Lehrstuhl für Rechnertechnik und Rechnerorganisation der
Technischen Universität München

Application Specific, Automatic Distributed Evaluation of Performance Data on the Grid

Hamza Mehammed

Vollständiger Abdruck der von der Fakulität für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. H. M. Gerndt
Prüfer der Dissertation:	
	1. Univ.-Prof. Dr. A. Bode
	2. Univ.-Prof. Dr. R. Wismüller, Universität Siegen

Diese Dissertation wurde am 30.04.2009 bei der Technischen Unversität München eingereicht und durch die Fakulität am 23.07.2009 angenommen.

Abstract

The exploitation of distributed evaluation of data in parallel tools has been one of the main concerns of tool developers in their effort to meet the ever-increasing demand on scalability of online parallel tools. Recently, the inherent complexity of parallel machines is enlarged with the sharing, transparency and dynamicity of the available resources in the environment like the Grid introducing a new challenge on performance analysis. In a grid computing environment which is devoted for a coordinated utilization of geographically distributed large parallel computing resources, most of the available parallel tools (like performance analyzers, debuggers and load balancers), which relied on a centralized evaluation of data, often fail to scale well since the centralized evaluation in such modestly sized computing environment mostly leads to a bottleneck at the front-end. Specially, in on-line parallel tools, the results computed at the back-ends must all arrive at the front-end without any loss of data in order to be evaluated properly. This is hard to achieve when the number of back-ends increases and/or they produce computed data more frequently.

This thesis addresses a novel method of automatic, distributed evaluation of performance data in order to decrease the frequency and the amount of data transferred from the back-ends to the front-end to ensure the scalability of the tool. The main research topic is the automatic generation of application specific virtual networks used for the distribution of the subtasks that can be computed on a given location independently. Building such virtual networks depend merely on the metrics specification provided by the users. These specifications are used to describe applicable objects and partner objects (sites, hosts and processes), code regions (portion of source code or location of certain methods) and time specifications (time interval, a point in time or virtual time) which are going to be used for a computation to enhance a very flexible multicast reduction network.

For the realization of the virtual networks, an augmented dataflow model based on the classical dataflow model is developed which supports a hierarchical parallel execution of subtasks and enables also the reassembly of the measurement results asynchronously. The reassembly process includes correlation, aggregation and synchronization of the result data values. Through its hierarchical nature, the virtual network allows the evaluation of data at their origin (instead of transferring them to a central location) by taking the abstraction level of the defined applicable objects into consideration.

The feasibility of the distributed evaluation methodology is performed in a real parallel tool called Grid Performance Measurement Tool (GPM). In GPM, the number of communications between the front-end and the back-ends increases rapidly since the performance measurements are resolved in time, location in the system and location in the code. Using the automatic, distributed evaluation of the performance data, a well manageable front-end is achieved which increases the scalability of the parallel tools.

Acknowledgement

This thesis wouldn't have been possible without the assistance of many people to whom I would like to express my sincere appreciation.

First of all, I would like to express my special gratitude to Prof. Dr. Arndt Bode for providing me with an excellent research environment to work on my thesis. I am also pleased to gratefully acknowledge the contribution of Prof. Dr. Roland Wismueller to my thesis through valuable comments and detailed improvement. In addition, I would like to thank Prof. Dr. Gerndt for all his assistance and advice. As any other research, this work builds upon previous ideas and achievements of many other researchers to whom I would like to express my gratitude.

I would also like to thank Mrs. Margret Bezold-Chatwin for her detailed comments and corrections on language issues. Finally, I would like to express my deepest gratitude to my family for their love, patience, encouragement, and support during the work of this thesis.

Hamza Mehammed
Munich, Germany
April 2009

Contents

Chapter 1

Introduction and Motivation

1.1 Motivation

The demand for powerful distributed computing like the Grid has lead to significant increases in the number of resources residing on different sites which are unprecedented both in amount and geographical distribution. In order to address one of the today's challenge of having on-demand access to any computational service, the "computing as service" vision, the number of processes in modern super computers grows up to hundred of thousands of processors. These super computers can be one of the sites used in the Grid environment which increases the overall number of processes exponentially.

Clusters, as another basic component of the Grid computing, gained renewed importance as the "super-servers" of the emerging Grid infrastructure since the use of computational and data resources in high-performance applications used in a Grid environment have started to become reality. The wide proliferation of grid computing is shown by different projects worldwide. In those projects, resources may reside in different sites of a country (like the D-Grid[1] project), in different countries of a continent (like the CrossGrid [17] project), or in different countries residing in different continents (like the resources of the Enabling Grid for E-sciencE[2] (EGEE) project). As a connection component between those clusters and super computers, the network connection can be one of the critical factors for some applications (like search engines) to run on the Grid, which may raise the necessity to provide an independent network connection to minimize such problems.

In order to support the development of distributed applications running on such environments, parallel tools like performance analyzers, debuggers or load levellers are developed whose scalability issue in such an environment becomes a paramount importance. Those parallel tools are usually used to optimize the distributed application running in a Grid environment. The main activities of such distributed tools are computation, communication and storage, and if the cost of

[1] http:// www.d-grid.de

[2] http://www.eu-egee.org/

any of these activities is larger than the underlying system can support, the overall scalability will be limited by the factor of that cost. The functional components of most of those runtime tools are divided into two parts. The front-end, which usually consists of the user interface for the interaction with the users and for the visualization purpose, and the back-ends, which control the process and manage the collected monitored data.

In most of the existing parallel tools, a single front-end process controls the interactions between back-end tool processes and the process at the front-end. Thus, the front-end spends mostly unacceptably long times managing the reassembly of the results from the back-ends in such a high scaled environment. Therefore, there should be a new way of managing those big amount of data produced in such an environment to replace the classical approach of a centralized evaluation. The centralized approach leads mostly to a bottleneck at the front-end and increases the complexity of the interactions between the front-end and back-ends. Unfortunately, the centralized evaluation mechanism is the design pattern of most of the existing important parallel tools. Because of the performance degradation caused by this kind of centralized computation, failure in large-scale systems become commonplace.

It is difficult for the users to search for bottlenecks in the application which are, for example, related to the Grid environment or application specific data. For example, the determination of inefficiencies in application is a time-consuming and a very complex task but is a necessary step to ensure that the application behaves as expected and users can use the provided expensive resources efficiently. One of the important tool is a performance analysis tool on the Grid which is used for the identification of bottlenecks and provides different information in order to find the real cause of the inefficiencies.

The context of this thesis is a parallel tool in general and an on-line performance analysis tool in particular. On-line tools are closed-up tools that dynamically control the application or system based on their analysis while the programs as well as the monitoring are running which makes the computation more complex. The distributed evaluation provided in this thesis deals with the performance analysis tools' computation and communication activities.

1.2 Challenge

In order to satisfy the aggressive optimization of Grid applications, the parallel tools used for the optimization (like a performance analysis tool) must be adapted to the challenging new Grid environment with an enormous amount of resources. One of the challenges to cope with the increasing number of resources is to provide an efficient way of managing those thousands of processes. A decentralized evaluation process is one of the approaches used usually to solve such problems. A distributed evaluation as proposed by this thesis, is based on a multicast and re-

duction mechanisms. This kind of approach can also be used as a possible solution for other parallel tools in the same category.

One of the main prerequisites for an on-line performance measurement tool is that the measurement results computed at the back-ends must be delivered to the components at the front-end without any significant lose of data [1, 2]. The reason for those possible loose of data is that the front-end communicates unnecessarily too often with the back-ends over a network to collect and analyze the measurement results and this increases the number and size of the data to be managed by the front-end. The intensive communication results also a complex and unmanageable interaction between the front-end and the back-ends.

One of the main challenges of this thesis is to provide a solution on how such scalability problems can be solved with an efficient distribution and collections of the necessary computations. This includes:

- management of a large number of communication partners and processing their monitored data in a distributed way.
- efficient communication between the front-end and back-ends in both directions which includes an efficient broadcasting control commands, and collecting the possible aggregated result data.
- partition of the main computation task into subtasks which can be evaluated on remote hosts independently, aggregation of the results of such subtasks, as far as possible, and sending only the necessary results to the front-end.
- managing the aggregation of time-aligned computed data which belongs to the same time interval or point of time, and synchronizing results that belong together.
- avoiding the participation of the user in all distributed evaluation activities by performing all the steps automatically, as far as possible.
- computing application, as well as Grid infrastructure specific measurements by supporting predefined as well as user defined high-level measurements.
- supporting request as well as event triggered measurement evaluation.

One of the challenges is managing the partition of the main task into its subtasks and to distribute them to their corresponding remote host. An efficient distribution of those subtasks which can be performed at a specified location without the need of further communication with other processes residing on another hosts, is very important in the Grid environment where inter-site communication can limit its over all performance. This is also a reason why communication intensive applications are not suitable for the Grid.

Most of the performance analysis tools available today provide low-level information which desires expertise knowledge to deal with them. High-level and

application oriented metrics are meaningful not only for the application domain specialist but also for the application users. Providing a possibility to deal with such metrics enables custom tailored solutions for the applications and the Grid environment based measurements.

In the process of performance measurement (as it is also the case for monitoring, debugging, and visualization tools), the effect of concurrency and non-determinism play an important role in an event based evaluation. A Distributed application consists of processes which communicate by messages and execute sequences of elementary or atomic actions called events. Therefore, a proper understanding of a distributed program and its execution through determining the causal and temporal relationship between the events that occur in its computation is indispensable. Event based measurements enable, for example, phase based performance analysis of parallel programs, which are poorly addressed by the existing tools.

Managing all the above listed challenges will improve the scalability behavior of performance analysis tool by preventing a scalability bottleneck within the tool system itself by avoiding resource saturation, which may be caused by the tool processes. The distributed evaluation enables obtaining not only global application performance data (such as CPU or Memory utilization across all processes), but also complex aggregated values.

It is desired that all the possible computation details of the provided solution be performed in an automated and user-friendly way since application developers are not expected to be performance analysis experts. In addition, the realization of all the suggested concepts must not have the effect of any performance degradation on the tool to assure its quality.

1.3 Solutions

Since most HPC systems usually have a life period of five years, providing a solution depending on the design of a system, which may vary in the next design generation, will not be a good solution. An important goal of this thesis is therefore, to develop a powerful execution model based on a distributed evaluation to achieve a scalable distributed computing environment.

The scalability problem mentioned above can be reduced to its minimum by distributing the evaluation of some tasks from the front-end to the back-ends. Through this distributed evaluation, customizable, scalable and high-throughput communication software system applications and parallel tools can be developed. This solution provides a more accurate performance measurement result in an online and scalable way through off-loading the computing activity from the tools' front-end to the back-ends.

This distributed evaluation is designed through developing an Augmented Dataflow Model (ADFM) based on the classical Dataflow Graph (DFG). One of the advantages using a dataflow model is that the execution sequencing is constrained

only by data dependencies. And as a result of this the control information is the same as the control of data. Using the DFGs for software based solution has two main synergistic parts: first, an efficient way of distributing subtasks can be achieved and second, a high throughput data assembly can be efficiently realized.

Since distributed systems are loosely coupled in the sense that the relative speed of their local activities is usually not known in advance, it is also necessary to synchronize the result values to assure a proper computation. This kind of synchronization is supported by the DFG automatically. The DFG used is composed of dataflow nodes and data providers where dataflow nodes are used to realize the firing rule and the aggregation methods. In addition, those nodes are used to enforce synchronization of result values. The data processing abstraction embodied in the dataflow nodes of the DFG facilitates the aggregation and reduction operation on in-flight result packages. Multiple result packages are supported by the DFG as input as well as output providing the same result values for different result consumers, which is a shortage of other approaches based on, for example, a tree-like structure. The data providers are used to store the result values and facilitate the synchronization of result values.

The topology of the DFG is determined for every measurement implicitly by the user during the specification of the metrics in a very flexible way. Therefore, there is no need of having statically defined DFG topologies. Since every back-end is responsible for the subtasks to be computed there, there is also no need of assigning extra resource for the dataflow nodes of the DFG. To control the complexity of the generated dataflow graphs, sub-DFGs describing the same tasks are combined into a composite sub-DFG. In effect, application processes, nodes or sites are clustered dynamically based on their computational behavior and this enables an earlier data reduction and efficient multicast of subtasks to be performed on the corresponding site, node or process independently.

Using the DFG approach allows the building of an overlay network built on the real network. This facilitates to control the execution of the measurements. Using this kind of computation provides a very flexible distributed computing and shows significantly better performance than the centralized approach. Since the on-line nature of some parallel tools results in a collection of stream data, and most of the computed data are coming in an asynchronous manner, the issues associated with the assigning of the proper consumer of those data must be addressed. This must be determined during the distribution of the subtasks and by the time of building the requests to the monitoring system.

In the course of finding the performance bottleneck in an application, the developer of the application might want to measure some metrics that are critical for the application. Describing those metrics is simplified using Performance Metrics Specification Language (PMSL). For the aggregation, correlation and reduction purpose, all the mathematical operations are supported. Using this dedicated specification language, one can combine and chain the provided mathematical relations and operations to realize any desired custom data reduction. The distributed eval-

uation of this concept is performed on a real-world on-line semi-automatic performance measurement analysis tool using real world interactive applications.

1.4 Contribution of the work

The main research goal of this thesis is not only the development of an effective, distributed evaluation of performance measurements, but also presenting a powerful execution model for the same sort of computation. This is achieved by using the proposed dataflow model which increases the efficiency and throughput of the executions by supporting parallelism automatically.

A scalable data analysis with a possible minimum overhead is also achieved by supporting aggregation and correlation chaining which facilitates a complex time-aligned data aggregation mechanism. This guarantees the management of large volume of dataflow by providing an automatic partition of the dataflow graph created from an intermediate representation of an execution and reassembling the appropriate results automatically.

For the purpose of distributed evaluation, a complete process, as depicted in the Fig. 1.1 has been developed that consists of the parsing process (arrow 1) to translate the measurement specification into an intermediate representation in form of DAG which is optimized to its minimum number of nodes and sub-DAGs (arrow 2), the generation of a DFG from the intermediate representation (arrow 3), generating the desired monitoring requests to facilitate an efficient and fast processing of the computed data (arrow 4). For the reassembling the computed result data which are arriving asynchronousely and must be routed to the proper consumers, a notification mechanism (arrow 9) is used. In order to evaluate the specified measurement in a distributed manner, the DFG representing the whole measurement task is partitioned in to different sub-DFGs which represents subtasks which can be evaluated in their corresponding hosts independently as shown by arrow 4 in Fig. 1.1. Following that a communication (arrow 9) between those sub-DFGs and the main DFG at the front-end will be set up in order to enable result data assembly when the measurement is performed. The final result will be then sent to the user interface as described by arrow 11 in Fig. 1.1.

As a real-world tool to demonstrate the feasibility of the distributed evaluation idea, the Grid Performance Analysis Tool (GPM) and its underlying Grid-enabled OMIS[3]-compliant Monitoring System (OCM-G) is used. Both of them are developed in the CrossGrid IST-2001-3224 project (the EU 5^{th} Framework Program). One of the GPM components, as depicted in Fig. 1.1, is called Performance Measurement Component (PMC) and deals only with the built-in metrics. In order to evaluate those predefined metrics, the PMC communicates directly with the underlying monitoring system OCM-G using the OMIS-Interface [31] as shown by the arrow 6b.

[3]On Line Monitoring Interface Specification

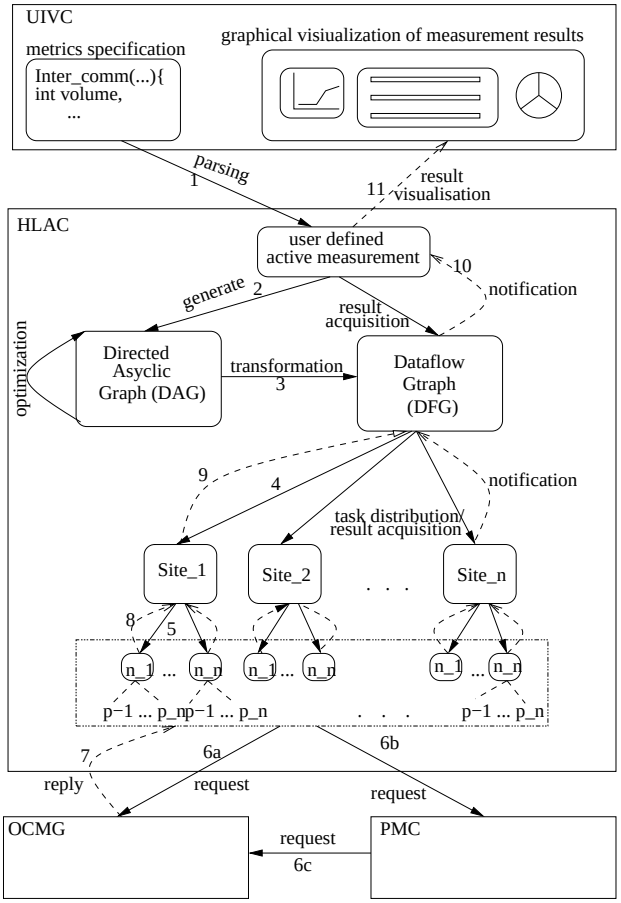

Figure 1.1: The overall processes of the distributed evaluation using GPM and OCM-G

All those activities are performed without the participation of the user and are performed in an online and distributed manner without altering the application runtime behavior. This online behavior allows the user to use the experience gained in the previous measurements into the next measurements. Lastly, a graphical representations of all intermediate representations (including the DAG, DFG and partitioned sub-DFGs) is generated automatically in order to provide an insight into sequence of computations and helps the user to verify the specification of the measurements.

1.5 Scope of the Thesis

The rest of the thesis is organized as follows:

In Chapter 2, the background of the thesis is discussed which includes performance analysis, the Service-Oriented Approach (SOA) of Grid computing and

the interactive applications of the CrossGrid project. It shows that a performance analysis is a very vital process in developing Grid applications in general and interactive application in particular.

An important but non-trivial aspect for understanding the behavior of distributed programming is to understand the causality and its relation to logical time. Therefore, some fundamental characteristics of event triggered measurements and their influence on the performance measurement data evaluation is discussed in chapter 3. In the same chapter the design and development of the proposed ADFM for the distributed evaluation based on the classical DFG is formally described.

In chapter 4 a brief survey of related works is presented. It discusses the state-of-the-art of distributed evaluation which mostly relies on a tree-based approach, the developments of different specification languages used to describe measurements, and the origin and scope of dataflow model. Chapter 5 is devoted to the discussion of the performance analysis tool GPM and the monitoring tool OCM-G, which are used as context of this thesis.

Chapter 6 illustrates the main work of the thesis and shows how the distributed evaluation is designed and developed. It includes the description of the Measurement Specification Language (PMSL), the specification of measurements, creating intermediate graphs, performing monitoring request and reassembling remotely evaluated measurement results. The comparison of the performance gained by using the distributed versus centralized evaluation, some usage scenarios used to show the distributed evaluation and the generation of the graphical intermediate representation (DAGs and DFGs) created during the distributed evaluation are discussed in chapter 7. Finally, the last chapter presents the outlook and summary of the thesis.

Chapter 2

Background of the Thesis

2.1 Performance Analysis

In order to obtain the results of many applications developed in different fields of science in a very short period of time or to handle the large possible problem size of those applications, the physicists, biologist, chemists, etc. are interested in environments like the Grid. Grid computing provides an efficient way of solving data and compute intensive problems by providing a big amount of resources. Nevertheless, the programmers should exactly know the performance of his application in such an environment in order to use the provided efficiency. Even a profound knowledge of the code of the applications does not contribute much for its detailed run-time behavior and thus parallel tools like performance analyzers should be applied for that purpose. Since the Grid environment is not under the control of a single administrator and involves a dynamically changing system resulting in a non-reproducible working condition, performing performance analysis on it can be a complex task.

Performance analysis on the Grid is not yet defined precisely. Even the term Grid does not have an exact definition [6, 47] since there are different kinds of Grids for different purposes which makes it difficult to provide a general solution concerning performance analysis in the Grid environment. In general, performance analysis deals with the qualitative, quantitative and economical aspects of the resources. Resources on the Grid includes but is not limited to servers, clients, special equipments, networks, storages service instants, and the ultimate goal of the Grid computing is to provide the resources as well as services for co-ordinated usage.

In order to have a benefit from the promising approach of parallel and Grid computing to solve complex problems in many scientific fields, the application in focus must be designed to perform properly. To achieve this goal, a performance tuning must be accomplished after the implementation and functional test of the application. To perform the performance tuning, performance analysis must be performed that includes the detection of performance bottlenecks, and pinpointing the reason of the performance flaws. Following this, the application code responsi-

ble for the performance degradation must be modified to optimize and thus improve the over all performance quality. Thus, performance analysis is a cyclical process consisting of those steps which must be repeated until the desired performance behavior is achieved. This time-consuming and complex process must be accomplished before the application is used in a production environment. In addition, the analysis processes must be performed in a similar environment as intended for the production usage.

To support the performance analysis of the application, information taken from a performance prediction can be used. When there is no suitable performance prediction available for the desired analysis, an unexpected behavior of the application can be taken as a starting point. Consuming an unacceptable much time for the communication by certain processes is an example of such poor performance. Usually, performance experts should do the task of performance analysis unless the process is automated to some extent.

In general, there are lots of reasons for the performance degradation of an application which can not be identified using a performance analysis tools. Using a wrong algorithm, for example, cannot be identified through any of the performance analysis tools available today. Another category of problems is functional incorrectness of the applications, which will results in computational error. Therefore, applying a performance analysis tool can have an advantage only when it is applied on applications which have proven to be functionally correct and use an optimized and efficient algorithm.

Normally, performance analysis should be used to check the performance of the application in the context of the distributed environment and the suboptimal usage of the underlying programmer model such as waiting for a message. Performance analysis covers, in general, the topics of data acquisition and data processing. The former deals with monitoring system observing the instrumented application whereas the latter deals with analysis and visualization of the computed data. In order to get the data about the application and/or Grid infrastructure, the performance analysis tool uses the monitoring system. Usually, the type and amount of data collected will have an influence on the expressiveness of the performance problem that can be detected.

2.1.1 Methods for Collecting Monitoring Data

There are different ways to perform performance analysis, which can be categorized according to the way the monitored data are collected. The well-known mechanisms handling the monitored data are tracing, profiling and the online approach. Those mechanism are described as follows:

Tracing: Using this method, all the desired and undesired data are recorded from the sequence of events occurring during the application run. Since the amount of data collected using this mechanism can be very huge, specially for long running applications, there must be a way of filtering the necessary information for the visualization. Since extracting the desired information will have a very big over-

head, the analysis will be performed after the program has finished its execution. Therefore, the use of this mechanism is devoted for an offline usage and for applications not running for a long period of time (see also 2.1.3). ***Vampir***[1] [9] is a good example for an offline performance analysis tool which uses a stream of recorded data from a trace file.

Profiling: This is a way of collecting data desired for performance analysis in a selective way. Particularly, the result of a profiling is the data collected about the frequence and duration of function calls in a program. Using this information, an optimization of the program code can be performed for those methods which are executed very often and are highly time-consuming at the same time. Comparing this to the tracing mechanism, profiling reduces the amount of data to be collected and gives summarized information about the applications behavior. The Unix tool ***gprof***, for example, is a profiling tool which presents the complete call graph analysis listing each function and how much of program execution time is consumed by each of them.

The online approach: The more interesting approach is to analyze the monitored performance data in an online fashion that provides the unique opportunity to change the measurement constraints and to see the changes immediately at the run time - as expected by the interactive applications. Since in the online approach the data are not stored persistently, they are used immediately in the computation, or accumulated in the aggregator. The main advantage of this mechanism is the possibility to access the data as soon as they are produced and that there is no need to deal with huge amount of data stored (see also 2.1.3).

2.1.2 Measurement Analysis

To start the performance measurement of an application, one can look for undesired behavior of the application. Fortunately, certain parts of the code are usually known to be critical for the performance of the application, especially routines used for the communication and I/O. In general, the cause of performance problems related communication can be identified at different levels [18]: applying communication library which is not optimized for a given system, using unnecessary blocking time in a receive primitive, operating system features (e.g. inappropriate buffer size), incapability of the underlying hardware concerning latency, bandwidth, to mention a few. In case of MPI, for instance, the performance degradation of a CPU usage can be one of the hints for imbalance in the volume of communication. Inspecting the delay behavior of each process may then help to find out where the reason for the bottleneck could be.

Even if a lot of tools are provided for the performance analysis and monitoring of applications running on the Grid, no single tool may satisfy all the needs of the end users (Grid users, administrators or tool developers). This is due to the

[1] *http://www.vampir.eu*

diversity of the requirements of the applications and their type as well as the type of Grid.

Despite this fact, a significant set of functionality can be achieved by using complementary tools together as suggested in [20, 40]. Using GRM/PROVE and mercury [66], for example, will provide most of the desired functionalities. In this case GRM is used for the application performance analysis whereas PROVE is applied to visualize trace data online, and mercury to provide Grid performance monitoring based on the push and the pull mechanisms. Another example in this category is the combination of the SCALEA-G [42] and the Askalon [61] tools. In this combination, SCALEA-G (based on the OGSA standard) provides online measurement using the pull and the push approach for profiling and tracing application related as well as Grid infrastructure related information. On the other hand, the Askalon tool is used for the visualization purpose. Since Askalon is based on Java and XML technologies, it is easier to use it in other tools.

In the EU-DataGrid[2] which is a homogenous Grid environment (supporting only RedHat Linux operating system), almost 10 different complementary tools are used to satisfy the diversity of requirement arising there. This clearly shows the difficulty to find a suitable performance and monitoring tool, which satisfies the desire of the majority of the users. Since most of the tools are not interoperable with each other, the same functionalities provided by different tools co-exist unnecessarily (like the performance analysis ability of both Askalon and SCALEA in the example above). Providing interoperability at least at the level of standard, self-defining data format for the input and outputs may help.

When performance analysis and monitoring tools are used in a Grid environment, the security issue of the Grid environment (as discussed in 2.2.4) must be taken into consideration. Even if security is not a big issue in the process of performance analysis, there are reasons in some commercial sectors to deal with it. Some companies do not want to reveal how and when they use the provided resources. Monitoring the application belonging to other users must also be protected, since some monitoring systems facilitate, for example, the manipulation of process activities.

2.1.3 Online versus Off-line Performance Analysis

There are two different ways of analyzing the performance of an application in a distributed environment: online and offline. Depending on the environment and types of applications, those two mechanisms are used in different manners. The offline approach uses tracing and profiling, whereas the online approach tries to evaluate the data as they are produced and uses also profiling to have an access to the summarized computed values.

[2] http://eu-datagrid.web.cern.ch/eu-datagrid/

On-line Performance Analysis

For the long-running and/or interactive applications, the method of collecting all possible monitored data and extracting the desired data is not suitable. The results of the performance analysis must be provided immediately to give a feedback to the user in order to adapt their activity in response to the application's behavior during the run time of the application. The available result values can be used again to set up new measurements during the application run.

The online approach enables defining and processing performance related measurement at run-time since it makes possible to focus on specific execution aspects which can be the cause of the performance problem. As a consequence, tuning the application can be performed on the fly without stopping, recompiling, or rerunning the application. Comparing this to the offline approach, performance problems can be identified earlier.

In addition to this fact, the Grid environment can not be reproduced once again to achieve the same resources utilization due to the dynamic nature of the Grid, and this makes the collected data to be a special case which may not represent the next run time results and this is one of the reasons not to collect data in a Grid environment. Nevertheless, there are also disadvantages of the online approach. One of them is the overhead of processing the monitored data. To reduce this analysis overhead, there is a possibility to shift the analysis process to a dedicated machine since this overhead must be kept to its minimum in order not to perturbate the running application's behavior. The instrumentation overhead can be reduced to its minimum by inserting and removing the instrumentation at the run time according to the actual behavior of the application.

Off-line Performance Analysis

The offline performance analysis, which is carried out in a post-mortem manner, is a widely used method for analyzing the performance of parallel programs. In this case, the traced data containing information about the interactions between different processes or threads that occurred during communication or synchronization operations will be examined. As a result, information how concurrent activities influence each other's performance can be studied.

The performance analysis based on this approach allows investigating all the event data gathered during the application run. Since it is usually not known a priori which data are important, detailed information about the application must be recorded which requires the generation of many events by the instrumented code. Therefore, this mechanism results in processing a huge amount of data that must be stored permanently in a centralized storage, and thus must be transferred to the centralized component.

Using the traced data, a comprehensive analysis can be performed since all the necessary information is available, and time is not a critical factor in such cases. This also allows reproducing the application scenarios on demand. That means,

the investigation about the application's behavior can be performed after the application finished its execution. This method of analysis does not introduce any extra overhead in to the execution of the application, with the exception of the overhead results from the monitoring process since the analysis as well as the transfer of data can be performed after the application is executed. Using some patterns to identify the most important performance bottlenecks, the analysis process of the traced data can be simplified. In addition, the recorded trace data are also important for statistics and archiving purposes.

However, recent applications are too large to use this approach. Monitoring an interactive application in an offline mode may be feasible, but will not be efficient and easy to implement. In this case, not only the information about the application's behavior will increase the amount of trace data to store, but also the user's interaction must be monitored and stored. For long running application, it is not convenient to use offline monitoring since the user wants to see the results of the interaction immediately to enable application steering.

2.1.4 Automatic Performance Analysis

The huge amount of data produced in a Grid environment needs an intelligent and automatic filtering, aggregating, and/or converting mechanism to provide a clear and abstracted view representing only the necessary information (see also [55]). Automated mechanisms are indispensable to deal with such enormous amount of data through which a transparent (from the user's point of view) transfer of low level data to a high level compact data will be performed. The row data provided by the monitoring system are enough to give useful information to the end users.

The heterogenous Grid infrastructure, its dynamic nature and the different administrative domains used in the Grid environment adds an enormous amount of complexity to the performance analysis process by producing further performance flaws that can hardly be allocated without an automatic mechanism. Without reducing the amount of data automatically, it will be very difficult for the end user to deal with all the monitored data manually. That means that user involvement for the performance analysis process must be as minimal as possible to achieve a highly effective large scale performance tuning. Even by using an online performance analysis tool where monitored data are not collected but analyzed as they are available, the monitored data volume in a large-scale application can be huge enough to create management problems. This automated approach also relieves the user from the difficult tasks of identifying which performance data is important and which is not. In addition, the complexity of the interaction between the performance analysis tool, and the underlying monitoring system will be transparent to the users when writing applications to run in a Grid environment in a distributed way and to perform performance measurements to tune the application.

2.1.5 Instrumentation

Instrumentation is a mechanism to insert code portions into programs or library interposition linked to the application to observe the execution of certain part of the code and to provide monitored data [38]. Adding a code to a program or library can be performed at different levels of the program execution: before or during the compilation of the program code, at the time of linking a program, or at the run time. Since each of those levels contains different kind of information, the corresponding collected data differ in quality and quantity.

From all the instrumentation mechanism, the binary or dynamic run-time instrumentation is a complex but efficient way of instrumentation. An API for such purpose can be found in [23]. This mechanism is used to collect performance data for the actual measurement at the location where the important performance data are expected. This allows to point out the nature and location of the performance bottleneck more precisely. Since this kind of instrumentation is performed while the application runs, there is a certain amount of overhead which can alter the behavior of the application execution and thus must be taken into consideration. Due to this overhead, dealing with such instrumentation on the fly may not be possible or may perturbate the application's behavior, therefore the modification to be performed must not involve very complex processes.

2.2 Grid Computing

The vision of Grid computing [3, 6, 47] is, in general, the transition of today's world of static, manual, application-oriented world towards the dynamic, virtual, automated and service-oriented approach. Grid computing started as a generalization of cluster computing, promising to deliver unprecedented levels of parallelism to high-performance applications by crossing different administrative boundaries. Grid evolved to support on-demand access to a composition of different computational services provided by multiple independent sources. Grid computing provides the solution to utilize computing resources of organizations (both in academia and in industry) in an efficient way. Since most of the Unix servers are idle more than 90% of the time and most of the desktop machines are busy less than 5% of the time, exploiting this issue will bring the advantage of increasing the resource utilization enormously. This gap of utilization can be compared to an airplane using just 10% of its passenger seats. Grids are also used for collecting, processing, and searching large data sets.

Unleashing the computer power of an organization will provide benefits to have a huge business, education and research facilities which will be used to reduce the inefficiency caused by idle resources through harnessing the technology innovation. Using the concept of virtual organization will also enable sharing the avaliable, locally controlled resources easily and efficiently through providing a simplified collaboration between different heterogenous organizations which may have different policies.

One of the goals of the Grid computing is to solve problems arising during the running of a program on different machines belonging to different organizations. An existing remotely executable application which can be broken down to a number of jobs on a remote machines, may need to fulfil some special hardware, software and resource requirements which should be managed in an efficient way. Thus, the main problem areas of Grid computing includes security, data processing, and data and information management between systems distributed across the world. In the near future, Grid computing can also be used to solve real time problems, as far as the reliability of the desired resources is guaranteed. Such kind of approaches can be seen as a beginning of utilizing autonomic computing in a Grid environment, which automatically tries to repair problems.

Since the resources used in a Grid environment are unlikely to be of the same type or running the same operation system, open standards like Open Grid Service Architecture (OGSA) and Web-Service Description Language (WSDL) are used to solve problems concerning interoperability of services provided by different organizations. The same question will be raised for the security issue, which may be satisfied by using the Grid Security Infrastructure (GSI) technology as discussed in 2.2.4. Interactions in a Grid computing environment are not just client/server, but also service-to-service which require service access on behalf of the user which needs the delegation of rights by user to services which is also covered by GSI.

Types of Grid

Even if there is no clear boundary for the type of Grids [3], the following types of Grids can be identified from which computational Grid has reached today much higher level of maturity than the other type of Grids.

Computational Grid is a type of Grid computing that most products are applying today. This is a term used for aggregated processing power involved in the Grid environment. The components of these types of Grid are mostly high performance servers, which provide a very big amount of computing power for applications that need those big super computers for their computation. There are a lot of projects[3] around the world which represent this type of Grid. TeraGrid, NorduGrid and EGEE are good examples for this category.

Data Grid is a kind of Grid computing that provides a secure access to distributed, heterogenous data which includes a federated database. Data Grid is often underestimated in the sense that it is supposed to be the largest challenge in the Grid computing field. The questions how to bring structure and order into the large amount of data produced, for example, during the research process of the LHC (Large hadron Collider) Computing Grid (LCG) are challenging. A sophisticated, scalable and robust data management is required to deal with the exponential growth of data which are presented in many different formats, distributed over many sites and replicated for the sake of international collaboration in a virtual

[3] http://www.globus.org/alliance/projects.php

organizations. The main problem areas are heavy load, performance bottlenecks, and dealing with different policies. In spite of this fact, the response time for the services dealing with those data must be kept at a minimum.

Scavenging Grid is another type of Grid which is mostly related to the computational Grid. It uses a large number of desktop machines to provide a large amount of CPU capacity. As an example, ***SETI@home***[4] uses the idle CPU cycle of personal computers which is mostly 95% of their running time. In this case, those idle CPU cycles are used to analyze a radio transmission received from outer space. This kind of Grid is useful to come up with problems which requires more computational power than available. Most of the time, this kind of computation is not suitable for real time computation, since the speed up of the computation of an application depends highly on how network intensive those application will be.

2.2.1 Resource Sharing

The most common resource of a Grid environment that can be shared is the computing cycle used to run applications that cannot be processed in a single local environment. The second most used resources are storage capacities, which can either be memory or secondary storages. Another big storage can be achieved through using federated databases which contain different types of database from different providers accessible in uniformed manners.

Other resources focus on data communication capacities. These kind of resources are very important for the success of Grid computing since the Grid resources can be dispersed geographically. As an Example, the DEISA project uses a dedicated network for its distributed computing to avoid such problems. The communication capacity of the Grid environment can be used not only to transfer the jobs to the remote machine and to fetch the result from them, but also to duplicate a large amount of data used as input for applications. In addition to those resources, software, special equipments, and licenses can also be shared.

To access those resources, an organization can use its own policy to assure priorities not only for resources but also for users. Using a Service Level Agreements[5] (SLA) as a contract that can exist between two service providers, the negotiation of services, priorities, etc. can be arranged to specify the levels of availability, serviceability, performance, or other attributes of the service.

For instance, licenses that are purchased in a limited amount in an organization because they are very expensive, can be made accessible using a Grid environment to get their full advantage. This can be performed by sending jobs to the remote machines where those licenses are available. Expensive special devices like telescopes which cannot be replicated and thus may be accessed remotely, can be used in a Grid environment in an efficient way.

[4] http://setiathome.ssl.berkeley.edu/

[5] http://www.grid scheduling.org/

2.2.2 Grid Applications

Not every parallel job should run in an environment like the Grid. Suitable and simple jobs for Grid computing are especially batch jobs that expect a set of input data to produce sets of output data. If the input data is of large amount and continuously needed during the runtime of the application, this data can be replicated and managed automatically using the Grid computing service for data replication and file stage-in mechanisms are used to transfer the files before and after the application is execution. Jobs that expect a small number and amount of input data for their execution, which may take a large amount of time and produce any type of output data, are best examples of simple Grid jobs. On the other hand, jobs which may have an intensive communication of the processes running on different machines which are geographically distributed, will spend a lot of time for the communication to send data over the network and they are thus not suitable for the Grid.

Grid computing includes also replicating shared data to avoid transferring a huge amount of data repeatedly. Through replicating the necessary data, input data are brought near to the host where the application is running. Some highly advanced scheduling techniques use this future to increase their efficiency. Scheduling is used to minimize the execution time of tasks through providing the required resources, and thus increase the overall performance.

Grid Computing can also be used to balance the resource utilization of an organization through migrating running jobs with their current state. Such job migration is used in case the resource in focus fails or some priority must be achieved. This is the case when an organization has an expected peak of activities, which demands more resources.

Grid applications are, in general, single-instruction-multiple-data (SIMD). Enabling an application in order to run it in the Grid environment is not a trivial task and at present there is no automated way of performing such processes. Through the network limitation, it is also clear that not all applications can be Grid enabled to achieve high performance. Since almost all new parallel programs are well optimized to be executed in parallel in a Grid environment, such issues are mainly concerning applications designed earlier. One of the most attractive features of Grid computing is to provide massive parallel CPU capacity. This requires that the applications use algorithms which support the partition of the main task into its sub tasks so they can be executed independently.

Unfortunately, there are indeed a lot of factors which prevent the partition of a task to its sub tasks using input and output data exclusively. This can have both design and/or algorithm backgrounds. The scalability of an application depends mostly on the algorithms used to partition the main job into sub-jobs. Theoretically, a perfect decomposable application would be n times faster when it uses n number of resources. That means that this application will have n tasks running independently. From the user point of view, a Grid environment depends on the running application and the speed of the interconnecting network. In general, the

time spent on data exchange must be negligible compared to that of processing the corresponding data.

2.2.3 Virtual Organization

In a Grid environment, different resources belong to different administrative domains, which can be federated to form virtual organisations. The Grid can offer a resource balancing effect by scheduling Grid jobs within the virtual organization. An advanced scheduler for load balancing purpose can use those virtual resources. This minimizes the communication traffic since individual resources can provide their availability and capacity, which will be used for resources brokering.

Virtual organization can also be used as a basis for a functional reliability. Instead of using duplicated resources (like power suppliers), an organization can share resources in order to achieve reasonable reliability. Some virtual organization use a mountable file system like AFS (Andrew file system), NFS (Network file system), DFS (distributed file system), or GPFS (global parallel file system), which increases the capacity of the virtual system. Some of these systems support an advanced synchronization that can be used to reduce inconsistency whenever many different users shared data.

A virtual organization can offer the management of priorities among different projects. Through controlling and managing the underutilized resources of certain projects and using them in projects which may have a higher priority for a given period of time.

2.2.4 Components of Grid Computing

The basic set of components of the Grid computing generally encompasses the following components: grid security infrastructure, resource management, information management and, data management components. There are different kinds tool which either support the development of Grid software (e.g. Globus) or providing software which enables some Grid functionality (e.g. Uniform Interface to Computing Resources (UNICORE[6])). Since the standardisation process bei OGF is not yet completed, functionality of different middleware co-exist on the same host. In the reference installation of the D-Grid project, the three well-known grid software are used at the same time on the same host [4].

Grid Security Infrastructure (GSI)

Security is a basic factor in any IT-System. In a Grid environment, it is mostly divided into three fundamental services: Authentication, authorization and Encryption of the data. Authentication is a process to find out whether an individual is the one who is claiming to be whereas authorization is used to control the access right to an individual. On the other hand, encryption of the data will insure that data

[6] http://www.unicore.eu/

are neither destroyed nor altered by unauthorized persons during the data transfer between resources which assures the data integrity.

The GSI [7] is based on the Private Key Infrastructure (PKI) containing private and public key pairs. A Certificate Authority (CA) is used to guarantee the ownership of each of the public keys. The basic of GSI is the Generic Security Service Application Interface (GSS-API) used mainly for single sign-on mechanism and is a standard specified by Internet Engineering Task Force (IETF). Beside the functionality mentioned, it includes the confidentiality issues involved during the communication between Grid resources. This privacy prevents that no data will be accessed by individuals that it was not intended for.

For a secure data transfer, TLS is be used to encrypt all data transfer between the Grid sites to assure the data confidentiality and integrity. The encrypted tunnel created by TLS uses a Grid session ID. To make the data transfer faster, the asymmetric encryption usually used by all Grid components is replaced by symmetric encryption.

Authentication, Authorization and Encryption

Generally, in a Grid computing environment, not only the users, but also every other resources must authenticate itself before accessing any Grid functionality. This authentication is based on a mutual authentication of both sides using TLS. From the Grid user's point of view, it is necessary to have a proxy before submitting jobs. Myproxy server are also used as online credential repository to store x.509 proxy credentials for a later retrieval over the network. Myproxy is used mainly by Grid portals (which provide Grid functionality using web browsers) for the authentication purpose since it provides an automatic credential update functionality, in case the proxy used is expired. To realize the single sign-on concept of Grid computing, a remote delegation of credentials (supported only by GSI) is used to submit jobs to a remote machine which may in turn submit some sub-jobs to other remote machines on behalf of the user.

In order to have a secure remote communication using a GSI, GSI-SSH can be used. This enables the user a secure remote authentication with a local proxy and can also be configured to support remote delegation as stated above. If there is no local proxy, or GSI-SSH is not configured on the remote host, a usual SSH will be automatically used in place.

Resource Management

The resource management component provides an interface for requesting and using remote system resources especially for the execution of jobs. This addresses a range of jobs where reliability, stateful monitoring, credential management, and file staging are important. Therefore, the resource management component plays a

[7] http://www.globus.org/security/overview.html

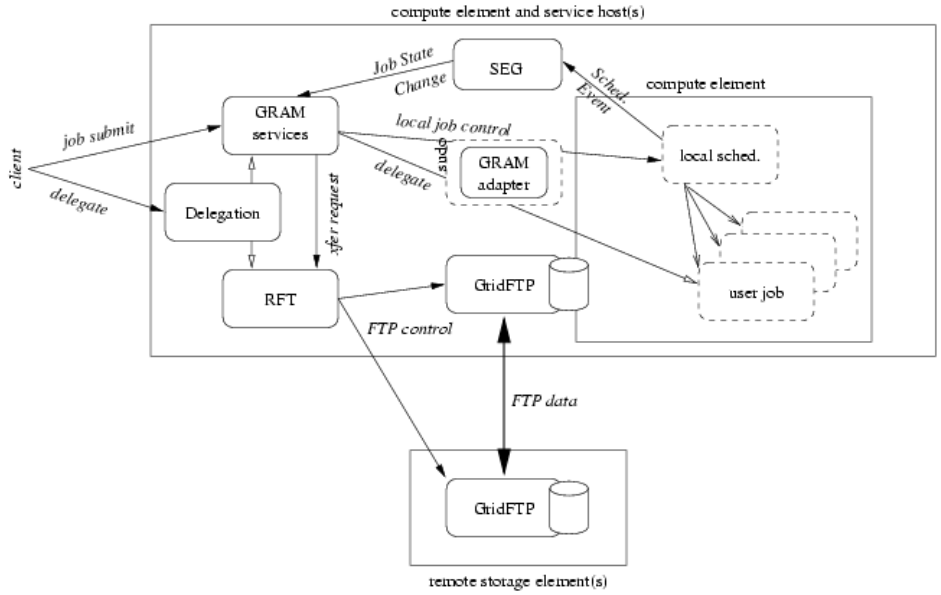

Figure 2.1: Job submission using GRAM

decisive role in a Grid environment by acting as an interface between the difference heterogenous resources and the consumer of those resources.

In general, the resource management helps to ensure an optimal work of all available elements of Grid solutions in an organization. Workload management helps to assure, for example, that SLA can be met. This can be challenging when multiple applications are running from the same user on the Grid. If the Grid environment in focus is fully utilized, jobs with lowest priority may be suspended for the time being to execute the jobs with higher priority in order to assure the negotiations, which guarantee the Quality of Services (QoS) as desired by the SLA.

The corresponding component in the globus toolkit is the Globus Resource Allocation Management (GRAM), which supports a stateful job control and assures the reliability of the submitted jobs by allowing an asynchronous monitoring and control of job activities. It uses a file staging mechanism to transfer input and output data using RFT as depicted in Fig. 2.1. Through implementing a protocol enabling a communication with scheduler, GRAM forwards batch jobs to a local scheduler, which can be PBS, LSF, Condor or SGE. Using the meta-scheduler GridWay[8] makes the choice of the schedulers transparent for the user. For the security of the network, a firewall[9] should be used. GRAM is more complex than, for example, Remote Procedure Call (RPC), since its life cycle includes file staging, execution and cleanup mechanism. Fig. 2.1 shows the interaction of the components used in GRAM.

[8]http://www.Gridway.org

[9]http://www.globus.org/toolkit/security/firewalls/

Information Management

One of the important features of a Grid environment is providing dynamic and distributed resources. To cope with constantly changing resources, the availability of the resources and their status must be updated as often as possible. Those information resources include but not limited to software, network, CPU load, memory and storage space.

Information management is concerned with extracting, distributing, indexing, and processing information about the configuration and states of the services and resources that can be used to search suitable resources and to allocate them. This also provides a specialized view to support discovery of resources in a VO community. To provide Grid-wide resource information, usually a distributed indexing mechanism based on XML, which facilitates querying using e.g., XPath is used. The distribution is realized by collecting the local indices of each installation in a tree-like hierarchy. Each directory index service provides the necessary static and dynamic information of the local resources. This enables workload management tools or schedulers to retrieve actual information to find a suitable resource for a given application.

It is possible and is also advantageous to have similar kinds of resources providing the same functionality in a single Grid environment. Providing such redundant resources insures the QoS of the Grid environment by providing alternative resources in case some resources fail, or when the same kind of resource is desired by different component at the same time.

Data Management

The data management part is concerned with the transfer, allocation and management of distributed data and provides a robust, secure, fast and efficient transfer of a bulk of data. In the Globus toolkit, GridFTP and RFT are used in such a way that GridFTP takes the secure data transfer part by providing stripped and parallel stream mechanism whereas RFT deals with the reliability of the transfers used to recover from failure of network outages, and server and client failures. For the proper functionality of the GridFTP, Data storage Interface (DSI) must be available, usually achieved by using the standard POSIX system. DSI for Storage Resource Broker (SRB) and high performance storage systems is also available.

A number of storage devices connected together in a Grid environment can be regarded as a single massive data storage system. As one of the leading projects dealing with data in a Grid environment, LHC provides the date storage capacity as its main shared resource. In the near future, the service provided by LHC will have the same type of services provided by WWW at present. This fosters to achieve the ultimate goal of the current Grid development to provide a single global Grid environment.

Other important components of data management are Replica Location Service (RLC) and Data Replica Service (DRS). The former is used to keep track where

replicated data are kept on a physical storage system through using a distributed registry mechanism to register files from users or services. This is performed by using logical and physical names when files are created. Managing those replicated data can be challenging if the number of those data are very high. For instance, the RLS provided by the Globus Toolkit manages some 40 million files across 10 Grid sites at present. DRS provides a high level data management web service, which uses a pull based reallocation capability for the files. It uses the functionality of RFT and RLS. In order to deal with data stored in a different database (which may based on XML or relational database), OGSA-DIA provides a web service allowing to query, update, transform and deliver structured data as well as semi structured data.

2.3 Interactive Applications

The CrossGrid project[10] aimed to extend the functionality of Grid computing with interactive applications (having a typical characteristics of involving a human being in the processing loop for computational steering) provides four interactive Grid applications which are discussed below. Tools facilitating the development and tuning of interactive applications and to adapt existing applications for use within a Grid environment are presented. There are also services developed which supports the new interactive functionality as shown in the Fig. 2.4. The description of GPM and OCM-G can be found under the tools and services provided by this project and are discussed briefly in chapter 5.

The CrossGrid project co-operated with other European and international Grid projects, such as Globus, LCG (former DataGrid), EGEE (former EuroGrid), Grdi-Lab and GridStart, to provide services to those projects and to have also an access to the services provided by those projects. For instance, it developed enhancements to the LCG middleware to submit interactive jobs transparently to the Grid.

One of CrossGrid's research and development effort was the development of the four user-friendly Grid-enabled interactive applications, both to serve as proof of concept for the project approach, and to solve actual important scientific problems within the EU. Due to the difference of scientific fields and the corresponding different tools used by the CrossGrid project, the software modules developed was heterogenous in nature, which results in using non uniform methodology and programming models. This was a barrier to have a unified architecture. In spite of this fact, the products of the projects are based on Web-Service and XML technologies in order to facilitate its future integration to the products based on the OGSA and WS-RF based Grid environment.

[10] http://www.eu-crossgrid.org

2.3.1 CrossGrid's Interactive Applications

The interactive applications developed in the CrossGrid project have the objective to design and develop large-scale Grid-enabled applications for simulation and visualization that require real-time responses from the system. Thus, the main challenges are pointing to the distribution of source data, simulation and visualization. The main activities of the provided tools and services was porting those models to the Grid, integrating them to the CrossGrid testbed and optimizing the underlying simulations. Those applications cover simulation of vascular blood flow, flood crisis support tool, meteorology and air pollution simulation, as well as data mining in High Energy Physics. In order to show the complexity of these interactive applications, their description will be presented here.

Interactive Simulation and Visualization of a Biomedical System

This work has demonstrated the Grid-based Problem-Solving Environment (PSE) for the vascular reconstruction procedure, which includes a solver for blood flow simulation, a Virtual Radiology Explorer (VRE) system for applying the bypass procedure as shown in Fig. 2.2, and Grid Visualization Kernel (GVK) for the Grid-enabled visualization of the result of the blood flow simulation.

The simulation volume is divided in to several sub-volume, and each sub-volume is processed concurrently in order to achieve parallel execution of the components. The inputs for this simulation are scanned data of the patient, which are first segmented so that the arterial structures of interest remain in the dataset. The segmented data are then converted into a computational mesh as shown in the Fig. 2.2. Some other natural inputs for the Virtual Reality (VR) are context sensitive interaction by voice, hand gestures and direct manipulation of virtual 3D objects generated from 2D scans. The VRE runs on a local machine. There is also a lightweight version of the portal deployed on a PDA as a thin-client, which is a Java portlet and enables monitoring the simulation progress.

The demonstration of this interactive simulation shows the possibility of having a virtual surgery procedure on the Grid using Grid resources, which allows the medical experts, mainly radiologist, to intervene in the simulation to get a support in their pre-operative decision-making. A performance analysis tool GPM plays a big role here to optimize the complex interaction to be performed. Using the available environment in the project, the following results are achieved in this context: secure Grid access, node discovery and registration, Grid data transfer, application initialization, medical data segmentation, segmented data visualization, distributed blood flow visualization and bypass creation as shown in the Fig. 2.2[11].

The pictures 2.2 show the overall process of the medical simulation. From top left to bottom right the process can be described as follows: A patient is scanned in the Netherlands; the result data will be stored (e.g. in Poland); the data will be segmented, filtered and cropped, using a Grid service; a bypass is added; a

[11] http://www.ercim.org/publication/Ercim_News/enw59/sloot.html

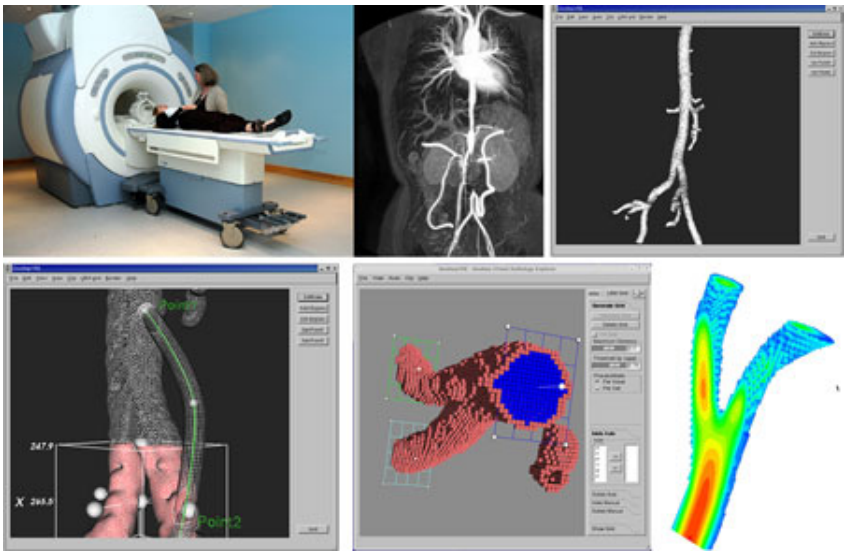

Figure 2.2: Distributed blood-flow simulation and visualization on the Grid

computational mesh is generated (on the local machine) and given to the parallel flow solver, running Amsterdam and Spain; the resulting flow fields are displayed on the local machine using visualization services offered by the GVK in Austria.

Flooding Forecasting Application Simulation

The flood simulation is used to forecast a possible flood in a specific region and to compute borders of disaster areas. The application core consists of two Grid services: Workflow engine and metadata catalog as shown in the Fig. 2.3. Those services operate on another service provided by the CrossGrid and LHC Grid middleware. The application itself is built on metrological, hydrological and hydraulic simulation models, and post processing tools, connecting those simulations and forming the Grid-Work-flow.

The metrological model used to forecast precipitation used by hydrological model for computation of discharge of the river, is a parallel application using MPICH-P4 for interprocess communication. This application is communication intensive (circa 4 MB/s on 100 MB Ethernet network) with scalability of up to 8-10 nodes depending on the size of the problem. The hydrological models are sequential jobs running only a few seconds. Thus, a multiple parallel execution of the jobs with modified parameters was possible.

The result of the whole simulation can be registered to the replica manager, which is used to download a file found in the meta-data catalog, and the meta-data describing those results will be stored in the meta-data catalog service for later search and retrieval. The CrossGrid Portal and the Migration Desktop as shown in Fig. 2.4 provide a complete data management enabling the user to search and

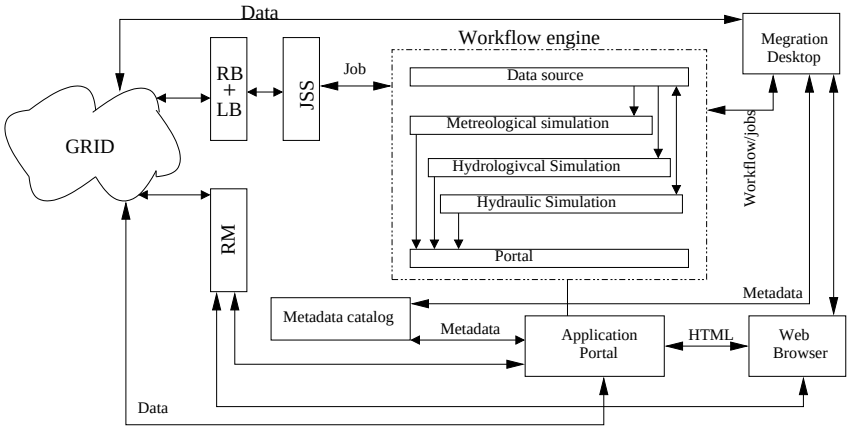

Figure 2.3: The architecture of the flood application

browse the meta-data. The work-flow engine uses the job submission service (JSS), which uses the resource broker (CrossBorker) based on the scheduler developed in DataGrid project, and Logging and Bookkeeping (L&B) services to select the appropriate resources that will be used for job execution and job state change events logging as shown in the Fig. 2.3.

To manage the creation, execution and deletion of the user's work-flows, a work-flow service providing an interface to the work-flow engine and database, runs the whole work-flow in the Grid, and handles the job dependencies automatically. This service stores also the previous work-flow result in order to provide a possibility to compare the results.

Distributed Data Analysis in High Energy Physics (HEP)

The next generation of HEP like LHC at CERN requires unprecedented computing resources for data analysis. In LHC, a proton will be accelerated in 2008 to get hundreds of times heavier particles called Higgs-Boson, which may occur once in 10^{12} collisions and is the key to understand the origin of a mass and consequently the origin of the universe. The amount of data which should be collected and analyzed to get this information is 15 Petabytes ($15 * 10^{15}$) of data annually coming from 10^9 proton-proton collisions every second. Those data should be stored and computed in a distributed way and data mining will be applied for the analysis.

To facilitate this, this part of the CrossGrid project develops an end-user application providing the interactive histogram and parallelized Artificial Neural Network (ANN) training using the Migration Desktop as a user interface.

Whether Forecast and Air Pollution Simulations

This application focuses on long- and medium-range weather forecasts for the Baltic Sea basin, as well as comprehensive air pollution modeling for selected sites. The simulation consists of a metrological modeling, which is the driving force data delivery service, atmospheric pollution and wave models for the use by atmospheric and oceanographic community.

The atmospheric model produces data for air pollution models, wave sea models, and data mining applications. This MPICH-P4 application is communication intensive and thus unsuitable to run in an environment having more than 16 CPUs. The aim of running this application in CrossGrid environment was to have a possibility of running different setups of the model parallel in different domains. The air pollution model uses metrological data and runs MPICH-G2 application requesting three-dimensional wind and rain information. Investigating the distribution of air pollution, which is one of the tasks beside the whether forecasting, uses information about the chemical concentration of the air, the topology of the region and metrological data.

2.3.2 Tools and Services for the Interactive Applications

To facilitate the development and tuning of distributed, compute- and data-intensive, interactive applications on the Grid as mentioned above, a set of tools and services was developed and integrated in the CrossGrid project, which foster the deployment of the parallel applications to the Grid. Those components are shown in Fig. 2.4. The main tools developed include MARMOT, a debugging and verification tool for MPI programs for both C and FORTRAN language binding; a Grid Performance Measurement Tool (GPM) to detect application-specific bottleneck; a performance Prediction Component (PPC) to predict the behavior of both application dependent and general purpose computations and communications; and a custom tailored benchmarks dealing with data transfer, synchronization, I/O delay and CPU utilization.

Beside the monitoring service OCM-G for application monitoring (which is the context of this thesis), the following services are developed to enhance the deployment and development activities of the interactive applications: Grid portal and Migration Desktop (MD) for the user friendly interaction between the user and the provided services; a roaming access, i.e., a mobile personalized environment, supported by a dedicated Roaming Access Server (RAS); CrossBroker for the purpose of job scheduling; JIMS for the infrastructure monitoring; and a Grid-enabled System Area Network Trace Analysis (SANTA-G) for monitoring mainly the network traffic. The Grid Visualization Kernel (GVK) is also used as standalone service dealing with the visualization of the results of the simulation of an interactive application. Those services are also intended to augment the middleware components developed in Globus Toolkit, DataGrid, and EGEE Projects. In order to assure the

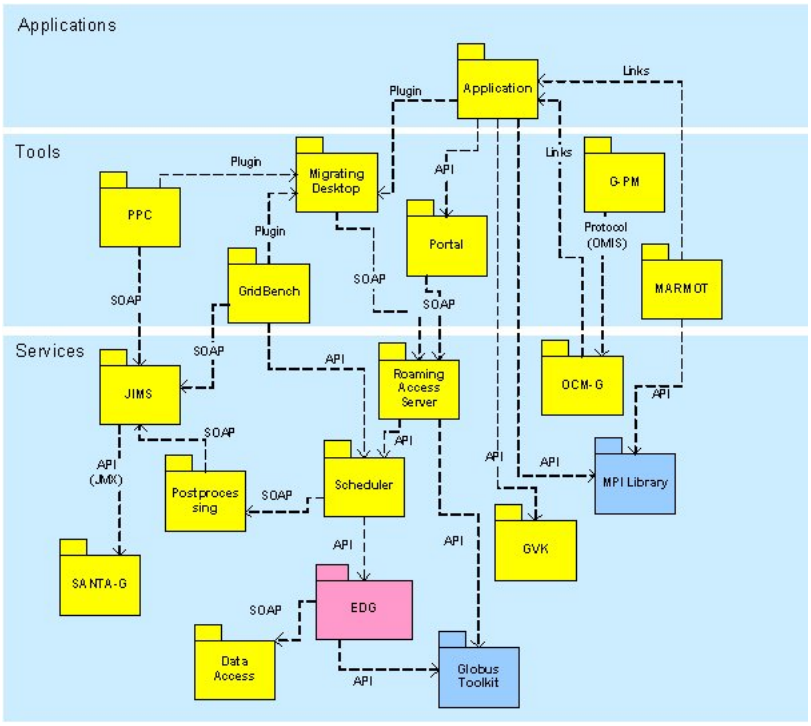

Figure 2.4: Components developed in the CrossGrid project

integration between the applications, programming tools and the new services, a realistic Grid-environment through an international testbed was set up.

Chapter 3

Evaluation of Measurement Data

3.1 Introduction

In this section the measurement issues of event triggered measurements and an efficient execution model for the computed result data based on a Dataflow Graph (DFG) approach is discussed.

It is often the case that the same event in an application can occur at different times in different processes depending on, for example, the load of the machine in focus. Because of this, one cannot assume concurrency of events in a parallel computing environment. A metrics measurement for an interval can result in inaccurate results since the desired event detection may occur out of the time interval scope. To deal with this and similar measurement behaviors, the first section of this chapter deals with different measurement issues. Following this, the development of an efficient execution model based the flow of data is illustrated. Using a dataflow model in the parallel computing environment is a novel approach that provides a clean model for parallel computation of continuous streaming data since it automatically supports parallelism. The Dataflow Model is suitable for composability and decomposability of tasks which are vital for the distributed evaluation.

3.2 Measurement Issues

Tools for performance analysis measurement, like GPM [16, 74], usually show the behavior of applications which are described using the corresponding metrics. Some of those tools visualize the result of those metrics measurements in an online fashion; others use the post-mortem technique providing the result values in an offline manner. GPM, as one of the tools using the online approach, displays those results while the application is running. Providing measurement results in an online manner is much more difficult than the offline approach due to the fact that the results must be provided on time and the mechanism used must not have an effect on the application running.

The prerequisite to measure any quality is that the object to measure has a well-defined value at any point in time [7]. For instance, a tree has a well-defined length at any point in time, which can also be constant for a long period of time. If the tree, for example, is cut down and processed to produce papers, the length can be considered as zero. There are also cases where it does not make sense to measure. For example, measuring the velocity of a house will not have any meaning, as far as the house is not sinking.

With the same sense, a well-defined measurement metrics describes a measurable quantity, which has a defined value at any point in time as discussed in [7]. Therefore, the value of a metrics is a function of time : $V(t)$. Depending on the visualization component, the value of this metrics at discrete points $t_1, t_2, t_3, t_4, \ldots$, t_n are sampled values at those points in time and given as:

$$V_1 = V(t_1), V_2 = V(t_2), V_3 = V(t_3), \ldots, V_n = V(t_n)$$

In order to show more accurate measurement result values, the update interval to collect those sampled values must be as short as possible. Having a short update interval, on one hand, will result in an intensive communication between the front-end and the back-ends via the network leading to unmanageable front-end. On the other hand, the underlying monitoring system may not be able to provide results of measurements in such a short period of time which may result in a less accurate result values. Those and related problems and their possible solutions will be discussed in this section. Even though different components of a parallel tool can be consumer of the computed measurement result values, a visualization component is choose to be as a consumer of the measured result values to simplify the discussion.

3.2.1 Acquisition of Monitoring Data

For the visualization of performance measurement result values, we will discuss the two different approaches of measurement visualization. The first one shows the current measurement values using, for example, bar graphs. The second one presents an aggregated measurement value showing the values evolution over time, for example, using a curve diagram. In both cases, the update interval of the measurement values, as determined by the visualization component is given as $\triangle t$:

$$\triangle t = \max(\triangle t_i)$$

where

$$\triangle t_i = t_{i+1} - t_i, \text{ with } i \in [1, n-1]$$

In order to show the current measurement value V_{cur} at a time t_{dis} where d_{min} and d_{max} denotes the minimum and maximum communication delay (latency), because of the delays in communication, needed to transfer the result data from the back-ends to the front-end, the following equation should be satisfied where t_{cur} donates the current time.

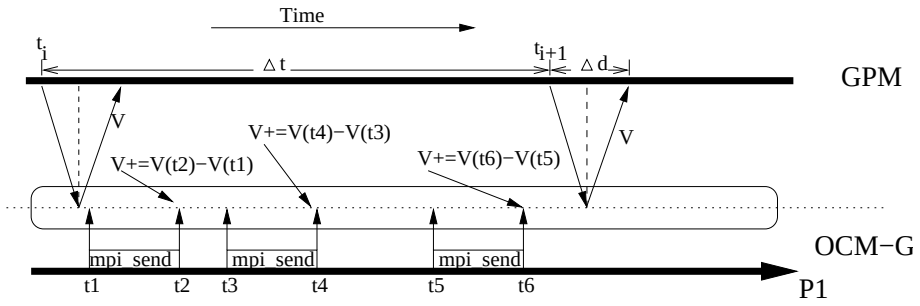

Figure 3.1: Interaction between front-end and back-end

$$\exists t_{dis} \in [t_{upper} - \triangle t, t_{lower}] : V_{cur} = V(t_{dis})$$

where

$$t_{upper} = t_{cur} - d_{max} \wedge t_{lower} = t_{cur} - d_{min}$$

When $\triangle t \rightarrow 0$, the graph showing the values evolution over time converges to the function graph $V(t)$. In order to achieve this optimal value, the update interval must be short enough. This means that the data acquisition of the visualization component must happen more frequently. This increases the communication between the visualization component and the underlying monitoring system, which may be result in bottleneck at the front-end.

To show the aggregated values using curve diagram, the d_{min} and d_{max} above will not play an important role since the current results will not affect the graph that much. Fig. 3.1 shows the data acquisition by the visualization component where $\triangle d \in [d_{max}, d_{min}]$. In this example, the data returned by the volume V is accumulated from all ***MPI_Send*** occurred in the given measurement interval $(\triangle t)$**.**

In some special cases, as it is sketched further below, achieving those theoretical values, however, is not as feasible as it appears to be. In order to aggregate different values at different time, the following additivity property of the measured values must be fulfilled which is a pre-request for further computations based on this assumption.

$$\forall t_0, t_1, t_2 \text{ with } t_0 \leq t_1 \leq t_2 :$$
$$V(t_0, t_2) = V(t_0, t_1) + V(t_1, t_2)$$

This equation enables us to compute measurement results in a measurement interval as shown below, where t_s represents the start time of the whole measurement:

$$V(t_1, t_2) = V(t_s, t_2) - V(t_s, t_1)$$

To compute the time derivative measurement values, which gives, e.g., the bandwidth measured when sending data, the following equation can be used which

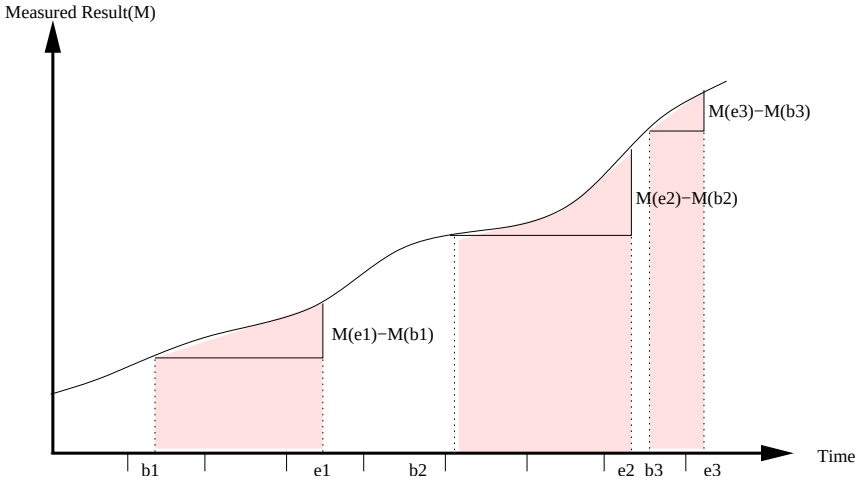

Figure 3.2: Curve diagram showing measurements at the begin and end events

does not require the result value at the start of the measurement measured by $M(t_s)$. The additive metrics assigned by $M'(t_{last}, t_{cur})$ is used to measure the computed data between t_{last} and t_{cur}. Those points are represented by begin (b) and end (e) events in Fig. 3.2.

$$M'(t_{last}, t_{cur}) = \frac{M(t_s, t_{cur}) - M(t_s, t_{last})}{t_{cur} - t_{last}}$$

$$= \frac{M(t_{last}, t_{cur})}{t_{cur} - t_{last}}$$

3.2.2 Acquisition of Event Based Monitored Data

Since we are dealing with different processes which communicate by messages and execute events, one can aggregate the values delivered by those events by associating the event occurrence.

This is possible since those events occur at a particular process are assumed to be atomic in nature and linearly ordered by their local sequence of occurrence. In addition to this fact, those kinds of events can be modeled as having a negligible duration. There are different possibilities to deal with event related data. One can compute the collected data whenever an event occurs. In order to measure computed data in an interval, the beginning and the end of the interval can be associated with two different events which can be associated to each other. In [16] it is discussed how to detect the reason for a performance degradation, which involves event based measurement evaluation. Use cases as discussed in section 7.2) illustrates how the distributed evaluation is performed using the dataflow technologies.

There is also a possibility to use an event in a loop to monitor the activities of every iteration. For event triggered measurements, since the necessary actions

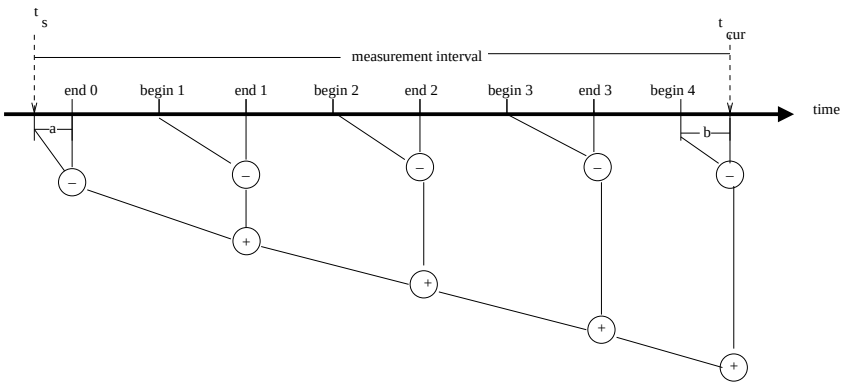

Figure 3.3: Accumulating the result values of event based measurement within a measurement interval

are modelled by events, time needs only be advanced with the occurrence of an event and is therefore discrete. Thus, it is obvious that with respect to the causality relation the exact global time at which those events happen is of no concern. Therefore, the so-called virtual time is used which is an arbitrary and monotonically increasing integer number used to associate events belonging together.

Using the information delivered by the begin and end events togather with one or more built-in metrics, some meaningful metrics can be specified. A value of such metrics is computed by subtracting an accumulated sampled value of the standard metrics at the end event from the one at the beginning, as shown in Fig. 3.2 and 3.3. For a single event occurring in the application loop, the value of a previous iteration will be subtracted from the current one.

Actually, this is an ideal model which cannot be met easily in a distributed computing. The very first prerequisite for this kind of computation is that the begin and the end of the measurements are picked up on all the processes involved to compute the desired metrics at the same time. This is not feasible in a distributed environment where the relative speed of the process's local activity is usually not known in advance. Even if one repeats the execution of the same algorithm in different processes, the execution time and message delays may vary substantially. In addition to this fact, there is no perfectly synchronized system clock and this can affect some measurement activities. Therefore, it is not easily feasible to identify concurrent activities in such a distributed computations which does not share memory.

In general, we are dealing with a distributed system consisting of n sequential processes P_1, P_2, ..., P_n communicating by the means of messages. A coordinated execution of the same algorithms on all of those processes (except the master process which usually manage the computed result values coming from the backend processes) forms a distributed computation. Thus, a convenient way of representing such a computation would be a space-time diagram as sketched in Fig. 3.4.

Since simultaneity cannot be guaranteed in a distributed system, an event cannot trigger a measurement on a remote node properly. In Fig. 3.4, the measurement values marked with “a” will not be taken in to account even if it belongs to the measurement at the begin event . A computed value of P_1 is sent to P_2 after the begin event detected at P_2. The values computed at marker “a” and/or “b” will not also be taken in to consideration because of the very same reason.

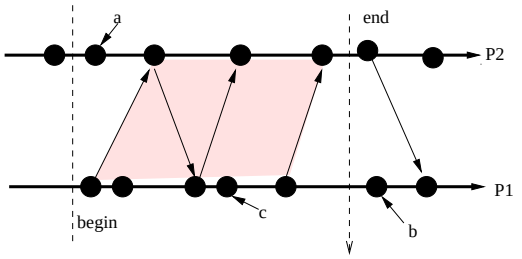

Figure 3.4: Causal relationship between actions performed by different processes

A possible solution, which also results in providing more accurate measurement results for this kind of problems, is to compute the measurements between those events for each process separately and to aggregate them when the corresponding end events come up. This technique helps us also to evaluate measurement results at the place where their corresponding events occur which is an important factor for the distributed evaluation of measurements. Even if the measurements between the begin and end events are measured separately for each process, there are other measurement problems to deal with.

One of the problems arises because of the communication delay of the underlying monitoring processes. That means, measurement result values at the beginning and at the end of an interval may be handled in the way which may not be desired by the user. The user may intend to include all end events in spite of the fact that they may be out of the measurements interval scope, or he may be interested only for the values computed in the measurement interval. This is due to the fact that some measurements at the end and at the beginning of the measurement interval may not be taken into consideration or they will be taken into account mistakenly since they may be out of the scope of the measurement interval like the first begin event of P_2 and the last end event of P_4 as shown in Fig. 7.11.

If, for example, one of the pair events is not detected at all since it is not within the measurement or update interval, we cannot compute the measurement result value for that pair of events without modifying the provided approach. When the visualization component tries to read the measurement results between pair of events belonging together as depicted in Fig. 3.3, we must also compute the measurement results computed until the first end event is detected.

This can happen, especially, when the update interval of the visualization component is shorter than the interval between the begin and the corresponding end event. That means, if the visualization component reads a result value after a begin

event is detected, but the corresponding end event does not yet occurs, then the result value computed between the time when begin event is detected and the time when GPM reads the result value must be computed. On the other hand, when the measurement is started between a begin and an end event, the value computed between the start of the measurement and the begin event must be deducted from the final result.

Let us now see the ideal case where the measurements are computed for all pair events belonging together. Assuming that the measurement is started between any two consecutive events, the result of the measurement M (providing the measurement results between the start of the whole measurement and the current read time which is also between to consecutive events) can be modelled as follows:

$$M(t_s, t_{read}) = \sum_{i|t_s+\triangle d/2<t_i^b,t_i^e\leq t_{read}+\triangle d/2} (V(t_i^b) - (V(t_i^e))$$

Where t_s represents the start time of the whole measurement, t_{read} donates the time when the measurement is read for the last time, t_i^b describes the i^{th} begin event, and t_i^e the corresponding end event. Assuming that the request and response delay for the data acquisition represented by $\triangle d/2$ are equal, $t_s + \triangle d/2$ and $t_{read} + \triangle d/2$ describes the accurate start and read time of the monitoring system, respectively. As shown in the Fig. 3.1, $\triangle d$ describes the latency between the monitoring interface and the visualization component, therefore the time elapsed at the beginning till the monitoring system get the request is $\triangle d/2$ and is also the same at the end of the measurement during the data acquisition by the visualization component. This shows that the measurement value $M(t_s, t_{read})$ can be measured at any point in time.

In order to implement the measurement M, the following algorithm can be used:

1. $M_{-1} := 0$, to initialize the result value
2. $S_i = V(t_i^b)$, when a begin event is invoked
3. $M_i = M_{i-1} + V(t_i^e) - S_i$, when an end event is invoked

From the observation above, it follows that:

$$M(t_s, t_{read}) = M_n$$

where n describes the index number of pair detected events so far and satisfies the following equation

$$n = \sup\{i|t_s \leq t_i^b, t_i^e \leq t_{read}\}$$

If the frequency of the update interval of the visualization component is higher than the event rate, then we may get an inaccurate result value. The reason for such kind of problems is that the probability to read results between two pair of

events will be higher. A possible solution for this kind of problem can be achieved by returning the values when measurement results are acquired between the two paired events. This is performed by providing result values between t_s and t_i^e and t_i^b and t_{read}. If the end event of a certain measurement is not detected, then the value M will be adjusted to its accurate value as shown below.

$$M(t_s, t_{read}) = M_{cur} + V(t_{read}) - V(t_{n-1}^b)$$

where

$$M_{cur} = \sum_{i|t_s+\triangle d/2 \leq t_i^b, t_i^e < t_{read}+\triangle d/2 \wedge 0 \leq i < n-1} (V(t_i^e) - (V(t_i^b))$$

where n represents the number of start and end event pairs as before.

Consequently, if a user starts a measurement while the execution of the application is just between a begin and an end event. The measurement result between the start of the measurement and the first upcoming end event must also be taken into account. In this case, if the algorithm described above is used without any modification, reading the results after the t_i^e would be $V(t_i^e)$ and this includes the results between t_i^b and t_s which is not correct. Therefore, the equation must be adjusted so that the result will be reduced by the value computed between t_i^b and t_s as shown below:

$$M(t_s, t_{read}) = V(t_0^e) - V(t_s) + M_{cur}$$

where

$$M_{cur} = \sum_{i|t_s+\triangle d/2 \leq t_i^b, t_i^e < t_{read}+\triangle d/2 \wedge 0 < i \leq n-1} (V(t_i^e) - (V(t_i^b))$$

If both of the above cases are happening at the beginning and at the end of certain measurements, the following modified equation will provide the correct result value:

$$M(t_s, t_{read}) = V(t_0^e) - V(t_s) + M_{cur} + V(t_{read}) - V(t_{n-1}^b)$$

where

$$M_{cur} = \sum_{i|t_s \leq t_i^b, t_i^e \leq t_{read} \wedge 0 < i < n-1} (V(t_i^e) - (V(t_i^b))$$

The algorithm for this equation can be obtained by extending the algorithm above with the initial value $H_0 = V(t_s)$. If the first event is a begin event then H_0 will be overwritten with $V(t_i^e)$ otherwise if the next event is an end event the value between the start of the measurement and this end event will be added to the result values.

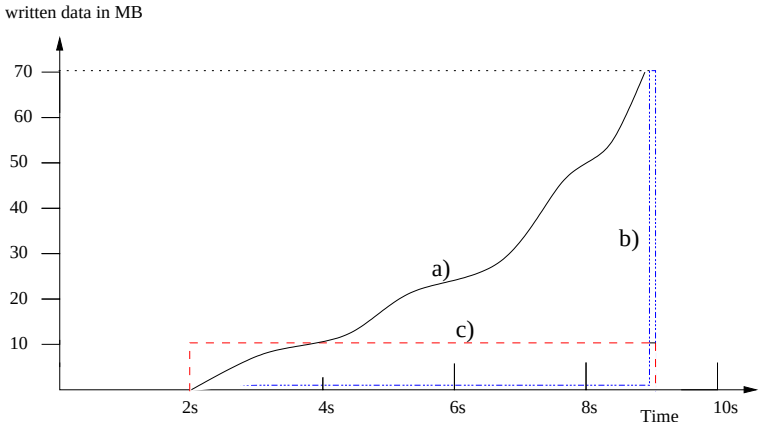

Figure 3.5: Writing data to a disc.

3.2.3 Difficult Measurement Issues

Providing result values between a begin and the corresponding end event can only be possible if the measurement provides the desired data at the time of reading the measurements. This enables to provide a more accurate result data. Unfortunately, this is not possible for all kinds of measurements. There are measurements which cannot provide interim result at an arbitrary point in time. For example the operating system function ***write()*** provides results only at the end of the writing process. If the writing process is not accomplished, reading the result values will be impossible.

Let us assume that writing 900MB of data to a disc takes 6 seconds and the update interval of the visualization component is one second. In this case, if the visualization component wants to access the results between the first and the 6th second, the result value returned will be 0. Providing the amount of data written to the disc before the writing process is accomplished is not supported by the OS and thus requires to go deep into the operating system kernel which may be time-consuming and may adulterate the final results. The problem is also that after 7^{th} second the visualization component will get information that 900MB are written to the disc in 1 second which is obviously not true.

In such a case, a better solution is to return no result at all until the writing process is finished, instead of returning a wrong result. At the end of the writing process, a summary result value can be returned which may provide the average result value as shown in Fig. 3.5c. The summary result, in this case, is 900MB in 6 seconds which is 150 MB/s.

Another problem involved in such measurements is concerning about the visualization of those results. For the curve diagram, the summary value, as described in Fig. 3.5c, will be shown with a delay of the writing process time, i.e., with a delay of 6 seconds. Unfortunately, there is no way to present these results using

bar graphs, since bar graphs are only used to show the current result values of the online measurements.

3.3 Developing Augmented Dataflow Graph

3.3.1 Introduction

In order to achieve a manageable parallel tools to be used in a Grid environment, the underlying design of most of the parallel tools which are used for multi clusters and super computers must support hundreds of thousands of processes. If this is not the case, the design must be adapted to the new challenging environment in order to profit from the variety of resources provided on the Grid.

In this section an approach based on a distributed evaluation of data using an augmented dataflow model will be presented which provides an efficient way of computation in a distributed way. This includes, in general, the distribution of the tasks, collecting, and processing the desired result values. In order to have an efficient execution model, the dataflow graph must be application specific since the flow of data depends on the desired measurement scenario. Even for the same application, the flow of data can be different if the applicable objects used are not the same. This and other issues enforce the development of a new and modified dataflow model, which also support the dynamical creation of measurement scenario specific dataflow graphs.

Since the augmented dataflow graph used for the distributed evaluation of performance data is based on the classical dataflow model, the next section describes the formal specification of the dataflow computing model. Following that, it will be shown how the classical dataflow model is modified to suite the requirement needed to build an efficient multicast/reduction overlay network using the Augmented Dataflow Model (ADFM).

3.3.2 Dataflow Model (DFM)

Several ideas of dataflow models are developed and different software are also implemented using the dataflow model as discussed in section 4.4. By the implementation of a software, a dataflow graph can be used to generate intermediate representations, specially for data processing. Using this model, the execution sequence of certain expression can be altered without affecting the function computed, as a partially ordered set of expressions. This increases the total throughput and decreases the execution time.

As the underlying mathematical model, the Dataflow Model (DFM) prescribes the essential data dependencies. To achieve a better utilization of the processing elements and to create a more balanced load between front-end and back-ends with master/slave architecture, the ADFG is developed which is based on the classical Dataflow Graph (DFG) [35, 36]. The DFG technique has proven to provide a

powerful computation model since its fundamental properties are to support parallelism automatically, and to enable the execution of scalable, high performance computations on continuously streaming data.

In addition to the behavior mentioned above, composability, decomposability and functionality are also the exciting behavior of the DFM to be strongly considered in the distributed programming.

The Classical Dataflow Graphs

Despite the fact that there are various implementations for the DFM, all dataflow graph schemes are based on a number of common concepts. A DFG consists of nodes (actors) and arcs representing data dependencies (the node of the DFG will be called Dataflow Node (DFN) hereafter). While the composability behavior of the dataflow graph enables combining different sub-DFGs to form a new single DFG, its decomposition behavior enables to create multiple different sub-DFGs from a single DFG. The functional behavior of a DFG depends on the firing rules possessed by its nodes.

A DFN, in general, has a finite number of inputs and a finite number of outputs, and is describes by the mathematical function it consists. The outputs are the result of absorbing the input tokens and applying a firing rule on them. A firing rule is a condition under which the actor may compute the input tokens and generates output token. Thus, the evaluation of a dataflow graph is equivalent to the evaluation of the firing rules it consists. The DFM is also self-scheduling in that instruction sequencing is constrained only by data dependency and flow of control is encapsulated with the flow of data.

The precondition for the firing rule is the availability of all input tokens which are necessary to compute the function describing the firing rule. This uniform firing rule comes from the classical approach. As it will be described for ADFG, this can be modified to suite the underlying algorithm. The simplest mechanism is that the DFN fires when its operands are made available by the children DFNs and the results are passed to the parent DFNs without a control whether the token produced before is consumed or not.

Formally, a DFG is an abstract structure represented by Directed Acyclic Graph (DAG). Such a graph consists of actors as operator that fulfils a partial order relation.

Definition: A directed graph [32] $G = (V, E)$ consists of a set of vertices

$V = \{n_1$ **,...,** $n_k\}$ and a set of edges $E = \begin{pmatrix} V \\ 2 \end{pmatrix}$ of pairs $\{(v, w) : v, w \in V\}$

Definition: In a graph $G = (V, E)$ a path p is a sequence of vertices $p = \{v_1, ..., v_k : v_i \in V, with\, i \in [1,k]\}$ such that $\{(v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k)\} \subseteq E$

Let $p(G)$ be the set of all paths in graph G.

Definition: A cycle is a path $p = \{v_1, ..., v_k\}$ such that $k \geq 2 \wedge v_1 = v_k$

Definition: A DAG G is a directed graph with no cycles.

Definition: A dataflow graph

$$DFG = (V, E)$$

is a DAG used for the flow of data from the leafs to its root.

This classical DFG has drawbacks which avoids its direct usage in some software development. For the scalability reason, a buffer (which will be called as Data Provider (DP) hereafter) must be available between any two consecutive DFNs to store data which cannot be consumed immediately. This DP should have an unlimited length to overcome the scalability requirement of the augmented new model. In order to build an efficient DFG, this must be extended to support the desired feature.

Beside this, it must be possible to have a bi-directional communication between any DFN and DP to enable the flow of control data in a top-down manner as well as to support a flow of result data in a bottom up fashion. Those drawbacks are the basic requirements arises from the need of having a more efficient, scalable, flexible, dynamic and powerful model which should result in a high throughput to distribute control information and to collect computed date in tools which operates in an online manner. In addition to this, the new ADFM supports a suitable dynamic dataflow splicing mechanism which simplify distribution of the subtasks represented by sub-DFGs. Thus, it is possible to create and destroy all or part of the dataflow graph dynamically.

In order to synchronize the results using a pull or push mode [7], it is also necessary to have a bi-directional communication between the DPs and the DFNs. Whether all the input token are consumed by the DFNs and/or a result goes to all the output arcs, should also be decided as a result of computing the firing rule of the specified DFN. In order to fulfill this and other requirements an ADFG will be presented in the next subsection.

3.3.3 Augmented Dataflow Graph

This subsection describes the modified DFM which fulfill the necessary requirements to build an efficient broadcast/reduction overlay network. The first major difference between this and the classical DFG approach is that this model provide a first-in, first-out queue as DP between any two consecutive DFNs which makes it scalable. This DP may generally be used to control the flow of data (which will be called result token hereafter). The DFN which needs to write the data in to the DP must not wait untill the data written into this DP are consumed. At the same time, the DFN which is going to read those data must not also read those result token immediately to make the DP ready for the next activities. That means that all DFN

can read and write the result token on demand, which is also helpful to synchronize the data belonging together. Therefore the ADFG is be defined as follows:

Definition: An ADFG

$$ADFG = (DFN, DP)$$

is DAG where its edges are DPs.

To provide a single result of the ADFG, the root of an ADFG is always a DP. All the leaves are DFNs which provide the input data, and every DFN can have multiple child DPs and also (except for the root DFN) multiple parent DPs whereas every DP can have only a parent and a child DFN as shown in Fig. 3.6. The different properties of these DFNs and DPs are described as follows:

Dataflow Nodes (DFNs) and Data Providers (DPs)

An ADFG consists of a finite number of DFNs and DPs. The DFN processes a stream of input data provided by its child DPs. After processing these values the DFN produces an output result token only to those DPs which are specified to consume the computed result value. To simplify the formal description of the ADFG, the i^{th} DFN of the ADFG is represented by DFN_i where i is a unique identifier and the root DFN is represented by DFN_0. Since we are interested with backwards as well as forwards flow of information, we write

$$DFN_i \rightarrow DFN_j$$

to represent the flow of data from the i^{th} DFN to the j^{th} DFN in a top-down and

$$DFN_j \leftarrow DFN_i$$

in bottom-up manner.

A single DP is represented by DP_j where j, analogous to the DFN, represents a unique identifier and DP_0 represents the DP at the root DFN. The DFN which is a destination of a DP is called parent DFN of that DP and is represented by $P_{dfn}(DP_i)$ whereas that which is source is called child DFN and represented by $C_{dfn}(DP_i)$.

Similarly,

$$C^i_{dfn}(DFN)\, and\, P^i_{dfn}(DFN)$$

represents the i^{th} child and i^{th} parent DFN of a DFN. A single DFN can have none or multiple child DPs, but must have at least one parent DP. Analogous to the DFNs, the i^{th} child and the i^{th} parent DPs of a DFN are represented with

$$C^i_{dp}(DFN)\, and\, P^i_{dp}(DFN)$$

respectively. In both cases above if i is omitted, it means that the whole parent and child DFNs and DPs.

The number of those parent and child DFNs and DPs are given by

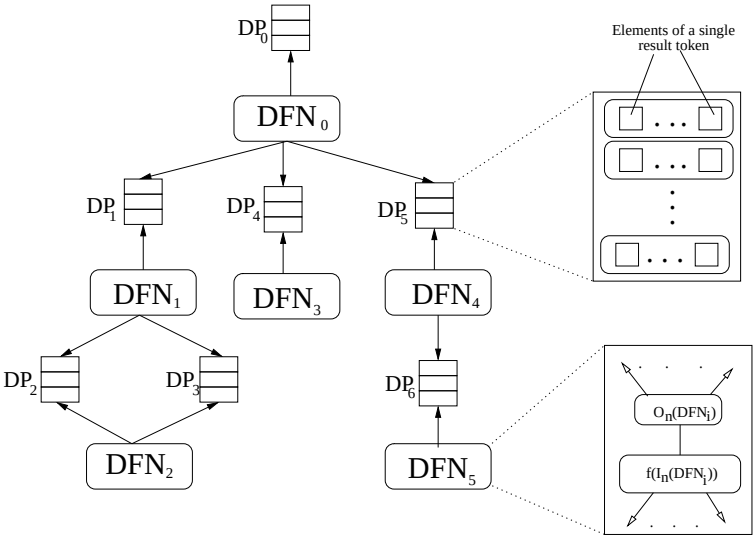

Figure 3.6: A simple example of an ADFG.

$$N(C_{dfn}(DFN)), N(P_{dfn}(DFN)), N(C_{dp}(DFN)) \, and \, N(P_{dp}(DFN))$$

respectively.
The i^{th} child and parent DFN of a DFN are assigned as follows:

$$C_{dfn}(DFN_i), \, and \, P_{dfn}(DFN_i)$$

Since a DFN can be used to merge or sink the input DPs, the number of the input and the output DPs is usually not equal.

An input for a DFN is then described by a $\{DFN, DP\}$ set. $I_n(DFN_i, DP_j)$ is then the n^{th} input of the i^{th} DFN on its j^{th} input DP and $I_n(DFN_i)$ represents the set of all n^{th} input of all child DPs of the i^{th} DFN. Analogous to this, the output of the DFNs are given by

$$O_n(DFN_i, DP_j) \, and \, O_n(DFN_i)$$

respectively.

In order to avoid having the same DFN, which has the same parent DFN multiple times in a single DFG, the ADFG supports also multiple DPs between two consecutive DFNs as used by DFN_1 and DFN_2 as shown in Fig. 3.6. This actually results in a complex dataflow graph which allows defining the underlying functional elements of the DFN only once. At the same time, this allows to have an access to different results of the same DFN at different time which avoids accessing multiple results from the same DP and so simplify the computation.

DFN Functionality

After the above formal description, the functionality *(f)* of ADFG can be given as a mathematical function that consumes a set of input result tokens from all child DPs of a DFN and produce a single output result token to be forwarded to parent DPs.

$$f(I_n(DFN_i)) = O_n(DFN_i)$$

Since this function may result in a result token which may not be forwarded to all $P_{dp}(DFN_i)$, conditional forwarding can be enabled by using additional constraints like a requester identifier to be used by the firing rule. Clearly, outputs of child DFNs are inputs of the corresponding parent DFN of the DP in focus as shown below. In general, the n^{th} output of a DFN is not the n^{th} input of its parent DFN, because of the possible conditional forwarding of the result tokens.

$$\{O_n(DFN_l, DP_p), \ldots, O_n(DFN_l, DP_q), \ldots, O_o(DFN_m, DP_r), \ldots, \backslash$$

$$O_o(DFN_m, DP_s)\}$$
$$= I_k(DFN_u)$$

where

$$DFN_u = P_{dfn}(DFN_l) = \ldots = P_{dfn}(DFN_m)$$

and

$$[p, q] \subseteq P_{dp}(DFN_l)$$

and

$$[r, s] \subseteq P_{dp}(DFN_m)$$

Since every DP can contain m number of result tokens R, the i^{th} result of the j^{th} DP is given by $R_i(DP_j)$ and the numbers of results in a DP is given as $N(R_i(DP_j)$.

Composition and Decomposition

The ADFG supports also the composition of a set of sub-DFGs (SDFG) in to a single DFG and decomposition of a single DFG into a set of SDFG. A SDFG is a part of DFG which has at least a DP and a child DFN. In case of composition, in general, an ADFG is generated by successively inserting a SDFG to it. In order to differentiate the created ADFG from the SDFG, the former is assigned as a main DFG. SDFGs can also be built-up from a set of sub-DFGs, which are called SSDFGs.

Therefore, an ADFG can be described in terms of the possible SDFGs it contains:

$$ADFG = \{SDFG_1, SDFG_2, \ldots, SDFG_n\}$$

A SDFG can be given as

$$SDFG_i = \{SSDFG_1, SSDFG_2, \ldots, SSDFG_m\}, with\, i \in [1, n]$$

and $SDFG(DFN_i)$ represents the sub-DFGs rooted at the DFN_i.

Therefore, ADFG consists a set of SSDFGs. These SSDFG can be disjunct in such a way that there is no single SDFG which may represent them. Thus, those SDFG are allocated using the location where they reside. In spite of this fact, a virtual link will be created in that host between the main SDFG and those SSDFGs which are disjunct. This virtual link is used to have an overall control over all the disjunct SSDFGs.

Location (Host) of the SDFGs

The location of the SDFGs is given by $H(SDFG)$. The value of this is a monotonically increasing integer number and $H(SDFG) = 0$ shows always that the SDFG resides at the location where the main DFG resides. In addition, different SDFG can have the same location. That means if

$$H(SDFG_i) = H(SDFG_j) \neq 0$$

then both graphs are residing at the same remote host.

In order to avoid a frequent request of a DP by the corresponding parent DFN whether data is available or not, a notification from the DP to the DFN is necessary. For the synchronization purpose, the DFN must also be able to request the DP directly whether a data is available or not, without receiving a notification. This results in a bidirectional connection for the control flow between the DFNs and the corresponding child DPs as shown in the Fig. 3.7.

In addition to those bidirectional connections which are used only for control flow between DFNs and their DPs, there is always a single connection used for the flow of result tokens between them. In fact, these connections can also be used by DFNs to identify the type and location of the DFNs which are behind the DPs. For example DFN *B* in Fig. 3.7 can identify the DFN *A* by using the control flow connection "*g*" between DFN *B* and the DP and "*d*" between the DP and the DFN *A*. In a similar way, the DFN *A* can identify the DFN *B*. The four control flow connection, as depicted in Fig. 3.7, will be described as follows:

- the arc "*c*" and "*g*" are intended to be used by the DFN A and DFN B for a request purpose (e.g. to request the availability of data values in the DP by the arc "*c*").
- the arc "*d*" and "*h*" are supposed to be used for notification purposes (e.g. to inform DFN A that a token is available in the DP) and to identify a child DFN, respectively.
- the arc "*e*" and "*f*" are used to fetch a result token from the DP and to write a data token to the DP, respectively.

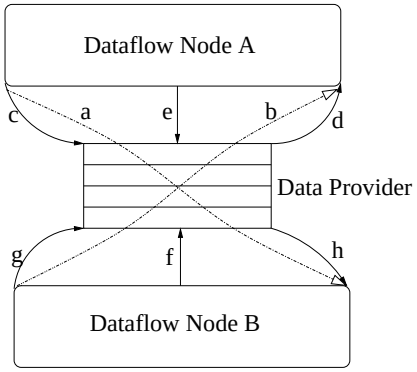

Figure 3.7: Bi-directional connection between a DFN and a DP.

The ADFG supports event oriented as well as request oriented flow of data which is also known as the push and the pull model, respectively. For the purpose of clarity, the control flow connections (like "*c*", "*d*", "*g*" and "*h*"), between the DP and the DFNs will be mostly omitted in the DFG and only the connection (like "*e*" and "*f*") for the flow of data will be used as an arc between any consecutive DFNs.

3.3.4 The Push and The Pull Models

In order to realize an automatic flow of result token from the leaves of the ADFG to its root DP and also to enable request oriented flow of data, this design supports also, as mentioned above, the pull as well as the push model.

The pull model enables a flow of data initiated by the root DFN in a top-down manner. This means the leaves of the DFNs of the ADFG are requested to provide result tokens to the consumer of those data (e.g., the UI used to visualized the results in case of performance analysis tools can be considered as the consumer of the final result tokens). In order to provide an efficient implementation, callback function can be used to deal with result tokens which are provided asynchronously since they cannot be accessed immediately. The push mode, as a bottom-up approach, is initialized by the events triggered to provide result token to the leaves of the ADFG. This means that whenever an event occurs, the triggered consumer DFN will get the computed result tokens.

In both cases, the received result tokens will be written automatically to the requester DPs using the communication assigned by row "*f*" in Fig. 3.7. The DP in turn notifies its parent DFN to access that result value using the communication represented by arc "*c*" and the parent DFN reads the result tokens using the communication assigned by arc "*e*". Since every DP notifies its parent DFN as soon as it gets the first result token and the parent DFN tries to read all the available result tokens from all its child DP, it is guaranteed that the flow of continuous data is automated.

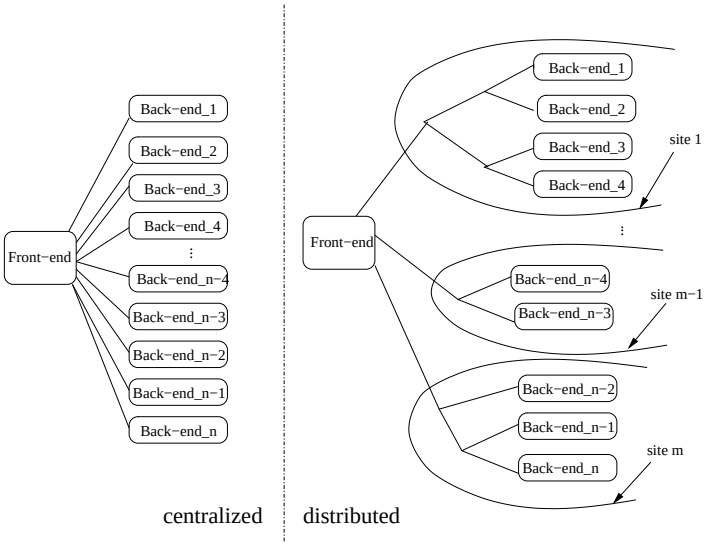

Figure 3.8: Centralized versus distributed architecture

In order to collect result tokens belonging together and evaluate them at any level of the DFG, those result tokens will be synchronized according to a specified constraint in the DFN. This method allows controlling the consistency of the data at every level of the DFG through checking the availability of the necessary result tokens which will be used by the firing rule of the DFNs. This means that the DFN will wait until all the results, which belong together, are available.

In this way, an automated parallel execution model is developed which is highly scalable and also able to cope with stream of time-stamped result tokens.

3.4 Distributed versus Centralized Evaluation

In this subsection we will discuss the differences between the centralized and distributed evaluations. Both methods are dealing with front-end and back-end components of parallel and distributed computing. The front-end is usually used to broadcast control information and collect the computed result values, whereas the back-ends compute the result values and send them to the front-end, as shown in Fig. 3.8. In other words, the back-ends are the input data producers, and the front-end is the consumer of the computation output. Such increase in number of back-ends has driven the desire of a distributed evaluation to make the evaluation manageable by the front-end and reduce communication overhead.

Specially, for the optimal computation in an environment like the Grid [5], where intersite communication must be reduced, the additional tools used with the application running on the Grid should not have an additional communication overhead which will reduce the performance of the whole Grid environment. In

general, an intensive communication between different processes residing specially on different sites will reduce the advantage gained through grid computing.

3.4.1 Centralized Evaluation

A centralized evaluation as used by most of the tools in a distributed computing environment works well in a small-scale environment. But, when systems and application get larger, those tools would have significant performance problems. The reason is that the front-ends of those tools would not be able to process quickly enough because of the huge amount of data sent to them.

The amount of data to be sent to the front-end increases immensely when the number of back-ends increases and/or when they produce measurement results more frequently. For instance, the front-end must be able to manage event based measurements which may produce hundreds of event based result values in a second resulting in intensive network communication making the front-end hardly manageable.

Most of the centralized components send measurement results to the front-end to evaluate there without leveraging the possibility to evaluate those results locally. As far as the number of result values sent to the front-end is not so big in a given time interval, this method will work well. Centralized evaluation may even be helpful if all the detail information is needed at the centralized component, for example, for the purpose of statistic. Centralized evaluation may not scale not only when the number of back-ends increases but also when the number of communication between a single back-end and the front-end increases which is application specific. To use a centralized evaluation, for example, for the performance analysis of interactive applications on the Grid will not scale since this must be done in an online fashion for the application which will run for a long period of time.

3.4.2 Distributed Evaluation

A performance bottleneck, which is the result of centralized evaluation, can be significantly reduced when the computation of the measurement results are computed as local as possible, and only the end results are sent back to the front-end. As local as possible means that measurements which can be computed at a single remote host must also be evaluated on the corresponding process in that host. And only the possible aggregated result value will be transferred to the front-end as shown in the left side of Fig. 3.8. Even at the front-end, it is possible to have another aggregation operations before the result values are sent to their end consumer.

The distributed evaluation presented in this thesis is briefly discussed in chapt. 6, and it is applied on a performance measurement analysis tool. This is achieved by introducing an enhanced dataflow model for an efficient evaluation of distributed data in an online fashion.

Chapter 4

Related Works

4.1 Introduction

Achieving scalable performance analysis of both infrastructure and applications related behavior in a distributed environment has always been a main concern and motivation for many research communities. This scalability issue demands a continuous effort for innovation a distributed evaluation. The increasing in number of processes of high-performance computing system from hundreds to hundreds of thousands has enforced the development of new programming models and infrastructure to cope with the new emerging demands.

There are a lot of efforts to provide efficient parallel tools which use different kinds of approaches to increase scalability. Those tools provide different methods to realize the distributed evaluation of data which are related to the solution presented in this thesis. Most of the solutions are motivated by the requirements of the different applications in focus. As a result, different tools are being developed for different kinds of applications. There are many solutions using software based collective communications infrastructure to support parallel tools and applications, and most of them are using tree based computation infrastructure which reduces the computational complexity from $\mathcal{O}(n)$ to $\mathcal{O}(log_2^n)$. However, most of the provided solutions are not suitable for scalable, interactive Grid applications. Even if most of the concepts and approaches are related to the distributed evaluation presented in this thesis, none of them are applying a dataflow mechanism, which is most efficient by supporting parallelism automatically. In addition, the provided solutions are based on different communication models, tool architecture and infrastructure and software engineering trade-offs than the one presented in this thesis.

At the same time, many tools are also trying to provide an efficient and flexible way of specifying metrics to describes the computation to be evaluated in a distributed way. In this section, therefore, first the approaches for distributed evaluation will be discussed which will be followed by the presentation of the available specification mechanisms. At the end of this chapter, the development of the dataflow model is discussed which shows the historical success of the dataflow

graph approach. Those approaches are the basic for the development of the augmented dataflow model presented in this thesis.

4.2 Methods of Distributed Evaluation

4.2.1 Paradyn/MRNet

Multicast Reduction NETwork (MRNet)[1] [13, 29] is part of an effort used to improve the scalability behavior of parallel performance and system administration tools and is used by the Paradyn performance analysis tool [54] developed in Computer Sciences Department University of Wisconsin. The Paradyn distributed evaluation tool contains components that belong to the front-end, the intermediate processes and the back-ends. It uses message multicast from the front-end to the back-ends, and data aggregation from the back-ends to the front-end using multiple data channels for logical streams of data.

Through incorporating a tree of processes as intermediate components between the tools front-end and back-ends, MRNet facilitates the distribution of tools' activities and the reduction of the computed data. Within the internal processes, filters (avg, sum, min, max and concat) are built-in to aggregate the data sent to the front-end. To extend the filter mechanism provided by the MRNet, a new filters must be provided by the user and will be loaded as required. The same is true for the configuration file consisting of the topology of the tree and the host assignment which must be available a priori to be used as a layout for the internal processes. The internal processes use balanced tree topologies to benefit from its regularity which makes the analysis process easier. Comparing the filtering mechanism of this tool with the one provided in this thesis, the filter used in MRNet are too complex to be extended by the user, whereas in our case, all filter functions can be specified in a very flexible way in the metrics specification and are automatically and efficiently implemented in the distribution evaluation process.

MRNet consists of two components: ***libmrnet*** and ***mrnet_comnode***. The former is a library linked to the front-end and back-ends and exports an API to enable the interaction of those two components using the network of internal processes. The latter is a distributed program which facilitates scalable communication and runs on intermediate nodes. Its instances are, therefore, the internal processes.

The back-end consists of communicators (as a container for the groups of end-points), which are created and managed by the front-end and provide a way to identify a set of end-points for point-to-point, multicast or broadcast communications (just like the communicators in MPI). For the flow of data, streams are used to connect the front-end to the end-points of a communicator. The MRNet network is instantiated by the network object. As shown in Fig. 4.1, after instantiating the MRNet network object at the beginning, a communicator object will be generated from it. A stream object, which will use the maximum floating point as a filter, will

[1] http://www.Paradyn.org/mrnet/release-1.1/UG.html

```
FrontEndMain(){
 float result; Packet *packet;
 Network * net = new Network(<confFile>, <exe>, <argv>);
 Communicator * comm = net->get_communicator();
 stream * stream=new Stream(comm,FMAX_FIL):
 stream->send(tag,"%d", FLOAT_MAX_INIT);
 stream->receive(&tag, &Packet);
 NetWork::unpack(Packet, "%f", result);
}
```

Figure 4.1: A simplified code for the front-end of MRNet.

```
BackEndMain(){
 Stream * stream;  Packet * packet;  int tag, val;
 Network * net=new Network();
 net->recv(&tag, &packet, &stream);
 NetWork::unpack(Packet, "%d", &val);
  if (val==FLOAT_MAX_INIT){
     stream-> send("%f", rand_float);
  }
}
```

Figure 4.2: A simplified code for the back-end of MRNet.

be then created and sent to the back-ends where the tag attribute specifies the nature of the message. At the end, a blocking receive function will be used to collect the computed data. The initialization of the corresponding back-end components is shown in Fig. 4.2.

The back-end sample code reciprocates the front-end actions. At the beginning, each back-end connects itself to the appropriate internal process. Instead of having stream based ***receive*** like the front-end, the back-end performs a stream anonymous ***receive***. It returns the tag sent by the front-end, the packet containing the actual data sent, and a stream object representing the stream established by the front-end. In this way, broadcast/reduction activities are realized. As shown in Fig. 4.1, a lot of important functionality must be fixed a priori which limits the flexibility of the tool. For instance, functional chaining is not possible since a single aggregation function must be determined for the given layout.

4.2.2 Lilith

Lilith [49] is a framework used to distribute user code across a heterogenous platform in an efficient way to improve the scalability of not only the distribution of control information but also the collection of the computed result values. The Lilith framework provides services to the user components deployed in it. Thus, using Lilith, the developer only needs to concentrate on the details of his code implementation to be executed in a distributed manner and Lilith takes all detailed tasks for

the propagation of the code to the destination nodes, and also for the communication among the nodes.

To achieve this goal, Lilith links recursively host objects on the adjacent nodes to generate a binary tree communication pattern. At the same time, the user code object called Lilim will be propagated to the hosts in the tree. Through this way, the development of a tool can be accelerated. Lilith handles synchronous wave messages by sending them to the root of its process tree. Written in Java, Lilith is platform independent and easy to use, but it does not support extensible data synchronization and flexible aggregation mechanisms.

4.2.3 Ganglia

As a widely used monitoring system, Ganglia [48] provides a scalable distributed monitoring system for multi-clusters and also for Grids. This system relies on a multicast-based listen/announce protocol to monitor states within clusters and a tree of point-to-point connection between the cluster nodes to aggregate their state. The main advantages of this approach are the automated discovery of nodes and the facility that every node knows the state of the cluster. This allows constructing the state of crashed nodes using the reliable information by polling any node in the cluster to obtain the entire cluster's state.

Even if its initial design was aimed to support only cluster monitoring, using the widely known technologies like XML for data representation, XDR for portable data transport, and Round Robin Database (RRD) tool for data storage and visualization enables the tool's interoperability behavior and integration to the new emerging distributed and loosely coupled systems. Providing the monitored data in an XML format, for example, enables Ganglia to be used in the Grid environment based on the widely used de-facto standard toolkit Globus by providing the monitored data to the Monitoring and Discovery Service (MDS4) of the toolkit as discussed in 2.2.4. MDS4 as well as other information services add query language and indexing to facilitate efficient information extraction using Ganglia.

To have an efficient evaluation of the monitored data, Ganglia defines a hierarchical design targeted at federations of clusters through building a tree like structure where aggregation at each point of the tree is done by polling child nodes at periodic intervals. Ganglia uses heartbeat message to identify the availability of nodes and provides not only system information, but also application-specific metrics, as used in the Network Weather Service [59].

4.2.4 Supermon

Supermon [50] is one of the tools which provide a scalable monitoring tool in heterogenous terascale cluster system. It allows to monitor characteristics of cluster behavior by supporting sample rates up to 66 000 per second.

The name Supermon is used as a data concentrator coming from a single node data server called ***mon***. ***Mon*** again is used to collect low level data from the

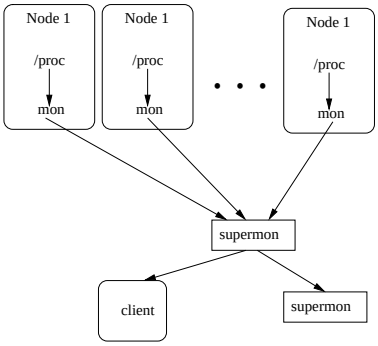

Figure 4.3: The architectural components of supermon

loadable kernel module. The Kernel, ***mon*** and ***supermon*** as used in the Network Weather Service together build the hierarchy in *Supermon* and use the same client-server protocol. To represent the data flowing at all its hierarchy levels, ***Supermon*** uses the so-called symbolic-expressions or s-expressions, as it is introduced in LISP programming language for the first time, in order to describe the self-defined version of data provided by operating systems like ***/proc*** file system. In addition, the s-expressions are architecture-independent and also in an ASCII format which increases the efficiency. Using the composability behavior of ***Supermon*** as shown in Fig.4.3. ***Supermon*** can act as a client and at the same time as a server by building a tree like structure. The ***mon*** server acts as an intermediate filter (using bitmax) between the ***/proc*** file system and ***TCP clients*** and parses the s-expression found in the dedicated files under ***/proc***. The ***supermon*** connects nodes running ***mon*** to provide information from a set of nodes. Another ***supermon*** can also be started to connect the created ***supermons*** and to build the hierarchy. This improves performance whenever the number of ***mon*** increases and/or the sampling rate is very high.

As a disadvantage of ***Supermon***, the tree structure is not determined automatically, but configured manually. In addition, there is no filter mechanism in the levels of ***Supermon*** which increase the amount of unnecessary data to be propagated to the client.

4.2.5 Periscope

Periscope [51] provides another approach to have an on-line distributed evaluation for automated performance analysis of application based on the Apart Specification Language (ASL) notation. Its main target systems are clusters of SMP nodes where the distributed analysis process is decomposed into entities called agents which are distributed over the parallel machines available for the computation. Those agents are arranged in a hierarchical way, providing a master agent at the top of

the hierarchy which is used as a connection point to the front-end. Beneath the master agent, there are high-level agents which perform aggregation functions on the node-level agents, located in the lowest level of the hierarchy.

The master agent disseminates commands coming from the front-end to the node-level agents. The node level agents on the other hand perform the actual performance analysis. Between all those hierarchy components, socket-based communication protocol is used to exchange messages. Building new performance properties and integrating them in to this tools is not as easy as in our case. In addition to this, user-defined event based measurement is not supported.

4.3 Methods of Performance Specifications

4.3.1 ASL and JavaPSL

The APART[2] [20, 40] Specification Language (ASL) is designed to formalize high-level specification of performance data and performance property to analyze MPI and OpenMP applications using an object-oriented specification model. The performance related data contained by ASL data model are static and dynamic data. Static data are gathered at compile time and include code regions, code version, source file and so on and the dynamic data are gathered at run time that include performance summaries of an experiment and of timing events. Load imbalance, communication and cache misses can be considered as an example of ASL performance properties.

In ASL, performance property represents specific performance behavior of an application which can be checked using certain conditions associated with a confidence value. A confidence value describes the existence of a performance property by giving a degree of confidence. Since not every performance property is a performance problem, a severity value determines the importance of a performance property in terms of its contribution to limiting the performance of the program. The severity value is determined by the user's threshhold to identify performance problems.

In order to give a generic representation for similar performance properties, a template is provided which can be used by substituting the template parameter with a concrete entity to instance the process. Meta-properties, on the other hand, support the specification of a high-level performance property using previously defined performance properties.

An example specification of a performance property based on a property template is shown in Fig. 4.4. Based on this property template, a meta-property can be defined for all properties, which specifies that a property holds for all processes in the system and the conditions and severity are also applied on all the processes.

[2] Automated Performance Analysis: Real tools

```
PROPERTY TEMPLATE CostPerProcess <float CostFunc(MPISummary)>
(Region r, Experiment e, Process p, Region RankBasis){
  LET  cost = costFunc(summery(r,e,p)) IN
  CONDITION: cost >0;
  CONFIDENCE: 1;
  SEVERITY: cost/duration(RankBasis,e);
}
float SyncCostFunc(MPISummary rs)=rs.SyncTime;
float CommCostFunc(MPISummary rs)=rs.CommTime;
PROPERTY CostPerProcess <SyncCostFunc> SyncCostPerProcess;
PROPERTY CommPerProcess <CommCostFunc> CommCostPerProcess;
```

Figure 4.4: Describing performance property using template in ASL.

JavaPSL[3] *[21],* as a Java version of ASL, is a flexible API using the syntax and semantic of the Java programming language to describe the performance properties and experiment related data of applications by using ASL concept. It not only provides an easy way to define new performance properties without the need of understanding the storage format of the experiment data, but also allows building complex properties by grouping property instances. Using Java classes for performance summaries, information about a specific execution of a Code Region in a version's source code can be obtained. While an application is modeled as a set of versions, an experiment refers to an execution with a specific parameter. To perform performance analysis, every property must implement three methods which are used to describe severity, confidence, and the value describing whether a property holds,.

4.3.2 Paradyn/MDL

As one of the most known parallel performance analysis tool, the Paradyn performance analysis tools [54] introduces the so-called W^3 search model which describes performance behavior along the three dimension: performance problems which are expressed in terms of thresh-hold, program resource which include hardware and software resources, and time.

A performance analysis process is facilitated by using a dynamic instrumentation approach [23] generating a code dynamically and incrementally, which will be inserted or removed to the running application on demand. This instrumentation enables to insert predicates and primitives at the desired points in a program and will be done at procedure granularity, i.e., the created code will be inserted at the entry and exit points, and call sites of a procedure. During the instrumentation, code snippets are inserted not only to the procedure, but also to the message passing routines listed at the beginning of the metrics specification (like the ***pvm_send*** and ***pvm_recv*** in the Fig. 4.5).

[3] http://www.dps.uibk.ac.at/projects/aksum/JavaPSL.php

The code to be inserted is written in a dedicated specification language called Metrics Definition Language (MDL) [22] which includes not only simple control and data operations but also enables to instantiate and control real and virtual timers. In addition, performance data can be constrained to different program components which can also be combined using the ”and” operations. Those components include modules (as a collection of procedures), procedure, nodes, files and message channels. For example, to restrict a metrics to measure a message sent or received by a module, a module constraint can be combined with a message type.

```
list pvm_msg_func is procedure {
flavor pvm; items {”pvm_send”, “pvm_recv”};
}
constraint procedure /Code is counter {
  append preInsn $constraint[0].entry
     (*procedure=1*)
  prepend preInsn $constraint[0].return
     (*procedure=0*)
}
metric msgs {
  name ”messages”;  units opsPerSecond;
  aggregateOperator sum;  flavor { pvm };
  constraint modul; constraint procudure;
  constraint msgTag; base is counter {
    for each func in pvm_msg_func
    append preInsn func.entry constrained
   (*msgs++;*)
  }
}
```

Figure 4.5: An example metrics specification using MDL

The MDL specification consists of a part which describes where the generated code will be inserted, and a part which shows what code will be inserted. A fully specified metrics to count messages sent and received by message passing routines ***pvm_send*** and ***pvm_recv*** is shown in Fig. 4.5.

This example shows a MDL specification, which uses all procedure found in the files under ***/Code*** directory and inserts a code snippet at the entry and exit location of all the procedures to count the message sent or received by the message passing routines ***pvm_send*** and ***pvm_recv.*** In order to increment the number of messages sent and received, a code snippet (***msgs++***) will be inserted conditionally (only when the corresponding procedures are called) into the message passing routines ***pvm_send*** and ***pvm_recv.*** In order to achieve this goal, an executable file is processed and most of the desired information (size and address of the code and data segment) is extracted from the symbol table. For instance, the start address of the function obtained from the symbol table is used as an entry point information. The code to be inserted will be placed in to trampolines which are dynamically

allocated patch areas. To insert the code, the application process will be stopped and OS facilities like ptrace or /proc will be used.

4.3.3 EARL/Expert

Tracing based post-mortem performance analysis solution for C/C++ and Fortran application using MPI and/or openMP message passing protocol is also provided by EXPERT [19, 27] which is embedded in the ESPRIT[4] working group APART. The performance analysis is performed in three dimensions: class of performance behavior, position within the dynamic call tree and location like node or process. Each of those dimensions arranged in a hierarchy to make the analysis comfortable. By providing a user interface which shows the severity of the performance property using different colors, a user can identify the performance problems easily. One of the recent extensions of EXPERT to realize multi-expert analysis by performing algebraic operations of the performance results.

To perform trace analysis using EXPERT by mapping them to a higher level abstraction, which is used to identify complex compound events, a language called EARL [26] is used. The EPILOG (Event Processing, Investigating and LOGging)[5], as a binary event trace format plus a run-time library for generating event traces of MPI and OpenMP applications is used, on the other hand, to investigate the trace file and produces also an input for the well known off-line performance tool Vampire. In this context, Opari is used as an instrumentation tool for OpenMP applications on the source code level allows tracing events and linking them back to the source code. For the MPI, PMPI is used as a profiling interface, whereas TAU [53] is used for user-defined functions.

4.3.4 Paraver

As a visualizer and analyzer tool for a parallel event traces of MPI, OpenMP, and Java programs, PARAVER[6] gives an overview on the quantitative analysis of trace files in an off-line fashion. The data are collected through instrumenting the libraries, and the data analysis is performed by a distributed memory machine simulator called DIMEMAS [67], which allows to reconstruct the behavior of the application using the trace file provided. To create the trace file, any tool which respects the PARAVER trace file format can be used including the DIMEMAS itself. The features of paraver include: detailed quantitative analysis of program performance, fast analysis of very large traces, support for hyprid (MPI/OpenMPI) programming and building of derived metrics.

[4] http://cordis.europa.eu/esprit/home.html
[5] www.fz-juelich.de/zam/kojak/
[6] http://www.cepba.upc.es/paraver/

4.3.5 Pablo

Pablo [24] is mainly used for the gathering and visualization of execution statistics using multi-dimensional scatter plot arrays and the hierarchy of source code to view process based information using compile-time program transformation for both sequential and parallel programs. Pablo is an off-line tool and supports MPI and PVM programs developed by Department of Computer Science (DCS) at the University of Illinois. The recorded traces are stored in a format called Self Defining Data Format (SDDF) which can be extended with desired attributes. Those trace files can be stored not only as ASCII, but also in a binary format depending on whether interoperability or compactness is more essential. As a descendant tool, SvPablo facilitates language independent performance analysis and visualization by providing a single interface for both instrumentation and visualization purposes. Both tools lack the ability to show important performance behaviors of communication bottlenecks and are also exposed to handle a large amount of data.

4.3.6 KappaPI-2

KappaPI-2 [44] is a descendent of Knowledge-based Automatic Parallel Program Analyzer for Performance Improvement (KappaPI) and uses trace files (generated by a tracer based on DyninstAPI [23] for MPI and TapePVM for PVM applications) to detect performance bottlenecks. It uses the idle processing intervals, determine their causes by applying certain rules and relate the performance degradation to the application source code in order to provide some recommendation to the user. A set of performance knowledge is coded in the kernel of the KappaPI tool which limits the flexibility of the tool to define new performance bottlenecks. To avoid such limitations, performance specification in KappaPI-2 is based on the compound event concept of the ASL language by translating it into XML syntax. Using a trace file and performance knowledge representation as an input, KappaPI-2 tries to identify behavioral patterns in the trace file and perform the desired analysis to determine the real cause of the bottleneck.

4.3.7 Other Tools

One additional tool which is not discussed in the section is TAG, which also provides one of the interesting approaches. It also contains a tree based aggregation infrastructure providing a query language to be used for the specification of tasks to be executed on the nodes of the sensor network which also supports aggregation operation based on a request/response manner. Other parallel database tools are also using several algorithms to achieve an efficient data aggregation mechanism.

Tree based approaches are also used to improve the scalability of some interfaces, like MPI, which uses broadcast and reduction primitives. MPI implementations mainly use serialized point-to-point operations to implement those functionalities. MagPI [39] proposes optimized versions of those primitives for geographically distributed applications which run on environments like the Grid by

providing an algorithm supporting hierarchical interconnections of MPI collective operations. Another example in this field is the ACCT [71] which tune the underlying algorithm of the MPI collective communication primitives automatically.

4.4 Methods of Handling the Flow of Data Items

Dataflow model is a drastically different way of looking at computation. Due to its elegance and simplicity, the dataflow model is used in different parts of hardware and software technologies. It is used to design different multi-threaded computers based on asynchronous instruction scheduling, for the optimization of compiler implementations [35] , during the development of compilers to facilitate code optimization and in the distributed computing sector. For example, the state-of-the-art in the field of multi-threaded computing, signal processing and reconfigurable computers is illustrated in [58, 73].

In the hardware sector, the dataflow concept and von Neumann architecture are handled as orthogonal computing paradigms. Comparing it to the von Neumann architecture where the main focus is the program counter, dataflow model is a different approach to handle computation in the computer architecture design since the movement and scheduling of data have priorities. The DFM provides solutions for problems of von Neumann computer concerning memory latency and synchronisation overhead. Several dataflow architectures have been proposed by various research groups around the world. Those are mostly used to build faster and more powerful computers. The list of most popular architecture includes the dataflow multiprocessor, dataflow machine using tagged token [33] and the Manchester dataflow machine [34] .

The basic idea of dataflow model was developed by Jack Dennis [60] in the early of 1970s. Dataflow model represents the flow of computed results and control data through a dataflow graph. In general, those methods are described by a dataflow graph where nodes are representing computations and arcs are used to schedule stream of data and thus to describe the dependency relationships between the nodes. Using a firing rule, a DFN consumes its inputs and generates the corresponding outputs. In addition, processes described by the nodes can only communicate through channels and use their firing rules to compute data items and to alter the routing of the computed data items and thus enable the development of data driven controlled environment which can be used for applications producing a continues stream of data. The firing rule performed by the dataflow nodes can also be used to relay, duplicate, and merge data. Since many nodes maybe able to fire simultaneously, they may represent many asynchronous concurrent computations of events.

As described in the following subsections, there are many kinds of process networks which differ in their model of computation (FIFO or synchronous communication) or execution (blocking and non-blocking characteristics).

4.4.1 Kahn's Dataflow Network (KDN)

In the early of 1970s, Gilles Kahn[7] [46] laid the theoretical foundation for DFM which can be seen as the extension of the DFM proposed by Jack Dennis. Despite this fact, Kahn's model is not used widely since it is too flexible to develop efficient models and not flexible enough for a wide class of applications. Using this model, it is not also possible to achieve statical scheduling.

Dataflow Process Networks (DPN) are used as a model of computation mostly in industrial practice in signal processing software environments as it is used in electrical engineering and also as a basis for different programming language design. The idea of developing DPNs is first described by Kahn in 1974 as a particular case of his processes networks called Kahn Processes Networks (KPN). In this model a group of deterministic sequential processes are communicating through unbounded uni-directional FIFO channels to support infinite stream of data emerging mostly from the signal processing systems. The data items to be handled by DPN can be very huge. For example, a digital audio stream may contain over 44 000 samples per second per channel and run for hours. Such channels represent the causality of dataflow between processes which are communicating by sending messages. This characteristic makes the dataflow communication more predictable than the shared memory approach.

In Kahn's model, a part of input sequence of data items can be used to produce part of output data through a property called monotonicity where writing to a channel is non-blocking and destructive, and reading from a channel is blocking. In addition to that this model does not allow processes neither to test the existence of data items without consuming them nor to wait for data items coming from multiple channels at once which results in determinism which means that for certain sequence of input, there is only one possible sequence of output. This is a disadvantage of this model since responding to unpredictable sequences of events is impossible. Non-deterministic characteristics are, for example, supported by most of modern programming languages to deal with exceptional unexpected situations.

The dataflow semantic, as proposed by this model, fits for signal processing since the algorithms of signal processing are expressed as block diagrams, which make it easy to apply visual syntax to specify the graph representing the network. This, as a graphical dataflow programming method, is used as a fundamental basic tool in industrial practice in a signal processing development environment which allows also mostly textual specification of the dataflow process networks. Matlab[8] is a good example using such graphical dataflow programming environment.

4.4.2 Synchronous Dataflow (SDF) Networks

This model is based on the Kahn's model with some restrictions. The basic idea of this schedulable dataflow model, as proposed by Edward Lee and David Messer-

[7] http://www.inria.fr/actualites/2006/gilleskahn/gk_hommage.fr.html
[8] http://www.mathworks.de/

chmitt [43] in 1987, is that each process reads and writes a fixed number of tokens each time a firing rule is applied which results in a deterministic characteristic. In order to perform the firing rule, there must be at least as many tokens on the input channels as it can be consumed. This atomic characteristics supports that the firing sequence can be determined statically in which a set of balance equation relate firing rates according to the production and consumption of data items. The limitation of this model is that it doesn't allow dynamic configuration. In contrast to the asynchronous message passing where tasks do not have to wait until outputs are accepted, in this model all tokens are consumed at the same time. The scheduling is performed using a finite amount of memory and not by using infinite buffers.

4.4.3 Tagged-token model

In this model the tag of the token will be used to relate the firing sequences. A tag is a context identifier that specifies the activation to which the token belongs. An operator is ready to fire when a matched set of input tokens arrives for all its input ports that have the same tag. Every output token is tagged with the same tag that the corresponding input token was assigned. For instance, in the MIT[9] (Massachusetts Institute of Technology) tagged-token dataflow project, the tagged-token approach is used to provide a general-purpose high-performance parallel computing.

The tagged-token dataflow graphs are directly executed on the MIT Tagged-Token Dataflow Architecture (TTDA) which is a multiprocessor architecture. To generate the dataflow graph, a high-level language is used which is compiled to dynamic dataflow graphs. One of the advantages of such tagged-token approach as an execution model, as described by Arvind and Gostelov [37], is that it prevents deadlock by using delay as an initial tagged-token on a channel.

4.4.4 Component Based Design versus Dataflow Processes

Systems that are composed of different components are suitable to be modeled with the help of, e.g. process networks or Petri Nets. Components, as black-box entities that provide services behind their interface, can be defined as process of networks communicates through their input/output ports which can be represented by the channels used in the dataflow model where the components are seen as dataflow nodes. For the composition of such components, the so-called interface automata are used to specify mostly the characteristics of components. Such interface can be used as a base of a contract between architecture designer and component developer.

Interface automata, as a bridge between the architectural model and heterogenous processes, are used mainly to assure the two main properties of DPN by taking the environmental assumption into consideration. These properties are safety and deadlock freedom. Safety means that no unexpected reception of data takes place whereas deadlock emerges when no process can make any progress since processes

[9] http://www.csail.mit.edu/index.php

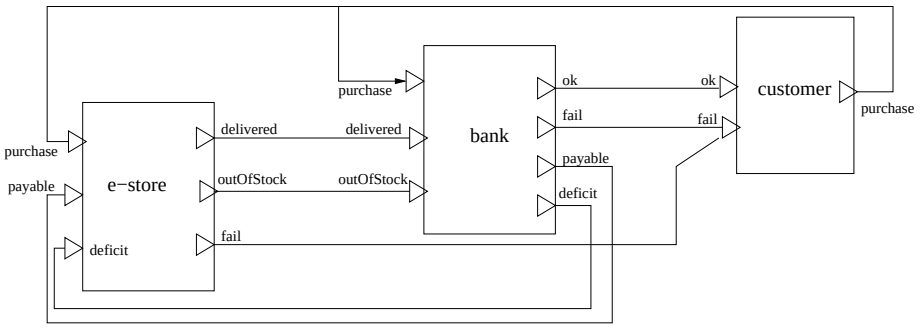

Figure 4.6: DPN used to purchase an online store

are blocking each other. This can be avoided by checking the consistency of components. Component based development and modular analysis of dataflow process networks are described in [56]. Component based approaches are used in different sectors of businesses. Fig. 4.6 shows a process used in a bank model which accepts purchase requests from customers and reports back whether the process is succeeded or fails [56].

4.5 Conclusion

In general, tree based computing networks have proven to provide scalability and efficiency to distribute control information and to collect measurement result values (Lilith, Supermon, Ganglia, etc.) through which the complexity of the analysis processes is reduced from linear to the logarithm of the number of processes in focus.

Most of the tree structures are used to organize the distributed processes in such a way that network communications can be reduced. In most of the tools, organizing the processes according to the site where they reside is not handled which is a very important level of abstraction for the Grid. Most of the tool uses configuration file for the layout used to determine the structure of a tree which requires expertise knowledge from the user, whereas our approach uses a dataflow graph which is related to the dataflow models discussed at the end of this chapter and generates its layout automatically. An efficient and flexible aggregation of the computed values is also very important to reduce the amount of data produced by the distributed processes back-ends and this issue is also not addressed in most of the tools.

Compared with all the related tools discussed above, our dataflow graph enables a very effective and efficient way of managing huge amounts of data produced by back-ends and provides a very flexible aggregation mechanism to deal with time-stamped computed result values in an efficient way. The way how metrics specifications are described is also well supported by using PMSL, which is highly configurable and user friendly specification language.

Most of the available tools, developing on-line or off-line analysis, provide very low-level information which is mostly communication or hardware events. Tools produced by the project Kojac, Paradyn, PERIDOT, SCALEA, APART and KappaPi, for example, are providing the user with a limited number of automatically created performance metrics. Only with such information, it is not possible to analyze application specific performance metrics and combining those metrics is usually complex. The possibility to support user defined metrics make our approach, therefore, very suitable to create more abstracted metrics measurements.

Chapter 5

Context of the Distributed Evaluation

5.1 Introduction

As the context of this thesis, a performance analysis tool and the underlying monitoring system are used together to perform on-line performance analysis in the Grid environment. When on-line performance analysis of an application is performed in the Grid environment, the underlying monitoring system plays a vital role not only by observing the behavior of the grid infrastructure and the application running, but also by facilitating the manipulation of the targets' runtime behavior as desired.

Therefore, in this section, we will discuss first the OMIS Compliant Monitoring Tool for the Grid (OCM-G) [11] used as the monitoring system, and then the Grid Performance Measurement tool (GPM) [1, 16]. GPM enables the user or developer to measure metrics describing performance behavior in an on-line fashion. That means that the performance measurement can be performed while the application is running. In order to specify the performance behaviors in focus, the Performance Metrics Specification Language (PMSL) [2, 10] is used, which will also be discussed, subsequently.

The on-line nature of the GPM tool enables the user to have a possibility to take part in the application steering process. On the other hand, the on-line nature of OCM-G allows the developer to have full control over the application. The combination of those two makes it possible to provide a powerful on-line performance analysis tool. For interactive grid applications which are supported, for example, in the ***CrossGrid*** [17] project and in its successor ***Interactive Grid project***[1], control information must be provided from the underlying monitoring infrastructure in an on-line fashion to facilitate the interactivity. When measuring the performance behavior of interactive applications, most researchers need answers in seconds, not hours. This performance analyses tool provides a very good opportunity to facili-

[1] http://www.interactive-grid.eu

tate their desire. In addition, it creates a most suitable environment to support grid infrastructure and application specific performance measurements.

The G-PM monitors the application by appropriate programming of the OCM-G since the initial processing of performance measurement results (e.g. partial aggregation of measured values) is performed by OCM-G. This approach significantly reduces the computational overhead associated with the monitoring, thus minimizing the influence of the G-PM on the grid application behavior. Furthermore, it reduces the overhead on the user workstation, which is important in case of monitoring large numbers of processes. The main advantage of using GPM, OCM-G and PMSL together is having the combination of on-line monitoring with the support for user-defined, application specific performance metrics.

There are a number of performance tools available that enable monitoring the performance of a distributed environment, although few of them deal with the performance analysis of a Grid application. Those tools either focus on hardware infrastructure or do not provide user-defined metrics whose functionality is comparable to G-PM. In addition, none of them support the performance analysis of an interactive application in a Grid environment.

Beside the tools mentioned in chapt. 4, the following tools are also related to the monitoring of performance behavior. The Globus Heartbeat monitor [65] which is used to monitor the state of processes in order to identify the status event exceptions, NetLogger [41] which enables real-time and post mortem performance analysis of applications and system-level data, and Network Weather Service [59] which enables to predict the performance of not only computational resources but also various networks.

5.2 OCM-G

Monitoring is the act of observing a system state via a set of sensors which provide values at a regular interval or only when they are requested to do so. These values can be used to compute desired behavior. In almost all monitoring systems, OS commands are used internally to get those desired values to obtain information about the machine in focus. For instance, a very simple method of monitoring a node in a cluster is to use the command ”ping”. A usual way of getting more interesting information is achieved by using information provided by the /proc file system and the ptrace system command. Most monitoring tools build GUI on top of such commands. Nevertheless, that information is rather low level and thus can’t represent high level information for the end user.

For the applications running on the grid in general and for interactive applications in particular, a monitoring environment is indispensable in order to detect bugs in the application code, find a bottleneck or to visualize the applications overall behavior. The tools using this monitoring system contain but are not limited to performance analyzers, debuggers, visualization tools and load balancers.

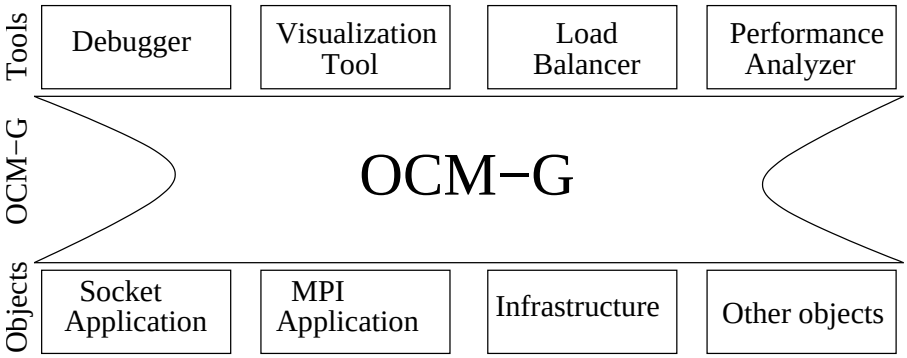

Figure 5.1: Layered structure of OCM-G environment

The OCM-G Monitoring System is designed as an autonomous infrastructure using the standard interface OMIS (Online Monitoring Interface Specification) and it facilitates a fast on-the-fly information delivery which is a main prerequisite for application steering on the Grid. As an intermediate component between parallel tools and the applications (as depicted in Fig. 5.1), the OCM-G provides services not only for collecting, but also for pre-processing information about the application at runtime. And these collected data can be used as input for the parallel tools used, and can be computed further and then visualized by the corresponding tools as desired. The bottom layer represents hereby the objects to be monitored, e.g., CPUs, Networks, application processes and grid infrastructure components.

This fully operational high performance system supports MPI and PVM applications running across multiple sites and can also be implemented to support other kind of applications. This system enables, for instance, the G-PM to get performance measurement data, like communication and synchronization delay, volume of communication and CPU cycle usage to mention a few.

This distributed monitoring system is intended to support the developers during the deployment of their application. This is most usefull especially for interactive applications. Therefore, this flexible, extensible, and interoperable system is used by the GPM, for instance, to show not only the progress of the application but also the activity of individual processes in order to figure out the possible bottlenecks. In addition to this functionality, this scalable system provides the observation of the communication patterns between different processes.

The unique characteristics of this user-installable system are a location transparent access to the monitored objects, supporting grid application running across multiple sites, guarantee extremely low overhead and high responsiveness, supporting interoperability of multiple tools monitoring the same application, the ability to invoke services on sets of objects, plus a fast data acquisition. Furthermore, this system can be programmed by the tools using monitoring requests comprising of information, manipulation and event services.

Even if the standardization of the interface between the applications and monitoring system is difficult due to the involvement of the operation system services, the interface between tools and the monitoring system is standardized with OMIS. This interface understands the target system as a hierarchy of applicable objects. These objects include processes, nodes and sites. From the OMIS point of view, those objects have a unique identification used as token. The OMIS interface decouple tool and monitoring system functionality and so provides abstraction and increases modularity.

This monitoring system provides the monitored data of the application using local buffering. Those data are used only on-demand. Using profiling, summarized information being stored in integrators can be accessed in an efficient manner. The information is gathered via selective run-time instrumentation that facilitates conditional executions and thus reduces the overhead through using the instrumented functions only when they are required for the actual measurement. This instrumentation of the applications is performed automatically on the run time and is based on binary wrapping through providing a pre-instrumented version of communication libraries. This instrumented libraries, in case of MPI, enables to evaluate application communication performance and parallelization-associated overhead. OCM-G contains also OCM-G tracer to record data related to some MPI functions which includes time stamp, process information and event specification parameters. Instead of providing a fixed set of predefined metrics, this system allows construction of service-driven metrics using lower level building blocks as desired.

The services provided by the OCM-G is useful not only to monitor the application as a whole or a particular execution of an application, but also to collect information about the grid infrastructure, such as network or grid site loads from which some of the grid services, such as resource allocation, job migration, resource monitoring and job brokers, are depending on.

5.2.1 Basic Concepts and Functionality of OMIS

In order to program this monitoring system by the tools, OMIS defines the following three types of services which are used intensively to realize the distributed evaluation handled by this thesis. Those services are:

- Information service
- Manipulation service
- Event services

The information service provides different information especially about the state of the objects, the manipulation service is used to operate on those objects by changing their states, for example. These kinds of services are actions which are performed without any preconditions to be fulfilled, and they result in an immediate action in the target system, as a response to the request.

```
: node_get_inf([],0)
: node_attach([n_1])
```

Figure 5.2: An example of unconditional requests

```
thread_has_started_lib_call([p_4], ”MPI_Send”):
pa_counter_local_increment(pa_lc_1, $par5)
```

Figure 5.3: An example of a conditional request

Examples of such kinds of services are depicted in Fig. 5.2. The first one lists all the information from all the nodes currently observed by the monitoring system, whereas the second one (as a manipulation service) results in the monitoring system attaching itself to the specified node ***n_1***. The first parameter in the former case shows that the request will be performed for all applicable objects whereas the second parameter of it indicates that all available information is requested. As a result of which, a long list containing all desired information about all the nodes in focus will be provided.

The event service, on the other hand, executes an information or manipulation services when a specified event occurs. This allows formulating Conditional Service Requests (CSR) which enables to trigger a list of actions to be executed, only when the corresponding events are detected. That means, when an event is detected, a chain of action can be executed which can contain one or more information and/or manipulations services. Whenever those events occur, a callback function for the triggered actions will be executed, asynchronously. An example of an event triggered request is shown in Fig. 5.3.

The first part of this request describes the event to be triggered, whereas the second part illustrates which action will be performed when that event occurs. That means, whenever the execution of the ***MPI_Send*** command is detected in a process ***p_4***, the counter ***pa_lc_1*** will be incremented with the sent volume so far. By combining those services, it is possible to perform non trivial monitoring requests which can include conditional, as well as unconditional requests. An unconditional service request can contain a list of information and/or manipulation services, which will be executed immediately, whereas conditional requests are composed of an event service and a list of actions. All the provided services can also be extended easily through defining new services which can be loaded dynamically at the run time.

The main interface procedure used between tools and monitoring system is shown in Fig. 5.4. This function is used to handle the co-operation for both conditional and unconditional requests. The parameters represent: the string representing conditional or unconditional requisites, a callback function which is activated whenever the specified action have been executed, immediate reply of the request

```
Omis_reply
omis_request(char * request,
             void (* callback (Omis_replay replay,
             void * param),
             void * param,
             Omis_flags flags);
```

Figure 5.4: An OMIS request function.

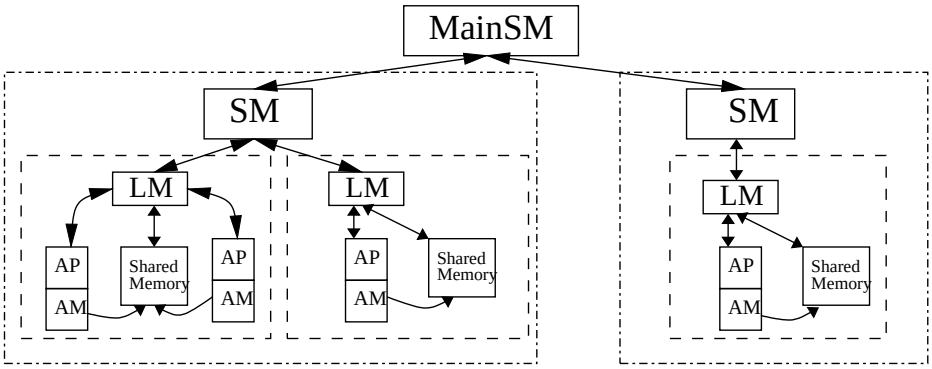

Figure 5.5: OCM-G components.

describing the status of the request, and flags which are used, for example, to select whether this call should be performed in a blocking or non blocking manner.

5.2.2 OCM-G Components

As shown in Fig 5.5, the decentralized system OCM-G consists of three main components: Main Service Manager (MainSM), Service Manager (SM) and Local Manager (LM). All three components use the OMIS protocol to communicate with each other.

The MainSM is the top-level component allowing tools to be connected to it and it is available per user. This component holds the necessary information about the whole application and is used to forward partial requests from the connected tool to the corresponding SMs as well as to collect all the results from the underlying SMs. It is also responsible to create and start the SMs. Since the grid-enabled and platform-independent start-up character of this component can be configured, this component can reside on the UI or any workstation depending on the permissions for incoming connections.

Using Globus, the Computing Elements (CE) of the remote host can be used to deploy the MainSM where specified ports can be used to avoid a firewall issues. Even if it seems to be a performance degradation to have this centralized component, the messages will not always be routed directly from the SMs to the MainSM, and also the localization information stored in MainSM can also be copied to other

SMs. This centralization is aimed to keep the localization information up-to-date. It is planed to support the discovery process by another component based on Grid information management system and to use the Grid job scheduler to get some more additional information.

The SM is available one per each site and is used to forward the partial requests coming from the MainSM to the LMs which also desire permission for an incoming connection. The LMs are yet available one per each host and are reside on the Worker Nodes (WN) where the application processes are running. Those LMs accept and execute monitoring requests for application processes resides in their own host. Those components of an application build together a Virtual Monitoring System (VMS).

As shown in Figure 5.5, there are additional components like Application Module (AM) which are libraries linked to the application in order to perform performance critical actions and to store performance data locally. Since the whole measurement depends on the integrator and counters at the operating system level, the computed result values are stored in the shared memory segment of every node which is mapped onto the virtual address space of the process within that node. This shared memory is accessible by the local monitor and application modules to avoid process switches which could be time consuming. Whenever an instrumented basic metrics is computed, those stored values are being updated.

In order to monitor an application, the user should start the MainSM, which returns a connection string composed of the host's IP address and the port number dedicated for incoming connections. After this, the application can be started on some WNs which then register in LMs and those LMs report those site names to the MainSM. The MainSM then starts SMs on those hosts to which the corresponding LMs register. At the end, the tool can be started and will be connected to the OCM-G.

This monitoring system uses RSA based encrypted connection and user authentication. For the communication between the components of this system, authentication using the Grid Security Infrastructure GSI is used, as discussed in section 2.2.4. In addition to this, a digital signature is used to avoid pretends and forged-component attacks.

Other tools in the same category are also providing related functionality. Those monitoring tools include: the Grid Analysis and Display System GrADS[2] [72], which focuses on an automatic performance tuning based on patterns and also supports System Level Agreement (SLA), GRM [57]/R-GMA [68], which define the monitoring architecture in terms of producer-consumer model and used to support batch processing in a DataGrid project, and Mercury [66], which uses performance prediction. In contrast to the OCM-G, those tools do not support interactive application in an on-line manner.

[2]http://www.hipersoft.rice.edu/grad

5.3 Grid Performance Measurement Tool (GPM)

5.3.1 Basic Concepts and Functionality of GPM

GPM is an application and grid infrastructure performance analysis tool for parallel grid applications. Its purpose is to monitor the run-time behavior of applications and is used to detect possible performance bottlenecks. It is designed to provide the user with performance data in an on-line fashion. As a result, it is not necessary to wait for the end of the application's execution in order to analyze its performance since the analysis is being performed constantly and the user can react to improper application behavior as it occurs. This is particularly important in case applications have a long execution time, which is quite common in the grid environment. The tool works in X-Windows environment and provides a convenient graphical interface to define measurement and to visualize performance data. One of the most important features of the G-PM is its flexibility. The tool can be customized to support a huge range of monitoring scenarios.

GPM plays a big role in providing the ability to perform performance analyses which are used to optimize the application and to improve the quality of the application code running in a Grid environment. One big challenge was to minimize the influence of such tools on the Grid application's behavior and reducing the overhead on the user workstation for large amount of processes.

In order to perform performance analysis in a dynamic Grid environment, the control of the application during its runtime is necessary. A common way of having low-level performance data, as it is used by most classical performance analysis tools, will only help the performance specialists. Those low-level measurements deal with the volume of send and receive date or I/O operations, delay of synchronization, memory and CPU usage, and many others.

Since the users of Grid applications are not all specialists, it is necessary to provide a way of having more abstract and application specific performance measurements. The G-PM provides a number of pre-defined performance metrics for MPI applications [30] and supports also both automatic and user-defined instrumentation. In order to support a user-defined metrics, PMSL has been developed, which enables the user to define new metrics out of pre-defined metrics and a user-defined instrumentation. Using this language and a manual insertion of functional calls, called probes, into the source code, the user is able to specify a high-level performance measurement which summarizes the information in a very suitable way. Those inserted probes do not define any measurement by themselves. They are only used to mark the region of interest, thus, the same probe can be used to define different metrics. The process of creating measurements from such specifications is briefly discussed in chapt. 6.

This tool is used to discover the performance bottlenecks of both sequential and parallel applications including MPICH-G2 (used by, e.g., flooding forecasting application) and MPICH-P4 (used by, e.g., whether forecast and air pollution simulations).

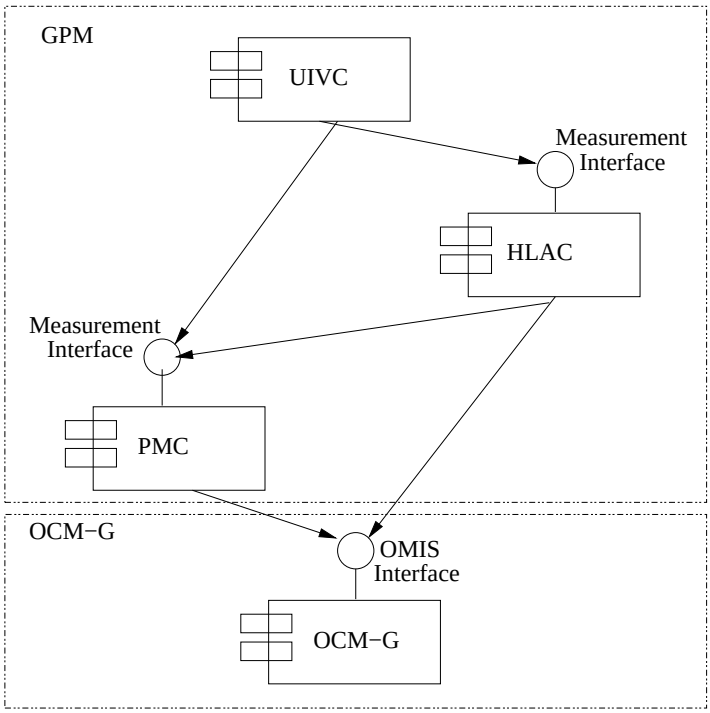

Figure 5.6: Module decomposition of the G-PM tool

5.3.2 GPM Components

As shown in the Fig. 5.6, GPM consists of three components: The High Level Analysis Component (HLAC), which deals with user defined metrics measurements, the Performance Measurement Component (PMC), which is used to create and manage low level metrics automatically, and the User Interface Visualization Component (UIVC), which is used to visualize the result of pre-defined as well as user-defined measurements.

The measurement interfaces used to define performance measurements, and also to read the performance related computed data, are also shown in Fig. 5.6. Those interfaces are provided for both HLAC and PMC components. In order to communicate with the monitoring system and to have an access on the monitoring data, the OMIS based OCM-G interface is also presented.

The Performance Measurement Component (PMC)

This component deals with all the built-in standard metrics, which provide the functionality for basic performance measurements of both Grid application and the Grid infrastructure. While the application-specific metrics deal with data transfer, resource utilization and delays, the infrastructure related metrics, provide information about the availability of resources, node load, and dynamic and static resource

information. This component communicates with the OCM-G monitoring service in order to have access to the raw data represented by the standard metrics. Those data are also used as an input to the HLAC and UIVC through the provided measurement interface.

The High-Level Analysis Component (HLAC)

This component is used to handle user defined measurements, which combine and correlate different built-in metrics measurements and user defined application instrumentation using probes. The functionality of HLAC is enhanced through an efficient implementation of probe-based metrics which can be used to measure complex measurements like the time used by one iteration of a solver, the response time of a specific request, convergence rates, etc. Finally, the results of the user defined metrics are presented in the same way as the built-in ones.

This component scans and parses the specifications written in PMSL, translates the specification of the built-in metrics into PMC requests, and handles the probe events, which can deliver application-specific information. To deal with measurements using probe events, HLAC communicates directly with the OCM-G. This component is vital to the concept of distributed evaluation of the measurements in this thesis.

The User Interface Visualization Component (UIVC)

This component is used to visualize the computed results of the low-level as well as the high-level measurement results by allowing the user to specify the quantities to be measured. It contains different graphical interfaces used to demonstrate the results. It provides a list of user-defined and built-in metrics, all available applicable objects and regions in the program code for which the performance measurement should be computed as depicted in Fig. 5.7. This enables to determine, for example, single or multiple origin and destination of a point to point communication and to restrict the measurements to the specified regions of code.

The main window of this GUI provides menus to manage the measurements by enabling the start and stop functionalities for the defined measurements. The visualization windows show performance data including a BarGraph, PieChart, Histogram, Matrix diagrams and value-versus-time function plots. A multi-graph bar can also be used to show multiple measurements at the same time so that a comparison between the same metrics with different parameters can be visualized easily.

One of the windows is also devoted for the specification of user defined metrics by providing a textarea. Loading a bulk of user defined metrics from a file is also supported which facilitates defining, for example, a group of platform dependent metrics. This allows also accompanying programming libraries with appropriate, library-specific performance metrics.

Built-in Metrics

Performance measurement using only a built-in metrics is performed automatically. Those metrics can be categorized in the following two ways: function-based metrics, which includes the instrumented functions of the MPI-library used and sampled metrics, which are not related to any function and thus can be measure at any point in time. The former types of metrics include delay, communication volume and invocation count of the ***MPI-Send***, ***MPI-BSend***, ***MPI_SSend***, ***MPI_RSend***, ***MPI_Recv***, and all other communication related MPI-functions. The sampled metrics consist of: ***Compute_time***, ***Node_load***, ***Time*** and ***Memory_size***. Both kinds of metrics are time-stamped so that a reasonable measurement result can be provided in spite of the possible delay of messages between nodes which could not be avoid even if the clock synchronization is performed for all available nodes.

Those built-in metrics are not high-level and application specific and therefore do not provide suitable measurements for complex applications running on the grid, but they can be used as build block for the high-level metrics since they can be combined and/or correlated together to build very useful metrics. By applying event detections defined by the user, yet another high-level metrics could be constructed which can be application or infrastructure specific.

In order to visualize the performance measurements of metrics, the specified measurement must be defined in a measurement definition window, and then the visualization type should be selected. After that, the user can observe the performance results as shown in Fig. 5.7.

Using Built-in Metrics

In order to investigate the reasons for the occurrence of performance problems, the following two built-in metrics in GPM can be used as an example. For the class of communication based metrics, the functional metrics ***MPI_Send*** can be taken into consideration. Defining and measuring those built-in metrics will be used to detect communication bottlenecks between the communication partners [2]. As an example, to visualize such behavior, the Matrix diagram component of the UIVC showing communication activities can be used. When all processes are executing the same algorithm, the metrics must show evenly distributed activities which will be described by the uniformly distributed colours of the Matrix diagram. If this is not the case, that is an indication of some irregularity in the ***MPI_Send*** function which might be the reason for the unbalanced load in the computation.

The second built-in metrics that can be used to show the load of the processes is the sampled metrics ***CPU_usage***. This can be visualized using a MultiCurve diagram showing the load of the processes. If the load decreases for some or all processes, performance degradation is detected. In order to find out which process is the reason for this performance problem, a histogram can be used to see how the ***MPI_Receive*** function is performing. If the result is not acceptable for all processes, a PieChart can be used to identify the exact process responsible for this

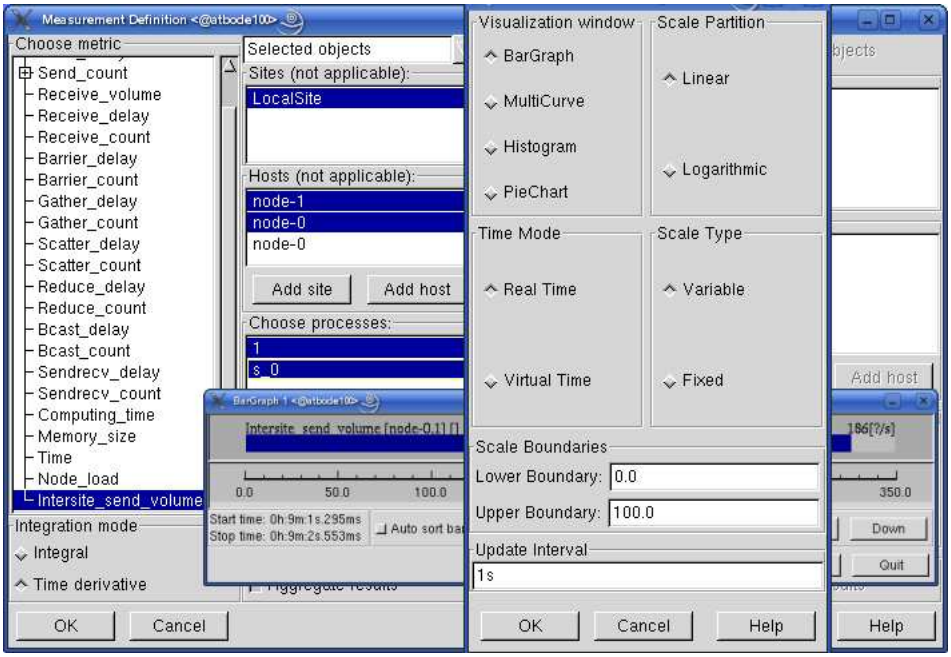

Figure 5.7: The GPM user interface.

imbalance. After running the application for a while, the PieChart shows for each process, whether an imbalance is detected in those ***MPI_Send*** and ***MPI_Receive*** functions.

Chapter 6

Overlay Networks for Distributed Evaluation

6.1 Introduction

The exploitation of distributed evaluation as a solution for scalability problems becomes essential for distributed tools like performance analysis, debuggers, and load levellers. This has been one of the main concerns of runtime tool developers in their effort to meet the ever-increasing demand of high-speed and high-throughput computation of on-line tools in the environments like the Grid.

The main concept of the distributed evaluation presented here is the evaluation of measurement results locally at the place where those results are produced, and to send the result values to the front-end where those measurement values may also be evaluated further or used directly by their consumer. Fig. 6.1 shows how, in general, distribution, aggregation and reassembling of results can be performed between nodes residing on different hosts that are controlled by different administrative domains, and the visualization component at the local machine. Specially, most execution models which have broadcast/reduction behavior can benefit from this approach to achieve an efficient high-speed and high-throughput computation processes.

In order to illustrate the distributed evaluation idea, a Grid Performance Measurement Tool [74] (GPM) as discussed in section 5.3, is used. In this tool, the user can perform performance measurements using the built-in metrics. This tool is extended to support user defined metrics using the Performance Measurement Specification Language (PMSL), which is developed for this purpose. Applying PMSL, the user can specify new metrics, which can be application and infrastructure specific. The user can also use this language to have an impact on the distributed evaluation of the measurement results by specifying the desired distribution. The High Level Analysis Component (HLAC), as one of the three components of GPM, handles all the user defined metrics and implements most of the solutions for the distributed evaluation.

For the distributed evaluation in GPM, its UIVC is used as a front-end, and the remote sites which are used for the computation of the user defined metrics are back-ends. The aim of the distributed evaluation is to compute the subtasks as local as possible, i.e., the computation must be performed at back-ends whenever possible. Especially, measurements which are event triggered should be evaluated at the back-ends as far as possible since the underlying events can occur very often and thus intensify the network communication. Through evaluating the measurements exactly at the location where the events occur and not in the centralized location, the unnecessary communication between the computing and the visualization components of the tool is avoided. This was, in most cases, the reason for bottlenecks at the front-end of the GPM tool. In other words, to increase the high-speed of a computation involved in a distributed computing, the frequency and the amount of data transferred from the back-ends to the front-end must be reduced.

The distributed evaluation presented in this work is realized using an Augmented Dataflow Model (ADFM) as briefly discussed in chapter 3.3. This supports not only the distributed evaluation by providing an automated parallel execution of the subtasks, but also the asynchronously reassembly of the measurement results. Through its asynchronous nature and because it does not impose any complete ordering on the elementary functions of a task, the DFM used for the distributed evaluation is suitable for the computation of measurement results which are derived at different time as they are triggered by different events. This is the case for GPM and other tools in the same category. This approach is also the best way to combine the measurement results to provide a final result value.

In order to realize the distributed evaluation, an ADFG is created from the measurement specification provided by the user. This ADFG is then used to build an efficient and flexible overlay network used for the cooperated computation of different sites, computer nodes and processors participating in the distributed evaluation. Under overlay network, in this context, is to understand that a transient network which overlie on the real network used for the communication between different computing components. This overlay network exists only, as will be explained latter, during the run-time of the corresponding measurement scenario. The behavior of this network facilitates destroying and creating those network communication on-demand. In other words, the overlay network is set up for every definition of the metrics specification and will be available as long as the measurement is not deleted. Fig. 6.1 shows two different overlay networks which are the results of defining a metrics for two different sets of applicable objects. Fig. 6.1a shows all the available sites and nodes whereas Fig. 6.1b shows only the sites and nodes chosen during the definition of the measurement in focus.

The main objective of building such an overlay network is to reduce the overhead due to token transfer through the communication network during the evaluation of the distributed computation. This can arise during broadcast as well as reduction operations. That means that this overlay network enables not only the flow of data from the back-ends to the front-end but also the flow of control data from the front-end to the desired back-ends. In contrast to the other approaches

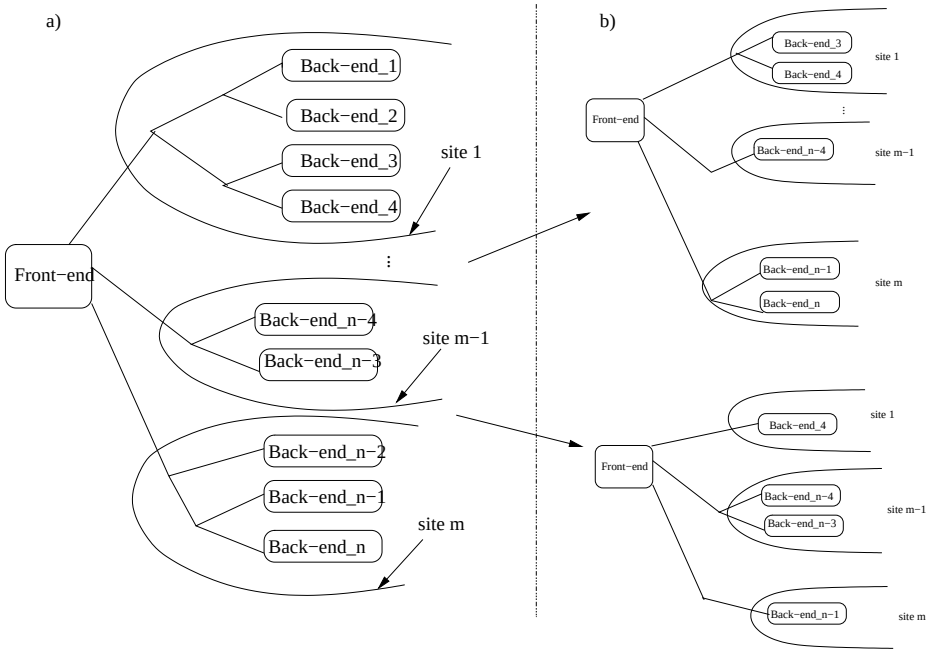

Figure 6.1: Multicast/reduction overlay network

(see also section 4.2) which deal with similar problems, this solution allows the user to have full control over the distribution of the computation since the overlay network is specific for the measurement scenario, and thus it depends only on the metrics specification and definition performed by the user.

The first step towards building this overlay network is to have a flexible way of describing the metrics to be measured. There are different ways to specify metrics as discussed in section 4.3. Those methods include the most widely used graphical or menu-driven and specification language based ones. Through menu-driven metrics specification, information is collected usually from the user interface using different kind of components (buttons, lists, combo-boxes, check-boxes, etc.). This approach is actually user friendly in such a way that the user can specify metrics fast and easily but is not flexible enough to specify a more abstract metrics since it is defined by the programmer of the user interface who can not cover all the possible measurement scenarios that the user may want to define.

An efficient way of specifying metrics is, therefore, using a dedicated specification language which is more flexible and gives individually more freedom of specifying metrics. Using a specification language can allow to express the aims of the user rather than that of the developer. Due to this fact, the dedicated specification language PMSL is used to specify all the metrics, as will be discussed in section 6.2.

In GPM, the performance measurement specified by the user defined metrics is usually based on pre-defined metrics, plus some events to be detected in the ap-

plication to be monitored. After the specification of a metrics using PMSL, the measurement definition must follow. As shown in the Fig. 6.1 defining a specified metrics for different parameters results in creating different measurement scenarios using different overlay networks. After the specification of the metrics and the definition of the corresponding measurement, the measurement can be started which in turn starts the process of distributed evaluation, automatically. The process of a distributed evaluation of user defined measurement, as it is realized in this thesis can be described with the list of subtasks as shown in Fig. 6.2.

1. Generating an intermediate representation in form of DAG from the metrics specification.
2. Generating a DFG from the corresponding DAG.
3. Partition of the DFG to sub-DFGs according to the remote locations where those subtasks will be computed.
4. Creating communication links between the front-end and the back-ends components of the DFG.
5. Creating monitoring requests for the whole measurement.
6. Distributing the sub-DFGs to their proper remote hosts.
7. Reassembling, synchronizing, and routing the measured result values to the consumer.

Figure 6.2: Steps of the distributed evaluation of performance data.

Through distributing the sub-tasks represented by sub-DAGs to their appropriate hosts and evaluating them in a distributed way and collecting the asynchronous measurement results dynamically on the run time through evaluating the main task represented by the main DFG, an overlay network between the front-end and the back-ends is established. This is used to manage the distribution, computation and reassembling of the measurement results. With this efficient distributed evaluation of computations, applications can be monitored with minimal and thus negligible additional overhead.

In general, every developer who programs an application to run in a distributed environment would like to see whether his application is executed correctly and behaves as expected in a given environment. There are applications which need process level investigation in order to find out, for example, the reason for the performance degradation. The developer, therefore, want to see, e.g., that each process of his MPI grid application uses the CPU more or less evenly, and that there is no process which shows a heavily loaded CPU. For MPI applications where every process, except the master process, usually behaves similarly since they are execut-

```
// test_probe.c file
void region_begin_event(int vt) {}
void region_end_event(int vt, double residuum) {}
void probe_in_iteration(int iter) {}
```

Figure 6.3: An example of a C file used to define probe functions

ing the same algorithm and synchronize themselves after each iteration, unevenly usage of some of the processes can be an indication for performance degradation.

Another important behavior for metrics based on communication is to achieve a balanced volume of communication. Since the performance for such an environment is reduced to the performance of the slowest process, a per-process evaluation of the application, as it is supported by the OCM-G, must be performed. Thus, OCM-G, as a hierarchical monitoring system, is a suitable infrastructure for such kind of computation.

In order to inspect application and grid environment specific behavior, one must be able to combine built-in metrics with each other. In addition to the built-in metrics, the user must also use probes (see also subsection 6.2.1) in order to have an access on event triggered measurement results and application specific data. Using those probes, the developer can specify a portion of source code of the application program as a location for a measurement. This is achieved through inserting pair of probes at the beginning and at the end of the desired location. Those probes can be used in different metrics without recompiling and re-running the application. As shown in Fig. 6.4, only the relevant places are marked in the source code using probes. The only additional information needed from the user is to define an empty probe function in a separate C file as shown in Fig. 6.3.

To avoid recompiling the source code, the developers of the applications can insert some more probes in the location where a performance problem is suspected or even in the locations which may contribute to find the location where the performance problem arises. The probes defined in Fig. 6.3 can then be used in the application code as shown in Fig. 6.4. The probe identified by ***probe_in_iteration*** function is used in two nested "for" loops to compute, for example, the data received in each outermost loop iteration using ***MPI_receive***. Another parameter of a probe in the metrics specification is a virtual time, which is an arbitrary, monotonically increasing integer value, used to identify not only different events, but also measurements belonging together. Thus, probes using the same virtual time can easily be combined together. For example, the ***region_begin_event*** and ***region_end_event*** in Fig. 6.4 can be used to compute the data provided by the built-in metrics in the application code between the two probes. In addition, the ***region_end_event*** probe provides the residuum value, which is an application specific data.

Using these probes, the developer can access computation results between the beginning and the end of the processing of a certain class of user interaction, which

```
// ... some application code here ...
int loop_iter = 0;
for ( ... ){
   for ( ... ){
   //some code for the communication
   //and computation here
   ...
   }
  probe_in_iteration(++loop_iter);
}
// ...some application code here ...
region_begin_event(vt);
// some application code here ...
//like MPI_send, MPI_receive, IO_volume, ...
...
double residuum = ...
region_end_event(vt, double residuum);
```

Figure 6.4: Using probe functions in an application code

enables to measure a metrics for a single interaction phase. Through using different virtual times for the same probes, it is also possible to measure, for example, the amount of data transferred to the other processes during application steering. This is possible while the value of the virtual time can be associated, for example, with the loop counter as shown in Fig. 6.4. Such issues are discussed briefly in chapt. 6 and 7. For those probes to work, a wrapper function creates code from the probe function definition during the compilation of the application to enable measuring the event triggered measurements.

6.2 Performance Metrics Specification Language (PMSL)

As discussed above, GPM provides low level built-in metrics which can be used to analyze basic behavior of applications, but the performance information provided by those built-in metrics are not abstract enough to evaluate metrics specifying a grid infrastructure or application specific computation. In contrast to these standard metrics, user defined metrics can combine and aggregate these built-in metrics and use event detection in order to have a complex and high level measurements. PMSL provides the possibility of defining user defined metrics which supports application dependent and/or grid environment specific measurements.

6.2.1 Basic Concepts of PMSL

PMSL [2, 10] is a declarative, functional language, which has been developed to support the user to describe new metrics and to simplify the distributed evaluation of metrics measurements. This language is used to specify performance metrics of MPI applications [30] running on the Grid. Using this dedicated specification language, it is possible for the user to specify a high level measurement metrics used not only to correlate the applications performance to the performance of the Grid environment, but also to inspect the performance of every single interaction of interactive applications. This language simplify the way how built-in metrics can be combined with each other and how to take application specific data into consideration. Through this way, the performance analysis tool can be adjusted to the needs of the user. In this section the syntax and semantic of PMSL will be discussed while examples and usage scenarios are illustrated in chapt. 7.

Combining Pre-defined Metrics

As a devoted language for the GPM tool, PMSL allows to combine and correlate all pre-defined metrics in order to provide more abstract metrics. The pre-defined metrics as discussed in section 5.3 provide data which are not enough to formulate more abstracted metrics, e.g., to describe complex performance measurement analysis of interactive applications on the Grid. For the combination and correlation, PMSL, as shown in Fig. 6.5. supports all usual arithmetic (plus, minus, times, div, mod, ...) and logical operations (as illustrated by the non terminal symbol ***"RELOP***"). Using some aggregation function (as specified by the non terminal symbol ***MATHFUNC***), computed values can be aggregated depending on certain conditions to be fulfilled. Those conditions can be restriction in time and/or location. For example, a selective computation can be performed only for certain applicable objects or virtual times within an interval.

Probes

As mentioned above, the detection of specific events occurring in the application can be realized by using probes. Those probes, used for event triggered measurements, are applied in the level of arbitrary code and require to be inserted at the relevant places in the source code. In the PMSL specification, a probe is defined using an arbitrary name followed by at least two parameters as shown in Fig. 6.5. Those parameters point out the process where events may occur and the virtual time. Since the same probe can be executed by different process when the corresponding event occur, the first parameter, which indicates the process in focus, can be used to evaluate per process computation. Using the virtual time parameter, probes can be paired or associated to each other through assigning the same virtual time for different probes.

Additional parameters can be used to access application specific data which is accessible under the scope of the probe function. A probe can also have a shifted

```
metricsDefinition: ID '(' parameterDecList ')' body
parameterDecList: parameterDec
  | parameterDecList ',' parameterDec
parameterDec: type ID
type: ID dimensions
dimensions: '' | dimensions '[' ']'
body: '{' statementList '}'
statementList: declaration statementList
  | assignment statementList | 'RETURN' expr ';'
declaration: type names ';' | type ID '=' expr ';'
  | 'PROBE' ID '(' probeParameterTypeList ')' ';'
names: ID | names ',' ID
probeParameterTypeList: probeParameterType
  | probeParameterTypeList ',' probeParameterType
probeParameterType: type | type ID
assignment: lhs '=' expr ';'
lhs: ID defIndices
defIndices: '' | defIndices '[' ID ']'
expr: - expr | + expr | expr '+' expr | expr '-' expr
  | expr '*' expr | expr '/' expr | expr '%' expr
  | term | term 'AT' ID '(' parameterList ')'
term: '(' expr ')'
  | MATHFUNC defIndices '(' expr optWhere ')'
  | MATHFUNC '(' expr ',' parameterList ')'
  | ID '(' parameterList ')' | DOUBLE | INTEGER
  | ID indices | term '.' ID | '[' ID ',' expr ']'
  | term '.' ID '(' parameterList ')'
indices: '' | indices '[' expr ']'
parameterList: expr | parameterList ',' expr
optWhere: ''
  | 'WHERE' boolExpr
boolExpr: boolExpr 'OR' boolExpr | boolExpr 'AND' boolExpr
  | 'NOT' boolExpr | '(' boolExpr ')' | expr 'IN' expr
  | expr RELOP expr
MATHFUNC: 'SUM' | 'PROD' | 'MIN' | 'MAX' | 'MEAN'
  | 'STDEV' | 'COUNT' | 'UNIQUE'
RELOP: '==' | '!=' | '>' | '<' | '>=' | '<='
```

Figure 6.5: An abstract grammar for the PMSL language

virtual time parameter, which enables to access result values computed at a previous virtual time. It is, therefore, possible to access, for example, result values in a loop that are computed previously than the actual loop.

PMSL Syntax

The syntax of the PMSL resembles a function definition of C/C++ as shown in Fig. 6.6. A metrics specification in PMSL contains the name of the metrics to be defined, a list of parameters, and a body as shown in Fig. 6.5. A single specification can contain one or more functions which may depend on one another. Each function must be specified with a unique function name and a return value. Optionally, functions can have different number and kind of parameters which are used to specify the constraint of the specified metrics. After specifying a user defined metrics, the metrics must be submitted to be measured. This results in inserting

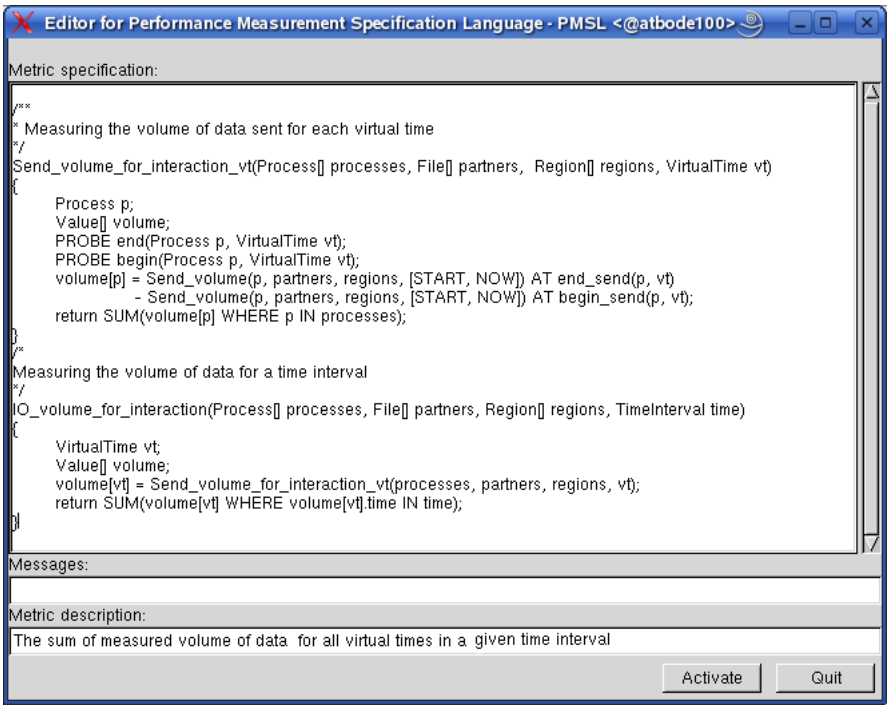

Figure 6.6: An example specification of user defined metrics

the new defined metrics (dynamically in the run time) to the list of available metrics. After the submission, the metrics will be parsed by the tool to create a C++ specification object representing the metrics. This new metrics can be measured and visualized in the same way as the built-in metrics. If an error occurs during the parsing process, the user will be informed about the exact location of the error in the specification.

During the specification of multiple metrics at once, if there is a dependency between them, as shown in Fig. 6.6 (see also [1]), their order must be taken into consideration. I.e. a user defined metrics to be used in another user defined metrics must be specified a priori. In our example above the function with the virtual time (_vt) must be defined before the function without it.

During the parsing process of the user defined metrics, the first element to parse is the name of the metrics, which is used to identify the metrics in the whole measurement process. Therefore, this name should be unique to the predefined metrics and to the metrics defined by the user previously. The superset of the parameters, which are the next to parse, contains the following number and type of parameters:

- APPOBJECT[] OBJECTS
- APPOBJECT[] PARTNERS
- REGION[] REGIONS

- TIMESPEC TIME

The first three parameters are optional. These parameters describe which object type is allowed to use for the measurements and which dimension they possess. Thus, the user can have different metrics definition with different numbers and combination of parameters. Every specification must posses at least one parameter describing the time of the measurement. These four possible parameters are explained as follows:

1. ***Objects:*** For every measurement, the location where the measurement will be performed, can be defined. Those are applicable objects including processes, nodes and sites. That means that the user can specify a measurement for the whole sites, for nodes in a particular site or for particular groups of processes within a node. Omitting this parameter in the parameter list of the specification will lead to the computation of the specified metrics for all possible applicable objects.

2. ***Partners:*** For measurements related with communications, the user can describe which partner object should be taken into consideration. This again can be specified as granular as the applicable objects.

3. ***Regions:*** The user can assign the region of the measurement for which the measurement will be performed. This could be either a portion of source code or the location of certain methods. That means that any measurement can be performed for the part of the code, or for the whole application.

4. ***Time:*** There are three different kinds of time specifications. Those are: a point of time, time interval and virtual time. To define a measurement interval, the following special values are used:

   (a) ***NOW*** describes the current time,

   (b) ***START*** provides the starting point of time for the whole measurement, and

   (c) ***LAST*** presents the point of time when a measurement value is read for the last time.

The body of the specification can contain a list of statements (declarations of variables and functions), assignments (which can be multi-dimensional), and a return statement used to provide the computed value of the specified metrics.

During the assignment, the expression on the right hand side must not contain any free variable that is not indexed. A free variable, in this case, is a variable in an expression, which has not yet a value assigned and into which a definite substitution may take place. Thus, a free variable is used as a placeholder. This declaration supports also multi-dimensional variables which are indexed variables realized by index substitution.

6.2.2 PMSL Usage

Using the text editor of UIVC of GPM, the user can specify any measurable metrics as shown in Fig. 6.6 which illustrates two metrics specifications showing the most important features of PMSL. In this metrics specification there are two different metrics dealing with the amount of sent data. The first one describes the sum of the amount of data sent in all the processes which are involved in the specified metrics measurement. Using the ***begin_send*** and ***end_send*** probes defined at the beginning of the specification, only those ***Send_volume*** values between the two probes are taken into account.

The specification ***Send_volume_for_interaction*** in Fig. 6.6 uses the specification ***Send_volume_for_interaction_vt*** to get the values of ***Send_volume*** computed between the two probe events within the specified time interval. Those time intervals can have a start point which can be the starting point of the whole measurement or the point of time when the measurement is read for the last time, whereas the end point is the actual measurement time.

The actual values of the parameters, except for the parameter dealing with the time specification, must be defined via the user interface during the measurement definition phase. The time parameter will be delivered from the underlying operating system after the measurement is started, when the measurement is read, and when an event in focus is occurred.

6.3 Evaluation of Metrics Specification

In order to measure performance data, the parameters of the metrics describing the measurement must be defined. Defining a metrics means assigning values to the parameters describing, the applicable objects, the partner objects, and the region of the measurement in the code of the application for which the specified metrics will be measured. The time parameter can only be defined fully when the measurement is started.

6.3.1 Creating Intermediate Representation (IR)

After the user writes the desired metrics specification, an intermediate representation in form of a user defined metrics object will be created during the parsing process of the specification. This user defined object contains the IR of the specified metrics in form of a DAG. Fig. 6.8 shows, for example, such an intermediate representation of the metrics specification as shown in Fig. 6.7. The generated intermediate representation will then be used as a template for the definition of the specified metrics. Depending on the parameters used during the definition, the corresponding template will be adapted to the required measurement scenario.

The example metrics, as depicted in Fig. 6.7, shows some important future of the PMSL (see also some other features in section 7.2) describing the load imbalance at barrier. For this example metrics, a bottom up concept is applied on

```
1. Load_imbalance_at_barrier(Process[] procs, VirtualTime vt)
2. {
3. PROBE loop_start(Process, VirtualTime);
4. PROBE loop_stop(Process, VirtualTime);
5. Process p;
6. Process p0 = UNIQUE(p WHERE p.rank == 0);
7. Value loop_time = Time(NOW) AT loop_stop(p0,vt)
                          - Time(NOW) AT loop_start(p0,vt);
8. Value[] barrier_time;
9. barrier_time[p] = Barrier_delay(p, [START,NOW]) AT loop_stop(p,vt)
                   - Barrier_delay(p, [START,NOW]) AT loop_start(p,vt);
10. Value bmax = MAX(barrier_time[p] WHERE p IN procs);
11. Value bmin = MIN(barrier_time[p] WHERE p IN procs);
12. return (bmax-bmin)/loop_time;
13. }
```

Figure 6.7: Metrics specification for load imbalance at barrier

the measurement specification to generate the desired intermediate representation. That means that starting from the leaves of the DAG, the whole DAG will be constructed successively. That means that whenever an assignment is parsed, a sub-DAG will be created and whenever the value of an assignment is applied, the created sub-DAG will be attached to the existing DAG to create the final DAG, which represent the specified metrics. For example, if we have an operation between any two expressions, an operation node will be created and those two expressions will be its child nodes. Those child nodes in turn can also have child nodes if they have to be computed further.

For example, at line 6 in Fig. 6.7 the first sub-DAG will be created which have a root node representing the ***UNIQUE*** operation returning a single arbitrary process with a rank 0. With the same mechanism, all other sub-DAGs will be created and will be joined together to create the final DAG rooted at ***DIV*** operation node using the return assignment at line 12. The root node contains the two sub-DAGs rooted at their ***SUB*** operation nodes as shown in Fig. 6.8. The first one contains the ***MAX*** and ***MIN*** operation nodes as its child nodes whereas the second one represents the value of the elapsed time assigned as a subtraction of the predefined metrics ***Time*** measured at two different probe events. In this way, the complete DAG is constructed.

To combine and aggregate built-in metrics (like ***Time***, ***Barrier_delay*** and ***Barrier_time*** metrics in Fig. 6.7), different mathematical operations are provided by the PMSL. Since all other operations are used as usual, the functionality of some special operators like ***AT*** and ***IN*** will be discussed here. The ***AT*** operator is a special expression used to specify the probe functionality. This operator takes the value of an expression when an event occurs. Therefore it must have at least two child nodes, as shown in line 7 and 9 of Fig. 6.7.

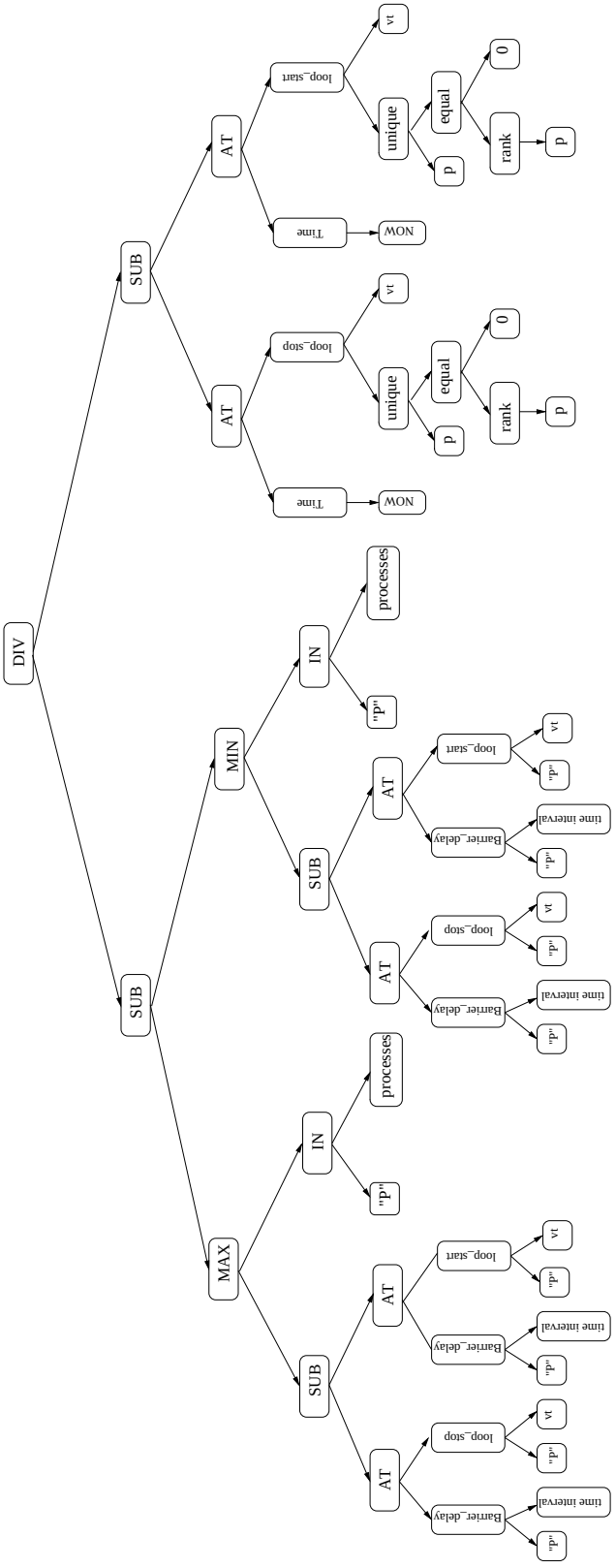

Figure 6.8: A DAG for the Load_Imbalance_at_barrier metrics in Fig. 6.7

The ***AT*** operation must have a sub-DAG and a probe node as its child node, and the child sub-DAG is not allowed to contain any specification describing event triggered computations. This sub-DAG can be a single node, which, e.g., may provide a constant value or a complex sub-DAG. The semantic of this intermediate representation is as follows: Whenever an event on the right hand side arises, the sub-DAG at the left hand side will be evaluated or a constant value will be delivered. For each pair of events (for example, a begin and an end event specified at line 7 and 9 in Fig. 6.7) the sub-DFG as shown in Fig. 6.11 will be generated.

The ***IN*** operation, as shown in the example metrics at line 10 and 11, is used to prove if an element belongs to a group of elements. This actually provides to some extent the functionality of control statements like ***if*** and ***else*** in classical programming languages. For example, the expression ***p IN processes*** returns true if and only if the process assigned with ***p*** belongs to the list of processes assigned with ***processes***. Since the ***IN*** operator is used most of the time with the ***WHERE*** functionality, Fig. 6.12 shows the part of the DAG for the ***IN*** operator together with the ***WHERE*** operator. A simple example of the ***WHERE*** expression is used in line 6, 10, and 11 and the corresponding DAG for such expressions is shown in Fig. 6.8. For this example metrics, if the applicable object parameter is defined for three processes, then three sub-DAGs will be created during the definition of the DAG.

The set operations ***MAX***, ***MIN*** and ***UNIQUE*** are used in the specification to perform the corresponding mathematical operations over all elements of the specified set. The DAG created for those operations can only have one or two child nodes depending on the parameter used. The first parameter is a simple expression and the second one is again an expression containing the precondition to be fulfilled in order to evaluate the first expression. These preconditions are realized using free variables which are used as an iterator. Those free variables are application objects, regions, or virtual times. Those two expressions are separated by a ***WHERE*** operator to perform the operations only on the selected set of applicable objects. The expression on the left hand side must not contain additional free variables. Fig. 6.12 shows the DAG and the corresponding DFG for the statement containing the WHERE expression. Line 6 in Fig. 6.7, for example, shows the usage of the attribute ***rank*** applied on a process object returning the master node.

In order to optimize the whole evaluation and reduce the complexity of the computation, an Common Subexpression Elimination (CSE) process is performed before the measurement is defined. This is realized by searching for common sub-DAGs and replacing them with a pointer to a single common sub-DAG to eliminate multiple copies. This is used mostly for the nodes representing the built-in metrics and enables to have their efficient implementation. Fig. 6.8 shows the intermediate representation of the example metrics used, whereas Fig.6.9 shows the DAG after the elimination of the common subexpressions of the corresponding DAG. After the optimization of the intermediate representation, the creation of the user defined specification object representing the whole user defined measurement specification is completed. This object contains an intermediate representation of the given spec-

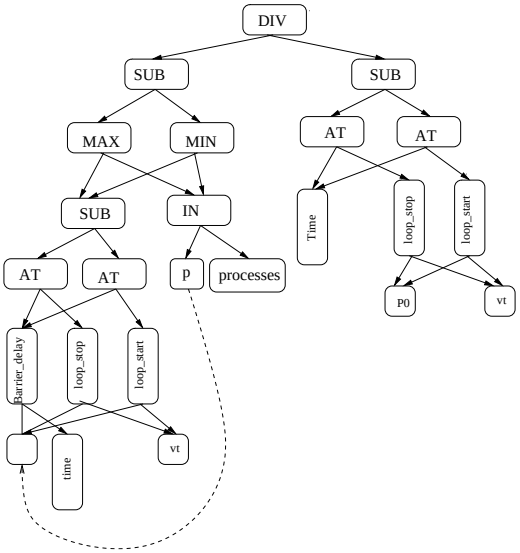

Figure 6.9: A template for the example metrics shown in Fig. 6.7

ification for the corresponding metrics as a DAG and is used to generate an Active User Defined Metrics object (AUDM), which will be used to control (specially to start, stop and delete) the whole measurement.

To simplify the complexity of the resulting graph, sub-DAGs having similar structure and having the same applicable objects are represented with a single sub-DAG (like the sub-DAG rooted at ***MAX*** and ***MIN***). This summarizes the number sub-DFGs representing the same qualitative information. When there is more than one user defined metrics specified, the same procedure will be applied for all other specifications. That means that for every single metrics there will be an intermediate representation, which specifies the user defined specification object. Those specifications can also consist previously defined metrics, which are specified by the user.

After the parsing process of the example metrics is accomplished, we will have the corresponding intermediate representation as shown in Fig. 6.9. As will be discussed in the following subsection, all possible parameters must be defined in order to measure the specified metrics.

6.3.2 Measurement Definition

After eliminating the common sub-expressions, the DAG has now an optimized intermediate representation in form of another DAG which will be used as a template for the specified metrics. Since a metrics is only a quantity description like area or volume that needs to be measured, the necessary additional information has to be provided by the user using the UIVC of GPM. During the definition of a specified

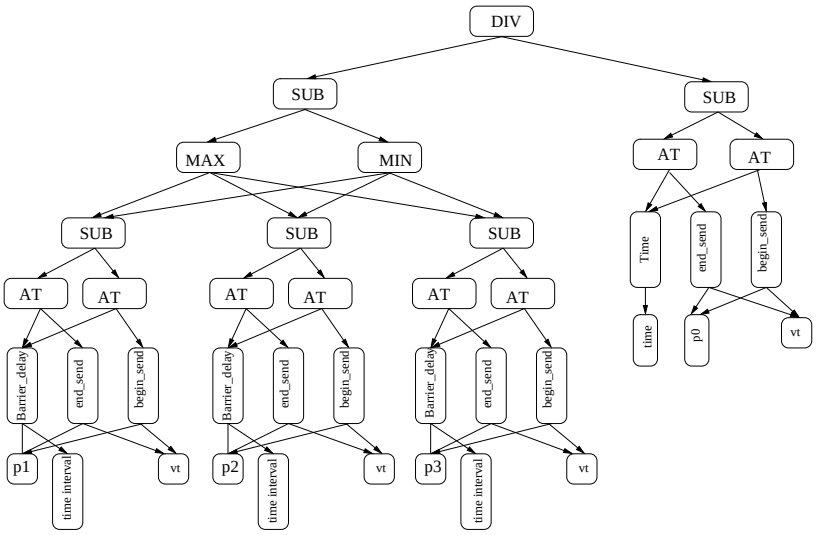

Figure 6.10: The result of metrics definition for the metrics specification in Fig. 6.7

metrics, the user must determine the applicable objects for which the measurement will be performed.

In order to provide the processes to be used for the evaluation of the specified metrics, the user must specify the Fully Qualified Domain Names (FQDM) of all the sites to be used for the distributed evaluation. Using these FQDN, the UI provides all the processes to be used for the computation. Thus, during a measurement definition, the corresponding template will be used to create a real measurement definition of the specified metrics using the provided information. When a measurement is defined for multiple applicable objects, the corresponding template will be used to generate a DAG for each applicable object and those DAGs will be summed up to a single DAG representing the whole measurement. Fig. 6.10 shows the definition of the example metrics only for three different processes (***p1***, ***p2*** and ***p3***).

Using the provided information for the metrics, different kinds of measurements can be created using the same template. As a result, once a metrics is specified and the necessary probes are created, a user can perform a parameter study through defining the same metrics using different applicable objects. In doing so, he can, for example, identify a suitable grid environment for his application through performing some metrics measurements using different sites and comparing their application progresses as will be discussed in section 7.2.

As discussed in subsection 6.2.1, the user defined metrics can have different number and types of parameters. The user should specify only those parameters which are going to be used even if parameters which are defined but not used will be ignored. For example, the metrics ***Load_imbalance_at_barrier***, as shown in

Fig. 6.7 is a kind of metrics in which the partner objects and regions are omitted in the parameter list since they are not required.

After the definition of a measurement, a partial evaluation of the DAG using those available parameters is performed automatically. All elements which are actually not related with the time parameter can be used for the partial evaluation of the DAG. The parameter describing the time of the measurement can only be substituted with a real value whenever the user starts measuring the defined measurements and the GPM reads the result values. Before starting the measurement, a user is only able to determine the summary and aggregation mode of the measurement, which has also an impact on the final measurement result values. As a result, the final value can be a single summarized value, or a set of individual measurement values come from a set of partner objects. The start point of a measurement can either be a start point of the whole measurement or the point of time where the last measurement has been read as discussed in section 6.2.

During the evaluation of a priori defined user metrics, their measurement definition will be used. This is the case for the ***Send_volume_for_interaction_vt*** metrics as depicted in Fig. 6.6. That means that the corresponding template DAG for this measurement definition will be used in the ***Send_volume_for_interaction*** measurement. During the metrics definition phase, the parameters of not only the user defined metrics and the built-in metrics, but also user defined measurements defined a priori will be assigned.

If the user decided to use an array of parameters, even if the measurement was specified to accept scalar parameters, the tool will perform an optimization automatically, so that for each combination of those parameters a sub-DAG will be created to be summed up to a single final DAG. For example, if the user wants to use **x** application objects, **y** partners and **z** regions, the number of sub-DAG to generate will be the result of multiplying the dimension of those three objects $(x * y * z)$.

6.4 Generating Augmented DFGs

After a measurement is specified and defined as desired, it will be evaluated in a distributed way. The defined measurement describes the main task to be performed. This main task will be partitioned into different subtasks. Those subtasks will be distributed and evaluated on a remote host independently. In order to distribute the subtasks into their appropriate locations and to reassemble the results effectively, the introduced ADFG (in order to simplify the description, we use DFG instead of ADFG hereafter) is used which is discussed briefly in section 3.3.

The introduced DFG created form the intermediate representation consists different kinds of DFNs with different kinds of firing rules. Those DFNs are used to aggregate, correlate, reroute and evaluate the result values properly whereas the entire DPs consume result values. This DFG is generated from the corresponding DAG after the definition of the measurement has taken place. The DFNs are cre-

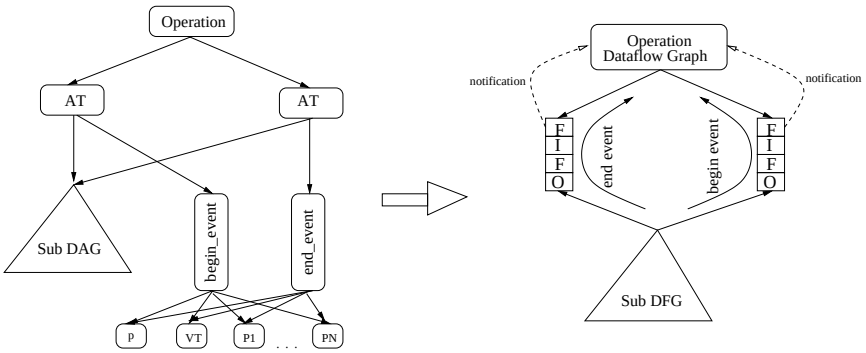

Figure 6.11: Transformation of a sub-DAG to the corresponding sub-DFG

ated from the respective nodes of the DAG, i.e., almost every node of the DAG will be converted to the corresponding node of the DFG.

For the event triggered measurements, for example, the ***AT*** node will be eliminated and its information will be embedded in the measurement object and in the corresponding monitoring requests as will be discussed in section 6.7. Fig. 6.11 shows the conversion of a DAG representing the AT operation into the corresponding DFG. Since the same DFN must handle different results, a requester identifier is used to route the results to the correct DPs. The DFG in the Fig. 6.11 shows how the result of a sub-DFG is forwarded to two different DPs depending on whether the result belong to the begin or end event. When the request to the monitoring system is created, the correct consumer is associated with the unique identifier. Therefore, it is possible to forward the result values to the proper requester using the requester identifier, which is associated with the measurement.

The ***IN*** operation is used to perform actions on a selected set of objects as discussed in subsection 6.3.1. For the set operations using the ***WHERE*** expression, the transformation is done as depicted in Fig. 6.12. In this case, the sub-DAG is evaluated for those objects which are selected using the ***IN*** operation. This may result in a number of sub-DAGs on which the set operation will be performed. If the ***IN*** operation is used to perform action for specified virtual times in a time interval, only one child DFG will be created and the virtual time operation node will collect the result values computed using the virtual time and wait until it is requested to provide its intermediate results.

In order to set up a communication link between two DFNs residing on different sites, the DPs serves also as a connector component. The main purpose of those DPs is to collect the measurement results, which eventually must be synchronized before they are aggregated. To manage the distribution of subtasks and to collect all the necessary result values, a bidirectional connection between any DFNs and their corresponding parent and child DPs, as discussed in section 3.3, is applied. The final DFG for the example metrics in the previous section is depicted in Fig. 6.13.

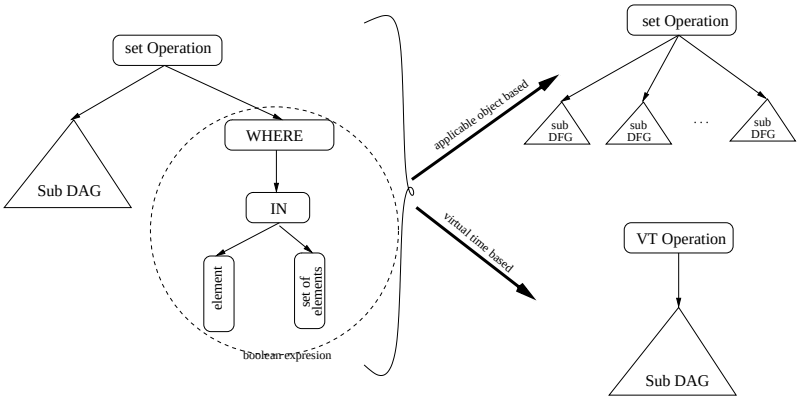

Figure 6.12: Transformation of sub-DFG using set operations for the aggregation

DFG Components

There are different kinds of DPs and DFNs used in the DFG.

Kinds of DPs

There are two types of DPs:

- *DPs for constant value*s: These date providers don't have child DFNs which provide them with result values. They are simply delivering a constant value, which was assigned to them during the specification of the measurement. Therefore, these DPs are not used to connect DFNs. Instead, they are used as producers of constant numerical data and thus reside at the endpoints of the DFG.
- *DPs for measurement values:* These types of DPs are FIFOs used to collect the result tokens delivered by their child DFNs. The collected values will then be propagated into the parent DFNs to be aggregated or combined on their way to the root of the DFG. This means that the child DFNs of a DPs write result values into the DPs and the DPs inform the availability of these result values to the parent DFNs which may try to collect the result values.

There are also different types of DFNs connected by DPs. These sorts of DFNs are described as follows:

- *Attribute DFNs*: Those nodes are used to evaluate the attribute of objects. They are used to find out, for example, the rank, site, and node of a process.
- *Iterator DFN*s: Such nodes are used to provide measurement results or constant result values to the parent DPs. They are created from the iterator nodes of the corresponding DAG. An event based measurement use such kind of

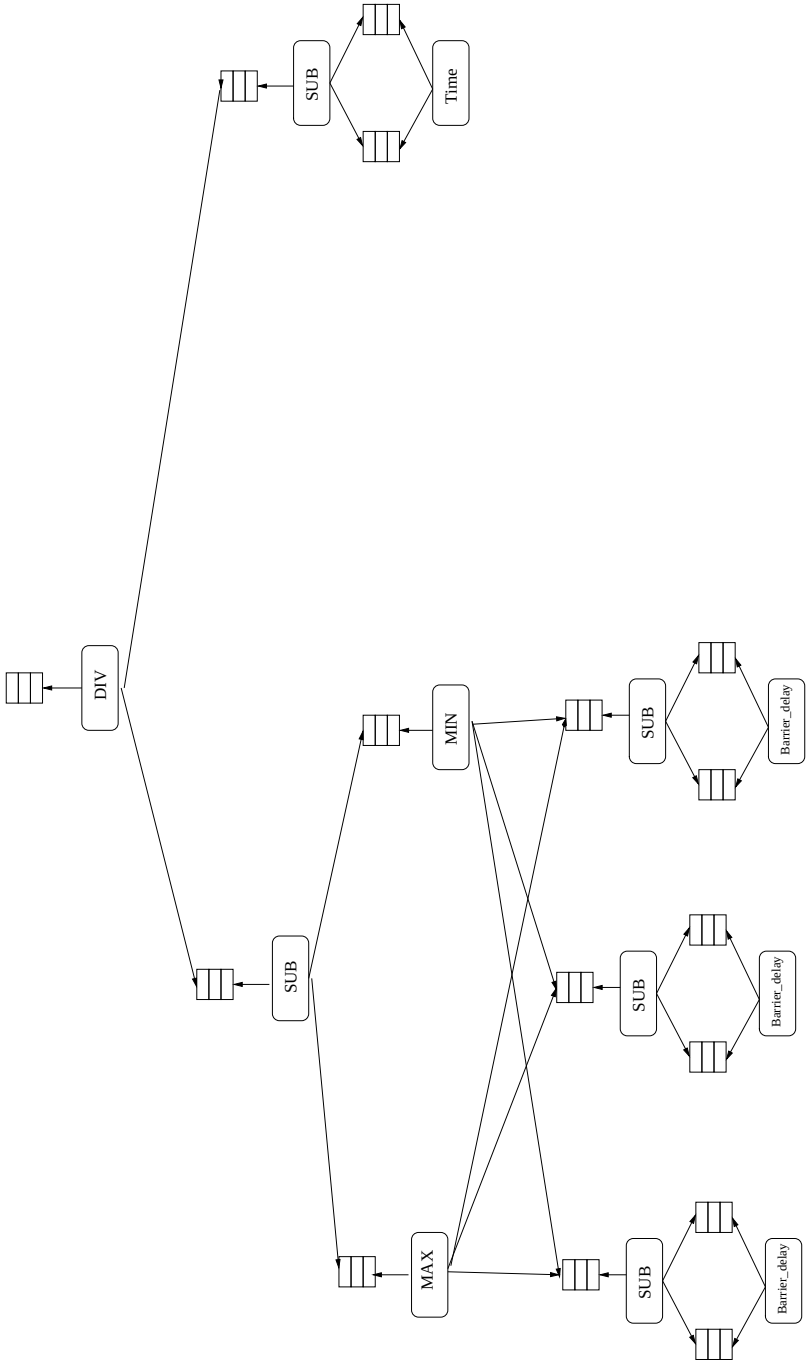

Figure 6.13: A DFG for the Load_at_barrier metrics

data producer to provide a constant value when the corresponding event occurs.

- *Operation DFNs*: Those nodes represent mathematical operators as discussed in subsection 6.2.1. Those nodes are created from the corresponding operation nodes of the DAG. Except for the "AT" operation node, which will be eliminated during the creation of a DFG as shown in Fig. 6.11, a DFN will be created for all other operation nodes.

- *Virtual time aggregated DFNs*: Those nodes accumulate result values according to the given virtual time. Those accumulated values will then be accessed whenever they are desired.

- *Virtual time shifted* DFNs: These nodes are used to have an access on the values computed for the previous virtual times than the actual one, and are created during the evaluation of the underlying DAG. They can be used to compute the values between the actual and the previous virtual times. Therefore, they are the results of probe nodes which have a virtual time shifted with an integer number.

- *DFNs for measurement results*: Such kind of DFNs are used to manage the measurement results provided by the built-in metrics and are also used as connector DFNs (see also section 6.5.2). Since the built-in metrics provide the results asynchronously via a callback function, an active measurement which is created from the built-in metrics assigns this DFN as a consumer object in order to forward the measurement results to this DFN. In addition to this information, an active measurement context containing the necessary information for the measurement will be created at the same time. This context object comprises information like requester identifiers, the consumer DFNs, the corresponding active measurements of the built-in metrics, a measurement specification of the built-in metrics, plus the information whether and how the results of the measurement can be read from within an OCM-G extension.

For the aggregation modes of the built-in metrics represented by the measurement result DFN, the following four time specification are used to provide the result of the measurement as an integral over time:

- *SINCE_START*: In this case, the result of the measurement will be computed for all of the results from the start of the measurement. For example, for the built-in metrics ***Send_volume***, the sum of the data sent from the beginning of the measurement are taken in to consideration.

- *SINCE_LAST_READ*: The result of a measurement using this integration mode is the difference between the current integral value and the integral value at the time of the previous reading of the measurement. This is used

specially by the user defined metrics in order to provide, for example, results related to user interactions.

- *SINCE_START_DIVIDE_BY_TIME*: The result of this integration mode is a measurement result divided by the duration of the time interval which has its starting point at the beginning of the measurement. The results are something like a mean derivative.

- *SINCE_LAST_READ_DIVIDE_BY_TIME*: For this mode, the result is the difference between the current integral value and the integral value at the time of the previous reading of the measurement, divided by the duration of the time interval. For example, for the metrics specifying the amount of communication, the result would be the obtained communication bandwidth.

6.5 Decomposition of the DFG

In this part, the cost-effective scheme for partitioning the main DFG will be presented. The main objective of this is to reduce the overhead due to token transfer through communication network. In other words, the partition of the main task to its subtasks leads to load distribution from the front-end to the back-ends.

Most of the metrics which can be measured in a distributed parallel computation environment can be reduced to different sub measurements, which can be measured independently in their remote location depending on the definition of the measurement in focus. In our case, the main DFG represents the whole task, which may be partition to sub-DFGs representing the subtasks. Therefore, the first step for the partition of the main DFG is to identify those subtasks. This is done through determining the remote host (such hosts will be referred to as access location, hereafter) for the sub-DFGs where they can be evaluated without communicating with objects which are reside in another host.

Before starting the partition process of the main DFG, it must be determined which sub-DFG is going to be taken into account. Thus. the first step is to determine whether a sub-DFG fulfils the criteria to be computed at a remote host. For this purpose, the access location of the sub-DFGs must be determined first. To achieve an efficient implementation, determining the access locations is performed on the DAG and the computed attributes describing the access locations are propagated to the DFG. Therefore, in the following subsection the determination of the access locations on the DAG will be discussed.

6.5.1 Determining the Access Locations of Sub-DFGs

In order to determine the access location of the sub-DAGs, a bottom-up approach is used on the main DAG. For this purpose, end-points of the DAG representing the back-ends return the necessary information to find out the access location of the sub measurement. This information is provided by the attribute specifying the access

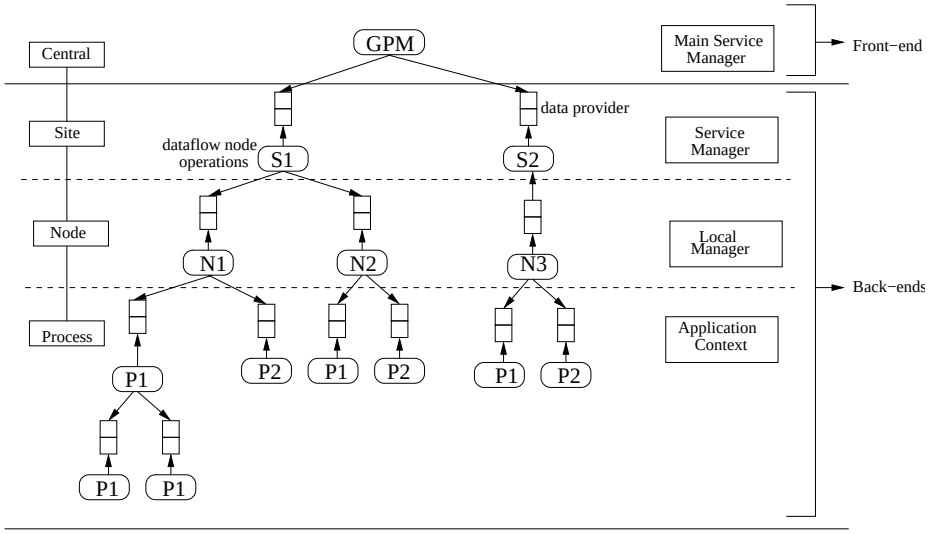

Figure 6.14: Hierarchy of applicable objects and their corresponding monitoring components

location. Since the end-points of the DAG are sub measurements specified by the built-in metrics or events to be detected, determining the access location of those end-points will be performed at first. For the sub measurements, the measurement specification object of the corresponding metrics will be used to get the desired measurement access location. The information to determine the evaluation location of a built-in metrics is extracted from the applicable object parameter.

For example, when a task represented by end-points of the DAG is going to be computed by the processes located on different nodes which reside on different sites, and when the processes must communicate with each other during the evaluation of the measurement, then the overall evaluation of this subtask can obviously be evaluated only globally at the front-end. That means that end-points of the DAG representing constant values or representing metrics which can be computed on every process, like the ***Time*** metrics (which returns the synchronized time), deliver the access location attribute which describes that the computation of that metrics can be performed on any process located on any host. In this way, the access locations of the end-points are determined correctly.

After determining the access locations of the end-points, the access location of the parent nodes can easily be determined depending on the information of the child nodes. The following algorithm describes the determination of the access location of the parent nodes, which are then recursively used to determine the access location of a sub-DAG. This is used to determine whether two applicable objects are located on the same host. Similar algorithm can be used to find out whether the applicable objects resides on the same site.

$$SameNode(a,b) := \begin{cases} a.node == b.node & if(a \in process \wedge b \in process) \\ a.node == b & if(a \in process \wedge b \in node) \\ a == b & if(a \in node \wedge b \in node) \\ false & else \end{cases}$$

where ***a*** and ***b*** are child nodes of a node calling the ***SameNode*** function which returns whether ***a*** and ***b*** are on the same node or not.

When a parent node has child nodes whose access location are on different hosts but on the same site, then the parent node becomes the common site as its access location otherwise the access location will be the front-end. In this way, the access location of all sub-DFGs will be determined. Fig. 6.14 shows the hierarchical view of the applicable objects and the corresponding monitoring components.

This determination process is slightly different for the AT operation node which deals with the event based measurements. Generally, the access location of a probe node is the location where the corresponding event arises and this is the same as the location where the action will be performed. This is usually a remote host except for the reason that the machine where the user interface runs is also used to compute measurement.

If the parameter used by the built-in metrics to determine the access location is one-dimensional array, the common evaluation location of the corresponding applicable objects must be determined. This means, if the processes of the applicable objects are on the same remote host, then the common evaluation location will be that host. Otherwise, the evaluation will be performed at the front-end in a centralized manner. In this way, the access location of every node of the DAG can be determined.

Specifying Subtasks

According to the definition of measurement metrics, the subtasks to be performed at the desired location at the back-ends can be determined. Subtasks to be sent to their corresponding remote hosts can consist of different sub-DFGs. Therefore, it is efficient to collect these sub-DFGs at the front-end before sending them to their corresponding back-ends.

Using the access location of the sub-DFG, we can search for sub-DFGs which can be computed on the remote hosts. However, the information provided by the access location of a sub-DFG will is not sufficient for the partition of the main task, since a subtask can contain multiple sub-DFGs. For this main reason, the determination of subtasks represented by similar sub-DFGs, which are going to be sent to the same remote host, must be done to achieve an optimized DFG partition.

To reach this goal, the first task is searching for sub-DFGs which can be evaluated at their remote host. This will be performed in a top-down manner as follows: Starting from the root node of the main DFG, all child nodes will be verified

whether they have an attribute representing a remote host as their access location. If they have a global access location, then their child nodes will be examined recursively. If a sub-DFG is going to be sent to its appropriate remote host, then its root DFN will be assigned as a parent DFN for all DFNs of this sub-DFG.

The following algorithm assigns the access location of the sub-DFG, recursively. The function ***setParent*** sets the root node of the sub-DFG to be the parent of the entire DFNs of that sub-DFG and the root DFN (DFN_0) of the main DFG will initiate the ***setRemoteHost*** function as shown below.

$setRemoteHost(DFN)$
{
  $for(i = 0, i < N(C_{dfn}(DFN)), i++)$
  {
    $if(H(C^i_{dfn}(DFN))! = 0)$
    {
      $setParent(C^i_{dfn}(DFN));$
    }$else${
      $setRemoteHost(C^i_{dfn}(DFN));$
    }
  }
}

Latter, if the root DFN of a sub-DFG is visited by another DFN which has the same access location, the parent DFN of this sub-DFG may be replaced by visitor's DFN. Through this process, the sub-DFG which belongs to the same destination host are collected together.

This task is essential since during traversing the graph, the same DFN can be visited by different parent DFNs which can be sent to the same destination host. The visited sub-DFG can be sent as part of the visitor's sub-DFG which is more general. Without resetting this attribute, a sub-DFG would be partitioned too early which can have an impact on the computational processes. In addition, it also avoids performing the same tasks which are necessary to disconnect the sub-DFGs from the main DFG, sending sub-DFGs to the same remote host and joining them on the remote host.

Only after this process, a root DFN of the sub-DFG can have its final status whether it will be sent as a sub-DFG or as part of another sub-DFG. As an example for this scenario, the connection between the DFN_n and DFN_i, as shown in Fig. 6.16 can be used in which case the sub-DFG rooted at DFN_i will be sent as part of the sub-DFG rooted at DFN_n since this will be visited first when using depth-first algorithm.

In addition to the root DFNs of the sub-DFGs, some DFNs are also used as connector points for different sub-DFGs residing at different remote hosts. Thus, the next issue is concerning about the entire DFNs or sub-DFGs within sub-DFGs to be sent to the remote host. I.e. it is possible to have some common sub-DFGs between two sub-DFG which can be sent to the same or to different destination

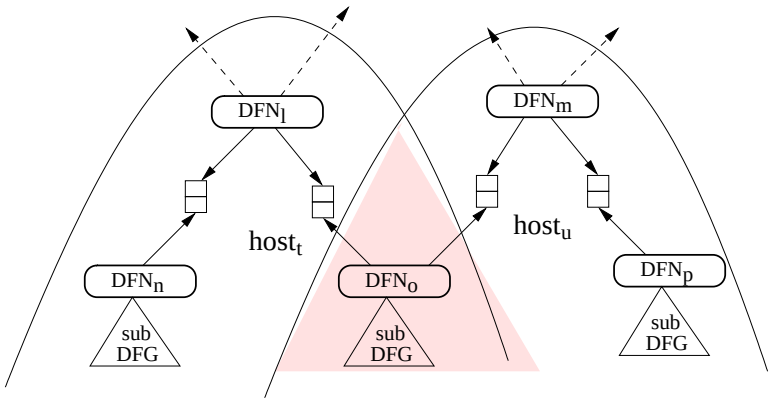

Figure 6.15: Handling common sub-DFGs within sub-DFG to be sent to remote host(s)

hosts. In addition to this the main DFG and sub-DFGs to be sent to the remote host may also share some common sub-DFG components as depicted in the Fig. 6.15. The kind of DFN which connect a sub-DFG with the main DFG will be referred to as connector DFN as discussed in section 6.17.

Concerning common sub-DFGs, we have the following three cases to differentiate. As discussed in section 3.3 $H(DFN_i)$ means that the access location of sub-DFG rooted at DFN_i and if its value is equal to 0 then it means that the access location is the front-end.

$if\,((H(DFN_o) == H(DFN_m))! = 0)$
{
// creating a link between sub-DFN_l and sub-DFN_m
// set sub-DFG_o be part of the sub-DFG_m
}
$if\,((H(DFN_o)! = H(DFN_m))! = 0)$
{
// copy the sub-DFG_o
}
$elseif((H(DFN_i)! = 0) \wedge (H(DFN_m) == 0))$
{
// create a communication link (see section 6.5.2)
}

As it is depicted in Fig. 6.15, the sub-DFG rooted at DFN_o is a common sub-DFG for the sub-DFGs rooted at DFN_l and DFN_m.

Those three cases are described according to the visitor's access location as follows:

1) The same remote host:

If $host_t$ and $host_u$ are the same hosts, the common sub-DFG between them will be sent as part of one of the visitor's sub-DFG whereas the other one will have only a pointer to the common sub-DFG. Through this process sub-DFGs are clustered to be sent to their remote host. At the destination host the original structure of the sub-DFGs will be reconstructed.

***2) Different remote hosts*:**

This implies that such a kind of common sub-DFG is independent from any access location and can be computed at a place where it is needed. In this case the common sub-DFG will be copied and sent to both destination ($host_t$ and $host_u$)**,** and this will not change the semantic of the computation since this common sub-DFG does not posses any access location. Because the two parent DFNs have different access location and the determination of the access location is performed in a bottom-up manner, this sub-DFG can not have any fixed access location and, thus, must be accessible everywhere.

***3) Remote host and front-end*:**

For the third case, one of the visitors DFN resides at the front-end and has a global access location since it must be computed centrally at the front-end whereas the other one is to be sent to the remote host. In this case, a communication link will be created as discussed in subsection 6.5.2. This communication link between the two hosts will be used to transfer the necessary measurement result values from the back-end to the front-end.

Let us assume, for example, the three sub-DFGs as shown in Fig. 6.16, rooted at DFN_i, DFN_k and DFN_n, respectively and these are going to be sent to the same destination host. Without resetting the parent DFN of some of the sub-DFGs, all the three sub-DFGs will be sent to the same remote host, separately. Since the sub-DFG rooted at DFN_i is part of the other two sub-DFGs, it will be sent multiple times. To avoid this, this sub-DFG must be sent once.

Since the sub-DFG rooted at DFN_k is on the higher hierarchical level, it will reset the "parent root node" of the sub-DFG rooted at DFN_i. That means, the sub DFN rooted at DFN_i will be part of the sub-DFG rooted at DFN_k. Through this process, it is possible to identify every DFN to which sub-DFG it belongs.

After it is correctly decided which sub-DFGs are going to be sent to the corresponding remote location, the process of detaching the sub-DFGs can be started. The detaching process itself is performed recursively on all of the child DFNs as far as possible. Before the sub-DFGs are detached, a communication link between the nodes which are going to be detached must be setup. Thus, the next section is devoted to illustrate how communication links between the main DFG and the sub-DFGs sent to their appropriate remote host will be created.

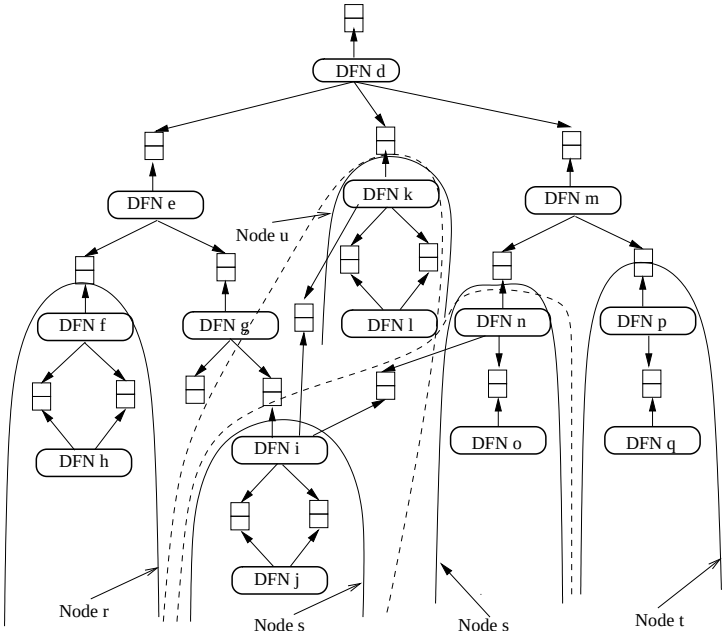

Figure 6.16: Collecting sub-DFGs belonging to the same remote host

6.5.2 Creating Communication Links

After the partition of the main DFG into sub-DFGs representing the subtasks to be computed at the remote hosts, there should be a way to connect those sub-DFGs to the main DFG. This connection is used to perform the reassembly of the measurement results computed by the sub-DFGs. The best way to do this is to create communication links between the back-ends and the front-end as depicted in Fig. 6.17.

Since the root DPs as well as the entire DPs of the sub-DFG are to be connected to the front-end, a communication links will be created not only between the root of the sub-DFGs at the back-ends and the corresponding DFN resides at the front-end, but also between the entire nodes of the sub-DFGs at the back-ends and the corresponding DFNs at the front-end as shown in Fig. 6.18. The only precondition for the creation of those communication links is that one of the DFNs resides on the main DFG at the front-end and the corresponding one resides on the sub-DFG to be sent to the remote host. As discussed in section 6.5.1, communication links are not necessary for the other scenarios since the common sub-DFGs are copied, or the common sub-DFGs are sent to the same remote hosts. For example, between two sub-DFGs residing on two different remote hosts as shown in Fig. 6.15.

A communication link is created between a DP at the back-end and a DFN at the front-end. This communication link allows a notification from those DPs to its corresponding DFN at the front-end. This allows to build up communication for the request and event triggered measurements. For the event triggered measurements,

it must be possible for the DP resides on the back-end to inform the availability of result data to the corresponding DFN resides on the front-end which may collect the available data. In case of request triggered mode, it must be possible for the front-end to have on demand access to the result values provided by the back-ends.

For this purpose, a new DFN is created on the front-end to be used as a consumer of data provided by the back-ends. Through this DFN, a communication between a front-end and a back-end will be established (hereafter, such DFN will be called connector DFN The parent DP(s) of the root DFN of the sub-DFG resides at the remote host will also be referred to us connector DP(s)).

At the front-end a connector DFN represents the whole sub-DFG resides on the back-end and also performs any desired communication between them at any point of time.

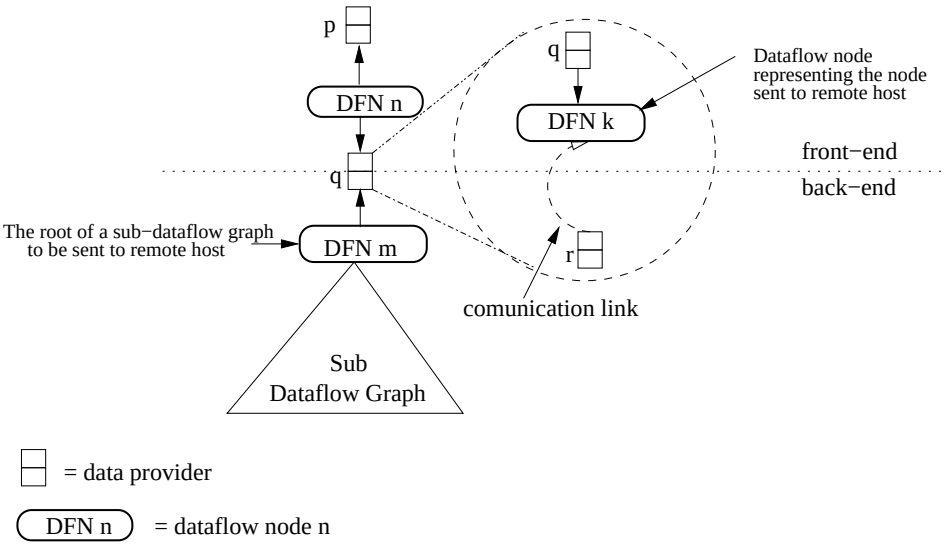

Figure 6.17: Building a communication link between a sub-DFG and the main DFG

As shown in Fig. 6.17 and heretofore we have the relationship

$$DP_q = C_{dp}(DFN_n) = P_{dp}(DFN_m)$$

before a communication link is created. After creating the communication link, we have the relationship

$$DP_q = C_{dp}(DFN_n) = P_{dp}(DFN_k)$$

at the front-end and

$$DP_r = C_{dp}(DFN_k) = P_{dp}(DFN_m)$$

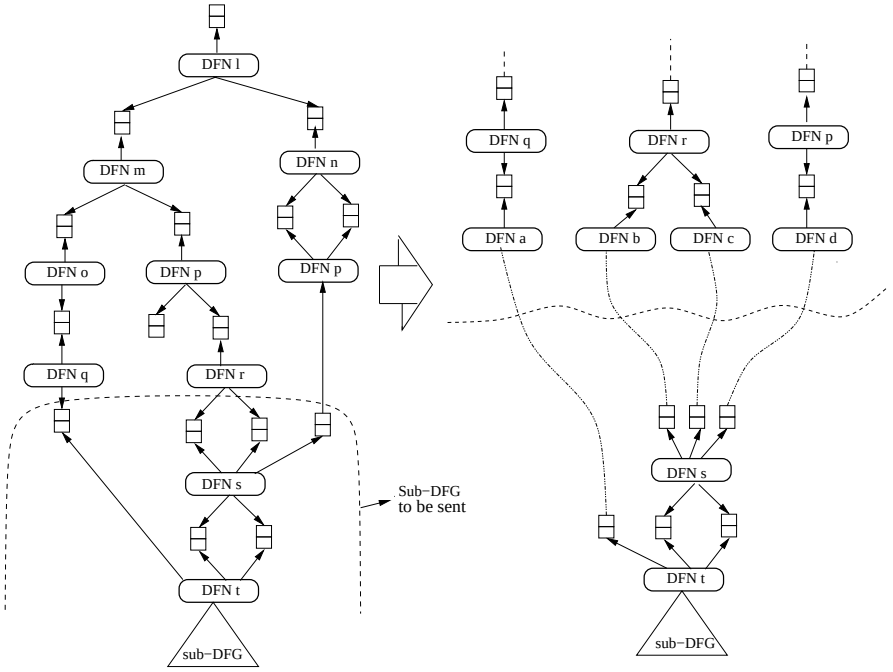

Figure 6.18: Multiple communication links

at the back-end where DP_r and DFN_k are the newly created DP and DFN, respectively, and their parent-child relationship is realized through the newly created communication link.

For the communication between the front-end and the back-end, the DP_r registers the DFN_k as a callee object, which will be used by the callback function. Only through this callee object, a DP which resides on the back-end and must setup a communication link towards to the front-end will find its appropriate connector DFN.

In order to evaluate the sub-DFG, when the whole DFG is launched, all the connector DPs will be read at the same time. To enable this process, all connector DPs of a sub-DFG are collected together. This avoids not only traversing the whole DFG during the result reading process, but also makes it easier to manage the OCM-G requests which will be discussed in detail in section 6.7. This speeds up also the whole computation since it provides all possible result values at the same time.

The same procedure will be used to set up a communication link between the nodes which are entire nodes of a sub-DFG. This is shown through the communication between DFN_t and DFN_q in Fig. 6.18 for which a connector DFN_a is created. The connector DFNs $(DFN_b, DFN_c\,and\,DFN_d)$ are connected with the three root connector DPs of the $SSDFG(DFN_s)$ which represents the sub-DFG rooted at DFN_s (hereafter, $SDFG$ will be used to represents the set of such sub-DFG residing at the same remote host as discussed in 3.3).

During these processes and specially through generating the bidirectional connections, the $SSDFG(DFN_s)$ is disconnected from the main DFG. That means that the partition of the $SSDFG(DFN_s)$ is performed automatically during the creation of the bidirectional connections between the front-end and the back-end. As shown in Fig.6.17 and 6.18, the newly created DFN is a type of node used to collect measurement results, and whenever result data is available at the connector DP, a communication link will be setup to have an access on those data.

6.6 Distributing Subtasks

The *SSDFG* to be sent to the back-ends are now determined and also detached from the main DFG but reside still on the front-end and should be sent to their appropriate remote host. In order to achieve this goal, those *SSDFG* should be serialized and sent via the network. Arriving at the remote host, those DFGs must be de-serialized to reconstructed the desired DFG again. For this purpose, OCM-G plug-ins are developed which provide the desired functionalities to perform the necessary tasks and the OCM-G is programmed to use them as desired.

The marshaling process of a *SSDFG* is done by parsing and creating its string representation. The parsing process encodes not only the DFNs and DPs but also the necessary attributes attached to them. In addition, all the information used to connect the *SSDFGs* with the main DFG is encoded. Those attributes include parameters identifying the requester of result values and virtual times in order to associate the correct result values to the corresponding requester. In order to build up the string form of the *SSDFG*, every DP and DFN, starting from the root node of the *SSDFG*, will be requested to provide the string form of its components and the important attributes belong to them.

Every DP, for example, will be encoded with the parameters which describe its unique identifier, the type of the DP and the requester identifiers it possesses. For the DFNs, there are additional attributes like the type of operators for the operation and aggregation nodes.

6.6.1 Plug-ins to Transfer the Sub-DFG

As mentioned earlier, for the whole communication between the GPM tool and the underlying monitoring system, the OMIS standard interface is used. In order to send the *SSDFG* to their remote host, the OCM-G is extended with two plug-ins. Transferring of a sub-DFG is performed in two consecutive steps: creating an empty *SDFG* and adding *SSDFG* to that *SDFG*. The creation of an empty *SDFG* will be performed only once for every back-end. This *SDFG* will collect the root DFNs of all *SSDFGs* sent to that back-end.

OMIS requests to create a sub-DFG

The only parameter needed to create a *SDFG* at the remote by the OCM-G plug-in is a unique identifier of the remote host which looks like $n_{1,}, n_{2,}, \ldots, n_{x,}$. where x stands for the number of hosts used for the computation. The immediate result of the ***OMIS*** request after the creation of a sub-DFG at the remote host is a token, which represents the created DFG within OCM-G. This token is used to manage (localize, delete, ...) the created *SDFG*.

This token, together with a host identifier, will be stored globally at the front-end and used also to indicate the existence of an empty DFG in the specified remote host. The connector DFN, as discussed previously, will also store the *SDFG* token belongs to the host where the corresponding connector DP is residing. In order to add a *SSDFG* to a *SDFG* at the remote host, its string representation is used by the OCM-G plug-in. After arriving at the destination host, its string form will be parsed back to reconstruct the *SSDFG*.

During the creation process of the *SSDFG* at the remote host, a map data structure is used to map the pointers to the DFG component against their unique identifier. This avoids handling parts of the same sub-DFG multiple times. This method is also used to join the common sub-DFGs of the *SSDFGs* sent to the same destination host as shown in Fig. 6.15. If the *SSDFGs* have no DFNs in common, i.e., if they are disjoint, they will evaluate the result values independently and send the results to the corresponding consumer residing at the front-end as shown in Fig.6.19. In order to have a full control of the all *SSDFGs*, their root DFNs are collected in a list data structure as depicted in Fig. 6.19, and the root DFN of the *SDFG* have an access on this list.

Fig. 6.19 shows

$$SDFG(DFN_l)$$

and

$$SSDFG(DFN_i) \text{ where } i \in \{l, q, s\}$$

where $SSDFG(DFN_l)$ is at the same time $SDFG(DFN_l)$ since it has a pointer on the list containing all $P_{dp}(DFN_i)$ where $i \in \{l, q, s\}$.

6.7 Generating Monitoring Requests

In order to be able to monitor the application and access the monitoring result values, the performance measurement tool must request the monitoring system for the desired information. In this subsection, setting up the requests to access the desired computed data from the underlying monitoring system will be illustrated. Depending on the specified metrics, there are two types of measurements: request

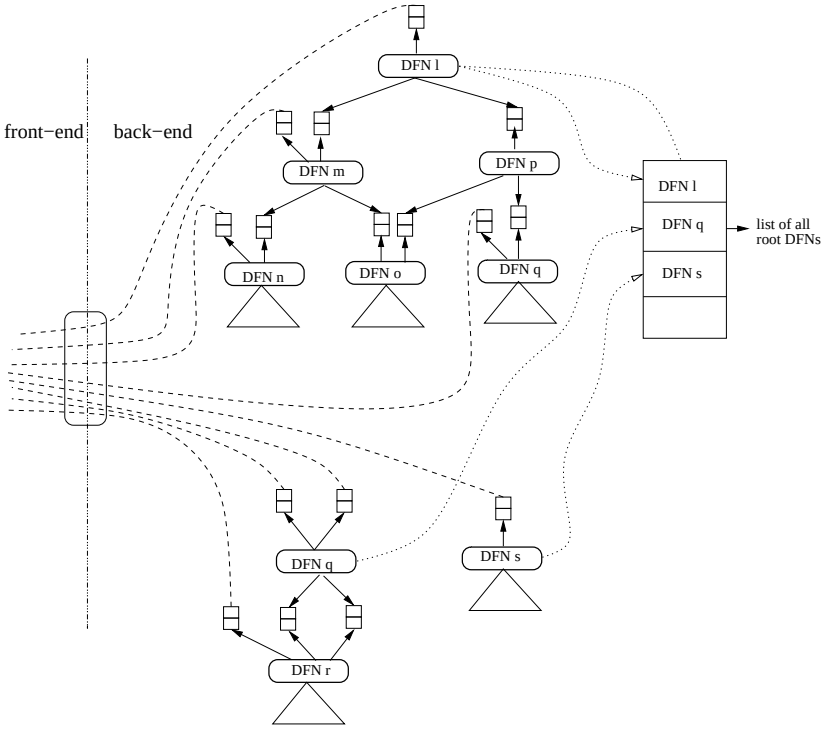

Figure 6.19: Disjointed *SSDFGs* on a single remote host

and event triggered. While generating the monitoring request for the request triggered measurement is simple, in this section the event triggered measurements are discussed briefly.

Request Based Measurements

The very first important component used to generate the monitoring request for the request triggered measurements is the consumer DFN of the result values. This DFN is one of the components of the active measurement object created during the definition of the measurement. Whenever the request to read the results of such measurements is performed, the corresponding active measurement reads the underlying built-in metrics which in turn provide the results to the consumer DFN using the specified callback function. The importance of this arises from the fact that the reassembling of the result values is performed asynchronously as shown in Fig. 6.22. The result value will be forwarded automatically using the dataflow technique.

Event Based Measurements

For event triggered measurements, the result values have been determined by sampling the specified probe. As a result, whenever an event occurs, the corresponding

action will read the required results and forward those data asynchronously to the correct consumer DFN. As we have seen in sub section 5.2, request strings are used to describe the requests which will be used as a parameter for the request functions provided by the OMIS-Interface [31].

In order to deal with event triggered measurements, an OCM-G request must be based on event/action paradigm of the OCM-G. The OCM-G plug-ins developed for this purpose accept string request, which describe the event to be detected and a list of actions to be performed. The events are identified by their names as they are defined by the user in the application code. The actions to be performed when the corresponding event occurs are OCM-G functions. Before defining such event based requests, we need objects which are going to be used as a consumer of the data replied by the OCM-G plug-ins. In our case, those data consumers are DFNs. Thus, a DFG token identifying a DFG where a specified DFN resides and an identifier of that specific DFN are desired to define the proper request string. This is the main reason why it was not possible to define such OCM-G requests before the partition of the main DFG which provides us the desired DFG token of the SDFG.

6.7.1 Event Based Actions

In order to associate the results to the correct consumer DFN, some additional information must be provided. That information includes the identifier of the requester and placeholder variables for the virtual time. The former is used to propagate the result values to the proper consumer. Since the same consumer DFN can be used to handle result values coming from different monitoring objects, which will be propagated further to different requester, a unique identifier of a result value used. On the other hand, the virtual time is used by the event triggered measurements to associate result values of different events monitored at the same virtual time. Since the virtual time is determined by the OCM-G, a placeholder is used during the requesting process.

Since many different actions can be triggered by a single event, the occurrence of such an event can thus results in performing multiple actions. In order to manage the entire event triggered measurements in a single measurement, the probe nodes dealing with those measurements will be collected. According to the locations where those actions are going to be performed, two different lists are used to collect the actions to be performed locally or globally. Global actions are actions to be performed on the front-end whereas local actions will be performed at the back-ends. The collected probes are used to create the corresponding event triggered request. Defining a probe event and the corresponding action(s) to be performed will result in a Conditional Service Request (CSR) token. That means that a probe node will have exactly one conditional request token through which the corresponding active measurement will controlled of event triggered measurements. In order to enable, disable or delete event triggered measurements, an OMIS request will use this conditional service request token.

Every CSR contains a condition and a list of actions to be performed. The condition for each probe is their execution in the application code. Generally, the request string for such conditions looks like as follows:

$$thread\_executes\_probe([< Host >], " < ProbeName > ")$$

This condition will be fulfilled when a probe with the name ***ProbeName*** is detected on a host specified by the host token ***Host***. A user can specify some more action requests which will be appended to this request string.

Depending on the specification of the metrics, all actions that are either event or request triggered may be performed at the front-end and/or at the back-end. The following subsection shows how the OCM-G requests for the user defined measurements will be constructed. Since their requirement is different, the requests for the front-end and back-ends are handled separately.

6.7.2 Types of Actions for the Distributed Evaluation

OCM-G is extended to support the actions needed for the distributed evaluation. Fig. 6.20 shows that a user defined measurement can consist a list of probe elements collected together to enable an efficient implementation. Since the results are provided asynchronously, each action has in turn a callback element which performs one of the reading actions listed. The callback object containing all information that are necessary to perform the desired actions will be inserted to the action list. This action list may be extended with another type of actions belonging to the same event as shown below. All those actions will then be performed whenever the corresponding event occurs.

Depending on the type of measurements, there are four different types of actions to be performed by the OCM-G plug-in: assigning a constant numerical values, returning application specific data, reading measurement results provided by the built-in metrics, and reading virtual time aggregated result values. Those activities cover all the requirements specified by the metrics for the event as well as for request triggered measurements.

To assign the constant value when a corresponding event occurs, additional information describing the requester identifier plus a consumer DFN which will be used as a callback object are needed. To deal with the application specific data which are going to be returned when the corresponding probe occurs, a callback element will be created for every specific data of the application to be returned. The information needed by the callback object will then be inserted to the corresponding callback element. Following this, the callback element will be inserted to the list of callback elements which belongs to this probe event as shown in Fig. 6.20.

Collecting part of the information needed to read the measurement results will also be performed in a similar way. The active measurement object is one of the important additional information desired in this case. If the underlying built-in

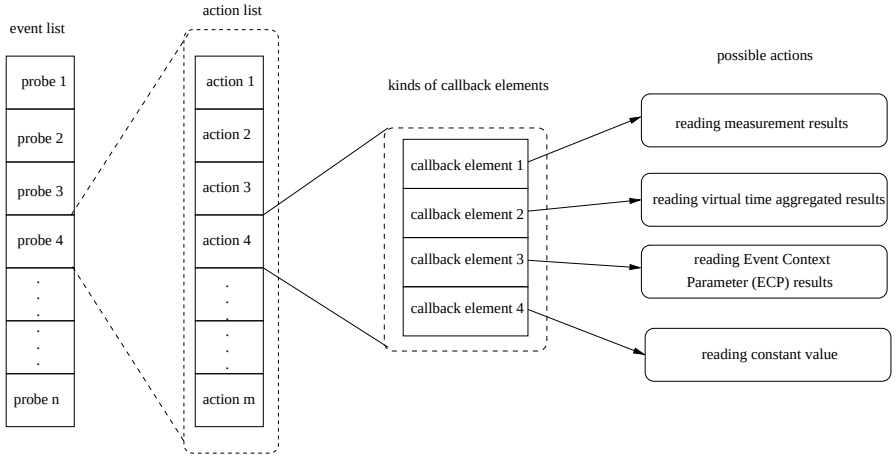

Figure 6.20: List of probe elements and the corresponding actions

metrics provide service information, that information will also be specified in the request string as shown below. For every measurement represented by the measurement result DFN, there are two ways to access the result values from the local monitoring system, which depend on the integration mode used. That service information can either be reading an integrator or a counter.

Those services and the corresponding parameters are shown below:

- To read a counter:
  - Service: $patop\_measurement\_counter\_read$
  - Parameter: $[< applicableObjecID >] < aggregationmode >$
- To read an integrator:
  - Service: $patop\_measurement\_integrator\_read$
  - Parameter:$[< applicableObjecID >] < aggregationmode >$

Both services are going to be applied on applicable objects by taking the given aggregation mode into consideration. Depending on the specification, the request describing the service and the aggregation mode will be added to the global action request. In this case the callback element to be created contains the active measurements, the number of such entries if they contain information for the monitoring system describing which service will be used, and the requester identifier.

A callback element will also be created for each of the virtual time aggregated data to be read whenever their corresponding event arises. In this case, the virtual time aggregated DFN and the requester identifiers are desired to setup the corresponding OCM-G requests.

6.7.3 Requests for the Actions to be Performed at the Front-end

Some measurements can only be evaluated at the front-end since they are results of some distributed measurements which must be aggregated at the front-end, or they are collecting single results which must have been evaluated at different hosts and sites. Depending on whether the measurements are request or event based, there are different requests to be performed.

Usually, request triggered measurements consist merely of built-in metrics and constant values which will eventually be combined or aggregated. The measurement requests to be sent to the OCM-G regarding request triggered measurements, therefore, consists information which are essential to read and forward the specified built-in measurement. That information includes an active measurement, which contains the built-in metrics to be measured and the consumer DFNs, which will be used by the callback object. This information is sufficient to compute the results for the built-in measurements at the front-end. For every built-in measurement an active measurement object will be created, which will be used to access the measurement results of those metrics.

For event triggered measurement to be evaluated at the front-end, the following request will be appended to the event request. This provides some important information from the underlying monitoring system whenever an event occur which will be described as follows:

$$: \mathit{print}([\$\mathit{time}, \$\mathit{par}_0, \$\mathit{par}_1, \ldots, \$\mathit{par}_n])$$

The first parameter ($\$\mathit{time}$) returns the timestamp, the second one ($\$\mathit{par}_0$) specifies the virtual time of the probe to be returned and the rest of the parameters (from par_1 to par_n), which are called Event Context Parameter (ECP), return application specific information provided by the probes as they are programmed in the source code of the application used. Those parameters returning the application specific data are used only when the probe is defined to return them, whereas the first two parameters will be returned by default.

To send a conditional request of an event based measurement to be evaluated at the front-end which may consist a list of actions to be performed, an OMIS-request will be performed. This request string consists of the name of the probe to be detected, the name of a callback function to be called whenever the event arises, and an action containing all the actions to be performed. This interface is described as follows:

$$\begin{aligned}\mathit{Omis\_reply}\ \mathit{csrReply} = \mathit{omis\_request}(&\mathit{requestStering},\\ &\mathit{OMISCallbackInfo} :: \mathit{callback},\\ &\mathit{actionList},\\ &\mathit{OMIS\_WAIT\_FOR\_FIRST\_REPLY});\end{aligned}$$

The immediate reply of this request is a token which will be used to start, stop, or delete those event triggered measurement.

6.7.4 Requests for the Actions to be Performed at the Back-ends

After distributing the SSDFGs representing subtasks, the results computed by those subtasks residing at the remote host will be collected and forwarded to the correct consumer at the front-end. Since the monitoring system can not guess a priori where the monitored data are going to be sent, the request must include information about the consumer DFN in a specific SSDFG. Depending on the kind of result values to be collected, additional information may be required. In general we have the following four cases:

*Handling results values of built-in metrics***:** In order to have an access on the data produced by the built-in measurements, in addition to the information on the location of the consumer DFN, the following information are necessary: information about the service, the requester identifier, and a placeholder for the virtual time.

Handling result values to be aggregated: In order to have an access on the data to be aggregated and residing at the back-ends, the information needed are the same as those which are used to access the results of built-in measurements, except the service information.

Handling constant numerical result values: To have an access to the constant values which are specified by the user during the specification of the metrics at the beginning, additional information is needed which include the constant value itself and a placeholder for the timestamp. This timestamp is not necessary in all other cases since it is provided by the result value delivered by the monitoring system.

Handling application specific date: To provide application specific data delivered by the probe function inserted to the application code, an OCM-G request must be performed for every event context parameter. The event context parameter indicates the index number of the probe parameter, which returns the application specific data. After creating all the necessary requests for a single probe event including the string describing the event condition, the request string will be concatenated and will be sent to the OCM-G. To send an OCM-G request for every probe based measurement, the ***omis_request*** function as shown above is used.

As mentioned earlier, there is always a user defined active measurement for each definition of the specified metrics. All the collected measurement information is stored in this active measurement object. This object is responsible for starting, stopping, or deleting the whole measurement. Since this active measurement is the callee object of the root DP of the main DFG, it will be notified by the root DP to read the result values in the case of event triggered measurements. If the whole measurement is request triggered, then the corresponding active measurement tries to read the result values from time to time, without receiving any notification from the root DP. These two types of requests are used together to enable an automated result token propagation from the monitoring system to the consumer. The part

of the measurement which are event triggered provides the results as soon as the corresponding event arises and propagate the results as far as possible to the desired DFN and then at some point those result tokens will be requested and sent to the root DP where they will be read by the active measurement.

Starting the measurement

After the specified metrics is defined and the necessary OCM-G requests are performed, the user can start the whole measurement. Since a measurement contains a set of sub measurements, starting the whole measurement means starting all the sub measurements of the built-in metric.

Starting the active measurements means that the corresponding counter or integrator of the underlying measurement gets initiated in order to set the start time of the measurements. For the event based measurements, it is enough to enable the conditional service requests as discussed above. Whenever the events arise, the computed data will then be propagated automatically to the requester DFNs. The final measurement result can have results which can be two-dimensional since they can be computed for a list of applicable objects with the corresponding partner objects. The monitoring system provides those results in a serialized form which must be de-serialized to build result token objects.

6.8 Processing the Measurement Result Values

After the measurements are started, the result values can be assembled, which will be performed for both request and event triggered measurements in different ways. For every request triggered measurement, the result values are accessed whenever the user interface at the front-end updates its measurement result values. This updated process is performed by requesting the corresponding active measurement to provide the appropriate result values as shown in Fig. 6.21. The active measurement reads in turn its children active measurements which request the underlying monitoring system for the result values.

In case of event triggered measurement, those result values are getting updated whenever the corresponding event comes up. Depending on the application, those events can occur up to several hundred times a second. In one of the CrossGrid interactive application, as discussed in chapter 2 which is dealing with the flood scenario, it was observed that up to 400 events per second arise.

In both measurement modes, the result tokens go through different aggregation processes on their way to the front-end. Before the aggregation process takes place, synchronization, as shown in Fig. 6.23, will be performed since the results are coming at different times. On the other hand, Fig. 6.21 shows the reading process for two different user defined measurements. If two user defined active measurements are trying to read the same active measurement of a built-in metrics, the last requester will be ignored. Before starting the reading procedure, the

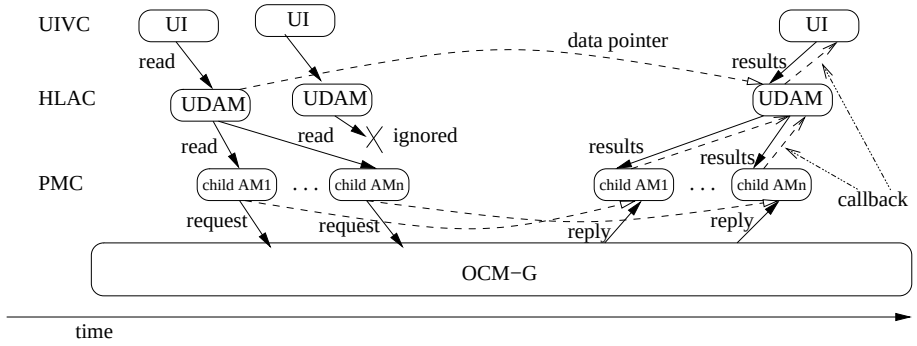

Figure 6.21: Reading the user defined active measurement and its children active measurements

information about the caller active measurement will be stored in a list if the measurement is request triggered. This enables latter that the result will be read for the correct requester.

Since the OCM-G interface library always checks for incoming messages when sending a request, the execution of the callback objects can be called recursively while the read measurement call for the previous request is performed. Thus, in order to execute all commands in their order, it is important to put their commands in a global list before starting to execute them which insure their order of execution. In order to put the necessary information needed by the computation of the stored elements, the command itself and additional information, like the time of the measurement, the virtual time, and the result parameters are stored together.

Reading Measurement Results

When a consumer requests to read result values of a measurement, the user defined active measurement representing the whole measurement tries to read the result of all sub active measurements, all virtual time aggregated measurements and then the results provided by the connector DPs. The active measurements are measurement of predefined metrics which are just being measured. The DFG aggregates and/or correlates these results to provide the final result to the consumer. Since all those reading processes are performed asynchronously, we are dealing with an embedded multiple asynchronous calls to get the final result values in automated way as shown in the Fig. 6.22.

Since the active measurements are the inputs for all the computation and are located at the end-points of the sub-DFGs, reading them must be performed before reading the aggregation and the connector DPs. In order to enable event triggered aggregation, the DFN dedicated for the aggregation accumulate the result values till they are requested. Through this process, the DFG processes stream of result

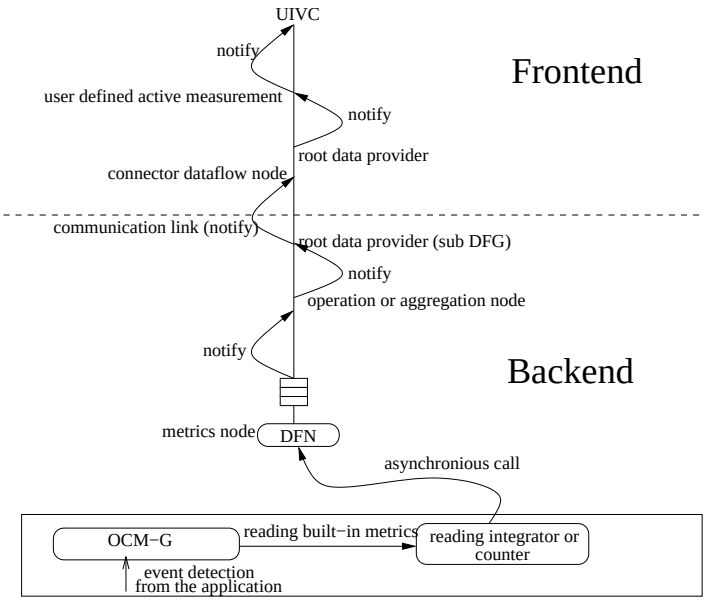

Figure 6.22: Asynchronous calls of the distributed evaluation

tokens as illustrated in Fig. 6.22. Since the final result values of the SSDFGs will be collected at the connector DPs and these connector DPs notify their corresponding connector DFNs to fetch the result values, date are sent from the back-end to the front-end automatically. Lastly, the root DP of the main DFG at the front-end notifies the user defined active measurement through a callback function to read the result values.

If active measurements are going to be evaluated globally at the front-end, the read method of the active measurement will initiate the result evaluation. During this reading process, this method returns no result. Instead, the callback object defined for the active measurement of the built-in metrics during the definition of the user defined metrics will be invoked when the desired results are available. To read the aggregated measurement results computed at the front-end, the read function of the virtual time aggregated DFN itself will be used to generate a result token out of the aggregated values. An OCM-G plug-in function is used to read the aggregated result tokens.

For the event triggered measurements, the occurrence of the events are responsible to activate the whole measurement mechanism and thus to forward the result values. These results are identified by their virtual time value and their requester identifier. Therefore, the result of the event based measurement contains, in addition to the result components provided by request triggered measurements, virtual time and identifier of the requesters. The virtual time value is used to identify measurement results belonging together whereas the identifier of the requester is needed when the same measurement is read by different callback object.

For the request triggered measurements residing on the back-end, An OCM-G plug-in is used to read and forward the results of the child active measurements, which then write the result data to the appropriate consumer DFN directly.

Reading the Connector DPs

Each root DFN of a SSDFG contains in turn a list of all connector DPs. As discussed previously, every SSDFG contain at least one connector DP which is used to build-up a communication link to the front-end. However, every DFN of a SSDFG, including the root DFN, can have multiple connector DPs. To read the data out of those DPs, the connector DFN at the front-end uses OCM-G plug-ins. Such a reading process will be performed in both service request modes: conditional service request and unconditional service request. For event based measurements, the action to read the connector DP will be performed by the DFN at the front-end whenever the connector DP gets its first result, whereas for the request triggered measurements, this will be performed when the DFN at the front-end is requested to provide result values.

Result Assembly

The result replied by the OCM-G, which is going to be forwarded to the consumer connector DFN at the front-end, have the following structure.

$$n, [result_1, result_2, \ldots, result_n]$$

Where n stands for the number of provided results and the result array contain these n result values. Each result $(result_i)$ contains all the necessary values used to compute the result token. This result token is given with:

$$result_i = (startTime_i, stopTime_i, virtualTime_i, requesterID_i, hasVT_i, hasTime)$$

The descriptions of these results is as follows:

startTime:	describing the start time of the measurement.
stopTime:	describing the time when the measurement ended.
virtualTime:	providing the virtual time of the measurement.
requesterID:	providing the identifier of requester of the measurement.
hasVirtualTime:	describes whether a virtual time is available.
hasTime:	describes whether the measurement results has a timestamp.

This result token object will be used to synchronize and evaluate the result values according to the firing rule of the DFN it passes through. Such a DFN is depicted in Fig. 6.23.

For the distributed evaluation, this token will flow through the SSDFG. In general, the serialization and de-serialization of the result token will be performed whenever result values are going to be transferred from the back-ends to the front-end and whenever monitored data are provided by the OCM-G. After arriving at the connector DFNs, they will be again de-serialized to be transferred to the front-end. In this way the result value will be forwarded automatically to the root DP of the main DFG. If the active measurement of the user defined metrics is notified to read the final results, it will read (in case of event triggered measurements) all the available result values form the root DP of the main DFG. For a request triggered measurement, the active measurement will read the results from time to time.

To de-serialized the results, parsing each of the results will be performed in OCM-G according to the defined FORM STRING *"1,1,0,1,[%s,0,0,%s,0]"*. Since the result values are ordered in a matrix form, the first two values indicate the x and y values of that matrix. The third element describes whether there is a virtual time. This is followed by a numerical value describing the number of available results followed by a list providing the result elements.

6.8.1 Evaluation of the Result Values

Those DFNs which are responsible for the aggregation or correlation of result values, like operation DFNs, must be notified to collect the results from their children DPs. This notification occurs whenever a specific DP gets its first result value. After the notification by its child DP, a DFN tries to read a single result token from each of its child DPs continuously until at least one child DP is empty. That means that a DFN waits until each of its child DP receive at least one measurement result value. After the DFN reads the first element of each of its child DP, it synchronizes and computes these result values to provide a single result token as shown in Fig. 6.23.

The result token returned by the monitoring system is given by

$$v = (p, q, r, s)$$

where

$$p = result\ value$$

$$q = timestamp$$

$$r = virtual\ time$$

$$s = requester\ identifier$$

As discussed in section 3.3 the DFN uses its function (f) to compute the input result tokens provided by its child DPs and provides a new result token as shown below.

The functionality of the DFN mentioned above is given as

$$f(I(DFN)) = (p,q,r,s)$$

which is shown below for m result values in the n child DPs ($N(C_{dp}(DFN))$) and $\emptyset$ indicates that there was an error during the computation of the result token.

$$\forall k \in [1,m] : f((p_{1k}, q_{1k}, r_{1k}, s_{1k}), (p_{2k}, q_{2k}, r_{2k}, s_{2k}), ..., (p_{nk}, q_{nk}, r_{nk}, s_{nk}))$$

$$:= \begin{cases} \{p'_k, q'_k, r'_k, s'_k\} & iff\ \forall i,j \in [1,n] : r_{ik} = r_{jk} \\ \{\emptyset, q'_k, r'_k, s'_k\} & iff\ \exists i,j \in [1,n] : r_{ik} \neq r_{jk} \end{cases}$$

where

$$p'_k = NodeOperation(p_{1k}, p_{2k}, ..., p_{nk}) \wedge$$

$$q'_k = average(q_{1k}, q_{2k}, ..., q_{nk}) \wedge$$

$$\exists t \in [1,n] : r'_k = r_{tk} \wedge$$

$$\exists l \in [1,n] : s'_k = s_{lk}$$

As shown above, a final result value for the DFN is computed using the operation specified by the DFN and its timestamp is determined using the average timestamp of all the result values. The value for the virtual time as well as for requester identifier is determined by taking an arbitrary value from one of the corresponding input values.

6.8.2 Synchronization of Result Values

The synchronization process depends only on the virtual time. This means all the result values must have either the same virtual time or no virtual time at all. After a successful synchronization, the functionality of the DFN is used to produce a new result token as discussed above. Fig. 6.23 shows such a scenario for n child DPs and l parent DPs. An empty result value in the parent DPs shows that some of the result tokens are not propagated for all the parent DPs. In this example, the first result token is propagated to all the parent DPs which has the index $11, 21, \ldots, l1$, the second one contains no valid result and is used to report an error to the parent DPs and is not sent, for example, for the second parent DP, the third one with the index $13, 23, \ldots, l3$ is not sent to the first and the last parent DPs, and so on.

The final measurement result of those DFNs will then be forwarded to the parent DPs according to the value of the requester identifier (see also section 6.8.3) of the result value. In case of an error, a new measurement result containing *NAN* as

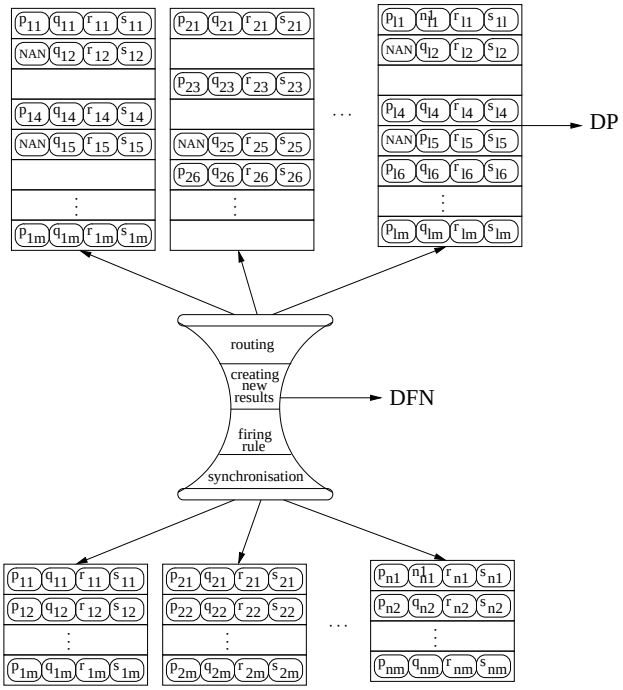

Figure 6.23: Result synchronization

measurement value will be constructed and forwarded to the parent DPs to notify that an error has occurred during the computation of the specified measurement with the specified virtual time value and timestamp.

This procedure is applied only for internal DFNs. DFNs which are end-points and thus have no children DPs forward the measurement results computed at these DFNs without any aggregation or correlation to the parent DP according to their requester identifier. These back-ends compute built-in measurements, like the amount of data sent or received, which read their results using a callback mechanism delivering the results asynchronously.

6.8.3 Routing of Result Values Using Firing Rules

In order to optimize a measurement, the same built-in metrics measurement can provide result values to different requesters at different time. This is the case when an event triggered measurement must provide results for different event occurrence (like the begin and end events) which have the same virtual time.

In order to enable such result propagations, requester identifier assigned when the measurement request is sent to the monitoring system are used, as shown in the Fig. 6.24 where $s_m = 0$ and the requester identifier zero is reserved to be used as a wildcard. That means that the DP possessing this wildcard as requester identification will get every result values from the child DFN propagated. Such a kind of DPs are used to forward the result tokens to the parent DFN which may then

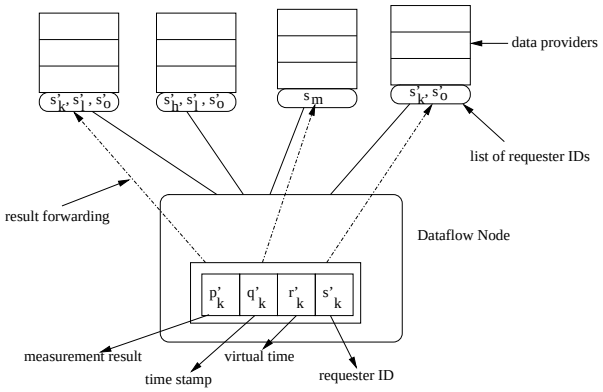

Figure 6.24: Routing the results according to the requester identifiers

propagate the result tokens according to the value of their requester identifier or when the DFN handles results which are not request triggered.

The routing algorithm used as a firing rule in a DFN is then as shown below:

$$
\begin{aligned}
&for(i = 0, i < N(P_{dp}(DFN)), i{+}{+}) \\
&\{ \\
&\quad if\,(((P^i_{dp}(DFN)).requesterID == s'_k) \vee \\
&\qquad ((P^i_{dp}(DFN)).requesterID == 0)) \\
&\qquad\quad (P^i_{dp}(DFN)).push(p'_k, q'_k, r'_k, s'_k); \\
&\}
\end{aligned}
$$

where $P^i_{dp}(DFN)$ describes the i^{th} parent DP of the DFN.

In this way all the results will be evaluated correctly and forwarded to the correct requester of the measurement result token and will be propagated to the root DP.

Chapter 7

Usage Scenarios and Evaluation

7.1 Introduction

In this chapter, we will discuss some usage scenarios, which show the advantage and feasibility of the distributed evaluation discussed in the previous chapter. In contrast to the centralized evaluation, the distributed evaluation reduces the communication between back- and front-ends immensely. Without it, tools like performance analyzer couldn't be used effectively in a dynamic Grid environment where the amount of back-ends, which are grid resources, can be very huge. All the metrics used in this chapter are dealing with event triggered measurements using virtual time since such kind of measurements are mostly communication intensive and need to be evaluated in a distributed way. At the end of this chapter, the tool used in GPM to visualize the intermediate results graphically will be introduced. This Unix filtering is called DOT, which is used mostly to draw the structure of a programs dynamically.

7.2 Usage Scenarios

In this sub section, some usage scenarios will be shown that explain the necessity of the automated, distributed evaluation of measurement data. Most of the example metrics are used in the real interactive applications developed in CrossGrid project. Specially for event triggered measurements, the distributed evaluation shows an enormous reduction of the number of communications, since events can occur more frequently.

To evaluate the performance analysis tool GPM, the interactive grid applications (as discussed in section 2.3.1) are used. One of those applications is a biomedical prototype application representing a system for pre-treatment planning in vascular intervention and surgical procedures. In each iteration of the solver used in this application, ***MPI_Sendrecv*** is used to exchange the results of adjacent processes using a bidirectional ring as a communication pattern which require synchronization of the iterations in all processes. The iteration is structured in such

```
Loop_Counter(Process p, TimeInterval)
: Unit("iterations"){
PROBE iteration_end(Process, VirtualTime);
VirtualTime vTime;
Value value;
val[vTime]=1 AT iteration_end(p,vTime);
return SUM(value[vTime] WHERE value[vTime].time IN t);
```

Figure 7.1: A specification to count iterations within a time interval

a way that a compute and output phase are separated. In order to handle these separate regions differently, three different probes, as discussed in section 6.1, are inserted into the iteration phase. One probe is placed at the beginning, another one between the two compute and output phases, and a third one at the end of the iteration. To identify the occurrence of each probe events, the loop counter of the iteration is used as a virtual time for the probe function. Using those probes, it was possible to specify different kinds of metrics as discussed below.

Fig. 7.1 shows a simple metrics used to compute the number of iterations executed in a loop in a given time interval. Since this specification is automatically translated into proper requests for the underlying monitoring system OCM-G as discussed in section 6.7 which monitor the execution of the specified events, it is possible to evaluate the measurement for each event occurring at the end of every iteration step. The speed of an execution can be estimated by the speed of the iterations, which, e.g., perform activities like communication IO, and so on. Since such measurements may base on the occurrence of events in each iteration, a distributed evaluation in this case means evaluating the results at the location where the events occur. Without the distributed evaluation, the back-ends would just be communicating with the front-end whenever any event occurs. As it is common in an interactive application to have several hundred event occurrences within a second, a centralized evaluation of such metrics would result in having unmanageable front-end.

If such metrics are going to be measured on different sites, and if the measurements are started at the same time, one can use a bar graph to see which site is suitable to run the application by comparing the speed of the executions which is expressed by the number of iterations executed in a give period of time. One of the problems in a grid environment is that the users have no knowledge about the performance of the provided resources for their specific applications. Using such metrics, the users can inspect which resources are suitable for their application by executing a small part of the application on all available resources for a short period of time, provided that the loop in focus is going to be executed in this execution time. This functionality can also be implemented in to grid information providers like the Monitoring and Discovery services (like MDS4 provided by the Globus Toolkit (as discussed in section 2.2.4)) or by meta schedulers (like the GridWay) to

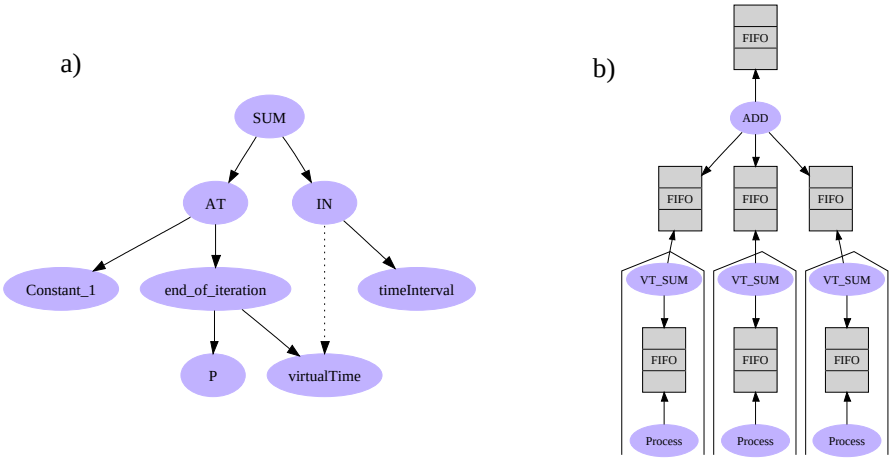

Figure 7.2: Automatically generated graphs used to show the distributed evaluation

achieve an efficient scheduling of jobs. This can be used to support the automatic allocation of the appropriate resources for a given application.

Distributed Evaluation of a Loop Counter Measurement

In order to achieve the distributed evaluation for the loop counter metrics above, the GPM tool creates a template DAG and a DFG out of the specification as shown in Fig.7.2a and 7.2b, respectively. The template for the DAG is used during the definition of the measurement and the DFG is then used to evaluate the distributed computation. In this example, the DAG is defined for three different processes residing on same site and node. As it is shown in Fig. 7.2, when the measurement is started, the number of iterations executed will be stored at the ***VT_SUM*** DFN residing at the back-ends, and will not be sent to the front-end until an explicit request is made by the front-end. For example, if 400 events occur per second and the update interval of GPM is one second, then the number of communications between the front-end and a back-end will be reduced from 400 to one in every second. This situation is illustrated in Fig. 7.3 which shows that for a single back-end in a centralized evaluation the number of communications increase rapidly when the number of events increases.

For the distributed evaluation, the increase in number of events executed with in the given update interval doesn't affect the number of communication since it depends only on the number of nodes and sites used for the evaluation of the metrics as will be discussed below. That means that the number of communication stays constant despite the fact that the number of event occurrence increases rapidly.

Using the distributed evaluation, the reassembling process depends only on the size of the update interval. Depending on the specification of the metrics used, a number of nodes and sites can be specified for every measurement which may increase the communication exponentially if a centralized evaluation of the measure-

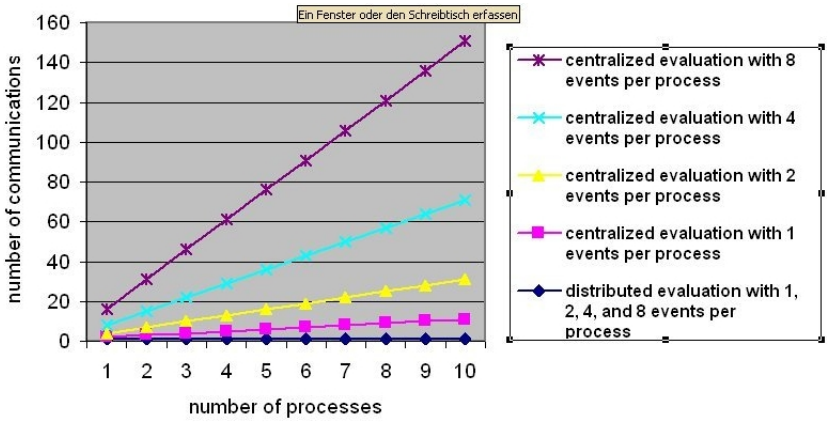

Figure 7.3: A comparison of distributed versus centralized evaluation of a measurement using multiple processes in a singe node

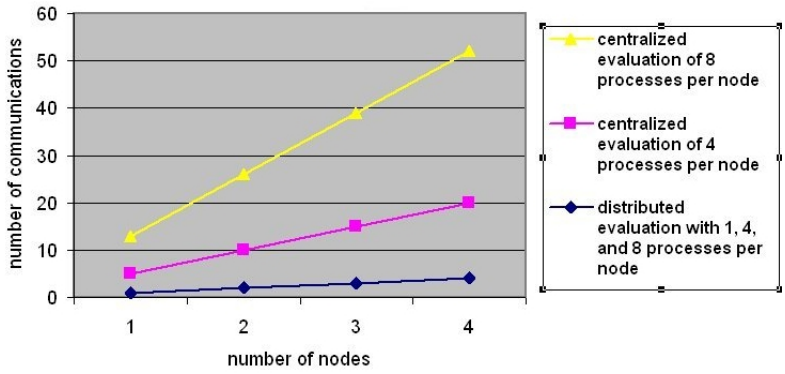

Figure 7.4: A comparison of distributed versus centralized evaluation of a measurement using multiple nodes in a singe site

ment results is applied. Since the distributed evaluation presented here supports not only process oriented but also node and site based distributed evaluation, the best scalability is achieved by the distributed evaluation approach when different nodes and sites are used. Fig. 7.4 shows that the same measurement is performed on multiple nodes in a single site whereas Fig. 7.5 shows the advantage of the distributed evaluation, when different sites are used for the measurement evaluation.

Aggregating Results

While the metrics in Fig. 7.1 describes a measurement which simply counts the iteration in the given time interval, a more higher level metrics is described in Fig. 7.6. It specifies a metrics used to compute aggregated result values describing the value of the mean time spent by each iteration. This mean time is calculated for all iteration within the specified time interval. In the metrics specification, the probe

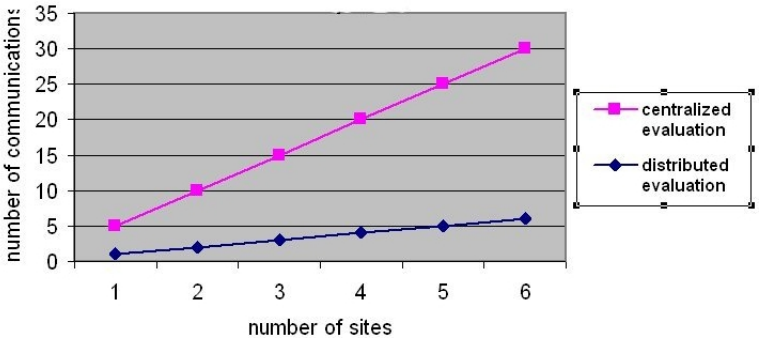

Figure 7.5: A comparison of distributed versus centralized evaluation of a measurement using multiple sites

```
Iteration_time(Process p, VirtualTime vt)
{
  PROBE iteration_end(Process, VirtualTime);
  return Time(NOW) AT iteration_end(p, vt)
       - Time(NOW) AT iteration_end(p, vt-1);
}
Mean_iteration_time(Process p, TimeInterval time)
{
  Value[] it_time;
  VirtualTime vt;
  it_time[vt] = Iteration_time(p, vt);
  return  SUM(it_time[vt]
            WHERE it_time[vt].time IN time) /
          COUNT(it_time[vt]
            WHERE it_time[vt].time IN time);
}
```

Figure 7.6: A specification describing a mean iteration time

used in the previous example which is placed at the end of the iteration loop is used again. A pre-defined metrics ***Time***, which returns the synchronized clock time is used to compute the elapsed time. The ***Iteration_time*** metrics uses a shifted virtual time in order to have an access to the results of a previous iteration.

As shown in Fig. 7.7 the ***VT_SUM*** DFN accumulates the elapsed time for each iteration while the ***VT_COUNT*** DFN counts the number of iterations within the given time interval. Since both of these functions are applied on the same kind of objects, one of the optimizations implemented for the DAG convert the DAG so that both of them point to the same DFN. The same kind of optimization is also applied for the shifted virtual time. As a result of this, a single DFN will be used to provide the computed results for both computations as shown in Fig. 7.7b. In order to distribute this computation, those SDFG will be sent to their corresponding remote host and only the result value will be sent to the front-end. In order to find

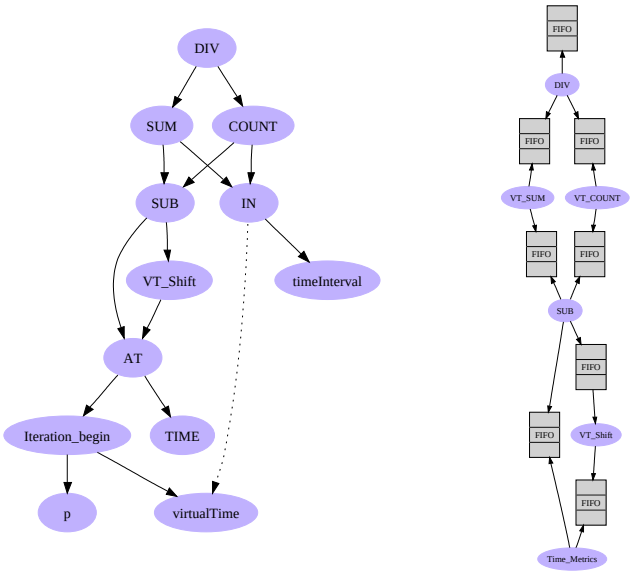

Figure 7.7: A template and its corresponding DFG used to compute the mean time spent by each iteration

out which site has the smallest mean time to compute the same loop, either the same metrics may be defined for each site or another aggregation ***MIN*** can be performed on the results of the computed metrics coming from different sites. In the former case, the user can compare the results by creating, for example, a bar graph of this measurement for each of the hosts used.

Another more sophisticated use of metrics specification and distributed evaluation is demonstrated in one of the CrossGrid interactive applications which are used to prevent flooding of international rivers through flood forecasting mechanism (see section 2.3.1). In the so-called DaveF application, a probe inserted at the end of the time loop is again used to trigger the request for the monitoring system. This measurement demonstrates a measurement resolution in time for each virtual time as well as in location for each process as discussed below.

The metrics ***Comm_delay_per_loop_execution***, as depicted in Fig. 7.8, is used to determine the percentage of the time spent for sending and receiving messages in a single iteration represented by a virtual time. This is determined by subtracting the send and receive delay of a previous iteration from the current iteration. Using this metrics, the metrics ***Max_comm_delay_per_loop_execution*** determines first the maximum delay of an iteration executed in each given process, and then the maximum of the delays in all the iterations executed in the processes in focus within the given time interval. Finally, a single result value will be returned by the second metrics which represents the maximum delay from all the iterations executed in all the processes. A similar situation is also illustrated in Fig. .7.11.

```
Comm_delay_per_loop_execution(Process p,
                              VirtualTime vt) {
  Value time = Iteration_time(p, vt);
  Value commtime =
  (Send_delay(p, [START,NOW])+ Receive_delay(p, [START,NOW]))
             AT Iteration_time(p,vt)
- (Send_delay(p, [START,NOW]) + Receive_delay(p, [START,NOW]))
             AT Iteration_time(p,vt-1);
  return commtime/time*100;
}
Max_comm_delay_per_loop_execution(Process[] procs,
                                  TimeInterval t){
   Process p;  VirtualTime vt;  Value[] val;
   val[vt] = MAX(Comm_delay_per_loop_execution(p,vt)
                 WHERE p in procs);
 return MAX(val[vt] WHERE val[vt].time in t);
}
```

Figure 7.8: Metrics to compute the maximum communication delay

From this metrics specification, a template as shown in Fig. 7.9 is used for the measurement definition. In order to present a simple DFG created from this template, the measurement is defined only for two processes residing in two different hosts. The generated DFG is shown in Fig. 7.10.

That means that the measurement will be performed for all applicable objects and all virtual times laying within the measurement interval, as shown in Fig. 7.11, in which case there are two virtual times (1 and 2) between the measurement interval of a measurement performed for four processes (P3 - P6). In this case, measurement result values will be computed only for the two virtual times and for the four processes if the time interval and the applicable objects are defined so. Special cases where parts of the virtual times do not lie in the measurement interval are discussed in chapt. 6. Since every metrics specification describes only a measurable quantity and does not imply any measurement, a measurement definition of the specified metrics should follow.

As it is shown above, a high level user defined metrics can be used not only to inspect the performance of the application but also to have an insight in to an application's behavior.

7.3 Automatic Visualization of DFGs Using DOT

In order to provide an insight to the intermediate representation which can be used to check whether metrics are specified as desired and to support debugging between the steps of specifying the metrics and distributing the subtasks, a graphical output is generated automatically. Those graphs are created using the Unix filter called

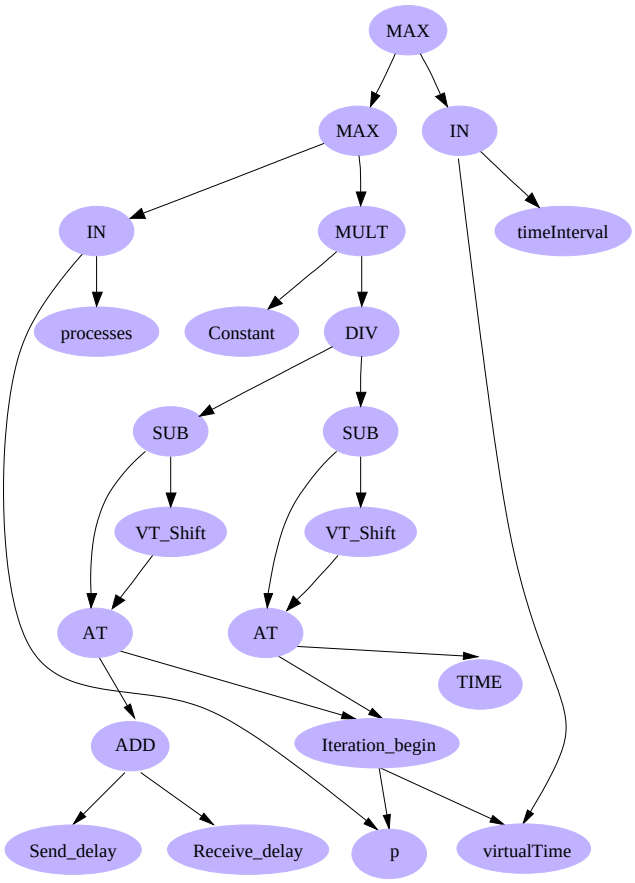

Figure 7.9: Template created from the specification in Fig. 7.8.

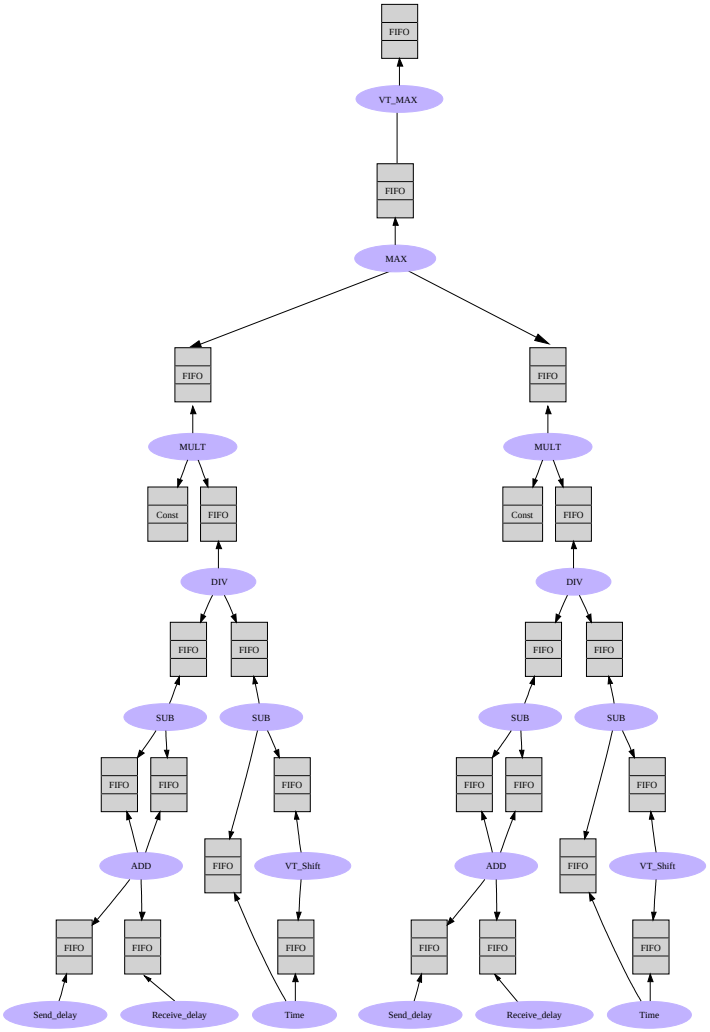

Figure 7.10: DFG representing used to calculate the maximum communication delay

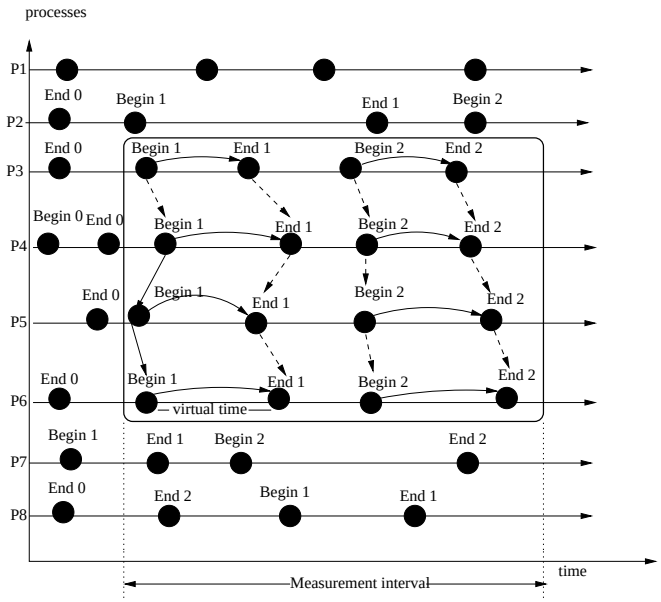

Figure 7.11: Supporting computation for selective applicable objects in an interval of time

DOT[1] . Using a specification language to describe the objects of the graph, DOT enables to draw different kinds directed graph hierarchically.

The output format of DOT varies from GIF, PNG, and SVG to Postscript. Using DOT, one can visualize DFGs, DAGs, cluster layouts, data structures and so on. DOT is provided as command line program and web visualization services. In addition, it runs also with a compatible graphical interface. Since the DOT layout supports only acyclic graphs, it removes all cyclic edges in the specification by reversing their internal direction before the output is generated.

DOT describes all drawings using the following three objects: graphs, nodes and edges. Using the object graph, different sub graphs can be represented in the main graph. The attributes used in the DOT file describe the type of graph (directed or not), size, colour, and so on. Nodes are created when their name first appears in the DOT specification file and edges are created when nodes are connected by the edge operation "$->$". A simple command line to create a postscript file *outputGraph.ps* from an input file *inputGraph.dot* looks like:

```
dot -Tps inputGraph.dot -o outputGraph.ps
```

While Fig. 7.12 shows an input file written in DOT language, Fig. 7.13 shows the corresponding generated graph.

The size attribute in line 2 of Fig. 7.12 controls the size of the whole drawing which will be adjusted when the drawing is too big. The edge in line 4 is drawn as

[1] http://www.graphviz.org/

```
1:  digraph G {
2:  size ="4,4";
3:  A [shape=box,label=head, color=orange];
4:  A -> B [styl=dotted, dir=re];
5:  A -> C;  C -> {E;F;G} [color=red,label="child"];
6:  E [shape=polygon,sides=5,label=center];
7:  F [shape=polygon,color=green,style=filled];
8:  A -> F [label="child \n of \n head"];
9:  G [shape=invtriangle]; B [shape=Mdiamond]; B->G
10: node [shape=record,color=blue];
11: I [shape=polygon,skew=.4,label="grand child"];
12: E->J; J [label="left|{up | down}| right"];
13: K [ width=0.02,label="{ F | I | F | O}"];
14: G->K; F->I; I->I; A->J; }
```

Figure 7.12: An example DOT specification which shows its most important features.

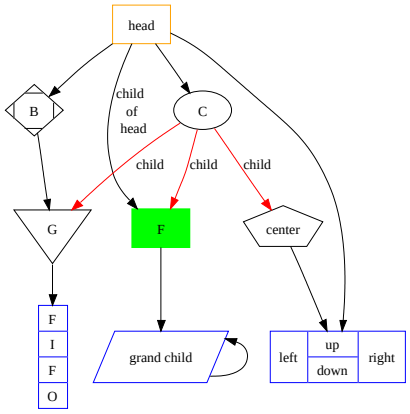

Figure 7.13: An automatically generated Graph using DOT as specified in Fig 7.12

a dotted line whereas all the three edges in line 5 are labelled with the same string ”child” and all having red colour instead of the default black one as described by the corresponding attributes. Using the node shape record, nodes with embedded boxes can be generated as shown in line 10, 12 and 13.

7.3.1 Visualization of the IR Graphs of GPM Using DOT

DOT is used to draw all intermediate graphs generated by the high level analysis component of GPM. This includes the DAG, the main DFGs and the SDFG. The header of the DOT file describing the output of those graphs in the root node of the graphs. This header contains all the attributes which are going to be used for all graphs. After generating the header descriptions, the nodes and edges are drawn recursively.

```
1: digraph G {
2:   graph[ ...];
3:   node [...];
4:   subgraph DAG {
5:     // attribute of the subgraph and
6:     // recursive call to draw DAG components.
     }
7:    subgraph DFG_1 {
8:     // attribute of the subgraph and
9:     // recursive calls to draw all DFG components
     }
10:    subgraph DFG_2{
     }
11:     subgraph subDFG_1{
     ...
     }
12:     subgraph subDFG_n{
     ...
     }
```

Figure 7.14: Content of the DOT specification for a single measurement metrics

Since a single DOT specification file can be used to draw different graphs as sub graphs, the DOT specification file as depicted in Fig. 7.14, is used to provide the desired intermediate representation in the GPM tool. All DAGs and DFGs of a metrics are specified together in one file using the DOT option for sub graph as shown in Fig. 7.14.

In line 2 and 3 the attributes for all the graphs and nodes are described, respectively. In line 4 the sub graph for the DAG is presented. Using the recursive call in line 6, the desired information for the nodes and edges of the DAG will be collected. Similarly, the recursive call to draw the main DFG of the given metrics is described at line 9 which provide the information for the DFNs and for the data providers belonging to the main DFG. For the main DFG after the distribution, the sub graph of the DOT at line 10 is used. Starting from line 11 on, the specification for SDFGs will be generated.

Chapter 8

Summary and Outlook

8.1 Summary

The main aim of Grid computing is to enable a coordinated resource utilization, which are geographically distributed and controlled under different administrative domains. As an example, EGEE is the first worldwide distributed Grid computing environment consisting of several institutions from around 50 countries worldwide. At present, the number of processes in EGEE reaches more than 100, 000 (with 15 PB storage ca 200 Grid sites) and is used to provide a reliable worldwide Grid infrastructure mainly for researchers. Recently, EGEE hits 100,000 jobs per day.

The trend towards extremely large scaled Grid computing environments is also expected to continue. That means that scalability of the available parallel tools will be more important in the future. For instance, measuring the performance of applications running on such heterogenous and dynamic environments becomes more challenging. Unfortunately, there are few runtime tools used for such modestly sized computing environments which can be used to solve very complex problems. This architectural shift from few to many computing components is causing tool designers of high performance systems to revisit numerous design issues to improve the scalability and manageability of their parallel tools.

Only accessing an application faster will not be a solution in the future, since visualization and managing the data produced from the execution will be a very important factor. To achieve a high-performance computing in such environment, the distributed evaluation facility of parallel tools is presented in this thesis. The solution presented in this thesis minimizes the cost which comes from the intensive network activities by reducing it through enabling an evaluation of the computation as local as possible to avoid the unnecessary communications between the front- and back-ends. This scalability work supports a distributed strategy to evaluate application-specific as well as Grid infrastructure based measurements by reducing the cost of the computation used for data analysis between front-end and back-ends.

In this thesis, the distributed evaluation is evaluated for a performance analysis tool where the design and evaluation of performance measurement data in an

automated, distributed way was the basic challenge of the thesis. The proposed mechanism of the distributed evaluation supports a large number of geographically distributed processes by providing an efficient control mechanism for large number of back-ends using a dataflow approach. The technique used in the dataflow model fosters not only a scalable computation of tasks, but also an efficient communication with a maximum of logarithm complexity $\mathcal{O}(log_2^n)$ for the data aggregation.

By supporting an efficient reassembly of the distributed data asynchronously in parallel, the tool's centralized activity is reduced immensely. This avoids also resource saturation at the front-end. Using the provided mechanism, even a program phase based measurement evaluation was possible which can also be used to determine a local as well as a global behavior of applications with a large number of processes. In order to tune the complex interactive grid applications mentioned in section 2.3.1, a manageable performance analyzer was indispensable which could be achieved by using the proposed distributed evaluation of the monitored data.

The distributed evaluation discussed in this thesis includes the description of how an intermediate representation (IR) in form of DAG is created from the metrics specification written in PMSL, which is evaluated when the measurement is defined. Following this, the process of generating DFGs from the DAG is performed. The DFG is then partitioned to create the sub-DFGs representing the subtasks. This sub-DFGs are then sent to their corresponding hosts according to the metrics definition. To aceive this goal, the OMIS interface is used to create a proper monitoring requests to assure the data assembly. During the runtime reassembly, synchronization and evaluation of the final measurement results are performed. All this is done for the request triggered as well as for event triggered measurement evaluations. Specially, for event triggered measurements, evaluation of the computation at the location where those events are detected was very necessary to guarantee the scalability and manageability of the tool.

Through creating an overlay network using an ADFG, it was possible to create a well manageable front-end. This front-end was usually the bottleneck in many cases. For example, when the number of hosts used increases, which is the case in the grid computing environment, and if the back-ends are sending information to the front-end more frequently which increased the complexity of the computation.

In online measurement tools like the GPM, the measurement results must be provided to the front-end possibly with out any noticeable delay. Therefore, the concept of evaluating a computation as local as possible is the basic idea of this distributed evaluation approach. An ADFG model is used to distribute the subtasks to the appropriate remote location and to reassemble the computed result values automatically and asynchronously. The awareness of the grid environment and supporting online measurements at the same time enables the method developed in this thesis to provide a powerful execution model based on the distributed evaluation. This execution model provides scalable computation processes, which is the very first requirement of distributed online measurement tools.

Since it does not introduce a notable overhead, which might disturb the application execution, the proposed approach does not affect the running application.

Understanding the work in this thesis helps to realize a scalable, distributed computation specially in the field of debugging, performance analysis, load levelers and other related fields including job, resource and application monitoring tools. In addition, the tools which are based on the OCM-G monitoring system can adopt the implementation idea of this work directly.

8.2 Outlook

The discussed ADFG has proven its utility in the real world performance analysis tool and it is also expected that another field of study dealing with stream of data items will benefit from this approach. This kind of solution can be used in different communication patterns, for example, to develop an algorithm supporting hierarchical implementation of the MPI point-to-point operations since the current implementation of MPI does not support hierarchical interconnections.

The distributed evaluation concept as provided in this thesis can be used not only for performance analysis and related parallel tools but also in many applications which usually extract information describing the relationship between the datasets and statistical results. This includes but not limited to tools used for information retrieval on the web [25], data mining used in the field of electronic commerce and intrusion detection [28, 69], and special data base systems [70]. It is also expected that a dataflow approach to be as effective in application domain as it has proven to be in tool domains.

As an improvement, the result token flowing through the augmented DFG, can be presented in a standard format which facilitates the interoperability of the proposed solution with other tools. An XML kind of representation of the result token can, for example, be used to provide information to the MDS4 which is de facto standard monitoring tool in the grid computing environment. In order to support the dynamic nature of the grid, the actual resource information provided by the monitoring services like MDS4 can be used to have an updated information of the resources to be used by GPM. This may avoid using different tools to monitor jobs and resources simultaneously. Another similar existing combinations of Grid monitoring are, for example, MDS4/ganglia and MDS4/Nagios. GPM together with MDS4 would even be the best solution to provide application specific information combined with resource information.

Bibliography

[1] Wismuller, R., Bubak, M., Funika, W., and Balis, B.: A Performance Analysis Tool for Interactive Applications on the Grid. International Journal of High Performance Computer Applications, Fall 2004, SAGE Publications.

[2] R. Wismüller, H. Mehammed, M. Gerndt, and A.Bode: "Performance Monitoring and Analysis for the Grid". In Engineering The Grid: Status and Perspective, from Beniamino Di Martino among others, American Scientific Publishers, 2006

[3] I. Foster, C. Kesselman: The Grid. Blueprint for a new computing infrastructure. Morgan Kaufman, 1998

[4] M. Alef, T. Fieseler, S. Freitag, A. Garcia, C. Grimm, W. Guerich, H. Mehammed, L. Schley, O. Schneider, G.L. Volpato (2008): Integration of Multiple Middlewares on a Single Computing Resource, The International Journal of Grid Computing: Theory, Methods and Applications, Elsevier (accepted).

[5] I. Foster: The Grid: A New Infrastructure for 21st Century Science. Physics Today, 55 (2). 42-47. 2002

[6] I. Foster, C. Kesselman, S. Tuecke, The Anatomy of the Grid: Enabling Scalable Virtual Organizations. International Journal of High Performance Computing Applications archive,Volume 15 , Issue 3 (August 2001) pp. 200 - 222 ISSN:1094-3420

[7] R. Wismüller, M. Bubak, W. Funika, T. Adrodz, and M. Kurdziel. Performance Measurement Model in the G-PM Tool. In M. Bubak et al., editors, Computational Science, ICCS 2004, 4th International Conference, volume 3036 of Lecture Notes in Computer Science, pages 462-465, Krakow, Poland, June 2004. Springer Verlag.

[8] T. Ludwig, R. Wismüller, V. Sunderam, AND A. BODE: OMIS On-line Monitoring Interface Specification (Version 2.0), vol. 9 of LRR-TUM Research Report Series, Shaker-Verlag, Aachen, Germany, 1997. ISBN 3-8265-3035-7.

[9] Vampirtrace, ZIH, Technische Universitat, Dresden: http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih.

[10] R. Wismüller, M. Bubak, W. Funika, T. Arodz, M. Kurdziel: Support for User-Defined Metrics in the Online Performance Analysis Tool G-PM. European Across Grids Conference 2004: pp.159-168

[11] B. Balis, M. Bubak, W. F, T. Szepieniec, R. Wismueller, and M. Radecki: Monitoring Grid Applications with Grid-enabled OMIS Monitor, in Proc. First European Across Grids Conference, F. Rivera et al., eds., vol. 2970 of Lecture Notes in Computer Science, Santiago de Compostela, Spain, Feb. 2003, Springer-Verlag, pp. 230-239.

[12] B. Mohr and F. Wolf: KOJAK - A Tool Set for Automatic Performance Analysis of Parallel Programs, Euro-Par 2003 Parallel Processing ISBN 978-3-540-40788-1 pp. 1301-1304, Springer, ISSN 0302-9743,2004

[13] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller: "MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools", in SC2003 (Phoenix, Arizona, November 2003)

[14] R. Wolski, N. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. Journal of Future Generation Computing Systems, 15(5-6):757-768, October 1999

[15] B. Balis, M. Bubak, W. Funika, T. Szepieniec, and R. Wismuller.: An Infrastructure for Grid Application Monitoring. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface, 9th European PVM/MPI Users' Group Meeting, Sept. - Oct. 2002, Linz, Austria, Lecture Notes in Computer Science 2474, pp. 41-49, Springer-Verlag, 2002.

[16] M. Bubak, W. Funika, R. Wismüller, T. Arodz, M. Kurdziel: The G-PM Tool for Grid-Oriented Performance Analysis. European Across Grids Conference 2003: pp. 240-248

[17] CrossGrid - Development of Grid Environment for interactive Applications, EU Project, IST-2001-32243, Technical Annex. http://www.eu-crossgrid.org

[18] T. Margalef, J. Jorba, O. Morajko, A. Morajko, E. Luque, Different Approaches to Automatic Performance Analysis of Distributed Applications. Performance Analysis and Grid Computing (Getov, Gerndt, Hoisie, Malony, Miller), 2002, Dagstuhl, Germany, PP, 93-107.

[19] B. Mohr, A. D. Malony, S.r Shende, F. Wolf: Design and Prototype of a Performance Tool Interface for OpenMP. The Journal of Supercomputing Volume 23 , Issue 1 (August 2002) pp: 105 - 128 Year of Publication: 2002

[20] T. Fahringer, M. Gerndt, G. Riley, and J. L. Tra: Knowledge Specication for Automatic Performance Analysis. APART Technical Report, ESPRIT IV Working Group on Automatic Performance Analysis, Nov. 1999. http://www.fz-juelich.de/apart-1/reports/wp2-asl.ps.gz

[21] T. Fahringer and C. Seragiotto: Modeling and Detecting Performance Problems for Distributed and Parallel Programs with JavaPSL. In 9th IEEE High-Performance Networking and Computing Conference, SC'2001, Denver, CO, Nov. 2001.

[22] J. R. Hollingsworth, B. P. Miller, M. J. R. Goncalves, Z. Xu, O. Naim, and L. Zheng: MDL: A Language and Compiler for Dynamic Program Instrumentation. In Proc. International Conference on Parallel Architectures and Compilation Techniques, San Francisco, CA, USA, 1997. ftp://grilled.cs.wisc.edu/technical papers/mdl.ps.gz

[23] Buck, B. R. and Hollingsworth, J. K. "An API for Runtime Code Patching." Journal of High Performance Computing Applications, 14 (4) (Winter 2000), pp. 317-329.

[24] University of Illinois. Pablo Performance Analysis Environment: Data Analysis. http://www-pablo.cs.uiuc.edu/Project/Pablo/PabloDataAnalysis.htm

[25] M. Kobayashi and K. Takeda. Information retrieval on the web. ACM Computing Surveys, 32(2):144-173, 2000.

[26] F. Wolf and B. Mohr: EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs. In Proc. of the 7th International Conference on High- Performance Computing and Networking (HPCN 99), Lecture Notes in Computer Science, pp. 503-512, Amsterdam, 1999. Springer-Verlag.

[27] F. Wolf and B. Mohr: Automatic Performance Analysis of MPI Applications Based on Event Traces. In Euro-Par 2000 Parallel Processing, 6th International Euro- Par Conference, Lecture Notes in Computer Science 1900, pp. 123-132, Munich, Germany, Aug. 2000. Springer-Verlag.

[28] R. Kohavi and F. Provost. Applications of data mining to electronic commerce. Data Mining Knowledge Discovery, 5(1-2):5-10, 2001

[29] Philip C. Roth, Dorian C. Arnold, and Barton P. Miller, "Benchmarking the MRNet Distributed Tool Infrastructure: Lessons Learned", in 2004 High-Performance Grid Computing Workshop, held in conjunction with the 2004 International Parallel and Distributed Processing Symposium (IPDPS 2004, Santa Fe, New Mexico, April 2004).

[30] M. Bubak, W. Funika, K. Iskra, R. Maruszewski, and R. Wismueller: Enhancing the Functionality of Performance Measurement Tools for Message Passing Environments, in Recent Advances in Parallel Virtual Machine and Message Passing Interface, Proc. 6th European PVM/MPI Users Group Meeting, J. Dongarra, E. Luque, and T. Margalef, eds., vol. 1697 of Lecture Notes in Computer Science, Barcelona, Spain, Sept. 1999, Springer-Verlag, pp. 67-74.

[31] T. Ludwig,, R. Wismueller, V. Sunderam, and A. Bode, OMIS On-line Monitoring Interface Specification (Version 2.0), vol. 9 of LRR-TUM Research Report Series, Shaker-Verlag, Aachen, Germany, 1997. ISBN 3-8265-3035-7.

[32] M. Aigner, Diskrete Mathematik, 4. Aufl. Vieweg, Braunschweig, Wiesbaden 2001.

[33] Bohm, A.P.W.; Sargeant, J.: Code optimization for tagged-token dataflow machines Computers, IEEE Transactions on Volume 38, Issue 1, Jan 1989, pp.4 - 14

[34] Yasuhiro Inagami, John F. Foley, The specification of a new Manchester Dataflow machine, International Conference on Supercomputing Proceedings of the 3rd international conference on Supercomputing, Greece, pp. 371 - 380, 1986, ISBN:0-89791-309-4

[35] David W. Goodwin: Interprocedural dataflow analysis in an executable optimizer. Conference on Programming Language Design and Implementation archive. Las Vegas, Nevada, United States, pp. 122 - 133, 1997, ISBN:0-89791-907-6.

[36] B. Jonsson Swedish, A fully abstract trace model for dataflow networks. Annual Symposium on Principles of Programming Languages Proceedings of the 16th ACM SIGPLAN-SIGACT. Austin, Texas, United States, pp. 155 - 165, 1989 ISBN:0-89791-294-2

[37] Arvind, Rishiyur S. Nikhil: Executing a Program on the MIT Tagged-Token Dataflow Architecture. IEEE Trans. Computers 39(3): pp. 300-318, (1990)

[38] J. Cargille and B. P. Miller, Binary Wrapping: A Technique for Instrumenting Object Code, ACM Sigplan Notices, 27 (1992), pp. 17-18.

[39] T. Kielmann, R.F.H. Hofman, H.E. Bal, A. Plaat, R.A.F. Bhoedjang. MagPIe: MPI's Collective Communication Operations For Clustered Wide Area Systems. ACM SIGPLAN Notices 34, 8, August 1999, pp. 131-140.

[40] Michael Gerndt, John Gurd: Special Issue: European-American Working Group on Automatic Performance Analysis (APART). Concurrency and

Computation: Practice and Experience (CONCURRENCY) 19(11): pp. 1447-1449 (2007)

[41] Dan Gunter, Brian Tierney: NetLogger: A Toolkit for Distributed System Performance Tuning and Debugging. Integrated Network Management 2003: pp. 97-100

[42] Hong Linh Truong, Thomas Fahringer: SCALEA-G: A Unified Monitoring and Performance Analysis System for the Grid. European Across Grids Conference 2004: pp. 202-211

[43] Edward A. Lee, David G. Messerschmitt: Synchronous Data Flow: Describing Signal Processing Algorithm for Parallel Computation. COMPCON, 1987: pp. 310-315

[44] J. Jorba, T. Margalef, E. Luque: Performance Analysis of Parallel Applications with KappaPI 2, Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005, John von Neumann Institute for Computing, Jülich, NIC Series, Vol. 33, ISBN 3-00-017352-8, 2006, pp. 155-162.

[45] F. Vraalsen, R. Aydt, C. Mendes, and D. Reed, Performance contracts: Predicting and monitoring grid application behavior, in Proceedings of the 2nd International Workshop on Grid Computing/LNCS, Springer-Verlag Lecture Notes in Computer Science, Denver, Colorado, November 12, 2001, Volume 2242, GRID 2001, pp. 154-165.

[46] Kahn, G.: The semantics of a simple language for parallel programming. Information Processing, 1974, pp. 471-475.

[47] I. Foster, C. Kesselman, J. Nick, S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[48] M. L. Massie, B. N. Chun and D. E. Culler, The Ganglia Distributed Monitoring System: Design, Implementation, and Experience, Parallel Computing 30(7) (July, 2004).

[49] D.A. Evensky, A.C. Gentile, L.J. Camp, and R.C. Armstrong. Lilith: Scalable Execution of User Code for Distributed Computing. Sixth IEEE International Symposium on High Performance Distributed Computing (HPDC 97), Portland, Oregon, 1997, pp. 306-314.

[50] M. J. Sottile and R. G. Minnich. Supermon: A high-speed cluster monitoring system. In CLUSTER '02: Proceedings of the IEEE International Conference on Cluster Computing, page 39, Washington, DC, USA, 2002. IEEE Computer Society. http://supermon.sourceforge.net/SupermonArchitecture.html

[51] K. Fürlinger and M. Gerndt, Periscope: Performance Analysis on Large-Scale Systems, InSiDE – Innovatives Supercomputing in Deutschland , Volume 3 (2, Autumn), pp. 26-29, 2005

[52] Balis, B., Bubak, M., Funika, W., Szepieniec, T., and Wismuller, R.: Monitoring and Performance Analysis of Grid Application. In: Computational Science - ICCS 2003, June 2003, St. Petersburg, Russia, Lecture Notes in Computer Science 2657, pp. 214-224, Springer-Verlag, 2003.

[53] Sameer S. Shende , Allen D. Malony, The Tau Parallel Performance System, International Journal of High Performance Computing Applications, v.20 n.2, pp. 287-311, May 2006

[54] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam and Tia Newhall, "The Paradyn Parallel Performance Measurement Tool", IEEE Computer 28, 11, (November 1995): 37-46. Special issue on performance evaluation tools for parallel and distributed computer systems. http://www.cs.wisc.edu/paradyn/papers/overview.ps.gz.

[55] N. Podhorszki AND P. Kacsuk, Presentation and Analysis of Grid Performance Data, in EuroPar 2003 - 9th International Conference, Klagenfurt, Austria, H. H. H. Kosch, L. Boszomenyi, Springer Verlag, 2003.

[56] Yan Jin, Robert Esser, Charles Lakos, Jörn W. Janneck: Modular Analysis of Dataflow Process Networks. FASE 2003: pp. 184-199

[57] Balaton, Z., Kacsuk, P., Podhorszki, N., and Vajda, F.: From Cluster Monitoring to Grid Monitoring Based on GRM. In: Euro-Par 2001 Parallel Processing, 7th International Euro-Par Conference, August 2001, Manchester, UK, Lecture Notes in Computer Science 2150, pp. 874-881, Springer-Verlag, 2001.

[58] Walid A. Najjar, Edward A. Lee, Guang R. Gao: Advances in the dataflow computational model. Parallel Computing 25(13-14): 1907-1929 (1999).

[59] R. WOLSKI, N. SPRING, AND J. HAYES, The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing, Journal of Future Generation Computing Systems, 15 (1999), pp. 757-768.

[60] Jack B. Dennis, David Misunas: A Preliminary Architecture for a Basic Data Flow Processor. ISCA 1974: 126-132

[61] ASKALON Visualization Diagrams. Institute for Software Science, University of Vienna. www.par.univie.ac.at/project/askalon/visualization/index.html

[62] P. Kacsuk: Performance Visualization in the GRADE Parallel Programming Environment, HPCN Asia, Beijing, China, 2000.

[63] B. Tierney et al.: The NetLogger Methodology for High Performance Distributed Systems Performance Analyser Proc. of the IEEE HPDC-7 (July 28-31, 1998, Chicago, IL) LBNL-42611

[64] C. Jin, R. Buyya, L. Stein, and Z. Zhang: A Dataflow Model for .NET-based Grid Computing Systems, Proceedings of the 3rd International Workshop on Grid Computing and Applications, June 6-7, 2007, World Scientific Press, Singapore.

[65] The Globus Heartbeat Monitor Specification http://www-fp.globus.org/hbm/heartbeat_spec.html

[66] N. Podhorszki, Z. Balaton, G. Gombás: Monitoring message-passing parallel applications in the grid with GRM and Mercury monitor: Lecture Notes in Computer Science, Laboratory of Parallel and Distributed Systems Type : Folyóiratcikk, Volume no.: 3165, pp. 179-181, 2004

[67] S. Girona, J. Labarta, R. M. Badia: Validation of Dimemas Communication Model for MPI Collective Operations. PVM/MPI 2000: pp.39-46

[68] Andrew W. Cooke, Alasdair J. G. Gray, Lisha Ma, Werner Nutt, James Magowan, Manfred Oevers, Paul Taylor, Rob Byrom, Laurence Field, Steve Hicks, Jason Leake, Manish Soni, Antony J. Wilson, Roney Cordenonsi, Linda Cornwall, Abdeslem Djaoui, Steve Fisher, Norbert Podhorszki, Brian A. Coghlan, Stuart Kenny, David O'Callaghan: R-GMA: An Information Integration System for Grid Monitoring. CoopIS/DOA/ODBASE 2003: pp. 462-481

[69] D. Barbara, editor. Special Section on Data Mining for Intrusion Detection and Threat Analysis, volume 30(4) of ACM SIGMOD Record, pp. 4-64. ACM Press, New York, NY, USA, 2001.

[70] R. H. Guting. An introduction to spatial database systems. The VLDB Journal, 3(4): pp. 357-399, 1994.

[71] Fagg, G. E., Vadhiyar, S. S., and Dongarra, J. J. 2000. ACCT: automatic collective communications tuning . Proceedings of EuroPVM-MPI 2000, Lecture Notes in Computer Science Vol. 1908, Springer-Verlag, Berlin, pp. 354–361 .

[72] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski. The GrADS Project: Software Support for High-Level Grid Application Development. International Journal of High Performance Applications and Supercomputing 15(4), 2001.

[73] Christophe G. Giraud-Carrier: A Reconfigurable Data Flow Machine for Implementing Functional Programming Languages. SIGPLAN Notices, pp. 22-28 , Volume 29, Number 9, 1994

[74] R. Wismüller, M. Bubak, and W. Funika. High-Level Application Specific Performance Analysis using the G-PM Tool. Future Generation Computer Systems, 24(2):121-132, February 2008.