# Using Static Code Analysis to Improve Performance of GridRPC Applications

Oleg Girko
School of Computer Science and Informatics
University College Dublin
Dublin, Ireland
e-mail: olegs.girjko@ucdconnect.ie

Alexey Lastovetsky
School of Computer Science and Informatics
University College Dublin
Dublin, Ireland
e-mail: alexey.lastovetsky@ucd.ie

*Abstract*—**The paper presents an approach to improve performance of GridRPC applications by statically analysing dynamic workflows. An extension to GridRPC API is used to specify the region of code to apply static code analysis to, during the compilation phase. The information collected is then used at runtime for building a graph of dependencies between tasks, which is analysed to assign servers to tasks in an optimal way, minimising the time of computation and communication. This approach handles branching and looping correctly by building an extended dependency graph, which covers all branches of the code.**

**The experimental results are provided to show that in many practically important cases this approach leads to better results than individual mapping of tasks or run-time task discovery.**

***Keywords- Grid; GridRPC; task mapping; task scheduling; workflows mapping***

## I. INTRODUCTION

GridRPC [1] is a standard API for grid computing, promoted by Open Grid Forum. It specifies a set of functions which can be used to start computations on remote servers and wait for their results. There are several popular implementations of GridRPC API, including GridSolve [2], Ninf-G [3] and DIET [4].

The advantage of GridRPC is its simplicity. Any application written in a modular way can be converted into a Grid client by replacing function calls performing expensive computations with GridRPC calls performing the same computations remotely.

The important feature of GridRPC is that the programmer does not have to specify a server to run a task on: the server can be assigned by middleware. Also, unlike other RPC implementations, GridRPC does not use client-side IDL to build stub functions for RPC calls. All information about remote function argument types is provided by servers at runtime.

The simplicity of GridRPC API leads to some limitations. Remote task calls are independent of each other, and a server is assigned to each task call individually, without taking other tasks into account. This can lead to non-optimal load distribution.

Also, the client-server nature of RPC causes arguments and results of each call being sent and received directly from and to client. This leads to unnecessary communication overhead in a very common case when a result of one task is used by another task, but not directly used on the client. The servers running both tasks can be connected by a high speed link, so direct transfer of data would lead to much better performance than communication through the client.

There are several approaches to improve performance of GridRPC applications by extending GridRPC API, but they either put strict limitations on application code, or require using an additional description of application's performance model.

This paper describes another approach to building application performance model automatically by applying static code analysis to the application's source code. This approach does not incur additional restrictions on application code and does not require manual efforts for describing the performance model.

The rest of the paper is structured as follows. In Section II, existing approaches to mapping remote tasks to servers are discussed, with their advantages and limitations. In Section III, the new approach using static code analysis is presented. In Section IV, the experimental results showing advantages of static code analysis approach over runtime discovery are presented. In Section V, factors contributing to speed improvement are described. Section VI concludes the paper.

## II. EXISTING APPROACHES TO MAPPING TASKS TO SERVERS

### *A. Individual Mapping*

Individual mapping is traditionally used by GridRPC implementations. RPC calls are performed independently from each other, all arguments are sent by the client, and all results are returned to the client.

Figure 1gives an example of application using GridRPC API. First, function handles `t1`, `t2`, `t3` and `p1` are initialised for remote tasks `T1_cond`, `T2_cond`, `T3_cond` and `P1_cond` respectively. Then two instances of remote task `T1_cond` are started in parallel asynchronously: the `grpc_call_async()` function starts a remote task, but does not wait for its completion. Then the program waits for all remote tasks completion using `grpc_wait_all()` function. The remote tasks `T2_cond` and `P1_cond` are run sequentially and synchronously: the `grpc_call()` function starts a remote task and waits for its completion. The result of the remote task `P1_cond` is saved in variable

```
grpc_function_handle_t t1, t2, t3, p1;
grpc_sessionid_t s0, s1;
grpc_function_handle_default(&t1, "T1_cond");
grpc_function_handle_default(&t2, "T2_cond");
grpc_function_handle_default(&t3, "T3_cond");
grpc_function_handle_default(&p1, "P1_cond");
grpc_call_async(&t1, &s0, a0, b0, c0, s, c);
grpc_call_async(&t1, &s1, a1, b1, c1, s, c);
grpc_wait_all();
grpc_call(&t2, c0, c1, d, s, c);
grpc_call(&p1, d, s, c, x,&p);
if (p) {
  grpc_call_async(&t3, &s0, c0, a0, s, c);
  grpc_call_async(&t3, &s1, c1, a1, s, c);
  grpc_wait_all();
} else {
  grpc_call_async(&t3, &s0, c0, b0, s, c);
  grpc_call_async(&t3, &s1, c1, b1, s, c);
  grpc_wait_all();
}
```

**Figure 1. GridRPC program**

`p`, which is used as a condition in the `if` statement. Both branches of this statement run two instances of the same T3_cond remote task in parallel and then wait for their completion, but the input arguments for these tasks are different.

The individual mapping works well when the client starts remote tasks sequentially, when next task is called only after the previous one has finished.

However, non-optimal load distribution is likely when remote tasks are run in parallel. For example, the client can start two tasks, a simple one and a complex one, and then start waiting for their results. When the simple task is started the fastest server will be allocated to this task in order to minimise its execution time. Then, when the complex task is started, the fastest server is already busy, and a slower server will be allocated. If the information about the parallel remote tasks was somehow available prior to mapping the tasks to servers, it would be possible to assign servers in a more optimal way: the slower server to the simpler task, the faster server to the complex task.

Another problem with individual mapping is non-optimal communication. A GridRPC client is usually connected to the computational cluster via public internet, whereas servers inside cluster are interconnected using high-speed local area network. This means that it is much faster to transfer data between servers than to the client and then back to the server in the case when one remote task uses results from another remote task. Unfortunately, the direct server-to-server communication is impossible in the framework of the unmodified GridRPC API.

### B. *Runtime Discovery: SmartGridRPC and SmartGridSolve*

SmartGridSolve [5] is an extension of GridSolve middleware. Its design was inspired by mpC programming language [6]. It implements an extended version of GridRPC API for C programming language called SmartGridRPC [7], and uses runtime discovery for collective mapping of tasks to servers and enabling server-to-server communication to achieve improved performance.

```
grpc_function_handle_t t1, t2, t3, p1;
grpc_sessionid_t s0, s1;
grpc_function_handle_default(&t1, "T1_cond");
grpc_function_handle_default(&t2, "T2_cond");
grpc_function_handle_default(&t3, "T3_cond");
grpc_function_handle_default(&p1, "P1_cond");
grpc_map("ex_map") {
  grpc_call_async(&t1, &s0, a0, b0, c0, s, c);
  grpc_call_async(&t1, &s1, a1, b1, c1, s, c);
  grpc_wait_all();
  grpc_call(&t2, c0, c1, d, s, c);
  grpc_call(&p1, d, s, c, x,&p);
}
if (p) {
  grpc_map("ex_map") {
    grpc_call_async(&t3, &s0, c0, a0, s, c);
    grpc_call_async(&t3, &s1, c1, a1, s, c);
    grpc_wait_all();
  }
} else {
  grpc_map("ex_map") {
    grpc_call_async(&t3, &s0, c0, b0, s, c);
    grpc_call_async(&t3, &s1, c1, b1, s, c);
    grpc_wait_all();
  }
}
```

**Figure 2. SmartGridRPC program**

In SmartGridSolvethe collective mapping applies to a C block marked by a`grpc_map()` preprocessor directive.

Figure 2 shows an example of the same algorithm as in Figure 1 but changed to use SmartGridRPC API. Three regions of code are marked for collective mapping. The reason why the whole algorithm is not marked for collective mapping will be explained later.

Internally, the `grpc_map()` directive is implemented as a `while` loop which executes its body block twice. The first pass is called *discovery phase*. During this pass all GridRPC calls are recorded but not executed. After the first loop pass, the mapping takes place. The client sends the list of remote tasks to the SmartGridSolve agent along with information about their arguments. The agent uses this information to build the graph of dependencies between tasks, which is sent back to the client along with information about servers which can perform those tasks, their performancecharacteristics and the throughput of links between them. This information is used by the client to map tasks to servers for optimal computation and communication performance. The actual computation is done on the second pass of the loop, which is called *execution phase*.

The approach used in SmartGridSolve is simple and efficient, but it puts significant restrictions onthe code inside the block marked for collective mapping. The fact that the code runs twice means that all side effects will happen twice as well. Therefore, the local code should either have no side effects, or those side effects should yield the same result when the code is run more than once. There is a workaround for this: a block of code can be marked with a SmartGridRPC `grpc_local()` directive, making this block be executed only during the execution phase. However, the way fault tolerance is implemented in SmartGridSolve does not guarantee that this block is run

```
grpc_function_handle_t t1, t2, t3, p1;
grpc_sessionid_t s0, s1;
grpc_function_handle_default(&t1, "T1_cond");
grpc_function_handle_default(&t2, "T2_cond");
grpc_function_handle_default(&t3, "T3_cond");
grpc_function_handle_default(&p1, "P1_cond");
grpc_map("ex_map") {
  grpc_call_async(&t1, &s0, a0, b0, c0, s, c);
  grpc_call_async(&t1, &s1, a1, b1, c1, s, c);
  grpc_wait_all();
  grpc_call(&t2, c0, c1, d, s, c);
  grpc_call(&p1, d, s, c, x,&p);
  if (p) {
    grpc_call_async(&t3, &s0, c0, a0, s, c);
    grpc_call_async(&t3, &s1, c1, a1, s, c);
    grpc_wait_all();
  } else {
    grpc_call_async(&t3, &s0, c0, b0, s, c);
    grpc_call_async(&t3, &s1, c1, b1, s, c);
    grpc_wait_all();
  }
}
```

**Figure 3.SmartGridRPC program with side effects**

only once. If an error happened during the execution phase, all subsequent GridRPC calls and `grpc_local()` blocks would be skipped and the loop would be run once again from the beginning, with a different mapping. This leads to `grpc_local()` blocks, which were run before the GridRPC call where an error happened, to be run once again. Only those `grpc_local()` blocks which are located after the last GridRPC call are guaranteed to be run once.

There is another important restriction on the code inside the `grpc_map()` block. The workflow of the code during discovery and execution phases should be exactly the same. This means that all loops inside `grpc_map()` block should have exactly the same number of iterations during discovery and execution phase, and all branches should be exactly the same as well. As a result, the flow control inside the`grpc_map()` block should be pre-determined before the execution phase, and thus not dependent on remote task execution results.

The violation of the restrictions on code mentioned above can lead to disastrous results, causing a program to behave in unexpected way. It is very easy to make a mistake and violate those restrictions, and there is no automatic detection of such mistakes.

An example of such violation is shown on Figure 3. The difference between the original SmartGridRPC code on Figure 2 and this example is that the whole algorithm is marked for collective mapping. This will not work correctly with runtime discovery.Flow control of the code inside the `grpc_map()` block depends on the value of variable p, which depends on the result of a remote task call. This result is unknown during the discovery phase because GridRPC calls are not being executed during this phase. Hence, it is impossible to make the workflow inside the `grpc_map()` block run the same way during both iterations. The only way to make this algorithm to run correctly with SmartGridSolve is to map both branches of the conditional statement separately, as shown on Figure 2.

```
modulecndalg(int s, int c, double x,
intcndtrue, intcndfalse)
{
  component:
  task "tgtest_cond.idl"
  T1_cond,T2_cond,T3_cond,P1_cond;

OBJ:
  DOUBLE(s)a0, a1, b0, b1, c0, c1, d;
  INTEGER p;

algorithm:
  parallel {
    T1_cond:(a0, b0,@s,@c)->(c0);
    T1_cond:(a1,b1,@s,@c)->(c1);
  }
  T2_cond:(c0, c1,@s,@c)->(d);
  P1_cond:(d,@s,@c,@x)->(p);
  parallel {
    if (cndtrue)
      parallel {
        T3_cond:(a0, c0,@s,@c)->(a0);
        T3_cond:(a1,c1,@s,@c)->(a1);
      }
    if (cndfalse)
      parallel {
        T3_cond:(b0, c0,@s,@c)->(b0);
        T3_cond:(b1, c1,@s,@c)->(b1);
      }
  }
}
```

**Figure 4. ADL specification**

### C. *Static Discovery with External Description: ADL*

Another approach to collective mapping is the use of ADL [8], the Algorithm Definition Language. It uses an extension to GridRPC API similar to pure SmartGridRPC. The block of code for collective mapping is also specified using a `grpc_map()` directive, but the code inside this block will be run only once. The task dependency graph and application's performance model are specified separately, using ADL.

Figure 4 gives an example of an ADL specification for the algorithm in Figure 3. The component section specifies the remote tasks required for the algorithm. The OBJ section specifies non-scalar objects used in the algorithm. The algorithm section describes the workflow of the algorithm, the order of remote task execution and what arguments are involved in this execution. The specification in this example has 5 parameters. The complexity of tasks and the workflow of the algorithm are dependent onthe actual values of these parameters, which are specified at runtime. For example, the size of vectors depends on parameter `s`. Parameters `cndtrue` and `cndfalse` specify the likelihood of actual execution of each branch of the conditional statement in the algorithm. If both of those parameters have nonzero value at runtime, both branches are mapped as if they are executed in parallel.

The advantage of this approach over runtime discovery is the absence of restrictions on the code of the`grpc_map()` block. This means that it allows collective mapping of iterative algorithms with loops having the number of

iterations dependent on remote task results, and conditional algorithms with branching dependent on remote task results. Hence, the algorithm in Figure 3works correctly when using ADL specification in Figure 4.

The disadvantage of this approach is that it requires a programmer to describe the mapping scenario of the algorithm using ADL in addition to the program itself. This means that significant additional efforts are needed to enable more efficient mapping and to keep the program and its ADL description in sync. If the program and ADL diverged somehow, the mapping will be non-optimal, and there is no way to check for this problem automatically.

## III. Using Static Code Analysis For Collective Mapping

The approach proposed in this paper is an attempt to combine advantages of both pure SmartGridSolve and ADL-enabled SmartGridSolve, while avoiding their disadvantages. This is achieved by using static code analysis to extract from the application code itself as much information as possible in order to build the task dependency graph before the execution of the application, without a separate run-time discovery phase. Like the ADL-based approach, this approach does not incur limitations on code side effects imposed by pure SmartGridSolve and allows for loops and branches.On the other hand, it does not require an additionalspecification of the algorithm in ADL and eliminates the problem of synchronization of the algorithm specification with the application code.

The proposed approach is implementedas a modified version of SmartGridSolve, providingSmartGridRPC API with minor extensions. It accepts any GridRPC source code with or without SmartGridRPC extensions:`grpc_map()` and `grpc_local()` blocks.

For example, code on Figure 3 works correctly with static code analysis approach, but there is no need to supply this code with additional algorithm specification, like the one presented in Figure 4.

Static analysis is applied to the source code before its compilation to extract as much information as possible about the algorithm. The extracted information is functionally equivalent to ADL specification, but uses different format.Then the code is modified to add the following stages before the algorithm is run:

1. building application performance model;
2. building extended dependency graph;
3. using mapping heuristics.

These stages use the information collected during the static code analysisfor making the optimal decision on task-to-server mapping and server-to-server communication.

Static code analysis and the runtime stages are described in details below.

### A. Analysing Code To Find GridRPC Calls

The algorithm of the block for collective mapping is analysed on the source code level using Clang [9], a frontend for the C family of languages (C, C++, Objective C), which is a part of LLVM [10], a toolkit for building compilers. Clang is implemented as a modular library, allowing using it as a toolkit for analysing C code.

First, all source code modules are parsed by Clang generating their respective Clang-specific internal AST (abstract syntax tree) representations.

Then, these abstract syntax trees are analysed together in order to find blocks marked by `grpc_map()` directives. These blocks are analysed for conditional statements, loops and function calls. GridRPC function calls are being recorded; all other function calls are analysed recursively.

Arguments of GridRPC calls are analysed across function calls to find if they reference the same values even if they are used deeply inside function call hierarchy. First arguments of GridRPC calls are analysed to find the corresponding function handle initialisation functions, and thus to find the names of remote tasks which are run by those calls. All other arguments are analysed and the numerical ID values are assigned to them so that the same arguments had the same ID values.

The analysing algorithm tries to guess the number of loop iterations and the values of scalar GridRPC call arguments which determine the size of other non-scalar arguments (vectors and matrices). The warning is produced if the algorithm is unable to guess these values. In this case the programmer can either simplify algorithm or provide the most likely value using `grpc_likely()` directive. This is similar to specifying these values in ADL.

Also, the analysing algorithm tries to guess which branch is executed in conditional statement if this is dependent on a value computed outside of `grpc_map()` block only. Otherwise, both branches of the conditional statement are analysed as if they are run in parallel.

The result of this analysis is a C code with `grpc_map()` blocks modified by adding static variable declarations initialised with information collected during static code analysis and code to build application performance model at runtime. The information stored in these static variables is used during runtime to build the dependency graph for the `grpc_map()` block. This information is a parameterised application performance model, which can be used later to build actual performance model by substituting parameters with actual runtime values. Functionally this information is similar to algorithm description used in ADL approach, but without the need to write this description manually.

### B. Building Application Performance Model

When the program is run, it uses the parameterised performance model built during static code analysis and stored in global variables to build actual application performance model at runtime. The parameters are substituted with actual runtime values. Then the resulting performance model is sent to the SmartGridSolve agent for building the extended task dependency graph.

### C. Building Extended Dependency Graph

The SmartGridSolve agent uses the application performance model to build a task dependency graph by analysing the order of tasks to be run and their arguments.

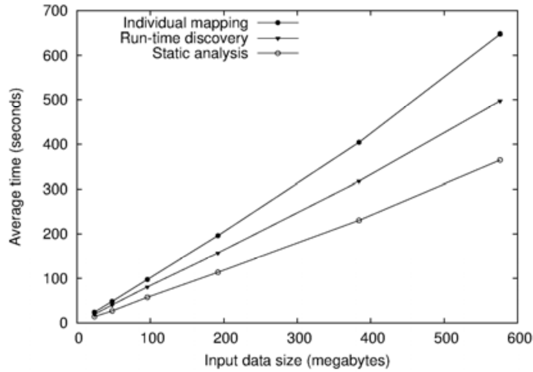


**Figure 5. Experimental results**

The task A is considered dependent on task B if it is started later than task B and uses a task B's output argument as its input argument. The resulting task dependency graph is sent back to the client along with a list of servers the tasks can be run on and those servers' performance characteristics.

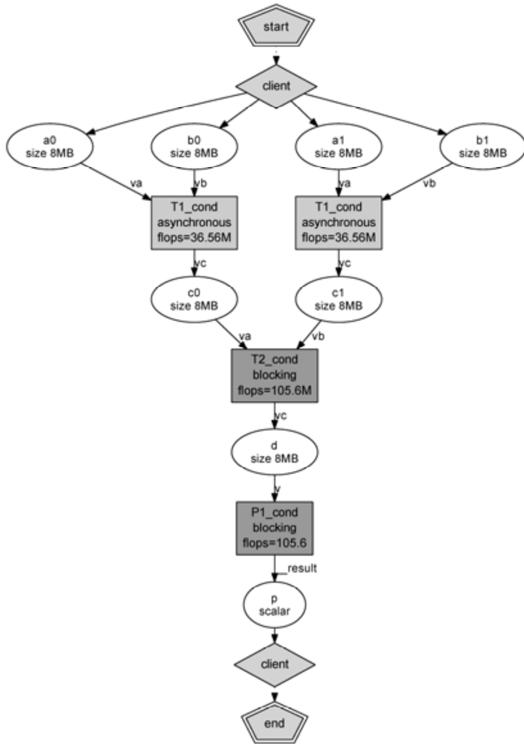


**Figure 6. Task graph for the first region of code**

This stage is the same as in unmodified SmartGridSolve, but with one important difference. The dependency graph is extended, covering the remote tasks which can potentially be run. It is however possible that some tasks in this dependency graph will be skipped during the actual algorithm execution. This can happen if there are branches in the algorithm's code.

### *D. Using Mapping Heuristics*

The task dependency graph is used to build a mapping between tasks and servers the same way as SmartGridSolve client does. The heuristic specified by `grpc_map()` argument is used for this task. The optimal mapping algorithm is NP-complete, so it is the programmer's responsibility to choose the heuristic which produces best results in acceptable time [11].

## IV. Experimental Results

The proposed approach based on the static code analysis outperforms the runtime discovery approach used in unmodified SmartGridSolve in many important cases. It allows for applying the collective mapping to larger regions of code when loops or branches dependent on remote task results are present. For example, the code in Figure 3will not work correctly with pure SmartGrisSolve, so it should be modified to reduce the size of code regions marked for collective mapping to contain no flow control dependent on remote task results, leading to the code shown in Figure 2, which produces much less optimal mapping.

To validate these performance advantages experimentally, the algorithm in Figure 1, Figure 2 and Figure 3 was tested using SmartGridSolve with individual mapping, SmartGridSolve with runtime discovery and SmartGridSolve modified for the static analysis approach

**Figure 7. Task graph for `then` branch of `if` statement**

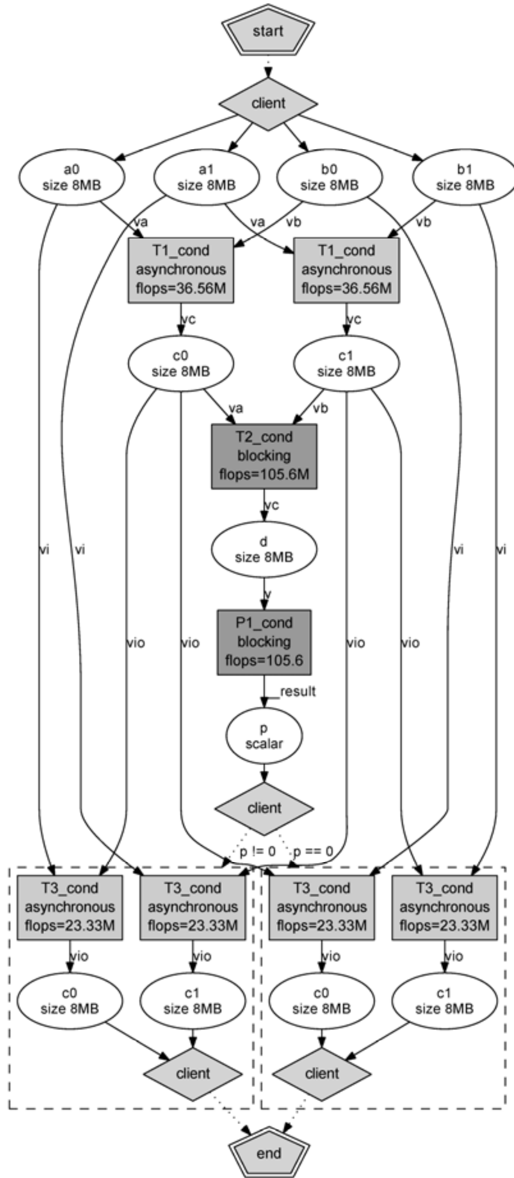


**Figure 8. Task graph for the whole algorithm**

respectively.

The hardware setup used in the experiments consists of 4 heterogeneous servers with performance ranging from 422 to 531 MFlops and 1GB of memory each interconnected with 1Gbit/s Ethernet switch and a client machine which has 100Mbit/s connection to server network.

Figure 5shows the results of the experiment with input data sizes 24, 48, 96, 192, 384 and 576 megabytes. The average time is calculated from 10 executions for each data size and implementation.

The experimental results show that the larger region of code for collective mapping allowed by static code analysis approach consistently yields better performance for all input data sizes.

Figure 6 and Figure 7 show the task graphs generated for two regions of code marked for collective mapping in algorithm in Figure 2, before the conditional statement and inside its then branch. Solid arrows show data communications between the client and remote task and between remote tasks. These arrows are labelledwith the names of input and output arguments. The oval shapes represent variables in the program. The grey squares represent remote tasks.

Arrows between remote tasks going through a variable represent server-to-server communication. Although it looks like one remote task stored its result in the variable and another remote task received its argument from this variable in the program code, the client is not involved in this communication, and the result is directly sent between servers running remote tasks. Only arrows going from and to grey diamonds represent communications involving the client.

Figure 8 shows the task graph generated for the whole algorithm in Figure 3. The resulting communication is much more optimal. For example, `c0` and `c1` arrays are sent directly from servers running task `T1_cond` to servers running tasks `T3_cond`, which does not happen when conditional statement's branches are mapped separately. Also, arrays `a0`, `a1`, `b0` and `b1` are being sent to servers running tasks `T1_cond` and `T3_cond` in parallel, further improving communication speed.

## V. Factors Contributing To Speed Improvement

Although the detailed analysis of speed improvement provided by collective mapping is not given here, we did it in another paper [12], which shows that there are three factors contributing to it.

- The primary factor is server-to-server communication. Computational grid usually consists of nodes connected by high-speed local area network, whereas client connects to the grid using low-speed Internet connection. Direct server-to-server communication allows avoiding sending intermediary results through low-speed link to the client or even avoiding sending data at all in cases when tasks run on the same server. ADL and static code analysis approaches allow extending theregion of code for collective mapping to cover loops and branches, which in turn allows more data to be covered by server-to-server communication.
- The secondary factor is better distribution of computational resources. This factor becomes more prominent in case of smaller data size or faster link between client and grid. Extended regions of code for collective mapping provided by ADL and static code

analysis approaches allow to take more tasks into account and therefore produce better mapping leading to better distribution of computational resources.
- The tertiary factor is the direct result of server-to-server communication: avoiding using client's memory resources to store intermediary results. Clients are usually just regular desktop or laptop computers, whereas grid nodes are dedicated servers with vast amounts of memory. Storing intermediary result on a client just for sending to another server is not only inefficient way of communication; it also can lead to paging on the client, slowing down the whole algorithm.

## VI. Conclusions

In this paper, we have proposed the new approach to mapping tasks to servers in GridRPC algorithms using static code analysis. This approach lifts restrictions on the code imposed by the runtime discovery approach and provides better performance by allowing mapping the whole algorithms with flow control dependent on remote task results, not just regions with predetermined flow control.

## Acknowledgment

This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant Number 08/IN.1/I2054.

## References

[1] K. Seymour et al., "Overview of GridRPC: A Remote Procedure Call API for Grid Computing," *Proceedings of the Third International Workshop on Grid Computing*,, *Lecture notes in computer science*, vol. 2536, pp. 274-278, 2002.

[2] A. YarKhan, K. Seymour, K. Sagi, Z. Shi, and J. Dongarra, "Recent Developments in GridSolve," *International Journal of High Performance Computing Applications*, vol. 1, no. 20, pp. 131-141, 2006.

[3] Y. Tanaka, H. Nakada, S. Sekiguchi, T. Suzumura, and S. Matsuoka, "Ninf-G: A Reference Implementation of RPC-based Programming Middleware for Grid Computing," *Journal of Grid Computing*, vol. 1, no. 1, pp. 41-51, 2003.

[4] E. Caron and F. Desprez, "DIET: A Scalable Toolbox to Build Network Enabled Servers on the Grid," *International Journal of High Performance Computing Applications*, vol. 3, no. 20, pp. 335-352, 2006.

[5] T. Brady, M. Guidolin, and A. Lastovetsky, "Experiments with SmartGridSolve: Achieving higher performance by improving the GridRPC model," *Proceedings of the 9th IEEE/ACM International Conference on Grid Computing (Grid2008)*, pp.49-56, 2008.

[6] D Arapov, A Kalinov, A Lastovetsky, A Ledovskih, and T Lewis, "A programming environment for heterogeneous distributed memory machines," in *Proceedings of the 6th IEEE Heterogeneous Computing Workshop (HCW`97)*, Geneva, Switzerland, 1997, pp. 32-45.

[7] T. Brady, J. Dongarra, M. Guidolin, A. Lastovetsky, and K. Seymour, "SmartGridRPC: The New RPC Model for High Performance Grid Computing," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 22, pp. 2467-2487, 2010.

[8] M. Guidolin, T. Brady, and A. Lastovetsky, "How Algorithm Definition Language (ADL) Improves the Performance of SmartGridSolve Applications," *Proceedings of the 7th High-Performance Grid Computing Workshop*, Atlanta, USA, 2010.

[9] C. Lattner, "LLVM and Clang: Next Generation Compiler Technology," in *The BSD Conference*, Ottawa, Canada, 2008.

[10] C. Lattner, "Introduction to the LLVM Compiler System," in *XII International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, Erice, Sicily, Italy, 2008.

[11] T. Braun et al., "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *Journal of Parallel and Distributed Computing*, vol. 6, no. 61, pp. 810-837, 2001.

[12] M Guidolin and A Lastovetsky, "Grid-Enabled Hydropad: a Scientific Application for Benchmarking GridRPC-Based Programming Systems," *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, 2009.