UNIVERSITY COLLEGE DUBLIN

Using Static Code Analysis for Improvement of Programmability and Performance of GridRPC-Based Applications

Oleg Girko

This thesis is submitted to
University College Dublin
for the degree of
Doctor of Philosophy in Computer Science
in
College of Science

September 2014

School of Computer Science and Informatics
Supervisor: Alexey Lastovetsky
Head of School: Pádraig Cunningham

Acknowledgements

I would like to thank my supervisor Alexey Lastovetsky for offering me opportunity to work on my PhD in Heterogeneous Computing Laboratory. It would be impossible to complete this work without his guidance, support and advise.

Thanks to everyone from HCL group: Amani, Ashley, David, Jean-Noël, Jun, Ken, Khalid, Kiril, Robert, Tania, Vladimir, Ziming. It was great pleasure to work with such talented people.

Thanks to University College Dublin for becoming my Alma Mater and providing me facilities for productive work.

I would like to also thank Science Foundation Ireland, who funded my research (Grant Number 08/IN.1/I2054).

And of course, I would like to thank my friends and family for supporting and encouraging me during this long journey.

Abstract

This thesis presents static code analysis approach to gathering information about remote tasks to be executed by applications using GridRPC API for collective mapping.

Collective mapping of tasks to servers has significant performance benefits over mapping individual tasks independently. However, it requires information about remote tasks an application is going to execute to be available before the application starts executing these remote tasks.

This thesis presents static code analysis approach, which allows collecting essential part of this information by analysing code during compilation. The application is modified automatically by preprocessor to utilise collected information and build optimal task-to-server mapping during runtime.

Advantages of static code analysis over other approaches to collective mapping are outlined, and experimental results using real world application are presented.

Contents

1 Introduction **1**
1.1 Clouds and Grids 1
1.2 A popular programming model: RPC 2
1.3 GridRPC: RPC for grids 6
1.4 GridRPC implementations 10
1.5 GridRPC limitations 11
1.6 Beyond GridRPC: collective mapping 12
1.7 Contributions 12
1.8 Structure 12

2 Motivation **15**
2.1 GridRPC program example 15
2.2 First problem: non-optimal task-to-server mapping 17
2.3 Second problem: non-optimal communication 17
2.4 Third problem: lack of communication parallelism 18
2.5 Solution: collective mapping 19

3 Related research **21**
3.1 Approaches to grid optimisation before collective mapping 21
3.1.1 Task farming 21
3.1.2 Task sequencing 22
3.1.3 Distributed storage infrastructure 22
3.1.4 Arranging server-to-server communication manually 23
3.2 Advantages of collective mapping 24
3.3 Information needed for collective mapping 25
3.4 Runtime discovery 26
3.4.1 Basic Functionality 26
3.4.2 Fault Tolerance 28
3.4.3 Limitations 29
3.5 ADL 31
3.5.1 Basic Functionality 31

3.5.2 Limitations 34
3.6 Workflow submission 34
3.6.1 MA_{DAG} 34
3.6.2 Gwendia 36
3.6.3 Limitations 38

4 Static code analysis 39
4.1 Overview 39
4.2 Runtime parameters 41
4.3 Analysing and preprocessing code 42
4.3.1 Application build process 42
4.3.2 Information gathered during static code analysis 45
4.3.3 Expanded expressions 49
4.3.4 Preprocessing source files by inserting data definitions and code 50
4.4 Extended client library 53
4.5 Running application 54
4.5.1 Building application performance model 54
4.5.2 Building task dependency DAG 56
4.5.3 Applying heuristic 56
4.5.4 Server-to-server communication 58
4.6 Restrictions on code to be analysed statically 58

5 Use case: HydroPad 61
5.1 HydroPad overview 61
5.2 Modifications to HydroPad 64
5.3 Experimental results 65
5.3.1 Hardware configuration 68
5.3.2 Experimental data 68
5.3.3 Experimental results 68

6 Conclusions 73
6.1 Future work 74
6.1.1 Improving static code analyser to cover more cases 74
6.1.2 Introducing fault tolerance support 76
6.1.3 Improving portability 76
6.1.4 Extending analyser to support more programming languages 77

A Extended API reference 79
A.1 grpc_map_static() 79
A.1.1 Synopsis 79

A.1.2 Description 79
A.2 grpc_likely() 79
A.2.1 Synopsis 79
A.2.2 Description 80

B Static code analyser and preprocessor reference 81
B.1 Synopsis 81
B.2 Options 81
B.3 Description 81

C HydroPad source code for runtime discovery 83
C.1 Source code for `main()` function 84
C.2 Source code for algorithm initialisation 87
C.3 Source code for simulation step 89

D HydroPad source code for static code analysis 93
D.1 Source code for `main()` function 94
D.2 Source code for algorithm initialisation 98
D.3 Source code for simulation step 100

List of Figures

1.1 The GridRPC model 6

4.1 Regular application build process 42
4.2 Application build process for static code analysis 43
4.3 Task dependency graph generated for example application 57

5.1 HydroPad application workflow 63
5.2 Wrong HydroPad application workflow detected without adaptation 66
5.3 HydroPad task dependency graph 67

Listings

1.1 GridRPC Program Example . . . 8
2.1 GridRPC Program . . . 16
3.1 SmartGridRPC Program . . . 27
3.2 SmartGridRPC Program With Undetermined Data Flow . . . 30
3.3 SmartGridRPC Program For ADL . . . 32
3.4 ADL Specification . . . 33
3.5 MA_{DAG} Workflow Language Example . . . 35
3.6 Gwendia Language Example . . . 37
4.1 SmartGridRPC Program For Static Code Analysis . . . 40
4.2 Makefile For Building Regular Application . . . 43
4.3 Makefile For Building Application With Static Code Analysis . . . 44
4.4 Example of Algorithm Tree structure . . . 48
4.5 Code Inserted By Preprocessor . . . 51
C.1 HydroPad source code: `main_smart.c` . . . 84
C.2 HydroPad source code: `initialize_smart.c` . . . 87
C.3 HydroPad source code: `evolve_smart.c` . . . 89
D.1 HydroPad source code: `main_static.c` . . . 94
D.2 HydroPad source code: `initialize_static.c` . . . 98
D.3 HydroPad source code: `evolve_static.c` . . . 100

Chapter 1

Introduction

Performance of computers keeps growing exponentially since the early years of computing. However, a single computer is still not powerful enough to solve problems dictated by modern science and technology. Advances in modern networking and computing architecture allow interconnecting multiple general-purpose processing units (CPUs) and special-purpose processing units (like GPUs, for example) efficiently to work in parallel to solve complex tasks. As a result, parallel computing became a very important topic in this time.

1.1 Clouds and Grids

Cloud infrastructure working on computational clusters consisting of thousands of nodes is able to process online transactions simultaneously for millions of customers of web search, social networks, remote file storage, video and audio streaming and many other services. Most of clusters serving cloud services are centralised (controlled by single commercial companies), dedicated to the service they provide and optimised for processing a steady stream of short transactions, mostly through HTTP interface.

Scientific computations usually have a different nature. Each computation can be split into separate tasks which can be computed in parallel, but these tasks are themselves very different than short online transactions. These tasks can require various computational resources, and some of them can take a long time to complete even on modern hardware. For example, the fastest known square matrix multiplication algorithm has complexity of $O(n^{2.3728639})$ for matrices of $n \times n$ size, so it can require quite a lot of computational resources for sufficiently large values of n. Also, scientific institutions usually have no financial resources to build huge clusters dedicated to scientific computations which can be matched by number of nodes to leaders of commercial cloud services. On the other hand, in-

stead of competing like commercial entities do, scientific institutions have good cooperation, so they can share their resources to build infrastructure for parallel computations.

As a result, natural model for scientific computation is quite different: a grid rather than a cloud. Grid is a very popular form of distributed computing, which is designed for non-interactive workloads with large number of tasks run in parallel to achieve a common goal as part of a parallel algorithm. Unlike clusters dedicated to clouds, grids are usually built using heterogeneous approach. They consist of nodes having different computational power or even different processor architecture, and these nodes are interconnected by networks having various bandwidth. Also, grids are often spread between different geographical locations, so some of grid parts are interconnected via public Internet rather than dedicated local area network [48].

1.2 A popular programming model: RPC

There are different ways to write parallel algorithms for grids [36]. There are specialised languages and compilers, libraries for existing programming languages, web services.

One of the popular programming models used for parallel computing is RPC (remote procedure call) [5]. This programming model is implemented as library API for existing programming languages. To run a remote task, a client performs local function (procedure) or method call.

A remote task name can be either specified (directly or indirectly) as an argument of this function, or derived from function or method name. In the later case mapping between functions or methods and remote procedure names has to be somehow established. There are two common approaches for that.

Using dynamic binding.

Interpreted programming languages allow to create objects or functions dynamically. Client library can request server about available remote procedures and create all necessary functions or an object with respective methods using this information, making newly created functions or object available to the client program.

Also, popular interpreted programming languages have a feature to intercept a call for unknown object's method (the one which was not explicitly defined or inherited) and preform some action instead of throwing an error or exception. This action has information about name of the method which was attempted to be called, so it can use this name as a remote procedure name.

Using IDL (interface definition language).

External language to define interface to remote service can be used to generate stub functions for the client side, each of which perform remote procedure call using respective remote procedure name and pass its arguments to remote server (probably converting them to external representation).

Also, IDL can be used to generate server-side skeleton functions or methods which can be used as placeholders to help programmer to write server-side part by filling in these placeholders with actual code preforming actual remote computations.

A server to run the task on can also be specified as a function argument, or assigned automatically. Also, server name or other server location data can be stored in a handle which is passed as a function argument or in an object which is used to invoke a method.

Usually remote procedure's arguments are specified by using function's or method's arguments. Alternatively, argument list can be formed dynamically inside some data structure which is passed as an argument. Remote procedure's result can be stored in function's or method's return value or output arguments.

Running remote task is performed with these steps:

1. converting input arguments from internal representation to external server-specific one and sending them to the remote server;
2. remote task execution;
3. receiving results from the remote server, converting them back to client's internal representation.

There are many different RPC protocols and their implementations. Some of popular general-purpose RPC protocols which are used for invoking remote services (not just grid computing) are listed below.

ONC RPC (Sun RPC) [38]

Initially implemented for SunOS and Solaris operating systems designed by Sun Microsystems, ONC (Open Network Computing) RPC became de facto standard in Unix and Unix-like operating systems, and its support is included in standard C libraries supplied with them. Popular services like NFS [40] and NIS [42] are implemented on top of ONC RPC. ONC RPC has a standard API for C programming language. Remote service is identified by unique program number and version number. Remote procedure is identified by procedure number which is unique inside remote service. There is high-level and low-level API for ONC RPC. The high-level API

uses stub functions generated by `rpcgen` program from IDL. These functions are convenient wrappers for remote procedures, passing their arguments and converting them to external data representation (XDR). The low-level API uses `callrpc()` function. In this case programmer is responsible for specifying remote program, version and procedure number as well as pointers to input and output conversion functions. Server name should be specified explicitly in either case, as first argument of `callrpc()` function or stub functions generated by `rpcgen`.

XML-RPC[35]
XML-RPC is a simple standard way to encode function calls with their arguments as well as call results as XML documents. Usually remote procedure calls using XML-RPC are preformed using HTTP requests, and results are returned in HTTP response body, but there are implementations using different communication transports (like XMPP). There is no standard API for XML-RPC, but there are popular libraries for different programming languages, each with its own API.

SOAP [25]
SOAP is a standard protocol for web services. Like XML-RPC, it uses XML for encoding remote calls and their results, but it uses more complex and structured XML with namespaces, relying on XML Infoset for its message format. There is XML-based IDL for SOAP called WSDL, which can define not just data types and remote methods with their arguments, but also URLs of web services which serve these remote methods. There is no standard API for XML-RPC, but there are popular libraries for different programming languages each with its own API, some of them using stub functions generated from WSDL.

JSON-RPC [18]
JSON-RPC is another standard way to encode function calls with their arguments as well as call results in JSON format. Usually remote procedure calls using JSON-RPC are preformed using HTTP requests, and results are returned in HTTP response body. but other transports can be used, like plain TCP connection. As with XML-RPC, there is no standard API for JSON-RPC, but there are popular libraries for different programming languages each with its own API.

Java RMI [21]
Java RMI (Remote Method Invocation) is a Java-specific object-oriented RPC for invoking methods of remote Java classes. It supports sending serialised Java classed and distributed garbage collection. Also, it can perform

RPC over IIOP for interoperability with CORBA. Java RMI has a standard API for Java programming language. It doesn't need IDL to generate stub methods because Java is interpreted language, so it can generate objects with needed methods dynamically.

CORBA (GIOP)

General Inter-ORB Protocol (GIOP) is a family of protocols used for CORBA [24]. It has several variants which use different transports for communication, for example:

IIOP: TCP connection;

SSLIOP: TCP connection encrypted with SSL;

HTIOP: HTTP request-response.

CORBA (Common Object Request Broker Architecture) is distributed object model allowing unified access to objects and their methods no matter whether an object's implementation resides in program's own address space of on a remote server. Objects are managed by Object Request Brokers (ORBs), which provide object references to the client program. Access to objects managed by the ORB located in program's address space are provided to the client program directly. Requests to methods of objects managed by other ORBs are send by local ORB to remote ORB using RPC provided by variants of GIOP. There are standard APIs for many programming languages which use IDL to generate client-side stub functions or methods.

D-Bus [37]

Although not designed for distributed computing, D-Bus is a notable example of object-oriented RPC designed for communication between programs inside an operating system. It uses Unix-domain sockets, an interprocess communication mechanism used in Unix-like operating systems, as its transport. D-Bus protocol uses simple binary encoding for data transfer. All objects have unique pathnames. Set of functions and their signatures provided by an object is determined by interface name and XML-based IDL associated with it. There is a low-level API for C and C++ programming languages provided by libdbus and libdbus-c++ libraries. Also, there is more high-level API provided by Qt toolkit which provides several ways to call remote methods, either by specifying object, interface and method names explicitly or by using stub methods generated by `qdbusxml2cpp` IDL compiler.

RPC is a simple and straightforward programming model, which makes it quite popular in many areas of distributed computing, not just grid computing.

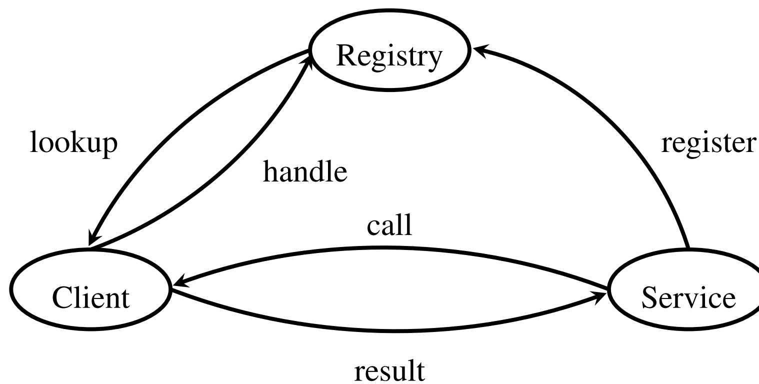


Figure 1.1: The GridRPC model

1.3 GridRPC: RPC for grids

Unlike general-purpose RPC protocols and APIs listed above, there is an RPC API specially designed for grid computing: GridRPC. Promoted by Open Grid Forum, GridRPC [45] is a popular API for running remote tasks in a grid. The underlying communication protocol is out of scope of GridRPC specification, so different implementations use different protocols.

GridRPC is a kind of RPC interface specially designed for scientific computations using client-server model. Its API specifies functions for a client program to start remote tasks on servers in a grid and wait for completion of these tasks.

The basic structure of GridRPC architecture is shown on Figure 1.1. It consists of 3 parts.

Service
: Provides tasks for remote execution. Servers in a grid can provide multiple services, and a remote task can be available on multiple servers.

Registry
: A central authority keeping information about remote tasks available and servers which provide these tasks as services. There is only one registry in a grid, but it can be distributed between several nodes.

Client
: A program which implements parallel algorithm and requests services from the grid to run remote tasks for this algorithm.

When a server in a grid starts up, it registers all its services with the grid's registry.

A client performs the following steps to run remote task.

- Initialising GridRPC client library with `grpc_initialize()` function. This function must be called before calling any other GridRPC API functions.

- Initialising function handle with `grpc_function_handle_init()` or `grpc_function_handle_default()` function. These functions have pointer to object of opaque `grpc_function_handle_t` type as their first argument and remote task name as the last argument. These functions perform lookup step by sending remote task name to the grid's registry and using reply to initialise function handle with information about the task, its arguments and the server assigned to the task. The `grpc_function_handle_init()` function has an additional argument specifying the name of the server to assign to the task. The `grpc_function_handle_default()` function requests grid's registry to assign a server to the task automatically.

- The remote task is started by calling `grpc_call()` or `grpc_call_async()` function. The `grpc_call()` function starts a remote task synchronously, waiting for remote task's completion. The `grpc_call_async()` function starts a remote task asynchronously, not waiting for its completion. The first argument of both functions is pointer to initialised function handle. The `grpc_call_async()` function has also argument pointing to an object of opaque `grpc_sessionid_t` type which stores session ID which can be used to identify a running remote task. All remaining arguments of both functions specify remote task's arguments.

- The client can wait for completion of remote tasks which were started asynchronously by using `grpc_wait()` function. Session ID of the remote task is passed as argument to this function.

Listing 1.1 at page 8 shows example program running a remote task asynchronously using GridRPC. [1]

GridRPC has three distinctive features among other RPC APIs.

- It's allowed to not specify server name to run a remote task on. This is achieved by using `grpc_function_handle_default()` function to initialise a function handle. In this case, server is assigned automatically.

- IDL is not used to generate client-side stub functions. Instead, task name is specified when initialising function handle, and pointer to this function handle is used as an argument for `grpc_call()` or `grpc_call_async()`

[1] This is a complete GridRPC program with proper error handling. We will omit error handling in further examples for brevity.

```c
#include <stdio.h>
#include <stdlib.h>
#include <grpc.h>

int main() {
  grpc_function_handle_t handle;
  grpc_sessionid_t sess_id;
  grpc_error_t status;
  int *ivec, *retval;
  int i, n = 10;

  ivec = (int *)malloc(n * sizeof(int));
  retval = (int *)malloc(n * sizeof(int));

  for (i = 0; i < n; i++) ivec[i] = 10;

  if (grpc_initialize(NULL) != GRPC_NO_ERROR) {
    grpc_perror("grpc_initialize");
    exit(1);
  }

  status = grpc_function_handle_default(&handle,
                                        "return_int_vector");
  if (status != GRPC_NO_ERROR) {
    fprintf(stderr, "Error creating function handle\n");
    exit(1);
  }

  status = grpc_call_async(&handle, &sess_id, ivec, n, retval);
  if (status != GRPC_NO_ERROR) {
    grpc_perror("grpc_call_async");
    exit(1);
  }

  status = grpc_wait(sess_id);
  if (status != GRPC_NO_ERROR) {
    grpc_perror("grpc_wait");
    exit(1);
  }

  grpc_finalize();
  return 0;
}
```

Listing 1.1: GridRPC program example.

or any other function that starts a remote task. All information about argument types and their directions (in, out or inout) is retrieved from the server which is assigned to run the task. However, IDL may be used for server-side programming for generating services which provide remote tasks.

- Although no stub functions are used on client side, the API is still high-level. Arguments for remote tasks are passed naturally as arguments for functions starting remote tasks. However, for better flexibility a way to build argument list is also provided, as well as `grpc_call_arg_stack()` and `grpc_call_arg_stack_async()` functions which use prepared argument list to pass to the remote task.

GridRPC has several advantages:

Standard API.
The same program can be recompiled with different GridRPC implementations without changing source code.

High level.
GridRPC allows to run remote tasks on a grid by specifying task names. All input arguments can be specified natural way as function arguments to GridRPC calls.

Flexible.
There are different variants of GridRPC call functions allowing either to pass arguments as regular function arguments or to create argument list in advance. GridRPC calls can be synchronous (making a program to wait for task completion), or asynchronous (allowing a program to start multiple tasks in parallel and then wait for completion of selected tasks). Server to run a task on can be either specified explicitly or omitted to allow grid middleware to select a server for the task.

Easy to program.
GridRPC API is very simple. There are very few functions to remember, and they are very easy to use. Unlike other RPC implementations, GridRPC needs no client-side IDL (interface definition language) to define argument types: argument types are provided by servers at runtime.

Easy to convert existing applications.
Just taking a linear program and replacing some function calls with GridRPC calls makes the program ready for grid computing.

1.4 GridRPC implementations

Due to its advantages, GridRPC was implemented shortly after it was standardised by several grid middleware projects. As a result, now there are several popular GridRPC implementations.

GridSolve [49]

NetSolve [15] project was started in 1994 to provide a simple and convenient way to organise distributed computational resources provided by loosely connected servers, which can be dispersed between different geographical locations. Its architecture consists of two kinds of servers: computational and communication ones. Computational server is a server providing computational resources: tasks which can be activated remotely. Communication servers constitute a distributed NetSolve agent, which have function similar to GridRPC registry: providing information about servers and tasks available on these servers. Also, NetSolve provides client libraries for C and FORTRAN with its own simple API, as well as interactive command line interface and interface for MATLAB.

After GridRPC programming model and API was standardised, a new version of NetSolve called GridSolve was released in 2003, providing full implementation of GridRPC. For historical reasons, GridRPC registry is called agent in GridSolve.

Ninf-G [46]

Ninf [44] project was also started in 1994 with the same goals as NetSolve. Its architecture also consists of servers providing tasks to be run remotely and metaserver which provides information about servers and tasks available on them. There can be multiple Metaservers providing distributed registry. Hence, basic architecture of Ninf is similar to one provided by NetSolve and GridRPC architecture which was standardised later. However, it has different client API and different approach to managing servers and installing tasks on them. Ninf API is available for C, FORTRAN, Java and Lisp programming languages.

Ninf-G is a new version of Ninf implemented on top of the Globus toolkit [22], providing GridRPC API and programming model.

DIET [13]

DIET (Distributed Interactive Engineering Toolkit) project was started in 2002. It has hierarchical architecture. This architecture has the following components (bottom to top).

CRD
: On the bottom of hierarchy there is Computational Resources Daemon (CRD), which represents computational resource.

SeD
: There is Server Daemon (SeD) which manages several CRDs. SeD is responsible for a server and has information about tasks available on the server and its status.

LA
: Local Agent (LA) aggregates information from multiple SeDs, but doesn't make scheduling decisions.

MA
: Master Agent (MA) is on top of hierarchy. It further aggregates information collected by multiple LAs. Also, it serves requests for computations from clients and assigns servers for these tasks. There can be multiple MAs for redundancy, providing distributed registry.

Originally DIET was using CORBA for invoking remote tasks [14], but later it was extended to implement GridRPC interface.

1.5 GridRPC limitations

GridRPC provides a powerful but simple and convenient API. However, GridRPC has some limitations due to its simplicity.

- A server is assigned to run each task independently because grid has no information about which tasks a program is going to run in advance. This leads to non-optimal assignment of servers to tasks (task-to-server mapping). It's possible to specify servers for each task explicitly in a program, but this approach requires prior knowledge of grid structure and list of its servers and makes the program fail if grid structure has changed or the program is run on a different grid.

- As most other RPC implementations, GridRPC has strict client-server architecture. This means that all input arguments are passed only from client to server, and all output arguments (results) are passed only from server to client. This leads to non-optimal communication: if the second task uses results of the first task, these results are first sent from the server running the first task to the client and then send from the client to the server running the second task, even if these servers are connected with high-speed network (or both tasks run on the same server).

1.6 Beyond GridRPC: collective mapping

A solution to individual task-to-server mapping provided by GridRPC (and resulting non-optimal server assignment and communication) is collective mapping: assigning servers to all tasks collectively. This allows not only to allocate computational resources more optimally, but also utilise direct server-to-server communication and take communication into account to optimise computation and communication time together.

However, collective mapping needs information about tasks to be run by a program, their sequence and data dependencies between them to be available in advance, prior to running parallel algorithm implemented by the program. There are different approaches to collect this information: runtime discovery and ADL. However, these approaches have their limitations: runtime discovery puts strict restrictions on application code, ADL requires a separate algorithm definition.

1.7 Contributions

This thesis presents a new approach to collecting information needed for collective mapping by using static code analysis to extract algorithm workflow from the source code of the program implementing this algorithm.

Algorithm workflow is extracted by a program and is stored in a tree structure. Then this structure is used to preprocess source files and insert additional data definitions containing extracted information and code which uses these data to build proper application performance model, assign servers to tasks collectively and arrange server-to-server communication.

This approach has advantages over runtime discovery and ADL approaches: less restrictions on source code, no need to supply a separate algorithm definition.

1.8 Structure

The following chapters are structured as following.

Limitations of individual mapping and advantages of collective mapping are outlined in chapter 2.

Existing approaches to optimise grid performance without using collective mapping, as well as approaches to collective mapping are discussed in chapter 3.

A new approach to building application performance model by applying static code analysis to a program is introduced in chapter 4. Changes needed to application build process to use this approach, how source code is analysed and what information is extracted, which data and code is inserted during preprocessing

and how this code works is described in details in this chapter. This is the main contribution of this thesis.

Applying static code analysis approach to real-world application is described in chapter 5.

Conclusions are presented in chapter 6.

Chapter 2

Motivation

Regular GridRPC implementations use individual task-to-server mapping. Each server assignment to remote task is performed individually because grid has no prior information about which tasks a program is going to run later.

2.1 GridRPC program example

Listing 2.1 at page 16 gives an example of parallel algorithm inside GridRPC program. [1] The algorithm does the following.

1. Function handles are initialised at lines 15 to 21. The grpc_function_handle_default () function handle initialisation variant is used, which allows to specify just task name, omitting server name to allow grid middleware to assign a suitable server to each task.
2. The parallel algorithm is inside the block beginning at line 23.
3. First, two instances of `T1_cond` task are started at lines 24 and 26.
4. Then the program waits for completion of both tasks at line 28.
5. After that, two tasks are run synchronously one after another: `T2_cond` at line 29 and `P1_cond` at line 30.
6. The `p` output argument of `P1_cond` task is used in condition at line 31.

[1] This is not a complete program, just a function implementing parallel algorithm. We assume that arguments of type **double** $*$ point to already allocated memory sufficient to hold vectors of double-precision floating point numbers used in this algorithm.

```
#include <grpc.h>

void run(unsigned int size, unsigned int count,
         double *a0, double *b0, double *c0,
         double *a1, double *b1, double *c1,
         double *d)
{
    grpc_function_handle_t t1_0, t1_1;
    grpc_function_handle_t t2_0, t2_1;
    grpc_function_handle_t t3_0, t3_1;
    grpc_function_handle_t p1;
    grpc_sessionid_t sid0, sid1;
    int p;
    /* Function handles initialisation. */
    grpc_function_handle_default(&t1_0, "T1_cond");
    grpc_function_handle_default(&t1_1, "T1_cond");
    grpc_function_handle_default(&t2_0, "T2_cond");
    grpc_function_handle_default(&t2_1, "T2_cond");
    grpc_function_handle_default(&t3_0, "T3_cond");
    grpc_function_handle_default(&t3_1, "T3_cond");
    grpc_function_handle_default(&p1, "P1_cond");
    /* Parallel algorithm. */
    {
        grpc_call_async(&t1_0, &sid0, a0, b0, c0,
                        size, count);
        grpc_call_async(&t1_1, &sid1, a1, b1, c1,
                        size, count * 2);
        grpc_wait_all();
        grpc_call(&t2_0, c0, c1, d, size, count);
        grpc_call(&p1, d, size, count, (double)0.5, &p);
        if (p) {
            grpc_call_async(&t3_0, &sid0, c0, a0,
                            size, count);
            grpc_call_async(&t3_1, &sid1, c1, a1,
                            size, count * 2);
            grpc_wait_all();
            grpc_call(&t2_1, c0, c1, d, size, count);
        } else {
            grpc_call_async(&t3_0, &sid0, c0, b0,
                            size, count);
            grpc_call_async(&t3_1, &sid1, c1, b1,
                            size, count * 2);
            grpc_wait_all();
            grpc_call(&t2_1, c0, c1, d, size, count);
        }
    }
}
```

Listing 2.1: GridRPC program. Error handling omitted for brevity.

7. If `p` is non-zero, two instances of `T3_cond` task are started at lines 32 and 34, then the program waits for both running tasks at line 36, then `T2_cond` task is run synchronously using results from previous tasks at line 37.

8. Otherwise, if `p` is zero, two instances of `T3_cond` task are started at lines 39 and 41 with different arguments, then the program waits for both running tasks at line 43, then `T2_cond` task is run synchronously using results from previous tasks at line 44.

2.2 First problem: non-optimal task-to-server mapping

The first problem with individual mapping is non-optimal allocation of computational resources to tasks.

Let's assume that computational complexity of `T1_cond` task has linear dependency on numerical value of its last argument. This means that the second instance of this task started at line 26 will take twice longer than the first instance of this task started at line 24 to complete on the same hardware.

Let's assume that we have just two servers in the grid: `S1` and `S2`, and `S1` is twice faster than `S2`.

When our program starts the first instance of `T1_cond` task, grid middleware has no way to find out which tasks will be started after that, so it's logical to assume that the fastest server `S1` will be assigned to this task. Then the second instance of `T1_cond` is started, which requires twice more computational resources than the first one. However, grid middleware has no other choice as to assign server `S2` to this task. As a result, we have faster server `S1` handling less complex task and slower server `S2` handling more complex task, so the first task will complete 4 times faster than the second one. If we somehow knew that more complex task is coming after the first one, it would be possible to assign `S2` to the first simple task, and then faster `S1` to the second more complex task, allowing both tasks to finish approximately at the same time, making this part of the algorithm to run twice faster than with individual mapping.

2.3 Second problem: non-optimal communication

The second problem with individual mapping is non-optimal communication.

In the same example, let's assume that `d` argument of `T2_cond` task at line 29 is output argument, and it's also an input argument for `P1_cond` task at line 30. When `T2_cond` task is run, there is no information that `d` argument will be used

after that by P1_cond task, so this argument will be sent from the server running T2_cond task to the client and then the client will send it to the server running P1_cond later. If this information was available, it would be possible to arrange sending d argument from server running T2_cond directly to server running P1_cond, bypassing the client (or avoid sending completely if both tasks run on the same server).

This would allow significantly improving communication time because servers in a grid are usually connected by high-speed local area network, whereas client is usually connected to the grid through public Internet using much slower ISP subscriber link.

Also, sending large amounts of data directly from server to server bypassing client can have impact on client performance by preventing paging in case these data are large enough to not fit in client's RAM. Client is usually a regular desktop or laptop PC, whereas server has much more powerful hardware and greater amount of RAM.

2.4 Third problem: lack of communication parallelism

The third problem with individual mapping also results from its strict client-server model.

Before a task is started on a remote server, input arguments are sent from the client to the server sequentially, and output arguments are returned to the client sequentially as well after task's completion. Even if it was possible to send and receive arguments in parallel, it would bring no improvement, or even make things worse: client-to-server link is usually just a single slow ISP subscriber line which constitutes a bottleneck in data flow and can be saturated easily with big amounts of data. There is no point to send arguments in parallel if they traverse the same narrow data link anyway.

Also, starting remote tasks individually without prior knowledge about further tasks to be started precludes overlapping inbound and outbound data transmission: receiving output arguments from the previous task simultaneously with sending input arguments to the next task.

Another important source of communication parallelism is server-to-server communication, which can be performed simultaneously between different servers and between some server and client as well. Unfortunately, individual mapping precludes direct server-to-server communication at all, so this opportunity is missed completely.

2.5 Solution: collective mapping

Collective mapping solves these problems by using application performance model built prior to running parallel algorithm. The information containing in this model is sufficient for assigning servers to tasks more optimal way to optimise computation and communication time, as well as arranging direct server-to-server communication.

Chapter 3

Related research

In chapter 2 we explain limitations of individual mapping and necessity of collective mapping to overcome these limitations. This chapter presents more detailed discussion of collective mapping and existing approaches to implement it.

3.1 Approaches to grid optimisation before collective mapping

Even before GridRPC was standardised, there were attempts to optimise grid middleware to overcome limitations of individual mapping.

3.1.1 Task farming

Task farming [16] support was added to NetSolve in 1999. There is a large class of algorithms which can be split into multiple independent parallel tasks which can be started simultaneously and no intermediate results are needed before all tasks have been completed. The idea behind task farming is to start all these tasks with a single RPC request. This allows to assign servers to all these tasks collectively, achieving optimal task-to-server mapping.

Task farming was implemented in NetSolve in client library as `farm()` function, which puts all tasks to be run in a queue and preforms scheduling of pending tasks.

However, this is approach has the following limitations.

- Only simple parallel algorithms can take advantage of this approach. An algorithm should have all tasks that can be run in parallel, with no data dependencies, no need for intermediate results, and no conditional task execution.

- The `farm()` function is a single atomic blocking call, so the client can't perform additional computations or start other tasks until farmed tasks has been completed.

- There are no data dependencies between farmed tasks, so there is no improvement in communication which is still strictly client-server without any parallelism.

3.1.2 Task sequencing

Task sequencing [3] support was added to NetSolve in 2000. It allows to run sequence of tasks where each subsequent task requires results from the previous task. As with task farming, it's implemented as a single RPC call which runs all tasks in the sequence on a single server, preserving task results for use by subsequent tasks. This way more optimal communication is achieved (no intermediate results sent at all, just stored on the server).

Distributed task sequencing was proposed as an extension to GridRPC and implemented in Ninf-G [47] in 2006. Unlike task sequencing in NetSolve, distributed task sequencing allows to run tasks on different servers.

Both task sequencing approaches have the following limitations.

- Only sequential algorithms can utilise this feature. No conditional task execution allowed, no access to intermediate results.

- Task sequencing is preformed by a single atomic blocking call, so the client can't perform additional computations or start other tasks until tasks sequence has been completed.

- Only communication is optimised this way, not computation.

3.1.3 Distributed storage infrastructure

Distributed storage infrastructure [4] (DSI) was introduced in NetSolve in 2002. It allows clients to store data objects on remote storage (called storage depot) using IBP [41] (Internet Backplane Protocol). These data objects can be used instead of client-local data as arguments of remote tasks.

Although it's less optimal than direct server-to-server communication, storage depot can be located in the same local area network as servers in the grid, so repeated retrieval of large data arrays from storage depot to grid servers can still be much faster than repeated upload of the same data arrays from the client.

Although DSI brings some improvement to communication speed, this approach has the following limitations.

- Communication is still far from optimal. For example, if task `B` uses results from task `A`, instead of sending these results directly from the server running task `A` to the server running task `B` (or completely eliminating communication if both tasks run on the same server), results are first uploaded to storage depot and then downloaded from it.
- Task-to-server mapping is still performed individually for each task. This means that opportunity for better allocation of computational resources is missed.
- Storing data with DSI should be arranged manually. This means that instead of passing local arguments to RPC calls, program should be modified to first create objects in storage depot and then use references to these objects as RPC arguments. This leads to more complex code and increases possibility of software bugs as a result.

3.1.4 Arranging server-to-server communication manually

Direct server-to-server communication [20] was introduced into NetSolve in 2004. This feature (called Data Persistence and Redistribution) allows to arrange direct server-to-server communication by explicitly specifying which input and output arguments of running tasks should be used as input arguments of a new task to be started.

The following steps should be added to the program to use this feature.

1. An array of `ObjectLocation` structures is filled in with the following information for each argument to be sent directly from servers running other tasks to the server which is about to run the new task:
   - request ID (the same as session ID in GridRPC API) of the running task;
   - object (non-scalar argument) number;
   - argument direction (input or output).

   This array is called redistribution.
2. The filled in redistribution is passed (along with task name and other arguments) to the special `netslnbdist()` function which starts specified task using redistribution provided. As a result, arguments specified in the redistribution are sent to this task from other servers, not from the client.

Data Three Manager [19] was introduced into DIET in 2005. This is another approach for manually arranging direct server-to-server communication, adapted to DIET's hierarchical architecture.

A non-intrusive and incremental approach to enabling direct communications [32], [31] was introduced into NetSolve in 2006. Later it was called NI-Connect [50]. This approach combines convenience of DSI (using handles as arguments) with efficiency benefits of data persistence and redistribution. An interesting aspect of this approach is that it doesn't require modifications to NetSolve. Instead of that, just a wrapper client library and a special service managing server-to-server communication should be installed in NetSolve. However, this approach still uses individual mapping to run tasks.

Approaches for arranging server-to-server communication manually listed above have the following limitations.

- Task-to-server mapping is still performed individually for each task. This means that opportunity for better allocation of Computational resources is missed.
- Specifying data redistribution required manual program modification, which requires additional labour and can lead to difficult to detect bugs due to human error, especially when algorithm has been changed, but redistribution was not kept in sync with these changes.
- Because of individual task-to-server mapping, server to run a task is unknown before the client preforms RPC call to run this task. This means that if task `B` uses results of task `A`, server running task `A` can't start sending results to server which is going to run task `B` in advance, before client requests start of task `B`. This communication can be started only after the client performs RPC call to run task `B`, making this task to wait for task `A` results' transfer completion.

3.2 Advantages of collective mapping

As the name implies, collective mapping is the way to map task to servers collectively, taking the whole parallel algorithm into account.

Collective mapping has the following advantages over individual mapping.

Better computation time can be achieving by finding more optimal task-to-server mapping.

Better communication time can be achieved by taking communication time into account when finding optimal task-to-server mapping.

Direct server-to-server communication can be arranged to send data from one server to another in advance if there are tasks using results from other tasks.

Avoiding unnecessary communication can be achieved in case when data already available on the server from the previous task. Also, returning output arguments to the client can be omitted if these arguments are used by other tasks, but not the client program.

Communication parallelism is a direct result of server-to-server communication, which can be performed in parallel between different servers as well as with client-server communication.

3.3 Information needed for collective mapping

As it was mentioned in chapter 2, in order for collective mapping to work, a prior knowledge about remote tasks to be run during parallel algorithm and their arguments is needed. This prior knowledge should contain the following information.

Computational complexity of remote tasks
Computational complexity of remote tasks run by the program in some units of computation (like flops). This is necessary to estimate task completion time on each server evaluated for running this task, provided that we know speed of each server.

Size of input and output arguments of remote tasks
This information allows to determine amount of data sent from client to servers running tasks, from these servers back to client, and between servers, so communication time can be taken into account, provided that we know throughput of communication channels between client and servers as well as between servers.

Order of remote tasks execution
The order in which the program starts remote tasks and waits for their completion. This information is necessary to find out which tasks are running in parallel and in which order.

Data dependencies between tasks
Arguments produced by one task as output arguments and then consumed by another task as input argument is data dependency between these tasks. Data dependencies constitute a direct acyclic graph (DAG) with tasks as nodes and set of arguments with amount of data transferred as vertices (arrows). This information is necessary to arrange server-to-server communication.

Once all necessary information is somehow collected, it can be used to find optimal task-to-server mapping which takes into account computation and communication times and allows to arrange direct server-to-server communication.

Finding this optimal mapping is NP-complete task, but there are several heuristics available [12], which can be used to produce sub-optimal mapping, which is still good enough to be practical.

Approaches to collective mapping have different implementations, but the most important difference is how they collect information listed above.

3.4 Runtime discovery

An approach called runtime discovery was originally implemented in NetSolve resulting in SmartNetSolve [7]. Later it was adapted to a new generation of NetSolve called GridSolve, which implemented GridRPC API, resulting in SmartGridSolve [6], [8]. Extension to GridRPC provided by SmartGridSolve is called SmartGridRPC [9], [10].

High-level design of SmartGridRPC model and the way it uses application performance model to map tasks to heterogeneous servers was inspired by mpC [29], [2] and HeteroMPI [30].

3.4.1 Basic Functionality

Listing 3.1 at page 27 gives an example of parallel algorithm using SmartGridRPC API.

The main difference with GridRPC algorithm is blocks marked with `grpc_map()` directive at lines 23, 33 and 42. These blocks mark parts of algorithm for collective mapping.

The reason why the whole algorithm is not marked for collective mapping and split into three parts instead is that runtime discovery doesn't work correctly with branching and other control flow which depends on results of computation. We'll discuss this in subsection 3.4.3 below.

The `grpc_map()` directive is not a C language extension, it's a preprocessor macro which is expanded into a `while` loop. The loop condition is an internal function (not a part of public API) which implements finite state machine which stores its state in internal global variables and controls loop execution making the loop body to be executed twice and performing different actiond dependent on which loop pass in precedes. The behaviour of GridRPC calls is different dependent on which loop pass is in progress.

The loop FSM controlled by loop condition function works the following way.

```
1  #include <grpc.h>
2
3  void run(unsigned int size, unsigned int count,
4           double *a0, double *b0, double *c0,
5           double *a1, double *b1, double *c1,
6           double *d)
7  {
8      grpc_function_handle_t t1_0, t1_1;
9      grpc_function_handle_t t2_0, t2_1;
10     grpc_function_handle_t t3_0, t3_1;
11     grpc_function_handle_t p1;
12     grpc_sessionid_t sid0, sid1;
13     int p;
14     /* Function handles initialisation. */
15     grpc_function_handle_default(&t1_0, "T1_cond");
16     grpc_function_handle_default(&t1_1, "T1_cond");
17     grpc_function_handle_default(&t2_0, "T2_cond");
18     grpc_function_handle_default(&t2_1, "T2_cond");
19     grpc_function_handle_default(&t3_0, "T3_cond");
20     grpc_function_handle_default(&t3_1, "T3_cond");
21     grpc_function_handle_default(&p1, "P1_cond");
22     /* Parallel algorithm. */
23     grpc_map("ex_map") {
24         grpc_call_async(&t1_0, &sid0, a0, b0, c0,
25                         size, count);
26         grpc_call_async(&t1_1, &sid1, a1, b1, c1,
27                         size, count * 2);
28         grpc_wait_all();
29         grpc_call(&t2_0, c0, c1, d, size, count);
30         grpc_call(&p1, d, size, count, (double)0.5, &p);
31     }
32     if (p) {
33         grpc_map("ex_map") {
34             grpc_call_async(&t3_0, &sid0, c0, a0,
35                             size, count);
36             grpc_call_async(&t3_1, &sid1, c1, a1,
37                             size, count * 2);
38             grpc_wait_all();
39             grpc_call(&t2_1, c0, c1, d, size, count);
40         }
41     } else {
42         grpc_map("ex_map") {
43             grpc_call_async(&t3_0, &sid0, c0, b0,
44                             size, count);
45             grpc_call_async(&t3_1, &sid1, c1, b1,
46                             size, count * 2);
47             grpc_wait_all();
48             grpc_call(&t2_1, c0, c1, d, size, count);
49         }
50     }
51 }
```

Listing 3.1: SmartGridRPC program. Error handling omitted for brevity.

Preparation
The first invocation of loop condition function just stores information about loop pass internally and returns non-zero value to allow the first loop pass.

Discovery phase
The first loop pass is called discovery phase. GridRPC calls are being recorded in this phase, but they don't start actual tasks on remote servers. Information about function handles (which include task names) and call arguments is stored for further use in collective mapping.

Mapping
The second invocation of loop condition function uses information collected during discovery phase to perform collective mapping. It stores results of collective mapping internally and returns non-zero value again to allow the second loop pass.

Execution phase
The second loop pass is called execution phase. GridRPC calls start remote tasks during this phase, using servers assigned to tasks as a result of collective mapping.

Exit from loop
The third invocation of loop condition function returns zero value, making control flow exit the loop.

3.4.2 Fault Tolerance

There is fault tolerance implemented in SmartGridSolve and it has two modes.

Simple mode requires special extended versions of GridRPC calls to be used. In this mode, failed GridRPC call is retried again on another server, and individual mapping is used to assign a new server to failed task. Also, server-to-server communication is not used because choosing a new server for a failed task disrupts pre-arranged server-to-server communication plan. This causes reduced performance improvement compared to individual mapping because more optimal communication and communication parallelism opportunity is missed in this case.

Advanced mode utilises another loop pass in case of remote task failure. If a remote task has failed during execution phase, the FSM switched to error mode. GridRPC calls don't start any new remote tasks in this mode, they are just being ignored. When loop is done, loop condition function is called

again. It performs another collective mapping, with a server where task failed excluded from a list of servers considered as candidates to run tasks. Then the loop condition function returns non-zero value to make the loop run once again, starting the algorithm from its beginning, with revised task-to-server mapping.

3.4.3 Limitations

Runtime discovery is a simple approach requiring minimal modifications to source code of GridRPC application. However, it has a significant limitation: source code inside `grpc_map()` block should be written very carefully to either make this code idempotent (having side effects to have exactly the same effect if code is run multiple times as if code is run only once) or not allow side effects at all.

This restriction on code is a result of the fact that the `grpc_map()` block is run at least twice: in discovery phase and in execution phase. There is a workaround in SmartGridRPC API for code with side effects: a `grpc_local()` directive. The statement (usually a block) following this directive is executed only during execution phase and is completely skipped during discovery phase.

However, it's not guaranteed that the code inside `grpc_local()` block will be executed only once. It can happen that a remote task failure happens after some `grpc_local()` blocks has been already run. In this case FSM enters error mode if advanced fault tolerance mode is used, all remaining GridRPC calls and `grpc_local()` blocks are ignored, but the loop is restarted after that in execution mode again. The only `grpc_local()` blocks which are guaranteed to not be run multiple times are the ones which follow the last GridRPC call in `grpc_map()` block.

Another restriction on code for runtime discovery is that the data flow in the code should be exactly the same during discovery an execution phases. This means that this data flow must be pre-determined and not depend on remote task results or any other data which can be different during discovery and execution phases.

Listing 3.2 at page 30 gives an example of parallel algorithm using Smart-GridRPC API with data flow which is not pre-determined. There is `if` statement at line 31. Which branch of `if` statement is run depends on value of `p` variable which is assigned as a result of remote task run at line 30. However, remote tasks are not run during discovery phase, `p` variable is not assigned value during the first loop pass, so the `if` statement's is selected for execution based on undefined value. The branch executed during execution phase based on actual result of remote task may be different, and runtime discovery results used for task-to-server mapping and (more importantly) server-to-server communication may be wrong.

Failure to make the code inside `grpc_map()` block idempotent (except code

```
1  #include <grpc.h>
2
3  void run(unsigned int size, unsigned int count,
4           double *a0, double *b0, double *c0,
5           double *a1, double *b1, double *c1,
6           double *d)
7  {
8      grpc_function_handle_t t1_0, t1_1;
9      grpc_function_handle_t t2_0, t2_1;
10     grpc_function_handle_t t3_0, t3_1;
11     grpc_function_handle_t p1;
12     grpc_sessionid_t sid0, sid1;
13     int p;
14     /* Function handles initialisation. */
15     grpc_function_handle_default(&t1_0, "T1_cond");
16     grpc_function_handle_default(&t1_1, "T1_cond");
17     grpc_function_handle_default(&t2_0, "T2_cond");
18     grpc_function_handle_default(&t2_1, "T2_cond");
19     grpc_function_handle_default(&t3_0, "T3_cond");
20     grpc_function_handle_default(&t3_1, "T3_cond");
21     grpc_function_handle_default(&p1, "P1_cond");
22     /* Parallel algorithm. */
23     grpc_map("ex_map") {
24         grpc_call_async(&t1_0, &sid0, a0, b0, c0,
25                         size, count);
26         grpc_call_async(&t1_1, &sid1, a1, b1, c1,
27                         size, count * 2);
28         grpc_wait_all();
29         grpc_call(&t2_0, c0, c1, d, size, count);
30         grpc_call(&p1, d, size, count, (double)0.5, &p);
31         if (p) {
32             grpc_call_async(&t3_0, &sid0, c0, a0,
33                             size, count);
34             grpc_call_async(&t3_1, &sid1, c1, a1,
35                             size, count * 2);
36             grpc_wait_all();
37             grpc_call(&t2_1, c0, c1, d, size, count);
38         } else {
39             grpc_call_async(&t3_0, &sid0, c0, b0,
40                             size, count);
41             grpc_call_async(&t3_1, &sid1, c1, b1,
42                             size, count * 2);
43             grpc_wait_all();
44             grpc_call(&t2_1, c0, c1, d, size, count);
45         }
46     }
47 }
```

Listing 3.2: SmartGridRPC program with undetermined data flow. Error handling omitted for brevity.

in `grpc_local()` blocks following the last GridRPC call) and data flow predetermined can lead to wrong results which are hard to find and debug, or even errors which are hidden during normal algorithm execution and have effect only when fault tolerance is triggered.

3.5 ADL

Another approach to collective mapping is the use of ADL [27], the Algorithm Definition Language.

3.5.1 Basic Functionality

ADL implementation uses an extension to GridRPC API similar to 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.

Listing 3.3 at page 32 shows the same algorithm as in Listing 3.2 at page 30, but modified for use with ADL.

The `grpc_map()` directive has slightly different syntax with additional arguments besides heuristic name:

- `ADL` constant specifies that ADL is used;
- `cndalg` is a name of a variable storing internal ADL representation generated from ADL definition;
- format string similar to the one used by `printf` function from the standard C library; needed because of variable number of remaining arguments;
- actual values of runtime parameters passed as ADL definition's arguments.

The task dependency graph and application's performance model are specified separately, using ADL.

Listing 3.4 at page 33 gives an example of an ADL specification for the algorithm in Listing 3.3. The `module` definition specifies the name of ADL module (used as a name of generated variable name in `grpc_map()` directive) and definition's runtime parameters' types and names. 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 on the actual values of these parameters, which are specified at runtime. For example, the size

```
1  #include <grpc.h>
2
3  void run(unsigned int size, unsigned int count,
4           double *a0, double *b0, double *c0,
5           double *a1, double *b1, double *c1,
6           double *d)
7  {
8      grpc_function_handle_t t1_0, t1_1;
9      grpc_function_handle_t t2_0, t2_1;
10     grpc_function_handle_t t3_0, t3_1;
11     grpc_function_handle_t p1;
12     grpc_sessionid_t sid0, sid1;
13     int p;
14     /* Function handles initialisation. */
15     grpc_function_handle_default(&t1_0, "T1_cond");
16     grpc_function_handle_default(&t1_1, "T1_cond");
17     grpc_function_handle_default(&t2_0, "T2_cond");
18     grpc_function_handle_default(&t2_1, "T2_cond");
19     grpc_function_handle_default(&t3_0, "T3_cond");
20     grpc_function_handle_default(&t3_1, "T3_cond");
21     grpc_function_handle_default(&p1, "P1_cond");
22     /* Parallel algorithm. */
23     grpc_map("ex_map", ADL, cndalg, "%d,%d,%d,%d", size, count, 1, 1) {
24         grpc_call_async(&t1_0, &sid0, a0, b0, c0,
25                         size, count);
26         grpc_call_async(&t1_1, &sid1, a1, b1, c1,
27                         size, count * 2);
28         grpc_wait_all();
29         grpc_call(&t2_0, c0, c1, d, size, count);
30         grpc_call(&p1, d, size, count, (double)0.5, &p);
31         if (p) {
32             grpc_call_async(&t3_0, &sid0, c0, a0,
33                             size, count);
34             grpc_call_async(&t3_1, &sid1, c1, a1,
35                             size, count * 2);
36             grpc_wait_all();
37             grpc_call(&t2_1, c0, c1, d, size, count);
38         } else {
39             grpc_call_async(&t3_0, &sid0, c0, b0,
40                             size, count);
41             grpc_call_async(&t3_1, &sid1, c1, b1,
42                             size, count * 2);
43             grpc_wait_all();
44             grpc_call(&t2_1, c0, c1, d, size, count);
45         }
46     }
47 }
```

Listing 3.3: SmartGridRPC program for ADL. Error handling omitted for brevity.

```
1  module cndalg(int size, int count, int cndtrue, int cndfalse)
2  {
3  component:
4    task "tgtest_cond.idl" T1_cond, T2_cond, T3_cond, P1_cond;
5
6  OBJ:
7    DOUBLE(size) a0, a1, b0, b1, c0, c1, d;
8    INTEGER p;
9
10 algorithm:
11   parallel {
12     T1_cond: (a0, b0, @size, @count)->(c0);
13     T1_cond: (a1, b1, @size, @count)->(c1);
14   }
15   T2_cond: (c0, c1, @size, @count)->(d);
16   P1_cond: (d, @size, @count, 0.5)->(p);
17   parallel {
18     if (cndtrue)
19       parallel {
20         T3_cond: (c0, a0, @size, @count)->(c0);
21         T3_cond: (c1, a1, @size, @count)->(c1);
22         T2_cond: (c0, c1, @size, @count)->(d);
23       }
24     if (cndfalse)
25       parallel {
26         T3_cond: (c0, b0, @size, @count)->(c0);
27         T3_cond: (c1, b1, @size, @count)->(c1);
28         T2_cond: (c0, c1, @size, @count)->(d);
29       }
30   }
31 }
```

Listing 3.4: ADL specification.

of vectors depends on `size` parameter. 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 Listing 3.3. works correctly when using ADL specification in Listing 3.4.

3.5.2 Limitations

The main limitation 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.

3.6 Workflow submission

Another radical approach to collective mapping is using some kind of algorithm definition language similar to ADL as the only representation of parallel algorithm.

3.6.1 MA_{DAG}

A special MA (Master Agent) called MA_{DAG} [1] which allows clients to submit workflows for whole algorithms was introduced into DIET in 2006. The workflow contains task dependency DAG in form of XML document. This entire workflow is submitted to to MA_{DAG} using DIET-specific API (not GridRPC). Upon workflow submission, MA_{DAG} schedules the algorithm, performing collective task-to-server mapping. Then the whole algorithm is run on servers assigned by MA_{DAG} and the results returned to the client.

Listing 3.5 at page 35 gives an example of workflow in MA_{DAG} language. This example computes $(2(n+1)+2(n+1))^2$ where $n=56$. The meaning of XML elements is following.

- Tasks are specified by `node` elements.

```
1  <dag>
2    <node id="n1" path="succ">
3      <arg name="in" type="DIET_INT" value="56"/>
4      <out name="out1" type="DIET_INT"/>
5      <out name="out2" type="DIET_INT"/>
6    </node>
7    <node id="n2" path="double">
8      <in name="in" type="DIET_INT" source="n1#out1"/>
9      <out name="out" type="DIET_INT"/>
10   </node>
11   <node id="n3" path="double">
12     <in name="in" type="DIET_INT" source="n1#out2"/>
13     <out name="out" type="DIET_INT"/>
14   </node>
15   <node id="n4" path="sum">
16     <in name="in4" type="DIET_INT" source="n2#out"/>
17     <in name="in5" type="DIET_INT" source="n3#out"/>
18     <out name="out" type="DIET_INT"/>
19   </node>
20   <node id="n5" path="square">
21     <in name="in" type="DIET_INT" source="n4#out" />
22     <out name="out" type="DIET_DOUBLE" />
23   </node>
24 </dag>
```

Listing 3.5: MA_{DAG} workflow language example.

- Dependencies between tasks are specified by `source` attributes of `in` elements.

The parallel algorithm consists of the following steps.

1. $n + 1$ is calculated by the task in node `n1`.
2. Result from `n1` node are doubled in parallel twice by tasks in `n2` and `n3` nodes.
3. Results from `n2` and `n3` nodes are added together by task in `n4` node.
4. Result from `n4` node is squared by `n5` node.

Input parameter n (argument named `in` of `n1` node) is written in the algorithm definition itself. There is no way to define parameters for algorithm, all input data are listed in XML file itself.

3.6.2 Gwendia

In 2010 Gwendia [39] functional language support was introduced into DIET, and MA_{DAG} was extended to support it. This language is also XML-based, but unlike MA_{DAG} workflow language, it has support for conditions and loops, as well as input parameters, which are stored in a a separate XML file and can be passed to the workflow.

Listing 3.5 at page 35 gives an example of workflow in Gwendia language. This example also computes $(2(n + 1) + 2(n + 1))^2$, but n can be a vector of integer numbers, and it's not specified inside workflow definition itself. Instead it should be specified in a separate data file. The steps of the algorithm are the same as in MA_{DAG} example above, but the syntax is different.

- Input and output parameters of the whole algorithm are specified by `source` and `sink` elements inside `interface` element.
- Tasks are specified by `processor` elements inside `processors` element.
- Dependencies between input and output parameters of tasks and algorithm are specified by `link` elements inside `links` element.

```
<?xml version="1.0" encoding="UTF-8"?>
<workflow name="func_scalar">

  <interface>
    <source name="wf_input" type="DIET_INT"/>
    <sink name="wf_output" type="DIET_DOUBLE" />
  </interface>

  <processors>
    <processor name="n1">
      <out name="out1" type="DIET_INT"/>
      <out name="out2" type="DIET_INT"/>
      <diet path="succ"/>
    </processor>

    <processor name="n2">
      <in name="in" type="DIET_INT" />
      <out name="out" type="DIET_INT"/>
      <diet path="double"/>
    </processor>

    <processor name="n3">
      <in name="in" type="DIET_INT"/>
      <out name="out" type="DIET_INT"/>
      <diet path="double"/>
    </processor>

    <processor name="n4">
      <in name="in1" type="DIET_INT"/>
      <in name="in2" type="DIET_INT"/>
      <out name="out" type="DIET_INT"/>
      <diet path="sum"/>
      <iterationstrategy>
        <dot>
          <port name="in1" />
          <port name="in2" />
        </dot>
      </iterationstrategy>
    </processor>

    <processor name="n5">
      <in name="in" type="DIET_INT"/>
      <out name="out" type="DIET_DOUBLE"/>
      <diet path="square"/>
    </processor>

  </processors>

  <links>
    <link from="wf_input" to="n1:in"/>
    <link from="n1:out1" to="n2:in"/>
    <link from="n1:out2" to="n3:in"/>
    <link from="n2:out" to="n4:in1"/>
    <link from="n3:out" to="n4:in2"/>
    <link from="n4:out" to="n5:in"/>
    <link from="n5:out" to="wf_output"/>
  </links>

</workflow>
```

Listing 3.6: Gwendia workflow language example.

3.6.3 Limitations

The approaches of submitting the whole workflow for remote execution listed above have the following limitations.

- The algorithm is submitted as a single entity to the agent and is executed entirely remotely. Intermediate results are not available to the client, client can't perform its own computations while the algorithm is running remotely.
- DIET API is used, so algorithm can't be ported to other GridRPC implementations.
- The algorithm for collective mapping is written using a special XML-based language, not a normal general-purpose programming language. There are tools for writing XML more conveniently, but this doesn't solve the problem of converting existing algorithms written on regular programming languages for grid computing.

Chapter 4

New approach to collective mapping: static code analysis

4.1 Overview

We propose a new approach to collective mapping that utilises static code analysis to collect information about application performance model. The proposed approach is an attempt to combine advantages of both pure SmartGridSolve and ADL-enabled SmartGridSolve, while avoiding their disadvantages [23].

This is achieved by using static code analysis to extract as much information as possible from the application code itself 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 restrictions on code side effects imposed by pure SmartGridSolve and allows loops and branches in the algorithm. On the other hand, it does not require an additional specification of the algorithm in ADL and eliminates the problem of of keeping the algorithm specification in sync with the application code.

The proposed approach is implemented as a modified version of SmartGridSolve, providing SmartGridRPC API with minor extensions. It provides compatibility with GridRPC and SmartGridRPC and accepts any GridRPC source code with or without SmartGridRPC extensions: `grpc_map()` and `grpc_local()` blocks. An additional extension for SmartGridRPC is `grpc_map_static()` directive which marks block of code for static code analysis. A different directive for static code analysis allows to combine this approach with runtime discovery approach in different blocks of the same program.

Listing 4.1 at page 40 gives an example of parallel algorithm modified for static code analysis approach. The only difference with Listing 3.2 at page 30 is using `grpc_map_static()` directive instead of `grpc_map()`. There is no

```
1  #include <grpc.h>
2
3  void run(unsigned int size, unsigned int count,
4           double *a0, double *b0, double *c0,
5           double *a1, double *b1, double *c1,
6           double *d)
7  {
8      grpc_function_handle_t t1_0, t1_1;
9      grpc_function_handle_t t2_0, t2_1;
10     grpc_function_handle_t t3_0, t3_1;
11     grpc_function_handle_t p1;
12     grpc_sessionid_t sid0, sid1;
13     int p;
14     /* Function handles initialisation. */
15     grpc_function_handle_default(&t1_0, "T1_cond");
16     grpc_function_handle_default(&t1_1, "T1_cond");
17     grpc_function_handle_default(&t2_0, "T2_cond");
18     grpc_function_handle_default(&t2_1, "T2_cond");
19     grpc_function_handle_default(&t3_0, "T3_cond");
20     grpc_function_handle_default(&t3_1, "T3_cond");
21     grpc_function_handle_default(&p1, "P1_cond");
22     /* Parallel algorithm. */
23     grpc_map_static("ex_map") {
24         grpc_call_async(&t1_0, &sid0, a0, b0, c0,
25                         size, count);
26         grpc_call_async(&t1_1, &sid1, a1, b1, c1,
27                         size, count * 2);
28         grpc_wait_all();
29         grpc_call(&t2_0, c0, c1, d, size, count);
30         grpc_call(&p1, d, size, count, (double)0.5, &p);
31         if (p) {
32             grpc_call_async(&t3_0, &sid0, c0, a0,
33                             size, count);
34             grpc_call_async(&t3_1, &sid1, c1, a1,
35                             size, count * 2);
36             grpc_wait_all();
37             grpc_call(&t2_1, c0, c1, d, size, count);
38         } else {
39             grpc_call_async(&t3_0, &sid0, c0, b0,
40                             size, count);
41             grpc_call_async(&t3_1, &sid1, c1, b1,
42                             size, count * 2);
43             grpc_wait_all();
44             grpc_call(&t2_1, c0, c1, d, size, count);
45         }
46     }
47 }
```

Listing 4.1: SmartGridRPC program for static code analysis. Error handling omitted for brevity.

need to supplement this code with additional algorithm specification, like the one presented in Listing 3.4.

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 internally.

After information is extracted, the code is pre-processed and modified to add the following stages before the algorithm is run:

- building application performance model;
- building extended dependency graph;
- using mapping heuristics.

These stages use the information collected during the static code analysis for 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 in section 4.3 and section 4.5 respectively.

4.2 Runtime parameters

Runtime parameters are expressions algorithm workflow is dependent on. There is no need to specify runtime parameters for the algorithm explicitly, as in ADL approach described in section 3.5. The following expressions that affect application performance model and task dependency graph are detected automatically and become implicit runtime parameters.

Scalar arguments of GridRPC calls
: Values of these arguments may affect non-scalar argument sizes and computational complexity of remote tasks.

Expressions determining number of loop iterations
: These are expressions from `for` loop or `grpc_likely()` directive in the beginning of loop body. Knowing (or estimating) number of loop iterations is necessary to build proper algorithm workflow.

Values of these parameters may be unknown at compile time, so algorithm definition extracted from the source code during static code analysis is not sufficient to build application performance model. That's why building application performance model along with task dependency graph is deferred until runtime using code injected by preprocessor.

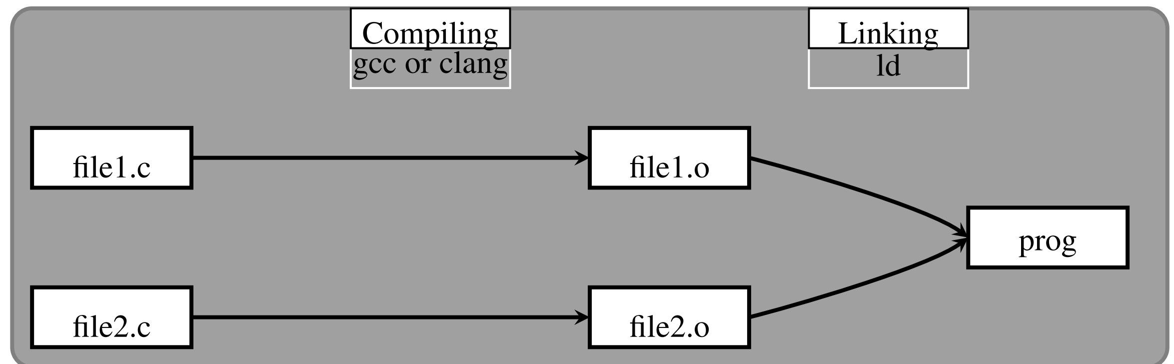


Figure 4.1: Regular application build process

For example, `size` and `count` arguments of `run()` function are automatically detected as runtime parameters for algorithm in Listing 4.1 at page 40. They affect size of non-scalar objects and computational complexity of remote tasks.

4.3 Analysing and preprocessing code

The static code analyser and preprocessor is implemented using Clang [34], a compiler for C family languages (C, C++, Objective C). Clang is written as a modular C++ library. The part of this library which parses source code and builds AST (abstract syntax tree) is used for static code analysis. Clang is a part of LLVM [33], a toolkit for building compilers.

4.3.1 Application build process

To explain how application build process has to be changed to adopt static code analysis, let's consider an application called `prog` which is built from two source modules written in C programming language: `file1.c` and `file2.c`.

A regular application build process is shown on Figure 4.1. This process is pretty simple and consists of two steps.

Compiling
All source modules are compiled into object code.

Linking
All resulting object modules are linked into the program.

Listing 4.2 at page 43 shows a Makefile for GNU make to build the `prog` program using this process.

```
CC = gcc
CFLAGS = -DGS_SMART_GRIDSOLVE=1
LIBS = -lgridsolve -lgssmartmapper -lm

SOURCES = file1.c file2.c
OBJS = $(SOURCES:.c=.o)

all: prog

%.o: %.c
        $(CC) $(CFLAGS) -c $*.c

prog: $(OBJS)
        $(CC) $(OBJS) -o prog
```

Listing 4.2: Makefile for building regular application. Uses GNU make extensions.

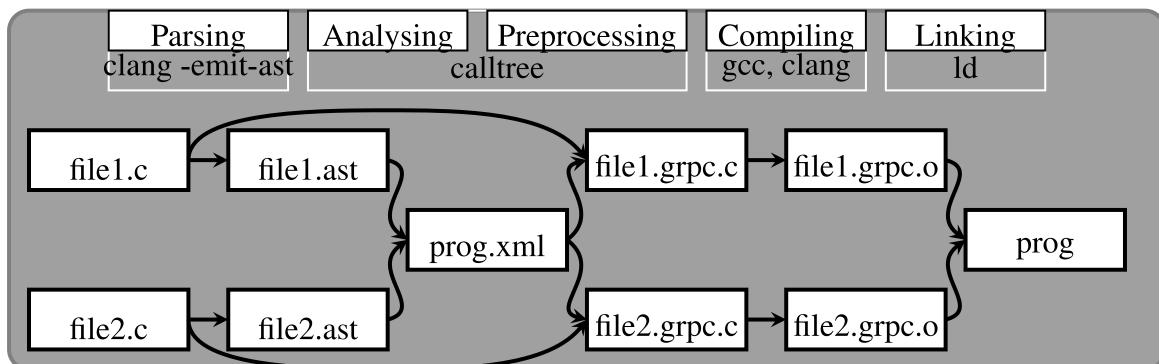


Figure 4.2: Application build process for static code analysis

```
CC = gcc
CFLAGS = -DGS_SMART_GRIDSOLVE=1
LIBS = -lgridsolve -lgssmartmapper -lm
PARSE = clang -emit-ast
COMBINE = calltree --print-tree

SOURCES = file1.c file2.c
ASTS = $(SOURCES:.c=.ast)
PROCESSED_SRCS = $(SOURCES:.c=.grpc.c)
OBJS = $(PROCESSED_SRCS:.c=.o)

all: prog

%.ast: %.c
	$(PARSE) $(CFLAGS) $*.c

$(PROCESSED_SRCS) prog.xml: $(ASTS)
	$(COMBINE) $(ASTS) > prog.xml

%.o: %.c
	$(CC) $(CFLAGS) -c $*.c

prog: $(OBJS)
	$(CC) $(OBJS) -o prog
```

Listing 4.3: Makefile for building application with static code analysis. Uses GNU make extensions.

Build process modified for use with static code analysis is shown on Figure 4.2. This process is more complex and contains additional steps.

Parsing
Source files are parsed by Clang using `-emit-ast` option. The resulting ASTs (abstract syntax trees) are saved in Clang-specific binary format.

Analysing
ASTs are analysed together by analyser program. The result is a tree-based internal representation, which can also be saved as XML.

Preprocessing
Files containing blocks marked for collective mapping using `grpc_map_static()` directive and GridRPC calls are being preprocessed. The information gathered during the previous step is used to generate source files modified to contain this information and code to preform collective mapping at runtime.

Compiling
The modified source modules are compiled into object code.

Linking
The resulting object modules are linked into the program.

Listing 4.3 at page 44 shows a Makefile for GNU make to build the `prog` program using this process. Note that each generated source file (with `.grpc.c` suffix) depends on *all* AST files (with `.ast` suffix) because whole program needs to be analysed before generating preprocessed source files.

The reason for separate parsing step is to allow using different compilation flags with each source module. Compilation flags can affect code semantics and how AST is built. For example, different source modules can have different C language dialects or require different C preprocessor macros defined in command line.

Although analysing and preprocessing are logically separate steps, they are performed by a single program called `calltree`.

4.3.2 Information gathered during static code analysis

AST files generated during parsing step are analysed together during analysing step. This is necessary because program data flow can span multiple source modules via function calls. The result of this analysis is a tree of objects that can be dumped as XML.

The analysis is done as following.

- For each AST file its internal AST representation is loaded and stored.
- All function declarations are found and stored, indexed by function name.
- The `main()` function is found and AST analysis is started recursively from its body.
- All function calls except calls related to GridRPC are analysed recursively. Function declaration is looked up by function name. If the declaration is found, mapping between function argument names and expressions representing arguments this function was called with is stored, and function body is analysed recursively. Otherwise, we assume that this is a function from an external library, so it doesn't contain GridRPC-related calls.
- Blocks of code marked for collective mapping with `grpc_map_static()` directive are found. Then these code blocks are analysed recursively to find all loops, `if` statements and GridRPC calls.
- GridRPC function handle initialisation calls are analysed both inside and outside `grpc_map_static()` blocks because function handle can be initialised outside this block and then used by GridRPC calls inside it. Analysing function handles initialisation functions is necessary because they contain task name, whereas GridRPC calls use handle instead of task name to specify the task to run remotely. Hence, it's important to correlate GridRPC calls with calls used to initialise function handles used in these calls. For each GridRPC function handle initialisation call, task name indexed by handle's expanded expression tree is stored. This mapping is used to retrieve task name by handle expression for GridRPC calls. More detailed description of expanded expressions is below in subsection 4.3.3.
- Non-scalar arguments of GridRPC calls are correlated and assigned unique numerical IDs, so the arguments pointing to the same arrays have the same IDs. This is done by storing argument IDs indexed by argument's expanded expression tree. For each argument its ID is looked up. If not found, a new ID is assigned and stored.
- All GridRPC calls are assigned numerical IDs called *call IDs*. These IDs are stored for each asynchronous GridRPC call indexed by session ID argument's expanded expression tree. This mapping is used for correlating GridRPC wait calls with their corresponding asynchronous GridRPC calls. GridRPC calls located in the same place of the program but called from different places have different call IDs.

- All GridRPC calls are assigned numerical IDs called *call site IDs*. Unlike call ID, call site ID identifies a place in the program where GridRPC is located, no matter where it's called from. Call site IDs are stored for each place where GridRPC call is located, indexed by file location. This mapping is used for inserting calls to `grpc_map_embed_call()` function which uses call site ID as its first argument. This is discussed in more details below in subsection 4.3.4.

A tree structure is formed as a result of this analysis. Listing 4.4 at page 48 shows this structure for the program in Listing 4.1 in XML format. [1]

The tree consists of the following elements.

function
: This is the root element.

grpc_map_static
: Block marked for collective mapping. Contains information about its location in source code, its heuristic name and its body tree.

grpc_handle_init
: GridRPC function handle initialisation. Contains information about its location in source code, its task name and expanded handle expression tree.

grpc_call
: GridRPC call. Contains information about its location in source code, task name, call ID, expanded expression trees for function handle and session ID (for asynchronous calls), expression trees for all arguments along with argument IDs for non-scalar arguments.

loop
: Loop of any kind: `while`, `for` or `do`. Contains information about expanded expression trees for loop condition expression (if any), loop initialisation and increment expressions (in case of `for` loop), and expanded expression representing number of loop iterations (if known at compile time). Also contains its body tree.

if
: Conditional statement (`if`). Contains trees for its then and else branches.

[1] This is not a complete tree. Attributes and elements containing code location, expression trees for function handles, session IDs and arguments and other information non-essential for this example are omitted for brevity.

```
<function name="main">
  <body>
    <grpc_handle_init taskname="T1_cond"/> <grpc_handle_init taskname="T1_cond"/>
    <grpc_handle_init taskname="T2_cond"/> <grpc_handle_init taskname="T2_cond"/>
    <grpc_handle_init taskname="T3_cond"/> <grpc_handle_init taskname="T3_cond"/>
    <grpc_handle_init taskname="P1_cond"/>
    <grpc_map_static function="run" heuristic="&quot;ex_map&quot;">
      <grpc_call call_id="0" call_site="0" taskname="T1_cond" blocking="false">
        <arg id="0"/> <arg id="1"/> <arg id="2"/>
        <arg value="size" type="unsigned_int"/>
        <arg value="count" type="unsigned_int"/>
      </grpc_call>
      <grpc_call call_id="1" call_site="1" taskname="T1_cond" blocking="false">
        <arg id="3"/> <arg id="4"/> <arg id="5"/>
        <arg value="size" type="unsigned_int"/>
        <arg value="count*2" type="unsigned_int"/>
      </grpc_call>
      <grpc_wait_all/>
      <grpc_call call_id="2" call_site="2" taskname="T2_cond" blocking="true">
        <arg id="2"/> <arg id="5"/> <arg id="6"/>
        <arg value="size" type="unsigned_int"/>
        <arg value="count" type="unsigned_int"/>
      </grpc_call>
      <grpc_call call_id="3" call_site="3" taskname="P1_cond" blocking="true">
        <arg id="6"/>
        <arg value="size" type="unsigned_int"/>
        <arg value="count" type="unsigned_int"/>
        <arg value="(double)0.5" type="double"/>
        <arg value="p" type="int"/>
      </grpc_call>
      <if>
        <then>
          <grpc_call call_site="4" taskname="T3_cond" blocking="false">
            <arg id="2"/> <arg id="0"/>
            <arg value="size" type="unsigned_int"/>
            <arg value="count" type="unsigned_int"/>
          </grpc_call>
          <grpc_call call_id="5" call_site="5" taskname="T3_cond" blocking="false">
            <arg id="5"/> <arg id="3"/>
            <arg value="size" type="unsigned_int"/>
            <arg value="count*2" type="unsigned_int"/>
          </grpc_call>
          <grpc_wait_all/>
          <grpc_call call_id="6" call_site="6" taskname="T2_cond" blocking="true">
            <arg id="2"/> <arg id="5"/> <arg id="6"/>
            <arg value="size" type="unsigned_int"/>
            <arg value="count" type="unsigned_int"/>
          </grpc_call>
        </then>
        <else>
          <grpc_call call_id="7" call_site="7" taskname="T3_cond" blocking="false">
            <arg id="2"/> <arg id="1"/>
            <arg value="size" type="unsigned_int"/>
            <arg value="count" type="unsigned_int"/>
          </grpc_call>
          <grpc_call call_id="8" call_site="8" taskname="T3_cond" blocking="false">
            <arg id="5"/> <arg id="4"/>
            <arg value="size" type="unsigned_int"/>
            <arg value="count*2" type="unsigned_int"/>
          </grpc_call>
          <grpc_wait_all/>
          <grpc_call call_id="9" call_site="9" taskname="T2_cond" blocking="true">
            <arg id="2"/> <arg id="5"/> <arg id="6"/>
            <arg value="size" type="unsigned_int"/>
            <arg value="count" type="unsigned_int"/>
          </grpc_call>
        </else>
      </if>
    </grpc_map_static>
  </body>
</function>
```

Listing 4.4: Example of algorithm tree structore. Many XML elements and attributes are omitted for brevity.

4.3.3 Expanded expressions

An expanded expression is built from expression by replacing variables with their initialisation expressions and replacing function arguments by expressions this function was called with as arguments. This process continues recursively either until there is nothing to replace or until some specified level is reached: either top level (`main()` function) or level of `grpc_map_static()` block.

For example, GridRPC call argument is `(x * 3)`, but its expanded representation is `(10 + 1) * 3` in the following code.

```
1  void foo(grpc_function_handle_t *h, int x)
2  {
3      grpc_call(h, x * 3);
4  }
5
6  int main()
7  {
8      grpc_function_handle_t h;
9      grpc_function_handle_default(&h, "task1");
10     grpc_map_static("ex_map") {
11         int a = 10;
12         foo(&h, a + 1);
13     }
14 }
```

The expansion in this example is done by replacing `x` argument with actual argument expression `a + 1` and then replacing `a` variable with its initialisation value `10`.

Expanded expressions can exist in tree form and in textual representation. Tree form is used when building algorithm tree structure. Listing 4.4 at page 48 has expression trees omitted for brevity, but `arg` element at line 16 actually looks like following.

```
1  <arg value="count*2" type="unsigned int">
2    <BinaryOperator kind="*" type="unsigned int">
3      <DeclRefExpr name="testcount" type="unsigned int"
4                   kind="Var" context="43762856"/>
5      <ImplicitCastExpr kind="IntegralCast"
6                        type="unsigned int">
7        <IntegerLiteral type="int">2</IntegerLiteral>
8      </ImplicitCastExpr>
9    </BinaryOperator>
10 </arg>
```

The `value` attribute contains textual representation of argument's expression expanded until `grpc_map_static()` block. This expression is used as runtime parameter of the algorithm. Runtime parameters are described in section 4.2. Expression representing this parameter's value at the scope of `grpc_map_static()` block is inserted during preprocessing step as runtime parameter's value as described below in subsection 4.3.4.

The tree inside `arg` element is tree representation of argument's expression expanded until `main()` function. The `DeclRefExpr` element of the tree represents a variable called `testcount` in `main()` function's scope. This reveals that this variable is passed as `run()` function's `count` argument. The `context` attribute is a unique ID of the scope where the variable is declared. Address of variable's primary declaration context is used for this ID. Using context ID allows to differentiate between variables with the same name declared in different contexts.

By storing values indexed by expanded expression tree, the following items are being correlated to point to the same value:

- GridRPC calls with GridRPC function handle initialisations to find task named for GridRPC calls;
- session IDs in asynchronous GridRPC calls with corresponding GridRPC wait calls;
- non-scalar GridRPC call arguments to find data dependencies between tasks later.

4.3.4 Preprocessing source files by inserting data definitions and code

After AST trees are analysed and analysis tree is built, files containing `grpc_map_static()` directive and GridRPC calls are preprocessed. Information collected during static code analysis and code to use this information is inserted into new source files built from the original ones.

Listing 4.5 at page 51 shows parts of the code inserted by preprocessor after processing source module listed in Listing 4.1 at page 40.

The following information is inserted into `grpc_map_static()` block as static data definitions.

Argument descriptions for each task

Argument ID for non-scalar arguments and runtime argument ID for scalar arguments. Examples of these descriptions are shown at lines 3 and 10.

```
1   grpc_map_static("ex_map") {
2       /* —— STATIC INITIALISATION —— */
3       static grpc_task_arg __grpc_task0_args[] = {
4         {GRPC_TASK_ARG_ID, 0},
5         {GRPC_TASK_ARG_ID, 1},
6         {GRPC_TASK_ARG_ID, 2},
7         {GRPC_TASK_ARG_RUNTIME, 0},
8         {GRPC_TASK_ARG_RUNTIME, 1},
9       };
10      static grpc_task_arg __grpc_task1_args[] = {
11        {GRPC_TASK_ARG_ID, 3},
12        {GRPC_TASK_ARG_ID, 4},
13        {GRPC_TASK_ARG_ID, 5},
14        {GRPC_TASK_ARG_RUNTIME, 0},
15        {GRPC_TASK_ARG_RUNTIME, 2},
16      };
17      ...
18      static grpc_task_descr __grpc_tasks[] = {
19        {"T1_cond", __grpc_task0_args, 5, 0},
20        {"T1_cond", __grpc_task1_args, 5, 1},
21        ...
22      };
23      ...
24      static grpc_task_event __grpc_task_events_1[] = {
25        {GRPC_TASK_START, &__grpc_tasks[4]},
26        {GRPC_TASK_START, &__grpc_tasks[5]},
27        {GRPC_TASK_WAIT, &__grpc_tasks[4]},
28        {GRPC_TASK_WAIT, &__grpc_tasks[5]},
29        {GRPC_TASK_START, &__grpc_tasks[6]},
30        {GRPC_TASK_WAIT, &__grpc_tasks[6]},
31      };
32      ...
33      static grpc_task_par __grpc_task_par_0[] = {
34       {__grpc_task_events_1, 6},
35       {__grpc_task_events_2, 6},
36      };
37      ...
38      static grpc_task_event __grpc_task_events_0[] = {
39        {GRPC_TASK_START, &__grpc_tasks[0]},
40        {GRPC_TASK_START, &__grpc_tasks[1]},
41        {GRPC_TASK_WAIT, &__grpc_tasks[0]},
42        {GRPC_TASK_WAIT, &__grpc_tasks[1]},
43        {GRPC_TASK_START, &__grpc_tasks[2]},
44        {GRPC_TASK_WAIT, &__grpc_tasks[2]},
45        {GRPC_TASK_START, &__grpc_tasks[3]},
46        {GRPC_TASK_WAIT, &__grpc_tasks[3]},
47        {GRPC_TASK_PARALLEL, __grpc_task_par_0},
48      };
49      /* —— RUNTIME INITIALISATION —— */
50      grpc_map_runtime_arg __grpc_map_runtime_args[6];
51      __grpc_map_runtime_args[0].type = GRPC_MAP_RUNTIME_ARG_INT;
52      __grpc_map_runtime_args[0].value.i = size;
53      ...
54      grpc_map_build_app_pm("greedy_map",
55                            __grpc_tasks, 10,
56                            __grpc_task_events_0, 10,
57                            __grpc_map_runtime_args, 6);
58      /* —— USER CODE —— */
59      grpc_map_embed_call(&t1_0, 0);
60      grpc_call_async(&t1_0, &sid0, a0, b0, c0,
61                      size, count);
62      ...
63  }
```

Listing 4.5: Code inserted by preprocessor. Only part of the code is shown.

Task descriptions
: Task name, arguments and call site ID for each task. Example of task description array is shown at line 18.

Task events
: Array of events describing program data flow. This array constitutes event block. There can be several task event arrays for different event blocks. The reason for multiple event blocks is described below. Examples of these event blocks are shown at lines 24 and 38.

Data flow in the algorithm is described by task events. Task events have different types and all of them reference data specific to their type. There are following task event types.

START
: Starting remote task. References task description for task to be started.

WAIT
: Waiting for remote task completion. References task description for task to be waited for.

PARALLEL
: Branching inside `if` statement, which is treated as parallel code execution. References array or two structures which in turn reference event blocks for then and else branches.

LOOP
: Repeating events inside a loop. References a structure which references in turn event block for loop body and contains index of runtime parameter for number of loop iterations.

The following runtime initialisation is inserted after static data definitions into `grpc_map_static()` block.

- Assignment of actual values to runtime parameters. Example is shown at line 50.
- `grpc_map_build_app_pm()` function call with the following arguments:
  - heuristic name;
  - task descriptions array;
  - task events array for the main event block;
  - runtime parameters array.

Example is shown at line 54.

The `grpc_map_build_app_pm()` function builds application performance model and task dependency graph and stores them in static data structure. Then it uses specified heuristic to find task-to-server mapping and also stores it in static data structure.

Also, calls of `grpc_map_embed_call()` functions are inserted before each GridRPC call. The `grpc_map_embed_call()` function uses call site ID passed as its second argument and uses is as an index to find suitable server for the remote task and embed it into GridRPC function handle. Also it arranges server-to-server communication and stores relevant information into function handle. Example is shown at line 59.

4.4 Extended client library

Running code adapted to use static code analysis approach requires extended API provided by modified SmartGridSolve client library. The following extensions are used.

`grpc_map_build_app_pm()` **function**
: This function is inserted by preprocessor into the beginning of code in `grpc_map_static()` block. See description of this function above in subsection 4.3.4.

`grpc_map_embed_call()` **function**
: This function is inserted by preprocessor before each GridRPC call in `grpc_map_static()` block. See description of this function above in subsection 4.3.4.

`grpc_map_static()` **directive**
: This directive marks block of code containing algorithm for collective mapping as described in section 4.1. This is a macro expects one parameter called `name` and expands into the following code.

```
if (grpc_map_static_init((name)))
```

The `grpc_map_static_init()` function is declared as static and it does nothing besides returning `1` constant.

Hence, preceding code block with `grpc_map_static()` directive changes nothing from compiler's point of view. It's just needed to make static code analyser detect this meaningless `if` statement as block of code for collective mapping.

`grpc_likely()` **directive**

This directive is actually just a static function that accepts any arguments, returns void and does nothing.

The only purpose of this function is making static code analyser detect expression passed as its argument as likely number of loop iterations if call to this function is placed at the beginning of a loop.

Modifications to SmartGridSolve client library to implement extended API described above are very limited and non-intrusive. Only new functions and data type definitions were added, but no existing functions and data structures (either internal or publicly visible) were changed.

However, extended API functions use private SmartGridSolve functions, data structures and RPC calls to retrieve information needed for collective mapping, build application performance model and embed task-to-server mapping and information needed to arrange server-to-server communication info GridRPC function handles.

4.5 Running application

The `grpc_map_build_app_pm()` function uses the following information to build application performance model and task dependency graph:

- information collected during static code analysis: descriptions of tasks, task arguments and task events;
- actual values of runtime parameters;
- GridRPC remote task call signatures collected from remote servers to determine which arguments are input, output or input-output.

4.5.1 Building application performance model

The original task events list contains static data which is generated by preprocessor as a result of static code analysis and compiled in as static data definitions inside the body of `grpc_map_static()` block. Each START and WAIT event point to static task description which contain information discovered during static code analysis. However, this static data is not sufficient for building application performance model and task dependency graph for the following reasons.

- There is additional information which is discovered at runtime:

  runtime parameters values: depend on expressions calculated at runtime;

argument sizes: depend on runtime parameters and task signatures retrieved from servers;

argument direction (in, out, inout): depend on task signatures retrieved from servers.

- Task descriptions inside a LOOP event block correspond to more than one remote task invocation. It's impossible to expand a loop into its body repeated several times during static code analysis and preprocessing because number of loop iterations can depend on runtime parameter which is known at runtime only.

The first thing `grpc_map_build_app_pm()` function does after initialising its internal data structures, is expanding task events list. The LOOP events are replaced with their event blocks repeated number of times which is already known and represented by loop's runtime parameter's value. Also, instead of pointing to static task descriptions, expanded event list points to task nodes, which are generated at runtime and contain the following information.

- Pointer to static task description this task node corresponds to.
- Pointer to problem description retrieved from server. This description contains additional information about the task:
  - arguments direction;
  - expressions to compute non-scalar argument sizes based on scalar argument values;
  - expression to compute task complexity based in scalar argument values.
- Task's computational complexity.
- Subset of servers this task is available on (from a global set of servers available in the grid).
- Lists of task dependencies. This list is empty on this stage; it will be used at the next stage described in subsection 4.5.2.

This expanded event list is essentially application performance model.

4.5.2 Building task dependency DAG

Task dependency graph is a DAG (direct acyclic graph) with tasks as nodes and data dependencies between tasks as vertices. Task nodes built in the previous step described in subsection 4.5.1 are used as nodes in this DAG.

Task B is dependent on task A if the following conditions are met.

- Task B is guaranteed to start when A is finished. This condition is met when WAIT event for A comes before START event for B.
- There is a non-scalar output (or input-output) argument of task A and a non-scalar input (or input-output) argument of task B with the same argument ID. The same argument ID means that both arguments point to the same data, so output from task A is used as input for task B.

There are as many arrows from A task's node to B task's node as there are arguments passed from A to B. Each dependency makes its argument eligible for server-to-server communication.

Task dependency graph generated by `grpc_map_build_app_pm()` function for example application on Listing 4.1 at page 40 is shown on Figure 4.3. Each arrow's label contains three numbers: argument ID, output argument number (counted from 0), input argument number (counted from 0). For example, the arrow between leftmost `T1_cond` and leftmost `T2_cond` has the label: 5: $2 \rightarrow 1$, which means that argument ID is 5, data comes from `T1_cond` task as output argument number 2 and is passed to `T2_cond` task as input argument number 1.

The purpose for task dependency DAG is arranging server-to-server communication.

Please note that task dependency DAG on Figure 4.3 contains nodes for tasks in both branches of `if` statement, despite that only one branch will be executed in the algorithm. This is because it's unknown in advance which branch will be executed, so both branches are treated as being executed in parallel. Hence, the resulting DAG covers the actual task dependency DAG which is unknown in advance, containing superset of its nodes.

4.5.3 Applying heuristic

The next thing `grpc_map_build_app_pm()` function does is finding task-to-server mapping using heuristic with a name which is specified as an argument for `grpc_map_static()` block.

The task of finding optimal server-to-server mapping is NP-complete. There are several heuristics available to find mapping which is good enough [12]. It is programmer's responsibility to choose the heuristic which produces best results in acceptable time.

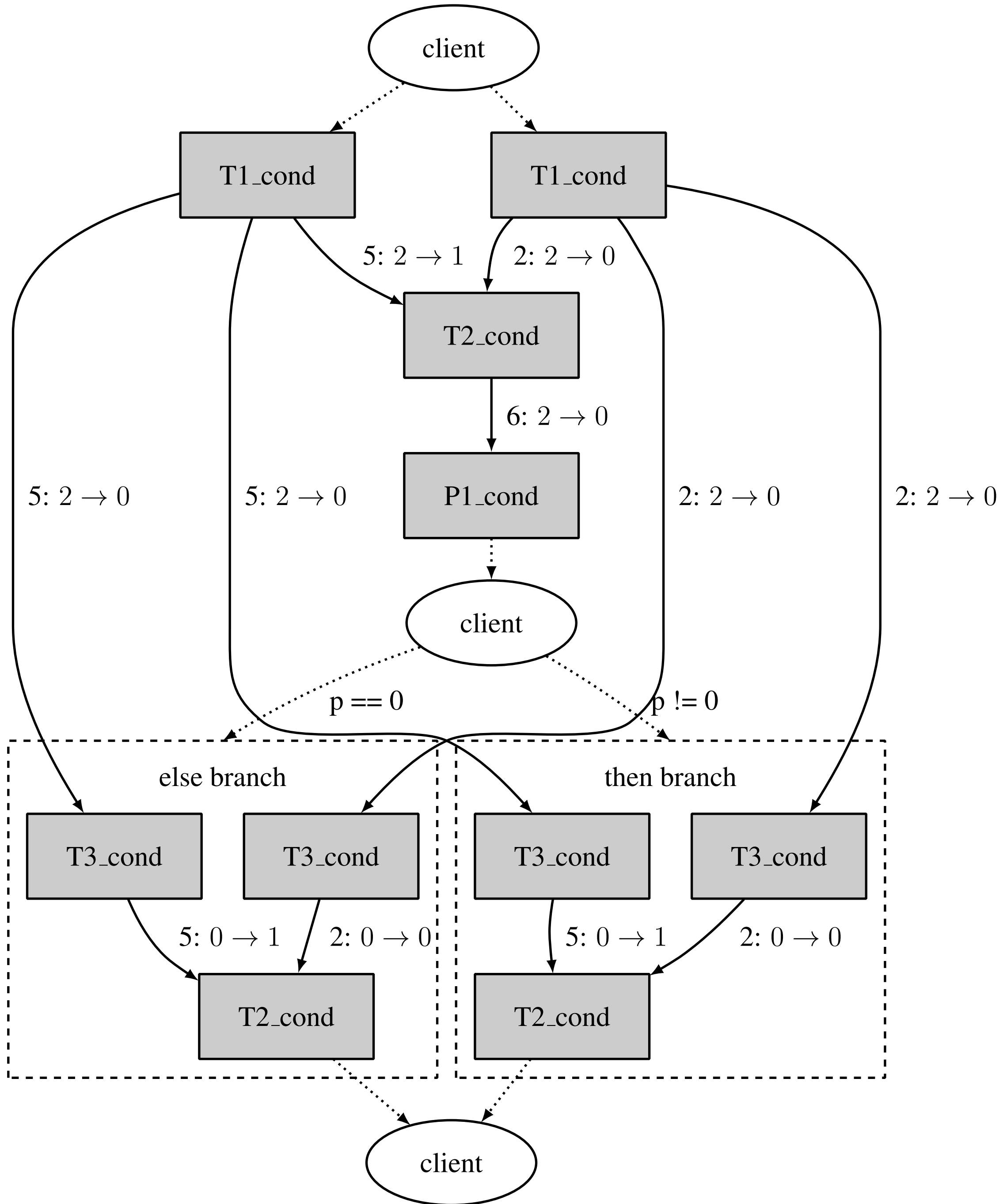


Figure 4.3: Task dependency graph generated for example application

There is also heuristic called `ex_map` available, which implements exhaustive search. This heuristic has complexity of s^t, where s is number of servers and t is number of tasks. It can be used when number of servers and tasks is very low.

4.5.4 Server-to-server communication

The `grpc_map_embed_call()` function calls inserted by preprocessor before all GridRPC calls write data into function handle to arrange server-to-server communication.

For better efficiency, SmartGridSolve uses push communication. This means that server-to-server communication is initiated by a server that produced results when a task running on this server has been completed, rather than by a server expecting input arguments for a task that is about to be started. Hence, server running a task should have information where to send its output arguments after the task has been completed. There are already fields in function handle used for SmartGridRPC for storing a list of servers to send output to as well as file names it expects to find input arguments in. Values in these fields are sent in string fields of RPC request which starts a task.

When a server receives RPC request to start a task, it waits for files where it expects to find input arguments to appear in its filesystem. Only after all files containing input arguments have appeared, the task is started. Upon task completion, output argument sending is initiated.

A file name for input and output arguments is based on argument ID. This way it's guaranteed that name of a file task expects to find its input argument is the same as name of file where this argument is sent from another task.

It's not a problem if an input argument can come from either one or another server dependent on which branch of `if` statement has been executed. The server waits for a file where the input argument is expected to appear in filesystem, and the file name will be the same, no matter which server it came from and which task has produced this file as its output, because file name is dependent on argument ID.

Also, it's not a problem if tasks in both branches if `if` statement depend on the same output argument. The server running the task producing this output argument will send it to servers assigned to tasks in both branches just in case, because it's unknown in advance which branch will be executed.

4.6 Restrictions on code to be analysed statically

Although static code analyser can cover many regular cases, it's far from perfect and can be confused if code is too complex. This section lists known cases when

code can't be fully analysed and its workflow can't be extracted correctly.

Local and non-local jumps

Although control flow used in normal structured programming (loops, conditional statements, function calls) is recognised, all other ways to jump in code is not taken into account. Exceptions, `goto` statements, non-local jumps with `setjmp()` and `longjmp()` functions are not considered when extracting algorithm workflow. These methods to exit scope are usually used for handling exceptional situations, so using them instead of normal control flow statements is not a good practice anyway.

Other control flow statements

The `return`, `break` and `continue` statements are not recognised as well when extracting algorithm workflow. It's not recommended to have these statements inside `grpc_map_static()` block. If a loop is exited with `break`, it's recommended to rewrite it to use more complex loop condition. If there are GridRPC calls after `break` or `continue` statements inside a loop which are dependent on some condidion, it's recommended to eliminate these statements by placing remaining part of the loop inside `if` statement. Otherwise, extracted workflow may be incorrect.

Runtime parameters unknown at the beginning of algorithm

Workflow can be dependent on runtime parameters. As it was described in section 4.2, these parameters are detected automatically and can affect number of loop iterations as well as sizes of input and output arguments of GridRPC calls. The problem is, value of these parameters has to be known at the point when `grpc_map_build_app_pm()` function inserted in the beginning of `grpc_map_static()` block performs collective mapping. All runtime parameter expressions are expanded as described in subsection 4.3.3. If this conversion is impossible, algorithm workflow can't be extracted. This means that all runtime parameter values should be known at the beginning of code in `grpc_map_static()` block. There is still common case when number of loop iterations is not known in advance because loop completion depends on remote computation results. In this case programmer should estimate number of iterations and use `grpc_likely()` directive in the loop beginning to specify this estimated number.

Using different variables to access the same data object

Sometimes static analyser can fail to recognise that different expressions point to the same non-scalar data object (vector or matrix). Expressions are expanded as described in subsection 4.3.3, and arguments are not recognised as pointing to the same data if expanded expressions are different.

Failing to recognise that two argument expressions point to the same data can lead to data dependency not recognised, and server-to-server communication not arranged.

Local modifications to data objects

More serious problems can happen when wrong data dependency is detected. This can happen if data object is locally modified between GridRPC calls when next call uses results from previous call. Server-to-server communication is arranged in this case, and the next task will receive unmodified result from previous task. Also, server-to-server communication can preclude sending result to the client, so local modifications will be made on completely bogus data (garbage remaining in data object) in this case. This can lead to incorrect algorithm execution. It's recommended to write algorithm the way that all data are prepared before starting parallel algorithm in `grpc_map_static()` block and not modified in the middle.

GridRPC calls in libraries

All code sufficient for extracting workflow should be available for static code analysis. All functions that have no definition with body available for static code analysis are considered to belong to libraries which contain no GridRPC calls. If there are still functions with code not available for static code analysis that perform GridRPC calls, these calls are not included in collective mapping. As a result, individual mapping is used for these calls.

Using function pointers to call GridRPC API

Static code analyser can detect only direct GridRPC API calls. Any indirection in calling these functions, like using function pointers, will confuse static code analyser. GridRPC API should be used directly, without any kind of indirection.

Chapter 5

Use case: HydroPad

Listing 4.1 at page 40 is a simplified version of the application that was used for testing static code analyser, code generated by preprocessor and library code added to SmartGridSolve during software development. The full version of this application is a full program containing correct error handling. Also, a modification of this program with loops was used to test static code analyser and preprocessor when loop unrolling support was introduced in its code. The remote tasks used during this testing are not very useful for real scientific computations: they just add or multiply vectors of size specified by `size` argument number of times specified by `count` argument.

In order to prove that static code analysis approach presented in this thesis is actually a viable and useful solution, it has to be applied to a real scientific application. To fulfil this role, HydroPad [43], a real life astrophysics application was selected, which simulates the evolution of galaxy clusters in the Universe.

HydroPad was already modified to use GridRPC and SmartGridRPC API [26], and it was extensively with SmartGridSolve [8]. As a result, HydroPad became a natural choice for applying static code analysis approach.

5.1 HydroPad overview

HydroPad is an astrophysics application that simulates development and evolution of the Universe. The simulation assumes that all matter in the Universe consists of two types: baryonic matter and dark matter. Baryonic matter is what all observable objects are made of. Dark matter is a matter which is not observable by regular means, but can be detected only via its gravitational force. The most prevalent theory is that most of gravitational mass in the Universe consists of dark matter.

Dark matter behaviour is computed using Particle-Mesh N-Body algorithm [28].

It simulates large number of collision-less particles in gravitation field, which is calculated using density grid. Particles are moved between grid cells according to gravitational force in cells where they are located.

Baryonic matter behaviour is computed using hydrodynamic algorithm called Piecewise-Parabolic-Method [17], a higher order method for solving partial differential equations.

Densities of baryonic and dark matters after each time step are used to calculate gravity force field using the Fast-Fourier-Transform (FFT) method to solve the Poisson equation.

HydroPad adapted to GridRPC uses the following remote tasks.

initgrav
Initialise the shape function.

usegrafic
Use the program grafic to initialise the dark matter.

densitydm
Calculate the density of the dark matter.

densitydmtsc
Calculate the density of the dark matter.

initbm
Initialise baryonic components.

initvel
Initialise velocity components.

fields
Calculate the gravitational component.

barmatter
Calculate the baryonic matter component.

darkmatter
Calculate the dark matter component.

darkmattertsc
Calculate the dark matter component.

Note that `densitydm` and `darkmatter` tasks have variants `densitydmtsc` and `darkmattertsc`, which preform the same calculations with triangulated shape clouds algorithm. A variant to use is selected by command line argument (TSC variant is used by default).

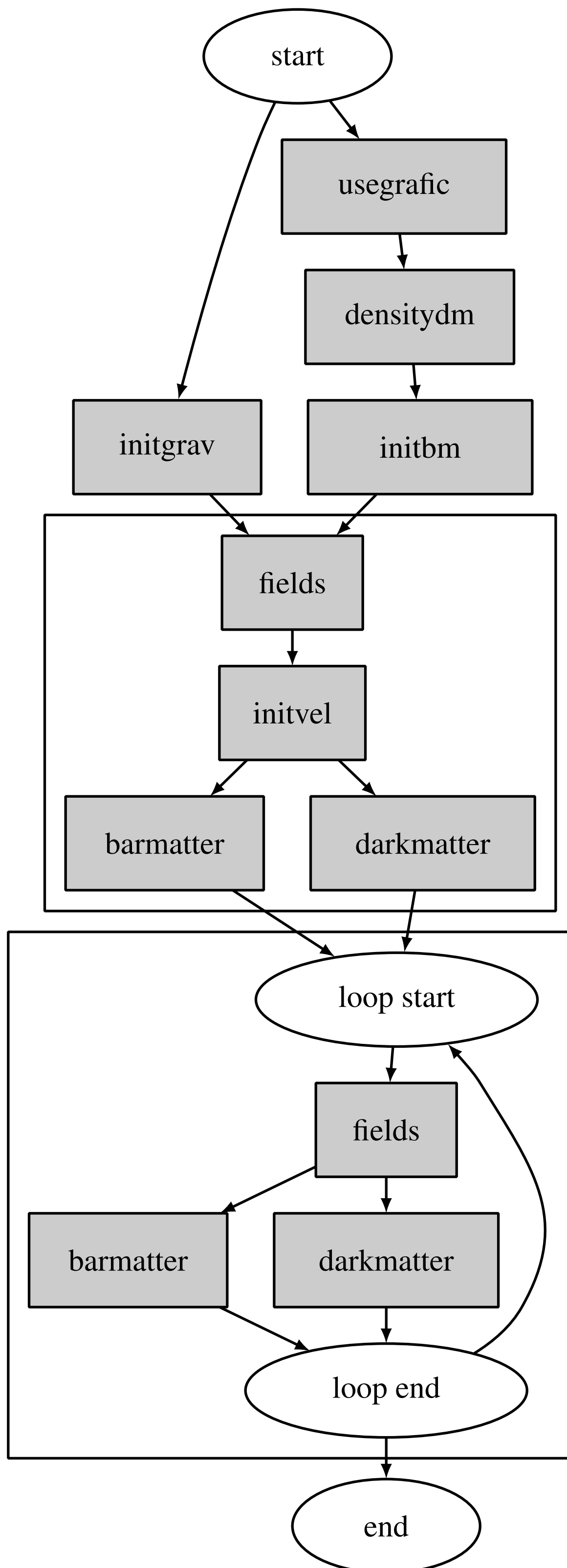


Figure 5.1: HydroPad application workflow

HydroPad algorithm works as following. First, baryonic matter and dark matter simulations are initialised in parallel. Then, the main loop is started, which contains the following steps.

- Gravitational field is calculated with `fields` task.
- On the first iteration, velocity components are initialised with `initvel` task.
- Baryonic and dark matter behaviour is simulated in parallel with `barmatter` and `darkmatter` tasks.

Number of main loop iterations is specified by command line argument. HydroPad algorithm workflow is shown on Figure 5.1.

5.2 Modifications to HydroPad

HydroPad package adapted to SmartGridRPC uses GNU autotools for its build. Its generated `Makefile` has several build targets:

hydropad_seq: sequential program without using remote tasks,

hydropad_gs: parallel program using regular GridRPC,

hydropad_smart: parallel program using SmartGridRPC.

To adapt HydroPad to static code analysis approach a new build target was added: `hydropad_static` by copying `hydropad_smart` target and its sources with following modifications.

Adapting to new interface.
Directive starting block of code for collective mapping was changed from `grpc_map()` to `grpc_map_static()`. Also, `grpc_local()` directives were removed because they are not needed anymore.

Making first loop iteration a special case.
First loop iteration is special because it contains a call to `initvel` task. Originally it's done inside `if` statement which checks for iteration number. Unfortunately, this approach confuses static code analyser because it doesn't recognise that `initvel` task is run only on the first iteration, so it adds unnecessary task dependencies involving `initvel` task for every loop iteration.

This problem was solved by making first iteration a special case and moving it outside of the loop.

Removing a choice between dark matter simulation algorithms.

There is a command line option selecting which of two variants of dark matter simulation algorithms to select. Originally, algorithm to use was selected by `if` statement in every loop iteration, checking the value of the flag set by this option. Unfortunately, this approach confuses static code analyser because it doesn't recognise that the same branch is always selected in every loop iteration. As a result, both tasks are considered for execution, adding unnecessary task dependencies involving them.

The problem was solved by hardcoding the algorithm to use.

Using initialisation instead of assignment for runtime parameter

There is `ga` variable which is recognised as a runtime parameter. Unfortunately, static code analyser doesn't recognise variable assignments properly, so it was unable to expand the expression representing this runtime parameter.

The problem was solved by changing the code to initialise this variable in its declaration instead of assigning it later in the middle of code.

Without two last changes, HydroPad algorithm workflow was incorrect, as shown on Figure 5.2, leading to many unnecessary task dependencies.

After changes listed above were made, HydroPad started working correctly with static code analysis approach.

Task dependency graph of HydroPad algorithm with 3 iterations is shown on Figure 5.3.

Essential part of HydroPad source code adapted for SmartGridRPC is listed in Appendix C. The same code modified for static code analysis approach is listed in Appendix D.

5.3 Experimental results

This section presents experimental results confirming performance improvements of HydroPad application when using collective mapping compared to plain GridRPC. These experiments were made earlier and are presented in this thesis: [6]. They compare performance of GridRPC and SmartGridRPC versions of HydroPad.

The main advantage of static code analysis approach is not improved performance compared to other approaches to collective mapping, but rather improved programmability achieved by less restrictions on code compared to runtime discovery approach and more convenient programming model compared to ADL and workflow submission approaches, while keeping the same performance.

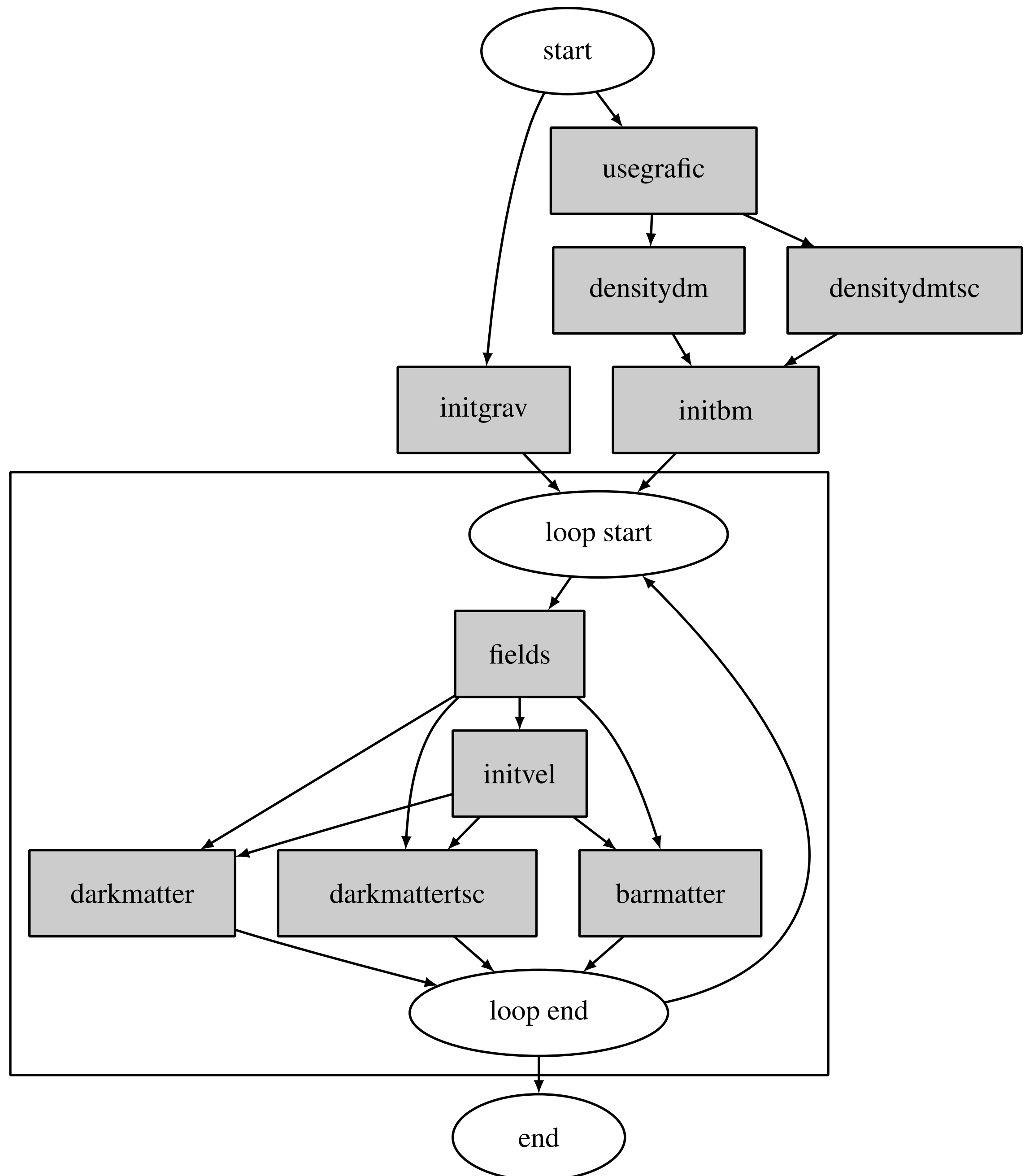


Figure 5.2: Wrong HydroPad application workflow detected without adaptation

Figure 5.3: HydroPad task dependency graph

Application performance model built from application workflow extracted from HydroPad using static code analysis approach is exactly the same as he one built as a result of runtime discovery used by SmartGridRPC. Hence, resulting task-to-server mapping and server-to-server communication is exactly the same.

5.3.1 Hardware configuration

The experimental grid used in the experiments consists of two heterogeneous servers, $S1$ and $S2$, with 498 and 531 MFlops performance respectively and having 1 GB of memory each. Both servers are interconnected by 1 Gbit/s link.

Four client setups were used in experiments. These setups are described in the following table.

Setup name	Network connection	Amount of memory
C1-1	1 Gbit/s	1 GB
C1-256	1 Gbit/s	256 MB
C100-1	100 Mbit/s	1 GB
C100-156	100 Mbit/s	256 MB

5.3.2 Experimental data

Experiments were conducted with different initial parameters resulting in different memory usage. These experiments are listed in the following table.

Problem ID	N_p	N_g	Data size
P1	120^3	60^3	73 MB
P2	140^3	80^3	142 MB
P3	160^3	80^3	176 MB
P4	140^3	100^3	242 MB
P5	160^3	100^3	270 MB
P6	180^3	100^3	313 MB
P7	200^3	100^3	340 MB
P8	220^3	120^3	552 MB
P9	240^3	120^3	624 MB

5.3.3 Experimental results

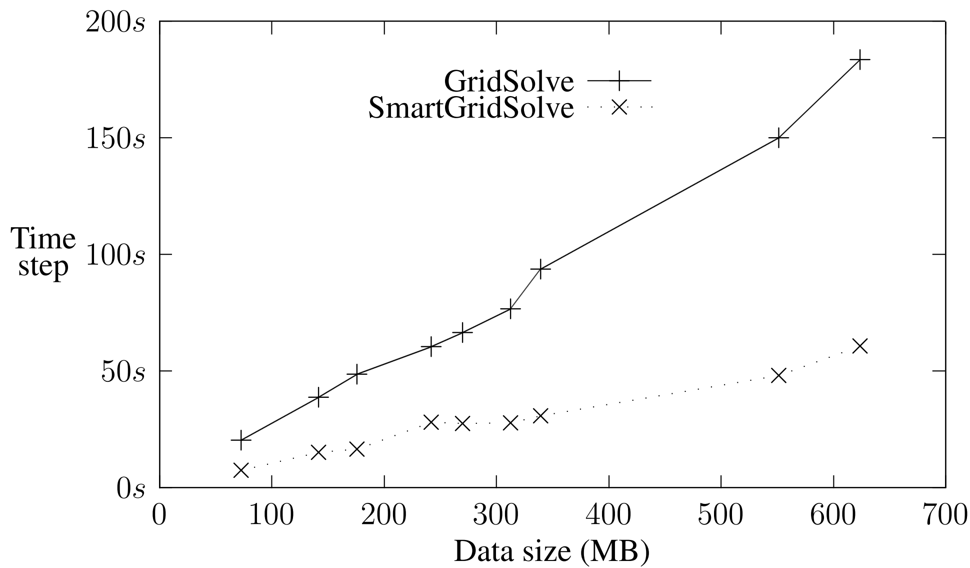
Results of experiments for different client setups are listed in the following tables.

It's clearly visible that algorithm performance is always better when collective mapping is used. Also, algorithm performance with collective mapping depends less on client configuration because most of the data objects are stored on servers

and are sent between servers directly, without involving the client in this communication.

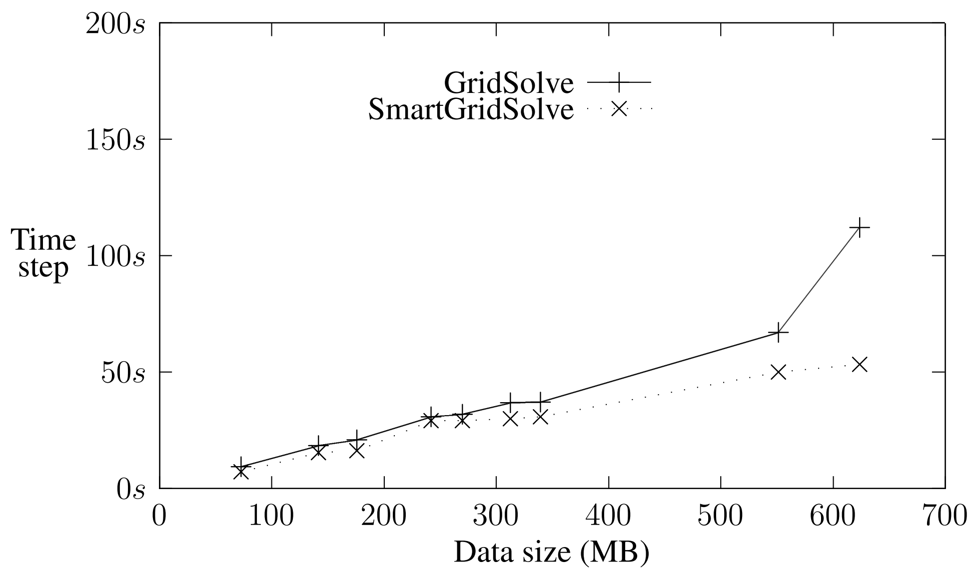
C100-256. This client setup has slow network connection and small amount of memory. Paging starts on the client for P7, P8 and P9 problems without server-to-server communication.

Problem ID	Time step (GridSolve)	Time step (SmartGridSolve)
P1	20.26s	7.31s
P2	38.75s	15.06s
P3	48.65s	16.36s
P4	60.48s	28.06s
P5	66.43s	27.54s
P6	76.76s	27.78s
P7	93.74s	30.81s
P8	150.03s	48.04s
P9	183.45s	60.74s



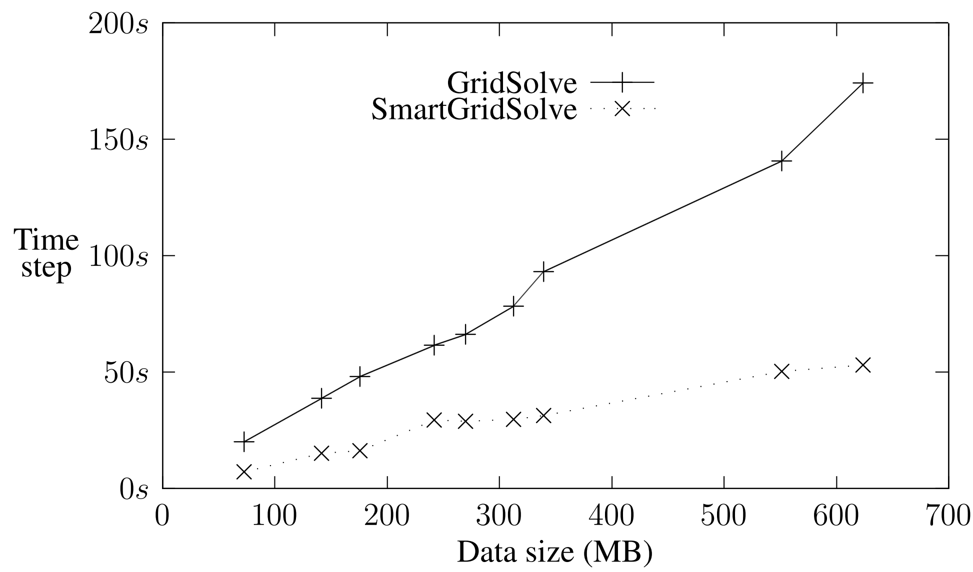
C1-1. This client setup has fast network connection and amount of memory large enough to accommodate data for any problem.

Problem ID	Time step (GridSolve)	Time step (SmartGridSolve)
P1	9.40s	7.09s
P2	18.38s	15.27s
P3	20.82s	16.17s
P4	30.81s	29.02s
P5	32.00s	28.99s
P6	36.81s	29.88s
P7	37.22s	30.88s
P8	67.04s	50.05s
P9	112.05s	53.35s



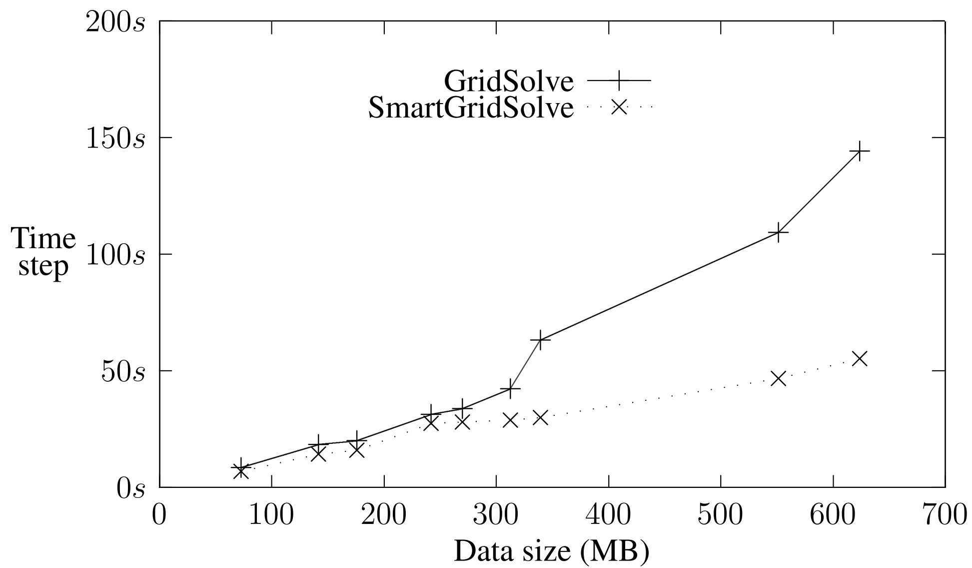
C100-1. This client setup has slow network connection and amount of memory large enough to accommodate data for any problem.

Problem ID	Time step (GridSolve)	Time step (SmartGridSolve)
P1	19.97s	7.24s
P2	38.73s	15.17s
P3	48.20s	16.24s
P4	61.59s	29.42s
P5	66.26s	28.91s
P6	78.16s	29.73s
P7	93.20s	31.25s
P8	140.53s	50.20s
P9	174.14s	53.02s



C1-256. This client setup has fast network connection and small amount of memory. Paging starts on the client for P7, P8 and P9 problems without server-to-server communication.

Problem ID	Time step (GridSolve)	Time step (SmartGridSolve)
P1	8.6s	7.0s
P2	18.4s	14.4s
P3	20.1s	15.8s
P4	31.3s	27.5s
P5	33.7s	28.1s
P6	42.3s	28.8s
P7	63.1s	30.0s
P8	109.3s	46.6s
P9	144.3s	55.1s



Chapter 6

Conclusions

In this thesis, a new approach to collecting information needed for collective task-to-server mapping is presented. This approach allows to improve convenience of writing or porting GridRPC applications to take advantage of collective mapping, which in turn allows server-to-server communication.

Regular GridRPC programming model is discussed in section 1.3. It uses individual mapping and strict client-server communication. The best solution to individual mapping is collective mapping. Collective mapping allows to map tasks to servers more optimal way leading to improved computational performance. Also, collective mapping allows to arrange direct server-to-server communication, that leads to improved communication performance and allows communication parallelism. Collective mapping and its advantages is discussed in section 3.2.

Direct server-to-server communication arranged as a result of collective mapping is push communication that is more efficient than one allowed by approaches not involving collective mapping: all task-to-server mapping is known in advance, so sending results of tasks to servers where they are expected by future tasks can be started immediately after tasks produced these results, not waiting for tasks expecting these results to be started by the client. Approaches to grid optimisation other than collective mapping are discussed in section 3.1.

Collective mapping requires information about algorithm workflow to be available right before the first remote task of the algorithm has been started. Existing approaches are discussed in chapter 3. However, these approaches have their own limitations.

Runtime discovery approach discussed in section 3.4) puts very strict restrictions to algorithm code which are difficult to meet and violations of them are very difficult to detect and lead to incorrect algorithm behaviour.

ADL approach discussed in section 3.5 requires a separate language describing algorithm workflow to be written and kept in sync with the actual program

implementing this algorithm.

Workflow submission approach discussed in section 3.6 requires the whole algorithm to be written in a specialised workflow description language.

Static code analysis approach presented in this thesis works by extracting algorithm workflow from the code of program implementing the algorithm. It doesn't require writing a separate workflow description manually and places much milder restrictions on code than runtime discovery approach. Also, some of these restrictions can be lifted gradually when static code analyser improves and becomes more sophisticated during the course of its future development. Results of this work were published here: [23], [11].

It's impossible to extract correct workflow from all all possible algorithms. However, workflow extracting heuristic used by static code analyser presented in this thesis is good enough to be applied to very wide range of real world applications. For example it was successfully applied to HydroPad, an astrophysics application that simulates development and evolution of the Universe. This was done with minimal modifications to HydroPad. Some of these modifications can be made unnecessary by further development of static code analyser. Applying static code analysis approach to HydroPad is discussed in chapter 5.

6.1 Future work

Static code analyser and preprocessor presented in this thesis produces good results and works with wide range of cases. However it's far from ideal and can be improved in several ways.

6.1.1 Improving static code analyser to cover more cases

Most important direction for further development is improving static code analyser's workflow extraction heuristic to cover more cases.

The following improvements can be made in this area through incremental development.

Analysing assignments.

Currently expression expansion support (discussed in subsection 4.3.3) is currently quite rudimentary. It takes passing function arguments and variable initialisations into account, but not variable assignments. By taking assignments into account, more accurate expression expansion can be achieved, so more cases can be covered this way.

For example, analysing assignments can reveal cases of aliasing, when two variables point to the same data, allowing better detection of task dependencies.

Also, analysing assignments can reveal access to non-scalar data, revealing cases of object modifications which should preclude task dependency and resulting server-to-server communication.

This feature will make the fourth change to HydroPad discussed in section 5.2 unnecessary.

Detecting values unchanged inside algorithm

Currently runtime parameters (discussed in section 4.2) are detected only for values which must be runtime parameters, like GridRPC call scalar arguments and expressions specifying number of loop iterations. However, there are cases when expressions can have different meaning for algorithm workflow dependent on whether their value is predetermined before algorithm start or is calculated and assigned somewhere inside the algorithm.

One of such cases can be condition of `if` statement. If this condition expression is predetermined before algorithm start, it can be treated as a runtime parameter, and a branch of `if` statement to be executed is already known at the point of collective mapping. This means that code in another branch should not be considered for collective mapping at all, making task dependency DAG more simple and adequate to the real algorithm execution.

Implementing assignment analysing support discussed above will help in implementing this feature.

This feature will make the third change to HydroPad discussed in section 5.2 unnecessary.

Detecting variables dependent on loop iteration.

Some variables act as counters in a loop. Loop control variable in `for` statement is a simple case, but there can be other variables which are incremented and decremented in a loop. Improving heuristic to detect these variables can also produce more accurate expression expansion, allowing to cover more cases.

For example this feature will allow to cover a case when multiple GridRPC function handles are initialised in a loop, function handles are stored in an array indexed by loop variable, and later multiple tasks are started also in a loop using handles stored in the same array, also indexed by loop variable.

Combined with more intelligent `if` statement condition handling, this feature will make the second change to HydroPad discussed in section 5.2 un-

necessary.

6.1.2 Introducing fault tolerance support

Implementing fault tolerance is an important area of further improvement. Fault tolerance is easy to implement in case of individual mapping: the failed task can be restarted on another server with the same arguments. However, it becomes a problem in case of collective mapping, when server-to-server communication is used. A failed remote task can lead to its arguments computed by other tasks to be lost. In this case, it's not enough to restart the only task that has failed; other tasks it depends on have to be restarted as well.

Runtime discovery implemented in SmartGridSolve uses the simple approach: the whole algorithm is restarted from the beginning in case of a remote task failure, as described in subsection 3.4.2. This approach results in stricter constraints on code allowed for group mapping than expected, as it was described in subsection 3.4.3. Using the same approach will impose the same constraints and defeat benefits of single-pass algorithm execution allowed by static code analysis approach.

Instead of rolling back the algorithm to the beginning, which can be difficult or impossible considering side effects, another approach can be used.

The log of completed GridRPC calls is stored by the client along with their arguments which come from the client itself (without server-to-server communication). If a remote task has failed, all tasks from the whole dependency graph containing the failed task along with tasks it depends on are restarted. This will make all lost results to be recalculated, allowing the failed task to be restarted with the same arguments.

Also, improvements can be made on the server side by reducing likelihood of lost results due to task failures. For example, a server can store results of completed tasks little bit longer until all tasks dependent on these results are completed and their respective results are stored. Other forms of redundant result storage can be used as well.

6.1.3 Improving portability

Static code analyser makes no assumptions on when kind of grid middleware is used. However, the code generated by preprocessor contain calls to extended API to perform task-to-server mapping and arrange server-to-server communication. This extended API has been implemented inside extended client library from SmartGridSolve.

Extensions to SmartGridSolve client library are described in section 4.4. They only add new functions, but do not modify other existing library functions or data

structures used by them. However, extended functions use internal representation of data structures used by SmartGridSolve to arrange server-to-server communication and private RPC calls to retrieve information needed for collective mapping.

By isolating these private calls and private data access into a module, writing support for other GridRPC middleware and client libraries can be made more simple.

6.1.4 Extending analyser to support more programming languages

The current implementation uses Clang compiler suite's AST building facilities to analyse programs written in C programming language. It's relatively easy to extend it to support C++ and Objective C, but it requires writing a different parser to support other programming languages outside of C family.

Static code analyser can be further generalised and extended by making programming language support modular. This will allow to use existing language parsers or write new ones for creating language modules.

Another interesting approach would be to extract algorithm workflow from LLVM IR, an intermediate code representation

Another interesting approach would be to use LLVM IR (internal representation) for code analysis to extract algorithm workflow. This representation is a kind of SSA (static single assignment) form, an internal language-independent algorithm representation used by LLVM compiler construction toolkit.

Although this approach is more complex than analysing AST, it has several advantages.

- Independence of the programming language. Any language having compiler implemented using LLVM can be used for collective mapping of GridRPC calls.

- It's much easier to analyse flow control and find assignments of variables. Hence, it's much easier to find dependencies between GridRPC calls in more general way.

Appendix A

Extended API reference

The following directives are provided as extension of public SmartGridRPC API to be used for static code analysis approach for collective mapping.

These directives do nothing at runtime. Their sole purpose is to pass information to static code analyser when it extracts algorithm workflow during compile time.

Extended API is not limited to these directives. There are two functions calls to which are inserted into files modified by `calltree` program and data structures used by these functions, but these functions and data structures are not considered to be part of public interface.

A.1 grpc_map_static()

A.1.1 Synopsis

```
grpc_map_static(char *name) { /* STATEMENT... */ }
```

A.1.2 Description

The `grpc_map_static()` directive marks a block following it for collective mapping.

Algorithm workflow is extracted by `calltree` program from this block, and additional static data definitions and code are inserted into modified source file.

A.2 grpc_likely()

A.2.1 Synopsis

```
void grpc_likely(...);
```

A.2.2 Description

When placed as the first statement in a loop, this function instructs static code analyser to treat expression passed as its first argument as estimate of number of loop iterations.

Appendix B

Static code analyser and preprocessor reference

The static code analysis engine is the `calltree` program. It extracts algorithm workflow from the program and inserts extracted information into modified files.

B.1 Synopsis

`calltree` [*OPTION*]... [*FILE*]...

B.2 Options

`--help`
: Produce help message and exit.

`--print-tree`
: Output call tree extracted from input files in XML format.

`--loop-count=`***arg***
: Set assumed number of loop iterations. Default is 4. This number is used if static code analyser is unable to detect number of loop iterations, and `grpc_likely()` directive is not used to specify estimated number.

B.3 Description

The `calltree` program performs static code analysis and inserts information from extracted algorithm workflow as well as code to utilise this information into

modified source files. This code builds application performance model, performs collective task-to-server mapping and arranges server-to-server communication at runtime.

The `calltree` program performs its function in two steps.

1. Algorithm workflow is extracted from AST files specified as command line arguments and a tree structure is produced. This structure is printed to standard output if `--print-tree` option is present in command line.

   AST files must be produced from source files in C programming language by running `clang` with `-emit-ast` option on every source file of the program.

2. This tree structure is used and modified source files are produced. File names are taken from location information stored in AST files. Modified file names are produced by replacing `.c` suffix with `.grpc.c` suffix.

   Modified files should be compiled instead of original source files to produce the final program.

It's important to pass AST files for all source modules of the program as command line arguments.

Modified files are produces only for those source modules that contain `grpc_map_static()` block and GridRPC calls.

Appendix C

HydroPad source code for runtime discovery

This appendix contains 3 source code modules from HydroPad adapted for SmartGridRPC. These modules are essential for understanding how HydroPad works.

Source modules providing support functions as well as code for remote tasks performing computations are not included.

Source code was modified to fit better in listings by removing non-essential spaces, empty strings, comments and commented out code.

GridRPC API functions are not called directly in this code. Instead, the following support functions are called, which in turn call functions from GridRPC API.

`blockcall()`
: Initialises function handle with remote task name provided by the first argument and then starts blocking task using this handle.

`nonblockcall()`
: Initialises function handle with remote task name provided by the first argument and then starts non-blocking task using this handle. Session ID is saved into a field of data structure pointed to by the second argument.

`waitnonblock()`
: Waits for completion of the task. Session ID is retrieved from a field of data structure pointed to by its argument.

C.1 Source code for `main()` function

This module contains main program. It parses command line arguments, initialises data structures and then runs the parallel algorithm.

Listing C.1: HydroPad source code: `main_smart.c`

```
1  /*
2   Copyright (C) 2005 Michele Guidolin <michele.guidolin@ucd.ie>
3
4   This file is free software; as a special exception the author gives
5   unlimited permission to copy and/or distribute it, with or without
6   modifications, as long as this notice is preserved.
7
8   This program is distributed in the hope that it will be useful, but
9   WITHOUT ANY WARRANTY, to the extent permitted by law; without even the
10  implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
11 */
12
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <string.h>
16
17 #include "global.h"
18 #include "init.h"
19 #include "grpc.h"
20 #include "gs_smart_clib.h"
21
22 int main(int argc, char *argv[])
23 {
24   int cycles_index,nmap_index,length;
25   int nstepold;
26
27   /* Create global data */
28   global gb;
29
30   /* Process arguments */
31   arguments(&gb,argc,argv);
32
33   /* Set default */
34   gb.grpc  = 1;
35   gb.smart = 1;
36
37   /* Read input parameters */
38   length = strlen(gb.ifile);
39   indata(&gb.np,&gb.nx,&gb.ny,&gb.nz,&gb.box,&gb.hnow,
40          &gb.omega_dm, &gb.omega_bm,&gb.omega_v,&gb.tempnow,
41          &gb.hfrac,&gb.spectidx,&gb.sigma_8,gb.rseed,
42          gb.ifile,&length);
43
```

```
44   /* Initialize global scalar variable */
45   initscalar(&gb);
46
47   /* Alloca memory*/
48   allocagb(&gb);
49
50   /* Initialize gridrpc */
51   if(gb.grpc)
52     initgrpc(&gb);
53
54   printf("********************  Total steps %d  ********************\n\n",
55          gb.nmap*gb.cycles );
56
57   {
58     /* Initialize time value */
59     ETINIT(gb.ettotal);
60     ETINIT(gb.etinit);
61     ETINIT(gb.etinitvel);
62     ETINIT(gb.etgrafic);
63     ETINIT(gb.etinitbm);
64     ETINIT(gb.etgravshape);
65     ETINIT(gb.etbm);
66     ETINIT(gb.etshsave);
67     ETINIT(gb.etlxlylz);
68     ETINIT(gb.etexspeed);
69     ETINIT(gb.etdm);
70     ETINIT(gb.etnbody);
71     ETINIT(gb.etdensitydm);
72     ETINIT(gb.etgrav);
73     ETINIT(gb.etgravpot);
74     ETINIT(gb.etgravforce);
75     ETINIT(gb.etgravhalf);
76     ETINIT(gb.etevolve);
77
78     /* Start timing the total ammount */
79     ETSTART(gb.ettotal);
80
81     grpc_map("")
82       {
83         ETSTART(gb.etinit);
84
85         /* Initialize dark and baryonic matter*/
86         initializedmbm_nonblk(&gb);
87
88         ETSTOP(gb.etinit);
89
90         /* Initialize time parameters */
91         grpc_local()
92           {
```

```
93              inittime(&gb);
94            }
95
96          /* MAIN LOOP */
97
98          {
99            gb.nsteps=0;
100           cycles_index=gb.cycles;
101
102           for(;cycles_index>0;--cycles_index) /* Number of cycles */
103             {
104               grpc_likely(gb.cycles-1);
105               evolve_mainloop(&gb);
106             }
107         }
108       }
109
110     ETSTOP(gb.ettotal);
111
112   }
113
114   printf("\n          RUN ON : ");
115   if(gb.grpc)
116     {
117       if(gb.smart)
118         printf("SmartSolve.\n");
119       else
120         printf("GridSolve.\n");
121     }
122   else
123     {
124       printf("local computer.\n");
125     }
126
127   printf(" TOTAL EXEC TIME : %fs\n\n",ETAVG(gb.ettotal));
128   freegb(&gb);
129   closegrpc(&gb);
130   return 0;
131 }
```

C.2 Source code for algorithm initialisation

This module contains code performing initialisation step of the parallel algorithm.

Listing C.2: HydroPad source code: `initialize_smart.c`

```
1  /*
2   Copyright (C) 2005 Michele Guidolin <michele.guidolin@ucd.ie>
3
4   This file is free software; as a special exception the author gives
5   unlimited permission to copy and/or distribute it, with or without
6   modifications, as long as this notice is preserved.
7
8   This program is distributed in the hope that it will be useful, but
9   WITHOUT ANY WARRANTY, to the extent permitted by law; without even the
10  implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
11 */
12
13 #include "global.h"
14 #include "baryonic.h"
15 #include "init.h"
16 #include "dark.h"
17 #include "baryonic.h"
18 #include "grav.h"
19
20 #include <stdlib.h>
21 #include <stdio.h>
22 #include "gs_smart_clib.h"
23
24
25 void initializedmbm_nonblk(global *gb)
26 {
27
28   /*
29       Cause I'm using SmartSolve, the handle in taskvar cannot be destruct.
30       So taskvar is allocated and not freed.
31   */
32   taskvar *tinitgrav;
33
34   tinitgrav=(taskvar *)calloc(1, sizeof (taskvar));
35   tinitgrav->handle=
36     (grpc_function_handle_t *)calloc(1, sizeof(grpc_function_handle_t));
37
38     /* Init grav */
39   ETSTART(gb->etgravshape);
40   nonblockcall("initgrav",tinitgrav,
41             gb->gshape,&gb->nx,&gb->ny,&gb->nz);
42
43   /* Initialize dark matter*/
44   ETSTART(gb->etgrafic);
```

```
45      blockcall("usegrafic",
46                gb->x1, gb->x2, gb->x3, gb->v1, gb->v2, gb->v3,
47                &gb->nx,&gb->ny,&gb->nz,&gb->np,&gb->t0h0,&gb->at,&gb->vfact,
48                &gb->omega_dm,&gb->omega_bm,&gb->omega_v,&gb->hnow,
49                &gb->spectidx,&gb->sigma_8, gb->rseed);
50      ETSTOP(gb->etgrafic);
51
52      /* Initialize dark matter density */
53      ETSTART(gb->etdensitydm);
54      if(gb->tsc)
55        {
56          blockcall("densitydmtsc",
57                    gb->rhodm, gb->x1, gb->x2, gb->x3,
58                    &gb->nx,&gb->ny,&gb->nz,&gb->nparmax,&gb->amass);
59        }
60      else
61        {
62          blockcall("densitydm",
63                    gb->rhodm, gb->x1, gb->x2, gb->x3,
64                    &gb->nx,&gb->ny,&gb->nz,&gb->nparmax,&gb->amass);
65        }
66      ETSTOP(gb->etdensitydm);
67
68      ETSTART(gb->etinitbm);
69      blockcall("initbm",
70                gb->rhodm, gb->rhobm, gb->p,
71                &gb->nx,&gb->ny,&gb->nz,
72                &gb->at,&gb->pfact,&gb->omega_dm,&gb->omega_bm);
73      ETSTOP(gb->etinitbm);
74
75      grpc_local(){
76          waitnonblock(tinitgrav);
77        }
78      ETSTOP(gb->etgravshape);
79    }
```

C.3 Source code for simulation step

This module contains code performing a single simulation step of the parallel algorithm.

Listing C.3: HydroPad source code: `evolve_smart.c`

```
1  /*
2   Copyright (C) 2005 Michele Guidolin <michele.guidolin@ucd.ie>
3
4   This file is free software; as a special exception the author gives
5   unlimited permission to copy and/or distribute it, with or without
6   modifications, as long as this notice is preserved.
7
8   This program is distributed in the hope that it will be useful, but
9   WITHOUT ANY WARRANTY, to the extent permitted by law; without even the
10  implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
11 */
12
13 #include <stdio.h>
14
15 #include "global.h"
16 #include "init.h"
17 #include "dark.h"
18 #include "baryonic.h"
19 #include "grav.h"
20 #include "gs_smart_clib.h"
21
22 void evolve_mainloop(global *gb)
23 {
24   printf("*****************\n  Evolution step %d\n*****************\n\n",
25          gb->nsteps );
26   /* System evolution */
27   grpc_local()
28     {
29       timeinfo(&gb->at,&gb->atnew,
30                &gb->dat,&gb->datnew,
31                &gb->ath,&gb->dath,
32                &gb->t,&gb->told,
33                &gb->dt,&gb->dtold,
34                &gb->t0h0,&gb->hnow,&gb->nsteps);
35     }
36   evolve_nonblk(gb);
37   gb->nsteps++;          /* Increase the evolve step */
38 }
39
40 void evolve_nonblk(global *gb)
41 {
42   double ga;
43
```

```
44    /*
45       Cause I'm using SmartSolve, the handle in taskvar cannot be destruct.
46       So taskvar is allocated and not freed.
47    */
48    taskvar *tbarmatter;
49
50    tbarmatter=(taskvar *)calloc(1, sizeof (taskvar));
51    tbarmatter->handle=
52      (grpc_function_handle_t *)calloc(1, sizeof(grpc_function_handle_t));
53
54    assert(gb);
55
56    ETSTART(gb->etevolve);
57    grpc_local()
58      {
59        ga = gb->gconst/gb->at;
60      }
61
62    /* Calculate gravitational component */
63    ETSTART(gb->etgrav);
64    blockcall("fields",
65              gb->phi, gb->phiold, gb->rhodm, gb->rhobm,
66              gb->gshape, gb->gx, gb->gy, gb->gz, gb->gxold, gb->gyold, gb->gzold,
67              &gb->nx,&gb->ny,&gb->nz,&ga,&gb->dt,&gb->dtold,&gb->nsteps);
68    ETSTOP(gb->etgrav);
69
70    /* At the first step initialize the baryonic matter velocity field */
71    if(gb->nsteps==0)
72      {
73        ETSTART(gb->etinitvel);
74        blockcall("initvel",
75                  gb->phi, gb->vx, gb->vy, gb->vz,&gb->nx,&gb->ny,&gb->nz,
76                  &gb->omega_dm,&gb->omega_bm,&gb->omega_v,&gb->at,&gb->dat);
77        ETSTOP(gb->etinitvel);
78      }
79
80    /* Calculate baryonic matter component */
81    ETSTART(gb->etbm);
82    nonblockcall("barmatter",tbarmatter,
83                 gb->nes, gb->phi, gb->phiold, gb->p, gb->rhobm,
84                 gb->vx, gb->vy, gb->vz,&gb->nx,&gb->ny,&gb->nz,
85                 &gb->at,&gb->dat,&gb->ath,
86                 &gb->dath,&gb->dt,&gb->dtold,&gb->gamma,&gb->eta1,&gb->eta2,
87                 &gb->dmax,&gb->norm,&gb->nsteps,(int *)gb->lflow,&gb->bmvelmax);
88
89    /* Calculate dark matter component */
90    ETSTART(gb->etdm);
91    if(gb->tsc)
92      {
```

```
93          blockcall("darkmattertsc",
94                      gb->x1, gb->x2, gb->x3, gb->v1, gb->v2, gb->v3,
95                      gb->gx, gb->gy, gb->gz, gb->gxold, gb->gyold, gb->gzold,
96                      gb->rhodm,&gb->nx,&gb->ny,&gb->nz,&gb->nparmax,
97                      &gb->at,&gb->dat,&gb->ath,&gb->dath,&gb->dt,&gb->amass,
98                      &gb->dmvelmax);
99        }
100     else
101       {
102         blockcall("darkmatter",
103                     gb->x1, gb->x2, gb->x3, gb->v1, gb->v2, gb->v3,
104                     gb->gx, gb->gy, gb->gz, gb->gxold, gb->gyold, gb->gzold,
105                     gb->rhodm,&gb->nx,&gb->ny,&gb->nz,&gb->nparmax,
106                     &gb->at,&gb->dat,&gb->ath,&gb->dath,&gb->dt,&gb->amass,
107                     &gb->dmvelmax);
108       }
109     ETSTOP(gb->etdm);
110
111     grpc_local()
112       {
113         waitnonblock(tbarmatter);
114       }
115
116     ETSTOP(gb->etbm);
117     ETSTOP(gb->etevolve);
118
119     grpc_local()
120       {
121         /* Print execution time */
122         printet(gb);
123
124         /* Calculate new time step */
125         timestep(&gb->dmvelmax,&gb->bmvelmax,&gb->at,&gb->atnew,
126                  &gb->dat,&gb->datnew,&gb->ath,&gb->dath,
127                  &gb->t,&gb->told,&gb->dt,&gb->dtold,&gb->omega_dm,
128                  &gb->omega_bm,&gb->omega_v,&gb->t0h0,&gb->norm);
129
130         gb->bmvelmax=0;
131         gb->dmvelmax=0;
132       }
133 }
```

Appendix D

HydroPad source code for static code analysis

This appendix contains 3 source code modules from HydroPad adapted for extended SmartGridRPC API used in static code analysis approach to collective mapping. These modules are essential for understanding how HydroPad works and changes that were made to adapt the code to static code analysis.

Source modules providing support functions as well as code for remote tasks performing computations are not included.

Source code was modified to fit better in listings by removing non-essential spaces, empty strings, comments and commented out code.

GridRPC API functions are not called directly in this code. Instead, the following support functions are called, which in turn call functions from GridRPC API.

`blockcall()`
: Initialises function handle with remote task name provided by the first argument and then starts blocking task using this handle.

`nonblockcall()`
: Initialises function handle with remote task name provided by the first argument and then starts non-blocking task using this handle. Session ID is saved into a field of data structure pointed to by the second argument.

`waitnonblock()`
: Waits for completion of the task. Session ID is retrieved from a field of data structure pointed to by its argument.

D.1 Source code for `main()` function

This module contains main program. It parses command line arguments, initialises data structures and then runs the parallel algorithm.

There are the following differences with the similar module for SmartGridRPC.

- Using `grpc_map_static()` directive instead of `grpc_map()`.
- The first simulation step is treated as a special case and called outside of the main loop because `initvel` remote task is called only during this step. Number of loop iterations is decremented by one because of this special iteration moved outside of the loop.

Listing D.1: HydroPad source code: `main_static.c`

```
1  /*
2   Copyright (C) 2005 Michele Guidolin <michele.guidolin@ucd.ie>
3
4   This file is free software; as a special exception the author gives
5   unlimited permission to copy and/or distribute it, with or without
6   modifications, as long as this notice is preserved.
7
8   This program is distributed in the hope that it will be useful, but
9   WITHOUT ANY WARRANTY, to the extent permitted by law; without even the
10  implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
11 */
12
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <string.h>
16
17 #include "global.h"
18 #include "init.h"
19 #include "grpc.h"
20 #include "gs_smart_clib.h"
21
22 int main(int argc, char *argv[])
23 {
24   int cycles_index,nmap_index,length;
25   int nstepold;
26
27   /* Create global data */
28   global gb;
29
30   /* Process arguments */
31   arguments(&gb,argc,argv);
32
33   /* Set default */
```

```
34    gb.grpc   = 1;
35    gb.smart  = 1;
36
37    /* Read input parameters */
38    length = strlen(gb.ifile);
39    indata(&gb.np,&gb.nx,&gb.ny,&gb.nz,&gb.box,&gb.hnow,
40           &gb.omega_dm, &gb.omega_bm,&gb.omega_v,&gb.tempnow,
41           &gb.hfrac,&gb.spectidx,&gb.sigma_8,gb.rseed,
42           gb.ifile,&length);
43
44    /* Initialize global scalar variable */
45    initscalar(&gb);
46
47    /* Alloca memory*/
48    allocagb(&gb);
49
50    /* Initialize gridrpc */
51    if(gb.grpc)
52      initgrpc(&gb);
53
54    printf("******************** Total steps %d ********************\n\n",
55           gb.nmap*gb.cycles );
56
57    {
58      /* Initialize time value */
59      ETINIT(gb.ettotal);
60      ETINIT(gb.etinit);
61      ETINIT(gb.etinitvel);
62      ETINIT(gb.etgrafic);
63      ETINIT(gb.etinitbm);
64      ETINIT(gb.etgravshape);
65      ETINIT(gb.etbm);
66      ETINIT(gb.etshsave);
67      ETINIT(gb.etlxlylz);
68      ETINIT(gb.etexspeed);
69      ETINIT(gb.etdm);
70      ETINIT(gb.etnbody);
71      ETINIT(gb.etdensitydm);
72      ETINIT(gb.etgrav);
73      ETINIT(gb.etgravpot);
74      ETINIT(gb.etgravforce);
75      ETINIT(gb.etgravhalf);
76      ETINIT(gb.etevolve);
77
78      /* Start timing the total ammount */
79      ETSTART(gb.ettotal);
80
81      grpc_map_static("greedy_map")
82        {
```

```
83            ETSTART(gb.etinit);
84
85            /* Initialize dark and baryonic matter */
86            initializedmbm_nonblk(&gb);
87
88            ETSTOP(gb.etinit);
89
90            /* Initialize time parameters */
91            grpc_local()
92              {
93                inittime(&gb);
94              }
95
96            /* MAIN LOOP */
97
98            {
99              gb.nsteps=0;
100             cycles_index=gb.cycles-1;
101
102             evolve_mainloop_0(&gb);
103             for(;cycles_index>0;--cycles_index) /* Number of cycles */
104               {
105                 grpc_likely(gb.cycles-1);
106                 evolve_mainloop(&gb);
107               }
108           }
109         }
110
111       ETSTOP(gb.ettotal);
112
113     }
114
115     printf("\n          RUN ON : ");
116     if(gb.grpc)
117       {
118         if(gb.smart)
119           printf("SmartSolve.\n");
120         else
121           printf("GridSolve.\n");
122       }
123     else
124       {
125         printf("local computer.\n");
126       }
127
128     printf(" TOTAL EXEC TIME : %fs\n\n",ETAVG(gb.ettotal));
129     freegb(&gb);
130     closegrpc(&gb);
131     return 0;
```

```
132 | }
```

D.2 Source code for algorithm initialisation

This module contains code performing initialisation step of the parallel algorithm.

The only difference with the similar module for SmartGridRPC is hardcoded use of `densitydm` remote task variant (using another variant is commented out by `#if 0` preprocessor directive).

Listing D.2: HydroPad source code: `initialize_static.c`

```
1  /*
2   Copyright (C) 2005 Michele Guidolin <michele.guidolin@ucd.ie>
3
4   This file is free software; as a special exception the author gives
5   unlimited permission to copy and/or distribute it, with or without
6   modifications, as long as this notice is preserved.
7
8   This program is distributed in the hope that it will be useful, but
9   WITHOUT ANY WARRANTY, to the extent permitted by law; without even the
10  implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
11 */
12
13 #include "global.h"
14 #include "baryonic.h"
15 #include "init.h"
16 #include "dark.h"
17 #include "baryonic.h"
18 #include "grav.h"
19
20 #include <stdlib.h>
21 #include <stdio.h>
22 #include "gs_smart_clib.h"
23
24
25 void initializedmbm_nonblk(global *gb)
26 {
27
28   /*
29      Cause I'm using SmartSolve, the handle in taskvar cannot be destruct.
30      So taskvar is allocated and not freed.
31   */
32   taskvar *tinitgrav;
33
34   tinitgrav=(taskvar *)calloc(1, sizeof (taskvar));
35   tinitgrav->handle=
36     (grpc_function_handle_t *)calloc(1, sizeof(grpc_function_handle_t));
37
38     /* Init grav */
39   ETSTART(gb->etgravshape);
40   nonblockcall("initgrav",tinitgrav,
```

```
41                 gb->gshape,&gb->nx,&gb->ny,&gb->nz);
42
43    /* Initialize dark matter */
44    ETSTART(gb->etgrafic);
45    blockcall("usegrafic",
46                 gb->x1, gb->x2, gb->x3, gb->v1, gb->v2, gb->v3,
47                 &gb->nx,&gb->ny,&gb->nz,&gb->np,&gb->t0h0,&gb->at,&gb->vfact,
48                 &gb->omega_dm,&gb->omega_bm,&gb->omega_v,&gb->hnow,
49                 &gb->spectidx,&gb->sigma_8, gb->rseed);
50    ETSTOP(gb->etgrafic);
51
52    /* Initialize dark matter density */
53    ETSTART(gb->etdensitydm);
54  #if 0
55    if(gb->tsc)
56      {
57        blockcall("densitydmtsc",
58                     gb->rhodm, gb->x1, gb->x2, gb->x3,
59                     &gb->nx,&gb->ny,&gb->nz,&gb->nparmax,&gb->amass);
60      }
61    else
62  #endif
63      {
64        blockcall("densitydm",
65                     gb->rhodm, gb->x1, gb->x2, gb->x3,
66                     &gb->nx,&gb->ny,&gb->nz,&gb->nparmax,&gb->amass);
67      }
68    ETSTOP(gb->etdensitydm);
69
70    ETSTART(gb->etinitbm);
71    blockcall("initbm",
72                 gb->rhodm, gb->rhobm, gb->p,
73                 &gb->nx,&gb->ny,&gb->nz,
74                 &gb->at,&gb->pfact,&gb->omega_dm,&gb->omega_bm);
75    ETSTOP(gb->etinitbm);
76
77    grpc_local(){
78        waitnonblock(tinitgrav);
79      }
80    ETSTOP(gb->etgravshape);
81  }
```

D.3 Source code for simulation step

This module contains code performing a single simulation step of the parallel algorithm.

There are the following differences with the similar module for SmartGridRPC.

- The first simulation step made a special case because `initvel` remote task is called only during this step.
- Using `darkmatter` remote task variant is hardcoded (using another variant is commented out by `#if 0` preprocessor directive).
- A variable called `ga` that is runtime parameter of the algorithm is initialised in its declaration, not assigned later in code because static code analyser doesn't recognise assignments properly yet.

Listing D.3: HydroPad source code: `evolve_static.c`

```
1  /*
2   Copyright (C) 2005 Michele Guidolin <michele.guidolin@ucd.ie>
3
4   This file is free software; as a special exception the author gives
5   unlimited permission to copy and/or distribute it, with or without
6   modifications, as long as this notice is preserved.
7
8   This program is distributed in the hope that it will be useful, but
9   WITHOUT ANY WARRANTY, to the extent permitted by law; without even the
10  implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
11 */
12
13 #include <stdio.h>
14
15 #include "global.h"
16 #include "init.h"
17 #include "dark.h"
18 #include "baryonic.h"
19 #include "grav.h"
20 #include "gs_smart_clib.h"
21
22 void evolve_mainloop_0(global *gb)
23 {
24    printf("******************\n  Evolution step %d\n******************\n\n",
25           gb->nsteps );
26    /* System evolution */
27    timeinfo(&gb->at,&gb->atnew,
28             &gb->dat,&gb->datnew,
29             &gb->ath,&gb->dath,
30             &gb->t,&gb->told,
```

```
31              &gb->dt,&gb->dtold,
32              &gb->t0h0,&gb->hnow,&gb->nsteps);
33
34    evolve_nonblk_0(gb);
35    gb->nsteps++;          /* Increase the evolve step */
36  }
37
38  void evolve_mainloop(global *gb)
39  {
40    printf("*****************\n  Evolution step %d\n*****************\n\n",
41           gb->nsteps );
42    /* System evolution */
43    timeinfo(&gb->at,&gb->atnew,
44              &gb->dat,&gb->datnew,
45              &gb->ath,&gb->dath,
46              &gb->t,&gb->told,
47              &gb->dt,&gb->dtold,
48              &gb->t0h0,&gb->hnow,&gb->nsteps);
49    evolve_nonblk(gb);
50    gb->nsteps++;          /* Increase the evolve step */
51  }
52
53  void evolve_nonblk_0(global *gb)
54  {
55    double ga = gb->gconst/gb->at;
56
57    /*
58       Cause I'm using SmartSolve, the handle in taskvar cannot be destruct.
59       So taskvar is allocated and not freed.
60    */
61    taskvar *tbarmatter;
62
63    tbarmatter=(taskvar *)calloc(1, sizeof (taskvar));
64    tbarmatter->handle=
65      (grpc_function_handle_t *)calloc(1, sizeof(grpc_function_handle_t));
66
67    assert(gb);
68
69    ETSTART(gb->etevolve);
70
71    /* Calculate gravitational component */
72    ETSTART(gb->etgrav);
73    blockcall("fields",
74               gb->phi, gb->phiold, gb->rhodm, gb->rhobm,
75               gb->gshape, gb->gx, gb->gy, gb->gz, gb->gxold, gb->gyold, gb->gzold,
76               &gb->nx,&gb->ny,&gb->nz,&ga,&gb->dt,&gb->dtold,&gb->nsteps);
77    ETSTOP(gb->etgrav);
78
79    /* At the first step initialize the baryonic matter velocity field */
```

```
80     assert(gb->nsteps==0);
81       {
82         ETSTART(gb->etinitvel);
83         blockcall("initvel",
84                   gb->phi, gb->vx, gb->vy, gb->vz,&gb->nx,&gb->ny,&gb->nz,
85                   &gb->omega_dm,&gb->omega_bm,&gb->omega_v,&gb->at,&gb->dat);
86         ETSTOP(gb->etinitvel);
87       }
88
89     /* Calculate baryonic matter component */
90     ETSTART(gb->etbm);
91     nonblockcall("barmatter",tbarmatter,
92                  gb->nes, gb->phi, gb->phiold, gb->p, gb->rhobm,
93                  gb->vx, gb->vy, gb->vz,&gb->nx,&gb->ny,&gb->nz,
94                  &gb->at,&gb->dat,&gb->ath,
95                  &gb->dath,&gb->dt,&gb->dtold,&gb->gamma,&gb->eta1,&gb->eta2,
96                  &gb->dmax,&gb->norm,&gb->nsteps,(int *)gb->lflow,&gb->bmvelmax);
97
98     /* Calculate dark matter component */
99     ETSTART(gb->etdm);
100 #if 0
101     if(gb->tsc)
102       {
103         blockcall("darkmattertsc",
104                   gb->x1, gb->x2, gb->x3, gb->v1, gb->v2, gb->v3,
105                   gb->gx, gb->gy, gb->gz, gb->gxold, gb->gyold, gb->gzold,
106                   gb->rhodm,&gb->nx,&gb->ny,&gb->nz,&gb->nparmax,
107                   &gb->at,&gb->dat,&gb->ath,&gb->dath,&gb->dt,&gb->amass,
108                   &gb->dmvelmax);
109       }
110     else
111 #endif
112       {
113         blockcall("darkmatter",
114                   gb->x1, gb->x2, gb->x3, gb->v1, gb->v2, gb->v3,
115                   gb->gx, gb->gy, gb->gz, gb->gxold, gb->gyold, gb->gzold,
116                   gb->rhodm,&gb->nx,&gb->ny,&gb->nz,&gb->nparmax,
117                   &gb->at,&gb->dat,&gb->ath,&gb->dath,&gb->dt,&gb->amass,
118                   &gb->dmvelmax);
119       }
120     ETSTOP(gb->etdm);
121
122     waitnonblock(tbarmatter);
123     ETSTOP(gb->etbm);
124     ETSTOP(gb->etevolve);
125
126     /* Print execution time */
127     printet(gb);
128
```

```
129     /* Calculate new time step */
130     timestep(&gb->dmvelmax,&gb->bmvelmax,&gb->at,&gb->atnew,
131              &gb->dat,&gb->datnew,&gb->ath,&gb->dath,
132              &gb->t,&gb->told,&gb->dt,&gb->dtold,&gb->omega_dm,
133              &gb->omega_bm,&gb->omega_v,&gb->t0h0,&gb->norm);
134
135     gb->bmvelmax=0;
136     gb->dmvelmax=0;
137   }
138
139   void evolve_nonblk(global *gb)
140   {
141     double ga = gb->gconst/gb->at;
142
143     /*
144        Cause I'm using SmartSolve, the handle in taskvar cannot be destruct.
145        So taskvar is allocated and not freed.
146     */
147     taskvar *tbarmatter;
148
149     tbarmatter=(taskvar *)calloc(1, sizeof (taskvar));
150     tbarmatter->handle=
151       (grpc_function_handle_t *)calloc(1, sizeof(grpc_function_handle_t));
152
153     assert(gb);
154
155     ETSTART(gb->etevolve);
156
157     /* Calculate gravitational component */
158     ETSTART(gb->etgrav);
159     blockcall("fields",
160               gb->phi, gb->phiold, gb->rhodm, gb->rhobm,
161               gb->gshape, gb->gx, gb->gy, gb->gz, gb->gxold, gb->gyold, gb->gzold,
162               &gb->nx,&gb->ny,&gb->nz,&ga,&gb->dt,&gb->dtold,&gb->nsteps);
163
164     ETSTOP(gb->etgrav);
165
166     assert(gb->nsteps!=0);
167
168     /* Calculate baryonic matter component */
169     ETSTART(gb->etbm);
170     nonblockcall("barmatter",tbarmatter,
171                  gb->nes, gb->phi, gb->phiold, gb->p, gb->rhobm,
172                  gb->vx, gb->vy, gb->vz,&gb->nx,&gb->ny,&gb->nz,
173                  &gb->at,&gb->dat,&gb->ath,
174                  &gb->dath,&gb->dt,&gb->dtold,&gb->gamma,&gb->eta1,&gb->eta2,
175                  &gb->dmax,&gb->norm,&gb->nsteps,(int *)gb->lflow,&gb->bmvelmax);
176
177     /* Calculate dark matter component */
```

```
178     ETSTART(gb->etdm);
179
180 #if 0
181     if(gb->tsc)
182       {
183         blockcall("darkmattertsc",
184                   gb->x1, gb->x2, gb->x3, gb->v1, gb->v2, gb->v3,
185                   gb->gx, gb->gy, gb->gz, gb->gxold, gb->gyold, gb->gzold,
186                   gb->rhodm,&gb->nx,&gb->ny,&gb->nz,&gb->nparmax,
187                   &gb->at,&gb->dat,&gb->ath,&gb->dath,&gb->dt,&gb->amass,
188                   &gb->dmvelmax);
189       }
190     else
191 #endif
192       {
193         blockcall("darkmatter",
194                   gb->x1, gb->x2, gb->x3, gb->v1, gb->v2, gb->v3,
195                   gb->gx, gb->gy, gb->gz, gb->gxold, gb->gyold, gb->gzold,
196                   gb->rhodm,&gb->nx,&gb->ny,&gb->nz,&gb->nparmax,
197                   &gb->at,&gb->dat,&gb->ath,&gb->dath,&gb->dt,&gb->amass,
198                   &gb->dmvelmax);
199       }
200     ETSTOP(gb->etdm);
201
202     waitnonblock(tbarmatter);
203     ETSTOP(gb->etbm);
204     ETSTOP(gb->etevolve);
205
206     /* Print execution time */
207     printet(gb);
208
209     /* Calculate new time step */
210     timestep(&gb->dmvelmax,&gb->bmvelmax,&gb->at,&gb->atnew,
211              &gb->dat,&gb->datnew,&gb->ath,&gb->dath,
212              &gb->t,&gb->told,&gb->dt,&gb->dtold,&gb->omega_dm,
213              &gb->omega_bm,&gb->omega_v,&gb->t0h0,&gb->norm);
214
215     gb->bmvelmax=0;
216     gb->dmvelmax=0;
217 }
```

Bibliography

[1] Abdelkader Amar, Raphaël Bolze, Aurélien Bouteiller, Andréea Chis, Yves Caniou, Eddy Caron, Pushpinder-Kaur Chouhan, Gaël Le Mahec, Holly Dail, Benjamin Depardon, et al. Diet: New developments and recent results. In *Euro-Par 2006: Parallel Processing*, pages 150–170. Springer, 2007.

[2] Dmitry Arapov, Alexey Kalinov, Alexey Lastovetsky, Ilya Ledovskih, and Ted Lewis. A programming environment for heterogeneous distributed memory machines. In *Proceedings of the 6th IEEE Heterogeneous Computing Workshop (HCW'97)*, pages 32–45, Geneva, Switzerland, April 1 1997. IEEE Computer Society Press, IEEE Computer Society Press.

[3] Dorian C Arnold and Jack Dongarra. The netsolve environment: Progressing towards the seamless grid. In *Parallel Processing, 2000. Proceedings. 2000 International Workshops on*, pages 199–206. IEEE, 2000.

[4] Dorian C Arnold, Henri Casanova, and Jack Dongarra. Innovations of the NetSolve grid computing system. *Concurrency and computation: practice and experience*, 14(13-15):1457–1479, 2002.

[5] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[6] Thomas Brady. SmartGridRPC: A new RPC model for high performance grid computing and its implementation in SmartGridSolve. PhD thesis, University College Dublin, Dublin, June 2009.

[7] Thomas Brady, Eugene Konstantinov, and Alexey Lastovetsky. SmartNetSolve: High level programming system for high performance grid computing. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS 2006)*, Rhodes Island, Greece, 25-29 April 2006 2006. IEEE Computer Society, IEEE Computer Society. CD-ROM/Abstracts Proceedings.

[8] Thomas Brady, Michele Guidolin, and Alexey Lastovetsky. Experiments with SmartGridSolve: Achieving higher performance by improving the GridRPC model. In *The 9th IEEE/ACM International Conference on Grid Computing*, Tsukuba, Japan, Sep 29 - Oct 1 2008.

[9] Thomas Brady, Jack Dongarra, Michele Guidolin, Alexey Lastovetsky, and Keith Seymour. SmartGridRPC: The new RPC model for high performance grid computing. page 55, October 2009.

[10] Thomas Brady, Jack Dongarra, Michele Guidolin, Alexey Lastovetsky, and Keith Seymour. SmartGridRPC: The new RPC model for high performance grid computing. *Concurrency and Computation: Practice and Experience*, 22:2467–2487, 2010.

[11] Thomas Brady, Oleg Girko, and Alexey Lastovetsky. *Smart RPC-based computing in Grids and on Clouds*, chapter 12, pages 255–290. Wiley Series on Parallel and Distributed Systems. Wiley-Interscience, November 2013.

[12] Tracy D Braun, Howard Jay Siegel, Noah Beck, Ladislau L Bölöni, Muthucumaru Maheswaran, Albert I Reuther, James P Robertson, Mitchell D Theys, Bin Yao, Debra Hensgen, and Richard F Freund. A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems. *Journal of Parallel and Distributed computing*, 61(6):810–837, 2001.

[13] Eddy Caron and Frédéric Desprez. DIET: A scalable toolbox to build network enabled servers on the grid. *International Journal of High Performance Computing Applications*, 20(3):335–352, 2006.

[14] Eddy Caron, Philippe Combes, Sylvain Contassot-Vivier, Frédéric Desprez, Frédéric Lombard, Jean-Marc Nicod, Martin Quinson, and Frédéric Suter. A scalable approach to network enabled servers. Research report 2002-21, Laboratoire de l'Informatique du Parallélisme (LIP), May 2002. Also available as INRIA Research Report RR-4501.

[15] Henri Casanova and Jack Dongarra. NetSolve: A network-enabled server for solving computational science problems. *International Journal of High Performance Computing Applications*, 11(3):212–223, 1997.

[16] Henri Casanova, MyungHo Kim, James S Plank, and Jack J Dongarra. Adaptive scheduling for task farming with grid middleware. *International Journal of High Performance Computing Applications*, 13(3):231–240, 1999.

[17] Phillip Colella and Paul R Woodward. The piecewise parabolic method (PPM) for gas-dynamical simulations. *Journal of computational physics*, 54(1):174–201, 1984.

[18] Douglas Crockford. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627 (Informational), July 2006. URL `http://www.ietf.org/rfc/rfc4627.txt`. Obsoleted by RFCs 7158, 7159.

[19] Bruno Del-Fabbro, David Laiymani, J Nicod, and Laurent Philippe. Data management in grid applications providers. In *Distributed Frameworks for Multimedia Applications, 2005. DFMA'05. First International Conference on*, pages 315–322. IEEE, 2005.

[20] Frédéric Desprez and Emmanuel Jeannot. Improving the GridRPC model with data persistence and redistribution. In *Parallel and Distributed Computing, 2004. Third International Symposium on/Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks, 2004. Third International Workshop on*, pages 193–200. IEEE, 2004.

[21] Troy Bryan Downing. *Java RMI: remote method invocation*. IDG Books Worldwide, Inc., 1998.

[22] Ian Foster and Carl Kessleman. Globus: A toolkit-based grid architecture. In Ian Foster and Carl Kessleman, editors, *The grid: blueprint for a new computing infrastructure*, pages 259–278. 1999.

[23] Oleg Girko and Alexey Lastovetsky. Using static code analysis to improve performance of GridRPC applications. In *9th High-Performance Grid and Cloud Computing Workshop (HPGC 2012)*, Shanghai, China, May 21, 2012 2012. IEEE Computer Society, IEEE Computer Society.

[24] Object Management Group. *The Common Object Request Broker (CORBA): Architecture and Specification*. Object Management Group, 1995.

[25] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik Frystyk Nielsen, Anish Karmarkar, and Yves Lafon. SOAP version 1.2. *W3C recommendation*, 24, 2003.

[26] Michele Guidolin and Alexey Lastovetsky. Grid-enabled Hydropad: a scientific application for benchmarking GridRPC-based programming systems. In *The 23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 25–29 2009.

[27] Michele Guidolin, Thomas Brady, and Alexey Lastovetsky. How algorithm definition language (ADL) improves the performance of SmartGridSolve applications. In *The 7th High-Performance Grid Computing Workshop*, Atlanta, USA, Apr 19 2010.

[28] Roger W Hockney and James W Eastwood. *Computer simulation using particles*. CRC Press, 2010.

[29] Alexey Lastovetsky. mpC: A multi-paradigm programming language for massively parallel computers. *ACM SIGPLAN Notices*, 31:13–20, 1996.

[30] Alexey Lastovetsky and Ravi Reddy. HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. *Journal of Parallel and Distributed Computing*, 66:197–220, 2006.

[31] Alexey Lastovetsky, Xin Zuo, and Peng Zhao. A non-intrusive and incremental approach to enabling direct communications in RPC-based grid programming systems. In *Computational Science - ICCS 2006: 6th International Conference, Reading, UK, May 28-31, 2006, Proceedings, Part III*, volume 3993 of *Lecture Notes in Computer Science*, pages 1008–1011. Springer Berlin / Heidelberg, Springer Berlin / Heidelberg, 2006. ISBN 3-540-34383-0/0302-9743.

[32] Alexey Lastovetsky, Xin Zuo, and Peng Zhao. A non-intrusive and incremental approach to enabling direct communications in RPC-based grid programming systems. page 15, 2006.

[33] Chris Lattner. Introduction to the llvm compiler system. In *XII International Workshop on Advanced Computing and Analysis Techniques in Physics Research*, Erice, Sicily, Italy, 2008.

[34] Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

[35] Simon St Laurent, Joe Johnston, Edd Dumbill, and Dave Winer. *Programming web services with XML-RPC*. " O'Reilly Media, Inc.", 2001.

[36] Craig Lee and Domenico Talia. Grid programming models: Current tools, issues and directions. *Grid Computing: Making the Global Infrastructure a Reality*, 21:555–578, 2003.

[37] Robert Love. Get on the D-BUS. *Linux Journal*, 2005(130):3, 2005.

[38] Sun Microsystems. RPC: Remote Procedure Call Protocol specification: Version 2. RFC 1057 (Informational), June 1988. URL `http://www.ietf.org/rfc/rfc1057.txt`.

[39] Johan Montagnat, Benjamin Isnard, Tristan Glatard, Ketan Maheshwari, and Mireille Blay Fornarino. A data-driven workflow language for grids based on array programming principles. In *Proceedings of the 4th Workshop on Workflows in Support of Large-Scale Science*, page 7. ACM, 2009.

[40] Bill Nowicki. NFS: Network File System Protocol specification. RFC 1094 (Informational), March 1989. URL `http://www.ietf.org/rfc/rfc1094.txt`.

[41] James S Plank, Micah Beck, Wael R Elwasif, Terence Moore, Martin Swany, and Rich Wolski. The internet backplane protocol: Storage in the network. In *In Proceedings of the Network Storage Symposium*. Citeseer, 1999.

[42] Rick Ramsey. *All about administering NIS+*. Prentice-Hall, Inc., 1994.

[43] Dongsu Ryu, Jeremiah P Ostriker, Hyesung Kang, and Renyue Cen. A cosmological hydrodynamic code based on the total variation diminishing scheme. *The Astrophysical Journal*, 414:1–19, 1993.

[44] Mitsuhisa Sato, Hidemoto Nakada, Satoshi Sekiguchi, Satoshi Matsuoka, Umpei Nagashima, and Hiromitsu Takagi. Ninf: A network based information library for global world-wide computing infrastructure. In *High-Performance Computing and Networking*, pages 491–502. Springer, 1997.

[45] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of GridRPC: A remote procedure call API for grid computing. *Lecture notes in computer science*, pages 274–278, 2002.

[46] Yoshio Tanaka, Hidemoto Nakada, Satoshi Sekiguchi, Toyotaro Suzumura, and Satoshi Matsuoka. Ninf-G: A reference implementation of RPC-based programming middleware for grid computing. *Journal of Grid Computing*, 1(1):41–51, 2003.

[47] Yusuke Tanimura, Hidemoto Nakada, Yoshio Tanaka, and Satoshi Sekiguchi. Design and implementation of distributed task sequencing on gridrpc. In *Computer and Information Technology, 2006. CIT'06. The Sixth IEEE International Conference on*, pages 67–67. IEEE, 2006.

[48] Mei Yang, Yingtao Jiang, Ling Wang, and Yulu Yang. High performance computing architectures. *Computers & Electrical Engineering*, 35(6):815–816, 2009.

[49] Asim YarKhan, Keith Seymour, Kiran Sagi, Zhiao Shi, and Jack Dongarra. Recent developments in GridSolve. *International Journal of High Performance Computing Applications*, 20(1):131–141, 2006.

[50] Xin Zuo. Non-intrusive and incremental evolution of grid programming systems. Phd thesis, University College Dublin, Dublin, Ireland, 03/2011 2011.