A high-performance parallel radio coverage prediction tool for GRASS GIS

Lucas Benedičič, Felipe A. Cruz, Tsuyoshi Hamada, *Member, IEEE,* Peter Korošec, *Member, IEEE*

***Abstract*—We present the design and implementation of a parallel radio-coverage prediction tool for GRASS GIS. The radio-coverage prediction problem is used to analyze and introduce various patterns for parallel algorithm design within GRASS GIS. Based on the serial implementation of a similar tool, we propose a master/slave programming model for our parallel implementation. We provide an extended analysis of the results of the experiments, which are based on real data from a UMTS network currently deployed in Slovenia. According to the experimental results, which are performed on a computer cluster, the parallel radio-coverage prediction tool has very good scalability properties, meaning it is able to calculate the radio-coverage prediction of real-world networks, greatly reducing processing time and maximizing performance. Moreover, we are able to solve problem instances, which sizes are out of reach of the serial implementation.**

***Index Terms*—Mobile networks, GSM, UMTS, LTE, simulation, coverage, parallel, GRASS, GIS, MPI**

I. Introduction

More than 20 years have passed since the world's first GSM mobile call was made in Finland. Still, the coverage planning of the radio network remains a key problem that all mobile operators have to deal with. Moreover, it has proven to be a fundamental issue not only in GSM networks but also in modern standards, such as the third generation (3G) UMTS and the fourth generation (4G) LTE Advanced [18, 20, 23, 27]. In radio networks is generally the case that the radio stations are installed at fixed locations, for this reason one of the primary objectives of mobile-network planning is to efficiently use the allocated frequency band to assure that the whole of the geographic area of interest can be satisfactorily reached with the radio stations of the network. To this end, radio-coverage prediction tools are of great importance as it allows the network engineers to test different network configurations before physically implementing the changes. Nevertheless, radio-coverage prediction is a complex task due to the wide range of various combinations of hardware and configuration parameters which have to be analyzed in the context of different environments. The complexity of the problem means that radio-coverage prediction can be a computationally-intensive and time-consuming task, hence the importance of fast and accurate prediction tools.

Although different mathematical models have been proposed for radio propagation modeling, none of them excels in a network-wide scenario [20]. A combination of different models and parameters is generally needed in order to calculate radio-propagation predictions for particular environments. Moreover, since the number of deployed cells (transmitters) keeps growing with the adoption of modern standards [18], there is a clear need for a radio propagation tool that is able to cope with larger work loads in a feasible amount of time.

Despite various options of commercial tools, specialized in radio-propagation modeling, the common thread among all of the them is the restricted nature of its usage, mostly dominated by black-box implementations. This fact induces lack of adaptability, sometimes even combined with cumbersome user interfaces that are not suitable for big batch jobs, involving thousands of transmitters. Moreover, the evolution of any commercial tool is strictly bounded to its vendor, forcing the user to adapt its work-flow to it, when the opposite situation should be preferable.

To tackle the afore-mentioned issues, we present a high-performance parallel radio-prediction tool for the open source Geographic Resources Analysis Support System (GRASS). For its design, we have focused on scalability, clean design and open nature of the tool, inspired by the GRASS geographic information system (GIS). These facts make it an ideal candidate for calculating radio-predictions of real mobile networks containing thousands of transmitters. And also for the scientific research community, since our design may be used as a template for parallelization of computationally-expensive tasks within the GRASS environment.

A. Parallel computation on computer clusters

To reach high levels of performance and scalability, the presented work takes advantage of specialized hardware, e.g. a cluster of computers that is set up to share login credentials for one user and a networked file system, like NFS [21]. The key step for reaching high-performance levels is to distribute the computational load among the computing nodes that belong to the cluster.

Such clusters typically consist of several commodity PCs connected through a high-speed local network. One such system is the DEGIMA cluster [10] at the Nagasaki Advanced Computing Center of the Nagasaki University, which currently

Lucas Benedičič is with the Research and Development department, Telekom Slovenije, d.d., Cigaletova 15, SI-1000, Ljubljana, Slovenia, e-mail: lucas.benedicic@telekom.si.

Felipe A. Cruz and Tsuyoshi Hamada are with the Nagasaki Advanced Computer Center, Nagasaki University, 1-14 Bunkyo-machi, Nagasaki-city, Nagasaki, 852-8521, Japan, e-mail: fcruz@nacc.nagasaki-u.ac.jp, hamada@nacc.nagasaki-u.ac.jp.

Peter Korošec is with the Computer Systems Department, Jožef Stefan Institute, Jamova cesta 39, SI-1000, Ljubljana, Jamova cesta 39, SI-1000, Ljubljana, e-mail: peter.korosec@ijs.si.

holds the third place of the Green 500 list[1].

B. Objectives

The main goal of this work is to develop a high-performance parallel radio prediction tool (PRATO) for radio networks, which performance will allow its use in large real-world environments. Therefore, our focus is on the performance and scalability of PRATO, while other more dynamic aspects of radio networks are not considered. Among these aspects are code distributions, details of (soft) handover, and dynamics related to radio resource management.

The performance evaluation of PRATO in a distributed computing environment is a major objective of this work. Furthermore, by presenting a detailed description of the design and implementation of the parallel version of PRATO, we intend to provide guidelines on how to achieve high efficiency levels of task parallelization in GRASS GIS. Additionally, we introduce techniques to overcome several obstacles encountered during our research as well as in related work, which significantly improve the quality and performance of the presented implementation, e.g.:

- inability to use GRASS in a threaded environment,
- lowering overhead of I/O operations,
- saving simulation results asynchronously and independently from GRASS,
- improving load balancing with a new message-passing technique.

The paper is organized as follows. Section II gives a description of the radio prediction tool, including the propagation model and GRASS GIS. Section III concentrates on the design principles and implementation details of the radio propagation tool, for the serial and parallel versions. Section IV discusses the experimental results and their analysis. Finally, Section V gives an overview of relevant publications, describing how they relate to our work, before drawing some conclusions.

II. Description of the radio coverage prediction tool

PRATO is a high-performance radio-prediction tool for GSM (2G), UMTS (3G) and LTE (4G) radio networks. It is implemented as a module for the GRASS Geographical Information System (for details of GRASS see Section II-B). It can be used for planning the different phases of a new radio-network installation, as well as a support tool for maintenance activities related to network troubleshooting or upgrading.

As a reference implementation, we have used the publicly available radio coverage prediction tool, developed by Hrovat et al. [11]. The authors of this work have developed a modular radio coverage tool that performs separate calculations for radio-signal path loss and antenna radiation patterns, also taking into account different configuration parameters, such as antenna tilting, azimuth and height. The output result, saved as a raster map, is the maximum signal level over the target area, in which each point represents the received signal from the best serving cell (transmitter). This work implements some well-known radio propagation models, e.g. Okumura-Hata and COST 231, the later is explained in more detail in Section II-A. Regarding the accuracy of the predicted values, the authors report comparable results to those of a state-of-the-art commercial tool. Therefore, we use the implementation developed by [11] as the reference implementation for PRATO. Furthermore, to ensure that our implementation is completely compliant with the afore-mentioned reference, we have designed a comparison test that consists of running both the reference and PRATO with the same input parameters. The test results from PRATO and the reference implementation are identical.

A. Propagation modeling

The COST-231 Walfisch-Ikegami radio-propagation model was introduced as an extension of the well-known COST Hata model [19, 20], designed for frequencies above 2000 MHz. The suitability of this model comes from the fact that it distinguishes between line-of-sight (LOS) and non-line-of-sight (NLOS) conditions. Equation (1) describes the path loss when there is LOS between the transmitter and the receiver.

$$PL_{\text{LOS}}(d) = 42.64 + 26\log(d) + 20\log(F), \tag{1}$$

where d is the distance (in kilometers) from the transmitter to the receiver point, and F is the frequency, expressed in MHz.

On the other hand, while in NLOS conditions, the path loss is calculated as in Equation (2).

$$PL_{\text{NLOS}}(d) = L_0 + L_{\text{RTS}} + L_{\text{MSD}}, \tag{2}$$

where L_0 is the attenuation in free space, L_{RTS} represents the diffraction from roof top to street, and L_{MSD} represents the diffraction loss due to multiple obstacles.

In this work, as well as in the reference implementation in [11], the terrain profile is used for LOS determination. Besides, the wave-guide effect in streets of big cities is not taken into account, because the building data is not available. In order to compensate the missing data, we include a correction factor, based on the land usage (clutter data). This technique is also adopted by other propagation models for radio networks, like the artificial neural networks macro-cell model developed by Neskovic et al. [13]. Consequently, both Equations (1) and (2) have an extra term for signal loss due to clutter (L_{CLUT}), thus redefining the LOS and NLOS path losses as

$$PL_{\text{LOS}}(d) = 42.64 + 26\log(d) + 20\log(F) + L_{\text{CLUT}}, \tag{3}$$

and

$$PL_{\text{NLOS}}(d) = L_0 + L_{\text{RTS}} + L_{\text{MSD}} + L_{\text{CLUT}}. \tag{4}$$

[1]http://www.green500.org

B. GRASS Geographical Information System

As the target environment we have chosen the GRASS (Geographic Resources Analysis Support System)[14], which is an open source, free software (FOSS) Geographical Information System (GIS). This GIS software was originally developed at the US Army Construction Engineering Research Laboratories and is a full-featured system with a wide range of analytical, data-management, and visualization capabilities. Currently, the development of GRASS GIS is supported by a growing community of volunteer developers.

The use of GRASS GIS as an environment for PRATO presents many advantages. First, the current development of GRASS is primarily Linux-based. Since the field of high performance computing is dominated by Linux and UNIX systems, an environment with Linux support is critical for this work. Software licensing is another important consideration for choosing GRASS, since it is licensed under the GNU Public License [25]and imposes the availability of the source code. This allows us to make potential modifications to the system, thus adapting it for the parallel computation environment. Moreover, being an open system, GRASS provided us with a great deal of useful built-in functionality, capable of operating with raster and vector topological data that can be stored in an internal format or a relational database. For additional information about the GRASS, we refer the reader to the numerous guides and tutorials available online.

III. Design and implementation

A. Design of the serial version

This section describes the different functions contained in the serial version of PRATO, which is implemented as a GRASS module. Their connections and data flow are depicted in Figure 1, where the parallelograms represent input/output (I/O) operations.

Our design follows a similar internal organization as the radio planning tool developed by Hrovat et al. [11], but with some essential differences. Specifically, we have decided to avoid the modular design to prevent the overhead of I/O operations, which communicate data between the components of the modular architecture. Instead, we have chosen a monolithic design, in which all the steps for generating the radio coverage prediction are calculated inside one GRASS module. Regarding the the way results are saved, our approach employs a direct connection to an external database server, instead of the slow built-in GRASS database drivers. To explicitly avoid tight coupling with a specific database vendor, the generated output is formatted in plain text, which is simply forwarded to the database server. Any further processing is achieved by issuing a query over the database tables that contain the partial results for each of the processed transmitters.

1) Read input parameters: All input data are read in the first step ("Read input data" in Figure 1), e.g. digital elevation model, clutter data, transmitter configurations, and other service-dependent settings. Their format differs based on the data they contain, namely:

- GRASS raster files are used for the digital elevation model and clutter data, whereas

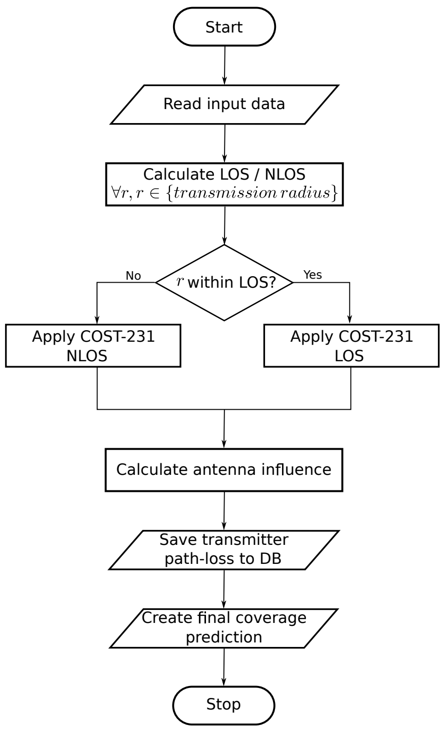
Start
Read input data
Calculate LOS / NLOS
$\forall r, r \in \{transmission\, radius\}$
No
r within LOS?
Yes
Apply COST-231 NLOS
Apply COST-231 LOS
Calculate antenna influence
Save transmitter path-loss to DB
Create final coverage prediction
Stop

Fig. 1. *Flow diagram of the serial version.*

- a text file is used for the transmitter configurations and other service-dependent options.

Since the module accepts a considerable amount of input parameters, they are read from a text-based initialization (INI) file. This is far more practical than passing them as command-line parameters, which would make them error-prune and difficult to read. Besides, the INI file may contain configuration parameters for many transmitters. The user selects which one(s) to use at run-time by passing a command-line option.

The INI file is split into two sections: common and transmitter-specific. An example INI file is given in Figure 2, which contains the configuration of one transmitter, TX_1. The common section `[Common]` contains parameters that are shared among all transmitters during the module execution. The section identifying the transmitter-specific configuration is marked `[TX_1]`. As it has been mentioned before, there may be many transmitter sections in one INI file.

2) Isotropic path-loss calculation: The first step here is to calculate which receiver points, r, are within the specified transmission radius ("transmission radius" in Figure 1). For these points, the LOS and NLOS conditions are calculated, with respect to the transmitter ("Calculate LOS/NLOS" in Figure 1). The following step consists of calculating the path loss for an isotropic source (or omni antenna). This calculation is performed by applying the COST-231 path-loss model, which was previously introduced in Section II-A, to each of the points within the transmission radius around the transmitter. Depending on whether the receiver point, r, is in LOS or NLOS, either Equation (3) or Equation (4) is respectively applied ("Apply COST-231, LOS" or "Apply COST-231, NLOS" in Figure 1).

Figure 3 shows a portion of a raster map with an example

```
[Common]
DEMMapName = dem25@PERMANENT ; name of the digital elevation model raster map
clutterMapName = clut25@PERMANENT ; name of the clutter map
receiverHeightAGL = 1.5 ; receiver’s height above ground level
frequency = 2040.0 ; transmitter frequency in MHz
radius = 20 ; calculation radius around a transmitter (in km)
antennaDirectory = ~/antenna ; directory containing antenna files
[TX_1]
cellName = TX_1 ; name of the transmitter
beamDirection = 20 ; antenna beam angle
electricalTiltAngle = 2 ;
mechanicalTiltAngle = 3 ;
heightAGL = 23.9 ; antenna height above ground level
antennaFile = 742212_2140_X_CO_M45_02T.MSI ; antenna diagram file
positionEast = 501152 ; coordinate of the transmitter
positionNorth = 142449 ; coordinate of the transmitter
power = 30.2 ; transmitter pilot power in dBm
…
```

Fig. 2. *Example of an INI file, containing input parameters for a module execution. Comments in an INI file start with the semicolon character and extend to the end of the text line.*

result of the isotropic path-loss calculation. The color scale is given in dB, indicating the signal loss from the isotropic source, located in the center. Also, the hilly terrain is clearly distinguished due to LOS and NLOS conditions from the signal source.

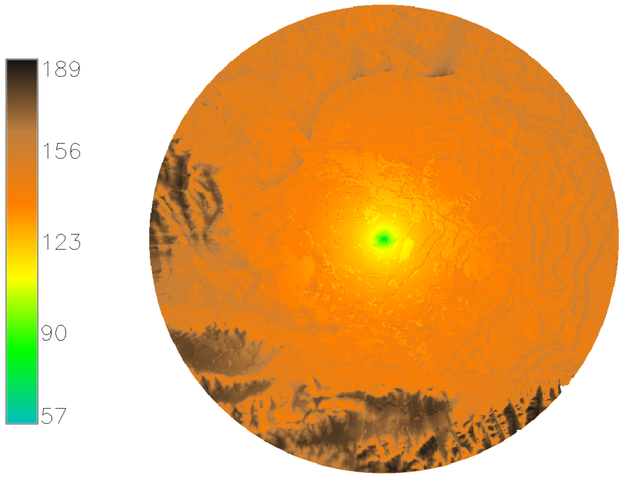

Fig. 3. *Example of raster map, showing the result of a path-loss calculation from an isotropic source.*

3) Antenna diagram influence: This step considers the antenna radiation diagram of the current transmitter and its influence over the isotropic path-loss calculation (“Calculate antenna influence” in Figure 1). Working on the in-memory results generated by the previous step, the radiation diagram of the antenna is taken into account (`antennaFile` in Figure 2), including beam direction, electrical and mechanical tilt. Figure 4 shows a portion of a raster map, where this calculation step has been applied to the results from Figure 3. Notice the distortion of the signal propagation that the antenna has introduced.

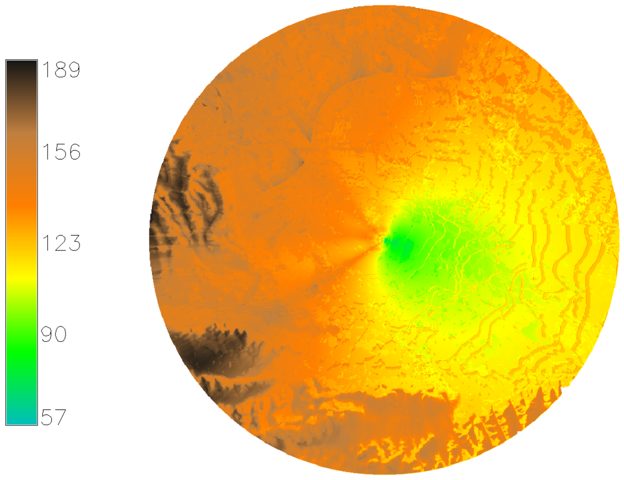

Fig. 4. *Example of raster map, showing the antenna influence over the isotropic path-loss result.*

4) Transmitter path-loss prediction: In this step, the coverage prediction of the transmitter is saved in its own database table (“Save transmitter path-loss to DB” in Figure 1), thus considerably enhancing the write performance during result dumping. This is accomplished by connecting the standard output of the developed module with the standard input of a database client. Naturally, the generated plain text should be understood by the database server itself. The following example shows a call of our coverage-prediction module, which standard output is connected to the standard input of a PostgreSQL 9.1 database client on Linux.

```
$ r.prato ini_file=./parameters.ini
tx_ini_section=TX_1 | psql -h server_name
-U usr_name db_name
```

Here, `r.prato` is the GRASS module implementing PRATO, `ini_file` specifies the parameter file to be used, and `tx_ini_section` specifies which transmitter to process from the given INI file. Many transmitters may also be given as a comma-separated list. Clearly, the values for `server_name`, `usr_name` and `db_name` should be adapted according to the database-server installation.

5) Coverage prediction: The final radio coverage prediction, containing an aggregation of the partial path-loss predictions of the involved transmitters, is created in this step (“Create final coverage prediction” in Figure 1). The received signal strength from each of the transmitters is calculated as the difference between its transmit power and path loss for the receiver’s corresponding position. This is done for each point in the target area by executing an SQL query over the tables containing the path-loss predictions of each of the processed transmitters.

Finally, the output raster is generated, using the GRASS built-in modules $r.in.xyz$ and $r.resamp.rst$, which create a raster map using the results of the above-mentioned query as input. The raster map contains the maximum received signal strength for each individual point, as shown in Figure 5. In this case, the color scale is given in dBm, indicating the received signal strength from the transmitters.

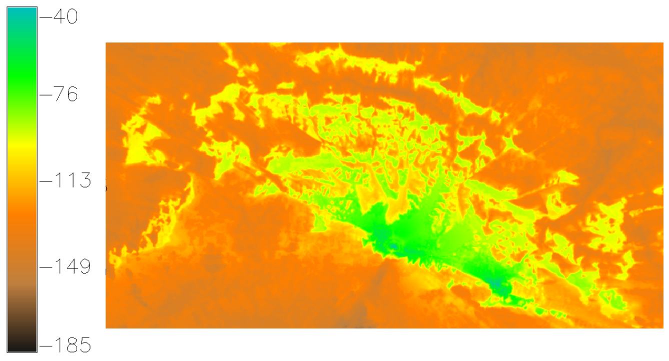

Fig. 5. *Example of raster map, displaying the final coverage prediction.*

B. *Design of the parallel version*

Keeping our focus on the performance of PRATO, we are introducing a new distributed implementation to overcome computational-time constraints that prevented the reference implementation from tackling big problem instances [11].

Some authors have already published their work on implementing parallel versions of GRASS modules for solving different time-consuming tasks Akhter et al. [1], Campos et al. [5], Osterman [17], Sorokine [24]. However, one major drawback of GRASS as a parallelization environment is that it is not thread-safe, meaning that concurrent changes to a data set have undefined behavior. To overcome this problem, we present a technique that saves the simulation results asynchronously and independently from the GRASS environment, e.g. into an external database system. This database system works also as an input source, serving data to GRASS, whether it is used to aggregate the partial results of the path-loss prediction or to visualize them. We also introduce a methodology that allows the parallel implementation to be almost completely GRASS independent. This means that a GRASS installation is needed on only one of the nodes, i.e. the master node of the target computer cluster. Also, a message-passing technique is proposed to distribute the work-load among nodes hosting the worker processes. Using this technique, computing nodes featuring more capable hardware receive more work than those with weaker configurations, thus ensuring a better utilization of the available computing resources despite hardware diversity.

1) Master process: As it has been suggested before, the parallel version of PRATO follows a master-worker model. The master process, for which the flow diagram is given in Figure 6, is the only component that should be run from within the GRASS environment. As soon as the master process starts, the input parameters are read, this step corresponds to "Read input data" in Figure 6. The reading of the input parameters is done in a similar way as in the serial version. In the next step, the master process dynamically initiates the worker processes using the available computing nodes (see the "Dynamic worker-process spawning" step in Figure 6), and takes into account the amount of transmitters for which the coverage prediction should be calculated. In other words, this means that master process never starts more worker processes than the number of transmitters to be processed. However, most often there are more transmitters than available computing nodes, therefore, the master process can assign several transmitters to each of the worker processes. For distributing the work among the worker processes, the master process then proceeds to decompose the loaded raster data into arrays of basic-data-type elements, e.g. floats or doubles, before dispatching them to the multiple worker processes (see the "Input data broadcasting" step in Figure 6). The decomposition of the data applies to the digital-elevation and the clutter data only. In the next step, the master process starts a message-driven processing loop (see the "Processing loop" step in Figure 6), which main task is to assign and distribute the calculation work, i.e. the configuration data of the different transmitters, among idle worker processes.

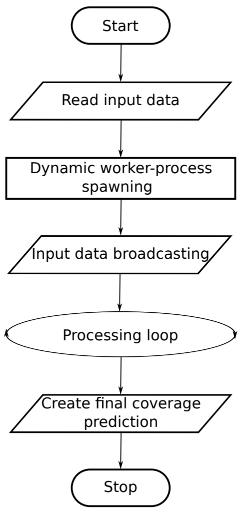
Start
Read input data
Dynamic worker-process spawning
Input data broadcasting
Processing loop
Create final coverage prediction
Stop

Fig. 6. *Flow diagram of the master process.*

The flow diagram shown in Figure 7 depicts in more detail the steps inside the "Processing loop" step of the master process. In the processing loop, the master process starts by checking which worker processes are available, so that they may calculate the radio coverage prediction for the next transmitter. It is worth pointing out that this step also serves as a stopping condition for the processing loop itself (see the "Any worker still on?" step in Figure 7). The active worker processes inform master they are ready to compute by sending an idle message (see the "Wait for idle worker" step in Figure 7). The master process then announces the idle worker process it is about to receive new data for the next calculation, and it dispatches the complete configuration of the transmitter to be processed (see the "Send keep-alive message" and "Send transmitter data" steps, respectively, in Figure 7). This is only done if there are still transmitters, for which the coverage prediction has yet to be calculated (see the "Any transmitters left?" step in Figure 7). The processing loop of the master process continues to distribute transmitter data among worker processes, which asynchronously become idle as they finish the coverage-prediction calculations for the transmitters they have been assigned by the master process. When there are no more transmitters left, all the worker processes announcing they are idle will receive a shutdown message from the master process, indicating them to stop running (see the "Send stop message" step in Figure 7). The master process will keep doing this until all worker processes have finished (see the "Any worker still on?" step in Figure 7), thus fulfilling the stopping condition of the processing loop.

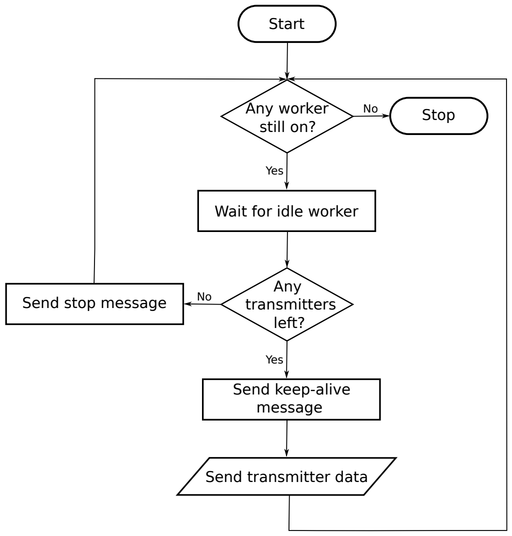

Fig. 7. *Flow diagram of the "Processing loop" step of the master process.*

Finally, the last step of the master process is devoted to creating the final output of the calculation, e.g. a raster map (see the "Create final coverage prediction" step in Figure 6). The final result for the coverage of all transmitters is an aggregation from the individual path-loss results created by each of the worker processes during the "Processing loop" phase in Figure 6, which provides the source data for the final raster map. The aggregation of the individual transmitter path-loss results is accomplished in a similar way as in the serial version.

2) Worker processes: An essential characteristic of the worker processes is that they are completely independent from GRASS, i.e. they do not have to run within the GRASS environment nor use any of the GRASS libraries to work. This aspect significantly simplifies the deployment phase to run PRATO on a computer cluster, since no GRASS installation is needed on the computing nodes hosting the worker processes.

The computations of the worker processes, for which the flow diagram is given in Figure 8, are initialized by data that are received from the master process at initialization time (see the "Receive broadcasted data" step in Figure 8). It is important to note that the received data contain the transmitter and terrain-profile information which iscommon to all the coverage-prediction calculations, therefore making each worker process capable of processing any given transmitter.

The reason for the worker processes to be independent from GRASS arises from the design of GRASS itself. Specifically, the existing GRASS library, distributed with the GRASS GIS package, is not thread-safe, because GRASS was designed as a system of small stand-alone modules and not as a library for multi-threaded programs [3]. Because of this limitation, it is not an option for a parallel implementation to create separate threads for each worker process, since this would mean worker processes should wait for each other to finish, before accessing the target data. Consequently, the scalability of such implementation would be very limited.

Because concurrent access to data within GRASS by multiple processes yields undefined behavior, i.e. it is not thread-safe, the results generated by the worker processes cannot be directly saved into the GRASS data set. One possible solution would be to save the transmitter path-loss prediction result through the master process, thus avoiding concurrent access. However, sending intermediate results back to the master process from the workers would represent a major bottleneck for the scalability of the parallel version, since the results generated by a parallel computation would have to be serially processed by the master process alone. Instead, our approach allows each of the worker processes to output its results into an external database server, following an asynchronous and decoupled design. Each of the transmitter path-loss prediction results are saved in separate tables, following a similar design as the serial version. Moreover, worker processes do this from an independent POSIX thread, which runs concurrently with the calculation of the next transmitter received from the master process. When compared to the serial version, the overlap between calculation and communication achieved by the use of an auxiliary POSIX thread completely hides the latency created by the result dumping task, and makes better use of the system resources [4].

After the broadcasted data are received by all the worker threads, each worker process proceeds to inform the master process that it is ready (in an idle state) to receive the transmitter-configuration data that defines which transmitter path-loss prediction to perform (see the "Send idle message" step in Figure 8). If the master process does not instruct to stop processing (see the "Has stop message arrived?" step in Figure 8), the worker process collects the transmitter configuration sent (see the "Receive transmitter data" step in Figure 8). However, in case a stop message is received, the worker process will wait for result-dumping threads to finish (see the "Wait for result-dump threads" step in Figure 8) before shutting down. The coverage calculation itself follows a similar design as the serial version (see the "Coverage calculation" step in Figure 8) and it is executed for the received transmitter.

As it was mentioned before, the worker process launches an independent POSIX thread to save the path-loss prediction of the target transmitter to a database table (see the "Threaded save path-loss to DB" step in Figure 8). This is done in the last step of the loop.

3) Master-worker communication: The selected message-passing technique introduced in this work might seem too elaborated, but important reasons lay behind each of the messages passed between master and worker processes. These decisions are supported by the experimental results, introduced in Section IV.

The first reason to implement the message-passing technique is to support heterogeneous computing environments.

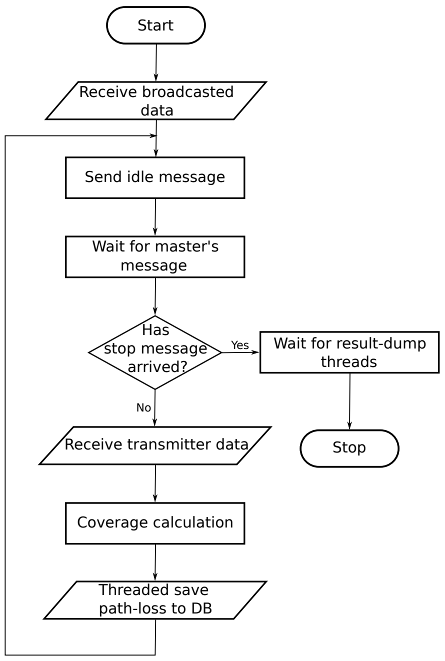
Start
Receive broadcasted data
Send idle message
Wait for master's message
Has stop message arrived?
Yes
No
Wait for result-dump threads
Stop
Receive transmitter data
Coverage calculation
Threaded save path-loss to DB

Fig. 8. *Flow diagram of one worker process.*

In particular, our approach focuses on taking full advantage of the hardware of each computing node, thus explicitly avoiding the possible bottlenecks introduced by the slowest computing node in the cluster. In other words, computing nodes that deliver better performance get more calculations assigned to the worker processes they host.

A second reason for selecting a message-passing technique is related to the flexibility for load balancing, which is of great importance on heterogeneous cluster. This can be seen in Figure 8 where the master process, before delivering the transmitter-configuration data, sends a message to the worker process indicating that it is about to receive more work. This a priori meaningless message has a key role in correctly supporting computer clusters. In general, there are many different ways a parallel program can execute, because the steps from the different processes can be interleaved in various ways and a process can make non-deterministic choices [22], which may lead to situations such as race conditions [7] and deadlocks. A deadlock occurs whenever two or more running processes are waiting for each other to finish, and thus neither ever does. To prevent the parallel version of PRATO from deadlocking, message sending and receiving should be paired, being equal number of send and receive messages on the master and worker sides [22].

Figure 9 depicts a diagram of the master-worker message passing, from which the transmitter-data transmission has been excluded for clarity. Note how each idle message sent from the worker process is paired with an answer from the master process, whether it is a keep-alive or a stop message.

C. The MPI framework

The implementation methodology adopted for the parallel version of the simulator uses the Message Passing Interface or MPI [9]. MPI is a standardized and portable library of

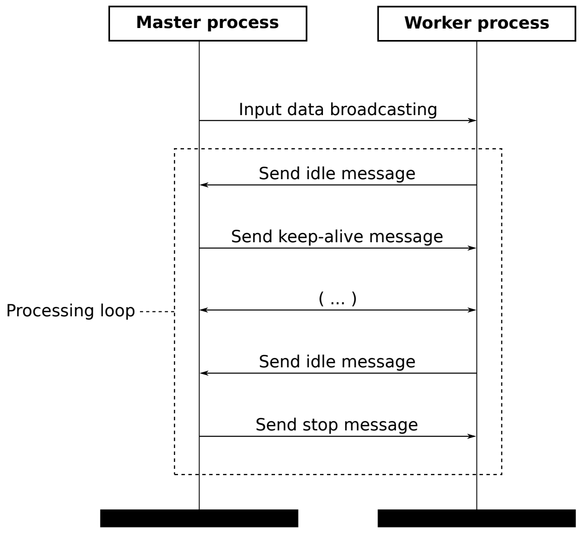

Fig. 9. *Communication diagram, showing message passing between master and one worker process.*

functions, designed to function on a wide variety of parallel computers, which syntax and semantics are defined by an open standard. The library allows implementing portable message-passing programs in Fortran and C programming languages. MPI was designed for high performance on both massively parallel machines and on workstation clusters. It has been developed by a broadly based committee of vendors, developers, and users.

IV. Simulations

This section presents the simulations and analysis of the parallel version of PRATO. Our aim is to provide an exhaustive analysis of the performance and scalability of the parallel implementation in order to determine if the objectives of this work are fulfilled. The most common usage case for PRATO is to perform a radio-coverage prediction for multiple transmitters, therefore, a straight forward parallel decomposition is to divide a given problem instance by transmitter, for which each coverage prediction is calculated by a separate worker process.

The following simulations were carried out on 34 computing nodes of the DEGIMA cluster. DEGIMA is a computer cluster located at the Nagasaki Advanced Computing Center (NACC), in the University of Nagasaki, Japan. The computing nodes are connected by a LAN, over a Gigabit Ethernet interconnect, and share a NFS partition, from which all input and intermediate files are accessed.

Each computing node of DEGIMA features one of two possible configurations, namely:

- Intel Core i5-2500T quad-core processor CPU, clocked at 2.30 GHz, with 16 GB of RAM; and
- Intel Core i7-2600K quad-core processor CPU, clocked at 3.40 GHz, also with 16 GB of RAM.

During the simulation runs, the nodes equipped with the Intel i5 CPU host the worker processes, whereas the master process and the PostgreSQL database server (version 9.1.4) run each on a different computing node, featuring an Intel i7 CPU. The database server is the only node not writing or reading data from the common NFS partition. Instead, all I/O is done on the local file system, which is mounted on a 8 GB RAM disk.

All nodes are equipped with a Linux 64-bit operating system (Fedora distribution). As the message passing implementation we use OpenMPI, version 1.6.1, which has been manually compiled with the distribution-supplied `gcc` compiler, version 4.4.4.

A. Test networks

To test the parallel performance of PRATO, we have prepared different problem instances that emulate real radio networks of different sizes. In order to create synthetic test datasets with an arbitrary number of transmitters we use the data of a group of 10 transmitters, which we randomly replicate and distribute over the whole target area. The configuration parameters of these 10 transmitters are taken from the UMTS network deployed in Slovenia by Telekom Slovenije, d.d. The path-loss predictions are calculated using the COST-231. The digital elevation model used has a resolution of 25 m^2, the same as the clutter data, containing different levels of signal loss based on the land usage. For all the points within a transmission radius of 20 km radius around each transmitter, we assume that the receiver is positioned 1.5 m above the ground, and the frequency is set to 2040 MHz. Table I summarizes these parameters, which are used for all further simulations.

TABLE I
Common simulation parameters.

Parameter	Value
Resolution (m^2)	25.0
Frequency (MHz)	2040.0
Receiver height above ground level (m)	1.5
Effective radius around transmitter (km)	20.0

B. Weak scalability

This set of simulations is meant to analyze the scalability of the parallel implementation in cases where the workload assigned to each MPI process (one MPI process per processor core) remains constant as we increase the number of processor cores and the total size of the problem, i.e. the number of transmitters deployed over the target area is directly proportional to the number of processor cores and worker processes. We do this by assigning a constant number of transmitters per core while increasing the number of cores hosting the worker processes. Consequently, we tackle larger radio-network instances as we increase the number of cores. Here we test for the following numbers of transmitters per worker/core: $\{5, 10, 20, 40, 80\}$, and we increase the number of workers/cores from 1 to 128 in powers of 2.

Problems particularly well-suited for parallel computing exhibit computational costs that are linearly dependent on the problem size. This property, also referred to as algorithmic scalability, means that proportionally increasing both the problem size and the number of cores, results in a roughly constant time to solution. Therefore, with this set of experiments, we would like to investigate how well-suited the coverage-prediction problem is for parallel computing environments.

1) Results and discussion:

The results collected after the simulations for the weak-scalability experiments are shown in Table II. All measurements express wall-clock times in seconds. Wall-clock time represents real time that elapses from the start of the master process to its end, including time that passes waiting for resources to become available. They are plotted in Figure 10, where the wall-clock time axis is expressed in base-10 logarithmic scale, whereas the axis representing the number of cores is expressed in base-2 logarithmic scale.

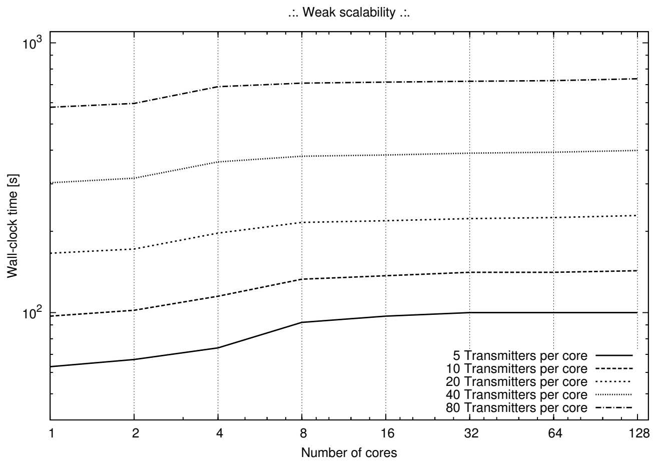

Fig. 10. *Measured wall-clock time for weak-scalability experiments as shown in Table II. Experiments performed assigned one MPI worker process per available core. The wall-clock time axis is expressed in base-10 logarithmic scale, whereas the axis representing the number of cores is expressed in base-2 logarithmic scale.*

The time measurements observed from the weak-scalability results show that the wall-clock times do not grow rapidly, especially when the number of cores is more than 8. Moreover, these times are almost constant for bigger problem instances, revealing that the achieved level of scalability gets close-to-linear as the amount of transmitters-per-core increases. Certainly, the parallel version of PRATO scales especially well when challenged with a big number of transmitters (10240 for the biggest instance) over 128 cores. This fact shows PRATO would be able to calculate the radio coverage prediction for real networks in a feasible amount of time, since many operational radio networks have already deployed a comparable number of transmitters, e.g. the 3G network within the Greater London Authority area, in the UK [15].

Not being able to achieve perfect weak scalability is due to a number of factors, namely:

- the overhead time of the serial sections of the master process grow proportionally with the number of cores, although the total contribution of this overhead remains low for large problem sizes;
- communication overhead grows linearly with the number of cores used.

To confirm these arguments, we analyze the times of each of the steps taken by the master process relative to the total

TABLE II
Wall-clock times (in seconds) of the simulation results for weak scalability.

	Number of cores							
Transmitters per core	1	2	4	8	16	32	64	128
5	92	99	118	122	123	124	125	126
10	140	152	171	175	177	179	180	182
20	244	260	278	282	284	285	287	290
40	451	470	491	497	500	502	504	509
80	865	892	920	925	928	931	937	948

processing time. To this end, we have created plots for three problem instances 5, 20 and 80 transmitters per core, which are shown in Figures 11, 12 and 13, respectively. The relative-processing-time plots follow the formula

$$RT = \frac{t_{\text{rd}} + t_{\text{ps}} + t_{\text{db}} + t_{\text{pl}} + t_{\text{cp}}}{t_{\text{total}}}, \qquad (5)$$

where t_{rd} is the “Read input data” wall-clock time, t_{ps} is the wall-clock time of the “Dynamic worker-process spawning” step, t_{db} is the wall-clock time of the “Input data broadcasting” step, t_{pl} is the wall-clock time of the “Processing loop” step, t_{cp} is the wall-clock time of the “Create final coverage prediction” step, and t_{total} is the total wall-clock processing time. For a reference of the different steps taking part of the master process, see Figure 6.

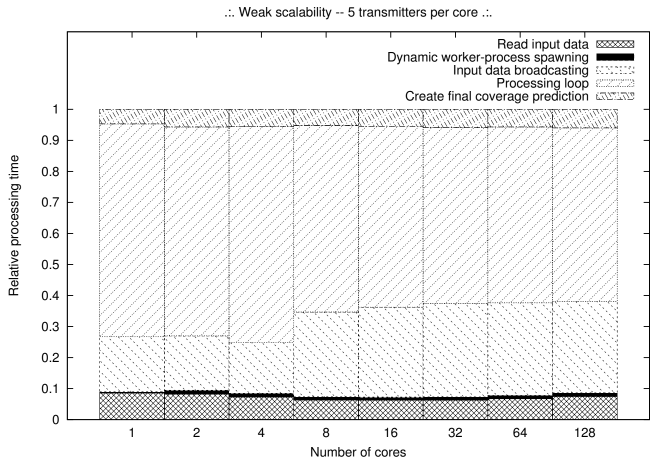

Fig. 11. *Relative times for the weak-scalability experiments for the 5 transmitters per worker/core case.*

From the relative-times plots, we see that, as we increase the number of nodes, the largest fraction of the run-time is spent on the parallel processing of transmitters, which scales notably well for larger problem instances. The plotted relative times show that there is no dependency between the relative processing times and the number of cores used, confirming the good weak-scalability properties noted before. Additionally, in all three plots we may observe a “jump” in the relative time for the “Input data broadcasting” step that takes place when comparing the result from 4 to 8 cores, i.e. from one to two computing nodes, as each node hosts “1 worker per core” or a total of “4 workers per node”. This “jump” is due to the use of network communication when more than one computing

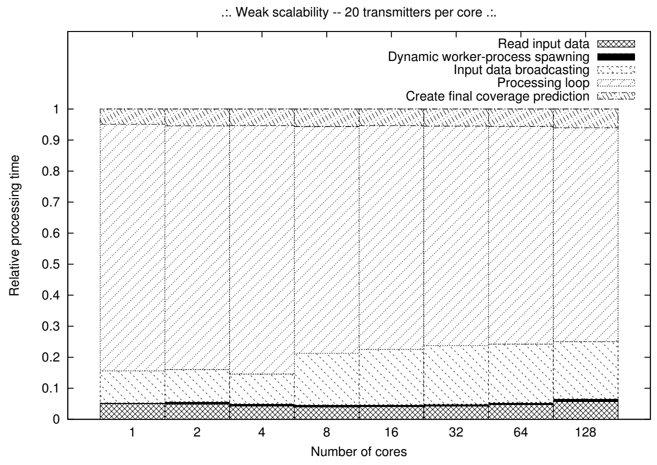

Fig. 12. *Relative times for the weak-scalability experiments for the 20 transmitters per worker/core case.*

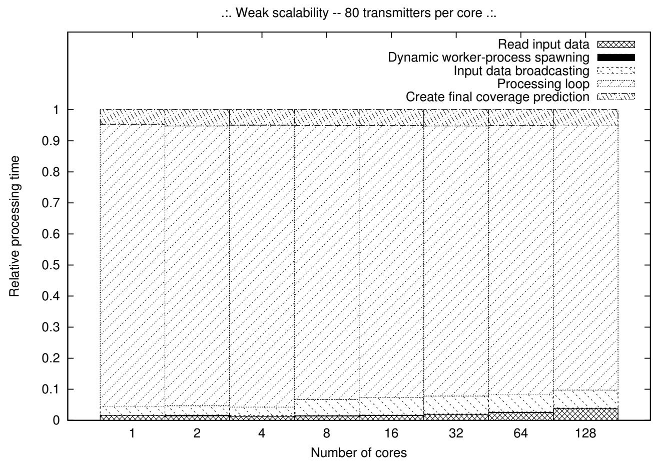

Fig. 13. *Relative times for the weak-scalability experiments for the 80 transmitters per worker/core case.*

node participates in the parallel processing. In addition, we may also conclude that the network infrastructure has not been saturated with the data-passing load, since the relative times for input-data broadcasting do not grow exponentially from 8 cores onward. Regarding the “Create final coverage prediction” step, we may see that as we increase the number of cores the relative times grow proportionally for all three problem sizes.

C. Strong scalability

This set of simulations is meant to analyze the impact of increasing the number of computing cores for a given problem size, i.e. the number of transmitters deployed over the target area does not change, while only the number of cores used is increased. Here we test for the following number of transmitters: $\{64, 128, 256, 512, 1024, 2048, 4096\}$, and we increase the number of workers/cores from 1 to 128 in powers of 2 for each problem size.

1) Results and discussion: The results of the time measurements collected after the simulations for the strong-scalability experiments are shown in Table III. All times are expressed in seconds. These wall-clock time measurements are plotted in Figure 14, where the time axis is expressed in base-10 logarithmic scale, whereas the axis representing the number of cores is expressed in base-2 logarithmic scale.

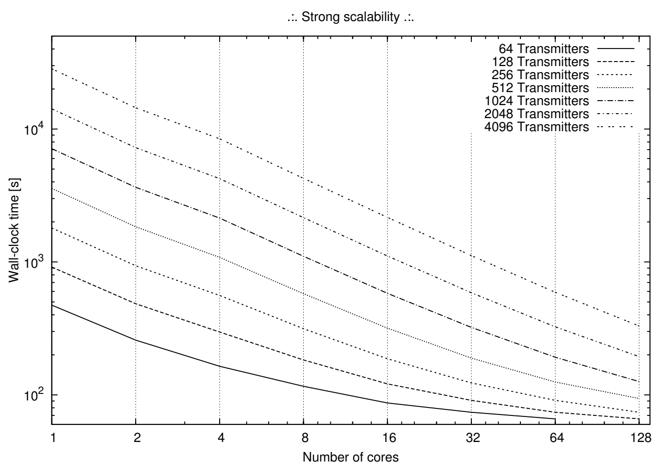

Fig. 14. *Measured wall-clock time for strong-scalability experiments as shown in Table III. Experiments performed assigned one MPI worker process per available core. The wall-clock time axis is expressed in base-10 logarithmic scale, whereas the axis representing the number of cores is expressed in base-2 logarithmic scale.*

The time measurements show that small problem sizes per core have a relatively large proportion of serial work and communication overhead. Therefore, the performance deteriorates as the number of transmitters per core approaches one. It can be observed in Figure III that as we increase the number of transmitters used to solve a given problem size, the slope of the curve generated by the progression of wall-clock times tends to a flat line, i.e. as we increase the number of transmitters there is no reduction in compute time. This idea is more clearly noted in the test with smaller problem instances, e.g. 64, 128 and 256 transmitters. In contrast, for the problems with a number of transmitters larger than 512 the relative contribution of the non-parallel steps to the wall-clock time is smaller, and a larger portion of the time is spent on computing the transmitters coverage in parallel (see Section III-B for details on the steps of PRATO algorithm). A more detailed discussion of the reasons for the loss of parallel efficiency will be presented towards the end of this section.

In order to observe how well the application scales when compared against a base case, we have also measured the performance of the parallel implementation in terms of the speed-up, which is defined as follows:

$$S(NP) = \frac{execution\, time\, for\, base\, case}{execution\, time\, for\, NP\, cores}, \tag{6}$$

where NTX is the number of transmitters processed, i.e. the problem size, and NP is the number of cores executing the worker processes. As the base case for comparisons we have chosen the parallel implementation running on only one core and decided against using the serial implementation. We consider that the serial implementation is not a good base comparison for the parallel results as: it does not reuse resources between each transmitter coverage calculation and it does not overlap I/O operations with transmitter computations. In practice, this means that several concatenated runs of the serial version would be considerably slower than the parallel but single worker implementation, because the serial implementation is not able to use all of the memory bandwidth and computing resources simultaneously. Therefore such comparison would be entirely biased towards the parallel implementation, showing super-linear scaling and speed-ups which would not be real, as the parallel version is better equipped to make use of the system resources by means of multiple POSIX threads.

Using the speed-up metric, linear scaling is achieved when the obtained speed-up is equal to the total number of processors used. However, it should be noted that perfect speed-up is almost never achieved, due to the existence of serial stages within an algorithm and communication overheads of the parallel implementation [8].

Figure 15 shows the speed-up of the parallel implementation for up to 128 cores (running one worker process per node) , and compares seven different problem sizes with 64, 128, 256, 512, 1024, 2048 and 4096 transmitters deployed over the target area. The number of transmitters used in these problem sizes are comparable to several operational radio networks that have already been deployed in England, e.g. Bedfordshire County with 69 UMTS base stations, Cheshire County with 132 UMTS base stations, Hampshire County with 227 UMTS base stations, West Midlands with 414 UMTS base stations, and Greater London Authority with 1086 UMTS base stations [15]. Moreover, consider that it is common for a single UMTS base station to host multiple transmitters.

We can see that the significant reductions in wall-clock time for large problem sizes shown in Figure 14 are directly correlated with the speed-up factors shown in Figure 15.

To study how well PRATO utilizes the available computing resources we consider the parallel efficiency of the implementation, i.e. how well the parallel implementation makes use of the available processor cores. The definition of parallel efficiency is as follows:

$$E(NP) = \frac{S(NP)}{NP}, \tag{7}$$

where $S(NP)$ is the speed-up as defined in Equation (6), and NP is the number of cores executing worker processes. Figure 16 shows the parallel efficiency of the parallel implementation

TABLE III
Wall-clock times (in seconds) of the simulation results for strong scalability.

	Number of transmitters						
Number of cores	64	128	256	512	1024	2048	4096
1	714	1392	2740	5437	10830	21562	43217
2	386	734	1419	2791	5535	10996	21987
4	232	408	751	1432	2811	5549	11042
8	155	242	409	754	1441	2817	5549
16	113	156	244	414	759	1447	2821
32	92	114	159	245	414	760	1449
64	82	94	115	159	245	420	764
128	-	83	94	116	159	248	423

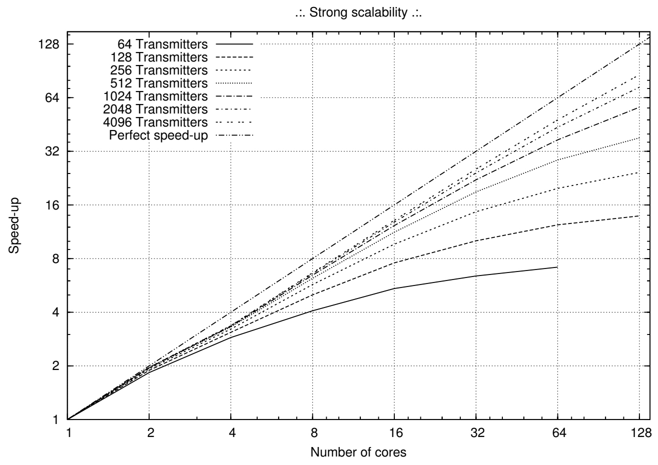

Fig. 15. *Measured speed-up for strong-scalability experiments. The speed-up axis is expressed in base-10 logarithmic scale, whereas the axis representing the number of cores is expressed in base-2 logarithmic scale.*

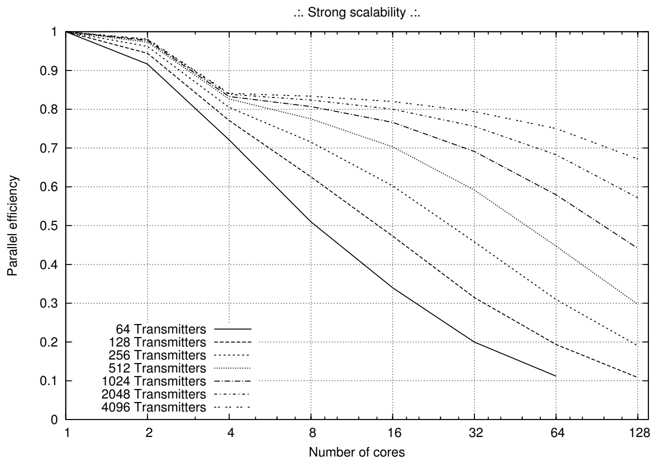

Fig. 16. *Measured parallel efficiency for strong-scalability experiments. The parallel-efficiency axis is expressed in linear scale, whereas the axis representing the number of cores is expressed in base-2 logarithmic scale.*

for different problem sizes as we increase the number of processing cores.

The ideal case for a parallel application would be to utilize all available resources, in which case the parallel efficiency would be constantly equal to one as we increase the core count. From the plot in Figure 16, we may observe that the efficiency is less than one, hence the computational resources are under utilized. In accordance to the previous analysis, the under utilization of the computing resources is more significant for the smaller problem sizes, where number of assigned transmitters per core approaches one. This is due to the increased relative influence introduced by serial and communication overheads, without which the parallel implementation would not be feasible. On the other hand, the relative time contribution of the serial and communication overheads is significantly reduced as the work-load per core increases. Unsurprisingly, these results confirm what it has previously been suggested during the weak-scaling analysis, i.e. it is not worth parallelizing small problem instances over a large number of nodes, since the time reduction due to computations that make use of the extra parallel resources is surpassed by the extra parallel initialization and communication overhead.

Similarly as in the weak scaling test, we study the relative contribution of each of the steps of the master process as we increase the number of cores used for a fixed problem size. In this case, we have created plots for four problem instances namely 64, 256, 1024, and 4096 transmitters, which are shown in Figures 17, 18, 19 and 20, respectively. The relative times shown are calculated using the formula depicted in Equation (5).

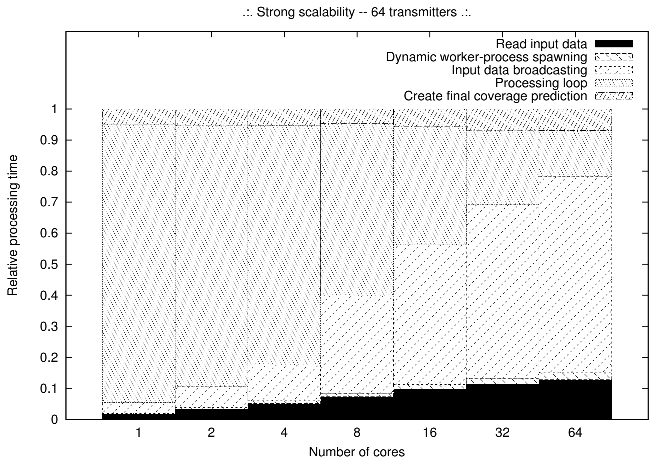

Fig. 17. *Relative times for the strong-scalability experiments, considering 64 transmitters.*

We may observe the non-parallel steps comprising "Read

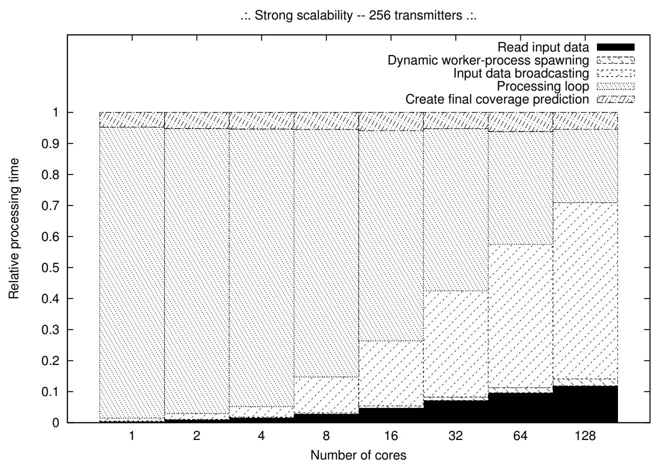

Fig. 18. *Relative times for the strong-scalability experiments, considering 256 transmitters.*

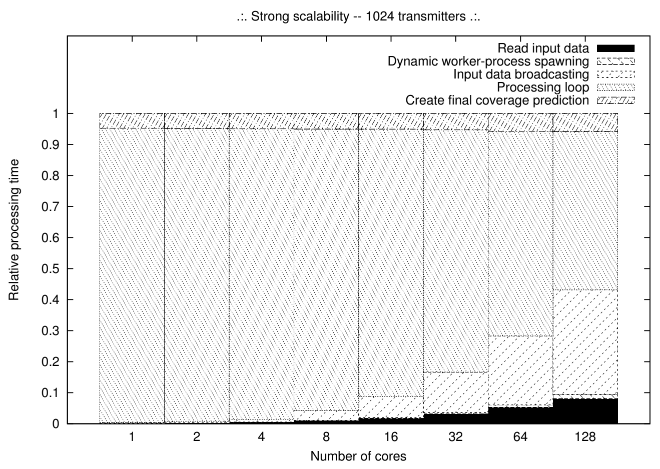

Fig. 19. *Relative times for the strong-scalability experiments, considering 1024 transmitters.*

input data", "Dynamic worker-process spawning", "Input data broadcasting" and "Final coverage prediction" contribute with a larger portion of time as we increase the number of cores, since the total wall-clock processing time decreases. Additionally, the low parallel efficiency for small problem sizes, particularly for 64 (Figure 17) and 256 transmitters (Figure 18), is validated as we see the relative small proportion of the radio-coverage calculation ("Processing loop") compared to the serial steps of the process.

D. Load balancing

In this section, we analyze the level of utilization of the computing resources available at the computing nodes hosting the worker processes. Computing-resource utilization is achieved by partitioning the computational workload and data across all processors. Efficient workload distribution strategies should be based on the processor speed, memory hierarchy and communication network [6].

The parallel implementation of PRATO performs load-balancing using point-to-point messages (see Section III-B3) between master and worker processes. When a worker process

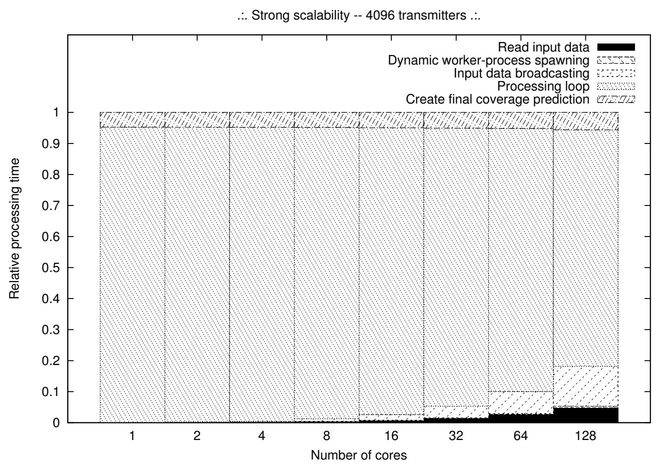

Fig. 20. *Relative times for the strong-scalability experiments, considering 4096 transmitters.*

issues an idle message (see the "Send idle message" step in Figure 9), the worker process will block until the message arrives to the master process. A similar situation occurs when the master process signals a worker back, whether to indicate it to shutdown or to continue working. Since the process-to-core mapping is one-to-one, blocking messages typically waste processor cycles on a computing node [2]. Specifically, we would like to verify the penalties that such synchronization technique has on the scalability of the parallel implementation.

We use the following metric as an indicator of the load balancing among processes:

$$LB(NP) = \frac{minimum\,execution\,time\,among\,NP\,cores}{processing\,loop\,time\,of\,\,master\,process}, \quad (8)$$

where NP is the number of cores executing worker processes. Taking the processing-loop time of the master process ensures we measure the overhead of the message passing, while the coverage prediction is being executed by the workers. This means that the time measurement is performed excluding the serial parts of the process, i.e. after the common data have been broadcasted to all worker processes ("Input data broadcasting" in Figure 6), until the beginning of the last step ("Create final coverage prediction" in Figure 6).

High performance is achieved when all cores complete their work within the same time, hence showing a load-balancing factor of one. On the other hand, lower values indicate disparity between the run times of the various worker processes sharing the parallel task, thus reflecting load imbalance.

1) Results and discussion: For this set of experiments, we have chosen the same problem sizes as for strong scalability of Section IV-C, where the coverage predictions are calculated up-to 128 cores, running on 32 computing nodes.

From the plot shown in Figure 21, it is clear that the influence of the message-passing overhead over the processing time is inversely proportional to the amount of work each worker process receives. Additionally, for the biggest problem instances (1024, 2048 and 4096 transmitters), parallel-process execution times are within 95% of a perfect load-balancing factor, and within 10% for problem sizes with 256

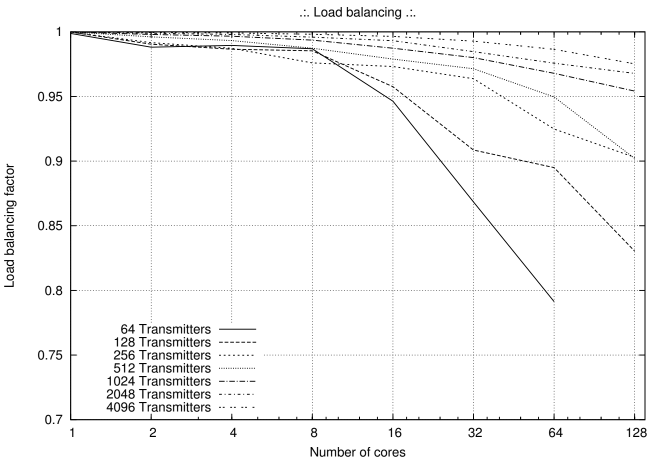

Fig. 21. *Load balancing among worker processes.*

and 512 transmitters, showing a very good performance of the dynamic task assignment, driven by our message-passing technique. For problem instances of 64 and 128 transmitters, the parallel-process times are within 80% of the perfect load balancing, showing that, as the number of transmitters per core approaches one, latencies introduced by several hardware and OS-specific factors (e.g. TurboBoost, process affinity, etc.) are influential over the total process time. Particularly, message-passing is not able to compensate these latencies as it is executed only once per worker process.

It is worth pointing out that the very good load-balancing factors shown before are not only merit of the message-passing technique. The result dumping of partial path-loss predictions, performed by the worker processes in a separate thread into an external database server, prevents data synchronization from occurring at each iteration of the parallel process, thus improving the load-balancing factors.

V. Related work

As it has been mentioned before, the reference implementation for PRATO is the work done by Hrovat et al. [11]. Although the reported results show a comparable quality to those of a professional radio-planning tool, there is an explicit concern about the performance of a specific module, *r.MaxPower*. The authors note that due to its memory inefficiency, it is not possible to consider large regions, since the afore-mentioned module runs out of memory. Although a performance comparison with this work would not be fair since it only deals with serial implementations, the parallel implementation of PRATO is not subject to the memory problem, making calculation over large regions with many transmitters perfectly possible.

A different example of a GIS-based open-source radio planning tool, called Q-Rap, has been presented in [26]. Developed by the University of Pretoria and the Meraka Institute of South Africa, the software was made publicly available in May 2010. Its design is driven as an end-user tool, with a graphical user interface, not appropriate for big batch jobs involving thousands of transmitters, or even parallel job execution. It is implemented as a plug-in for the Quantum GIS (QGIS) open source system [16].

The task-parallelization problem within the GRASS environment has been addressed by several authors in different works. Campos et al. [5] present a collection of GRASS modules for watershed analysis. Their work concentrates on different ways of slicing raster maps to take advantage of a potential MPI implementation, but there are no guidelines for work replication. Moreover, the hardware specification, on which the experiments have been run, is missing, making it very difficult to build upon this work.

Showing that GIS applications may also take advantage of GPU hardware, Osterman [17] has presented a CUDA-based module to calculate optical visibility (or line of sight - LOS) based on the digital elevation model. The experimental results report that the module performance could be up-to three size classes faster than the serial implementation.

On the field of high-performance computing, Akhter et al. [1] have presented implementation examples of a GRASS raster module, used to process vegetation indexes for satellite images, for MPI and Ninf-G environments. The main drawback with their methodology is the compulsory use of GRASS libraries in all the computing nodes that take part in the parallel calculation, making them more difficult to setup. Moreover, the authors explicitly acknowledge a limitation in the performance of their MPI implementation for big processing jobs. The restriction appears due to the computing nodes being fixed to a specific range, since the input data are equally distributed among worker processes, creating an obstacle for load balancing in heterogeneous environments. It is worth pointing out that in the parallel implementation of PRATO we specifically address this problem with our message-passing technique.

Similarly, Huang et al. [12] use the parallel inverse distance weighting interpolation algorithm as a parallel-pattern example. Although it is not explicitly noted, it can be concluded that the computing nodes make use of the GRASS environment, again making them more difficult to setup. Moreover, since the amount of work is evenly distributed among all processes (including the master one), their approach would also show decreased efficiency in heterogeneous environments.

VI. Conclusion

We have presented the design and implementation of PRATO, a parallel radio-coverage prediction tool for GRASS GIS. Extensive simulations were run in the DEGIMA computer cluster of the Nagasaki Advanced Computing Center. The results have been analyzed to determine the level of scalability of the implementation, as well as the impact of the introduced patterns for parallel algorithm design within GRASS GIS.

The conducted analysis shows that PRATO is able to calculate the radio-coverage prediction of real-world radio networks in a reduced amount of time with a high scalability level. The promising results also show the great potential of our approach to parallelize other time-consuming tasks for GRASS GIS, although this point still has to be fully demonstrated. Particularly, the gathered results suggest that our approach would be also beneficial in the area of mobile network optimization, where thousands of simulations take part of the evaluation step during an optimization process. Still, further research is needed on how this method may be exploited.

Nevertheless, as PRATO is FOSS, it can be readily modified and extended to support, for example, other propagation models and post-processing algorithms. This characteristic defines a clear advantage when compared to commercial and closed-source tools.

Acknowledgment

This project was co-financed by the European Union, through the European Social Fund. Cruz and Hamada acknowledge support from the Special Coordination Funds for Promoting Science and Technology, Ministry of Education, Culture, Sports, Science and Technology, Japan (MEXT).

References

[1] S. Akhter, Y. Chemin, and K. Aida. Porting a GRASS raster module to distributed computing Examples for MPI and Ninf-G. *OSGeo Journal*, 2(1), 2007.

[2] M. Bhandarkar, L. Kale, E. de Sturler, and J. Hoeflinger. Adaptive load balancing for MPI programs. *Computational Science-ICCS 2001*, pages 108–117, 2001.

[3] R. Blazek and L. Nardelli. The GRASS server. In *Proceedings of the Free/Libre and Open Source Software for Geoinformatics: GIS-GRASS Users Conference*, 2004.

[4] D.R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[5] I. Campos, I. Coterillo, J. Marco, A. Monteoliva, and C. Oldani. Modelling of a Watershed: A Distributed Parallel Application in a Grid Framework. *Computing and Informatics*, 27(2):285–296, 2012.

[6] D. Clarke, A. Lastovetsky, and V. Rychkov. Dynamic Load Balancing of Parallel Computational Iterative Routines on Highly Heterogeneous HPC Platforms. *Parallel Processing Letters*, 21(02):195–217, 2011.

[7] C. Clemencon, J. Fritscher, M. Meehan, and R. Rühl. An implementation of race detection and deterministic replay with MPI. *EURO-PAR'95 Parallel Processing*, pages 155–166, 1995.

[8] F.A. Cruz Villaroel. *Particle flow simulation using a parallel FMM on distributed memory systems and GPU architectures*. PhD thesis, Faculty of Science, University of Bristol, November 2010.

[9] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: portable parallel programming with the message passing interface*, volume 1. MIT press, 1999.

[10] T. Hamada and K. Nitadori. 190 TFlops astrophysical N-body simulation on a cluster of GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–9. IEEE Computer Society, 2010.

[11] A. Hrovat, I. Ozimek, A. Vilhar, T. Celcer, I. Saje, and T. Javornik. Radio coverage calculations of terrestrial wireless networks using an open-source GRASS system. *WSEAS Transactions on Communications*, 9(10):646–657, 2010.

[12] F. Huang, D. Liu, X. Tan, J. Wang, Y. Chen, and B. He. Explorations of the implementation of a parallel IDW interpolation algorithm in a Linux cluster-based parallel GIS. *Computers & Geosciences*, 37(4):426–434, 2011.

[13] A. Neskovic, N. Neskovic, and D. Paunovic. A field strength prediction model based on artificial neural networks. In *Electrotechnical Conference, 1998. MELECON 98., 9th Mediterranean*, volume 1, pages 420–424. IEEE, 1998.

[14] M. Neteler and H. Mitasova. *Open source GIS: a GRASS GIS approach*, volume 689. Kluwer Academic Pub, 2002.

[15] Ofcom. Table of base station totals, accessed October 2012. URL http://stakeholders.ofcom.org.uk/sitefinder/table-of-totals.

[16] Open Source Geospatial Foundation. Quantum GIS, accessed October 2012. URL http://www.qgis.org.

[17] A. Osterman. Implementation of the r.cuda.los module in the open source GRASS GIS by using parallel computation on the NVIDIA CUDA graphic cards. *Elektrotehniški Vestnik*, 79(1-2):19–24, 2012.

[18] A.B. Saleh, S. Redana, J. Hämäläinen, and B. Raaf. On the coverage extension and capacity enhancement of inband relay deployments in LTE-Advanced networks. *Journal of Electrical and Computer Engineering*, 2010: 4, 2010.

[19] T.K. Sarkar, Z. Ji, K. Kim, A. Medouri, and M. Salazar-Palma. A survey of various propagation models for mobile communication. *Antennas and Propagation Magazine, IEEE*, 45(3):51–82, 2003.

[20] N. Shabbir, M.T. Sadiq, H. Kashif, and R. Ullah. Comparison of Radio Propagation Models for Long Term Evolution (LTE) Network. *arXiv preprint arXiv:1110.1519*, 2011.

[21] S. Shepler, M. Eisler, D. Robinson, B. Callaghan, R. Thurlow, D. Noveck, and C. Beame. Network file system (nfs) version 4 protocol. *Network*, 2003.

[22] S. Siegel and G. Avrunin. Verification of halting properties for mpi programs using nonblocking operations. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 326–334, 2007.

[23] Iana Siomina and Di Yuan. Minimum pilot power for service coverage in WCDMA networks. *Wireless Networks*, 14(3):393–402, June 2007.

[24] A. Sorokine. Implementation of a parallel high-performance visualization technique in GRASS GIS. *Computers & geosciences*, 33(5):685–695, 2007.

[25] R. Stallman et al. GNU general public license. *Free Software Foundation, Inc., Tech. Rep*, 1991.

[26] University of Pretoria. Q-Rap: Radio Planning Tool, accessed October 2012. URL http://www.qrap.org.za/home.

[27] A. Valcarce, G. De La Roche, Á. Jüttner, D. López-Pérez, and J. Zhang. Applying FDTD to the coverage prediction of WiMAX femtocells. *EURASIP Journal on Wireless Communications and Networking*, 2009:1–13,

2009. ISSN 1687-1472.