ComputErl – Erlang-Based Framework for Many Task Computing

Michał Ptaszek[1,2] and Maciej Malawski[1]

[1] Institute of Computer Science AGH, al. Mickiewicza 30, 30-059 Kraków, Poland
[2] Erlang Solutions Ltd., London, United Kingdom
michal.ptaszek@erlang-solutions.com, malawski@agh.edu.pl

Abstract. This paper shows how Erlang programming language can be used for creating a framework for distributing and coordinating the execution of many task computing problems. The goals of the proposed solution are (1) to disperse the computation into many tasks, (2) to support multiple well-known computation models (such as master-worker, map-reduce, pipeline), (3) to exploit the advantages of Erlang for developing an efficient and scalable framework and (4) to build a system that can scale from small to large number of tasks with minimum effort. We present the results of work on designing, implementing and testing *ComputErl* framework. The preliminary experiments with benchmarks as well as real scientific applications show promising scalability on a computing cluster.

Keywords: many task computing, Erlang, grid, distributed computing, parallelism.

1 Introduction

In modern times, when the magnitude of data that needs to be processed on the daily basis is often far too large to consider it to be suitable for a single workstation, the importance of taking advantage of machines that form a cluster or computing grid is increasing. In most cases grid systems are aimed at performing the *coarse* grained computations that last for a relatively long time. The typical usage is to employ a big number of loosely coupled workstations to perform a highly specified, number-crunching and computationally intensive job.

Erlang as a functional programming language, focusing on concurrency, distribution and robustness [1], has taken a measure of a tool that allows programmers to build a highly scalable systems. However, although Erlang has never had a strong position in the computational science, it has been used several times as a highly-scalable middleware layer responsible for coordination and message transport[1,2] as well as a tool acting as a key-value storage [2].

One of main goals for this work was to prove that Erlang is capable of handling a massive-scale computation coordination. We specifically focus on *fine-grained*

[1] http://www.heroku.com
[2] http://www.facebook.com/notes.php?id=9445547199

R. Page, Z. Horváth, and V. Zsók (Eds.): TFP 2010, LNCS 6546, pp. 184–197, 2011.

computational tasks in so-called *many task* computing model [3] which is gaining importance in many petascale applications [4]. A *task* is a small part of computing job, operating independently on its own data chunk, executed in parallel with its siblings. In the system we focus on, the number of tasks within a single job is very high (thousands/millions), however the processing time of each one is short (up to few minutes).

In this paper, we present the *ComputErl* framework written in Erlang which allows researchers to perform distributed jobs computing in heterogeneous environment involving utilization of the common computation paradigms. In section 2 we describe the main goals and requirements of the *ComputErl* framework. The analysis of existing solutions for many-task computing problems is given in section 3. The main concepts of our solution are presented in section 4, with details on supported multiple computational models in section 5. Section 6 gives the example applications and tests, while section 7 concludes our discussions and outlines the future work.

2 Goals and Requirements of *ComputErl*

The main goals of our work are to investigate whether Erlang can be used to build a scalable, flexible and extensible system supporting many task computing model. These goals can be summarized as follows:

Enlarging the scale
The system should be scalable to run on wide range of machine sizes: on a standalone desktop machine, on a local cluster, on top of computing grid systems (like Grid5000[3] or PL-Grid[4] infrastructure) as well in cloud environment (like Amazon EC2[5]).

Support for different computation models
Most of the tools that are already available provide only a very limited support for commonly used computation models, such as master-slave or map-reduce. Jobs submitted to the grid systems have generally the same workflow/dataflow structure, thus the tool should support formalization of typical processing paradigms and combining them hierarchically in a customizable and configurable way to fit the application structure.

Transparency of execution
Another requirement for the tool is to facilitate the adaptation to the new environment. Switching from several workstations to the large scale system is often painful and requires learning new interfaces of the grid middleware tools, rewriting job descriptors and sometimes even altering the whole architecture.

In order to reduce this effort, the created framework should hide all the difficulties related to the system-specific part, allowing to simply acquire the

[3] `https://www.grid5000.fr`
[4] `http://www.plgrid.pl`
[5] `http://aws.amazon.com/ec2`

access to the grid resources and start the tool on them without changing the application descriptors.

Extensibility

Nevertheless, the aforementioned computational models might not be sufficient for some family of problems, so the architecture of the framework ought to be flexible and extensible to support new models. The possible extension may be to support yet another model for parallel computations (for instance sorting networks) as well as the meta extension used for expressing the workflow in the system (like loops or *if-else* blocks).

Preparing a new extension responsible for handling the desired paradigm should be limited only to providing new code units: the core of the system should not be modified.

Support for heterogeneous environment

As the system is intended to hide the underlying environment from the user, it should also be able to run on top of different hardware configurations. Since Erlang executes its applications inside of its own virtual machine, the language itself provides an abstraction layer for the framework. Moreover, taking advantage of possibility of selecting and implementing new load balancing strategies leads to better hardware utilization and general performance improvement.

Fault tolerance

Since the probability of a single node failure increases together with the size of the cluster, the framework must be resistant to the breakdowns. Crash of one of the machines must not interrupt the processing that is in progress on other workstations. Additionally, the lost fragments of jobs should be rescheduled to the different, healthy nodes.

Apart from hardware failures, system ought to also be able to handle the internal crashes. Since the number of Erlang processes is going to reach huge number, the unpredicted behavior of one of them should not disturb the others. In order to achieve the proper internal isolation, framework is built obeying standard OTP design principles [5].

3 State of the Art

Since grid research area is very active nowadays, there are several solutions that might be used for many task computing family of problems.

Swift[6] is a system created at University of Chicago that supports specification, execution and management of science and engineering workflows. The tool provides its own specification language used to express and describe operations that should be performed on the data [4]. Unfortunately, *Swift* uses *Globus Toolkit* [6] as a middleware, thus it is a tedious job to configure and run it on the local, non-grid environment.

DIANE[7] is a tool for controlling and scheduling of computations on a set of distributed worker nodes [7]. The system is not limited to the grid infrastructure

[6] `http://www.ci.uchicago.edu/swift/index.php`

[7] `http://it-proj-diane.web.cern.ch`

and may be used with local resources. Moreover, *DIANE* makes use of *Ganga* – a front-end for job definition and management, thus its adaptation to the new job submission system should not cause any major problems [8].

The last presented solution - *DiscoProject*[8], is an implementation of the *Map-Reduce* framework for distributed computing. Although the core of the system is written in *Erlang*, the *map* and *reduce* functions provided by user are implemented in *Python*. The project has a large and active community and is still under heavy development.

From all approaches, *DiscoProject* is the most similar solution to the *ComputErl* system, however it is focused only on the Map-Reduce computing model.

4 Main Concepts of *ComputErl*

The system follows the algorithmic skeleton approach as the parallel design pattern [9]. Skeleton itself describes "(...) the structure of a particular style of algorithm, in the way in which *higher order functions* represent general computational frameworks in the context of functional programming languages. The user must describe a solution to a problem as an instance of appropriate skeleton". According to the design, each skeleton is implemented and considered independently from the other.

The concept of the system is shown in Fig. 1. The general idea is to provide a possibility to transparently execute multiple jobs/tasks at the same time, distributing them on the available resources. System, as the coordinator, will be responsible for handling the load balancing and communication, assigning tasks to nodes and monitoring the processing. The implication of that fact is that several different jobs might occupy the same node and tasks belonging to the same job might be spread over the cluster and executed in parallel on different machines. As one of the major requirements for the framework is to hide the infrastructure complexity, from the user point of view there is no difference, besides the performance, if the system is running on single machine or is using a network of loosely coupled nodes.

Since *ComputErl* has been designed and implemented as a tool for users who do not necessarily have to be Erlang programmers, the system can be perceived as a black-box that should be fed with the job description files. Each computation request submitted to the system must be described by two parameters: configuration file and the input data location.

The first parameter - configuration - is an Erlang parsable file, i.e. *file:consult/1*[9] function should be able to read and interpret its contents. The configuration file should consist of a single root element specifying an entry point for the job: {*computation_type, Type, Conf*}. *Type* parameter is an atom defining the computational model used in the given execution phase. The third element of the tuple, *Conf*, is a list of arguments that will be provided as a configuration to the main

[8] `http://discoproject.org`
[9] `http://www.erlang.org/doc/man/file.html#consult-1`

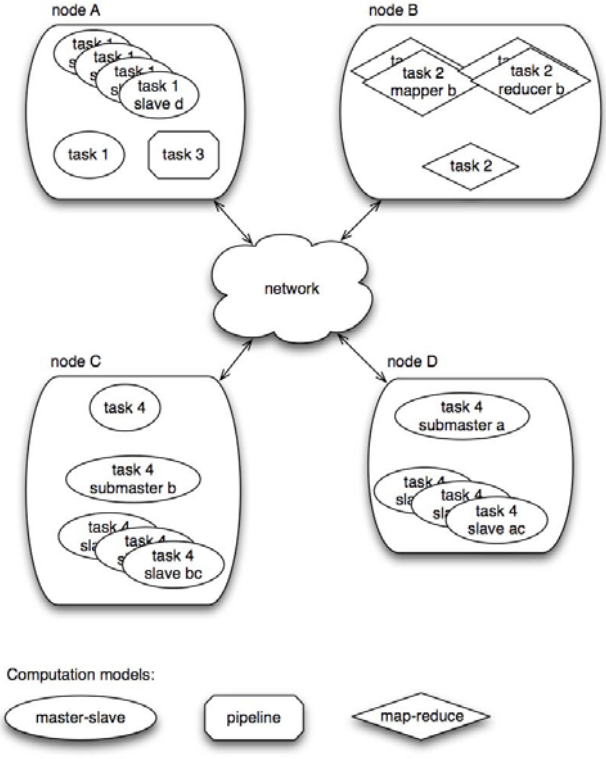


Fig. 1. Concept of the *ComputErl*: tasks belonging to multiple computation models are distributed over the computing nodes

process for the current step. The configuration parameters depend on the chosen computation model.

The programming language in which *ComputErl* has been written - Erlang - has been chosen because of the many reasons. The main advantages are:

lightweight processes – since the system is intended to be massively parallel, delegating well defined roles to the separate process instances simplifies the design and implementation. Developer does not need to bother about the operating system resources (each process consumes 331 words of memory at its startup), nor about creation time (circa 5-10 μs).

process isolation – together with the size of the system the probability of the failure grows. In order to avoid data loss or corruption, Erlang follows the *share-nothing* strategy. This means that no data (unless intentionally exposed) is shared between the processes. Crash in one of the processes does not cause the collapse of the whole system and is limited only to the small subset of processes which were linked to the erroneous one.

dynamic nature – in Erlang it is possible to inject new code without recompiling or stopping the whole system. Providing new computation models does

not require changing the core of the system. To enable new functionality, developer ought to implement a set of callback modules for the given *Type* which will take care of processing the incoming data.

transparency of execution – since process identifiers encapsulate all the information needed for communication, hence from the code point of view there is no difference if the target process resides on the same node or is evaluating its code on the remote VM. In consequence a developer might implement the logic of the application without knowledge of process location.

well-defined system structure – as the scale of the system is far beyond the size of regular desktop application, the process relationship should be well-defined and structuralized. Following OTP principles regarding the architecture it is possible to design and implement a system that is clear and easy to maintain. All processes running within *ComputErl* system are grouped under special supervisor instances which, in turn, form a hierarchical arrangement called supervision tree.

5 Supported Computation Models

The *ComputErl* framework provides a possibility to express the application structure using a dedicated job description language based on the standard Erlang syntax. Although the system is able to follow whichever model user chooses, though - as a proof of a concept - three major approaches have been implemented. These are:

- master-slave,
- map-reduce,
- pipeline.

5.1 Master-Slave

The **Master-slave** computing model defines two types of execution units [10]:

- master processes, usually one for the whole computation, take care of coordinating the slaves work: distributing the data among them and collecting the results.
- slave processes that belong to one master, execute the requested command on the given data chunk and produces the results that are sent back to the caller. The number of slaves is configurable for each job.

When the computation begins, each worker process gets the data chunk assigned from its master. As soon as the processing of that data part ends, the slave returns the result and indicate that it is ready to handle the next task. Each master has also the dedicated *result saver* process attached, which is responsible for collecting the results and saving them to disk. When master's pending chunk queue is empty and all the slave processes finished their work, the whole process terminates and stops all the workers as well.

Because the amount of data chunks assigned to the single master might be very large, in order to avoid the potential bottlenecks several improvements to this model have been made.

To eliminate a situation when one process is flooded by the results from hundreds of workers, it is possible to define the maximum number of data chunks per master. When the boundary is exceeded, the coordinator process, instead of spawning the workers, creates an additional layer of master processes, dividing the data equally among them. New masters location is chosen by the *scheduler* process running on a master node. The algorithm is repeated until the number of tasks assigned to the single master is less than a given limit. A number of masters created on the new level is also parametrized.

The second improvement is related to the location of workers against the master. Since the system has been designed to handle a great number of small tasks at the same time, the overhead caused by the data marshaling and unmarshaling might level the profit of parallel execution. To avoid such a situation the workers are spawned on the same machine as masters.

This architectural decision implies a fact that the hardware parallelism is determined by the number of sub-masters. The workers themselves are used to take advantage of the CPU cores available within the assigned node.

5.2 Map-Reduce

Map-reduce - computational paradigm originally designed by Google to support the parallel large data sets processing [11]. Users provide two functions that are operating on the inputs:

- *map* - takes an input chunk and emits a set of intermediate key/value pairs.
- *reduce* - accepts the intermediate key and a set of values connected to it. *Reduce* function should merge its input data usually into zero or one output values.

The outputs of the reduce functions are treated as the outputs of the whole computation process. In *ComputErl* there are three types of processes involved in the map-reduce phase:

- *coordinator* - a finite state machine that distributes the data among the children processes and is responsible for grouping the map outputs by the key,
- *mappers* - group of processes that execute the configured mapping function,
- *reducers* - group of processes that execute the configured reducing function.

Both number of mappers and reducers can be configured separately.

Since the communication between the job coordinator and mapper/reducer processes is not very intensive (both input data and computation results are passed as a single message), both mappers and reducers are spawned on the remote nodes in the cluster.

5.3 Pipeline

Pipeline - a meta-pattern for connecting and scheduling the subsequent phases of more complex computations. When using that model a user describes a flow of the data within the system.

The flow is basically a list of subtasks that should be executed in the given order. The processing starts from the first subtask, which is fed with the original input data. Then, the result of each phase becomes an input for the next one. Outcomes of the last step become a result of the whole pipeline flow.

In this model, each subtask might be computed using any of the available models: master-slave, map-reduce, pipeline or the one provided by the user. When using that model, only one coordinator process is created.

6 Sample Applications

In order to prove the correctness of the framework, it has been tested using four benchmarks, representing typical applications.

6.1 Sleep Benchmark

This benchmark has been used only to prove the tool is able to distribute the job parts over the workers without dropping on the performance. *Sleep* task accepts as an input an integer which is a number of seconds that worker process should sleep for. As a result the script yields a string containing hostname where it has been executed, time on that host and the assigned sleep interval. The computation model chosen to accomplish the goal was master-slave. The configuration describing such a job is presented in Fig. 2

In order to verify the correctness, benchmark has been run on up to 100 physical machines. The parallel efficiency and speedup plots are presented in Fig. 3.

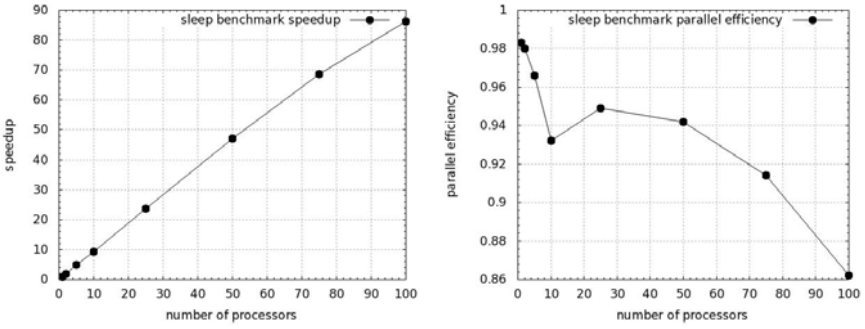
```
1 {computation_type, master_slave,
2  [{output_file, "/tmp/sleep_output.out"},
3   {script, "scripts/sleep.sh"},
4   {max_tasks_per_master, 1},
5   {slaves_no, 1}]}.
```

Fig. 2. Sleep benchmark configuration

6.2 Mandelbrot Set Generation

The second benchmark *ComputErl* framework has been tested on is a job that renders the Mandelbrot set.

The job for in this benchmark is for a given image size to produce a file consisting of pairs: X, Y, R, G, B; where X, Y are the coordinates of each point



(a) speedup (b) parallel efficiecy

Fig. 3. *sleep* benchmark speedup and parallel efficiency plots

on the picture, while R, G and B are the color coefficients in the RGB color representation model. All values are non-negative integers. Each task is defined as a computation of RGB value for a given pixel. The executable file uses *Python* programming language to implement the algorithm.

The job consists of two phases: input data generation – that is producing a list of all possible (X, Y) pairs and, in second step, computing the actual pixel color.

The chosen computation model for this job was pipeline that links two master-slave subprocesses. First master takes an image size as a parameter and creates a file with pixel coordinates (X, Y pairs) - one for each row. A reference to this file is passed to the second phase master which distributes the data inside to the linked workers. Each worker computes the color of the assigned point and returns it back to the coordinator.

Since the number of tasks grows very fast (for *MxN* pixel image its number reaches $M \cdot N$) the single master would be overloaded with the slaves result submissions and next chunk requests. Because of that there is a need to spawn at least one more masters layer in the master-slave tree.

The configuration used for those tests is presented in Fig. 4.

6.3 Distributed Grep

Next benchmark has been introduced in order to check how good the *map-reduce* paradigm implementation is. The purpose of the application is to extract and point out the lines in the huge sets of text files that match a given pattern. However, if the pattern we are looking for does not occur more than a specified number of times in the same file, that particular file should not be listed in the results.

The job consists of two Python scripts implementing map and reduce functions. The first script, *grep.py* iterates over a given file and for each line that matches the pattern given in the regular expression format, emits a pair: *(filename, line*

```
{computation_type, pipeline,
      [[{computation_type, master_slave,
        {slaves_no, 1}]},
       {output_file, "/tmp/coords.out"},
       {script, "scripts/coords.py"}],

       [{computation_type, master_slave,
       {result_delimiter, "\n\n"},

        {max_tasks_per_master, 20},
        {masters_per_level, 3},

        {slaves_no, 8}]},

      {output_file, "/tmp/mandelbrot.out"},
      {script, "scripts/mandelbrot.py -2 -1 1 1"}
    ]
]}.
```

Fig. 4. Mandelbrot benchmark configuration

number). The second one - *reduce.py* - accepts two parameters: the filename and a list of line numbers in which the pattern has been found. If the length of the list is greater than a given threshold, an output line in format *filename:line number* is produced.

As an input 3800 text files from *Project Gutenberg*[10] have been used. Total size of the input reached 1.6 GB. In the sequential approach processing time for the whole set came to 542 seconds.

The best results has been achieved when using the configuration in Fig. 5.

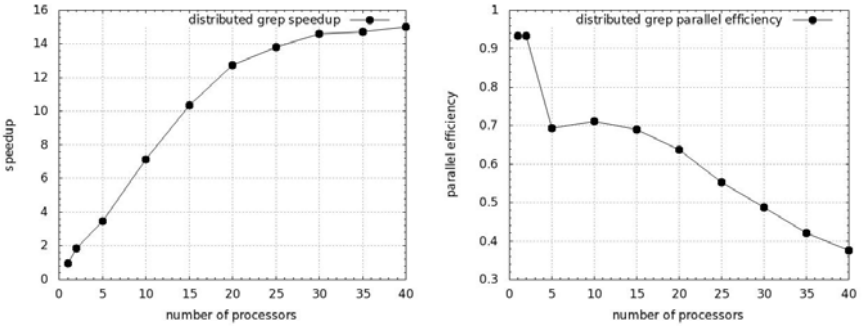
```
{computation_type, map_reduce,
 [{mappers_no, 40},
  {mapper_params,
   [{script, "scripts/grep/grep.py"}]},

  {reducers_no, 20},
  {reducer_params,
   [{script, "scripts/grep/reduce.py"}]},

  {output_file, "/tmp/dist_grep"}]
}.
```

Fig. 5. Distributed grep benchmark configuration

[10] `http://www.gutenberg.org/`



(a) speedup (b) parallel efficiency

Fig. 6. *distributed grep* benchmark speedup and parallel efficiency plots

The preliminary results of the testing are presented in Fig. 6. The tests were run on a cluster of machines (Zeus at ACC Cyfronet AGH), each node having 2 Xeon Quadcore 2.5 GHz processors with 16 GB of RAM and 10 Gb/s inter-node connection. The nodes use a shared Lustre filesystem[11].

We expect that further performance improvements may be possible by introducing a distributed filesystem which allows to exploit data locality [11]. In the current configuration all data access requires network transmission, which limits the performance of data-intensive computing.

6.4 Bioinformatics Application

The last benchmark approach was to test *ComputErl* framework on a real application from bioinformatics domain. The application predicts the active sites of proteins based on FOD model [12]. The scripts that are used during the tests has been provided by the scientists who are using them on the daily basis. Apart from the implementation the job also requires the data files containing protein structure.

The tests were executed using all possible scale-input data configuration, which gave 162792 single script runs (7752 inputs, 21 different scales). Job has been divided into 7752 tasks: a single task has been defined as a multiple profile generator for all the available scales (21 subproblems) for the given input data (protein description). Every script execution produced 21 generated profiles. Total size of output files exceeds 3.5 GB of disk space. Since the total size of the all output data produced was too big for a single process to handle, the additional data savers were attached to each master process.

In order to accomplish the goal, the simple configuration for master-slave has been used. Its listing shows that preparing *ComputErl* to execute new job is fast and easy.

The configuration used for this application can be found in Fig. 7.

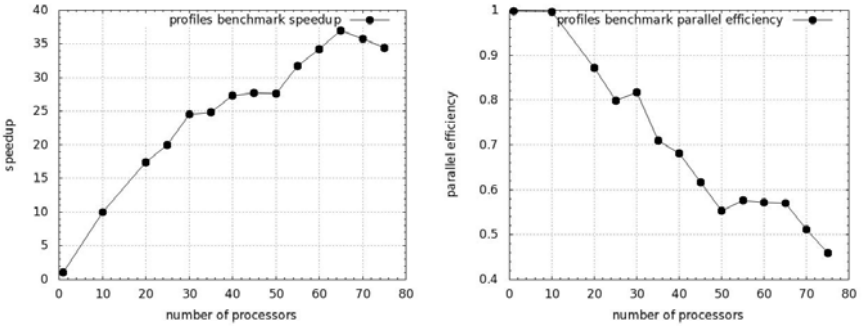
[11] `http://wiki.lustre.org/`

```
{computation_type, master_slave,
 [{script, "scripts/profiles.sh"},
  {output_file, "/tmp/profiles.out"}
 ]
}.
```

Fig. 7. Bioinformatics application configuration

Tests have been performed using the default settings for master-slave model: having 10 slaves under each master, maximum 100 tasks for each of the coordinator and spawning 10 new master processes per new level.

The graphical representation of results obtained on the same machine as the grep benchmark (Zeus cluster) is available in Figure 8.



(a) speedup (b) parallel efficiecy

Fig. 8. *profiles* benchmark speedup and parallel efficiency plots

7 Conclusions and Future Work

All the tests have been successfully executed and they have demonstrated that the framework does scale well on the large clusters of machines, and is capable of handling massive number of tasks at the same time.

Thanks to the OTP principles, controlling a highly concurrent, dynamic and parallel system turned out to be manageable: the supervision tree structure allowed building a clear and simple process' relationship graph, Erlang's built-in lightweight process support helped forget about the complexity of native operating system threads handling and synchronizing and finally the dynamic nature of language created an easy way to extend the existing solution via the user-provided callback modules. *ComputErl* turned out to be a very flexible utility, which can be run on top of potentially any of the modern computing grid systems or local clusters.

Nevertheless, *ComputErl* is still in its early days and a lot of work must be done before the tool would be regarded as a mature and competitive solution to the existing ones. Planned future work includes:

- Implementing and testing more meta-patterns, such as *if-else* branches, *for* or *while* loops. Having those structures may allow users to build more flexible and sophisticated data flow graphs for their computations.
- Load balancing strategy optimizations. One of the most interesting directions to go is to employ the existing tools to do the measurements (like *Ganglia* [13]) and basing on them, choose the least loaded target machine to deploy a given task on.
- Fault tolerance improvements. Using some kind of the persistent storage it should be possible to save (*checkpoint*) the intermediate data in between the phases of the computations. The persistence layer ought to protect the computed data from getting lost because of some unpredicted events, such as hardware failure. The framework, right after a failure detection, would be responsible for restarting the lost parts of the job on the other node, starting from the last snapshot that has been saved.

The *ComputErl* framework is available to the public as an open-source project at: `http://bitbucket.org/michalptaszek/gridagh`

Acknowledgments. The authors express their thanks to Katarzyna Prymula for providing the bioinformatics application for testing. The experiments were conducted on the Zeus cluster at ACC Cyfronet AGH. The research presented in this paper has been partially supported by the European Union within the European Regional Development Fund program no. POIG.02.03.00-00-007/08-00 as part of the PL-Grid project (`http://www.plgrid.pl`) and AGH grants 11.11.120.865 and UDA-POKL.04.01.01-00-367/08-00. We would also like to thank Jan Henry Nyström for his help on reviewing the paper.

References

1. Cesarini, F., Thompson, S.: Erlang Programming. O'Reilly Media, Sebastopol (2009)
2. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: SOSP 2007: Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles, vol. 41, pp. 205–220. ACM, New York (2007)
3. Foster, I.: Many Tasks Computing: What's in a Name? (July 2008)
4. Wilde, M., Foster, I., Iskra, K., Beckman, P., Zhang, Z., Espinosa, A., Hategan, M., Clifford, B., Raicu, I.: Parallel scripting for applications at the petascale and beyond. Computer 42(11), 50–60 (2009)
5. AB Ericsson: OTP Design Principles User's Guide (February 2010)
6. Foster, I.: Globus toolkit version 4: Software for service-oriented systems. In: Jin, H., Reed, D., Jiang, W. (eds.) NPC 2005. LNCS, vol. 3779, pp. 2–13. Springer, Heidelberg (2005), `http://dx.doi.org/10.1007/11577188_2`

7. Mościcki, J.T.: Diane - distributed analysis environment for grid-enabled simulation and analysis of physics data. In: Nuclear Science Symposium Conference Record, vol. 3, pp. 1617–1620. IEEE, Los Alamitos (2003)
8. Mościcki, J.T., Brochu, F., Ebke, J., Egede, U., Elmsheuser, J., Harrison, K., Jones, R.W.L., Lee, H.C., Liko, D., Maier, A.: Ganga: a tool for computational-task management and easy access to grid resources. Computer Physics Communications (June 2009)
9. Cole, M.: Algorithmic Skeletons: Structured Management of Parallel Computation. MIT Press, Pitman (1989)
10. Shao, G., Berman, F., Wolski, R.: Master/slave computing on the grid. In: Heterogeneous Computing Workshop, pp. 3–16 (2000)
11. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM 51(1), 107–113 (2008)
12. Bryliński, M., Prymula, K., Jurkowski, W., Kochańczyk, M., Stawowczyk, E., Konieczny, L., Roterman, I.: Prediction of functional sites based on the fuzzy oil drop model. PLoS Comput. Biol. 3(5), e94 (2007)
13. Massie, M.L., Chun, B.N., Culler, D.E.: The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. Parallel Computing 30(7) (July 2004)