Asynchronous PageRank computation in an interactive multithreading environment

Giorgos Kollias[1], Efstratios Gallopoulos[2]

[1] University of Patras, Computer Engineering and Informatics Department
26500, Patras, Greece
gdk@hpclab.ceid.upatras.gr

[2] University of Patras, Computer Engineering and Informatics Department
26500, Patras, Greece
stratis@ceid.upatras.gr

Abstract. Multiple-core processors set the new hardware standard for typical scientific computing platforms thus driving the need for algorithms making extensive use of Thread Level Parallelism (TLP). On the other spectrum end of concurrent platforms, large-scale distributed systems dictate the design of novel algorithms with minimum synchronization constraints and maximum self-adaptation to inevitable dynamic changes in their execution environment.
We advocate the asynchronous computation model [1] as a suitable framework for building such algorithms and propose `Jylab` [2] as an interactive multithreading workbench for comfortably exploring their behaviour. We demonstrate the flexibility of this environment in the asynchronous computation of PageRank and comment on some interesting properties of the computation itself. In this context a multicore machine naturally boosts the performance of an application consisting of asynchronously computing threads and `Jylab` facilitates such multithreaded designs.
Ultimately concurrency at CPU level can drive decisions for alternative ways of organizing concurrency at Internet level but can also serve as the final execution platform. [3] [4]

1 Introduction

1.1 Overview

Traditionally concurrent computations are implemented either over shared- or distributed- memory hardware architectures. These are either directly exposed to the application layer or used to build virtualizations of alternative but more convenient programming models. Both have found their way to commodity scientific computing platforms through Symmetric MultiProcessing (SMP) machines and Beowulf clusters.

[3] Partial support by a *Pythagoras-I (EPEAEK-II)* research grant-Code No:B365016
[4] Work conducted in collaboration with Daniel Szyld

Dagstuhl Seminar Proceedings 07071
Web Information Retrieval and Linear Algebra Algorithms
http://drops.dagstuhl.de/opus/volltexte/2007/1065

However in recent years opportunities for parallel execution have broadened their scope: Availability of increasingly faster network connections within Internet make it practical to engage existing local clusters or even isolated machines to distributed computations and multicore processors, literally realizing performant clusters within a desktop machine, dramatically lower the barrier -basically in terms of deployment and maintenance- in developing parallel solutions to problems.

These advances introduce new challenges in algorithm design and usability. E.g. Internet-scale distributed computations should not be direct ports of cluster-based (especially tightly-coupled parallel) scientific computations; per step synchronization and all-to-all communication, although possibly dictated by its sequential analog, could raise excessively long idle phases and unacceptable network overload. On the other hand a monolithic sequential implementation of an algorithm would lose the performance speedup offered by available CPU cores. Complementary usability issues are also important: In a typical scenario a scientist uses an interactive workbench for algorithm developement and he should be isolated by intricacies (e.g. fault and security management) of the underlying platforms he uses; nonetheless he should be given friendly access to their fundamental user-level abstractions.

Multicore machines encourage organizing an algorithm into interacting tasks (to be scheduled by the operating system to individual cores at runtime). Tasks are typically realized by processes or threads; in fact, the thread model is closer to multicore design. So the interactive use of threads during development should be regarded as a compelling feature of modern scientific workbenches, especially when run over multicore machines; the user can easily test alternative ways of "keeping his cores busy" and, perhaps more importantly, deepens his perception of algorithm parallelization issues. This is expected to be an important skill in software practice in years to come since fundamental limits in CPU design migrate the emphasis from increasingly *more powerful* CPUs to just increasingly *more* CPUs packaged together. Software approaches already accomodate this shift: OpenMP standard compiler directives [3], shared memory implementations of MPI specification, language dialects [4] and new languages [5].

1.2 Paper focus

`Jylab` [2] offers such interactive multithreading capabilities to the user. We use this system to conveniently evaluate the behavior of the asynchronous computational model as applied to a commonly encountered problem in Internet algorithmics, namely PageRank calculation. The emphasis is twofold: Demonstrate the flexibility of `Jylab` in building a multithreaded simulation environment and also stress some virtues of asynchronous computations -as drawn from subsequent experimentation- which make them attractive for large-scale parallel computations involving naturally distributed data-sets.

This paper is organized as follows: In the first two sections we present `Jylab` and PageRank computation within an asynchronous setting. These are followed by an extensive section containing information on experimentation setup and

numerical results interspersed with comments. We conclude with a synopsis of the main points in our approach and future research directions.

2 Jylab

`Jylab` is a new system built to enable access to parallel programming methodologies in scientific computing (multithreading, multiprocessing), however keeping attractive features of common workbench-type environments [6,7,8].

- *Interactive environment* and a *language for composing scripts* (all above systems have this feature). Such a language, rising in popularity and supporting object-oriented, procedural and even functional programming features is `Python` [9], which comes also with an interactive shell.
- *Maximum portability* which made `Java` attractive; `Jython` [10,11], i.e. `Python` implemented in `Java` has these features.
- Support for *interprocess communication across machine boundaries*, *numerical computations* (especially matrix based), *visualization* and *symbolics*; we integrated corresponding open-source class libraries from the Web.

It is important to note that it is always possible to speedup a computation by coding it in `Java` (recent versions of JVM rival even C++ performance [12]).

`Jylab` is actually an implementation of a portable Problem Solving Environment (PSE) [13] targeting a broad and extensible group of disciplines; PSEs can dramatically reduce the time-to-solution-by-computer for problems in science, by exposing computing capabilities in a most comprehensive, flexible and application-domain-friendly way.

Its core functionality is supported by a suite of packages for scripting (`Jython`), numerical linear algebra (NLA) computations (`Colt` [14], `MTJ`) [15]), interprocess communication (`Ibis` [16]), interactive visualization (`VisAD` [17]) and minimal computer algebra manipulations (`MathEclipse` [18]). However a collection of extension packages have also been successsfully tested within `Jylab` context to facilitate e.g. Grid computing (`JavaGAT` [19] and `ProActive` [20]), search engine pipelines (`Nutch` [21]) and Web graph analysis (`WebGraph` [22]).

3 Asynchonous PageRank computation

There are several ideas being used today for Web information retrieval, and specifically in Web search engines [23]. The PageRank algorithm [24] is one of those that introduce a content-neutral ranking function over Web pages. This ranking is applied to the set of pages returned by the Google search engine in response to posting a search query. PageRank is based in part on two simple common sense concepts: (i) A page is important if many important pages include links to it. (ii) A page containing many links has reduced impact on the importance of the pages it links to.

In order to compute PageRank a matrix representation of the Web link structure (Web matrix) is typically employed. However due to the size of such a -sparse- Web Matrix (representing more than 10^{10} pages with 10^{11} nonzero elements and 10^{12} bytes for storage) distributed memory, parallel implementations are commonly developed with each processor hosting part of this matrix in its memory and engaged in a long iterative computation with a tight-coupling communication pattern, also introducing synchronizing phases and corresponding delays. The main intention of this paper is to demonstrate that a PageRank computation can readily be ported to the asynchronous communication model which permits deployment to loosely-coupled distributted platforms with no synchronization constraints. Under this model a processor iterates without stepwise blocking in waiting for update information from other processors but instead carries on computing with whatever -most recent data- is available.

3.1 PageRank

In order to appreciate the PageRank computation, we present its standard formulation using the following set of four $n \times n$ matrices, where n is the number of pages being modeled.

An *adjacency matrix* A can be obtained through a web crawl or synthetically generated using statistical results, e.g., as in [25]. Thus, $\alpha_{ij} = 1$ *iff* page i points to page j, and $\alpha_{ij} = 0$ otherwise.

A *transition matrix* P has nonzero elements $\pi_{ij} = \alpha_{ij}/\deg(i)$ when $\deg(i) \neq 0$, and zero otherwise (in which case page i is called a *dangling* page); here $\deg(i) = \sum_j \alpha_{ij}$ is the outdegree of page i.

A *stochastic matrix* S is given by $S = P^\top + \mathrm{w}\ \mathrm{d}^\top$; $\mathrm{w} = \frac{1}{n}\mathrm{e}$, where e is the size n vector of all 1's, and d is the *dangling index vector* whose nonzero elements are $\delta_i = 1$ *iff* $\deg(i) = 0$.

The *Google matrix* G is [5] $G = \mu\ S + (1 - \mu)\ \mathrm{v}\ \mathrm{e}^\top$. For a random web surfer about to visit his next page, the relaxation parameter μ is the probability of choosing a link-accessible page. In choosing otherwise, i.e., with probability $1 - \mu$, from the complete Web page set vector v contains respective conditional probabilities of such *teleportations*. Typically $\mathrm{v} = \mathrm{w}$ and $\mu = 0.85$.

The PageRank vector x is the solution of the linear system

$$\mathrm{x} = G\ \mathrm{x}\ , \tag{1}$$

where the matrix G is an irreducible stochastic matrix, and thus its largest eigenvalue in magnitude is $\lambda_{max} = 1$ [26]. Thus, the PageRank vector x is the eigenvector corresponding to $\lambda_{max} = 1$, and when normalized, it is the stationary probability distribution over pages under a random walk on the Web, i.e., the

[5] For the sake of consistency with the Householder's notational conventions, we opt to use μ rather than α for the relaxation parameter, since the latter might mistaken as an element of A.

invariant measure of a Markov process modeled by matrix G. To make the computation of x practical for the problem sizes we are considering, it is necessary to employ an iterative method, e.g. executing until convergence

$$\mathrm{x}(t+1) = G\,\mathrm{x}(t), \quad \mathrm{x}(0) \text{ given.} \tag{2}$$

This is the well-known power method for finding the eigenvector of G corresponding to the eigenvalue of largest magnitude [26]

In a parallel implementation one usually partitions G into sets of rows and suitably distributes them to available units of executions (UEs); in a typical synchronous setting each of them will produce corresponding x fragments which will have to be communicated to all other peers in lockstep. Alternative approaches reformulate PageRank as the solution to a linear system of equations [27] and parallelize its computation accordingly [28]. For synchronous parallel PageRank implementations we refer to [29,30]

3.2 Asynchronous computation model

For an environment with p UEs, denote by $\mathrm{x}_{\{i\}}$ the set of indices assigned to i^{th} UE during the iterative computation, T^i the set of times at which $\mathrm{x}_{\{i\}}$ is updated (i.e., i^{th} UE finishes its computation) and $\tau_j^i(t)$ the time when the fragment $\mathrm{x}_{\{j\}}$, which is available at time t in the i^{th} UE, was actually produced at its respective j^{th} UE. Then for $t \in T^i$, the i^{th} UE updates

$$\mathrm{x}_{\{i\}}(t+1) \leftarrow f_i(\mathrm{x}_{\{1\}}(\tau_1^i(t)), \ldots, \mathrm{x}_{\{p\}}(\tau_p^i(t))), \tag{3}$$

while $\mathrm{x}_{\{i\}}(t+1) = \mathrm{x}_{\{i\}}(t)$ at other times. Delays due to omission of synchronization phases are expressed as differences $t - \tau_j^i(t) \geq 0$; here f_i expresses the distributed operator component executing at the i^{th} UE. Obviously the form of f_i is independent of the asynchronism introduced. It thus follows that the normalization-free power method for PageRank computation at the i^{th} UE reads

$$\mathrm{x}_{\{i\}}(t+1) = g_i\,[\mathrm{x}_{\{1\}}^\top(\tau_1^i(t)), \ldots, \mathrm{x}_{\{p\}}^\top(\tau_p^i(t))]^\top \tag{4}$$

for $t \in T^i$, and $\mathrm{x}_{\{i\}}(t+1) = \mathrm{x}_{\{i\}}(t)$ at other times, where g_i is a set of rows of the Google matrix G indexed by $\{i\}$. The lack of synchronization annuls the semantics of the original mathematical algorithm. Therefore, it becomes necessary to discuss the convergence properties of the asynchronous scheme (3). Basically, convergence of asynchronous iterative algorithms is usually established through constructing a sequence of nested boxed sets in the spirit of the following theorem [31]:

Theorem 1. *Let* $\{X(k)\} : \ldots \subset X(k+1) \subset X(k) \subset \ldots \subset X$, *with the following two conditions.*
Synchronous Convergence Condition: For all $k = 1, \ldots$, $\mathrm{x} \in X(k), f(\mathrm{x}) \in X(k+1)$, *and for* $\{\mathrm{y}^k\}$, $\mathrm{y}^k \in X(k)$: *the limit points of* $\{\mathrm{y}^k\}$ *are fixed points of* f.
Box Condition: For all $k = 1, \ldots$, $X(k) = X_1(k) \times \ldots \times X_n(k)$.
Then if $\mathrm{x}(0) \in X(0)$, *the limit points of* $\{\mathrm{x}(t)\}$ *are fixed points of* f, *where* $\{\mathrm{x}(t)\}$ *are given by (3).*

Process (4) involves a nonnegative matrix of unit spectral radius; it is proved in [32] that the corresponding asynchronous iteration converges to the true solution within a multiplicative factor that can easily be factored out at the end by renormalization; cf. [33]. In [34] a cluster-based implementation of (4) is evaluated; interesting P2P simulations have also been developed [35,36].

4 Experimentation

4.1 Setup

We used a notebook equipped with Intel dual-core T7200 processor (2GHz, 4MB L2 cache) and 2GB RAM, running Linux (kernel v2.6.17). Java virtual machine (v1.5.0) hosted `Jylab`'s scripting component, Jython (v2.1) and multithreaded code was conveniently developed in supported Python syntax making extensive use of a webmatrix class library separately coded in Java(see Fig 1). The transition matrix used in the experiments is mainly the Stanford Web matrix [37], generated from an actual web-crawl ($n = 281,903$ pages, $2,312,497$ non-zero elements, 172 dangling nodes) but in at least one test case, synthetic Barabasi-Albert [38] graphs were alternatively used. Local convergence threshold was set to 10^{-4} ($||\cdot||_1$). Note that in each case, blocks of consecutive $[n/p]$ rows were distributed among p computing peers (executing on a thread each); therefore, no load balancing was specifically attempted. Termination detection was determined by checking special global flags, set whenever "converged" status *persisted* locally (i.e. within a peer's execution context).

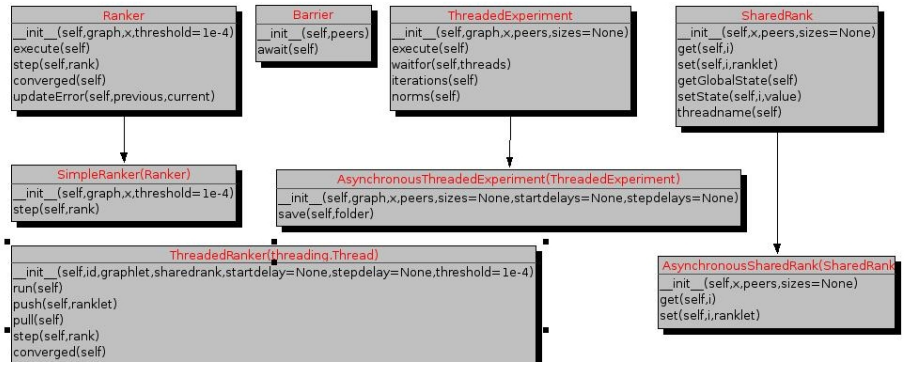

Fig. 1. Multithreaded code organization: class and method names are given

4.2 Results

Various configurations of up to 192 concurrently executing asynchronous peers were tested and convergence to PageRank vector was observed at all cases. All experiments are controlled by two parameter vectors:

`startdelays[]` contains values of time delays (in secs) for the respective peer to join the computation. A value of `None` at its i^{th} position simply states that peer with $ID = i$ started with no delay. This is very useful since it permits us to investigate the behavior of asynchronous PageRank under dynamic changes in the computation; a delayed peer when started introduces its corresponding part of the web link structure along. So this parameter can determine timing of massive and abrupt changes in processed data.

`stepdelays[]` contains values of delays (in secs) synthetically injected at each step of the iteration performed by peer with corresponding ID. This parameterizes heterogeneous machines in terms of speed: large values simulate slow machines. It follows that `None` lets the underlying (fair) thread scheduler decide.

`iterations[]` is an output vector of the total number of iterations performed by each asynchronously executing peer to reach convergence. Although large numbers are sometimes recorded here (in cases of either strong heterogeneity or late startup peers) the decisive comparison metric (e.g. to the synchronous parallel-synchronous case) then, should be the lowest number of iterations, because in a synchronous setting, execution time finish is governed by the last UE to converge (typically the slowest) and its start by the last UE to join (in a synchronous setting all peers should be simultaneously available at start).

Figure 2 shows a distinctive feature of asynchronous computations compared to their synchronous counterparts: In the course of an asynchronous iterative process a peer can enter and exit 'converged' status multiple times quite independently from other UEs. This happens as a possibly 'converged' process receives data -completely out of sync- from either asynchronous 'futures' or 'pasts' (i.e. other processes respectively having performed more or less iterations). Convergence in PageRank case is finally attained across all peers and this comes as a numerical verification of the general theoretical result in [32], as mentioned earlier. Predicting exact number of iteration steps per peer seems impossible. Contrast it to the synchronous case when convergence is 'synchronously' reached (in a precomputable number of steps).

`startdelays[]`	`stepdelays[]`	`iterations[]`
None	[None, 1., 2., 3.]	[2071, 82, 42, **28**]
None	[None, 2., 4., 6.]	[4605, 83, 42, **28**]
None	None	[58, 50, 57, 48]
[10.,None,20.,None]	None	[241, 307, 105, 260]

Table 1. Experiments (4 peers, Stanford Web matrix) for a variety of `startdelays[]` and `stepdelays[]` (in secs). Number of iterations to convergence for each peer (`iterations[]` vector) is given.

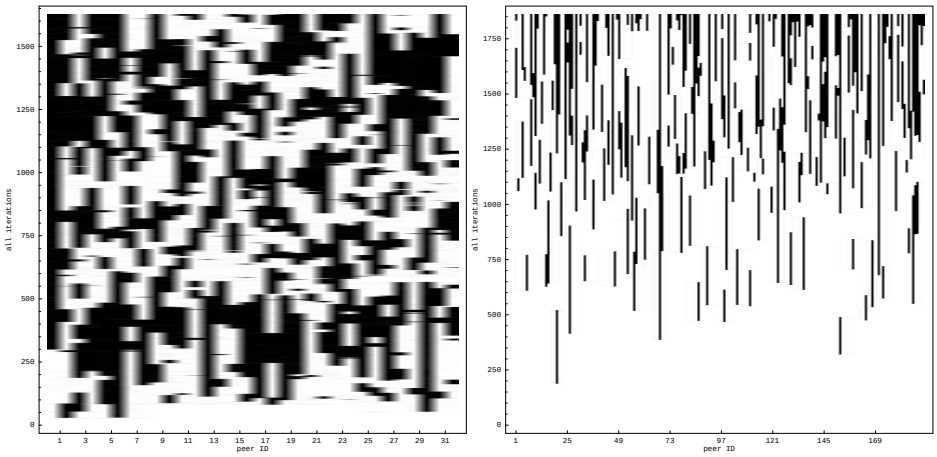

Fig. 2. Asynchronous convergence patterns: a vertical column represent the timeline of convergence status for a specific peer, with black denoting non-convergence, white that convergence has been reached. Global time runs downwards starting when less than half of the total number of peers were left in non-converget state for the first time. Left is for 32 peers (synthetic Barabasi-Albert Web matrix, 100k pages), right is for 192 peers (Stanford Web matrix). `startdelays=None`, `stepdelays=None`.

Tables 1 and 2 summarize some experiments with 4 peers. E.g. table 2 focuses on the scenario that one of the peers starts computing long after the others. In the extreme case when this peer is the slowest too, we can draw a very important conclusion: Note that this slow and late processor needs less than 39 iterations (this is for the sequential-synchronous case for the same error threshold). But since this processor decides results delivery time, we can say that it seems advantageous to start up computation immediately in the absence of slow machines to be available later on, rather than deferring calculation launch. Other results (some from table 1 also) generally suggest the following strategy: prefer to asynchronously start computing (if the alternative is to wait for all peers to become available) and although small the gain towards convergence per iteration (note the large iteration numbers at 'fast' machines) certainly it does no harm; on the contrary this helps to critically reduce iterations at slow peers.

Figure 3 graphs error evolution in the case of simultaneously started peers computing at different speeds: fastest one exhibits most variations in its graph. In Figure 3, on the other hand, peers are homogeneous but started at different times. We observe that the introduction of 'new' computing threads is clearly sensed by 'old' ones in the form of abrupt changes in their error profile (these points of change can also be used for translating iteration steps to real time intervals). The important thing here is that although severely perturbed, no special treatment is needed for the computation (e.g. no need to restart or checkpoint

`startdelays[]`	`stepdelays[]`	`iterations[]`
[3 x None, 15.]	None	[202, 207, 213, 52]
[3 x None, 15.]	[3 x None, 1.]	[486, 445, 475, **28**]
[3 x None, 15.]	[3 x None, 2.]	[743, 757, 801, **28**]
[3 x None, 15.]	[1., 1., 1., None]	[53, 53, 53, 1009]
[3 x None, 15.]	[2., 2., 2., None]	[46, 46, 46, 2068]

Table 2. Experiments (4 peers, Stanford Web matrix) for a variety of `stepdelays[]` when one of the peers joins the computation after it has started. Number of iterations to convergence for each peer (`iterations[]` vector) is given.

it); asynchronous computing seems to automatically adapt to dynamic changes (here more peers and drastic changes in link structure) and find its way to convergence.

5 Conclusions

Asynchronous computing is an alternative, albeit not a substitute, for the parallel synchronous model, especially suited whenever the latter becomes infeasible or impractical. Mature message passing technology both in terms of software libraries and hardware infrastructure is well established for the synchronous model; however when considering it we make an assumption that data to operate on is available at computation site.

On the other hand the asynchronous model appears to be the only way to go when one of the following applies: Data are naturally distributed and no central infrastructure exists for collecting and processing it, the underlying processing algorithms ideally demand tight communication between computing elements (but communication links are slow), hardware failures are very likely while a restart protocol is prohibitive, or participating peers are not all guaranteed to be readily available at the start.

One could perhaps point out shortcomings of this model: Not all algorithms can be correctly expressed under this model, there are difficulties in proving rate of convergence, common software stacks support it only at low-level. But our results expose some important features of asynchronous computations:

- they exhibit interesting behavior for slow or delayed-startup peers
- they are robust; dynamic changes are smoothly integrated in the course of the calculation without need for restarts.
- this is an adaptive and most scalable model; data could be engaged in a computation in place and communication patterns are drastically relaxed.

Our research so far, however, indicates that it is eminently suitable for problems of the sizes likely in Internet Algorithmics.

Our future work aims at extracting further numerical evidence on the behavior of the asynchronous model: Porting it to new execution platforms (e.g. Grid

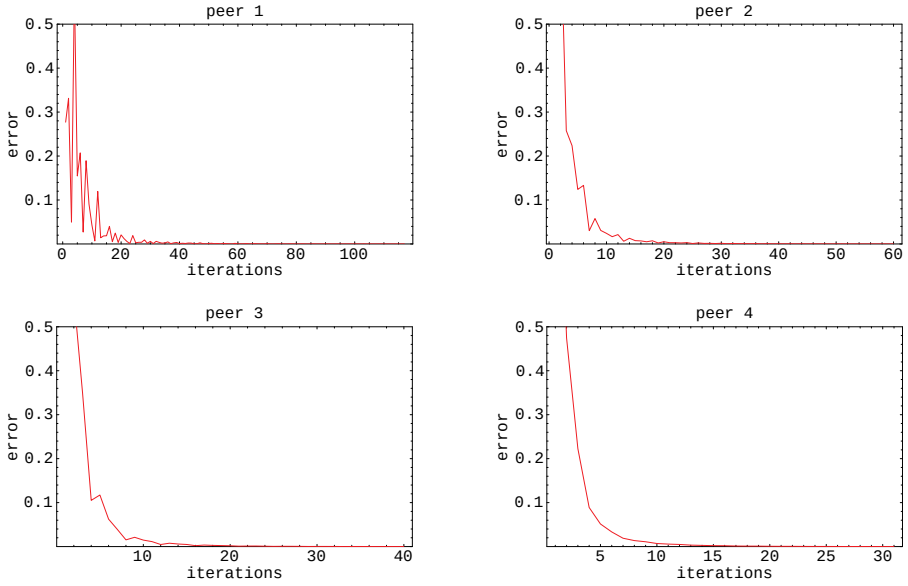

Fig. 3. The effect of heterogeneity: error vs number of iterations for 4 asynchronous peers working on Stanford Web matrix. All peers are started simultaneously but we artificially introduce per step delays (in secs) `stepdelays=[1.,2.,3.,4.]` for corresponding peers. Note that `iterations=[117,60,40,31]`

[39]) looks promising. We are also going to investigate the role of Web updates to the asynchronous iterative process and the feasibility of concurrent crawling and asynchronous ranking.

`Jylab` with its interactive route to full multicore utilization already helps us to shed light on the pragmatics of the promising asynchronous computational model and provide arguments for its integration in typical scientific computing practices.

References

1. Frommer, A., Szyld, D.B.: On asynchronous iterations. J. Comput. Appl. Math. **123**(1-2) (2000) 201–216
2. Kollias, G., Gallopoulos, E.: Jylab: A System for Portable Scientific Computing over Distributed Platforms. In: 2nd IEEE Int'l. Conf. on e-Science and Grid Computing(e-Science 2006): Session on Innovative and Collaborative Problem Solving, Amsterdam, IEEE (December 2006)
3. : OpenMP official site. http://www.openmp.org
4. Yelick, K., Semenzato, L., Pike, G., Miyamoto, C., Liblit, B., Krishnamurthy, A., Hilfinger, P., Graham, S., Gay, D., Colella, P., Aiken, A.: Titanium: A high-

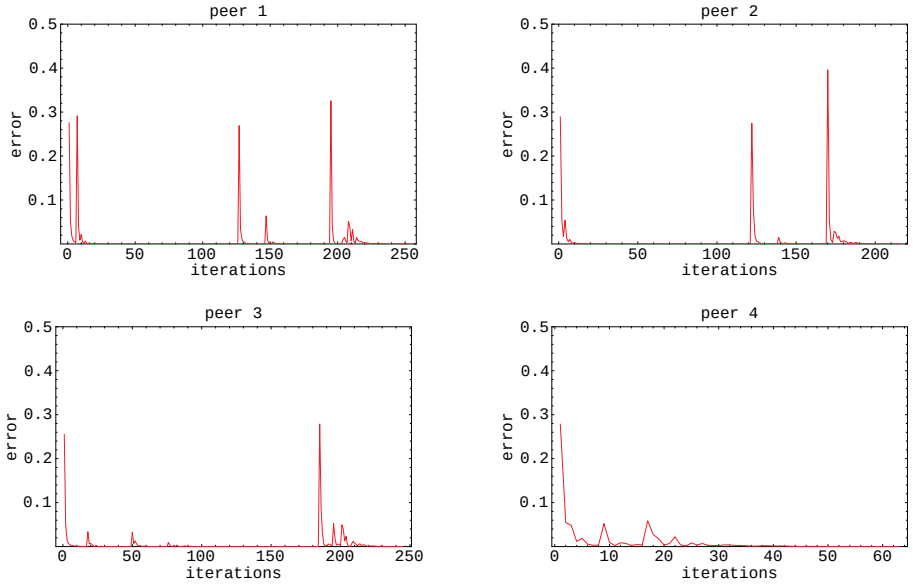

Fig. 4. The effect of massive changes in underlying Web matrix during the computation: error vs number of iterations for 4 asynchronous peers working on Stanford Web matrix. Peers 3 and 4 join the computation 10 and 20 (in secs) respectively later than peers 1 and 2 (which start computing simultaneously), i.e. `startdelays=[None,None,10.,20.]`. Note that `iterations=[252,215,245,63]`
and `stepdelays=None`

performance Java dialect. In ACM, ed.: ACM 1998 Workshop on Java for High-Performance Network Computing, New York, NY 10036, USA, ACM Press (1998)
5. : Fortress project website. http://fortress.sunsource.net/
6. : Matlab website. http://www.mathworks.com/
7. : Scilab website. http://www.scilab.org/
8. : Mathematica website. http://www.wolfram.com/
9. : Python language website. http://www.python.org
10. Hugunin, J.: Python and Java - the best of both worlds. In: Proceedings of the 6th International Python Conference, San Jose, Ca. (October 1997) 11–20
11. : Jython website. http://www.jython.org
12. : Java vs C++ performance benchmark data website. http://kano.net/javabench/data
13. Gallopoulos, E., Houstis, E., Rice, J.R.: Computer as Thinker/Doer: Problem-Solving Environments for Computational Science. IEEE Comput. Sci. Eng. **1**(2) (1994) 11–23
14. : Colt Java class library website. http://dsd.lbl.gov/~hoschek/colt/
15. : Matrix Toolkits for Java website. http://rs.cipr.uib.no/mtj/
16. van Nieuwpoort, R.V., Maassen, J., Wrzesinska, G., Hofman, R., Jacobs, C., Kielmann, T., Bal, H.E.: Ibis: a Flexible and Efficient Java based Grid Programming

Environment. Concurrency and Computation: Practice and Experience **17**(7-8) (June 2005) 1079–1107
17. : VisAD project website. http://www.ssec.wisc.edu/~billh/visad.html
18. : MathEclipse project website. http://www.matheclipse.org/me/
19. G. Allen, K. Davis, T. Goodale, A. Hutanu, H. Kaiser, T. Kielmann, A. Merzky, R. van Nieuwpoort, A. Reinefeld, F. Schintke, T. Schütt, E. Seidel and B. Ullmer: The Grid Application Toolkit: Towards Generic and Easy Application Programming Interfaces for the Grid. In: Proceedings of the IEEE. Volume 93. (2005) 534–550
20. : ProActive project website. http://www-sop.inria.fr/oasis/ProActive/
21. : Nutch project website. http://lucene.apache.org/nutch/
22. : WebGraph project website. http://webgraph.dsi.unimi.it/
23. Langville, A.N., Meyer, C.D.: A Survey of Eigenvector Methods for Web Information Retrieval. SIAM Rev. **47**(1) (2005) 135–161
24. Page, L., Brin, S., Montwani, R., Winograd, T.: The PageRank Citation Ranking: Bringing Order to the Web. Technical report, Stanford University (1998)
25. Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., A.Tomkins, Wiener, J.: Graph structure in the web: experiments and models. In: 9th International World Wide Web Conference. (2000)
26. Stewart, W.J.: Introduction to the Numerical Solution of Markov Chains. Princeton University Press (1994)
27. G.M. Del Corso, A.G., Romani, F.: Fast pagerank computation via a sparse linear system. In: Lecture Notes in Computer Science. Volume 3243. (2004) 118–130
28. Gleich, D., Zhukov, L., Berkhin, P.: Fast Parallel PageRank: A Linear System Approach. Technical report, Yahoo! Inc. (2004)
29. Kohlschütter, C., Chirita, P., Nejdl, W.: Efficient parallel computation of pagerank. In: Proceedings of the 28th European Conference on Information Retrieval (ECIR), London, United Kingdom (2006)
30. Manaskasemsak, B., Rungsawang., A.: Parallel pagerank computation on a gigabit pc cluster. In: Proceedings of the 18th International Conference on Advanced Information Networking and Applications, Los Alamitos, CA, IEEE Computer Society (2004) 273–277
31. Bertsekas, D.P., Tsitsiklis, J.N.: Parallel and Distributed Computation. Prentice Hall, Englewood Cliffs, NJ (1989)
32. Lubachevsky, B., Mitra, D.: A Chaotic, Asynhronous Algorithm for Computing the Fixed Point of a Nonnegative Matrix of Unit Spectral Radius. J. ACM **33**(1) (January 1986) 130–150
33. Szyld, D.B.: The mystery of asynchronous iterations convergence when the spectral radius is one. Technical Report 98-102, Department of Mathematics, Temple University, Philadelphia (October 1998)
34. Kollias, G., Gallopoulos, E., Szyld, D.B.: Asynchronous iterative computations with web information retrieval structures: The PageRank case. In Joubert, G., Nagel, W., Peters, F., Plata, O., Tirado, P., Zapata, E., eds.: Parallel Computing: Current and Future Issues of High-End Computing (Proceedings of the International Conference Parco05). Volume 33 of NIC Series., Jülich, Germany, John von Neumann-Institut für Computing (NIC) (2006) 309–316
35. Parreira, J.X., Donato, D., Michel, S., Weikum, G.: Efficient and decentralized pagerank approximation in a peer-to-peer web search network. In: VLDB. (2006) 415–426
36. Sankaralingam, K., Sethumadhavan, S., Browne, J.C.: Distributed Pagerank for P2P Systems. In: 12th International Symposium on High Performance Distributed Computing. (2003)

37. : Stanford Web Matrix. http://nlp.stanford.edu/~sdkamvar/data/stanford-web.tar.gz
38. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439) (1999) 509 – 512
39. Foster, I., Kesselman, C., Tuecke, S.: The anatomy of the Grid: Enabling scalable virtual organizations. Lecture Notes in Computer Science **2150** (2001)