Proceedings of the International Multiconference on
Computer Science and Information Technology pp. 637–645

ISBN 978-83-60810-27-9
ISSN 1896-7094

Multi-level Parallelization with Parallel Computational Services in BeesyCluster

Pawel Czarnul
Faculty of Electronics, Telecommunications and Informatics
Gdansk University of Technology
Poland
Email: pczarnul@eti.pg.gda.pl
http://fox.eti.pg.gda.pl/~pczarnul

***Abstract*—The paper presents a concept, implementation and real examples of dynamic parallelization of computations using services derived from MPI applications deployed in the BeesyCluster environment. The load balancing algorithm invokes distributed services to solve subproblems of the original problem. Services may be installed on various clusters or servers by their providers and made available through the BeesyCluster middleware. It is possible to search for services and select them dynamically during parallelization to match the desired function the service should perform with descriptions of services. Dynamic discovery of services is useful when providers publish new services. Costs of services may be incorporated into the selection decision. A real example of integration of a given function using distributed services has been implemented, run on several different clusters without or with external load and optimized to hide communication latency.**

I. Introduction

PARALLEL computing has come a long way from dedicated parallel applications to grid computing allowing to couple clusters to perform distributed computations. Traditionally, parallel computing has been performed either on shared memory machines or within dedicated multi-node clusters. For the former, threads or processes may be spawned and communicate using shared memory which is supported using e.g. Pthreads [1], Java threads [2], OpenMP [3]. For clusters, MPI [4] and PVM [5] allow development and running parallel programs using the message passing paradigm for communication. For fast communication between processes on one node MPI can use shared memory. MPI implementations may offer various support for threads. Furthermore, computational codes may be wrapped into services, discovered and integrated for distributed computations at a higher level.

Section II investigates existing approaches for parallelization among clusters and proposes a framework for multi-level parallelization through dynamic discovery of services in BeesyCluster that can make use of specific features of the latter. Section III presents a concept and an algorithm for parallelization of computations using BeesyCluster services while Section IV presents experimental results including the ability of the solution to hide latency of Web Service calls. Finally, Section V presents a summary and future work.

II. Related Work and Motivations

Grid systems like CrossGrid [6] or Clusterix [7] use grid middleware such as Globus Toolkit [8] to allow the user to manage, access and share various resources. Grid middlewares allow controlled resource sharing and offer uniform interfaces to access various resources in various administrative domains. In turn, parallelization among clusters has become easier especially for the master-slave or divide-and-conquer paradigms as previously used within clusters [9], [10]. Such parallelization is possible either by running grid-enabled MPI applications or composing distributed services.

Grid-enabled versions of MPI such as MPICH-G2 [11], PACX-MPI [12], BC-MPI [13] allow one application to run on more than one cluster using TCP for inter-cluster and usually faster Infiniband or Myrinet within each cluster. MPICH-G2 uses Globus for job control as can PACX-MPI and LAM/MPI [14] to couple remote clusters. BC-MPI can use BeesyCluster [15] for this purpose [13].

There exist several frameworks that allow management of distributed processing and computations. The MW toolkit [16] allows development of a master-worker style application that works in the distributed environment of Condor. [17] studies load distribution methods for a master-slave scheme on a multi-cluster architecture. Tuning of master-slave applications on grids is analyzed in [18]. Nimrod/G [19] allows resource management and scheduling in a global computational grid. It introduces a concept of computational economy that allows the user to specify the cost and the deadline for the work or negotiate for resources to do the job.

Grid or Web Services can be used as a front-end to HPC resources [20]. This allows parallelization using services. [21] and [22] contain an interesting comparison of performance of a computationally intensive application run on MPI and using Web Services as workers rather than front-ends to HPC resources. The latter showed higher execution times but has advantages in case of firewalls and better interoperability in heterogeneous environments. A methodology to migrate from MPI to Web Service base applications is shown in [22]. [23] implements an Application Management System into an MPI application and its ability to balance load in a computational grid is demonstrated for an application which

partitions initial data for parallel execution on a three site grid. [24] demonstrates an approach to find and use grid services for parallel solving of matrix multiplication – although sequential in the workers. The framework allows to specify priority QoS parameters like accuracy, speed, bandwidth.

In this paper, using BeesyCluster and combining features of the previous works, the author proposes a dynamic master-slave parallelization approach able to discover and invoke computational services spawning MPI applications on HPC resources. The algorithm finds and invokes ready-to-use possibly geographically distributed computational services derived from MPI applications published by possibly independent providers from various clusters/servers incorporating possibly changing costs of such services. The solution focuses on efficient parallelization of computations among the services with hiding communication latency and parallelization on clusters using MPI. Although the master-slave paradigm has been widely used in parallel programming and distributed processing on grids as discussed above, the contribution of this work is that the proposed scheme benefits from particular features offered by the BeesyCluster middleware:

- parallel and sequential applications created by users can be published instantly as services within BeesyCluster and made available as Web Services accessible from outside BeesyCluster. The publisher can grant rights to other BeesyCluster users to run the service and defines QoS parameters such as cost the client needs to pay to invoke the service. When a BeesyCluster client invokes the service, the cost is subtracted from their account. Our latest work [25] shows how thousands of existing Linux applications distributed in packages such as deb or rpm can be published in BeesyCluster and thus be available to be used in the proposed load balancing scheme.
- searching for services that might then be used in the load balancing scheme. Currently, each service deployed in BeesyCluster is described in the database as well as in a UDDI registry based on JUDDI and used by BeesyCluster. The latter may be queried using the UDDI API. We have also implemented [25] an intelligent search mechanism based on the similarity of the service description and user query that uses Apache Lucene. For instance, a user might request services for numerical integration and those with best matching descriptions are returned.
- incorporation of possibly changing costs when discovering and selecting services.
- access to the services is restricted only to BeesyCluster users to whom it has been granted. Also, services offered by *various* providers can be integrated in one load balancing scheme.
- in case of parallel applications run on clusters with queuing systems such as PBS, LSF, LoadLeveler etc., BeesyCluster can handle queuing transparently to the client.

BeesyCluster (Figure 1) is a middleware and front-end to several clusters and allows users to access their system accounts on clusters or servers using a web browser or web services [26]. BeesyCluster features an easy-to-use WWW interface with a file manager with drag and drop in the browser, ability to launch graphical or text sessions with remote clusters through a browser. Among others, users can publish own MPI applications from their system accounts as services (Figure 2) and assign a price for invoking the service and access rights to particular users or groups. The service can then be invoked on the cluster the provider has installed it through BeesyCluster via a web browser (Figure 3) or Web Services [26]. BeesyCluster uses SSH to access clusters (Figure 1). Moreover, such services can be combined into workflows [15]. Since the standard MPI and PVM allow parallelization within a cluster, an MPI or PVM application may be published as a service (also studied in [27] instantly with a few clicks (Figure 2, [26]) and used as a part of a framework for balancing work between clusters by passing distinct data sets to particular services. BeesyCluster co-developed by the author hides the underlying queuing (PBS, LSF etc.) system on the cluster the application was installed on.

III. Concept and Algorithm for Parallelization of Computations using BeesyCluster Services

For composing services, usually a workflow graph $G(V, E)$ is defined [28] where V is a set of vertexes denoting tasks while E is a set of edges denoting dependencies between the tasks. Out of a set of services for each task one is to be chosen so that certain QoS goals are optimized. As an example, this could be minimization of the workflow execution time with an upper bound on the total cost of selected services.

In this paper, we focus on multi-level parallelization of computations in a dynamic master-slave fashion using services published by independent providers from potentially geographically distributed clusters. Each service is a parallel MPI application.

The parallelization framework is implemented on the client side and thus does not depend on grid middleware and allows access to several BeesyCluster servers handling local computer centers in different administrative domains. The main steps of the strategy are shown in Figure 4:

1) Providers publish parallel MPI applications as BeesyCluster services (Figure 2). In case the cluster on which the application runs uses a queuing system, its usage is handled by BeesyCluster and hidden to the user. Providers specify costs which will need to be paid by the client when invoking particular services. Each BeesyCluster user has a virtual purse, can earn on own services used by others. A description of each BeesyCluster service is stored in BeesyCluster's own UDDI registry based on JUDDI. Thus the balancing thread can query for services using SOAP.
2) The client queries BeesyCluster for services capable of performing the given function [25] also taking into account the costs of available services. The client receives a list of capable services. The recently developed

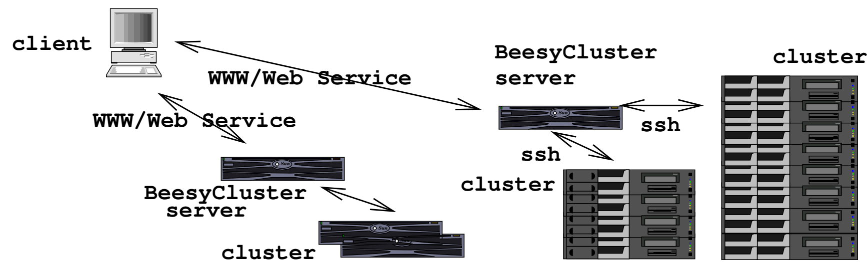

Figure 1: Architecture of BeesyCluster

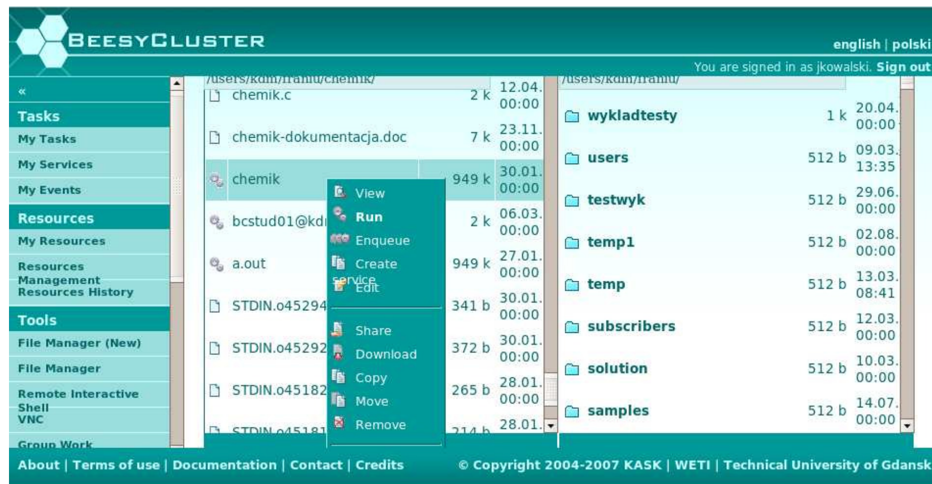

Figure 2: Publishing a Parallel MPI Application as a Service in BeesyCluster

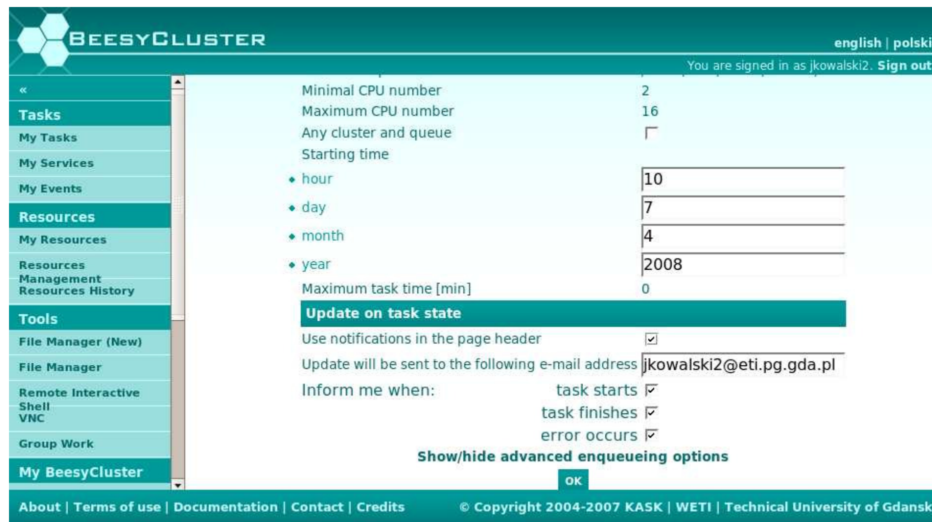

Figure 3: Running a Service Published by Another User and Derived from an MPI Application (WWW)

module in BeesyCluster allows for advanced search and matching of service descriptions to the user textual query [25] using the Apache Lucene engine. For the desired function expressed in the natural language (e.g. numerical integration etc.), BeesyCluster returns a list of services with numerical matching scores. Furthermore, the aforementioned balancing algorithm can choose services with greatest $clusterspeed/(1 + cost)$ ratios.

3) The balancing thread (Listing 1) divides the initial data into a predefined number of parts set by the programmer.

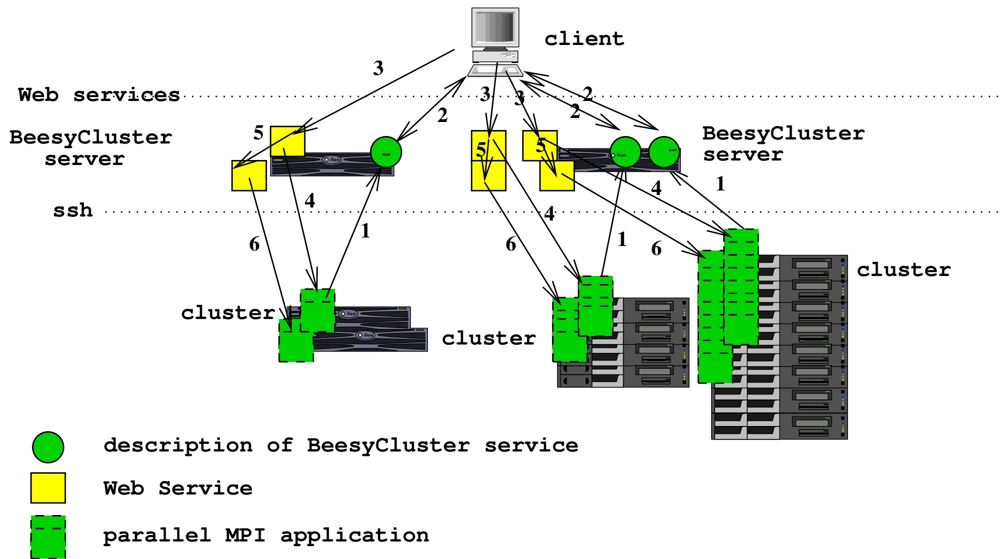

Figure 4: Interactions in the Solution

The thread also checks the speed of each cluster or server on which the services are installed. The balancing thread launches a new thread (Listing 2) for each part which then calls a Web Service in BeesyCluster.

4) The Web Service implementation within BeesyCluster starts an MPI application on a cluster remotely using ssh.
5) The framework launches a second instance of the Web Service for each service which is delayed compared to the first one to hide the communication latency upon termination of the first one.
6) As for the first one, the Web Service in BeesyCluster invokes the MPI application.
7) Then after the first application has terminated, another one is launched (Listing 1) while the one launched as second is still running. This way the cluster is kept busy. The framework proceeds until all data has been processed.

In the integration example considered in this paper the data part corresponds to a range to be integrated.

The algorithm can handle irregular problems taking different amounts of time on data sets of the same size. Examples of such applications include adaptive quadrature integration or $\alpha\beta$ searches [9].

IV. Experimental Results

A. Testbed Application and Environment

As an example, a parallel program for integration of a given function on the given range was used. Initially, the balancing thread divides the range into a predefined number of subranges. Then it launches services on clusters each of which represents a parallel MPI application and passes a single range $[a, b]$ and $accuracy$ which denotes width of polygons used to compute the integral.

The testbed environment used one BeesyCluster server installed at Faculty of Electronics, Telecommunications and Informatics which accesses in particular two clusters:

1) `holk` – a cluster with 288 dual core Itanium 2 processors connected with Infiniband, 2304 GB of RAM, 5.8 TB of disk space,
2) `KASK's cluster n01.eti.pg.gda.pl` – a cluster with 10 Dell PowerEdge 1850 nodes each with 2 dual core Intel Xeon processors, Infiniband interconnect, 40 GB of RAM.

For the tests, the client-BeesyCluster connection was at 2Mb/s while the BeesyCluster-cluster(s) connections were internal university links at 10Mb/s. Two nodes of `holk` and 9 nodes of `KASK`'s cluster were used as 11 virtual clusters on which the client called MPI integration applications as Web Services in BeesyCluster.

B. Hiding latency of the Web Service Calls

The goal of this test was to assess how running two parallel MPI simulations concurrently on one cluster, spawning one with a slight delay can mitigate communication latency. If only one application is run on a cluster (MPI-SA case), after it has terminated, the cluster is idle until a new one is started. If two are run, the second started slightly after the first one (MPI-2A case) the former can keep processors busy after the latter has terminated and a new application is being launched on this cluster. It happens at the cost of context switching between processes. Two nodes (`h001` and `h002`, each node with 2 dual core Itanium2 processors) of cluster `holk` were used. Both MPI-SA and MPI-2A ran with 4 processes per node. The range [1,1000000] with $accuracy$=3e-5 was used. Table I presents execution times for the aforementioned configurations. The variable is the number of ranges to integrate.

```
Vector pending; // with pending WorkRequestResult (cluster + data)
Vector finished; // with finished WorkRequestResult (cluster + data + result)

public void BalancingLoop() {
  // find/discover a certain number of capable services in BeesyCluster based on the
  // required functionality, costs and available budget

  // generate data
  int data_count=COUNT;
  type[] data;
  generate_data_and_store_in_data();

  // prepare data and start processing
  // there are two threads launching each service twice
  for(k=0;k<2;k++)
    for(i=0,j=0;j<services.length;j++) {
      WorkRequestResult wrr=new WorkRequestResult(j,data);
      synchronized(pending) { pending.add(wrr); } // add a request to a queue
      new Thread(this).start(); i++; // spawn a thread to get this data to process
    }
  WorkRequestResult wrr; int finishedcount=0; result=0;

  for(;(i<data_count) || (finishedcount<data_count);i++) { // main loop
    try {
        synchronized(finished) {
          if (finished.size()==0) finished.wait(); // notified by
        } // a notify() from method run()
    } catch (Exception e) { ... }

    // read data from the finished queue and then spawn another thread
    synchronized(finished) {
        wrr=(WorkRequestResult)finished.elementAt(0);
        finished.removeElementAt(0); }
    result=result.add(wrr.result); finishedcount++; // 'add' the result depending on
    // the problem
    if (i<data_count) {
        // now launch another thread for running the same service unless conditions have changed
        // - in the latter case discover next best service
        wrr.data=new data;
        synchronized(pending) { pending.add(wrr); } // add to the pending tasks
        new Thread(this).start(); } // spawn a thread to get this data to process
  } System.out.println("Result="+result);
}
```

Listing 1: Main Load Balancing Loop

```
public void run() {
  WorkRequestResult wrr;
  synchronized(pending) {      // first get a pending request
    wrr=(WorkRequestResult)pending.elementAt(0);
    pending.removeElementAt(0); }
  // process it on the cluster and update the result
  wrr.result=CallService(wrr.service,wrr.data);
  synchronized(finished) { // insert the result
    finished.add(wrr);  } //  into the finished queue
  // now wake up the main thread
  synchronized(finished) { finished.notify(); }
}
```

Listing 2: Thread Calling Services

Table I: Hiding Latency of Web Service Calls: Execution Time [s]

ranges	2	4	8	20	30	50
MPI-SA	64	68	76	86	117	156
MPI-2A		64	70	82	85	111

It is clear that launching two MPI applications (with 4 processes) each on a half of the initial range gives best possible results as there are just two calls (one to each cluster) and returning results. However, we obtain the same results for 4 ranges (2 calls to each cluster) run with concurrent applications on each node (MPI-2A). Here, the communication latency is practically hidden by the other applications running on the clusters. Moreover, increasing the number of ranges gives MPI-2A an edge over MPI-SA.

The larger the number of ranges the more perfect balance can be achieved especially if we consider potentially various speeds of clusters or additional load which may appear on some clusters.

For the following tests, the best execution times including

hiding communication latency using this technique are considered.

C. Execution Time under External Load

Table II: Execution Time under External Load [s], 11 clusters

ranges	18	36	72	144	288
execution time [s]	174	150	147	167	218

In this case, all 11 clusters were used while node `compute-0-0` of `KASK`'s cluster was loaded with computationally intensive processes launched by another user. Table II shows how the execution time of the integration application changes with the number of ranges. A small number of ranges means that the integration processes (on reasonably large subranges) on `compute-0-0` were considerably prolonged. On the other hand, a very large number of ranges resulted in a large overhead. 72 ranges turned out to be the best trade-off between the range counts tested. The range [1,1000000] with $accuracy$=3e-5 was used.

D. Scalability Results

For the first test, a single homogeneous cluster (`KASK`'s cluster was partitioned into up to 9 virtual clusters each composed of 4 processing cores. Execution times and speed-ups of the load balancing solution were measured and shown in Figures 5 and 6 given various numbers of ranges from the number of nodes up to 16 times the number of nodes used for the run. The range [1,1000000] with $accuracy$=3e-5 was used.

Similarly, the second test involved a heterogeneous environment with two nodes of cluster `holk` as 2 virtual clusters and up to 9 virtual clusters defined on the `KASK` cluster as in the previous experiment. Each processor of the first cluster is approximately 7 times faster than of the `KASK` cluster for the integration example. The balancing algorithm merges ranges for the faster cluster trying to obtain similar execution times for each requested range for each cluster. Figures 7 and 8 present the execution time and speed-up for integration of ranges [0,2e6], [0,5e6], [0,1e7] with $accuracy$=3e-5. All presented examples confirm good speed-ups (compared to a single fastest processor in the configuration) given the overhead of the client-BeesyCluster Web Service calls, BeesyCluster-cluster SSH communication and management of ranges.

V. Summary and Future Work

The paper presented an idea and an implementation of a framework for parallelization of computations in a dynamic master-slave fashion using distributed services installed in BeesyCluster and derived from MPI applications. The framework can use services published by various users and discover services at runtime for a desired application using service parameters such as cost. The algorithm is able to use the services to solve the problem on subsets of the initial data set and then merge into final results. An example of integration of a given function was presented and run on several modern clusters using services installed on them. Tests with and without external load for up to 11 clusters with varying speeds confirmed good scalability of the approach.

Future work includes testing more applications in the proposed scheme as well as tests of searching of services using ontologies.

Acknowledgments

Calculations were carried out at the Academic Computer Center in Gdansk, Poland. The work is sponsored by research grant MNiSW N516 383534.

References

[1] B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice Hall, 1999.

[2] R. Buyya, Ed., *High Performance Cluster Computing, Programming and Applications*. Prentice Hall, 1999.

[3] "OpenMP: Simple, Portable, Scalable SMP Programming," `http://www.openmp.org/`.

[4] W. Gropp and E. Lusk, *User's Guide for mpich,a Portable Implementation of MPIVersion 1.2.2*, 2001, `http://www-unix.mcs.anl.gov/mpi/mpich/`.

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam, *PVM Parallel Virtual Machine. A Users Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, 1994, `http://www.epm.ornl.gov/pvm/`.

[6] Official Crossgrid Information Portal, `http://www.crossgrid.org/main.html`, supported by Grant No. IST-2001-32243 of the European Commission.

[7] CLUSTERIX, *The National Linux Cluster*, `http://clusterix.pcz.pl`.

[8] B. Sotomayor, "The Globus Toolkit 4 Programmer's Tutorial," November 2005, `http://www.casa-sotomayor.net/gt4-tutorial/`.

[9] P. Czarnul, "Programming, Tuning and Automatic Parallelization of Irregular Divide-and-Conquer Applications in DAMPVM/DAC," *International Journal of High Performance Computing Applications*, vol. 17, no. 1, pp. 77–93, Spring 2003.

[10] R.D.Blumofe, C.F.Joerg, B.C.Kuszmaul, C.E.Leiserson, K.H.Randall, and Y.Zhou, "Cilk: An efficient multithreaded runtime system," in *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995, pp. 207–216.

[11] N. Karonis, B. Toonen, and I. Foster, "MPICH-G2: A Grid-Enabled Implementation of the Message Passing Interface," *Journal of Parallel and Distributed Computing (JPDC)*, vol. 63, no. 5, pp. 551–563, May 2003.

[12] R. Keller and M. M?ller, "The Grid-Computing library PACX-MPI: Extending MPI for Computational Grids," `www.hlrs.de/organization/amt/projects/pacx-mpi/`.

[13] P. Czarnul, "BC-MPI: Running an MPI Application on Multiple Clusters with BeesyCluster Connectivity," in *Proc. of PPAM 2007*, Springer-Verlag, Ed., vol. LNCS 4967, Poland, 2007.

[14] LAM/MPI Parallel Computing, `http://www.lam-mpi.org/`.

[15] P. Czarnul, "Integration of Compute-Intensive Tasks into Scientific Workflows in BeesyCluster," in *Proceedings of ICCS 2006 Conference,*. University of Reading, UK: Springer Verlag, May 2006, lecture Notes in Computer Science, LNCS 3993.

[16] E. Heymann, M. A. Senar, E. Luque, and M. Livny, "Adaptive scheduling for master-worker applications on the computational grid," in *GRID '00: Proceedings of the First IEEE/ACM International Workshop on Grid Computing*. London, UK: Springer-Verlag, 2000, pp. 214–227.

[17] A. E. D. Giusti, M. R. Naiouf, L. C. D. Giusti, and F. Chichizola, "Dynamic load balancing in parallel processing on non-homogeneous clusters," *Journal of Computer Science & Technology*, vol. 5, no. 4, December 2005.

[18] G. F. de Carvalho Costa, "Automatic dynamic tuning of parallel/distributed applications on computational grids," Ph.D. dissertation, Universitat Autonoma de Barcelona, May 2009.

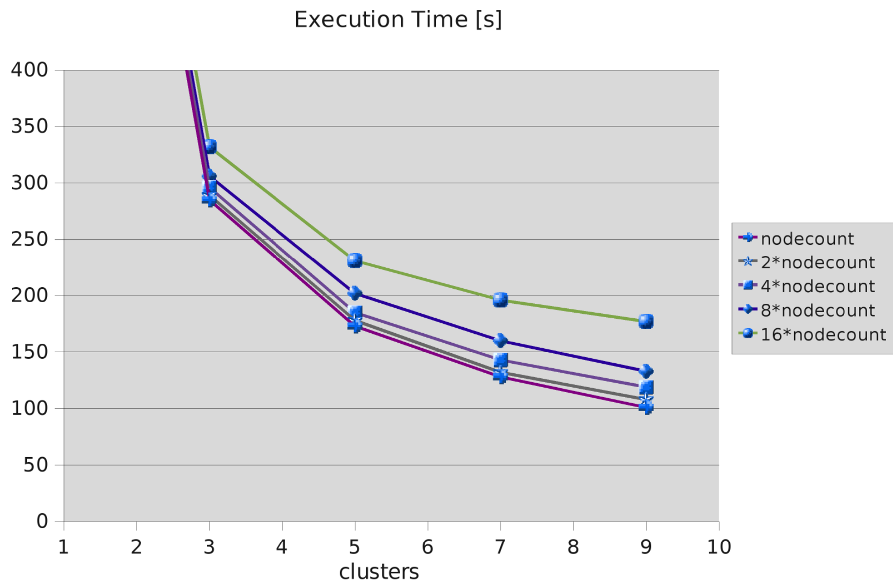

Figure 5: Execution Time in a Homogeneous Environment

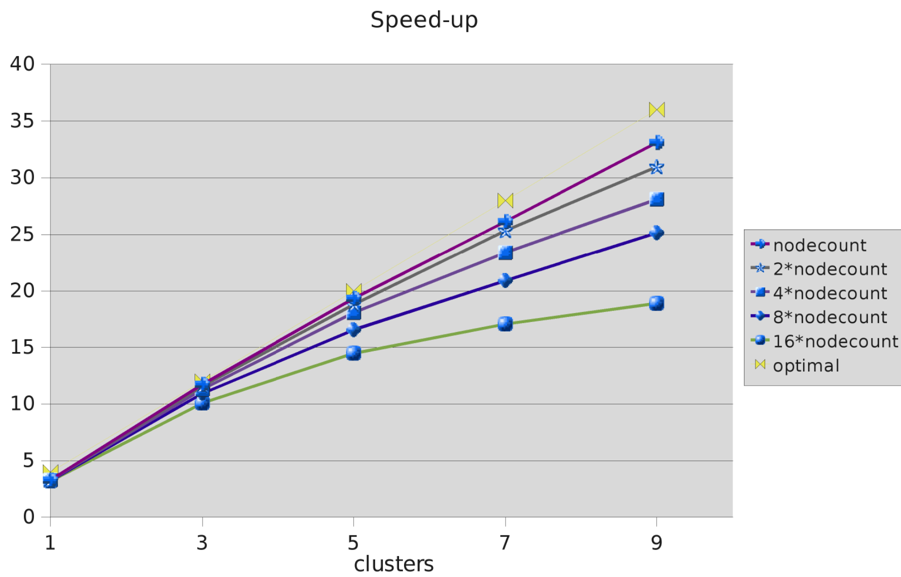

Figure 6: Speed-up in a Homogeneous Environment

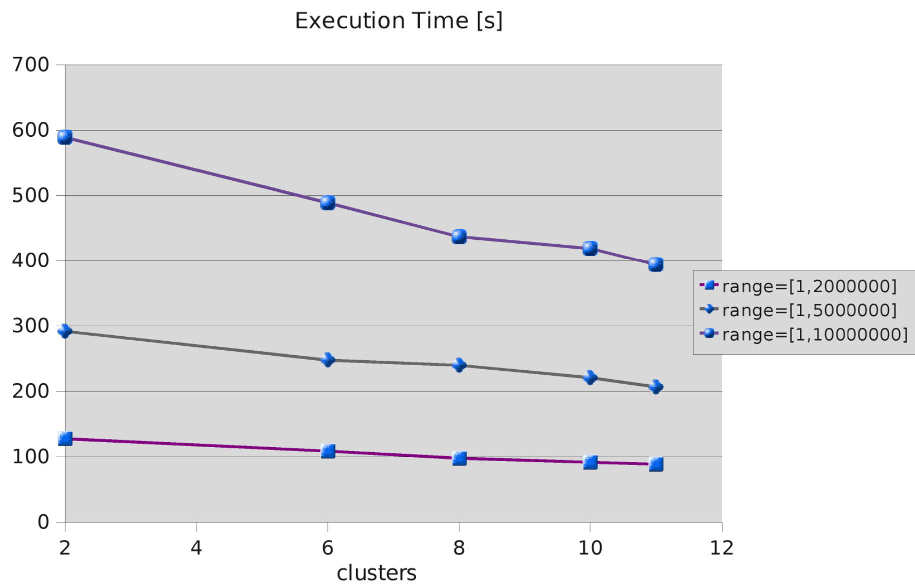

Figure 7: Execution Time in a Heterogeneous Environment

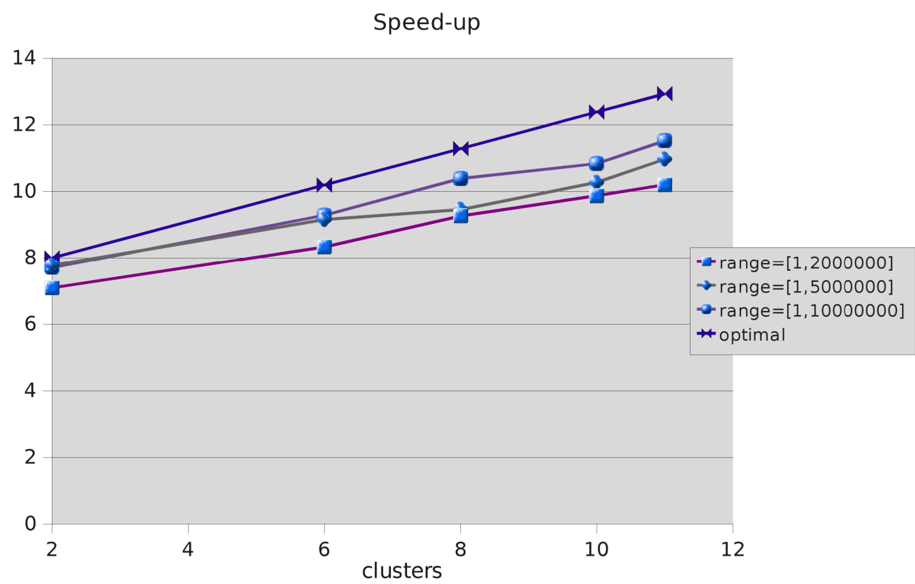

Figure 8: Speed-up in a Heterogeneous Environment

[19] R. Buyya, D. Abramson, and J. Giddy, “Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid,” in *Proceedings of the 4th International Conference on High Performance Computing in Asia-Pacific Region (HPC ASIA 2000)*.IEEE Computer Society Press, 2000, pp. 283–289.

[20] L. Dimitriou, “Distributed parallel computing with web services. a pivotal role on the back end,” SOA World Magazine, vol. 5, issue 2, February 2005, `http://soa.sys-con.com/read/48036.htm`.

[21] D. Puppin, N. Tonellotto, and D. Laforenza, “Using web services to run distributed numerical applications,” in *PVM/MPI*, ser. Lecture Notes in Computer Science, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., vol. 3241. Springer, 2004, pp. 207–214.

[22] D. Puppin, N. Tonellotto, and D. Laforenza, “How to run scientific applications over web services,” in *ICPP Workshops*. IEEE Computer Society, 2005, pp. 29–33.

[23] C. Boeres, A. P. Nascimento, V. E. F. Rebello, and A. C. Sena, “Efficient hierarchical self-scheduling for mpi applications executing in computational grids,” in *MGC ’05: Proceedings of the 3rd international workshop on Middleware for grid computing*. New York, NY, USA: ACM, 2005, pp. 1–6.

[24] M. Wurz and H. Schuldt, “Dynamic parallelization of grid-enabled web services.” in *EGC*, ser. Lecture Notes in Computer Science, P. M. A. Sloot, A. G. Hoekstra, T. Priol, A. Reinefeld, and M. Bubak, Eds., vol. 3470. Springer, 2005, pp. 173–183.

[25] P. Czarnul and J. Kurylowicz, “Automatic conversion of legacy applications into services in beesycluster,” in *Proceedings of 2nd International IEEE Conference on Information Technology ICIT’2010*, Gdansk, Poland, June 2010, in press.

[26] P. Czarnul, M. Bajor, M. Fraczak, A. Banaszczyk, M. Fiszer, and K. Ramczykowska, “Remote Task Submission and Publishing in BeesyCluster : Security and Efficiency of Web Service Interface,” in *Proc. of PPAM 2005*, Springer-Verlag, Ed., vol. LNCS 3911, Poland, Sept. 2005.

[27] E. Floros and Y. Cotronis, “Exposing mpi applications as grid services,” in *Euro-Par*, ser. Lecture Notes in Computer Science, M. Danelutto, M. Vanneschi, and D. Laforenza, Eds., vol. 3149. Springer, 2004, pp. 436–443.

[28] L. Zeng, B. Benatallah, M. Dumas, J. Kalagnanam, and Q. Sheng, “Quality driven web services composition,” in *Proceedings of WWW 2003*, Budapest, Hungary, May 2003.