Bioinformatics, 31(8), 2015, 1298–1301
doi: 10.1093/bioinformatics/btu818
Advance Access Publication Date: 12 December 2014
Applications Note

Genome analysis

Sputnik: *ad hoc* distributed computation

Gunnar Völkel[1,2,†], Ludwig Lausser[1,†], Florian Schmid[1], Johann M. Kraus[1] and Hans A. Kestler[1,3,*]

[1]Core Unit Medical Systems Biology, [2]Theoretical Computer Science, Ulm University, D-89069 Ulm, Germany and [3]Leibniz Institute for Age Research-Fritz Lipmann Institute and FSU Jena, D-07745 Jena

*To whom correspondence should be addressed.
[†]The authors wish it to be known that, in their opinion, the first two authors should be regarded as Joint First Authors.
Associate Editor: John Hancock

Received on August 26, 2014; revised on November 27, 2014; accepted on December 5, 2014

Abstract

Motivation: In bioinformatic applications, computationally demanding algorithms are often parallelized to speed up computation. Nevertheless, setting up computational environments for distributed computation is often tedious. Aim of this project were the lightweight *ad hoc* set up and fault-tolerant computation requiring only a Java runtime, no administrator rights, while utilizing all CPU cores most effectively.
Results: The Sputnik framework provides *ad hoc* distributed computation on the Java Virtual Machine which uses all supplied CPU cores fully. It provides a graphical user interface for deployment setup and a web user interface displaying the current status of current computation jobs. Neither a permanent setup nor administrator privileges are required. We demonstrate the utility of our approach on feature selection of microarray data.
Availability and implementation: The Sputnik framework is available on Github http://github.com/sysbio-bioinf/sputnik under the Eclipse Public License.
Contact: hkestler@fli-leibniz.de or hans.kestler@uni-ulm.de
Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

We introduce the Sputnik framework that manages parallelization of computational expensive algorithms to distributed inhomogeneous clusters of computing nodes. A task that is increasingly important in bioinformatic applications not only with large data sets but also with the post-processing and interpretation of the primary measurements. This framework is especially intended for parallelization of CPU-intensive computations. Sputnik accomplishes hereby two major goals: the actual fault-tolerant distributed computation and the lightweight *ad hoc* deployment of programs and data. Figure 1 illustrates the distributed computation with Sputnik. A program on the client creates a job consisting of many tasks and sends it to the server. The server schedules the tasks to the workers which execute them and send back the results. If a worker crashes, the server reschedules its assigned tasks to the other workers.

The intended usage scenario of Sputnik does not involve huge amounts of input data unlike the default scenario of applications using Hadoop (hadoop.apache.org). Also unlike the JPPF framework (www.jppf.org) Sputnik achieves a full utilization of all available CPU cores during job execution through its batch-wise task distribution to the worker. This is achieved via task scheduling strategies that assign new tasks to workers upon completion of previous tasks almost immediately. Due to its minimal requirements (a Java runtime and no administrator privileges) Sputnik can be easily deployed to heterogeneous groups of machines with respect to hardware and software. We implemented a feature selection based on a genetic algorithm (GA) to show the applicability of Sputnik.

2 Functionality

Our parallelization framework Sputnik is implemented in Clojure (www.clojure.org), a Lisp dialect based on the Java Virtual Machine (JVM). Clojure has a strong functional orientation and a built-in software transactional memory that facilitates the Sputnik server implementation.

2.1 Distributed computation

The implementation of Sputnik is divided into three parts one for each role (client, worker and server, see in Fig. 1). The communication between these programs is implemented on top of Java Remote Method Invocation and uses the Kryo library (github.com/EsotericSoftware/kryo) for data serialization. Custom serializers are added for the common data structures of Clojure. Optionally, data compression can be activated. Secure Socket Layer (SSL) encryption can be configured for secure communication. Authentication is implemented via SSL client certificates. Sputnik has a client implementation that provides the basic operations: connecting to the server, job submission and asynchronous task completion notification. The client implementation of Sputnik works asynchronously and thus allows other local computations to take place while the tasks are computed remotely. The Sputnik server manages all jobs and dynamically assigns their tasks to the worker nodes. Each worker node w has a maximum of tasks $T_{\max}(w)$ that it processes in parallel. Hence, the server only assigns a number of tasks proportional to $T_{\max}(w)$ at a time to a worker w. This dynamic scheduling of tasks results in a superior performance than dividing all tasks among the connected workers instantly on job submission as the slowest worker is not able to claim a large number of tasks. It also enables dynamic addition and removal of worker nodes. Additionally, in the final phase of the computation when all tasks are assigned and workers start to run out of tasks, the Sputnik server may start to assign tasks multiply to idle workers to reduce the overall runtime. For a given worker $w \in W$ the tasks $\gamma \in \Gamma$ with the lowest number of assignments to other workers are then selected first. Among these the tasks γ^* with the latest estimated minimal completion time are assigned first:

$$\gamma^* = \arg\max_{\gamma \in \Gamma}\left\{\min_{w \in W}\{t^{\text{compl}}(w,\gamma)\}\right\}$$

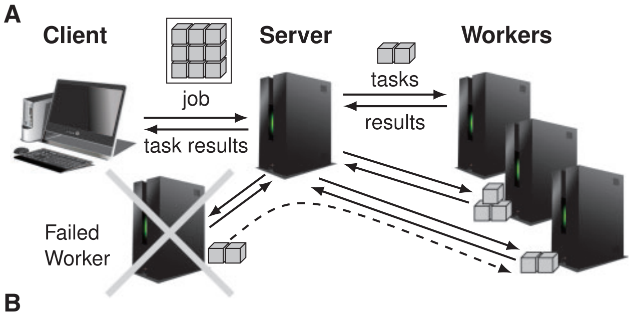
$$t^{\text{compl}}(w,\gamma) = \begin{cases} \dfrac{N_{\text{prev}}(w,\gamma)+1}{\bar{v}(w)} & \bar{v}(w)\text{ known} \\ 0 & N_{\text{prev}}(w,\gamma) = 0 \\ \infty & \text{otherwise} \end{cases}$$

$N_{\text{prev}}(w,\gamma)$ is the number of tasks that are assigned to worker w previously or at the same time as task γ. $\bar{v}(w)$ is the average computation speed of worker w. The first result for a task that was assigned to multiple workers is accepted, the others are discarded. Distributed computation via Sputnik is fault-tolerant with respect to severe failures at the worker nodes because in that case the Sputnik server will reassign tasks of the crashed worker node to other worker nodes.

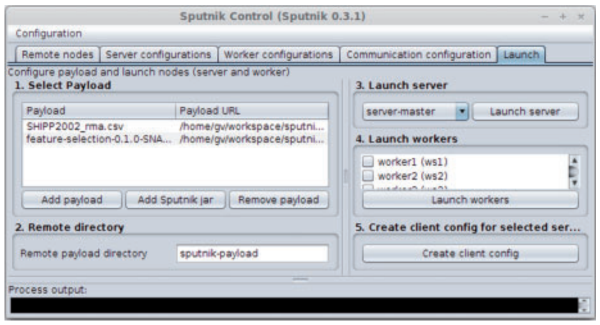
2.2 Lightweight installation and deployment

The deployment of the server and worker nodes requires only a regular user account (accessible via SSH) on the remote machines. No administrator rights are needed to deploy Sputnik nodes. In case there is no Java installed on the remote machines, Sputnik supports local installations of the Java Runtime in the home directory of the user. Sputnik uses the Pallet library (palletops.com) to distribute all needed files to the remote machines and to start the server and workers. The settings for a Sputnik setup are specified in configuration files. The mandatory settings for a remote machine are the node name, a user account name and its IP address. There are optional parameters that allow customization, e.g. non-standard SSH ports and custom JVM installations. Sputnik offers a graphical user interface for deployment (see Fig. 1) that allows easy configuration and launching of the server and the workers. The role of a node is specified similarly either as server or worker with corresponding settings. The deployment process of Sputnik creates a directory on the remote machine which contains a configuration file with the runtime settings for the node, the additional files that are needed for the computation and a startup script for the node. Hence, in case a severe failure occurs in a task and shuts down the worker (despite the worker being able to handle common exception scenarios) the worker can be manually restarted by the user. Based on the configuration, Sputnik deploys all needed files automatically to the remote machines.

A



B


Sputnik Control (Sputnik 0.3.1)

Configuration

Remote nodes | Server configurations | Worker configurations | Communication configuration | Launch

Configure payload and launch nodes (server and worker)

1. Select Payload

Payload	Payload URL
SHIPP2002_rma.csv	/home/gv/workspace/sputni...
feature-selection-0.1.0-SNA...	/home/gv/workspace/sputni...

Add payload | Add Sputnik jar | Remove payload

2. Remote directory

Remote payload directory: sputnik-payload

3. Launch server

server-master | Launch server

4. Launch workers

worker1 (ws1)
worker2 (ws2)

Launch workers

5. Create client config for selected ser...

Create client config

Process output:

Fig. 1. (A) Distributed computation scenario: the server assigns tasks of submitted jobs to available workers (reassignment on worker failure). **(B)** Graphical user interface for deployment: Settings for the server and the workers

2.2.1 Server user interface

The server node offers a web user interface which provides summary information about the performance of running worker nodes and the progress of the running jobs. After logging in with the configured user name and password more detailed information is displayed and the number of parallel computations on each worker node can be changed. A progress report for the running jobs and an estimation of their completion time is shown. Exceptions that occurred during the computations are also accessible in the web user interface.

3 Application: signature identification

We demonstrate the usage of Sputnik by the parallelization of a population-based feature selection algorithm for a nearest neighbor classifier (1-NN, see also Lausser *et al.*, 2014; Jirapech-Umpai and Aitken, 2005). A detailed description of the experimental setup can be found in the supplementary information. The method is based on a GA that utilizes the *Merit* measure for evaluating the fitness $f(s)$ of an individual s (feature combination) (Hall, 2000):

$$f(s) = \frac{k\overline{r_{\mathrm{cf}}}(s)}{\sqrt{k + k(k-1)\overline{r_{\mathrm{ff}}}(s)}},$$

where k is the total number of features, $\overline{r_{\mathrm{cf}}}(s)$ the average feature-class correlation and $\overline{r_{\mathrm{ff}}}(s)$ the average feature-feature intercorrelation of the feature combination s. Fitness evaluation is the most time-consuming part of the algorithm and is parallelized. As can be observed from Table 1, the classification results on the reduced data match those of utilizing all gene expression markers, while generating a substantially reduced signature (up to 99.78%). Also the calculation of the *Merit* measure scales well with the number of workers and cores used, see Table 2.

Table 1. Results for the classification experiments on the different datasets (2/3 training, 1/3 test)

Dataset	Accuracies with and (without) selection	No. of features	Feature selection
Armstrong	0.950 ± 0.04 (0.946 ± 0.02)	28.2 ± 8.2	0.22%
Golub	0.929 ± 0.05 (0.925 ± 0.03)	29.4 ± 4.7	0.41%
Shipp	0.924 ± 0.07 (0.924 ± 0.05)	28.4 ± 2.1	0.40%
West	0.806 ± 0.05 (0.775 ± 0.09)	21.8 ± 6.0	0.31%

For 10 independent runs, the following properties of the best individuals are reported: accuracies for 1-NN classifier with and without gene selection (using identical folds) as well as the selected number (mean ± SD) and percentage of features used.

Table 2. Runtime of parallel fitness function evaluations on the West dataset using different numbers of workers and CPU cores

No. of work. (No. of cores)	1 (10)	2 (20)	3 (30)	4 (40)	5 (50)	1 (1)
Average runtime (min)	12.76	6.57	4.51	3.42	2.70	109.56
Speedup	8.56	16.68	22.29	32.04	40.58	1.00

Average runtimes are over 10 repetitions limited to five generations. Ten threads per worker were used in all evaluations.

4 Comparison

In the following we compare Sputnik to other parallelization frameworks that are available for the Java platform: JPPF and Hadoop. We compare these frameworks with respect to the parallelization paradigm and the required setup procedure. Finally, we report results of an experimental comparison between Sputnik and JPPF.

4.1 Parallelization paradigm

The main parallelization principle differs between these three frameworks: Sputnik and JPPF employ task parallelism, whereas Hadoop uses data parallelism. As illustrated in Figure 2 for Sputnik and JPPF computational problems need to be structured into smaller tasks. In JPPF tasks need to be derived from the JPPF Task class such that for different computation tasks one class for each task is needed. Tasks in Sputnik are represented as pure data (function name and argument data). The data representation of tasks provides the flexibility to the client program to decide at runtime which function evaluations will be parallelized. Hadoop uses the map-reduce paradigm to parallelize computations usually on very large data collections. The data for the computation are distributed among the computers of a Hadoop cluster. A computational task to be solved by Hadoop requires a Java class implementing the map step and a Java class implementing the reduce step. Similarly to JPPF, different computations each need their own class, but for certain problems it might be possible to reuse existing map or reduce classes. Also the coding and setup effort to parallelize a sequential algorithm is quite different for the different paradigms (see Fig. 2). When compared with JPPF, Sputnik needs no additional task classes whereas Hadoop needs a

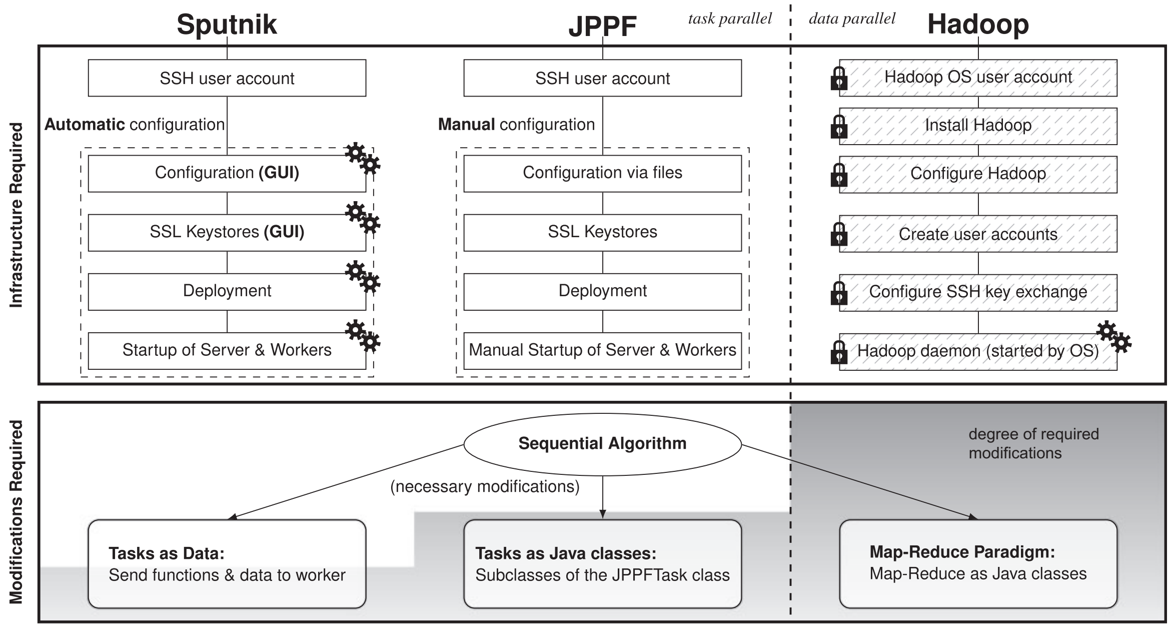


Fig. 2. Comparison of Sputnik, JPPF and Hadoop: For each framework, the necessary steps to setup the server and the workers are shown. The gears mark steps where the user is supported with automatic assistance by the tools of the framework. The padlock symbol marks steps that need administrator privileges. The second part of the graphic summarizes the necessary modifications to parallelize a computational intensive sequential algorithm with each framework. The background of the graphic visualizes the needed degree of required modifications to parallelize an existing sequential algorithm

substantial change in the code of the algorithm and also is more suited for processing distributed data. In the experimental evaluation we therefore focused on Sputnik and JPPF.

4.2 Setup and runtime comparison

The required setup steps for the three frameworks are summarized in Figure 2 (upper part—required infrastructure). The permanent Hadoop setup must completely be done by an administrator. JPPF can be set up either permanently or *ad hoc*. All these steps have to be performed manually. Sputnik is intended for *ad hoc* setup and provides tool support for every step. For Sputnik, configuration files and SSL keystores are generated from the graphical user interface which also offers automatic deployment and automatic startup on the server and on the workers. Sputnik and JPPF do not need administrator privileges, a user account is sufficient.

The basic principles of the scheduling strategies of Sputnik and JPPF are quite different. Sputnik uses a continuous streaming of tasks, whereas JPPF distributes tasks in batches. Sputnik schedules the tasks of a computation job such that the CPU utilization of the workers is maximized. The scheduling strategy of Sputnik tries to have $2n$ task at a worker that performs n computations in parallel. For each finished task a new pending task is sent to the worker as soon as possible. All JPPF scheduling strategies send batches of tasks (*bundles* in JPPF terminology) to each worker. Task results are only sent back from the workers when all tasks of the batch are completed. This can cause situations where only one processor core of the worker is working and $n-1$ cores are idling. The impact of these idle times increases when tasks have quite different runtimes.

We performed experiments with our parallel feature selection algorithm to compare Sputnik and JPPF. The runtime of the tasks of the feature selection algorithm does not vary much. For the experiment we modified the calculation runtimes of the tasks to create a scenario with two types of tasks. The first task type performs its regular calculation in runtime t. The second task type delays the regular calculation by an additional duration Δ. In the experiment every fifth task is of the second type using $t+\Delta$ total runtime on average. The experiments use the same configuration as in the previous experiment. The average task runtime is $t \approx 600\,\text{ms}$. Using $\Delta = 200\,\text{ms}$ the feature selection algorithm using JPPF needs 14% more runtime compared with the algorithm using Sputnik. For $\Delta = 400\,\text{ms}$ this increases to 23%. Even when $\Delta = 0\,\text{ms}$ is used the JPPF scenario needs 8% more runtime than the Sputnik scenario.

5 Conclusion

We devised and implemented a lightweight tool for code parallelization in shared and distributed memory. The tool support of Sputnik enables *ad hoc* on demand parallelization with a user interface for ease of configuration. The task as data paradigm of Sputnik allows decisions on parallelization at runtime. In our simulation experiments, we could successfully utilize the framework for biomarker selection. Furthermore, Sputnik also compares well both in setup and runtime to other frameworks. This supports the feasibility of our approach for applications in bioinformatics and systems biology that are demanding a high computational power and ease of setup.

Funding

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007–2013] under grant 55 agreement n°602783 (to H.A.K.), the German Research Foundation [DFG, SFB 1074 project Z1] (to H.A.K.) and the Federal Ministry of Education and Research [BMBF, Gerontosys II, Forschungskern SyStaR, project ID 0315894A] (to H.A.K.).

Conflict of Interest: none declared.

References

Armstrong,S. *et al.* (2002). MLL translocations specify a distinct gene expression profile that distinguishes a unique leukemia. *Nat. Genet.*, **30**, 41–47.

Golub,T. *et al.* (1999). Molecular classification of cancer: class discovery and class prediction by gene expression monitoring. *Science*, **286**, 531–537.

Hall,M.A. (2000). Correlation-based feature selection for discrete and numeric class machine learning. In: *Proceedings ICML*, pp. 359–366. Morgan Kaufmann, Stanford, CA.

Jirapech-Umpai,T. and Aitken,J.S. (2005). Feature selection and classification for microarray data analysis: Evolutionary methods for identifying predictive genes. *BMC Bioinformatics*, **6**, 148.

Lausser,L. *et al.* (2014). Identifying predictive hubs to condense the training set of k-nearest neighbour classifiers. *Comput. Stat.*, **29**, 81–95.

Shipp,M. *et al.* (2002). Diffuse large B-cell lymphoma outcome prediction by gene-expression profiling and supervised machine learning. *Nat. Med.*, **8**, 68–74.

West,M. *et al.* (2001). Predicting the clinical status of human breast cancer by using gene expression profiles. *PNAS*, **98**, 11462–11467.