TGrid – Grid runtime support for hierarchically structured task-parallel programs

Sascha Hunold and Thomas Rauber
Department of Mathematics and Physics
University of Bayreuth, Germany
{hunold,rauber}@uni-bayreuth.de

Gudula Rünger
Department of Computer Science
Chemnitz University of Technology, Germany
ruenger@informatik.tu-chemnitz.de

Abstract

In this article we introduce a grid runtime system called TGrid *which is designed to run hierarchically structured task-parallel programs on heterogenous environments and can also be used for common component-based grid programming.* TGrid *is built on top of a location-aware communication layer which enables the runtime system to cluster grid nodes. As a result, the component scheduler assigns a multi-processor task to a set of processors taking into account the spatial locality within the available processors. The multi-processor task directly benefits from having less network overhead and thus, the overall runtime of a grid-enabled multi-processor program is reduced.*

1. Introduction

Many large applications that require execution on a high-performance platform have an inherent modular structure of cooperating subtasks calling each other. Examples of such applications include environmental models combining atmospheric, surface water and water models, or aircraft simulations combining models for fluid dynamics, structural dynamics and surface heating. Coding such applications with hierarchically structured multiprocessor tasks (M-tasks) has been shown to be successful for parallel platforms with a distributed address space [13] , since the corresponding group-based execution of communication operations can help to reduce the communication overhead. This effect is most significant if collective communication operations like broadcast or gather are involved and can be exploited to obtain message-passing programs that are scalable for a larger number of processors. Hierarchical M-task programs are obtained by subdividing the application program into subprograms (M-tasks) that can be executed concurrently to each other or that must be executed sequentially one after another, as given by the dependencies of the application. Each subprogram can be structured correspondingly, leading to an hierarchy of M-tasks.

The runtime environment TGrid is our 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 M-tasks during execution, and the redistribution of data between M-tasks. M-tasks which can be executed in TGrid are able to take full advantage of the underlying communication network by using MPI. M-tasks for TGrid are written in Java and thus, they are completely platform independent which is elementary for the computation in heterogenous environments.

The contribution of this paper is to present the design and the implementation of the TGrid runtime system to execute hierarchically-structured programs on heterogenous systems and to present preliminary results with the current implementation of TGrid.

The rest of the paper is structured as follows. Section 2 describes the basic modules of TGrid. Section 3 shows experimental results of example programs. Section 4 discusses related work and Section 5 concludes the paper.

2. The TGrid runtime system in detail

The goal of the TGrid runtime system is to provide a heterogenous runtime platform for hierarchically-structured multi-processor tasks (M-tasks). Using multi-processor tasks can improve the performance of parallel programs due to a reduced communication overhead [9, 13]. In previous work we have proposed a concept how to execute M-tasks on heterogenous systems and grid environments [12] and we have presented an approach to provide efficient scheduling strategies for each M-task [14].

A sample grid configuration which is the target of the TGrid runtime environment is outlined in Figure 1. As depicted, the TGrid consists of a number of connected subnet

1-4244-0328-6/06/$20.00 ©2006 IEEE.

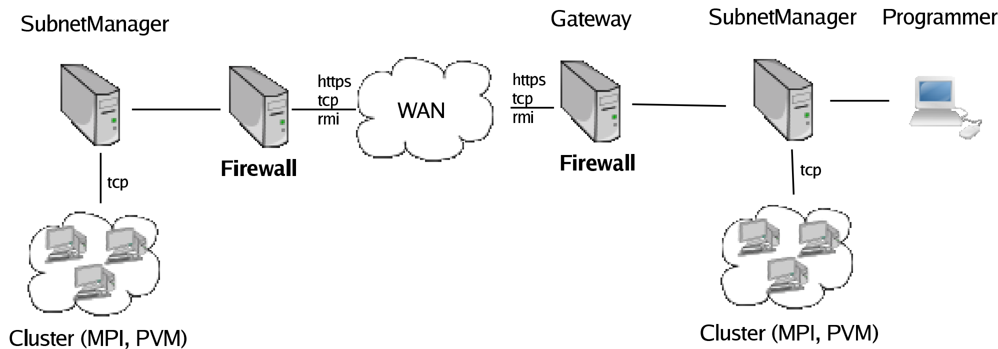

Figure 1. Sample Grid configuration for the TGrid environment

managers which have control over a homogenous cluster. Subnet managers should be able to connect to other subnet managers over a possible insecure WAN. Therefore, the subnet managers need to support several protocols such as https, ssh, etc. in order to pass the local firewall and also to perform encrypted data transfer. A developer has access to one of the subnet managers and can submit programs to the local subnet manager.

Each `TGrid` program can be represented as a directed task graph as depicted in Figure 2. The user submits a program to the `TGrid` runtime which executes the nodes of the task graph as components. As soon as all input data dependencies of a node are fulfilled a component is ready for execution. A component represented as node in the graph runs in a single subnet of the grid runtime system. All components can be arbitrary single-processor or multi-processor tasks. However, the main design goal of `TGrid` is the support of hierarchically-structured multi-processor tasks preferably based on `TLib` [13].

The `TGrid` connects different subnets to a grid-enabled runtime system. In general, a subnet is locally connected through high-speed switches provided by homogenous computing clusters. The subnet is controlled by a subnet manager that runs on one of the cluster nodes. The `TGrid` runtime system is in charge of selecting the next component to be executed as well as the subnet that this component will be scheduled to. In particular, the `TGrid` runtime is also responsible for the data redistribution between components which may require intra-cluster communication as well as inter-cluster communication.

The execution of a `TGrid` program consists of the following steps:

1. The user submits a `TGrid` program to one of the subnet managers currently connected.
2. A program processing thread is spawned. This thread is responsible for executing all components by traversing the task graph of the program.
3. Components that are ready for execution are passed to the component scheduler. A schedule is provided which takes into account the current workload of all connected subnets as well as the component's preferences and requirements, e.g. the minimum and preferred number of executing processors.
4. A component executor is requested on the target subnet which is used for running the component. For example, an MPI executor writes an MPI machinefile and calls the appropriate `mpirun` command.
5. The `mpirun` command will start the specified number of MPI processes. Each MPI process runs a 'component container' (task runner) which is able to run arbitrary MPI components. This container process registers with its subnet manager and requests a component to run.
6. The subnet managers sends the component's code to the requesting task runner. If the execution of this component requires input data the task runner will notify the subnet manager that is has received the component and waits until the redistribution is done.
7. Redistribution may be performed between the execution of components to make one component ready for execution by reorganizing the distribution of data structures according to the needs of the next component to be executed.
8. The component is executed inside the container and the task runner informs the subnet manager about the exit code of this component.

A sample program is given in Figure 3. In this example, a 16×16 matrix is allocated by a matrix generation component and then redistributed to a print component. For doing so, two `TGrid` components have to be instantiated, namely MatrixGenerate and MatrixPrint. Variables which need to be declared or to

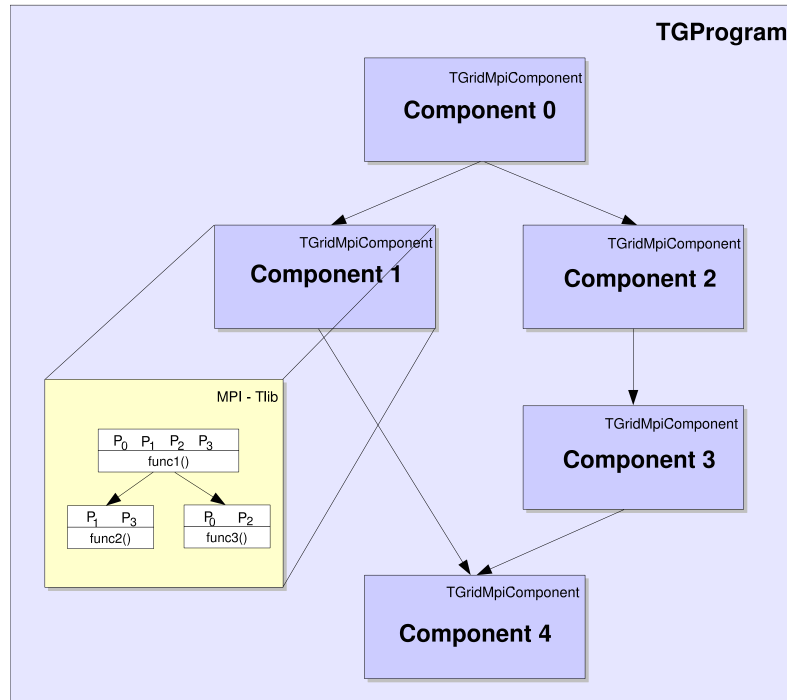

Figure 2. Structure of a TGrid program. The example shows a task graph of TGrid components which are MPI components in this case. Each MPI component can be a multi-processor task which might also exhibit a recursive structure.

be referenced for the redistribution are uniquely identified by constants `MatrixGenerate.GEN_MATRIX` and `MatrixPrint.PRT_MATRIX`. These identifiers have to be provided by a component's developer. The print component will only print the right upper square of the matrix. This information is part of the redistribution configuration (ArraySelection, ArrayMapping).

2.1. Grid configuration

All possible subnets which may participate in processing components are configured as described in an XML configuration file. Figure 4 shows a basic configuration file of the `TGrid` environment. In the *ports* section the user defines which ports will be used for intra-subnet communication. The element *contextdefinition* determines which underlying communication module will be selected for inter-subnet communication. Currently, communication between subnets is done via the TCP protocol. It is also possible to use JMS (Java messing service) as basic communication platform. `TGrid` is entirely built upon free software and hence, we use OpenJMS as JMS implementation. By using OpenJMS, the user can choose his preferred communication protocol, e.g. RMI, http, or https. So, the communication module can be adapted to any firewall configuration.

An important part of the configuration is the declaration of the subnets which are part of a `TGrid` runtime environment. The subnet configuration contains information about the subnet manager and the connected clients. Only clients that are listed inside the 'clients' tag are candidates as grid computing nodes. Listing the names of clients in the configuration file has two reasons. Many cluster systems are equipped with more than one inter-connection network. It is therefore crucial for high-performance to exactly indicate which inter-connection network should be used within multi-processor tasks. The second reason is to make the adjustment of the `TGrid` configuration technically easier since subnet managers only have to inform clients about their new affiliation.

To bring up the grid environment the user has to start `TGrid` client daemons on all clients. The subnet managers can then be started by passing the unique identification ('id') as parameter. A subnet manager reads the corresponding configuration and checks if the specified clients and the adjacent subnet managers are online. Until the subnet manager is shut down it keeps detecting whether the status of other subnet managers or local clients has changed.

```
public class MatrixTest implements TGProgram {

  public void run(TGRuntime runtime) throws TGridRunException {

    TGridComponent matGen = new MatrixGenerate();
    matGen.setDataDeclaration(MatrixGenerate.GEN_MATRIX, new ArrayDecl(16, 16));

    Schedule[] sched = TGridScheduler.getSchedulerInstance().schedule(matGen);

    TGComponentHandle matGenHandle = runtime.execute(matGen, sched[0]);
    matGenHandle.waitFor();

    TGridComponent matPrintComp = new MatrixPrint();
    matPrintComp.setDataDeclaration(MatrixPrint.PRT_MATRIX, new ArrayDecl(8, 8));

    Schedule[] sched2 = TGridScheduler.getSchedulerInstance().schedule(matPrintComp);

    TGComponentHandle matPrintHandle = runtime.execute(matPrintComp, sched2[0]);

    // information required to perform a redistribution
    // (schedules, component handles, unique id of variables)
    RedistConfigObject redistConfig = new RedistConfigObject(
       MatrixGenerate.GEN_MATRIX, matGenHandle, new ArraySelection(0, 8, 8, 8),
       MatrixPrint.PRT_MATRIX, matPrintHandle, new ArrayMapping(0, 0));

    Redistributor2 redist = new Redistributor2(runtime.getSubnetManager(),
       redistConfig);
    redist.start();
    redist.waitFor();

    // ...
  }
}
```

Figure 3. In this TGrid example, a matrix is instantiated by a component MatrixGenerate and is then transferred to the processors that run the MatrixPrint component which prints the matrix as its result.

2.2. Component structure of TGrid

The component is the central entity of `TGrid`. `TGrid` components can be assigned to different subnets, i.e. they can be moved to another cluster and are then executed remotely. A component represents an execution unit for a single task. The granularity of different tasks depends on the problem and the required performance. For example, for our experiments we have created a component that allocates arrays of doubles on a given set of processors. Another component might print some data to the terminal. To enable such a data flow from one component to the other, we need a way to identify data of components and to access and modify this data.

To execute a component in `TGrid`, the component has to implement the following interfaces:

- *Scheduling interface* to provide meta information about the component such as the minimum number of processors, the maximum number of processors, etc.
- *Data mapping interface* to get information about data distribution and mapping of a variable for a set of processors. The component returns the processor grid and the mapping of the variable onto each processor.
- *Data redistribution interface* to get and set data which is required by the redistribution thread. During the redistribution this thread receives data for a component's variable and but has no knowledge of the internal structure of the same. Therefore, these are crucial methods for the redistribution.
- *Data initialization interface* to get and set initial data declarations like the size of a matrix.

When all these interface definitions have been implemented the component can be used in any `TGrid` program and can be executed on each subnet, respectively.

2.3. Data redistribution

Data redistribution is a crucial task when executing M-tasks in grid environments. One main goal of the design of `TGrid` is that the programmer should not have to care about the details of data redistribution while developing components. The data redistribution is entirely the responsibility of the runtime system. So, the program developer only haves to define which data has to be moved to some other subsequent component, but the developer is not responsible for organizing the data transfer between components. This

```
<tgridconf>
  <ports>
    <property name=”portmanager” value=”3002”/>
    <property name=”taskrunner” value=”3003”/>
  </ports>
  <contextdefinition>
    <context type=”tcp”/>
  </contextdefinition>
  <subnetmanagers>
    <subnet id=”sn1”>
      <manager>
        <address>192.168.1.1</address>
        <!-- context for inter cluster communication -->
        <contexts>
          <context type=”tcp”>
            <property name=”port” value=”3004”/>
          </context>
        </contexts>
      </manager>
      <clients port=”3000”>
        <client>192.168.1.2</client>
        <client>192.168.1.3</client>
      </clients>
      <env> <property name=”prunjava” value=”prunjava.sh”/> </env>
    </subnet>
    <subnet id=”sn2”>
      <manager>
        <address>cluster</address>
        <contexts>
          <context type=”tcp”>
            <property name=”port” value=”3004”/>
          </context>
        </contexts>
      </manager>
      <clients port=”3000”>
        <client>192.168.2.1</client>
        <client>192.168.2.2</client>
      </clients>
      <env><property name=”prunjava” value=”prunjava.x86_64.suse93.cluster.sh”/></env>
    </subnet>
    <connectors>
      <connector>
        <endpoint id=”sn1”/>
        <endpoint id=”sn2”/>
      </connector>
    </connectors>
  </subnetmanagers>
</tgridconf>
```

Figure 4. Sample TGrid configuration file. The configuration defines the subnets ’sn1’ and ’sn2’ which are controlled by subnet manager 192.168.1.1 and cluster respectively.

dramatically reduces the complexity of writing grid-enabled task-parallel programs. Not only the complexity is reduced, also bugs in the data redistribution are avoided which directly enhances the program development speed.

The basic procedure for data redistribution was summarized by Jeannot and Wagner as (1) data identification, (2) message generation, (3) message scheduling and (4) communication [10]. The `TGrid` redistribution module extends this approach. The resulting steps to perform data redistribution within two components in `TGrid` are:

1. Register the variables with the redistribution thread which are to be moved between the source and the target component.
2. The redistributor collects the relevant meta data (hostnames, ports) of the sending and receiving side.
3. Create a communication schedule which includes the messages that each participating processor has to send or receive.
4. Convert the basic communication schedule to an extended schedule by adding meta data of each client and create proxy objects which perform the sending of messages transparent to the sender, i.e., the sender is not aware whether the receiving processor belongs to its subnets or not.
5. Send message lists to each client with the messages to be sent and to be received by the client.

6. The redistributor waits until it has been notified of the completion from all clients.

There are two different types of redistributions that may occur in TGrid. One is the redistribution inside one subnet which we will refer to as intra-cluster redistribution; the other is the redistribution between different clusters which is referred to as inter-cluster redistribution. The redistribution module of TGrid has been designed to gain high performance. Intra-cluster communication is done by sending messages directly to the receiving processors. Hence, we need an efficient port handling, since receiving data for a component requires to open a unique port. Thus, the sending processors can send directly to the receiving processors without bandwidth bottleneck. Such bottleneck cannot be avoided for inter-cluster redistribution. In this case, the processor sends its data to its local subnet manager which forwards the message to the target subnet. Hence, the bandwidth of the redistribution is limited by the bandwidth of the subnet manager. Since clients may be part of private networks, forwarding and routing messages is the only option to transfer data between processors of different subnets.

As a result, good scheduling strategies and an efficient mapping of tasks to processors and consequently to subnets is important for achieving high-bandwidth redistribution. Thus, the scheduler tries to map subsequent tasks that have a data dependency onto the same subnet if all requirements, like the number of processors, are fulfilled.

3. Experimental results

In this section, we present preliminary results for two applications on a grid system with two subnets.

3.1. Task-parallel implementation of Strassen's matrix multiplication algorithm

To evaluate TGrid we have chosen Strassen's matrix multiplication as first application. As stated in [7], the parallel execution of the matrix multiplication by Strassen is limited by the huge number of data dependencies between each recursion step. Hence, the resulting speedup cannot compete with the speedup of parallel-matrix multiplications on a homogenous environment. However, the Strassen matrix multiplications has many properties which makes it interesting to consider. First, we can decompose the algorithm into several tasks as shown in [9]. Moreover, the TGrid framework makes it easy for the programmer to define recursive algorithms for the grid. Unlike other grid environments, a TGrid program is not statically defined and so, a TGrid program gives the programmer full control of when to stop a recursion during execution. Another very important criterion of grid environments is the ability to redistribute data structures between components. For redistributing data structures like arrays the grid framework has to provide interfaces to select certain data from the source and map this data to the memory of the receiving component. In each recursion step of Strassen, the matrices to multiply have to be decomposed into four sub-matrices and those sub-matrices have to be mapped to a new data location on the target component. As shown in section 2.3, the TGrid framework supports these redistribution requirements (data selection, data mapping) by design.

The TGrid application to compute Strassen's algorithm consists of three components; a component for generating matrices, a component to print matrices, and the main Strassen component. The matrix generation component allocates a distributed matrix on the grid and the print component simply prints the result to a file or to stdout. The components for generating and printing matrices are generic and can therefore be reused by other matrix applications. The print component is the sink of the computation and it is also used as a target for gathering the sub-results. Figure 5 shows the resulting task structure.

Strassen's algorithm computes the result matrix $C = A \times B$ by dividing the matrix C recursively into four subproblems C_{11}, C_{12}, C_{21}, and C_{22}. The sub-problems are solved by seven recursive calls whose results are combined to yield C_{11}, C_{12}, C_{21}, and C_{22}. A TGrid component has been implemented to compute the solution for each of these four sub-problems.

For the sake of simplicity we decided to use only one recursion step for the TGrid program. So, another matrix multiplication algorithm is used to compute the subresult after the cutoff. This can be implemented as a single-processor task or as a multi-processor task. We have implemented a broadcast-multiply-roll style algorithm similar to the algorithm of Fox that is used as multi-processor task. If there is only a single processor to run the task, we simply perform a local matrix multiplication.

The tests were performed with two different grid configurations which are summarized in Table 1. In configuration (1), two subnets are used within the grid environment, namely the subnets denoted as (a) and (b). The subnet (a) consists of two P4 machines that are used to allocate the matrices A and B. The computation of the C_{ij} is performed on subnet (b). Parts of matrices A and B have to be redistributed to the corresponding target processor. Collecting and printing the result on a single processor is part of the program but not part of the measurement presented in the following figures. In configuration (2), the computation of the sub-matrices is done on different subnets. The tasks C_{11} and C_{12} are computed by Dual-Opterons in subnet (b), whereas the other two tasks are executed by P4s in subnet (a). The Dual-Opterons are connected via Gigabit Ethernet, and the subnet that contains the Pentiums is

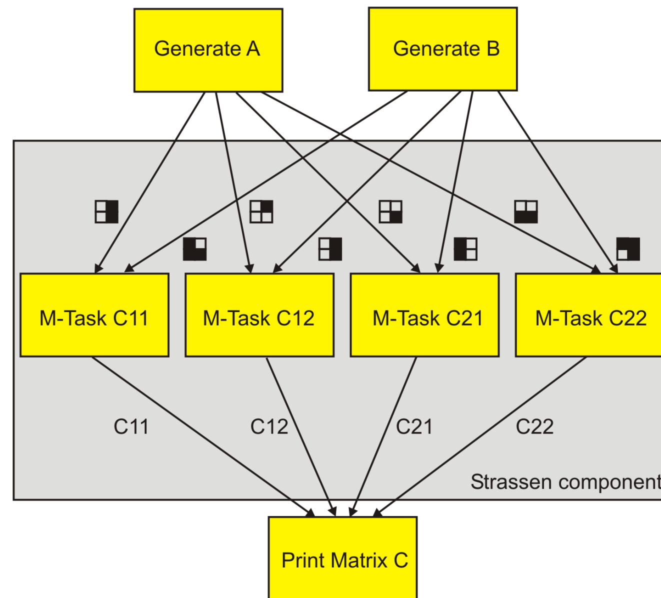

Figure 5. Task structure of the Strassen implementation for TGrid.

connected via Fast Ethernet. Table 1 also contains a column 'processes per node'. The value '2 x 2' means that the component C_{11} is assigned to 2 nodes where each node is running 2 processes and so, the component C_{11} will be executed by 4 processors.

Figure 6 compares the runtime of Strassen's algorithm on a single processor and on 16 processors in a `TGrid` configuration. For both configurations running Strassen's algorithm on `TGrid` will only pay off when an adequate task-size has been reached. We have included the runtime for smaller matrix dimensions (< 1024) only to give an impression about the time consumed for the grid management and the redistribution overhead. The black line denotes the speedup gained for each matrix dimension. It is not surprising that the speedup raises when the matrix dimension increases. In case of configuration (1) we get a speedup of about 7 which is respectable considering the huge number of redistributions that have to be performed before the computation can be started. When different subnets and therefore also different platforms work together as in configuration (2), we get a smaller speedup. The speedup of the entire grid environment is computed by dividing to the execution time on a single Opteron by the parallel execution time on the `TGrid` environment. In our configuration, the Opteron is faster than the Pentium 4 processor and so, the speedup of the heterogenous grid system is smaller.

3.2. Distributed computation of a Mandelbrot set on the grid

The computation of a Mandelbrot set is well-suited for the grid since there are only a few data dependencies between subsequent components. Unlike Strassen, only a few bytes of input data (image boundaries) are required to produce the corresponding part of the Mandelbrot image. However, the computation time of each pixel is irregular and so the decomposition strategy plays an important role for the parallel performance. Since we do not want to examine different decompositions in the first place, we decided to the equally decompose the number of lines onto the number of computation tasks. Although each component has the same number of lines to compute, the lines are mapped in round-robin fashion which avoids having components with a larger number of black bits (many iterations). Therefore, we simply modified sequential Java code and wrapped it inside a `TGrid`-component. The redistribution component of `TGrid` has support for block-partitioned matrices; a line-cyclic distribution of arrays could also be implemented on demand. Using the block-based array redistribution the output component of the Mandelbrot program will produce an image as shown in Figure 7.

The program consists of only two grid components. One is responsible for computing bits of the Mandelbrot set and the other is used for coloring each pixel and writing the image.

For this experiment we connected two `TGrid` subnets. The first subnet consisted of three Opterons running Linux in 64-bit mode. The Opterons are connected via a dedi-

Table 1. TGrid configuration for testing Strassen's matrix multiplication

Configuration (1)

Task	Processes per node	Processor	Subnet
Matrix gen.	1 x 1	P4 3.0	a
Matrix gen.	1 x 1	P4 3.0	a
C11	2 x 2	Opteron 2.0	b
C12	2 x 2	Opteron 2.0	b
C21	2 x 2	Opteron 2.0	b
C22	2 x 2	Opteron 2.0	b

Configuration (2)

Task	Processes per node	Processor	Subnet
Matrix gen.	1 x 1	P4 3.0	a
Matrix gen.	1 x 1	P4 3.0	a
C11	2 x 2	Opteron 2.0	b
C12	2 x 2	Opteron 2.0	b
C21	2 x 2	P4 3.0	a
C22	2 x 2	P4 3.0	a

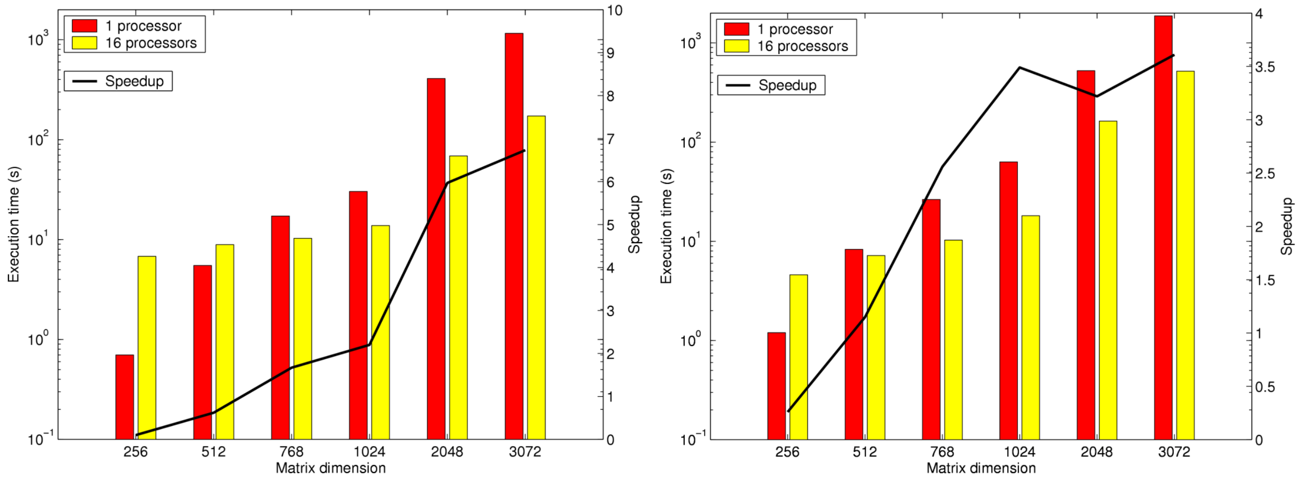

Figure 6. Runtime and speedup of Strassen's algorithm implemented on top of TGrid. On the left: All computations of sub-matrices C_{ij} are done on Opteron-based machines; configuration (1). On the right: Experiments were performed on two different subnets; configuration (2).

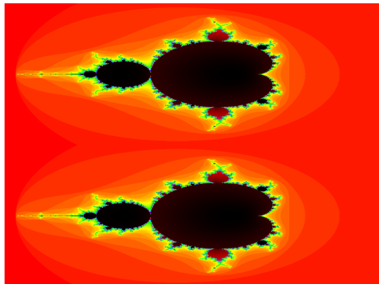

Figure 7. Example output of the TGrid Mandelbrot program computed by 2 spawned tasks.

cated high-speed Gigabit Ethernet switch. The second subnet contained three Pentium 4 at 3 GHz. The Pentiums and both subnets were connected via Fast Ethernet.

Figure 8 shows the experimental results of the Mandelbrot program. The results show that running the program on multiple processors on the grid notably reduced the runtime. A good speedup is achieved at the same time. We could have achieved an even better speedup if we had implemented a suitable load-balancing algorithm as mentioned above. But this is beyond the scope of this article.

4. Related work

Grid computing has been an active area of research in the last few years. Since there is an immense number of active projects, we only refer to grid projects which are directly related to `TGrid`. Similar to `TGrid`, the Ibis [17] programming environment is completely Java-based and offers a multi-layered architecture. The Ibis portability layer is the heart of Ibis. Ibis-based programming frameworks like Satin use this portability layer to abstract from the actual communication platform such as TCP, GM, or MPI.

Another grid programming environment is MPICH-G2 [11] which was also evaluated as communication platform to build `TGrid` upon. Similar to Ibis, there is no easy way

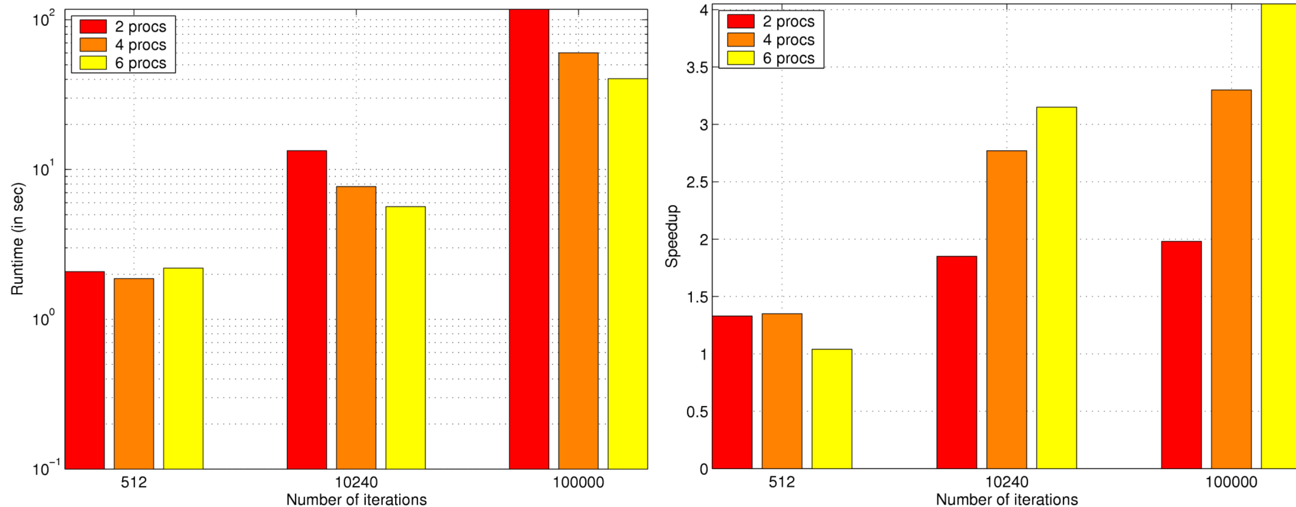

Figure 8. Experimental results of the Mandelbrot program on TGrid. On the left: Execution time with 2, 4, and 6 distributed processors. On the right: Corresponding speedup.

to determine the location of a processing element on the grid when using the communication stack. However, being based on the Globus Toolkit [8] offers a lot of advantages like security, information services, or data management.

The OurGrid project aims to provide computational power for bag-of-tasks applications [1]. Bags of MPI-tasks are also supported. Unlike tasks in `TGrid`, tasks that can be used in such bags-of-tasks are independent of each other.

The GridRPC model [15] provides a standardized and portable programming interface to the remote procedure calls (RPC) mechanism. In this model, a client calls some function on a service provider and the result is returned when the computation is completed. In NetSolve [2, 16], the server provides access to numerical functions like DGSEV (solution to a real system of linear equations). The communication scheme of GridRPC has been improved in [6] by avoiding sending results straight back to the client. Instead, the data is directly transferred to the server that will execute the next function using this result as input data. We have extensively evaluated GridRPC and its related projects to use it as platform for our task-parallel research. The biggest advantage of this approach is the performance that one component can be achieved when all provided functions have been priorly optimized for each computation server. Important parameters can be tuned for best performance like network parameters for parallel computations (logical block size of ScaLAPACK routines [5]) and block sizes of local routines can be adapted which reduces the number of cache misses [18]. On the other hand, this is a very time consuming approach when the grid environment has changed. Optimizing each server requires a lot of skills and experience and is less flexible when new services are required.

Another component-centered architecture like `TGrid` is the common component architecture (CCA) and the DCA (distributed CCA) [3]. The CCA is a component framework definition with three main criteria: a programming-language independent interface definition language, component 'ports' that define which component are able to transfer data and services which are provided by a component. The entire framework is quite complex and hence, it was not applicable to create runtime environment for hierarchically structured task-parallel programs. As for `TGrid`, the $M \times N$ problem (data redistribution) plays an important role for CCA components. The requirements for creating a data redistribution framework for CCA components is addressed in [4].

5. Conclusions

In this article we have presented the heterogeneous runtime environment `TGrid`. `TGrid` supports the execution of hierarchically nested multiprocessor tasks programs with arbitrary dependencies in distributed environments. `TGrid` is a location-aware environment which supports the scheduling of components in the grid. Since the components of `TGrid` can be nested or can be implemented as multi-processor tasks, placing them on the same subnet reduces the communication overhead of the program. Experiments have shown that the `TGrid` can lead to good speedups for suitable applications with coarse-grained tasks.

References

[1] N. Andrade, W. Cirne, F. Brasileiro, and P. Roisenberg. OurGrid: An Approach to Easily Assemble Grids with Equitable Resource Sharing. In *9th Workshop on Job Scheduling Strategies for Parallel Processing*, 2003.

[2] D. Arnold, S. Agrawal, S. Blackford, J. Dongarra, M. Miller, K. Seymour, K. Sagi, Z. Shi, and S. Vadhiyar. Users' Guide to NetSolve V1.4.1. Innovative Computing Dept. Technical Report ICL-UT-02-05, University of Tennessee, Knoxville, TN, June 2002.

[3] F. Bertrand and R. Bramley. DCA: A Distributed CCA Framework Based on MPI. In *18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, pages 90–97, Santa Fe, New Mexico, USA, April 2004. IEEE Computer Society.

[4] F. Bertrand, R. Bramley, K. B. Damevski, J. A. Kohl, D. E. Bernholdt, J. W. Larson, and A. Sussman. 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.

[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1997.

[6] F. Desprez and E. Jeannot. Improving the GridRPC Model with Data Persistence and Redistribution. In *3rd International Symposium on Parallel and Distributed Computing (ISPDC)*, Cork, Ireland, July 2004.

[7] F. Desprez and F. Suter. Impact of Mixed-Parallelism on Parallel Implementations of Strassen and Winograd Matrix Multiplication Algorithms. *Concurrency and Computation:Practice and Experience*, 16(8):771–797, July 2004.

[8] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[9] S. Hunold, T. Rauber, and G. Rünger. Multilevel Hierarchical Matrix Multiplication on Clusters. In *Proceedings of the 18th Annual ACM International Conference on Supercomputing, ICS'04*, pages 136–145, June 2004.

[10] E. Jeannot and F. Wagner. 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.

[11] N. T. Karonis, B. Toonen, and I. Foster. MPICH-G2: a Grid-enabled implementation of the Message Passing Interface. *Journal of Parallel and Distributed Computing*, 63(5):551–563, 2003.

[12] T. Rauber and G. Rünger. 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.

[13] T. Rauber and G. Rünger. Tlib - A Library to Support Programming with Hierarchical Multi-Processor Tasks. *Journal of Parallel and Distributed Computing*, 65(3):347–360, 2005.

[14] T. Rauber and G. Rünger. Anticipated Distributed Task Scheduling for Grid Environments. In *Proc. of the IPDPS Workshop on High-Performance Grid Computing (HPGC)*. IEEE, 2006.

[15] K. Seymour, H. Nakada, S. Matsuoka, J. Dongarra, C. A. Lee, and H. Casanova. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In M. Parashar, editor, *Grid Computing - GRID 2002, Third International Workshop, Baltimore, MD, USA, November 18, 2002, Proceedings*, volume 2536 of *Lecture Notes in Computer Science*, pages 274–278. Springer, 2002.

[16] K. Seymour, A. YarKhan, S. Agrawal, and J. Dongarra. NetSolve: Grid Enabling Scientific Computing Environments. In L. Grandinetti, editor, *Grid Computing and New Frontiers of High Performance Processing*. Elsevier, 2005.

[17] R. V. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a Flexible and Efficient Java based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, 17(7-8):1079–1107, June 2005.

[18] R. C. Whaley and J. J. Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee, 1997.