An SOA Approach to High Performance Scientific Computing: Early Experiences

Jaison Paul Mulerikkal, Peter Strazdins
School of Computer Science
The Australian National University
Canberra, 0200, Australia
Email: (jmulerik, peter.strazdins)@cs.anu.edu.au

***Abstract*—Service Oriented Architecture (SOA) has been embraced in enterprise computing for several years. The scientific community always felt the need of an SOA infrastructure not only with the convenience of enterprise SOA but also with expected level of high performance capabilities. Our research has produced an SOA middleware (ANU-SOAM) which supports an already popular enterprise SOA middleware API (*Platform Symphony* API) with the desired level of performance for scientific computations such as a Conjugate Gradient Solver. We have extended the compute services of ANU-SOAM with a common data service (CDS) between client and the service instances. The aim is to improve performance of applications by reducing communications or communication cost between the client and the service instances with the help of CDS. This is achieved by enabling tasks to perform a deferred *put* operation to the common data their service instances, with the results of the put operation only being visible to the next *generation* of tasks. These updates can be synchronised (committed) at CDS at the direction of the client. This property enables applications on ANU-SOAM to overcome latency of poor networks (or 'cloud') between client and service instances. Experimental results on a small Gigabit ethernet cluster show that, for the Conjugate Gradient Solver, the ANU-SOAM version suffers no appreciable performance loss over MPI versions and the CDS enhances N-Body Solver performance, with good scalability in both cases.**

I. Introduction

Scientists and researches are not necessarily computer programmers. But, they are always at the receiving end of needing high performance computing resources. The conventional message passing paradigms like MPI (Message Passing Interface) offer them a communication infrastructure with necessary HPC capabilities, but with an undesirable level of complexity and coding effort to develop applications. The SOA approach - computing paradigm that considers services as building blocks for applications [1] - seems to be more appealing to the scientific community with its easy to understand architecture and programming interface. Enterprise SOA middlewares (SOAM) like *Platform Symphony* have worked well for financial applications and its general architecture and API are very well accepted by even amateur programmers [2]. However, there are great challenges in performance, when those enterprise solutions are adopted for high performance scientific computing (HPSC) applications as they are suited for financial applications (which are generally embarrassingly parallel) rather than for scientific applications [3].

One of the major challenges in implementing important scientific computations, such as the Conjugate Gradient Solver (CGS) or N-Body Solver (NBS), is the inter-dependency of some of its underlying computational tasks. When such applications are parallelized, this inter-dependency shall compel atomic work units (called *tasks* in SOA) to progress in phases (which can be called *generations*) and may decrease task granularity in many cases. Decreased task granularity will result in increased communication and the communication cost along with it.

Our research is aimed at providing an efficient high performance SOA paradigm to solve these medium to fine grained and communication intensive scientific applications. In order to achieve the goal, we developed a SOA middleware (ANU-SOAM) and added a common data service (CDS) with its compute services. The experimental results show that ANU-SOAM applications perform as well as any other HPC paradigms such as MPI (Message Passing Interface) but with the added advantage of the SOA paradigm and its simpler API. The CDS enables applications to initiate *generations of tasks* which can get common data and put updates to it without losing meaning of data for a specific generation of tasks. This allows consumers to do more work at service instances with less communication costs, especially over low latency networks or clouds. A performance analysis of NBS application using the *generation* approach on ANU-SOAM indicates promising results over traditional SOA approach. The ANU-SOAM CGS and NBS applications also scale well, under our experimental conditions.

The rest of the paper is organized as follows: the next section gives a short background to SOA model for scientific applications, with sub-sections II-B and III-D giving more insights into challenges to SOA for HPSC and proposed CDS solution. Section III discusses the design of ANU-SOAM with its CDS. Section IV analyzes the performance of ANU-SOAM against MPI with a conjugate gradient solver. The same section also evaluates the experimental results of N-body solver application using *generation of tasks* approach with CDS against N-body solver application using traditional SOA on ANU-SOAM. Section V gives an

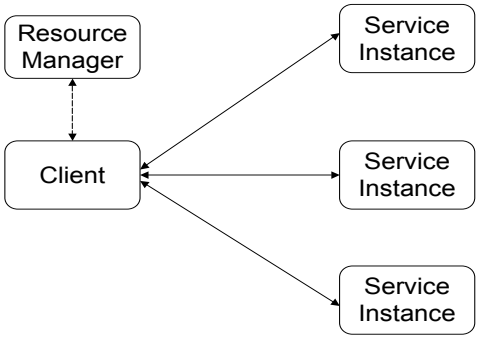

Figure 1. Traditional SOA Model

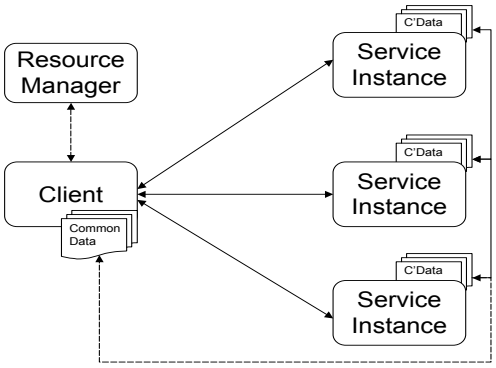

Figure 2. SOA Model with Data Service

account of related works in this area. The paper ends with conclusions and expected future functionalities of ANU-SOAM.

II. The SOA model for Scientific Applications

Ian Foster observed in [4] that the role of scientists is to convert data into insight. He also noted that with the help of advancements in Information Technology, especially with the help of SOA approach, scientists could automate many of those time consuming 'insight creating' jobs. He coined the word *Service Oriented Science*, which considers those 'insight creating' jobs as atomic blocks called services [4]. Efforts have been made from then on to make scientific applications run on SOA platforms. Later on, SOA was tried over grids which has given the notion of Service Oriented Grids (SOG) [5] and Global Grid Forum produced the Open Grid Services Architecture (OGSA) document to define standards for SOG [6].

SOG empowered scientists and researchers to address specific computational challenges in their own disciplines by providing custom built grid solutions. As a result there emerged a number of grid middlewares and tools which helped scientific community to use the extraordinary computing power of grids. Globus - an open source software toolkit that facilitates construction of computational grids and grid based applications, Unicore - a vertically integrated Java based grid computing environment, Legion - an object based meta-middleware system, Gridbus (recently renames as Cloudbus) - a set of grid services that support resource sharing, management and scheduling (including Alchemy and Aneka) are some examples which provided SOG solutions towards specific scientific needs [7], [8]. These SOA grid models are academic in nature.

Another major stream is enterprise grid solutions that use SOA paradigm mainly in financial sector. Platform Symphony is one of those widely used enterprise SOA middlewares. The general features of SOA on the basis of our investigation on Platform Symphony is explained in the next sub section.

A. Traditional SOA Model

In the traditional SOA model, as we have investigated in Platform Symphony, a consumer is called a *client* who is requesting *services* from compute nodes for an *application*. The client and service are pieces of software that run on hardware nodes as process instances. A client process can access (or even generate) a number of service processes that provides the same service. These service processes are called *service instances* (SI). Requests of client to SIs for a particular service along with its input data is called *tasks*. A group of tasks that share common characteristics, such as being part of a single application, constitute a *session*. An SI can accept any number of tasks within a session. Tasks are processed at SIs and results are send back to the client. Task results are collected at client process to produce final outcome. These actions constitute the compute services of general SOA middleware [9], [10].

Service providers can host and publish network-accessible service modules within their compute nodes. The client discovers these services with the help of a resource management module. The resource manager is also responsible for resource allocation and load-balancing of each application and its runtime needs. Using this mechanism, the client and the service instances orchestrate conversations to conduct long-lived flexible transactions [11] (Fig. 1).

B. Challenges and Possible Solution

The traditional approach of academic and enterprise SOA middlewares like Aneka of Gridbus (now, Cloudbus) and Symphony of Platform Computing work well for coarse-grained and 'embarrassingly parallel' applications [2], [12]. However, the challenges posed by many scientific applications with finer task granularity which result in higher communication costs, are not addressed by these existing

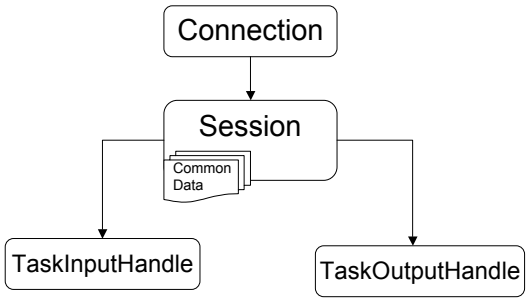

Figure 3. Symphony, ANU-SOAM - Client Classes

approaches [3], [12]. This is because, as task granularity decreases (as in medium to fine grained applications) periods of communication which separate periods of computation increases. This could lead to scenarios where the communication costs of parallelizing the application outperforms the computational advantages. This issue was very well reported in [12] with a block-based square matrix multiplication experiment. This problem will be acute if the consumer (client) tries to access computing power of SIs over a busy or slow network - to that matter 'a cloud'.

We believe that this problem can be addressed with the help of a flexible common data service (CDS) which allows SIs to communicate each other without tight coupling. This is done by updates of common data, where each SI merges changes made to the common data by recent service tasks run on each SI. This can reduce communications between client and SIs in many applications. The use of common data can also reduce input data size for the service task, thus further reducing communication costs.

The concept of CDS evolves from common data functionality in *Platform Symphony*. Using common data function, application programmer defined arrays or data elements common to all tasks could be set from a client to all SIs. This data is replicated at all compute nodes and persistent through out the life of a session and can be accessed by all SIs. The common data can also be updated with new values from the client side in between the application process. However common data function in *Platform Symphony* does not provide any mechanism to manipulate data from service side (from SIs). The ANU-SOAM CDS extends this function and provides *add, get* and *put* common data functions to the service tasks run by the SIs, but without losing data consistency for the current *generation* of tasks. The CDS also allows the client to control the synchronization of these updates. The implementation details of CDS are discussed in Section III-D and a graphic impression is given in Figure 2.

III. Design and Implementation of ANU-SOAM with Common Data Service

ANU SOAM implementation includes 3 major blocks - client, service and resource manager. We retain most of the *Platform Symphony* API with no or minimal changes. This API is exposed to application programmers which can be used to develop client and service application codes to run on ANU-SOAM. This approach helps already existing *Platform Symphony* applications to be easily ported to ANU-SOAM. MPI dynamic process creation techniques are used to create client and service processes. MPI point-to-point communications are used to communicate between client and service processes and collective calls are used to implement CDS.

A. Client

The client process starts with an MPI process creation. An API is provided to set session creation attributes, which will define the nature of the session. These application programmer defined session attributes include the name of the service requested by the client, maximum number of SIs (resources) expected by the client, path to the service executable at compute nodes, etc. With these attributes, the client initiates SIs using MPI dynamic process creation techniques (see Listing 1). This constitute a session between client and SIs and the SIs are ready to accept tasks from client. Tasks are generated by the client within the persistent session and send with the help of *task input handle* to appropriate SIs assigned by resource manager. As the task outputs return from service instances, they are accepted and enumerated with the help of the *task output handle* at the client. These results are made available to the application programmer in a meaningful order. Once the client process is over, it directs the service instances to die (or to hibernate) and ends the session. The major client classes and their relationships are given in Figure 3.

Listing 1. Session at Client

```
Session Connection::createSession
    (SessionCreationAttributes SesAttr){
    Session session;
    ...
    // spawning service instances
    MPI_Comm_spawn(SesAttr->name, ...);
    MPI_Intercomm_merge(...);
    ...
    session.setSessionComm(...);
    ...
    return session;
}
```

B. Service

Service Instances that are initiated (or requested) by the client, accept the parent communicator from client

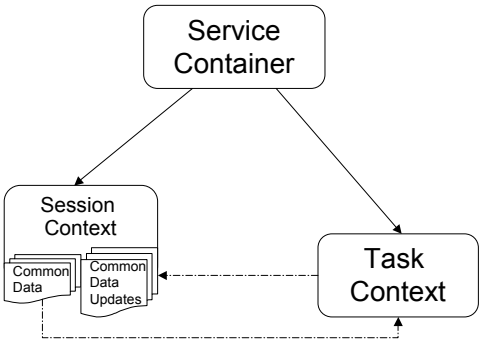

Figure 4. Symphony, ANU-SOAM - Service Classes

and create an inter-communicator. This inter-communicator helps communication between the client and the service instances. There is also another communicator which enables communication between the spawned SIs alone (that is, a second communicator which excludes client). Within a session, service processes (SIs) are set to an infinite loop to receive tasks and other instructions from the client. Looking at the message tag, a service process decides the nature of the client instruction (whether it is a task message or an instruction to end the session, etc) and respond accordingly (Listing 2). The SI can send back task results to the client at the completion of a task and wait for new tasks from the client. Major service classes and their relations are represented in Figure 4.

Listing 2. Session at Service Instances

```
void ServiceContainer::run(){
    MPI_Comm_get_parent(...);
    MPI_Intercomm_merge(...);
    ...
    // session as a loop
    while(1){
        MPI_Probe(...);
        // get tag & switch to
        // corresponding action
        switch (tag){
            ...
        }
        ...
    }
}
```

C. Resource Manager

In the present version of ANU-SOAM, the resource manager has to be supplied with prior knowledge of available compute resources. The application programmer can choose resource allocation policies like, random, round robin or weighted round robin to distribute tasks among compute nodes and SIs (a compute node can have more than one SI).

D. Common Data Service

Data common to all SIs can be set (*add*) using this service. The common data is replicated among all SIs and client process. Even though it may give rise to memory scalability issues[1], it enables the CDS to be fault tolerant This common data can be accessed (*get*) either by client or SIs. Updates to this common data (*put*) can also be made by individual SIs or client processes, without changing common data for the current *generation* of tasks. That means, *put* is a deferred operation in the CDS.

These updates (*put*) can be synchronised (*sync*) either among SIs or between the SIs and the client process according to application logic. The control of the *sync* operation remains with the client. The cost of synchronizing common data among SIs alone will be less compared to a global sync between client and service processes. This cost may increase considerably if the consumer(client) access those services over slow networks (e.g. Internet 'cloud'), which has high latency. But for many generations of tasks, global sync is not necessary in many scientific applications (e.g. CGS, NBS). ANU-SOAM gives a choice to take that decision and thereby making the application more efficient and cloud enabled.

Even though the nomenclature may look alike, *add, get, put* and *sync* of CDS are different from similar functionalities provided by many other paradigms. The CDS also provides a variant of *sync* and *get* functions to deal with synchronization event between SIs alone. They are explained below:

The *add* function adds common data, which can either be an element or an array of elements, but to be uniquely identified by a name. We can have multiple common data in a single application. When it is called by the client, the client keeps a copy of it in its memory and send it to all service processes to update their common data list. If it is called at service side, ANU-SOAM assumes that it is only a service specific common data and adds only to the service side common data list.

The *get* function enables client of service process to access a specific common data identified by its unique name.

The *put* is mostly a service specific function. It allows service instances to put updates to any common data. However unlike in other data services like Global Arrays, the updates does not change the present common data until *sync* is called by the client. In other words, data updates are remembered by the service processes with its meta-data. This can also be called by the client, then it instantaneously updates common data at the client and service sides.

[1]It may be noted that competing approaches, such as cluster OpenMP, suffer from similar limitations [13].

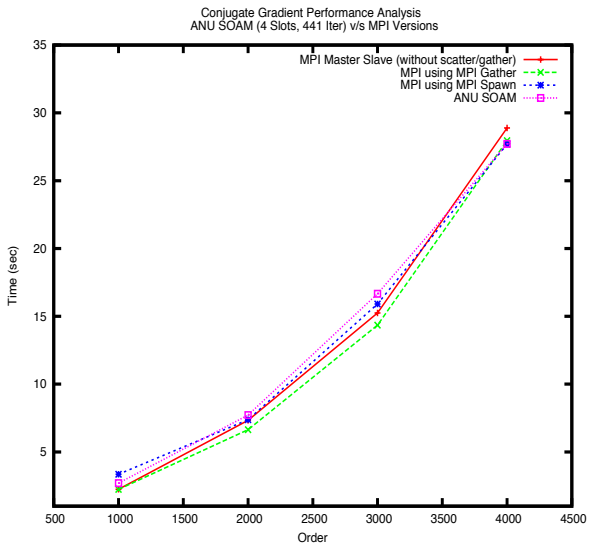

Figure 5. CGS - ANU-SOAM and MPI Versions - Performance Analysis

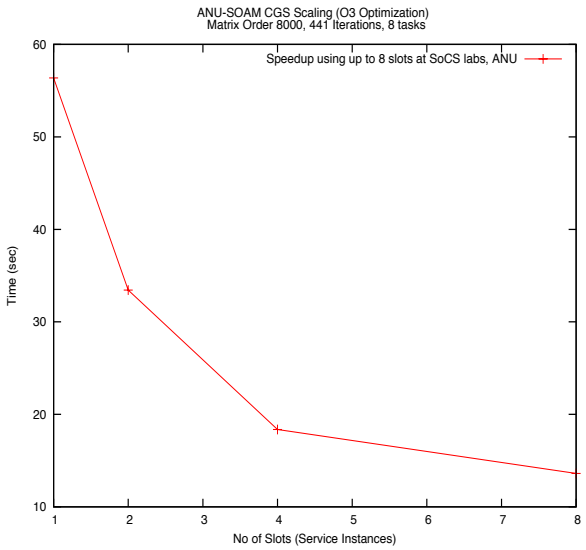

Figure 6. ANU-SOAM CGS Scaling

The *sync* function synchronizes updates to a specific common data identified by its name. That is, it commits all *put* operations since the last *sync*. This can only be called by the client. This is because there could be several tasks within the same SI that may be updating the common data and the client alone has got the knowledge of when all tasks in the current generation have been sent and the common data is ready to be updated. At this call the service processes communicate each other and commits all previous updates to that particular common data. The updated common data at SI is also communicated back to the client and gets updated at client side.

The variant function *iSync* applies sync only to service instances. The common data will be synchronised among SIs but won't be updated back to the client side. This function is useful, when the client does not want to know all the updates to common data at SIs for every generation. However, an *iSync* shall always be followed by an *iGet*, which will always fetch updated common data from the service side to ensure data consistency at the client side.

ANU-SOAM does not expose MPI inter-communicators as such to the application programmers. Instead, ANU-SOAM implements more convenient API calls of Platform Symphony as mentioned before. However, the application programmer has to decide and express which part of data has to be distributed, modified and synchronized. The Platform Symphony inspired ANU-SOAM API is given in the appendix to explain this aspect.

IV. Experiments and Performance Analysis

The Conjugate Gradient Solver (CGS) and N-body Solver (NBS) are well-known medium grained communication intensive computations. So, they were chosen to evaluate and compare the performance of SOAM. CGS was used to compare ANU-SOAM performance with that of pure MPI implementation and NBS was used to compare the advantage of ANU-SOAM generation approach with CDS to that of a traditional SOA approach, also implemented on ANU-SOAM. The results are discussed below.

A. Conjugate Gradient Solver

The Conjugate Gradient Solver (CGS) is considered to be a medium grained communication-intensive computation. So, it was chosen to compare the performance of ANU-SOAM with that of pure MPI implementations. The CGS algorithm, which is explained in our previous paper [3], is reproduced here for the completion of this paper.

The CGS is used to solve the linear equation: $Q*x-b=0$, where Q is a given $N*N$ symmetric positive definitive matrix and b is a given N vector. If the iteration number is i, x_i is considered as the solution approximation, d_i is termed as the search direction and r_i is the residual. The technique is to perform iterations in the direction of d_i till we find that the residual r_i is small enough to be ignored to find the solution, x_i. For the same, the iterations starts from an arbitrary initial guess of
$x_0 = 0$ and
$r_0 = d_0 = -Q * x_0 + b$

Constants α and β which are functions of d_i, r_i and Q determine the successive values of conjugacy constraints. This iterative process involves two main mathematical operations, namely,

1) Compute: $z = Qd_i$ (Matrix-Vector multiplication)
2) Reduce: x_i ,d_i (Vector-Vector operations) and Compute: r_i.

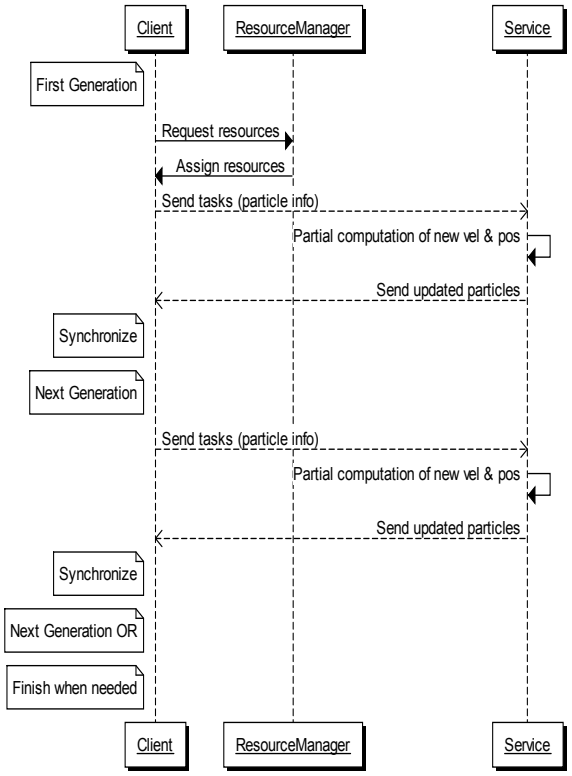

Figure 7. Sequence Diagram for NBS - Traditional SOA Approach

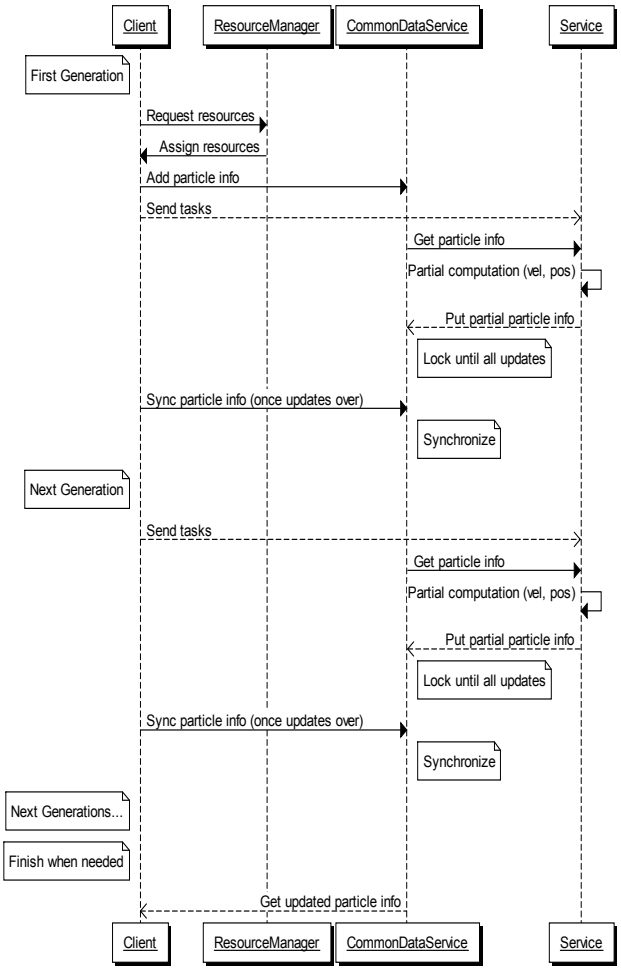

Figure 8. Sequence Diagram for NBS - With Common Data Service

An ANU-SOAM CGS program was developed without using CDS extension and was experimented on a Gigabit Ethernet cluster (with 2GHz AMD Opteron and AMD Athlon processors), with each SI having a dedicated CPU. Scalability tests were also conducted for hte ANU-SOAM CGS program for up to 8 SIs. In order to compare the results, a number of pure MPI CGS implementations using different MPI techniques - client-server model, MPI collective call model (using MPI_Comm_allgather) and MPI Spawn - were also developed and experimented on the same testbed. In order to get comparable results, the refinement of CGS result was limited to 441 iterations for all experiments. Experiments were conducted for dense square matrices of order from 1000 to 8000. The results are compiled in graphs 5 and 6.

1) CGS Performance Analysis: Figure 5 gives a comparison between ANU-SOAM and various MPI CGS programs. In the performance comparison graph, total time includes the session creation time, common data set up time and the total time taken for all tasks to complete. It shows that the ANU-SOAM implementation of CGS is as good as other MPI implementations. It suggests that architectural overheads in developing a SOA middleware using MPI techniques are kept to minimum in our implementation. This suggests that we have succeeded in designing an SOA middleware which has got the expected level of performance for scientific applications at HPC standards prevail in academic circles.

The ANU-SOAM CGS was also tested for scalability (with -O3 optimization). Each SI was assigned a dedicated compute node. In Figure 6, the x-axis shows the number of nodes (or the number of SIs) and the y-axis shows the total execution time for 441 iterations but with 8 tasks in all cases. The scalability graph shows that ANU-SOAM scales well as we move from single service instance to up to 8 service instances on 8 compute nodes, under our test conditions. Since we used 8 tasks for all experiments, the number of tasks to be served by an SI with a lower number of SIs would be higher. This scenario could also be attributed to the better than expected scaling effect we see on the graph.

B. N-Body Solver

The N-body problem in astrophysics and molecular dynamics is considered to be an important but highly computation and communication intensive scientific problem. For our experiments, we have adopted a naive or 'direct' implementation of N-body problem in astrophysics with

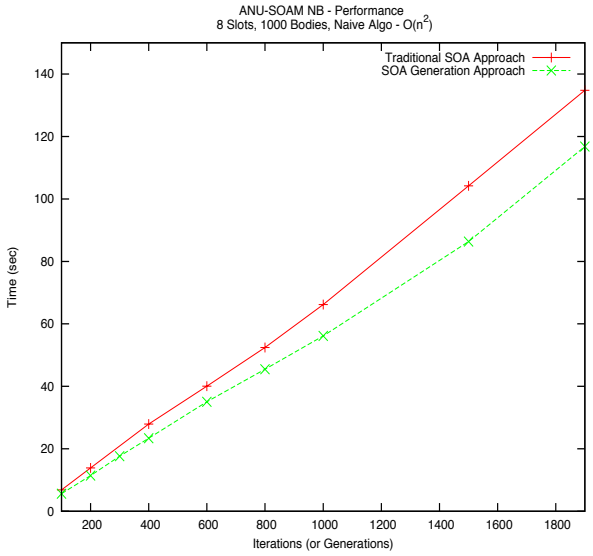

Figure 9. ANU-SOAM NBS Performance - Traditional SOA v/s Generation Approach

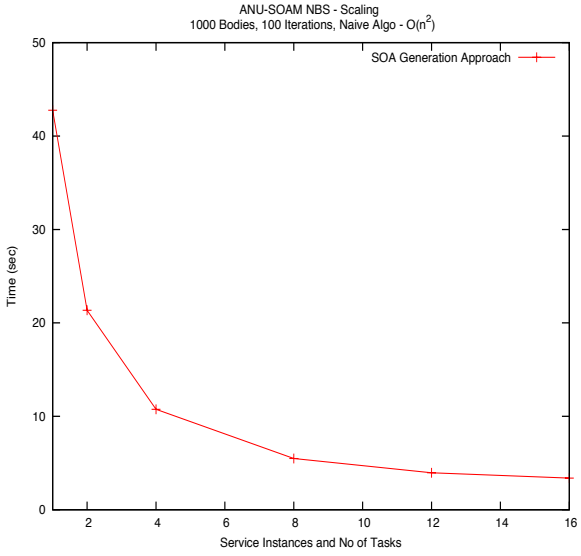

Figure 10. ANU-SOAM - NBS - Scaling

$O(n^2)$ computation complexity. While asymptotically more efficient algorithms exist, the direct algorithm is still widely used in astrophysics, due to concerns over the accuracy of faster methods [14].

The aim of our experiment was to analyze the advantages of *generation* approach using CDS to that of traditional approach on SOAM. The Naive NBS algorithm was a perfect choice for that purpose for we could update the new positions and velocities of N bodies in the problem only in phases. Since each and every particle is dependent on all other particles for their new positions after a time step, the NBS algorithm has to move in steps or *generations*. NBS algorithm for traditional SOA can be summed up as follows:

1) Send input data (position, velocity and mass of all particles) to service instances.
2) Do parallel updates of new position and velocity of (N/No. of service instances) of particles at each service node.
3) Send back partial particle updates to client and synchronize results (new positions and velocities).
4) Send input data again and repeat the process for required time steps (iterations).

This traditional algorithm can be twisted a bit with the CDS to help accepting a new *generation* of work at the service side without sending input data again and again for each iteration. The NBS algorithm with CDS will be as follows:

1) *add* input data as common data (position, velocity and mass of all particles).
2) Do parallel work and *put* updates of new position and velocity of (N/No of service instances) of particles at each service node.
3) Once a *generation* of tasks are over, send instruction for *iSync* by the client.
4) Service Instances apply sync among themselves and initiates new generation of work.
5) Continue the algorithm over the required number of time steps (or *generations*).
6) The Client gets the results (new positions and velocities).

The differences in these two methods are vividly portrayed in sequence diagrams (Figures 7 and 8) given. These two variants of ANU-SOAM prgrams were compiled with -O2 optimization and tested on 4 nodes of 2.4 GHz Intel Core2 Quad connected with Infiniband. However, the client was run from a different node in the same network but connected with Gigabit Ethernet connection. Experiments were conducted with 1000 bodies and time steps (iterations or generations) varying from 100 to 2000 for performance and scalability. The results are compiled in Figure 9 and Figure 10.

1) NBS Performance Analysis: The performance comparison between NBS with traditional SOA approach and *generation* approach recorded in Figure 9 shows clear advantage of *generation* approach over the traditional one. It is to be noted that in both variants of the program, the complexity of algorithm remains the same ($O(n^2)$). But the new approach allows a *service-centric* algorithm that replaces heavy communications between the client and service instances with that of light-weight *sync* and task instructions. This modification has increased the performance of NBS in our lab settings. This advantage could be more evident in scenarios where the client has to access SIs over the internet

cloud. The new approach has also reduced the coding effort (number of lines of the client code (main) has reduced from 160 to 120. The service code effort remain almost the same.

In the scalability experiments, each task was assigned a dedicated SI with up to 8 compute nodes. In Figure 10, x axis shows the number of SIs (more than one SI run on some compute nodes, when number of SIs exceed 8) and y axis shows the total execution time and for 1000 bodies for 100 iterations (time steps). We get good speedup up to 8 SIs and after that the computational advantages are overpowered by communication costs and memory constraints.

V. Related Work

There were attempts to make MPI system-aware to help development of efficient scientific applications. Research at the Universidade Federal Fluminense (UFF), Nitero'i, Brazil in this line has resulted in the development of EasyGrid. It is designed as an Application Management System (AMS) for MPI library [15]. It allows applications to be autonomous, which is not a property of pure MPI applications with the help of a Resource Management System (RMS) [16] who implements self-scheduling policies. EasyGrid has given positive results according to [17] and [15]. However, it is not clear that EasyGrid addresses the problems associated with task granularity and high communication costs expected in scientific applications.

Aneka is a software platform for developing distributed application which is initiated by CloudsLab of Melbourne University, Australia. It has been recently commercialized by Manjarasoft and intends to provide SOA platform as a service (PaaS). As mentioned before, early attempts have been made to identify the impacts of task granularity and associated costs in SOA enabled scientific applications using Aneka in [12]. However, there is no evidence that Aneka handles this issue even in its latest versions [18].

There exists some data service tools for SOG/SOA middlewares like Global Arrays (GA). GA is compatible with MPI [19]. It distributes the data among compute nodes and provides an addressing system to represent them as a single logical block. GA can query where the data is, before starting a computation. So, it is also useful for data intensive computations, when data affinity can be used to direct computations to particular node(s) which needs specific data [20]. GA allows global arrays to be updated and accessed using *get, put* functionalities. But GA *get* and *put* are not differed operations as in ANU-SOAM CDS and updates GA instantly in a sequential order. Moreover GA *sync* is only a barrier for all GA operations [21]. Another related work is reported in [22], which discusses PERSISTENT and PERSISTENT_RETURN mode of data transfer in Grid RPC (remote procedure call). It tries to avoid unnecessary data transfers within Grid RPC and supports asynchronous and coarse-grained parallel tasking.

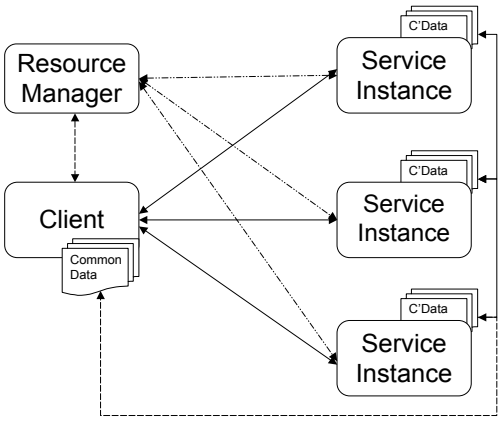

Figure 11. ANU-SOAM -Future Work

In order to help data intensive applications, some data grids ([23]–[25]) make use of distributed file systems (DFS) like Google File System [26] or Hadoop Distributed File System [27]. In those cases, DFSs serve the purpose of providing a distributed data space using full or partial replication of huge chunks of input data for computations. However this approach is neither meant nor effective for computation intensive applications where atomic units of work (tasks) are not independent from all other units within the same generation [28].

VI. Conclusions & Future Work

The development of ANU-SOAM has provided desirable level of performance for *Platform Symphony* inspired service oriented architecture which is evident from CGS experiments. The common data service provides tools to reduce communication costs either by reducing communications between client and SIs or by reducing data transactions between client and SIs. Within CDS, SIs can keep session specific data in their local memory and can *get* it. Tasks within SIs can also *put* updates to it but without changing the common data for a specific *generation* of tasks. Those local and partial updates can be synchronised (committed) by the client either among SIs or among client and SIs. By enabling this new data service along with the compute services of traditional SOA middleware, the efficiency of scientific SOA applications can be improved. A comparison between ANU-SOAM NBS implementation with CDS to that of traditional SOA approach gives positive indications in that line.

SOAM also offers the flexibility of using the *Platform Symphony* API in developing scientific applications. This will help wider scientific community to develop SOA scientific applications much easier. ANU-SOAM's compatible API will also help already existing *Platform Symphony*

applications to be ported to SOAM, with relative ease. The CDS is also expected to bring about new algorithms to solve classic problems like NBS.

ANU-SOAM has to be further tested and optimized for large scale HPC clusters. We intent to test it on larger systems like ANU Supercomputing Facility and Magellen Cloud for HPC at Argonne in the near future. The memory scalability issues of ANU-SOAM will be addressed in the future by proper data affinity, caching and data consistency techniques.

We expect to introduce reduction services like SUM, MAX etc as future extension to CDS . This will enhance the capacity of application programmers to develop more efficient scientific applications with relative ease. In order to make SOAM work better under heterogeneous conditions dynamic load balancing and scheduling policies are to be built into the resource manager module (Fig: 11). Fault tolerant improvements will be another major area to be worked on. With these improvements ANU-SOAM shall become a robust solution for cloud-enabled SOA for HPSC.

ACKNOWLEDGMENT

The authors would like to thank Australian Research Council for its support to this research through Linkage Grant LP0669762. We would also like to thank *Platform Computing* for their continued cooperation with the project.

REFERENCES

[1] D. Puppin, N. Tonellotto, and D. Laforenza, "How to run scientific applications over web services," in *ICPPW '05: Proceedings of the 2005 International Conference on Parallel Processing Workshops*, Institute for Information Science & Technology, Pisa, Italy. Washington, DC, USA: IEEE Computer Society, 2005, pp. 29– 33. [Online]. Available: http://dx.doi.org/10.1109/ICPPW.2005.39

[2] Platform Computing, "Grid-enabling and virtualizing mission-critical financial services' applications," Technical White Paper, August 2006.

[3] J. Mulerikkal and P. Strazdins, "Service oriented approach to high performance scientific computing," in *CCGRID '10: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, M. Parashar and R. Buyya, Eds. CPS, 2010, pp. 820–825.

[4] I. Foster, "Service-oriented science," *Science*, vol. 308, no. 5723, pp. 814 – 817, 2005.

[5] L. Srinivasan and J. Treadwell, "An overview of service-oriented architecture web services and grid computing," HP Software Global Business Unit, Tech. Rep., 2005.

[6] D. Gannon, B. Plale, M. Christie, L. Fang, Y. Huang, S. Jensen, G. Kandaswamy, S. Marru, S. L. Pallickara, S. Shirasuna, Y. Simmhan, A. Slominski, and Y. Sun, "Service oriented architectures for science gateways on grid systems," in *Service-Oriented Computing - ICSOC 2005, Third International Conference*, ser. LNCS, B.benatallah, F.Casati, and Traverso, Eds., ICSOC. Amsterdam, The Netherland: Springer-Verlag, December 2005, pp. 21–32.

[7] P. Asadzadeh, R. Buyya, C. L. Kei, D. Nayar, and S. Venugopal, *Global Grids and Software Toolkits: A Study of Four Grid Middleware Technologies*, Y. L and G. M, Eds. Wiley Press, USA, 2005. [Online]. Available: www.gridbus.org/papers/gmchapter.pdf

[8] R. Buyya and S. Venugopal, "A gentle introduction to grid computing and technologies," CSI Communicaitons, July 2005. [Online]. Available: www.gridbus.org/papers/GridIntro-CSI2005.pdf

[9] Platform Computing, "Symphony application development guide," Platform Computing Inc, Tech. Rep., 2008.

[10] ——, "Symphony reference," Platform Computing Inc, Tech. Rep., 2009.

[11] M. N. Huhns and M. P. Singh, "Service-oriented computing: Key concepts and principles," *IEEE Internet Computing*, vol. 9, no. 1, pp. 75–81, January-February 2005.

[12] X. Chu, K. Nadiminti, C. Jin, S. Venugopal, and R. Buyya, "Aneka: Next-generation enterprise grid platform for e-science and e-business applications," in *E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing (e-Science 2007)*. IEEE Computer Society, December 2007, pp. 151–159. [Online]. Available: www.gridbus.org/papers/AnekaNextGenGrid2007.pdf

[13] A. Basumallik and R. Eigenmann, "Towards automatic translation of OpenMP to MPI," in *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*. New York, NY, USA: ACM, 2005, pp. 189–198.

[14] R. Spurzem, "Direct n-body simulations," *Journal of Computational and Applied Mathematics*, vol. 109, pp. 407–432, Sept. 1999.

[15] A. C. Sena, A. P. Nascimento, C. Boeres, and V. E. F. Rebello, "Easygrid enabling of iterative tightly-coupled parallel mpi applications," in *Parallel and Distributed Processing with Applications, 2008. ISPA '08. International Symposium on*. IEEE Computer Society, December 2008, pp. 199–206. [Online]. Available: http://ieeexplore.ieee.org.virtual.anu.edu.au/search/selected.jsp

[16] C. Boeres and V. E. F. Rebello, "Easygrid: towards a framework for the automatic grid enabling of legacy mpi applications," *Concurrency and Computation: Practice & Experience*, vol. 16, no. 5, pp. 425 – 432, 2004.

[17] A. P. Nascimento, C. Boeres, and V. E. F. Rebello, "Distributed and dynamic self-scheduling of parallel mpi grid applications," in *MGC '08: Proceedings of the 6th international workshop on Middleware for grid computing*. New York, NY, USA: ACM, 2008, pp. 1–6.

[18] C. Vecchiola, S. Pandey, and R. Buyya, "High-performance cloud computing: A view of scientific applications," in *10th International Symposium on Pervasive Systems, Algorithms, and Networks (ISPAN)*. IEEE Computer Society, 2009, pp. 4 – 16.

[19] GlobalArrays, "Global arrays," web page: http://acts.nersc.gov/ga/index.html, 2009. [Online]. Available: http://acts.nersc.gov/ga/index.html

[20] J. Nieplocha, R. J. Harrison, and R. J. L. Eld, "The global array programming model for high performance scientific computing," *SIAM News*, vol. 28, 1995. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.3342

[21] GAToolkit, "The GA Toolkit," web page: http://www.emsl.pnl.gov/docs/global/, 2008. [Online]. Available: http://www.emsl.pnl.gov/docs/global/

[22] G. Antoniu, E. Caron, F. Desprez, A. Fèvre, and M. Jan, "Towards a transparent data access model for the gridrpc paradigm," in *Proceedings of the 14th International Conference on High Performance Computing*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 269–284.

[23] A. Chervenak, I. Foste, C. Kesselman, and C. Salisbury, "The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets," *Journal of Network and Computer Applications*, vol. 23, pp. 187–200, 2000.

[24] J. C. Werner, "Data and functional gridification for high energy physics." Available online: www.hep.man.ac.uk/u/jamwer/Grid2006.doc, January 2009. [Online]. Available: www.hep.man.ac.uk/u/jamwer/Grid2006.doc

[25] B. Allcock, J. Bester, J. Bresnahan, A. L. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke, "Data management and transfer in high-performance computational grid environments," *Parallel Computing*, vol. 28, no. 5, pp. 789–771, May 2002.

[26] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, 2003.

[27] D. Borthakur, "Hdfs architecture," Hadoop 0.19 Documentation, 2008. [Online]. Available: http://hadoop.apache.org/core/docs/current/hdfs_design.html

[28] (2010, May) What hadoop is not. [Online]. Available: http://wiki.apache.org/hadoop/HadoopIsNot

APPENDIX

The ANU-SOAM Common Data Service API:

- *add*
  add(cdName, cdElement)
  add(cdName, length, cdArray)
  - Adds common data (an element or array) to client and SIs, if called from the application client code. Adds SI specific common data, if called from the application service code.
- *get*
  get(cdName, cdElement)
  get(cdName, length, cdArray)
  - Gets common data. Can be called either by the client or service application codes.
- *put*
  put(cdName, updateElement)
  put(cdName, start , updateLen, updateArray)
  - Put updates to common data. The update is characterized by its start pointer and length within the common data. It is a differed operation. Mostly a service specific call.
- *sync*
  sync(cdName)
  - Commits updates to common data at both client and SIs. It can only be called by the application client code.
- *iSync*
  iSync(cdName)
  - Commits updates to common data at SIs alone. It can only be called by the application client code.
- *iGet*
  get(cdName, cdElement)
  iGet(cdName, length, cdArray)
  - Gets updated common data after *iSync* at client.