A distributed search infrastructure for Statistical Disclosure Control on a Grid

K. R. Mayes[1], M. J. Elliot[2], A. M. Manning[1], D. Haglin[3] and J. R. Gurd[1]

[1]Centre for Novel Computing, School of Computer Science, University of Manchester.
[2]Centre for Census and Survey Research, University of Manchester.
[3]Department of Computer and Information Sciences, Minnesota State University.
Email address of corresponding author: ken@manchester.ac.uk.

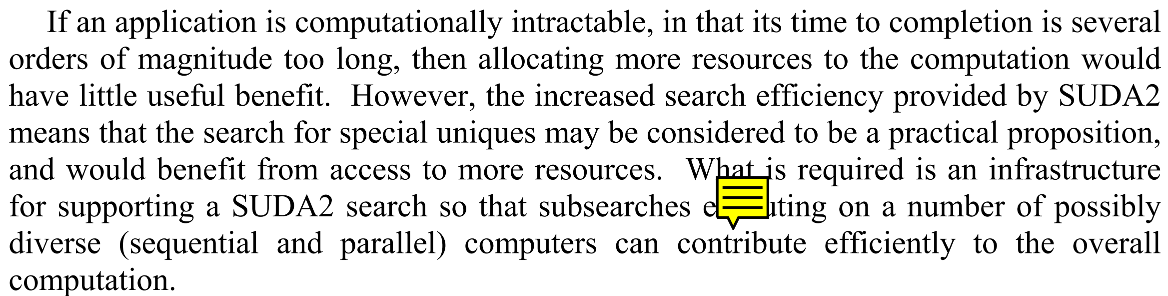
Abstract. Statistical disclosure control is concerned with preventing the release of data that might allow an individual population unit to be identified, or for new attributions to be made about specific population units. However, the search time for identifying records can grow exponentially as record size increases, thus severely restricting the usefulness of such methods. This paper presents a prototype infrastructure that allows SUDA2, an algorithm for efficiently identifying risky records in microdata, to execute on a heterogeneous network of computers. The search space is divided into subspaces, each of which can be searched on a separate computer. The work involved in searching a given subspace depends on the nature rather than the amount of data, and so is not always predictable. This means that the infrastructure must be able to act adaptively, particularly as resources may vary in their capabilities. The infrastructure is able to combine a master-worker computational structure with work stealing and checkpointing to attempt to keep resources busy for as long as possible. The infrastructure is able to reduce execution time of searches.

1 Introduction

Accurate assessment of disclosure risk is a necessary precursor to efficient statistical disclosure control (SDC). This assessment is often based on the notion of uniqueness, where a record is unique within the dataset (or population) on a specified set of characteristics. Identifying riskier or “special” uniques within microdata is a key element in the disclosure risk assessment associated with the release of such data (Elliot et al., 2002). The process of comprehensively locating and grading special uniques within microdata involves considering all combinations of data items, directly or indirectly, for uniqueness. Such combinatorial search can lead to an explosion in the amount of searching required and can grow exponentially with the number of items (Garey and Johnson, 1979). This combinatorial problem severely restricts the usefulness of the results of these computations, and means that even limited searches can take periods of days to run to completion. However, this problem may be ameliorated by two compatible

techniques: to increase the efficiency of the search algorithm, and to give access to more computing resources.

By developing efficient combinatorial search algorithms, it is possible to reduce the amount of computation required. In general this algorithmic approach involves exposing criteria for pruning out combinations of data that do not need to be searched. SUDA and SUDA2 (Elliot et al. 2002; Manning and Haglin, 2005) are algorithms for the detection of special uniques. The HIPERSTAD project[1] is currently putting SUDA2 into the realm of high performance computing – specifically implementing SUDA2 on highly parallel individual supercomputers using parallel execution models such as MPI[2].



If an application is computationally intractable, in that its time to completion is several orders of magnitude too long, then allocating more resources to the computation would have little useful benefit. However, the increased search efficiency provided by SUDA2 means that the search for special uniques may be considered to be a practical proposition, and would benefit from access to more resources. What is required is an infrastructure for supporting a SUDA2 search so that subsearches executing on a number of possibly diverse (sequential and parallel) computers can contribute efficiently to the overall computation.

2 The SUDA2 algorithm

The SUDA2 algorithm operates on data consisting of records, or *rows*, of attribute values, termed *items*. SUDA2 conducts a comprehensive search for unique item sets (*uniques*) without any unique subsets (i.e. *minimal uniques*) in order to avoid the use of redundant information, and to keep the classification process as focused as possible. The smaller the number of attributes contained in a unique pattern the more `risky' it is considered to be (Elliot et al. 2002). SUDA2 is generally applied to a *sample* of data and therefore the minimal uniques are referred to as *Minimal Sample Uniques* (MSUs).

The process of finding MSUs is known to be a difficult problem. Therefore the best that can be achieved in terms of reducing execution time is to find effective heuristics that prune large portions of the search space. A detailed definition of the SUDA2 algorithm was given in Manning and Haglin (2005). The following presents a more operational view.

The basis of the algorithm (given in pseudo-code in Section 2.5) is to drive the search by ranking the items on the frequency of their occurrence in the data, and then to recursively search the data on the basis of these rankings. The data table is scanned to produce a list of all items appearing at least once in table. This list of items is then sorted in order of increasing repetition count. The *rank* of an item is index of that item in the sorted list. The search consists of traversing this ranked-item list, from the item with lowest repetition count (at the start of the list) to the item with highest repetition count (at the end of the list). For each item in the ranked-item list, the algorithm finds all the MSUs containing that item and other items whose ranks are higher. Thus, as the item-rank list is traversed, each rank effectively forms an *anchor* for a search through a subspace in the data (termed *anchorRank* Section 2.5). It is this ability to divide the

[1] http://www.cs.manchester.ac.uk/cnc/projects/hiperstad.php

[2] http://www.mpi-forum.org

entire search space into subspaces that facilitates the distributed implementation, given that the data is replicated.

Given this basic approach, the SUDA2 algorithm relies upon a set of four properties that must hold true for every MSU. These properties are more fully described in Manning and Haglin (2005).

2.1 Support Row Property

The Support Row Property is a major criterion for testing whether a unique pattern of items is in fact an MSU. By definition, the subsets of an MSU must be non-unique (otherwise the MSU would not be minimal), so all subsets of an MSU must occur in other rows of the data. The *support rows* of an MSU are those rows whose items differ from the MSU by just one item. This means that if some row contained an MSU of say 4 items ({a,b,c,d}), there must be at least 4 support rows, each with 3 items, and each missing a different item from the MSU ({a,b,c, ,} ; {a, ,c,d} ; {a,b, ,d} ; { ,b,c,d}). This is where rank and repetition count are useful – a set of items in a row can only be an MSU of size 4 if each of its 4 items occur in at least 4 rows (i.e. the MSU row plus 3 of the support rows). The Support Row Property can be used to provide an upper bound on the size of candidate MSUs that are to be checked, and can give an efficient way to prune significant areas of the search space if several items have low repetition counts.

2.2 Uniform Support Property

This provides a simple criterion for testing the Support Row Property. A unique itemset that contains the same item in all of its support rows cannot be an MSU, since the Support Row Property does not hold. That is, there must be at least one support row that does not contain the otherwise common item. This property, again, provides a powerful tool for pruning redundant parts of the search space for MSUs

2.3 Recursive Property

The Recursive Property is central to the operation of the algorithm. It is used to identify *candidate* itemsets that might be MSUs. As can be seen in Section 2.5, anchor items are progressively removed from the main table, in recursive invocations of the algorithm, to produce ever smaller subtables, checking at each level of recursion for items that are unique in that subtable. An item that is unique in a subtable, combined with the items that have been removed in producing the subtable, form a candidate MSU. Each candidate MSU can then be checked, as the recursion unwinds, for being an MSU in the parent subtables. The Recursive Property of MSUs helps define the boundaries of the search space for MSUs by providing a clear indication of the maximum size of candidate MSUs. That is, the Recursive Property allows candidate MSUs to be detected by "growing them" from single unique items.

2.4 Perfect Correlation Property

Two items that appear in the same set of rows are said to be *perfectly correlated.* If two items are perfectly correlated then they cannot coexist in an MSU, as the Support Row Property does not hold. However, the knowledge that two items are perfectly correlated can be used to prune the search space, since only one member of a set of perfectly correlated items need be "active" in the search. The corresponding MSUs for

the ignored items are added at a later stage. However, the procedure required to check for perfect correlation between all items with the same repetition count is combinatorially explosive, and has the potential to become counter-productive in terms of execution time. SUDA2 balances this conflict by storing one of each adjacent pair of perfectly correlated items.

2.5 The SUDA2 algorithm

The SUDA2 algorithm can be summarized in pseudo-code. As can be seen on line 10, the ***SUDA2code*** is recursively invoked. The initial input is the entire data and the maximum MSU size required. Recursive calls are made with subtables generated by selecting rows with the current *anchorRank,* and decrementing *maxk*.

```
Input:  Table = input dataset with n rows and m columns of integers
Input: maxk = maximum size of MSU in the search
Returns:  A set of MSUs for dataset Table

SUDA2code (Table, maxk)
1.  compute rankItemList  = ordered list of all items in Table
2.  defer perfectly correlated items

3.  if maxk == 1
       /* stop recursion */
4.     Return all items of rankItemList that appear only once in Table
5.  else
6.     MSUSet = Ø
       /* main loop – traverse the rankItemList for this invocation.  Each rank is an
          "anchor" for a subspace */
7.     for anchorRank in rankItemList
8.        i = item in rankItemList whose rank is anchorRank
9.        table_i = subset of Table consisting of only rows of Table holding item i
          /* recursively generate subtables to find the candidate itemsets for i.
             SUDA2 uses rank and repetition counts to limit the potential search
             space */
10.       CandidateSet_i = recursive call to SUDA2code(table_i, maxk-1)
11.       for each candidate ItemSet ∈ CandidateSet_i
12.          if (ItemSet ∪ {i}) is an MSU in Table
13.             MSUSet ← MSUSet ∪ ( ItemSet ∪ {i})
14.             add MSUs for deferred perfectly correlated items to MSUSet
15.          endif
16.       endfor
17.    endfor
18.    return MSUSet
19. endif
```

The form of the SUDA2 algorithm as shown is essentially sequential. However, each iteration of the main for-loop can be executed independently, and thus concurrently,

providing the data is replicated. This main for-loop, as the algorithm iterates through the *anchorRanks* in the top-level (i.e. recursion level zero) *rankItemList*, can be executed in a parallel, distributed fashion.

3 The GridHIPERSTAD prototype infrastructure

There are two sets of concerns in this work:

1. Application concerns: to split up the overall job representing the entire search into subjobs, then to execute the subjobs, and then to assemble the results.

2. Infrastructure concerns: to provide mechanisms for dynamic deployment of subjobs, and to develop policies to maximise the usage of processors by using these mechanisms.

It is important that the design and prototype implementation of the infrastructure attempt to separate these concerns, so as to facilitate its re-use with other applications.

3.1 The SUDA2 implementation

A SUDA2 application can be represented as a scatter-gather structure, as shown in Figure 1. The search can be distributed by splitting the overall search space into subspaces which can each be individually searched. At its simplest, each subspace corresponds to one *anchorRank* in the main for-loop of the algorithm.

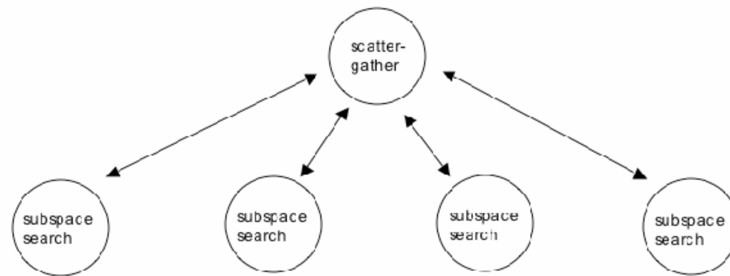


Figure 1. The SUDA2 search space can be split into subspaces, each of which can be searched on a different computer, with the results delivered to a scatter-gather node representing the entire search. Each subspace corresponds to a top-level anchor (see text, Section 3.1).

There are two existing implementations of the SUDA2 algorithm, both written in C++, one for sequential execution, and the other for parallel execution using MPI. Both of these codes have been modified to provide two sorts of component: a scatter-gather component, and a subspace search component.

The scatter-gather component simply does sufficient analysis of the target data to identify the set of *anchorRanks* in the top-level *rankItemList*. These top-level *anchorRanks* will henceforth be termed “anchors”. Each anchor represents a subspace to be searched. The subspace search components are able to take a single anchor and to search the subspace associated with that anchor. The current SUDA2 algorithm requires that the data is replicated on each participating machine, and so the anchor values represent the same subspaces on all subspace search instances. After generating the

subspaces to be searched, the scatter-gather component waits for the results of the subspace searches to arrive, and then delivers the results of the whole search.

The subspace search component is modified to search one subspace at a time. The search code makes calls out to the infrastructure, and the infrastructure is responsible for delivering the next anchor to the search code. The subspace search code will only run when an anchor is available.

It is possible to specify subsearches at the level below the anchor level, in order to reduce the work granularity, by dividing subspaces. This capability is used in a further modification to the SUDA2 subspace search code, to add a "checkpointing" capability. This allows subspace searches to be stopped and restarted. Here, a search can be interrupted at some arbitrary level of recursion, and is represented as a series of anchor values, say, <5.2.3.7.4>. Once stopped, a subspace search can be divided further if necessary, then redeployed and restarted.

3.2 The prototype infrastructure

The infrastructure architecture is shown in Figure 2. It is based on the PerCo performance control system that is being used with scientific simulation applications (Mayes et al., 2005). The scatter-gatherer of Figure 1 is shown as SSG in Figure 2. The role of this SSG is to generate the search subspaces, as a set of anchor values. Each anchor value represents an individual work unit or subjob, which can be executed by a subspace search component (SSS in Figure 2).

However, the policy of deploying the subjobs to the SSSs is an infrastructure concern, not an application concern. So, closely associated with the SSG is an Application Performance Steerer (APS), which receives the anchor value subjobs from the SSG, and is responsible for determining policy in deployment of the subjobs. The APS maintains a global work "queue", though this list of subjobs does not necessarily function as a pure queue (i.e. subjobs do not have to be taken only from the head of the list).

Subjobs are passed to the APS in an application-independent representation, so that the APS is not tied to particular applications. The infrastructure design allows a scatter-gatherer to utilise application-specific knowledge to associate a "work index" with each subjob, so that the APS can make decisions about deployment of work. In the case of SUDA2, the SSG can assign work indices using heuristics based on the likely distribution of work across the range of anchors. That is, most work will typically reside in subspaces found towards the middle of the *rankItemList* (see Section 2.5). However, this heuristic provides an approximate guide only, and does not indicate the potential large differences in work content between neighbouring anchors, that introduce unpredictability into the SUDA2 execution.

Each subspace search component (SSS) is wrapped in an interface (compif), which is intended to de-couple the SSS from the control infrastructure. This interface wrapper links the SSS with a Component Performance Steerer (CPS), which monitors the behaviour of its SSS and mediates between the SSS and the APS.

Mechanisms for both master-worker (e.g. Goux et al., 2001) and work-stealing (e.g. Blumofe et al., 1996) paradigms, for the flexible allocation of work, are provided by the infrastructure. In master-worker mode, each CPS requests work from the APS, and this work is allocated from the global work queue according to some scheduling algorithm in

the APS. These allocated work units are added by each CPS to the tail of its local work queue. At some point determined by the APS, the CPSs can be instructed to begin obtaining work from each other by work stealing. The local work queues are "double ended", in that work is removed from the head for local execution, but the work stealing mechanism removes work from the tail of the work queue.

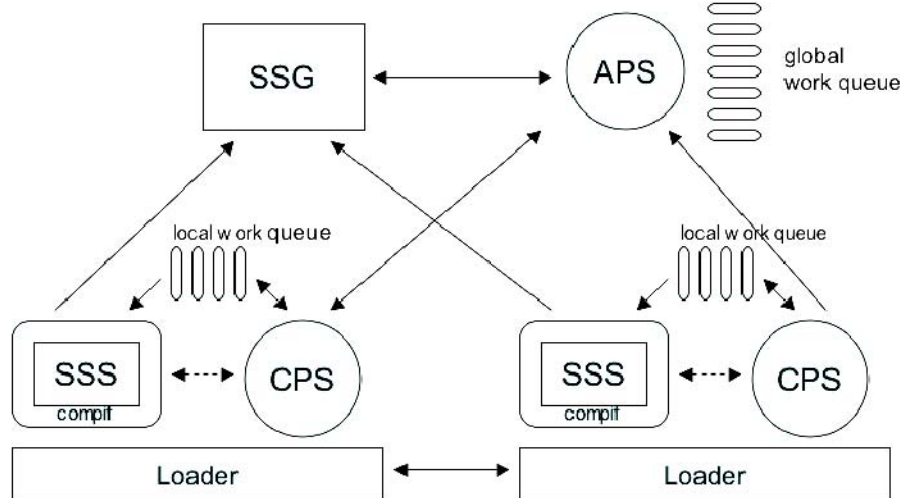


Figure 2. The infrastructure architecture is able to move work between machines (see text for description). SSG scatter-gather component; SSS subspace search component.

Underpinning the infrastructure are the Loaders, which are responsible for launching each SSS and CPS at boot-time, and for transferring the subjob anchor values between CPS instances during work stealing or redeployment.

The prototype infrastructure, indicating the Unix process structure and inter-component communications, is shown in Figure 3. Note that the infrastructure design and implementation is identical for the sequential and MPI versions of the SUDA2 algorithm. This is achieved by using only one of the MPI parallel processes to invoke the infrastructure. The Loader and SSG-APS executables are currently boot-loaded using Globus[3] facilities.

3.3 Strong work stealing and checkpointing

The aim of the system is to keep all machines busy, searching subspaces, for as long as possible. However, there are two problems:

1. The platforms hosting the subsearch code are heterogeneous and non-dedicated and so have differing processing capabilities.
2. The amount of work in a search subspace is dependent on the nature rather than the amount of data, and so may involve an unpredictable amount of work.

These problems require the system to have adaptive capabilities beyond "normal" work stealing. Although work allocation and work-stealing will suffice to keep machines busy by transferring work between global and local work-queues, a problem may arise towards the end of execution. That is, a large subspace search may have come to reside

[3] http://www.globus.org

on a slow machine, with the result that, since there is no other work to be done, other, faster, machines must remain idle. The entire search cannot complete until that final subspace search has completed.

It has been recognised that a central work-allocation policy of allocating the smallest work units towards the end of execution can reduce the effects of this potential final-stage load-imbalance (e.g. Yang and Casanova, 2003). However, with coarse grain work units,

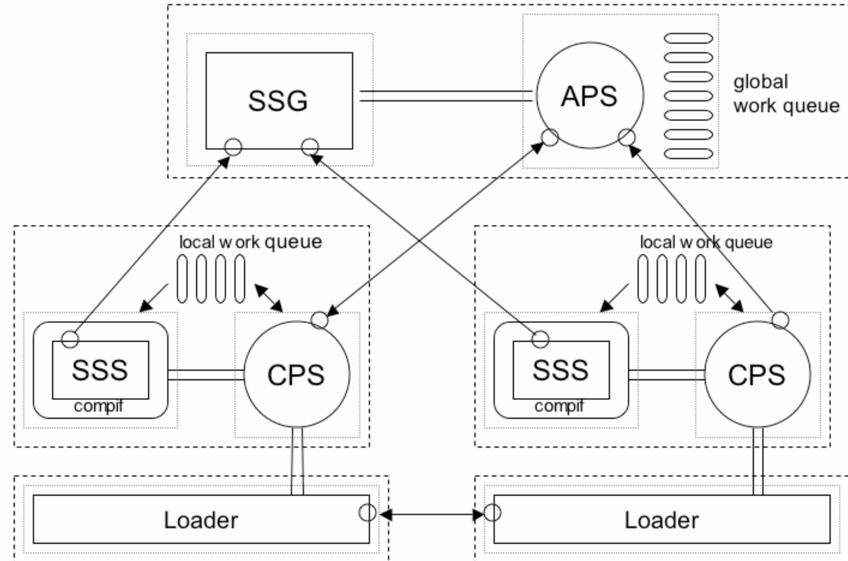


Figure 3. Current implementation of the infrastructure and SUDA2. Dashed boxes represent entities which reside in the same executable. Dotted boxes represent separate Unix processes. Small circles represent socket connections, and parallel lines represent pipes. The SUDA2 subspace search code (SSS) can be a sequential or MPI implementation

as here, the execution time for the entire search may be determined by the execution time of the largest subspace search. Although it may be possible for system to guarantee that the largest subspaces are executed on the fastest machines, it may be that lack of predictability in the application or platform causes pathological cases. In general therefore, a more adaptive approach may be required. Two related techniques are currently being investigated to solve this problem:

1. Strong work stealing. A CPS on a fast machine can cause a slow SSS to checkpoint its search, and redeploy the remainder of the search to a faster machine.
2. Subspaces can be divided into finer-granularity subspaces for distribution across machines.

4 Initial results and current status

The system is currently being investigated by executing on connect-4 data from Merz and Murphy (1996)[4]. This dataset has the characteristics of data with which SUDA for

[4] The connect-4 dataset contains all legal 8-ply positions in the game of connect-4 in which neither player has won yet and in which the next move is not forced. There are 67557 rows and

SDC would typically deal, in that the dataset consists of equal length records of discrete integer values, and fits into main memory. Moreover, the dataset is freely available and there are no confidentiality constraints on its use. This dataset is being used here because a search for MSUs of maximum size 15 items is of reasonable duration for experimentation. That is, when executing on a single 3.2GHz linux PC (the fastest machine used in the tests), the search completed in 398 minutes. This execution time has been shown to be reduced by executing the subspace searches concurrently, using more computers. By launching subspace searches concurrently on to a total of twelve machines of different capabilities, the sequential completion time of over six and a half hours was reduced to around 1 hour. The machines used were:

- ten linux PCs (with a mix of speeds from 870Mz to 3.2GHz), each running a sequential implementation of SSS,
- An SGI Origin machine (using 9 processors of 400MHz) and a Solaris machine (using 4 processors of 444MHz each) each running an MPI implementation of SSS.

A precise evaluation of factors affecting performance is not yet available, so the figures are intended to give only an indication of the likely capabilities of the system. Work was allocated randomly by the APS, in units of ten subspaces, to each of the SSSs when required. When the global work queue was empty, work was obtained by random work stealing by CPSs from other local work queues. Using this mechanism, in five experiments, the execution time was found to be between 65 and 93 minutes (mean 76 minutes). This wide variation is due to the chance of a large subspace search residing on a slow machine once the local work queues become empty and work stealing ceases. By using strong work stealing and dividing checkpointed work, more machines were able to perform searches for longer, and the execution time was reduced to between 55 and 64 minutes (mean 61 minutes).

Work is in progress to further investigate the usefulness of other policies that can be implemented in the infrastructure, both for this, and for other applications. One important point to note is that these experiments have demonstrated that different SSS implementations (i.e. sequential and MPI) can cooperate, and contribute to the same search.

5 Future work

5.1 Grid issues

The main motivation behind this work has been to provide an infrastructure which can adapt so as to reduce the execution time of a search. The prototype infrastructure described here has been developed on a local network of heterogeneous machines. Such a set of heterogeneous computers on a network may not actually constitute a “Grid”; it can be argued that in order to be considered as a true Grid, a network must cross administrative domains. However, there are problems encountered in developing, running and assessing distributed applications using such a wide-area Grid. For example communication ports may be routinely blocked by firewalls, and Grid resources, which

---

43 columns (one for each of the 42 connect-4 squares together with an outcome column – *win*, *draw* or *loss*).

should be co-scheduled simultaneously, may actually be allocated at different times from local batch queues. The organisational and practical problems in joining together separate Grids in UK and US, to run a large scientific simulation, have been found to be non-trivial (Chin et al., 2005). Ideally, from a purely computational, as opposed to a security, viewpoint, it would be useful to use the resources of the UK National Grid Service[5]. However, "back end" NGS computational resources are allocated via local batch queues, and cannot be named or accessed directly, meaning difficulties with socket-based communications.

Grid services (or more correctly Web-Service Resources) are intended to overcome the problems of location transparency and integration in Grids (Foster et al., 2004). It is possible to use services to implement master-worker applications such as SUDA2. A service-based, master-worker, application, for the analysis of astronomical data, has been described by Harbulot et al. (2006). This master-worker architecture was able to tolerate the problems arising from unreachable back-end machines and batch queue scheduling of the NGS. It is intended to migrate the SUDA2 infrastructure into this service-based setting, and so utilise general Grid environments. However, it is worth noting that the use of Web Service protocols for non-commercial applications is not as straightforward as might be supposed, due to the immaturity of the tools and standards (Harbulot et al., 2006).

From a purely computational viewpoint, service-based implementations may enable the use of multi-domain Grids for SDC. However, with SDC there are security issues arising from accessing confidential data on a public Grid. One obvious solution is to avoid general Grids and restrict the use of SDC to a local network of computers residing behind a firewall; that is, to use networks similar to the development environment used here. On the other hand, a multi-domain Grid implementation of the SUDA2 application would require replication of the search data at all participating sites, with messages passing between sites being encrypted. The encryption and decryption overhead would not be high compared to overall search time.

5.2 Fault tolerance

There are features which could be added to the existing prototype infrastructure to increase its usefulness in heterogeneous local network environments. In a context where many machines are being utilised, particularly where these are non-dedicated, there is always a chance that one of the machines will fail. In these circumstances, tolerance to hardware failure would be of benefit. The current architecture has a centralised component, the APS, which determines much of the policy. This centralised structure simplifies development, but is a restriction on scalability and is a central point for failure. It would be possible to distribute the state of the APS and SSG, but only at the cost of more complex communications. A further development of the infrastructure will enable subspace searches to spread on to previously unused platforms, for example in response to an existing machine becoming overloaded or failing. Work is currently beginning in this area.

[5] http://www.ngs.ac.uk

5.3 Further investigations

The prototype infrastructure has been implemented to possess a varied set of policies and mechanisms which will allow it to adapt to both unpredictable applications and execution environments. Many of these policies have not yet been thoroughly examined, and many questions remain to be investigated. For example given a network of machines with a certain range of capabilities, what factors determine the benefit, if any, of adding a further machine whose performance is much below that of the existing machines?

5.4 Use with other applications

The infrastructure has been designed and implemented so as to separate the concerns of the application from those of execution control. This means that the infrastructure can be used with other applications. So, work is commencing on using the prototype infrastructure to control the execution of parallel iterative algorithms, for solving partial differential equations (e.g. Bahi et al.,2005).

6 Concluding remarks

The methodology described here has the potential to reduce the time required to run disclosure risk analysis for microdata. This, in itself, would be a major benefit, significantly reducing the time between data collection and release, thus improving the timeliness of the data. However, disclosure risk analysis is moving away from such single shot analyses of the "data to be released" into analysis of the "data environment" (Purdam and Elliot, 2005). It is here that the methodology really has the potential to come into its own. In combination with other new methods; web crawling, data monitoring, synthetic data generation, Grid base disclosure risk assessment would enable the ongoing assessment of the risk contained within the data environment. This would give a far more thorough understanding of the disclosure risk situation than that which is possible through the orthodox data-centric approach.

7 Acknowledgements

This work is funded by the small grants scheme of the ESRC National Centre for e-Social Science, ESRC grant number RES-149-25-1007.

8 References

Bahi, J.M., Contassot-Vivier, S. and Couturier, R. (2005): `Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms', *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 4, April 2005, pp. 289-299

Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K. and Zhou, Y. (1996): `Cilk: An efficient multithreaded runtime system', *Journal of Parallel and Distributed Computing*, vol. 37, no. 1, August 1996, pp 55-69.

Chin, J., Harvey, J., Jha, S. and Coveney, P.V. (2005): *Scientific Grid Computing: The First Generation. Computing in Science and Engineering*, vol. 7, no. 5, September 2005, pp. 24-32.

Elliot, M. J., Manning, A. M. and Ford, R. W. (2002): 'A Computational Algorithm for Handling the Special Uniques Problem'. *International Journal of Uncertainty, Fuzziness and Knowledge Based Systems* vol. 5, no. 10, October 2002, pp. 493-509.

Foster, I., Frey, J., Graham, S., Tuecke, S., Czajkowski, K., Ferguson, D., Leymann, F., Nally, M., Sedukhin, I., Snelling, D., Storey, T., Vambenepe, W. and Weerawarana, S. (2004): `Modeling stateful resources with web services', http://www.globus.org/wsrf.

Garey M. R., and Johnson D. S. (1979): *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Co., New York.

Goux, J., Kulkarni, S., Yorder, M. and Linderoth, J. (2001): `Master-Worker: An enabling framework for applications on the computational grid', *Cluster Computing*, vol. 4, no. 1, March 2001, pp 63-70.

Harbulot, B., Keith, M. and Brooke, J. (2006): A Web-Service based Scheduling Framework for the Pulsar Virtual Observatory. *HPC-GECO/CompFrame 2006 workshop (at the HPDC-15 conference),* Paris, France, June 19-20 2006 (to appear).

Manning A. M. and Haglin, D. J. (2005) `A new algorithm for finding Minimal Sample Uniques for use in Statistical Disclosure Assessment', *Proceedings of the Fifth IEEE International Conference on Data Mining*, Houston, Louisiana, U.S.A., November 2005, pp 290-297.

Mayes, K.R., Lujan, M, Riley, G.D., Chin, J., Coveney, P.V. and Gurd, J.R. (2005): `Towards performance control on the Grid', *Philosophical Transactions of the Royal Society of London: Series A*, vol. 363, no. 1833, August 2005, pp 1793-1805.

Merz, G. and Murphy, P. (1996): UCI repository of machine learning databases. *Technical Report, University of California, Department of Information and Computer Science*, http://www.ics.uci.edu/~mlearn/MLRepository.html.

Purdam, K and Elliot, M.J. (2005): `The Data Environment Modelling Service', *Report to the Office for National Statistics*, October 2005.

Yang, Y. and Casanova, H. (2003): `RMUR: Robust scheduling for divisible workloads', *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, Seatle, USA, June 2003, pp 114-123.