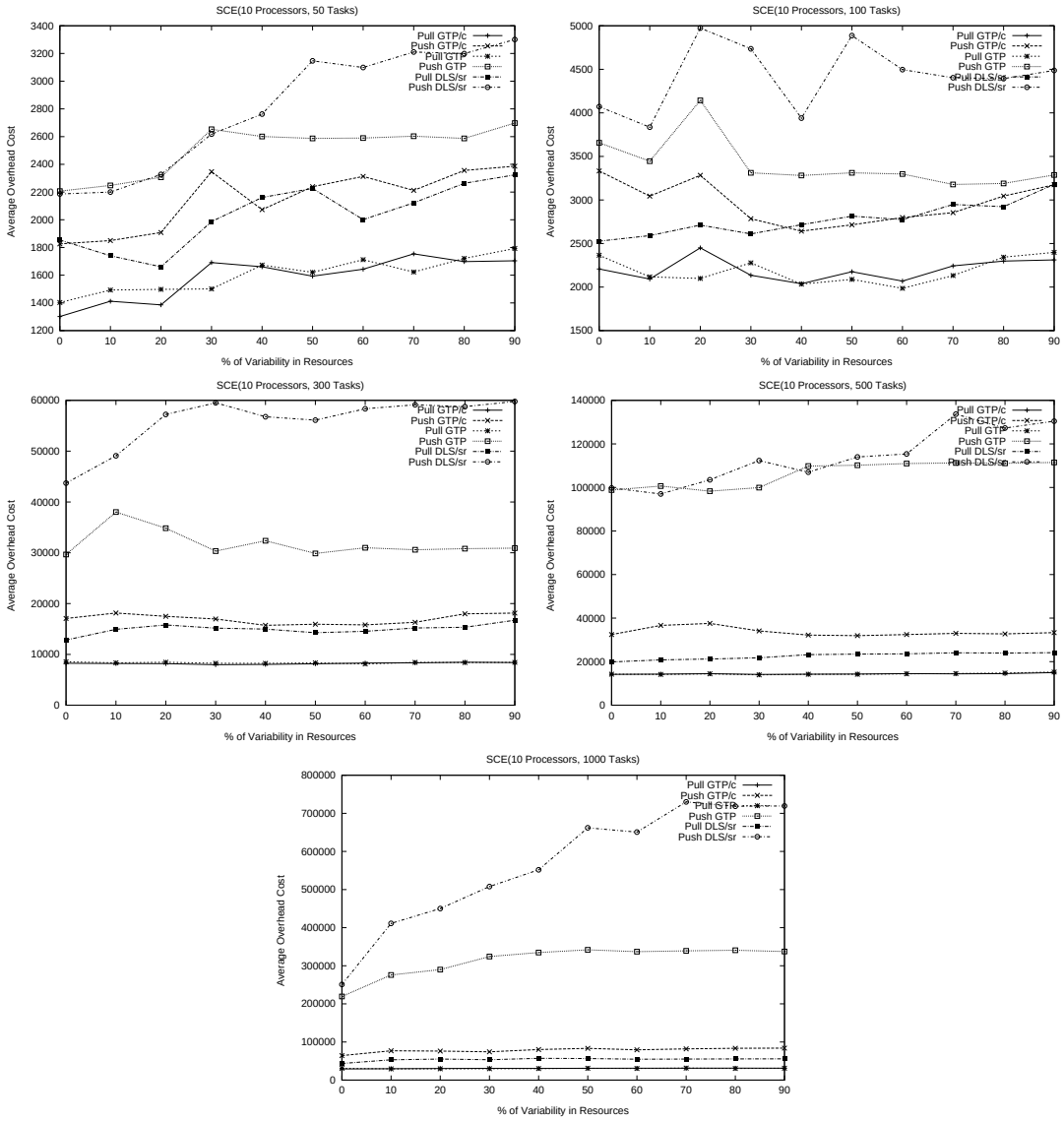


Figure 5.26: Comparison of average overhead cost for reactive methods with PUSH and PULL models

that of GTP/c with PULL. The number of remappings required when using the PUSH models is up to twice that for the PULL model. In the same manner the number of migrated tasks is doubled and the overhead cost 2.5 times higher for the PUSH model. In general terms, the problem of the communication model among tasks is highly dependent upon the shape and the size of the DAG applications. We showed that for the DAGs used in this experiments, the PULL models presented better performance than the PUSH models. The evaluation of our reactive mechanisms GTP, GTP/c and DLS/sr was carried out considering the PULL model. We note that new heuristic techniques are needed to exploit the nature of PULL models, for instance scheduling techniques based on the backward scheduling approach.

5.8 Reliable Task Scheduling with Rewinding and Migration

In this section we evaluate the performance of the GTP/r and $GTP/c/r$ systems which include the Rewinding Mechanism. To achieve this, we will use the characteristics of the second scenario, for which CCR = 0.5 and changing bandwidth over time, with the maximum bandwidth equal to one unit of data per unit of time. We injected randomly (but repeatably) into our scenarios failures which will set a particular processor as unavailable at the mid-point of the execution. Thus, when the failure is detected at the next rescheduling point (RP), the rewinding mechanism will be triggered to allow the application to complete despite the unavailable processor. Failures will be added to the same scenarios used to benchmark the GTP and GTP/c, so that we will be able to compare for each model the amount of extra time required by the application to finish execution when a failure occurs. This can be determined by the difference between the makespan obtained for the application with failure and the makespan obtained for application without failure, which is the same that of Section 5.4 and Section 5.5 for GTP and GTP/c respectively. It is important to remember that these makespans are for different circumstances, since GTP and GTP/c might simply fail to terminate in the presence of failure. In order to gain a better understanding in this area, we monitor a set of complementary metrics, defined in Section 3.5, which we believe are related to the performance of the application. These complementary metrics are the levels rewound (LR), the placed tasks rewound (PTR) and the rewinding overhead cost (ROO). Our experimental results show that the rewinding mechanism for the $GTP/c/r$ system

outperforms the GTP/r system in most cases. This can be observed in the graphics of Figure 5.15 where the average NSL for $GTP/c/r$ tend to be less than GTP/r for all the scenarios. This means that in the presence of a failure, an application using $GTP/c/r$ will require less extra time to finish execution than using GTP/r. To explain this, we will use the complementary metrics. We observe in Figure 5.15 that for $SCE(10,500,30)$ the average NSL for $GTP/c/r$ is up to 11% better than GTP/r. Now, from the complementary metrics, we observe in Figure 5.16 that GTP/r will need up to 5.8% more levels rewound than $GTP/c/r$, and the number of tasks to be recomputed is up to 8% more than $GTP/c/r$ (see Figure 5.17), generating 3% more rewinding overhead (see Figure 5.18). From this we learn that there exists a linear chain of events which links the number of levels rewound, the number of rewound tasks, the rewinding overhead cost and finally the real makespan of the application. 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. Thus, these tasks will not be rewound, reducing the impact of the linear chain of events on makespan.

In general terms our experimental results showed that the rewinding mechanism helps to preserve the execution of the application despite the presence of failure in particular processors. The performance of the rewinding mechanism for a particular scheduling system is highly dependent upon the details of the scheduling strategies used. We have shown that the strategy of maintaining reusable copies may help to reduce the impact of the failed processor on makespan. Obviously, mapping methods, which are not able to preserve the execution of the application in the presence of a faulty processor, will need to restart the execution of the application from the very beginning.

5.9 Summary

In this chapter we have presented the simulation results of our experiments, which included the use of DAGs with different shapes and sizes, SHCS architectures with different number of processors and a number of test scenarios involving a sequence of events, each simulating a resource change in either processor or bandwidth availability. We started by presenting the results of the evaluation of the static mapping methods $HEFT$ and DLS. Then we presented the results for the GTP, GTP/c and DLS/sr models. We noted that setting the frequency of the rescheduling points is an important element of cyclic mapping methods. Next, we showed that the consideration of using

PUSH or PULL models for data transfer in SHCS may affect the performance of the application. Finally we presented the results of GTP/r and $GTP/c/r$ which include the rewinding mechanism.

Chapter 6

Conclusions

In this chapter we present the conclusions of our research work. We start by presenting a summary of the results obtained in our experiments. Next, we suggest future work related to the rewinding mechanism as a scheduling strategy to map DAG applications onto heterogeneous and dynamic distributed computing systems. Finally, we express some final thoughts.

6.1 Summary of Results

This research work explored the problem of mapping parallel applications onto heterogeneous and dynamic distributed computing systems. 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. Thus, we presented in Section 3.2, the GTP system with the premise of addressing the dynamic nature of SHCS by allowing rescheduling and migration of tasks of an executing application in response to significant variations in resource characteristics. However, we found that our model GTP apart from reacting to the dynamic nature of SHCS, reacted to inaccurate predictions from previous schedules, mainly caused by the internal factors discussed in Section 5.3. This was shown in Section 5.4 when we evaluated GTP against the $HEFT$ algorithm in those scenarios with 0% of variability in resources. GTP proved to be competitive compared with other reactive scheduling mechanisms. This was shown in Section 5.4 when we evaluated GTP in more realistic SHCS scenarios and compared the performance of GTP against the DLS/sr approach. For reactive scheduling approaches allowing rescheduling and migration of tasks, a cost must be paid which is reflected in the overhead cost and directly related with the

number of migrated tasks. In this case, we observed that DLS/sr tends to generate a higher number of migrated tasks than GTP, which in the end will negatively impact the final makespan. This is mainly because of the combination of two factors: a) The first factor is related to the prediction of the spare time of tasks, which may be affected by the external and internal factors described previously. b) The second factor is related to the criterion to apply the selective rescheduling policy, which dictates that the spare time of tasks are evaluated when a task finishes execution. Thus, as the task graphs become larger and complex (500 and 1000 tasks), the combination of these factors may increase the number of rescheduling points, increasing the number of migrated tasks which will affect the final makespan. Concerning the size of the rescheduling points, we found that the strategy used in our experiments to evaluate the performance of the reactive mapping methods, which consider a fixed-period rescheduling cycle for the whole spectrum of bounds for each scenario, may not be adequate for some of the bounds. New efforts are required to optimize the size of the rescheduling cycles. We believe that the observed variability of resources can be a parameter to determine the length of the rescheduling cycle.

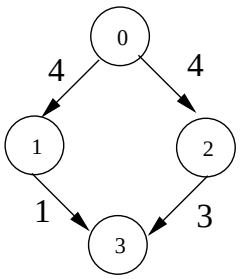
We showed in Section 5.5 that models allowing scheduling and migration of tasks may generate copies, which can be reused in subsequent scheduling decisions as a direct input for tasks which have migrated during the process. Based on this observation, we designed an extended version of GTP called GTP/c. We showed in Section 5.5 that using a small fraction of the total copies generated may improve the makespan of the application. This is because such copies avoid unnecessary data transfer between tasks and exploit the network link which offers the minimum data transfer cost according to the latest performance resource information. However, we believe that further efforts can be made to increase the number of copies used or to decrease them. Whatever the case, reusing data represents a viable approach to enhance the cyclic use of mapping methods.

Fault tolerance is an important issue in SHCS as the availability of resources cannot be guaranteed. Scheduling methods not considering this issue will have to restart the application from the very beginning in the presence of a processor failure. The rewinding mechanism described in 3.5 seeks to preserve the execution of DAG applications, despite the presence of a processor failure. We showed in Section 5.8 that the performance of the rewinding mechanism in a particular 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. Thus, another

benefit of reuse copies is that it allows a better performance of the rewinding mechanism. This is because, as we showed in Section 5.8 there exists a linear chain of events linking the levels rewound with the number of rewound tasks, which determines the overhead cost of rewinding the application. Then, the reuse of copies allows GTP/c to have fewer rewound levels than GTP, being reflected in the number of tasks rewound and in the overhead cost.

6.2 Future Work

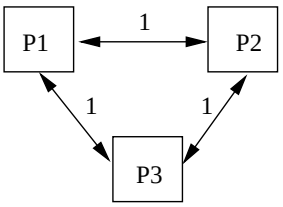
In our research we used the rewinding mechanism in the context of fault tolerance. It is also interesting to ask if there exists any other area in task scheduling in which the rewinding mechanism could be used effectively. We believe that the rewinding mechanism could be used as a scheduling strategy focused on minimizing the makespan of the application. For instance, most of the DAG schedulers in the literature tend to obtain a schedule of unfinished tasks, usually with the objective of minimizing the makespan. However, there could be some cases in which rewinding the DAG (recomputation of finished tasks) could derive a better makespan. To illustrate this, we will use the example of Figure 6.1 which shows the task graph, the SHCS architecture, the static information and the initial schedule obtained by HEFT.



a) DAG application

Task	P1	P2	P3
0	3	2	4
1	1	3	3
2	1	4	3
3	2	4	3

b) Heterogeneous Information



c) SHCS architecture

Task	PE	EST	EFT
0	P2	0	2
1	P2	2	5
2	P1	6	7
3	P1	7	9

d) Initial Schedule

Figure 6.1: Example (t=0) for reactive scheduling with rewinding

Thus, following our approach, and assuming a rescheduling point at t=2, Figure 6.2 shows the information updated for both the progress of tasks and the performance of resources. At $t = 2$, task v_0 has finished execution and we observe a drastic decrease in the availability for P2 from 1 to 0.40. Following the costing of candidate schedules, in which tasks are assigned to that processor which offers the minimum Earliest Finish Time, Figure 6.2(c) shows the Gantt chart of the new schedule generated by GTP, in which v_1 is migrated from p_2 to p_1, such that the new estimated makespan is equal to 10.

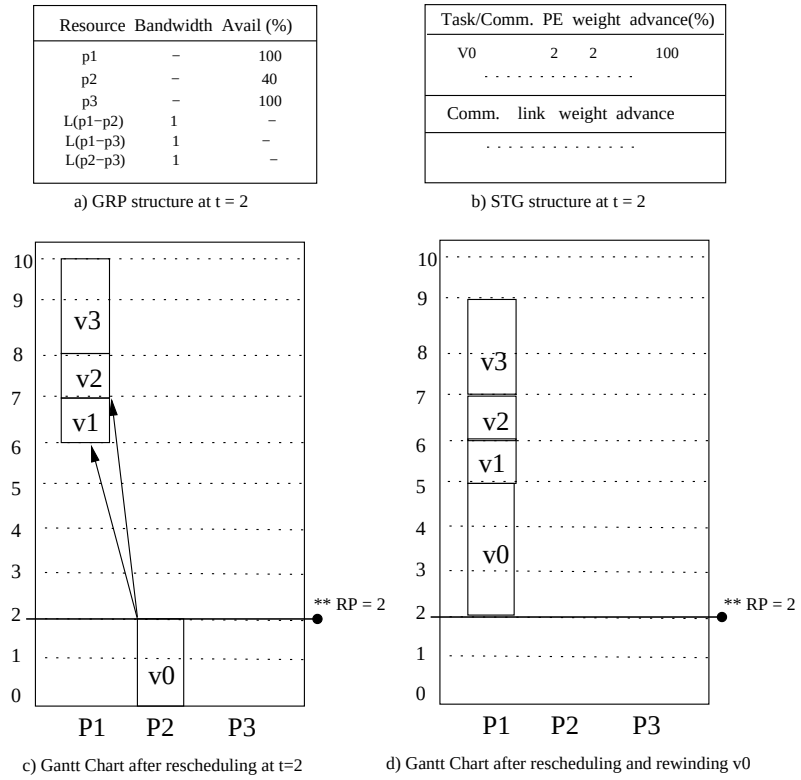


Figure 6.2: Example (t=2) for reactive scheduling with rewinding

However, if at $t = 2$ we first rewind the task v_0 and then apply the costing of candidate schedules for GTP, we will obtain the schedule showed in Figure 6.2(d), in which we observe that v_0 is recomputed at p_1. This action allows us to save 10% in the makespan compared with the previous example of GTP. Thus, the rewinding mechanism can be productively used as a part of scheduling strategy.

6.3 Final Thoughts

In this research, we place strong emphasis in four key aspects, which we believe are central when designing scheduling mechanisms to map DAG applications on SHCS: *reactivity, data-aware scheduling, data transfer flow and reliability*. The first aspect allowed us to explore reactive scheduling mechanisms in response to significant variations in resource characteristics. The strategy of migrating tasks allowed us to efficiently address the dynamic nature of SHCS. Since we believe that new classes of complex DAG applications will emerge to exploit the vast number of resources offered by SHCS, the second aspect was focused on understanding the behavior of the DAG application when it is executed in a reactive environment. Thus, we observed that reuse of data was possible and useful to reduce the impact of migration on makespan by avoiding unnecessary data transfer between tasks, exploiting the network links more efficiently. The third aspect concerns the relationship between the DAG application (defined by the owner of the DAG) and the mapping method (defined by the owner of the method). Thus, we identified two main models to allow the transfer of data among tasks, the *PUSH* and *PULL* models. We explored the impact of using either the *PUSH* or *PULL* model on makespan. We found that, ignoring this issue may negatively affect the performance of the application. Finally, the fourth aspect concerns the reliability of the reactive scheduling mechanisms, as some resources can fail during execution. Thus, we proposed a rewinding mechanism to preserve the execution of the application despite the presence of a processor failure.

Bibliography

Abawajy, J. (2004). Fault-tolerant scheduling policy for grid computing systems. *Symposium on Parallel and Distributed Processing*, pages 238–244.

Abraham, A., Buyya, R., and Nath, B. (2000). Nature's heuristics for scheduling jobs on computational grids. *IEEE Conference on Advanced Computing and Communications (ADCOM'2000)*, pages 45–52.

Adam, T., Chandy, K., and Dickson, J. (1974a). A comparison of list schedules for tasks using static scheduling on oscar. *Communications of the ACM*, 17(12):685–690.

Adam, T., Chandy, K., and Dickson, J. (1974b). A comparison of list scheduling for parallel processing systems. *Communications of the ACM*, 17(12):685–690.

Agarwal, T., Sharma, A., and Kale, L. (2006). Topology-aware task mapping for reducing communication contention on large parallel machines. *IEEE/IPDPS*, page 10 pp.

Ahmad, I. and Kowk, Y. (1998). On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9(9):533–544.

Almeida, V., Vasconcelos, I., and Menasce, D. (1992). Using random tasks graphs to investigate the potential benefits of heterogeneity in parallel systems. *In Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, pages 683–691.

Amstrong, R., Hensgen, D., and Kidd, T. (1998). The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions. *IEEE Heterogeneous Computing Workshop(HCW'98)*, pages 79–87.

Arora, M., Das, S., and Biswas, R. (2002). A de-centralized scheduling and load balancing algorithm for heterogeneous grid environments. *In Proceedings of the*

2002 IEEE International Conference on Parallel Processing Workshops, pages 499–505.

Bansal, S., Kumar, P., and Singh, K. (2003). An improved duplication strategy for scheduling precedence constrained graphs in multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(6):533–544.

Beaumont, O., Carter, L., Ferrante, J., Legrand, A., and Robert, Y. (2002). Bandwidth-centric allocation of independent tasks on heterogeneous platforms. *IEEE International Parallel and Distributed Processing Symposium (IPDPS'02)*, 67:62–72.

Beaumont, O., Legrand, A., Marchal, L., and Robert, Y. (2005). Independent and divisible tasks scheduling on heterogeneous star-shaped platforms with limited memory. *Proceedings of the Conference on Parallel,Distributed and Network-Based Processing(Euromicro-PDP'05)*, pages 179–186.

Beaumont, O., Legrand, A., and Robert, Y. (2003). Optimal algorithms for scheduling divisible workloads on heterogeneous systems. *IEEE International Parallel and Distributed Processing Symposium (IPDPS'03)*, page 14 pp.

Beguelin, A., Seligman, E., and Stephan, P. (1997). Application level fault tolerance in heterogeneous networks of workstations. *Journal of Parallel and Distributed Computing on Workstation Clusters and Networked-based Computing*, 43(2):147–155.

Bell, W., Cameron, D., Capozza, L., Millar, A., Zini, F., and Stockinger, K. (2003a). Optorsim: a grid simulator for studying dynamic data replication strategies. *The International Journal of High Performance Computing Applications*, 7(4).

Bell, W., Cameron, D., Carvajal, R., Millar, P., Stockinger, K., and Zini, F. (2003b). Evaluation of an economy-based replication strategy for a data grid. *International Workshop on Agent Based Cluster and Grid Computing*, pages 661–668.

Bokhari, S. (1981). On the mapping problem. *Transactions on Computers*, 30(3):207–214.

Braun, T., Siegel, H., and Beck, N. (1998). A taxonomy for describing matching and scheduling heuristics for mixed-machine heterogeneous computing systems. *Symposium on Reliable Distributed Systems*, pages 330–335.

Braun, T., Siegel, H., Beck, N., and Freund, R. (2001). A comparision of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed Computing*, 61:810–837.

Buyya, R., Murshed, M., Abramson, D., and Venugopal, S. (2005). Scheduling parameter sweep applications on global grids:a deadline and budget constrained cost-time optimization algorithm. *Software-Practice and Experience*, 35:491–512.

Cao, J., Jarvis, S., Saini, S., and Nudd, G. (2003). Gridflow:workflow management for grid computing. *In Proceedings of the 3rd IEEE International Symposium on Cluster Computing and the Grid(CCGrid'03)*, 2:198–205.

Casavant, T. and Kuhl, J. (1988). A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–153.

Chung, Y. and Ranka, S. (1992). Application and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed-memory multiprocessors. *Proceedings of Supercomputing*, pages 512–521.

Coffman, E. and Graham, R. (1972). Optimal scheduling for two-processor systems. *Acta Informatica*, 1:200–213.

Condor (2001). High throughput computing system. *http://www.cs.wisc.edu/condor*.

DAGman (2002). High throughput computing system. *http://www.cs.wisc.edu/condor/dagman*.

Deelman, E., Blythe, J., Kesselman, C., and Livni, M. (2004). Pegasus:mapping scientific workflows onto the grid. *LNCS, Grid Computing*, 3165:11–20.

Deelman, E., Kesselman, C., Blythe, J., and Gil, Y. (2003). Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1:25–39.

Duda, A. (1983). The effects of checkpointing on program execution time. *Information Processing Letters*, 16:221–229.

Dumitrescu, C. and Foster, I. (2005). Gangsim: A simulator for grid scheduling studies. *In Proceedings of the IEEE International Sysmposium on Cluster Computing and the Grid (CCGrid'05)*.

Ercegovac, M. (1998). Heterogeneity in supercomputing architectures. *Parallel Computing*, 7:367–372.

Eshaghian, M. (1993). A cluster-m based mapping methodology. *IEEE Parallel Processing Symposium*, pages 213–221.

Eshaghian, M. and Shaaban, M. (1994). Cluster-M parallel programming paradigm. *International Journal of High Speed Computing (IJHSC)*, 6(2):287–311.

Eshaghian, M. and Wu, Y. (1997). Mapping heterogeneous task graphs onto heterogeneous system graphs. *IEEE Heterogeneous Computing Workshop (HCW'97)*, pages 147–160.

Faerman, M., Birnbaum, A., Casanova, H., and Berman, F. (2002). Resource allocation for steerable parallel parameter searches. *In Proceedings of the Third LNCS International Workshop on Grid Computing*, 2536:157–168.

Foster, I., Czajkowski, K., and Tuecke, S. (2003a). Open grid services infrastructure (v.1). *http://www.globus.org/toolkit/draftggfogsi-gridservice33_20030627.pdf*.

Foster, I. and Kesselman, C. (1997). Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128.

Foster, I., Kesselman, C., Nick, J., and Tuecke, S. (2002). Grid services for distributed system integration. *IEEE Computer*, 35(6):37–46.

Foster, I., Kesselman, C., Nick, J., and Tuecke, S. (2003b). The physiology of the grid: An open grid service architecture for distributed systems integration. *at http://www.globus.org/alliance/publications/papers/ogsa.pdf*.

Foster, I., Kesselman, C., and Tuecke, S. (2001). The anatomy of the grid: Enabling scalable virtual organizations. *International Journal on Supercomputer Applications*, 15(3):200–222.

Freund, R., Gherrity, M., and Siegel, H. (1998). Scheduling resources in multi-user,heterogeneous,computing environments with smartnet. *Heterogeneous Computing Workshop*, pages 184–199.

Gary, M. and Johnson, D. (1979). Computers and intractability:a guide to the theory of np-completeness. *W.H.Freeman & Co.*

Gasperoni, F. and Schwiegelshohn, U. (1992). Scheduling loops on parallel processors:a simple algorithm with close to optimum performance. *LNCS Parallel Processing:CONPAR'92*, 634:625–636.

Gerasoulis, A. and Yang, T. (1992). A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291.

Gerasoulis, A. and Yang, T. (1993). On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701.

Girkar, M. and Polychronopoulos, C. (1987). Partitioning programs for parallel execution. *ACM Computer Science*, pages 216–229.

Graham, R. (1969). Bound on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429.

GridSim (2002). The gridsim project homepage. *http://www.gridbus.org/gridsim/*.

GridWay (2002). Metascheduling technologies for the grid. *http://www.gridway.org/*.

Hernandez, I. and Cole, M. (2007a). Reactive grid scheduling of dag applications. *In Proceedings of the 25th IASTED(PDCN)*, Acta Press:92–97.

Hernandez, I. and Cole, M. (2007b). Reliable dag scheduling with rewinding and migration. *to appear in First International Conference on Networks for Grid Applications (GridNets07)*, ACM Press.

Hernandez, I. and Cole, M. (2007c). Scheduling dags on grids with copying and migration. *to appear in Parallel Processing and Applied Mathematics (PPAM07)*, Springer LNCS.

Hu, T. (1961). Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848.

Huedo, E., Montero, R., and Llorente, I. (2004). Experiences on adaptive grid scheduling of parameter sweep applications. *Proceedings of the 19th IEEE Euromicro-PDP'04*, pages 263–275.

Hui, C. and Chanson, S. (1997). Allocating task interaction graphs to processors in heterogeneous networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(9):908–926.

Hwang, S. and Kesselman, C. (2003). Grid workflow: A flexible failure handling framework for the grid. *International Symposium on High Performance Distributed Computing(HPDC'03)*, page 126.

Ibarra, O. and Kim, C. (1977). Heuristic algorithms for scheduling independent tasks on non-identical processors. *Journal of the ACM*, 24(2):280–289.

In, J., Avery, P., and Ranka, S. (2005). Sphinx: A fault-tolerant system for scheduling in dynamic grid environments. *In Proceedings of the 19th IEEE IPDPS'05*, pages 12–22.

Jalote, P. (1994). Fault tolerance in distributed systems. *Prentice Hall*.

Jarry, A., Casanova, H., and Berman, F. (2000). Dagsim: A simulator for dag scheduling algorithms. *Ecole Normale Superieure de Lyon, Research Report No. 2000-46*.

Kasahara, H., Honda, H., and Narita, S. (1991). A multi-grain parallelizing compilation scheme for oscar. *In Proceedings of the 4th Workshop on Languages and Compilers for Parallel Computing*, 589:283–297.

Kasahara, H. and Narita, S. (1985). Parallel processing of robot-arm control computation on a multiprocessor system. *IEEE Robotics and Automation*, RA-1(2):104–113.

Kim, S. and Browne, J. (1988). A general approach to mapping of parallel computation upon multiprocessor architectures. *Proceedings of International Conference of Parallel Processing*, 2:1–8.

Kruatrachue, B. and Lewis, T. (1988). Grain size determination for parallel processing. *IEEE Software*, pages 23–32.

Kwok, Y. and Ahmad, I. (1997). A parallel algorithm for compile-time scheduling of parallel programs on multiprocessors. *Proceedings of the Conference on Parallel Architectures and Compilation Techniques(PACT'97)*, pages 90–101.

Kwok, Y. and Ahmad, I. (1999a). Static algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471.

Kwok, Y.-K. and Ahmad, I. (1996). Dynamic critical path scheduling: An effective technique for allocating task graphs to multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):506–521.

Kwok, Y.-K. and Ahmad, I. (1999b). Benchmarking the task graph scheduling algorithms. *Journal of Parallel and Distributed Processing*, 59(3):381–422.

Leangsuksun, C. and Potter, J. (1993). Problem representation for an automatic mapping algorithm on heterogeneous processing environments. *In Proceedings of the IEEE Workshop on Heterogeneous Processing (WHP'93)*, pages 48–53.

Legrand, A., Marchal, L., and Casanova, H. (2003). Scheduling distributed applications: the simgrid simulation framework. *In Proceedings of the 3rd IEEE International Symposium in Cluster Computing and the Grid (CCGrid'03)*, pages 138–145.

Lima, H., Hara, T., Ichimi, N., and N.Sannomiya (1999). Autonomous decentralized scheduling algorithm for a job-shop scheduling problem with complicated constraints. *Proceedings of the The Fourth International Symposium on Autonomous Decentralized Systems*, pages 366–369.

Maheswaran, M., Ali, S., and Siegel, H. (1999). Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 59(2):107–131.

Maheswaran, M. and Siegel, H. (1998). A dynamic matching and scheduling algorithm for heterogeneous systems. *In Proceedings of the 7th Heterogeneous Computing Workshop(HCW'98)*, pages 57–69.

Massie, M., Chun, B., and Cuellar, D. (2004). The ganglia distributed monitoring: Design, implementation and experience. *Journal of Parallel Computing*, 30(7).

McCreary, C., Khan, A., and Thompson, J. (1994). A comparison of heuristics for scheduling dags on multiprocessors. *Proceedings of the Parallel Processing Symposium*, pages 446–451.

MDS (2000). Monitoring and discovery system. *http://globus.org/mds*.

Medeiros, R., Cirne, W., Brasileiro, F., and Sauve, J. (2003). Faults in grids: Why are they so bad and what can be done about it? *In Proceeding of the International Workshop on Grid Computing*, pages 18–24.

Nath, B. (1997). A hybrid gs-sa algorithm for flowshop scheduling problems. *Proceedins of the International Conference on Computer Integrated Manufacturing (ICCIM'97)*, pages 462–471.

NWS (2002). The network weather service. *http://nws.cs.ucsb.edu*.

Pam, M. (1988). Software pipelining:an effective scheduling technique for vliw machines. *In Proceedings of the SIGPLAN'88*, pages 318–328.

Papadimitriou, C. and Steiglitz, K. (1998). Combinatorial optimization: Algorithms and complexity. *Dover Publications, INC.*

Papadimitriou, C. and Yannakakis, M. (1979). Scheduling interval-order tasks. *SIAM Journal of Computation*, 8(3):405–409.

Papadimitriou, C. and Yannakanis, M. (1990). Towards an architecture-independent analysis of parallel algorithms. *Journal of Computation*, 19(2):322–328.

Pegasus (2003). Planning for execution in grids. *http://pegasus.isi.edu/*.

Project, T. E. D. (2004). The european datagrid project. *http://www.edg.org*.

Ramakrishnan, A., Singh, G., Zhao, H., Sakellariou, R., Deelman, E., Vahi, K., Blackburn, K., Meyers, D., and Samidi, M. (2007). Scheduling data-intensive workflows onto storage-constrained distributed resources. *Symposium on Cluster Computing and the Grid (CCGrid'07)*, pages 401–409.

Ranganathan, K. and Foster, I. (2004). Computation and data scheduling for large-scale distributed computing. *Proceedings of the 19th IEEE Euromicro-PDP'04*, pages 263–275.

Sait, S. and Youssef, H. (1999). Iterative computer algorithms with applications in engineering. *IEEE Computer Society*.

Sarkar, V. (1989). Partioning and scheduling parallel programs for multiprocessors. *MIT Press,Cambridge, MA.*

Schopf, J. (2004). Ten actions when grid scheduling: the user as a grid scheduler. *Grid resource management: state of the art and future trends, Kluwer Academinc Publishers,Norwell,MA*, pages 15–23.

Senger, H., Silva, F., and Nascimento, W. (2006). Hierarchical scheduling of independent tasks with shared files. *In Proceedings of the International Symposium on Cluster Computing and the Grid(CCGrid'06)*, 2:16–19.

Shi, Z. and Dongarra, J. (2006). Scheduling workflows applications on processors with different capabilities. *Future Generation Computer Systems (FGCS)*, 22(6):665–675.

Sih, G. and Lee, E. (1993). A compile-time scheduling heuristic for interconnection-constrained heterogenous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187.

Simgrid (2001). The simgrid project homepage. *http://simgrid.gforge.inria.fr/*.

Sinnen, O. and Sousa, A. (2005). Communication contention in task scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):503–515.

Sinnen, O., Sousa, A., and Sandnes, F. (2006). Toward a realistic task scheduling model. *IEEE Transactions on Parallel and Distributed Systems*, 17(3):263–275.

Song, H., Liu, X., Jacobsen, R., Bhagwan, R., Zhang, X., Taura, K., and Chien, A. (2000). The microgrid: A scientific tool for modeling computational grids. *Proceedings of IEEE Supercomputing*.

Spooner, D., Jarvis, S., Cao, J., and Nudd, G. (2003). Local grid scheduling techniques using performance predictions. *IEEE Proceedings of Computers and Digital Techniques*, 150(2):87–96.

Spooner, D., Jarvis, S., Cao, J., and Nudd, G. (2005). Dynamic scheduling of scientific workflow applications on the grid:a case study. *ACM Symposium on Applied Computing*, pages 687–694.

STG (2000). The standard task graph project. *http://www.cs.wisc.edu/condor/dagman*.

Takefusa, A., Matsuoka, H., Nakada, K., Aida, K., and Nagashima, U. (1999). Overview of a performance evaluation system for global computing scheduling

algorithms. *In Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC8)*, pages 97–104.

Taverna (2004). Taverna user manual. *http://taverna.sourceforge.net/manual/docs.work.html*.

Thanalapati, T. and Dandamudi, S. (2001). An efficient adaptive scheduling scheme for distributed memory multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 12(7):758–768.

Topcuoglu, H. (2002). Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274.

Wang, Q. and Cheng, K. (1992). A heuristic of scheduling parallel tasks and its analysis. *SIAM Journal on Computing*, 21(2):281–294.

Wieczorek, M., Prodan, R., and Fahringer, T. (2005). Scheduling of scientific workflows in the askalon grid environment. *ACM SIMGMOD Record*, 34(3):56–62.

Xhafa, F. and Barolli, L. (2007). Immediate mode scheduling of independent jobs in computational grids. *International Conference on Advanced Information Networking and Applications*, pages 970–977.

Yang, J., Ahmad, I., and Ghafoor, A. (1993). Estimation of execution times on heterogeneous supercomputer architectures. *International Conference on Parallel Processing*, 1:219–226.

Yang, T. and Fu, C. (1997). Heuristic algorithms for scheduling iterative task computations on distributed memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 8(6):608–622.

Yang, T. and Gerasoulis, A. (1994). Dsc:scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967.

Yu, J. and Buyya, R. (2005). A taxonomy of workflow management systems for grid computing. *Journal of Grid Computing,Springer*, 3(3-4):171–200.

Yu, Z. and Shi, W. (2004). A dag-based xcigs algorithm for dependent tasks in grid. *Computational Science and Its Applications ICCSA'04*, 3044:158–167.

Yu, Z. and Shi, W. (2007). An adaptive rescheduling strategy for grid workflow applications. *IEEE, Symposium on Parallel and Distributed Processing*, pages 1–8.

Zhao, H. and Sakellariou, R. (2004a). An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm. *LNCS Euro-Par'03*, 2790:189–194.

Zhao, H. and Sakellariou, R. (2004b). A low-cost rescheduling policy for efficient mapping of workflows on grid systems. *Scientific Programming SPR*, 12(4):253–262.