Design and Evaluation of a Parallel Data Redistribution Component for TGrid

Sascha Hunold[1], Thomas Rauber[1], and Gudula Rünger[2]

[1] Department of Mathematics and Physics
University of Bayreuth, Germany
{hunold,rauber}@uni-bayreuth.de
[2] Department of Computer Science
Chemnitz University of Technology, Germany
ruenger@informatik.tu-chemnitz.de

Abstract. Data redistribution of parallel data representations has become an important factor of grid frameworks for scientific computing. Providing the developers with generalized interfaces for flexible parallel data redistribution is a major goal of this research. In this article we present the architecture and the implementation of the redistribution module of `TGrid`. `TGrid` is a grid-enabled runtime system for applications consisting of cooperating multiprocessor tasks (M-tasks). The data redistribution module enables `TGrid` components to transfer data structures to other components which may be located on the same local subnet or may be executed remotely. We show how the parallel data redistribution is designed to be flexible, extendible, scalable, and particularly easy-to-use. The article includes a detailed experimental analysis of the redistribution module by providing a comparison of throughputs which were measured for a large range of processors and for different interconnection networks.

1 Introduction

Heterogeneous distributed environments or grid environments provide large computation resources for the execution of extremely computation intensive scientific applications. These computing environments can be exploited to execute algorithms and applications with task-parallel structure. Those applications have to be transformed into modular component-based parallel programs which can be executed on a grid environment. A suitable programming model and an efficient runtime environment which implements the programming model are required for developing this kind of applications. The modular structure of programs can be expressed by a task graph. Each node in the task graph represents a single grid-enabled component. These components are implemented as multiprocessor tasks (M-tasks). A task graph containing M-tasks offers two levels of parallelism. Components can be executed concurrently but each component may also contain data-parallel code or may even contain a recursive structure of components.

The runtime environment `TGrid` is an approach for executing M-tasks in a heterogeneous distributed environment. The `TGrid` environment consists of

several modules which enable the processing of M-tasks, the mapping of M-tasks to processors for execution, the observation of running M-tasks, and the redistribution of data between M-tasks [1]. M-tasks which can be executed in the `TGrid` environment are able to take full advantage of the underlying communication network by using MPI. On the other hand, since the M-tasks are written in Java, they are also completely platform independent. The redistribution of data structures within cooperating components plays an important role for component-based grid environments. There are several requirements that a data redistribution component has to meet to provide a solid framework for component-based programming on the grid. The data redistribution component must be flexible to support various data types and to be easily extendible if required. It should be simple to use, i.e. the developer should only provide all the information to allow an efficient data redistribution, and the component should support the coupling of components.

The contribution of this paper is the design and the implementation of a data redistribution module which enables the coupling of M-tasks in heterogenous computation systems. The redistribution module is able to dynamically create data messages from information provided by the source component (M-task) and to automatically redistribute these data onto the processors of the target component. In particular, the redistribution module relieves the program developer from writing redistribution code for each component, which is tedious and error-prone.

The rest of the paper is structured as follows. Section 2 gives a short introduction to the `TGrid` runtime system. In Section 3 the architecture of the redistribution module of `TGrid` outlined. Section 4 addresses the implementation details and introduces the management protocol of the redistribution module. Section 5 presents experimental results. Section 6 discusses related work, and Section 7 concludes the paper.

2 Overview of TGrid

`TGrid` is a runtime system for a network of heterogenous parallel machines which allows the execution of hierarchically-structured multi-processor tasks. Multi-processor tasks (M-tasks) can improve the performance of parallel programs due to a reduced communication overhead [2, 3]. `TGrid` provides a framework to run these M-tasks on a heterogenous collection of clusters as proposed in [4]. `TGrid` consists of several subnets of which each is controlled by a subnet manager. The subnet manager is in charge of executing and observing tasks on its private network. Such a private network could be a homogeneous cluster or any other heterogenous collection of machines that share a private IP address space or are visible to each other. The subnet managers are able to transfer data through a WAN, which might be insecure. Therefore, the subnet managers support several protocols, such as https, ssh, etc., in order to bypass local firewalls and to perform encrypted data transfer.

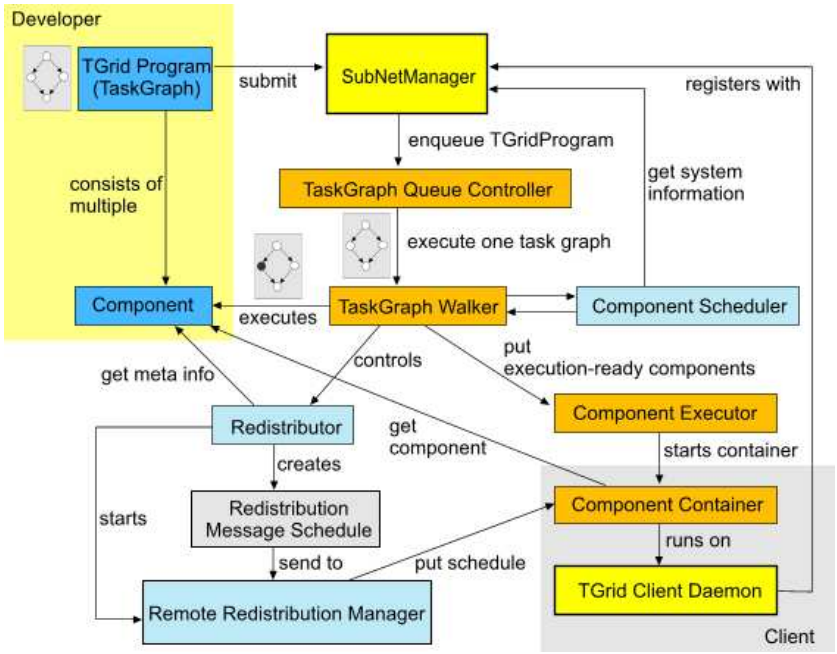


Fig. 1. TGrid architecture

The software architecture of the `TGrid` runtime system and an overview of the functioning of the subnet manager is sketched in Fig. 1. The program developer has access to one of the subnet managers and can submit programs to this local subnet manager. A `TGrid` program can be represented as a directed task graph where `TGrid` components are the nodes of the graph. Edges of the task graph represent dependencies between components, which includes data dependencies based on an output-input relation between components. If there is a data dependency between consecutive components A and B, such that B requires data structures produced by A, then the `TGrid` framework provides a redistribution component which can automatically satisfy the dependency (*coupling* of component). The subnet manager enqueues `TGrid` programs into a list of programs to be executed. The queue controller selects the next program to be executed on `TGrid`, spawns a TaskGraph Walker, and passes the user's program as parameter. The TaskGraph Walker executes the nodes of the task graph (components) as specified in the application program. An instance of a TaskGraph Walker observes the execution of a program throughout the entire life cycle of the program.

As soon as all input data dependencies of a node (component) are fulfilled a component is ready for execution. In `TGrid`, a component can only be executed within a single subnet of the grid runtime system. Components can be arbitrary single-processor or multi-processor tasks. `TGrid` is especially designed to support

the execution of hierarchically-structured multi-processor tasks based on `TLib` [2]. `TLib` is an MPI-based library that provides separate functions for the hierarchical structuring of processor groups and for the coordination of concurrent and nested M-tasks. `TGrid` components can also be executed on a remote subnet. If the component scheduler decides to execute a component in a remote subnet, the component will be sent to this target subnet and started remotely. As mentioned before, executing a component requires a component scheduling beforehand. The component scheduler maps a component to processors of a subnet taking into account the current workload of the subnet as well as the component's preferences and requirements, e.g. minimum and preferred number of processors to run this component. The scheduler can retrieve this information about the workload and the connected processors by calling interfaces provided by the subnet manager.

To eventually execute a scheduled component, the TaskGraph Walker starts a Component Executor which encapsulates the necessary actions to run the code. For instance, components using MPI have to be started differently than components using Java sockets. In case of MPI components, the client processor starts an MPI component container. An MPI component container initializes the MPI environment, registers itself with its local subnet manager, and waits for an MPI component to execute. The subnet manager sends the component to the component container. The component container checks if all data dependency of this component have been satisfied. If not, it notifies the subnet manager that it is ready for performing a data redistribution operation. When the data dependencies have been satisfied, the component container starts the execution of the component. The component container also informs the subnet manager when the component has finished its computation.

`TGrid` and all its components are written in Java which makes them completely platform independent. The current implementation supports components that are based on MPI. This approach allows us to run these components on arbitrary machines in the grid without having to recompile some parts. Moreover, the ability to access the MPI layer through JavaMPI enables the application to benefit from using the best network driver available on the corresponding machine.

3 Architecture of the MxN redistribution component

A parallel data redistribution, often known as MxN redistribution, is required for the coupling of programs (or components) which are executed in a data-parallel manner and have a data dependency. MxN stands for a data redistribution from M processors of the source program to N processors of the target program. Since all kinds of data redistributions between components require similar meta informations, the implementations follow similar patterns. The basic steps to redistribute data are data identification, message generation, message scheduling, and the actual communication. For `TGrid`, these basic steps are realized by the following tasks:

1. Specify the input and output *variables* of the redistribution. An output variable defines data which are provided by the sending component. Hence, input variables define data structures of the receiving component.
2. Define which parts of the source data has to be transferred to the target component (*selection*).
3. Define the data *mapping* between sending and receiving component.
4. Create a *schedule* of redistribution messages (communication schedule) that includes a sequence of messages that correctly moves data structures from the source to the target component.
5. Start communication and perform transfer.

In `TGrid`, the redistribution can be started when the source component has finished its computation and the target component has been initialized. The redistribution process has to determine all the meta information from both endpoints in order to create the communication schedule. Enumerating the information required to create a communication schedule may be straight-forward, however, it is difficult to design an easy-to-use and extendible (data-typing system) software component, i.e. `TGrid` provides generic data type interfaces which enables the developer to define and implement new data distribution types (e.g. block-cyclic 2-dimensional arrays of integers) if necessary.

A major concern is the throughput and the latency that can be provided in the communication stage. In order to gain good performance, we decided to differentiate between inter-subnet and intra-subnet redistribution. The data redistribution between two components which are part of the same subnet is referred to as intra-subnet redistribution. Intra-subnet redistribution in `TGrid` is characterized by direct message transfer from the sending processors to the receiving processors. In case that a component is executed by multiple operating system processes on one machine, direct message transfer to each of these participating processes is only possible if each processor can allocate arbitrary ports to receive data. Since most firewalls are configured to allow communication only on a few ports, we assume that unrestricted port management may only be allowed in private subnets. Unrestricted port management within a subnet enables the processors to directly transfer messages. As a result of the intra-subnet design, the message transfer within processors of the same subnet can be performed concurrently, i.e. no routing is involved. This ensures a high throughput and enables a fast redistribution within subnets. The communication between different subnets has different requirements. Many network configurations do now allow direct communication between a pair of processors, each located in a different subnet. However, firewalls may block traffic between two nodes and the IP address space may be completely private to a subnet. Instead, messages from one subnet to another have to be routed through the local subnet manager. The subnet manager has to be set up appropriately to support bypassing the firewall restrictions. The number of message hops increases when messages have to be routed over several other nodes which strongly influences the achievable latency and throughput.

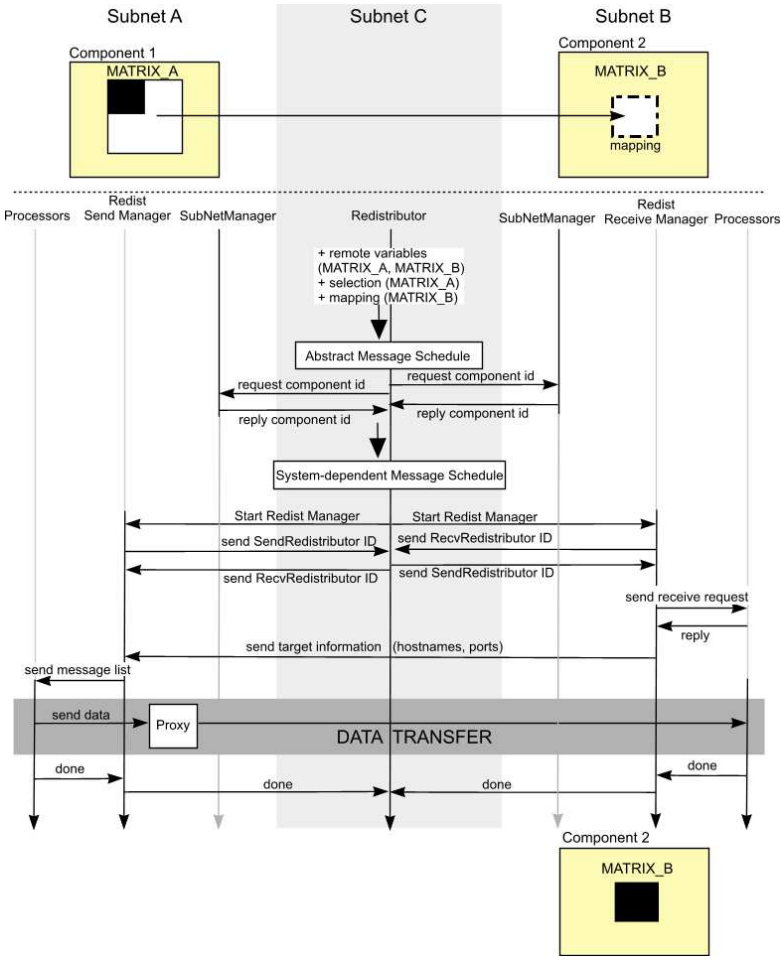


Fig. 2. `TGrid` Redistribution Protocol

Another important design goal for the redistribution module is to allow concurrent redistributions between components. The redistribution component of `TGrid` is implemented entirely in a multi-threaded way, i.e. different redistributions can be performed between each pair of components at the same time.

4 Data redistribution protocol of TGrid

Data redistribution is realized by the `TGrid` Communication Protocol, which is shown in Fig. 2. A common case is that a sub-matrix computed by a component (Component 1) is required as input by another component (Component 2). The processors that execute these components may be located on different subnets or may be part of the same subnet. The redistributor is the entity that manages

the entire redistribution and may be located on a third subnet, called Subnet C in Fig. 2. Since the redistributor is part of the execution of a `TGrid` program (see Fig. 1 in Sec. 2) it runs on the subnet where the user has submitted the application program. The user has to instruct the redistributor (by passing meta information) which data has to be transferred between the components. This meta information includes

- *unique identifiers of variables* which determine the source and the target data structure in source and target component
- the *data selection* in the source component and the *data mapping* in the target component as described in Section 3.
- the *component schedule* of the source and the target component (number of processors, mapping). It enables the redistributor to retrieve information about the data layout which is required for generating the communication schedule.
- *references* to *source* and *target component objects*.

Using this information, the redistributor creates an abstract communication schedule. The abstract communication schedule is composed of a list of abstract messages which do not contain system information such as host names or IP addresses of the `TGrid` environment. In order to deliver messages in the current runtime environment, the redistributor has to convert the *abstract communication schedule* into a *system-dependent communication schedule*. Therefore, abstract messages are extended with a header which is used to uniquely identify the destination of a target variables. A component's variable can be uniquely identified within `TGrid` by specifying

- the *subnet* to uniquely identify the subnet to route the message to.
- the *component* to locate the component in the subnet. `TGrid` components get a unique label when they are passed to the runtime environment for execution.
- the *name of the variable* which stores the data to be accessed.

The *name of the variable* and the name of the *subnet* can be easily determined. The *component id* in `TGrid` however is assigned at runtime. To retrieve the component identifier from a remote subnet, the restributor sends a request to the remote subnet manager. The subnet manager responds by sending the corresponding *component id* back to the redistributor. The redistributor has now all information to convert the abstract communication schedule into a *system-dependent communication schedule*.

The redistributor starts redistribution managers on both subnets which manage the redistribution on the remote subnets and are primarily used to avoid communication across subnet border when it is not required. One redistribution manager controls the actions taken by the sending processors and the other controls the receiving processors, respectively. Redistribution managers belong to the same subnet as the sending or the receiving processors. Therefore, only data messages from source processors to target processors have to be sent across

subnet borders. All other protocol messages that may be necessary for the redistribution, e.g. synchronization, require only communication within subnets (from processors to their redistribution manager).

`TGrid` is designed to run several redistributors and also redistribution managers concurrently. Therefore, each redistribution manager gets a unique ID when instantiated. In order to determine the correct redistribution manager for incoming messages the redistribution managers have to exchange their IDs at first. Each redistribution manager sends its ID to the redistributor which forwards it to the remote redistribution manager.

The redistribution manager that controls the processors which have to send messages is referred to as *send manager* and the manager for the receiving processors as *receive manager*, respectively. The receive manager assigns a *unique port* to each receiving processor and sends a *receive request* to the processors. When all processors have responded to this request, the receive manager sends meta data of each receiving processor (*hostname*, *port*, *local rank*, etc.) to the send manager. The send manager uses this information to create a proxy object through which the sending processors send their messages. *Proxy objects* are used to make the *sending* of data messages *transparent* to the sending processor, i.e. the processor is not aware whether the target processor is part of the same local subnet or located on a remote subnet. With the information about the target host, the send manager can assign the message list and proxies to each processor that has data to contribute. Both the send manager and the receive manager wait until all cluster machines have acknowledged that the message transfer has been completed.

The sending or receiving of messages is not done by the component itself. This would force the component developer to implement redistribution code (sending/receiving) for every single component. Instead, the data transfer responsibility is part of the component container, see Section 2. Since component containers have no knowledge of the internal data structure of a component, the data transfer between a container and a component is realized by abstract interfaces which include methods to retrieve meta information (variable identifiers, data selection, data mapping) as discussed in this section.

5 Experimental results

We performed a number of throughput experiments to evaluate the performance of the `TGrid` redistribution component. All intra-subnet tests were run on a cluster consisting of 64 Opteron processors (Model 246, 2 GHz). The cluster has three different interconnection networks per node; 100 MBit Ethernet, Gigabit Ethernet, and Melanox Infiniband. Currently, the `TGrid` framework provides a small number of distributed data types, e.g. distributed block-partitioned matrices, which play an important role in scientific applications.

The first experiment measures the throughput achieved when redistributing a block-partitioned matrix between two components in `TGrid`. Fig. 3(a) compares the throughput per node that was achieved with several matrix configurations

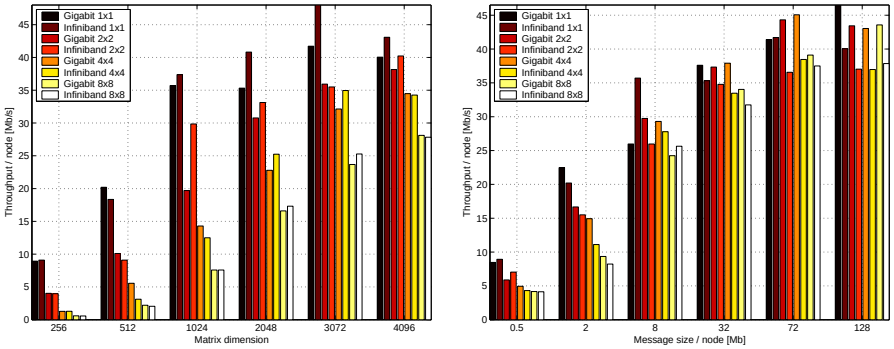
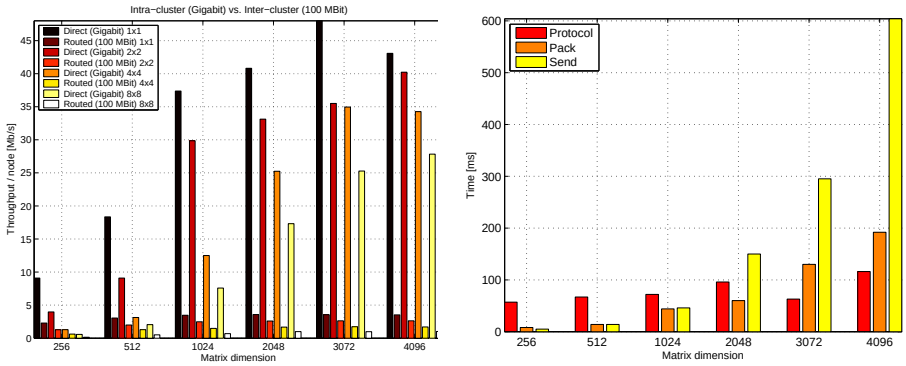


Fig. 3. (a) Intra-subnet data redistribution performance on Gigabit Ethernet and Infiniband. (b) Throughput comparison of several data redistributions on Gigabit Ethernet and on Infiniband. Constant message size per processor.

and on different interconnection networks. The label 'mxn' denotes the number of processors that were assigned to the source (m) and to the target component (n). For instance, the label 'Gigabit 2x2' stands for the data redistribution over Gigabit Ethernet between 2 processors which run the source component and 2 processors which run the target component. As expected, the throughput per node increases with larger matrices. The message size that each processor has to send or receive depends on the size of the input matrix and on the number of processors that store the matrix. If more processors are used to store a matrix the throughput will be smaller. On the other hand, the protocol overhead has less impact on the overall time when the message size increases. For instance, for matrix size 4096 the throughput between 2 nodes ('1x1') is higher than measured with 16 nodes ('8x8'). The small throughput that can be seen for a matrix size of 256 is a direct result of the protocol overhead of the redistribution. The throughput of the redistribution is also limited by the object serialization done at the Java layer. The object serialization in Java is a very powerful tool and makes it easy to send object with different implementations across the network. These limitations cause the similar performance of Gigabit Ethernet and Infiniband.

Since the size of each message depends on the number of processors which store a distributed matrix, it is also important to examine the throughput when the size of matrices is adjusted in the way that the message size per processor is held constant. Fig. 3(b) shows the throughput per node with a constant message size per node (we used one processor per node on SMP boards). Again, the impact of the protocol overhead decreases the throughput for a small message size (0.5 MB) in comparison to bigger messages. We can also observe that the throughput per node is almost equal for a specific message size taking advantage of concurrent message redistribution.

Another important test for grid environments is to measure the bandwidth between long-distance networks. As mentioned above, `TGrid` is able to execute



(a) (b)

Fig. 4. (a) Intra vs. Inter-cluster redistribution; (b) Time spent in each redistribution step (matrix redistribution from 4 source processors to 4 target processors, 4x4, 8 nodes). Interconnection network: Gigabit Ethernet.

components remotely and can also handle the data redistribution between components which are located in different subnets. Fig. 4(a) compares the throughput achieved by inter-cluster redistribution (routed) with the throughput achieved by intra-cluster redistribution (direct). The inter-subnet redistribution tests were performed on a single cluster. To perform these tests, we divided the number of nodes into two disjoint subnets where each subnet is controlled by one node of the corresponding subnet. We also used another IP range with the consequence that the transfer between both subnets as well as the communication in between the subnet is done over the 100 MBit interconnection network. It is not surprising to see such a huge performance difference. Considering that the 100 Mbit card has a maximum bandwidth of 12.5 MB/s. Since each message is routed through two different subnet managers and since each message can only be forwarded to the next destination when it was entirely received, the throughput of a single message has an upper bound of $12.5/3 \approx 4$ MB/s (3 message hops to destination). Additionally, the sending of message from multiple processors has to be synchronized on a subnet manager to some extend which in turn reduces the possible throughput per node.

Fig. 4(b) shows the partial times spent in different redistribution steps. The bar labeled with 'Protocol' denotes the protocol overhead introduced by using the `TGrid` redistributor. The time 'Pack' denotes is the time for packing messages and the time 'Send' denotes the time for transferring the message over the network. The packing of messages contains determining the matrix elements to be sent, allocating a message buffer, selecting and copying elements from the local matrix buffer into the message buffer, and writing a message header containing meta informations for message reconstruction. We can observe that the protocol overhead is constant for different matrix sizes. For this reason, the `TGrid` protocol overhead has less impact on the performance for larger matrices.

As we expected, the time for packing and sending messages increases linearly with the number of elements to be transferred.

The redistribution component of `TGrid` has shown good performance when taking into account that there is a price to pay for determining data, converting data, and managing arbitrary data inside a heterogenous grid environment.

6 Related work

Data redistribution in distributed or grid environments has been an active area of research in the last couple of years. The Parallel Application WorkSpace (PAWS) provides a framework for coupling parallel applications within a component-like model [5]. The PAWS approach is similar to the redistribution component of `TGrid`, however, on another level of granularity. In contrast to PAWS, `TGrid` uses the notion of redistribution within an application, i.e. data redistribution is required to start a particular M-task of the application. The Model Coupling Toolkit (MCT) [6] is a software library written in Fortran 90 which provides functions to transfer data structures between parallel applications. InterComm [7] is a redistribution library that moves the determination of data redistribution patterns inside the programs to be coupled. An ongoing research project is carried out by the CCA (common component architecture) forum. The MxN working group of the CCA forum is working on the definition and implementation of interfaces to transfer data elements between parallel components running with different numbers of processes in each parallel component [8]. The framework Seine [9] is a geometry-based interaction model which is encapsulated as a CCA compliant component within the Ccaffeine CCA framework. Other CCA compliant frameworks that support coupling of distributed components are DCA [10] and XCAT [11]. An effective algorithm for communication schedule generation for data redistributions is presented in [12]. Messages between different clusters are scheduled in order to avoid exceeding the bandwidth capacity of the backbone.

7 Conclusions

In this article, we have presented the data redistribution component used by `TGrid`. The redistribution component is fully capable of managing arbitrary MxN redistributions. The redistribution framework of `TGrid` works completely multithreaded so that multiple variables of components can be transferred to arbitrary receiving components in parallel. The redistribution component provides simple but flexible interfaces which make it an easy task to extend the implemented data types and algorithms. The redistribution component differentiates between intra and inter-subnet communication. The intra-subnet communication allows the parallel sending of messages and is therefore of major importance for high throughput. The redistribution component of `TGrid` has shown good performance in the experiments.

References

1. Hunold, S., Rauber, T., Rünger, G.: TGrid – Grid Runtime Support for Hierarchically Structured Task-Parallel Programs. In: Proceedings of the Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, IEEE Computer Society Press (2006)
2. Rauber, T., Rünger, G.: Tlib - A Library to Support Programming with Hierarchical Multi-Processor Tasks. Journal of Parallel and Distributed Computing **65** (2005) 347–360
3. Hunold, S., Rauber, T., Rünger, G.: Multilevel Hierarchical Matrix Multiplication on Clusters. In: Proceedings of the 18th Annual ACM International Conference on Supercomputing, ICS'04. (2004) 136–145
4. Rauber, T., Rünger, G.: M-Task-Programming for Heterogeneous Systems and Grid Environments. In: Proc. of the IPDPS Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models, IEEE (2005)
5. Beckman, P.H., Fasel, P.K., Humphrey, W.F., Mniszewski, S.M.: Efficient Coupling of Parallel Applications Using PAWS. In: HPDC '98: Proceedings of the The Seventh IEEE International Symposium on High Performance Distributed Computing, Washington, DC, USA, IEEE Computer Society (1998) 215
6. Larson, J., Jacob, R., Ong, E.: The Model Coupling Toolkit: A New Fortran90 Toolkit for Building Multiphysics Parallel Coupled Models. Int. J. High Perform. Comput. Appl. **19** (2005) 277–292
7. Lee, J.Y., Sussman, A.: High Performance Communication between Parallel Programs. In: IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 4, Washington, DC, USA, IEEE Computer Society (2005) 177.2
8. Bertrand, F., Bramley, R., Damevski, K.B., Kohl, J.A., Bernholdt, D.E., Larson, J.W., Sussman, A.: Data Redistribution and Remote Method Invocation in Parallel Component Architectures. In: Proceedings of the 19th International Parallel and Distributed Processing Symposium: IPDPS 2005. (2005) Best Paper Award.
9. Zhang, L., Parashar, M.: Enabling efficient and flexible coupling of parallel scientific applications. In: International Parallel and Distributed Processing Symposium IPDPS'2006, Rhodes Island, Greece, IEEE Computer Society Press (2006)
10. Bertrand, F., Bramley, R.: DCA: A Distributed CCA Framework Based on MPI. In: 18th International Parallel and Distributed Processing Symposium (IPDPS 2004), Santa Fe, New Mexico, USA, IEEE Computer Society (2004) 90–97
11. Krishnan, S., Gannon, D.: XCAT3: A Framework for CCA Components as OGSA Services. In: 18th International Parallel and Distributed Processing Symposium (IPDPS 2004), CD-ROM / Abstracts Proceedings, 26-30 April 2004, Santa Fe, New Mexico, USA, IEEE Computer Society (2004) 90–97
12. Jeannot, E., Wagner, F.: Messages Scheduling for data Redistribution between Heterogeneous Clusters. In: Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2005). (2005)