Università di Pisa
Dipartimento di Informatica

Technical Report: TR-06-03

The cost of security in skeletal systems

M. Aldinucci[a,b] & M. Danelutto[b]

[a]ISTI – CNR Pisa
[b]Dept. Computer Science – Univ. of Pisa
Address: via F. Buonarroti 2, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

The cost of security in skeletal systems

M. Aldinucci[a,b] & M. Danelutto[b]

[a]ISTI – CNR Pisa
[b]Dept. Computer Science – Univ. of Pisa

Abstract

Skeletal systems exploit algorithmical skeletons technology to provide the user very high level, efficient parallel programming environments. They have been recently demonstrated to be suitable for highly distributed architectures, such as workstation clusters, networks and grids. However, when using skeletal system for grid programming care must be taken to secure data and code transfers across non-dedicated, non-secure network links. In this work we take into account the cost of security introduction in `muskel`, a full Java skeletal system exploiting macro data flow implementation technology. We consider the adoption of mechanisms that allow securing all the communications happening between remote, unreliable nodes and we evaluate the cost of such mechanisms. In particular, we consider the implications on the computational grains needed to scale secure and insecure skeletal computations.

Keywords: skeletons, parallelism, security, scalability.

1 Introduction

Algorithmical skeletons represent a good tradeoff between expressive power and efficiency in the field of parallel/distributed programming. An algorithmical skeleton is nothing but a known, parametric parallelism exploitation pattern. It can be customized by programmers providing suitable parameters in such a way it matches the needs of the particular application at hand. Usually, skeletons can also be nested in such a way that by nesting simple skeletons users/programmers can exploit very complex parallelism patterns. Typical examples of skeletons are task farms, modeling embarrassingly parallel computations, pipelines, modeling computations organized in stages, map, reduce and prefixes, modeling classical apply-to-all and sum-up data parallel computations and several flavors of iterator skeletons, modeling different loop schemas.

After being introduced by Cole [7], algorithmical skeletons led to the development of several *skeletal systems*, that is parallel programming environments

[0]This work has been partially supported by Italian national FIRB project no. RBNE01KNFP *GRID.it* and by the FP6 Network of Excellence CoreGRID funded by the European Commission (Contract IS -2002-004265).

exploiting the skeleton concept in different flavors: libraries, new languages, coordination languages and patterns. Examples of such programming frameworks implementing skeleton programming languages are P3L [4] and ASSIST [18, 2]. They are both programming languages designed and implemented by our group in Pisa in '91 and in 2000 respectively. eSkel [8, 5], Muesli [14], Skipper [16] and `muskel` [10] are examples of libraries providing parallel skeletons. The first two are implemented in C and C++ and run on top of MPI. They have been recently designed by Cole and Kuchen respectively. Skipper is implemented in Ocaml instead, runs on top of plain TCP/IP workstation networks and uses the same macro data flow implementation model of `muskel`. `muskel` is our pure Java/RMI skeleton library derived from Lithium [1] and it is the library we used to perform the experiments discussed in this paper.

Recently, with ASSIST and `muskel`, we afforded to target heterogeneous workstation networks and grids. When such programming environments are taken into account, several further problems have to be dealt with in the implementation of skeletal systems: on the one side, firewall and high network latencies have to be taken into account, and on the other side security issues have to be safely handled.

In particular, security issues arise when skeleton programs are executed on distributed architectures whose remote nodes and clusters are interconnected via public and/or non-dedicated network infrastructures. In this case both code (the one staged to remote nodes for the execution) and data (input data and computation results) are flowing to and from remote nodes through potentially insecure network links. Data and code crossing insecure links can be easily snooped or spoofed by persons that are not those actually managing to perform the parallel computation. The answer is then to secure the connections flowing through non-secure network links. But this has a cost that has to be paid both on sending and receiving machines (that must cipher and decipher (possibly serialized) code and data) and in terms of network bandwidth (ciphered connections may require more communications and/or differently sized messages to complete).

In this paper, we try to figure out the order of the costs in securing communications in a skeletal system. The skeletal system used is `muskel`, which we shortly describe in Section 2. Section 3 outlines the security related issues and how they can be addressed in the `muskel` skeletal system. Eventually, Sec. 4 will present and discuss some experimental results achieved with secure `muskel` system.

2 `muskel`

`muskel` is a full Java, skeleton based, parallel programming library. It can be used to run parallel skeleton programs on network of workstations that support Java and RMI. It provides the user with a set of fully nestable stream parallel skeletons (pipelines and farms). Skeletons are implemented by transforming the user supplied skeleton program into data flow graph. Then, each task to

```
public static void main(String [] args) {
 Compute filter = new Filter(); // first stage is filtering
 Compute render = new Render(); // second stage is rendering
  // as rendering is heavier than filtering, let's farm it out in the pipe
 Compute main = new Pipeline(filter, new Farm(render));
 Manager mng = new Manager(main); // this is a library manager
 mng.setContract(new ParDegree(10)); // request 10 remote PEs
 mng.setInputStream("inputImages.raw"); // input data here
 mng.setOutputStream("resultImages.dat"); // output data must go here
 mng.compute(); // start the computation. Upon return everything is done!
}
```

Figure 1: Sample `muskel` code

be computed is used to provide the input token to a copy of such graph. The fireable instructions in the graph are then scheduled for execution onto remote data flow interpreter nodes, and the result tokens computed are either used to fire new instructions or to be output as the results of the program execution. All these process is completely transparent to the user that only has to provide code such as the one in Figure 1. Here we assumed that two Java classes exist that process medical images coming from some kind of scanner (PET, CAT, MNR) to filter them and then to suitably render the filtered images. The stream of images to be processed is stored in a file and the result images will eventually be stored in another file. The user asks to compute the program using 10 remote data flow interpreter nodes. Furthermore, as he knows the rendering phase takes sensibly longer than the filtering one, he asks to execute in parallel the second stage of this pipeline computation, by writing the second stage of the pipeline as a farm.

`muskel` uses `Manager`s to manage computations. The manager takes a skele-

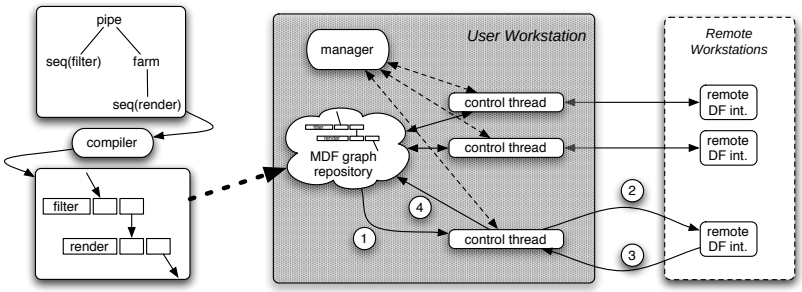

Figure 2: `muskel` functioning

ton program, input and output filenames and a performance contract (the parallelism degree, in this case). Then it arranges to discover and recruit a suitable number of remote interpreter nodes and forks a control thread for each one of the recruited interpreters. The control thread enters a loop. In the loop body, it fetches a fireable instruction from the MDF graph repository ①, delivers it to the remote interpreter ②, gets the results of the remote computation ③ and eventually either delivers the results ④ as tokens in the MDF graph or, in case they are final results, it delivers them to the output file. In case there is a problem with one of the remote interpreters (a remote node fault or a network problem) the control thread informs the manager and terminates. In turn, the manager tries to repair the situation by recruiting a new remote interpreter and putting back the uncomputed fireable instruction in the MDF graph repository.

The interpreters are launched on the remote nodes using a shell script once and forall (remote interpreters are plain Java remote objects running as standalone processes or as Java `Activatable` objects). They are specialized to execute the code of the application at hand by control threads forked by the manager. The control threads deliver to the interpreters the serialized version of the relevant `Compute main` classes just before starting the delivery of fireable MDF instructions. The process of recruiting remote interpreters can be executed in two different ways. In one case (version 1.0 of `muskel`), the addresses of the remote machines are retrieved by the manager from a text file hosting a ⟨machinename, port⟩ pair list. In another case (current version of `muskel`, 2.0), a peer-2-peer discovery protocol is started that eventually gathers answers from the remote machines were an interpreter was running hosting the same ⟨machinename, port⟩ info.

`muskel` has been tested on several configuration of networked workstations including plain, dedicated clusters (RLX Pentium III Blade chassis hosting 24 nodes), local network of different, production workstations (Pentium III to IV running Linux and Mac OS X Power PC G4 and G5 machines interconnected via Fast Ethernet and several Linux and Mac OS X laptops connected via wireless), geographical scale network hosting the same kind of machines in two sites separated by firewalls[1]. In all the cases, almost perfect scalability has been achieved, provided that suitably coarse grain programs are run. We showed that local network configurations (i.e. configurations hosting processing elements in a single LAN) scale well with skeleton code involving computations with a grain (ratio between the time spent in computing a remote DF instruction and the time spent in delivering the input tokens and retrieving the output tokens) around 100. Geographical scale networks, instead, required computations with a sensibly larger grain (1 to 2 orders bigger than the one scaling on the local network).

[1] ProActive [15] was used in this case to perform RMI call tunneling through `ssh`

3 Introducing security

When exploiting parallelism using nodes that are interconnected by public network links there is always the risk that communications are intercepted and relevant data is snooped by foreign, unauthorized people. Also, data can be snooped and substituted with other wrong or misleading data exploiting spoofing techniques, thus leading to incorrect computations. An even worst case concerns code. Take into account what happens in `muskel`: serialized code is sent to the remote interpreters that is then used to compute remotely the fireable macro data flow (MDF) instructions relative to the user skeleton code. If such code is changed, the remote nodes can be used to compute things they were not supposed to compute. Therefore is fundamental, in order to avoid both data and code problems, that 1) the access to the remote interpreters is authenticated in a secure way and 2) that the code itself is ciphered before being sent to the remote interpreters.

Authentication and code ciphering can be easily programmed using Java JSSE extensions, included in the JDK since version 1.4. Therefore, we decided to modify the `muskel` prototype to provide authentication, privacy and integrity in the communications happening among the control threads running on the user machine and the remote data flow interpreter instances running on the remote machines. In particular, we prepared a `muskel` version exploiting Java SSL library to perform communications involving remote processing nodes. SSL provides exactly authentication, using asymmetric keys, privacy, using symmetric session keys and integrity, using message digests, and overall represents a well known and assessed tool to secure remote communications over TCP. We then used the modified version of `muskel` (we'll refer to it as *secure* `muskel` from now on) to evaluate the impact of security on the raw performance of the skeletal system. Just to avoid interferences or difficulties in evaluating the experimental results due to any kind of additional mechanism, we stripped down the current `muskel` prototype by replacing the RMI remote interpreter access with plain TCP/IP sockets connections. In secure `muskel` we used the very same code modified just in the parts opening the sockets. Those parts dealing with the opening of plain TCP/IP sockets were modified to host the opening of SSL connections through proper calls to the SSL socket factories provided by Java 1.5. This process resulted in the implementation of two distinct versions of the base `muskel` engine able to compute in a distributed/parallel way sets of macro data flow instructions stored in the fireable instruction pool. The two versions have been used to evaluate the costs related to the introduction of security in the skeletal system, through the experiments described in the following Section.

4 Experiments with secure `muskel`

In order to figure out how the introduction of secure remote communications impact the execution of `muskel` programs we performed a set of experiments. All

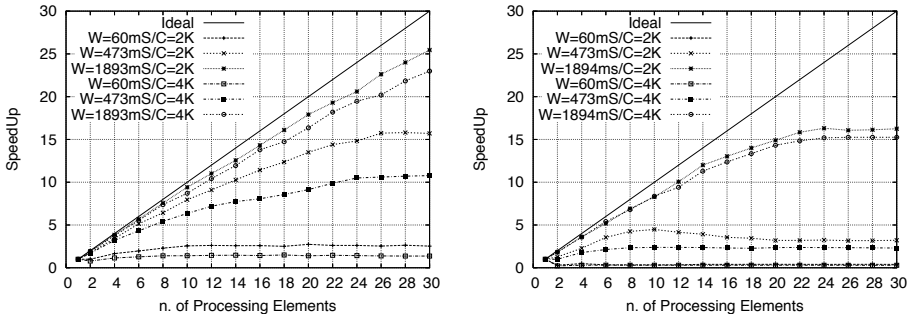

Figure 3: SSL vs. plain TCP/IP socket

the experiments were run on Fast Ethernet networks of Pentium III machines running Linux with a vendor modified 2.4.22 kernel. The first set of experiments measured the performance achieved when running the same `muskel` skeleton program onto a workstation network first using the original `muskel` prototype, with insecure communications, and then using the secure `muskel` prototype. We considered programs with different *computational grain*, i.e. programs whose macro data flow instruction have a different average computation to communication time ratio. In other words, we defined computational grain G as $G = \frac{T_w}{T_c}$, where T_w represents the time spent by a remote interpreter instance to compute the macro data flow instruction on the local data and T_c represents the time spent in transferring the input data to the remote interpreter instance plus the time spent getting back the computed results from the remote interpreter instance, and then we measured the performance of several programs with different values of G. Figure 3 shows the results we achieved. The left plot is relative to the original `muskel` runs and the right one is relative to the runs using the secure `muskel` version. In the legend, $W = x/C = yK$ means that the average T_w of macro data flow instructions was x and the amount of input data transferred to the remote interpreter to compute the instruction and the amount of output data retrieved from the remote interpreter was y Kbytes. The workstations used were dedicated to `muskel` runs, the programs were the same, the input data were the same also and therefore the only factor influencing the completion times is the usage of the SSL sockets. Both plots, the `muskel` and the secure `muskel` ones are actual speedup plots, rather than scalability plots: the point relative to 1 processing element is relative to the sequential execution of the macro data flow instructions on a single processor, rather than to the usage of just one remote interpreter instance. In this case, no communication overhead at all is counted in the execution time.

In order to better understand what's going on, we measured the raw communication bandwidth of `muskel` and secure `muskel`. Figure 4 left shows the bandwidth achieved in the two cases. The lower bandwidth of secure `muskel` is mostly due to the overhead introduced at processor level due to the ciphering/deciphering activity happening at the sending and receiving node. It is

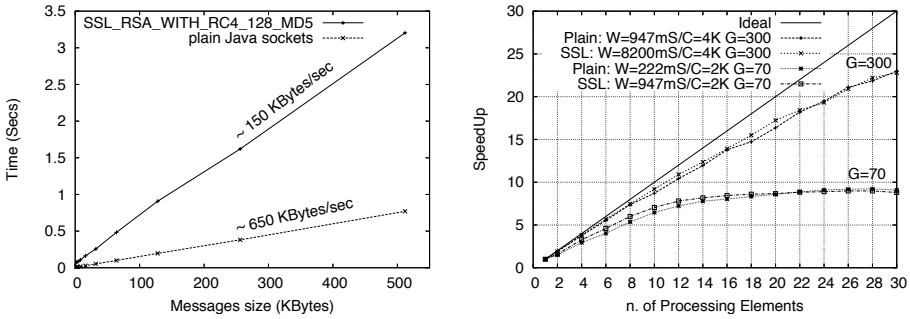

Figure 4: `muskel` vs secure `muskel` bandwidth (left, times include serialization time) and effect of grain (right)

only partially due to the initial key exchange handshake, which is performed once and for all, and to the slightly longer message encoding used in SSL, that happens to be less than 10%. From this measures we can conclude that the introduction of SSL impacts on the computational grain G needed to mask the longer communication times involved in `muskel` computations. Therefore we expect that coarser grain programs (that is programs with higher values of G) are needed to achieve good secure `muskel` performance figures.

We therefore run another experiment. We choose different values of G and run programs with that G value on both `muskel` and secure `muskel` prototype. As the T_c values depend on the communication library, we had to use larger data flow instructions in the secure `muskel` runs to get the same G with the same amount of data transferred to and from the remote interpreter instances. Figure 4 right shows the results achieved in this experiment. When the grain is high $G = 300$, both `muskel` and secure `muskel` scale pretty well (also in this case, the plot is relative to speedup, not to scalability). However, the secure `muskel` run required computations significantly longer (a more than 8 times longer T_w) in the macro data flow instructions than the standard `muskel` run, to get $G = 300$, due to the higher communication overhead of SSL sockets. When computational grain is smaller, however, both `muskel` and secure `muskel` stop scaling pretty early as shown by the $G = 70$ plots in the same Figure.

These experimental results clearly show that the costs involved in secure coding are definitely not negligible. Therefore, we must figure out how such costs can be optimized. We therefore considered that usually `muskel` skeleton programs are executed on a mix of local nodes (that is nodes belonging to the same LAN of the user machine running the `muskel main`) and non-local ones (that is nodes belonging to other, different LANs). In other words, users try to use first all the nodes available locally. These nodes can be reached with smaller communication delays with respect to non-local nodes, that is nodes that can be reached spending a substantial amount of time in routing messages across several routers and firewalls. Therefore computations using only local nodes perform better. This provided that the remote nodes are not sensibly faster than the

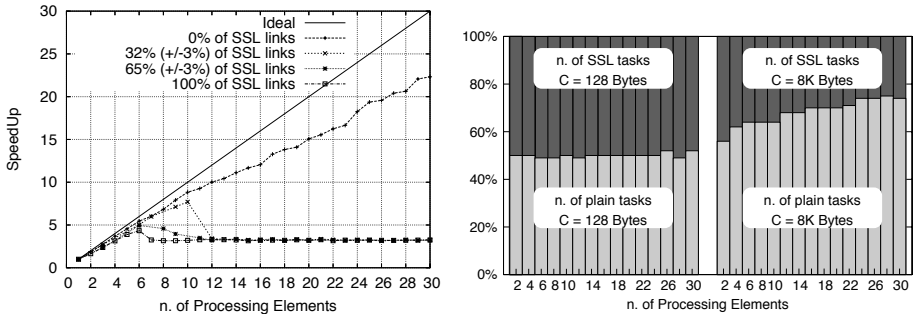

Figure 5: Mixed local nodes/non SSL and remote nodes/SSL `muskel` execution

local ones, at least. Now, in general, the local nodes happen to be operated in a controlled network environment. Then it is natural that those nodes are reached using non-secured, faster communication protocols. Both in `muskel` 1.0, the one picking up the remote node names from a ⟨machine, port⟩ file, and in version 2.0, the one looking up for remote interpreters via the peer-2-peer protocol, it is quite easy to figure out which remote interpreters behave to the same local network of the user node, just looking at the IP addresses after solving them with a `InetAddress.getByName` method call. Once the local nodes have been identified, the `muskel` manager can fork insecure communication control threads controlling those nodes, and secure communication control threads controlling the remote nodes.

We therefore run another experiment: we modified secure `muskel` to use SSL only with non-local nodes and to use plain TCP/IP sockets with the local nodes. Then, we run the same program on two clusters, with the same kind of machines, that is Linux machines with the same processors and the same amount of central memory. One cluster was in the same network of the user machine running the `main muskel` program. The other cluster was remote and therefore was managed by SSL `muskel` control threads. Actually, to remove the problem in the result analysis deriving from the different latencies in reaching local and remote nodes, we configured part of the local nodes as if they were non-local. Therefore, again, the only difference was in the usage of SSL `muskel` control threads rather than plain, non-SSL control threads. The results are shown in Figure 5. Figure 5 left shows the speedups achieved in runs of the same program performed using a variable mix of the distributed data flow interpreter instances placed on local machines and on remote machines. The speedups achieved in the mix runs are clearly smaller than the one reached in the local/insecure nodes only runs. However, `muskel` manager and control thread implement a self-adapting load balancing strategy. Each control thread only dispatches a new fireable MDF instruction when the results of the execution of the previous one have been received. Therefore "slow" remote interpreters get fewer tasks to be computed with respect to "fast" ones. Figure 5 right shows the measured percentage of fireable instructions (tasks) computed by each one of the remote interpreters. In

case the amount of data transferred to the remote interpreter instance is small (left part of the Figure), and therefore the weight of cipher/decipher is small, local and remote instances get more or less the same amount of tasks to be computed. However, when the amount of data transferred becomes significant (right part of the Figure), the remote interpreter instances get fewer tasks to be computed, due to the load balancing mechanism. Actually, this control mechanism was thought to solve load balancing in case of usage of heterogeneous workstations (different CPUs, different amounts of central store or even different operating systems) but it demonstrated very effective also in this case.

These experiments are not claimed to be definitive nor complete. But they clearly show two things: i) security has a high cost and ii) information about the remote interpreter machines can usefully be exploited to try to mitigate the overhead imposed by security mechanisms.

5 Related work

Currently available skeletal systems do not support any kind of security feature. The MPI libraries by Cole and Kuchen are thought to be run on MPI clusters, that are usually exploiting private, secure networks. Therefore the attention has been concentrated on other features related to efficiency and expressive power. ASSIST was designed to run on grids, either exploiting the Globus toolkit [13] or exploiting plain TCP/IP POSIX workstation mechanisms. In this latter case, it actually uses `ssh` and `scp` to perform remote commanding and data and code staging to and from remote machines. However, we never measured the impact of the usage of the `ssh/scp` tools. Recently, the Muenster university group leaded by Gorlatch introduced HOC [3, 11]. HOC (High Order Components) is a grid-programming environment jointly exploiting the skeleton technology and component technology. HOC provides predefined components providing the programmers with pipeline and task farm parallelism exploitation patterns. The implementation uses Web Services to manage grid related issues, such as data and code staging. At the moment, however, security issues are not yet taken into account in HOC although there are specifications to put security over standard XML/SOAP protocols used in web services [6].

More attention is paid to security issues in non-skeleton based grid programming system. The globus grid middleware [13] provides a full range of tools to handle security issues [19], for instance. And security is one of the key points to be addressed accordingly to the NGG reports [17]. Recently, in the framework of the CoreGRID European Network of Excellence [9], security has been considered an "horizontal issue" that is an issue to be considered in all the Institutes of the network, and a nice survey of security grid related issues has been produced [12]. We considered the results of all this experiences before investigating the impact of security in skeletal systems.

6 Conclusions

We discussed the cost of introducing security features into a skeletal system. The skeletal system taken into account is `muskel`, our full Java skeleton based parallel programming library. We modified the implementation in such a way the communications happening between the remote machines were insured with authentication, privacy and integrity, by exploiting SSL. We evaluated the cost of such operation. Then we showed how the exploitation of the information available at run time can mitigate its high cost. As security is a fundamental issue in highly distributed systems, such as multi cluster and grid architectures, we think this could be considered an interesting contribution to the skeletal system implementation technology.

References

[1] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003. Elsevier Science.

[2] M. Aldinucci, A. Petrocelli, E. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of grid-aware applications in ASSIST. In *11th Intl Euro-Par: Parallel and Distributed Computing*, volume 3648 of *LNCS*, pages 771–781, Lisboa, Portugal, August 2005. Springer Verlag.

[3] Martin Alt, Jan Dünnweber, Jens Müller, and Sergei Gorlatch. HOCs: Higher-order components for grids. In Vladimir Getov and Thilo Kielmann, editors, *Component Models and Systems for Grid Applications*, CoreGRID, pages 157–166. Springer Verlag, June 2004.

[4] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A Structured High level programming language and its structured support. *Conc. Practice and Experience*, 7(3):225–255, 1995.

[5] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with eSkel. In *11th Intl Euro-Par: Parallel and Distributed Computing*, volume 3648 of *LNCS*, Lisboa, Portugal, Aug. 2005. Springer Verlag.

[6] Geuer-Pollmann C. and Claessens J. Web Services and Web Service Security Standards. *Information Security Technical Report*, 10(1):15–24, 2005. Elsevier.

[7] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Parallel and Distributed Computing. Pitman, 1989.

[8] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[9] Coregrid home page, 2006. http://www.coregrid.net.

[10] M. Danelutto. QoS in parallel programming through application managers. In *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-based processing*. IEEE, 2005. Lugano.

[11] Jan Dünnweber and Sergei Gorlatch. HOC-SA: A grid service architecture for higher-order components. In *IEEE International Conference on Services Computing, Shanghai, China*, pages 288–294. IEEE Computer Society Press, September 2004.

[12] Alvaro A. et al. Survey Material on Trust and Security, 2005. Deliverable D.IA.03, CoreGRID, www.coregrid.org.

[13] Globus web site. http://www.globus.org.

[14] H. Kuchen. A Skeleton Library. In *Euro-Par 2002, Parallel Processing*, number 2400 in LNCS, pages 620–629. Springer Verlag, August 2002.

[15] ProActive home page, 2005. http:// www-sop.inria.fr/oasis/ ProActive/.

[16] J. Sérot and D. Ginhac. Skeletons for parallel image processing : an overview of the SKiPPER project. *Parallel Computing*, 28(12):1785–1808, Dec 2002.

[17] D. Snelling and K.-Jeffrey et al. Next Generation Grids 2 – Requirements and Options for European Grids Research 2005-2010 and Beyond, 2004.

[18] M. Vanneschi. The Programming Model of ASSIST, an Environment for Parallel and Distributed Portable Applications. *Parallel Computing*, 12, December 2002.

[19] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, and K. Czajkowski et al. Security for Grid Services. In *Proceedings of 12th IEEE International Symposium on High Performace Distributed Computing*. IEEE Computer Society Press, 2003.