Building a Service Oriented Cloud Computing Infrastructure Using Microsoft CCR/DSS System

Rakpong Kaewpuang,
Putchong Uthayopas and Ganid Srimool
HPCNC, Department of Computer Engineering, Faculty of Engineering
Kasetsart University, Bangkok, Thailand
rakpongkaewpuang@gmail.com,
{pu, g5165272}@ku.ac.th

Juta Pichitlamken
Department of Industrial Engineering
Faculty of Engineering
Kasetsart University, Bangkok, Thailand
juta.p@ku.ac.th

doi:10.4156/ jnit.vol2. issue1.8

Abstract

Cloud computing is new paradigm for provision of a computing infrastructure and services the over network using a pool of abstracted, virtualized, and scalable, computing resources. One of the challenges is the lack of standard in configuration, management, and programming. Thus, we propose that a service oriented cloud can be built and program using Microsoft Windows Server and program using Microsoft CCR/DSS. The architecture of this service oriented cloud is presented in this paper. In addition, a practical logistic application called Pickup and Delivery Problem with Time Windows (PDPTW) is developed to demonstrate the concept. The experiments show that a very high speed up of more than 50 times on both 2 two quad core node and 16 quad core node and 80-90% efficiency can be obtained using a cloud computing system based on our concept.

Keywords: *Cloud Computing, Service Oriented Cloud, Microsoft CCR/DSS, Pickup and Delivery Problem with Time Windows*

1. Introduction

Cloud computing is an emerging new paradigm that to can provide almost unlimited computing power and storage to user through the network. The goal is to substantially reduce the costs associated with the management of hardware and software resources throughout the organization. Cloud computing is now being employed to build a massive platform for many company such as Google, Microsoft, Facebook, Yahoo, Ebay, and many more. Underneath the cloud is the use of large scale clustering technology to link together a massive number of computing resources such as computing nodes and storages with high speed gigabit network. Currently, the multicore[3] technology even helps deliver a very high performance computing system at a very low cost for the cloud computing system [1]. Nevertheless, one of the main obstacles for a broad adoption of cloud computing technology is the lack of standard in system the configuration, management, and programming.

Most of the well-known cloud implementation is still rely on a proprietary technology. For example, a cloud can be viewed by programmers through various API such as Amazon EC2 API, GoGrid API, Sun Cloud API, ElasticHosts API. Although many open standard efforts are now underway such as OCCI by OGF, the work is still in a very early stage.

In this paper, we propose an architecture for a *Service Oriented Cloud Computing System (SOCCS)* built mostly from the standard components available from Microsoft Windows Cluster and Microsoft Visual Studio programming environment. Moreover, we show that Microsoft Concurrency and Coordination Runtime (CCR) and Decentralized Software Services (DSS) [2] system is a good candidate for the construction of this scalable cloud system. The CCR runtime enable a simple and scalable programming model that is easy to understand. Moreover, the system integrated well with modern development environment. In this paper, the development of a logistics application called *Pickup and Delivery Problem with Time Windows (PDPTW)* is chosen as an example. The intention is to demonstrate that CCR/DSS is a viable solution for the development of a high performance business application. The application can run well on single multicore computer system and scale to multiple computer without change.

The organization of this paper is as follows. Section 2 briefly describes the concept of cloud computing, CCR /DSS system, and Window HPC. Section 3 proposes the design concept of CCR/DSS based Service Oriented Cloud Computing System. Section 4 explains the design concept behind the development of PDPTW application. Section 5 discusses about the experiment conducted and results obtained. Finally, Section 6 gives a summary of the work done.

2. Background

Normally, a cloud computing [12][14] system is a large pool of reusable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to allow an efficient use of available resources. Mostly, a large cloud computing systems are built using Linux and open source software components.

Currently, many enterprises is using Windows server as a major operating infrastructure. Windows based system provides many advantages in term of the ease of use and very advanced software development tools. During the past few years, Microsoft has released a product called Windows High Performance Computing (or windows HPC), which make it easy to built a powerful platform for large and complex analysis problem. These problems can be solved by completing the calculations in parallel. Thus, large problems can take advantage of multiple compute nodes grouped into computational clusters. Few examples of the kind of applications that can take advantages of a computation clusters are: Financial models, computer animation, crash analysis, and drug modeling.

One of the challenges in the building of an analysis cloud is to choose a right programming model that is easy to learn and scale well in the cloud environment. In this work, we decided to base our software development on Microsoft Concurrency and Coordination Runtime (CCR). CCR provides a framework for the building of an asynchronous distributed application [5][6][7]. The CCR system composed of a managed code library and a dynamically linked library (DLL) of two subsystems; a lightweight distributed services-oriented architecture and a common language runtime (CLR).

In CCR, application is constructed using a set of services. Each service is implemented as a thread pool controlled by Dispatcher. When launched, the main part of the application will break the task to be executed into many small sub-tasks. Then, Arbiter will queue these computing request with data to the work queue of a Dispatcher. The role of Dispatcher is to dispatch the work unit to the threads. After the result has been generated, output is sent to Arbiter again as an event. Then, Arbiter can collect the result and save them to storage for later processing. CCR helps manages both port and threads with optimized dispatcher that efficiently iterate over multiple threads. According to this programming model, a Master/Worker paradigm can be directly apply to structure the parallel and distributed application. Since CCR can manage remote invocation of thread across machine automatically, a single program can easily be scaled to run on multiple machines with a very minor configuration. To support the framework, the Application Programming Interfaces (API) have been provided for developers with three main set of functionality as follows:

- *The Port and PortSet queueing primitives:* Port and PortSet are queue of item used as data item. These include items such as byte, short, int, float, double
- *The Coordination primitives (arbiters):* Arbiters are classes used to execute user code for coordination among components. The port must be registered to arbiter in order to register a receiver arbiter to the port
- *The Dispatcher, DispatcherQueue and Task primitives:* The Dispatcher (thread pool) uses operating system threads to load balance tasks and provided primitives support dynamic threading model and capability that include FromHandler ,Receive MultipleItemReceive, MultiplePortReceive, JoinedReceive, Choice, Interleave. The DispatcherQueue is queue of work items. These work items are passed to dispatcher for future processing

As the software scale across one machine, the Decentralized Software Services (DSS)[4], a layer of software on top of CCR, can be used to link multiple services component together across multiple machines. DSS provides a lightweight and representational state transfer (REST) oriented application model with a system level approach for building high performance, scalable applications. DSS particularly suited for creating coarse grain applications, and. DSS uses decentralized software services protocol (DSSP) and HTTP for interaction with services, and DSS provides a hosting environment, publish/subscribe, security, monitoring, logging, debugging, they are set of infrastructure services can use for create service.

3. The Proposed CCR/DSS Service Oriented Cloud

In this paper we propose that a Service Oriented Cloud Computing System (SOCCS) can be constructed by combining CCR/DSS Software to form scalable services to a client application. The proposed design and architecture is as shown in Fig. 1.

From Fig. 1, a Service Oriented Cloud Computing System consists of many layers of hardware and software. The lowest layer is the cluster hardware that composed of computing resources connected together with high speed interconnection network. Each computing node has its own Windows operating system installed as a local operating system. Microsoft CCR/DSS must be installed on every node in the system. One node will act as a special control node that running a controller component called *Cloud Service Management (CSM).* CSM acts as a resources management system that keeps track of the availability of services on the cloud. CSM can be is developed as a CCR service. Thus, there is no need for any specialized software integration at all.

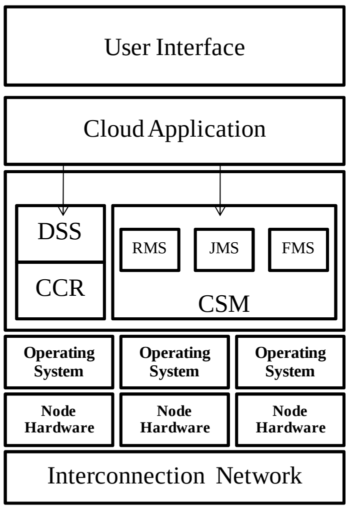

Figure 1. The Architecture of Service Oriented Cloud Computing Systems

Inside Cloud Service Management is consists of 3 service components are follows:

- *Resource Management Service (RMS):* RMS have responsibility for gather machines on network and keep information and status of each machine
- *Job Management Service (JMS):* JMS have responsibility for receive job from user, take job add to job queue, control and distribute job to group of compute machines and in case of some compute machine work fail JMS will resubmit job to group of compute machines again
- *Fault-Tolerance Management Service (FMS):* FMS have responsibility for detect missing of each compute machine

3.1. Resource Management Service (RMS)

Fig. 2. illustrates process of Resource Management Service. RMS consists of 2 main components are follows:

3.1.1. Management Compute Machine (Wait for worker)

Management compute machine (worker) provided 3 functions for manage workers are follows:

- *Register*: manage about registration of worker, this function will wait for connect from worker and then generate worker ID and send ID to worker, when worker received ID, worker will send information of worker back, and this function will add information of worker to database
- *Check status*: will check status of all workers, this function will send small message (called heart breath message) to group of workers every interval time. When worker received heart breath message, worker will send own ID back.
- *Unregister*: when worker want to exit, this function will remove information of worker from database

3.1.2. Management User (Wait for user)

- *Register*: manage user, when user connect to system, this function will generate user ID , then will send user ID to user and wait to receive information from user, finally add user to database
- *Unregiste*r: when user want to exit, this function will remove information of user from database and then will send unregister message to user

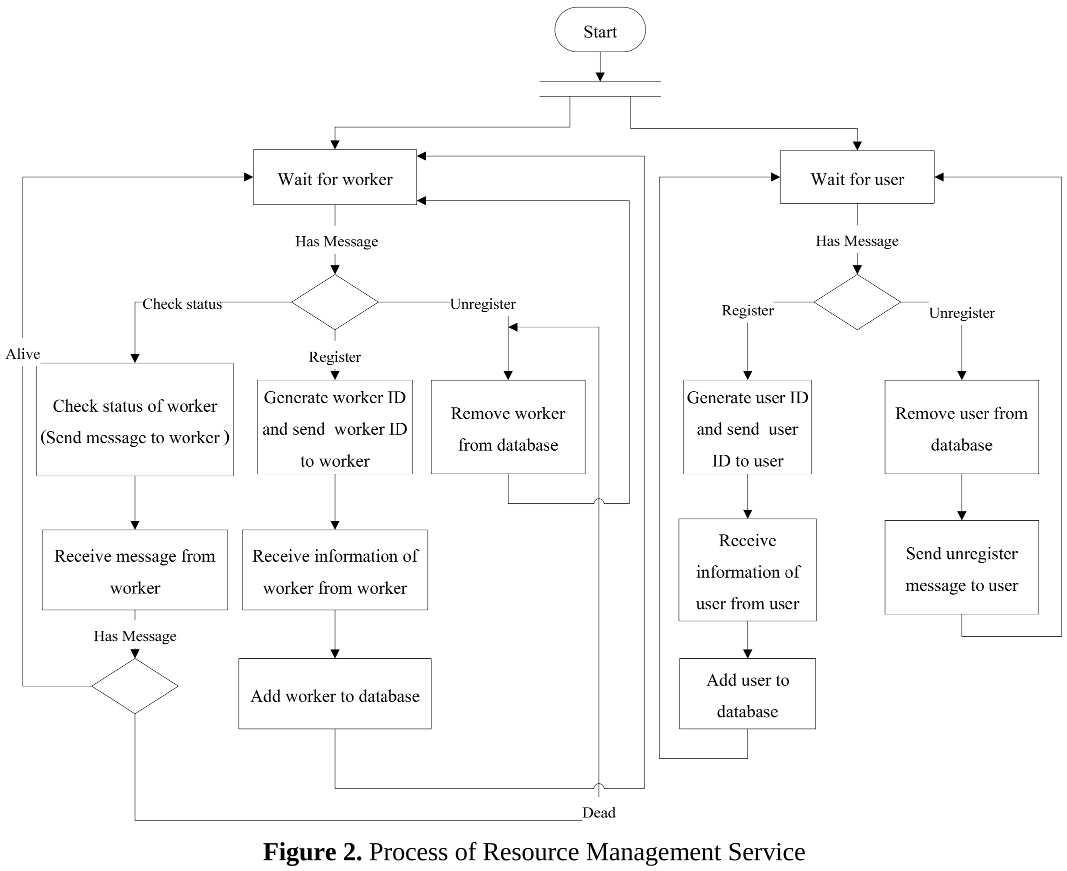
Start
Wait for worker
Wait for user
Has Message
Check status
Unregister
Register
Alive
Check status of worker (Send message to worker)
Generate worker ID and send worker ID to worker
Remove worker from database
Receive message from worker
Receive information of worker from worker
Has Message
Add worker to database
Dead
Has Message
Register
Unregister
Generate user ID and send user ID to user
Remove user from database
Receive information of user from user
Send unregister message to user
Add user to database

Figure 2. Process of Resource Management Service

3.2. Job Management Service (JMS)

Fig. 3. show process Job Management Service, JMS consists of 2 main components are follows:

3.2.1. Management Compute Machine (Send message to worker)

Management compute machine (worker) provided 4 functions for manage worker are follows:

- *Check status:* broadcast heart breath message to all workers every interval time and wait for receive respond message from workers
- *Job execute complete:* when worker execute job completed, worker send result to this function, and then this function receive result from worker and add result to result queue for send that result to user later
- *Unregister:* when worker is fail , this function will remove worker, then keep all jobs on that worker and add jobs to job queue again to execute on other workers are alive
- *Send job:* Dispatcher choose job and thread (from threads pool) and that thread send job to worker (job scheduler consider number of jobs on each worker, worker have less job than anyone is chose)

3.2.2. Management User

- *Wait for command from user:* when user submit job, this function will wait receive data and detail of job from user, next add job information to database and then add job to job queue for send to worker later
- *Send result to user:* manage send result of job back to user, when result of job appear in result queue, dispatcher choose result of job and thread, then remove job information from database and that thread will send result of job to user

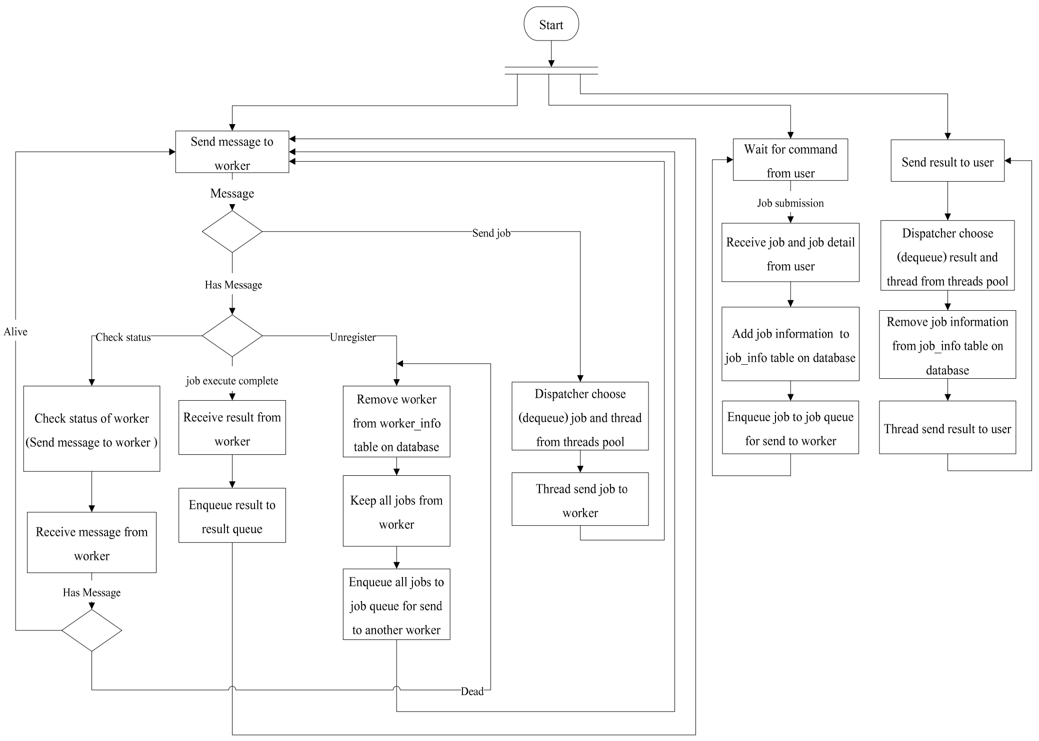
Start
Send message to worker
Message
Send job
Has Message
Alive
Check status
Unregister
job execute complete
Check status of worker (Send message to worker)
Receive result from worker
Remove worker from worker_info table on database
Dispatcher choose (dequeue) job and thread from threads pool
Thread send job to worker
Enqueue result to result queue
Keep all jobs from worker
Receive message from worker
Has Message
Enqueue all jobs to job queue for send to another worker
Dead
Wait for command from user
Job submission
Receive job and job detail from user
Add job information to job_info table on database
Enqueue job to job queue for send to worker
Send result to user
Dispatcher choose (dequeue) result and thread from threads pool
Remove job information from job_info table on database
Thread send result to user

Figure 3. Process of Job Management Service

3.3. Fault-Tolerance Management Service (FMS)

Fig. 4. show process of Fault-Tolerance Management Service. FMS will send heart breath message to all workers every interval time and wait for receive respond message from workers, in case of worker cannot send respond message, FMS separated cause into 2 cases as are following:

- *Worker is busy:* while worker is executing job, worker cannot send respond message, wait until worker execute complete, then worker will send respond message. FMS wait respond message of worker by depend on interval time of job (count wait times)
- *Worker is fail:* when worker cannot execute job, FMS will remove worker and keep all jobs on that worker and resubmit to job queue for execute to another workers

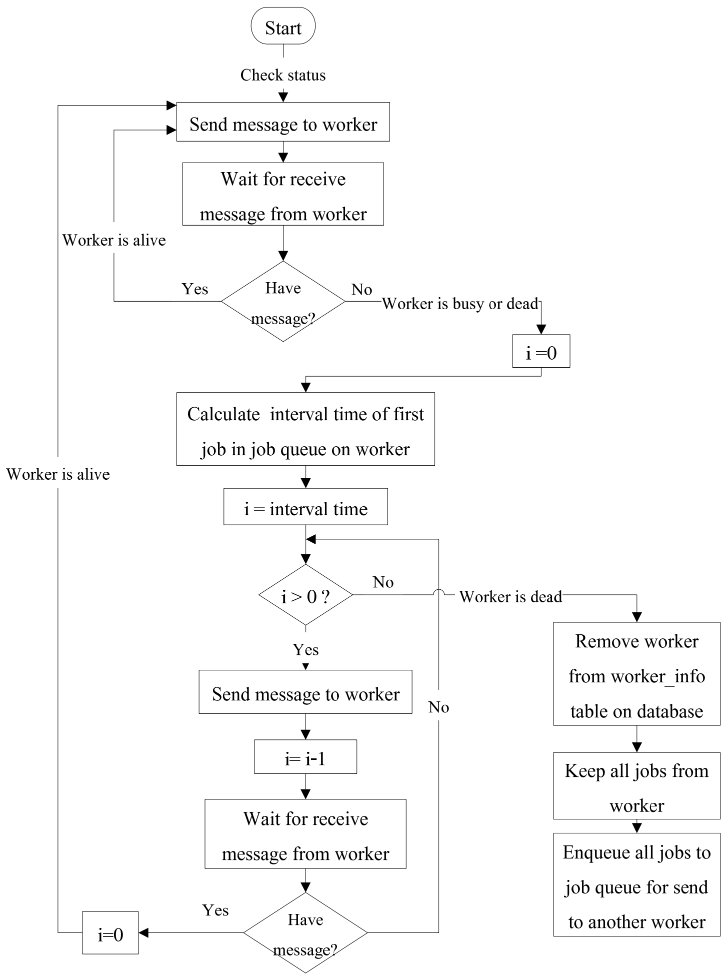
Start
Check status
Send message to worker
Wait for receive message from worker
Worker is alive
Yes
Have message?
No
Worker is busy or dead
i =0
Calculate interval time of first job in job queue on worker
Worker is alive
i = interval time
i > 0 ?
No
Worker is dead
Yes
Send message to worker
No
Remove worker from worker_info table on database
i= i-1
Keep all jobs from worker
Wait for receive message from worker
Enqueue all jobs to job queue for send to another worker
Yes
i=0
Have message?

Figure 4. Process of Fault-Tolerance Management Service

Fig. 5 illustrates the concept behind the software development paradigm used. To developed a cloud application, programmer must decompose the application is to a set of CCR/DSS services and cloud application that integrate and coordinate these services together. The decomposition of service is usually very straight forward in CCR system. An easy to use services development environment (as shown in Fig. 6) that comes with the system can dramatically help shorten the development time.

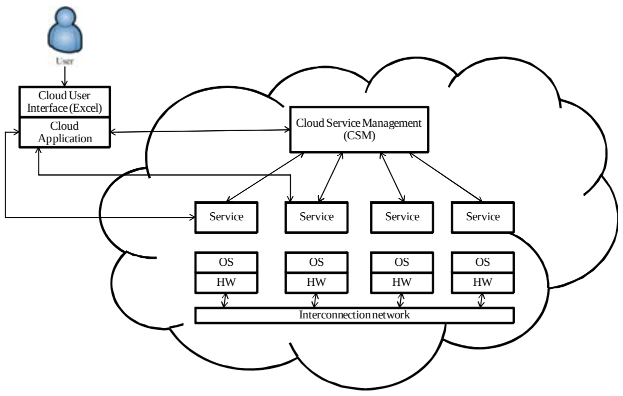

Figure 5. Cloud system configuration

When user application needs resources for the execution, *cloud application* will send the request to CSM. Then, CSM will discover and allocate a number of services (resources) to users in an on-demand basis. After CSM finally allocates services, it will send services information back to Cloud application. Finally, Cloud application can directly communicate with a group of services and coordinate the execution of services to solve the analysis problem. User can view the results using UI component for user interaction.

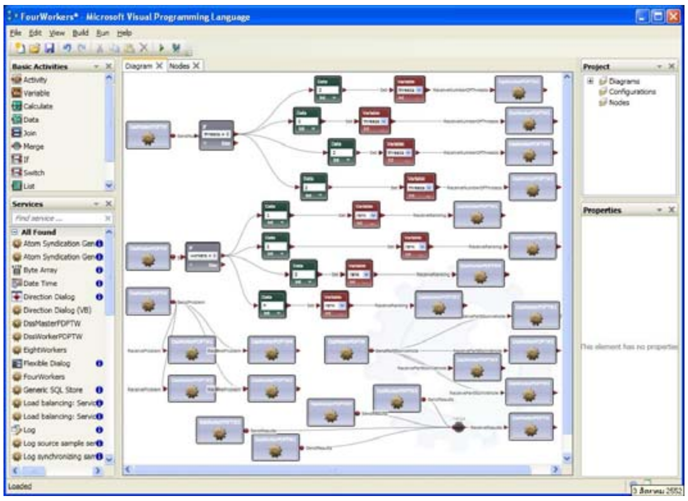

Figure 6. CCR Service development environment

4. The Design of a Proof-of-Concept Logistics Analysis Application

The Pickup and Delivery Problem with Time Windows (PDPTW)[10] is chosen as a proof-of-concept application. This PDPTW problem is a problem of serving a number of transportation requests based on a limited number of vehicles. Fig. 7 shows the concept of this problem.

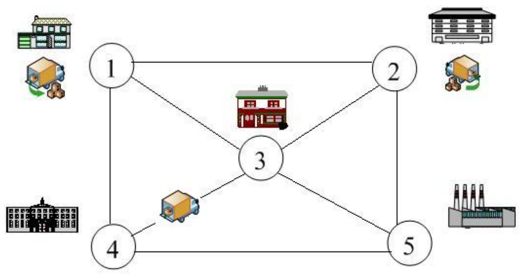

Figure 7. PDPTW problem

For this problem, a number of requests are available for a fleet of vehicles. A request consists of the order to pick up goods at one location and deliver these goods to another location. Two time windows are assigned to each request: pickup time window that specifies when the goods can be picked up and delivery time window that tells when the goods can be dropped off. Service times are associated with each pickup and delivery. The vehicle is permit to arrive at a location before the start of time window, but the vehicle must wait until the start time window initiating the operation. The objective of the problem is to minimize the sum of the distance traveled by the vehicles and minimize the sum of the time spent by each vehicle. This problem is very common in logistics application and also a highly compute intensive optimization problem [9]. The following steps are used to design and build this software based on CCR/DSS.

- Master/ Worker model [11] is adopted as a framework for service interaction
- The algorithm is partitioned using domain decomposition approach [13]. Since the target is to find the minimal number of vehicles that can satisfy the time windows condition, we decided to partition the computation based on a number of vehicles used. The lists of vehicles is generated and then send to worker so each worker can perform a computation separately based on a number of vehicles received
- The software is constructed as services on the cloud. Cloud application control the decomposition of the problem by sending each sub problem to worker service and collect the results back to find the best answer

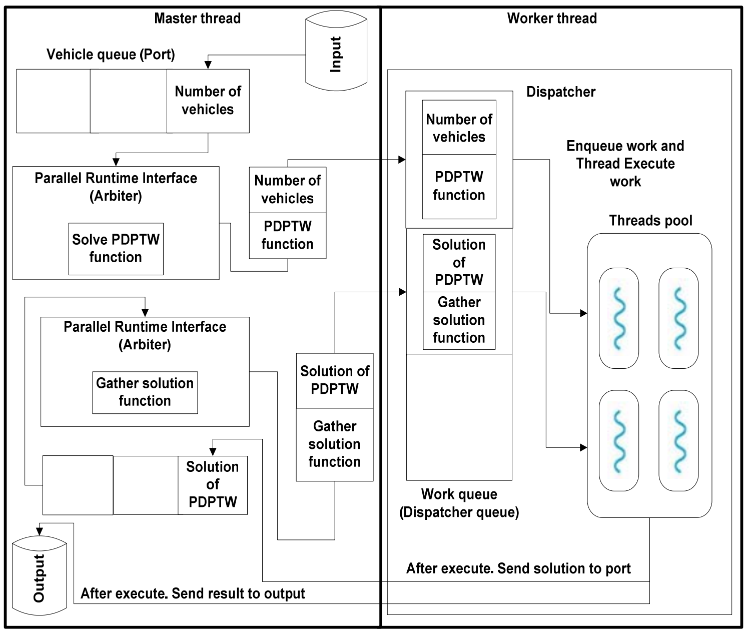

Figure 8. The CCR based computation structure

From Fig. 8, the flow of execution can be explained as follow. First, worker service creates a threads pool and all of the components mentioned earlier. Second, as the master service send work request to worker in form of a set of vehicle used, the work request will be posted on the vehicle queue. Third, arbiter take item from vehicle queue and attached this item with PDPTW method and write it to work queue. Forth, the dispatcher will regularly select the work request from the work queue to be executed on one thread from threads pool. Fifth, when any thread finishes its processing, the result is send to solution queue. Finally, another arbiter will wait for the completion of every work, and then find the best solution to be send back to master service. Master service will select the best result from all worker and output as the result of the execution.

One of the strength of this approach is the seamless integration of a user interface. In this work, we also developed an interface to Microsoft Excel. So, users can fill in a simple form in Excel, push a button created by VBA macro, and the parallel execution on the cloud will be started automatically. The screen shot of our user interface is depicted in Fig. 9 (a) and (b).

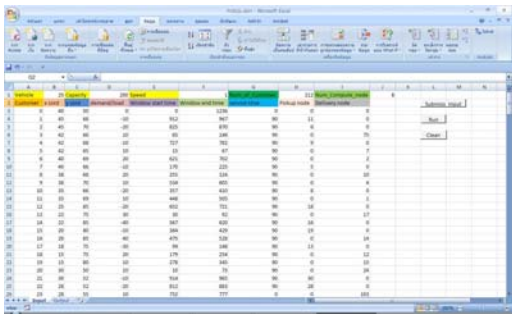

Figure 9. (a) Excel input sheet

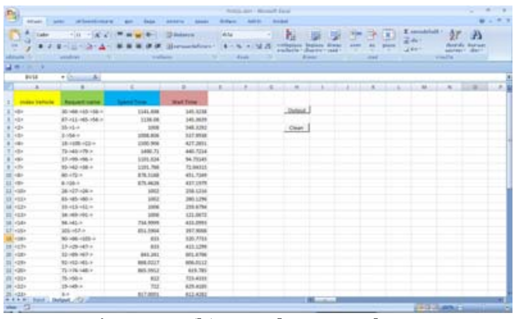

Figure 9. (b) Excel output sheet

5. Experimental Results

In this work, we divide the experiments into two parts. First, we test on two quad-core processors machine. And then we test on four processors machine. Experimental results are show following

5.1. Experimental Results on Personal Computer Cluster

For the experimental setup, we have setup and tested the parallel program and sequential program on a multicore machine. We used three machines in this works. Master machine consist of a single core single processor, 512 MB of main memory, and 80 GB SATA disk. Workers machine consist of two quad-core processors, 1 GB of main memory, and 200 GB SATA disk. Each machine is equipped with Microsoft Windows XP SP2 operating system, Microsoft Visual Studio 2008, Microsoft .NET framework SDK and Microsoft Robotics Developer Studio 2008. In our implementation, we write both the parallel and sequential program in C# language and use the DSS and CCR library for parallel computing. We use data from [8] as a test problem for our implementation.

To evaluate the work, we divide the experiment into two parts. First, we ran a series of tests on 1 machine, using 1 to 512 threads. Then, we ran a series of tests on 2 machines, using 2 to 512 threads. For two machines, we separated the test into 2 strategies as are following:

- Half-Half strategy: We divide half of threads between each machine
- Fill-Spill strategy: We fill the threads up to the number of core of the first machine, and then put the rest of the thread on another machine

We measured both the total execution time of sequential program and parallel program for PDPTW problem. We repeated each experiment at least 3 times and use the average value as the results. The results are as listed in Table 1 and Table 2

Table 1. Runtime (in minute) using 1 to 512 threads on 1 compute machine

Problem Sizes (locations)	**Number of threads**									
	1	***2***	***4***	***8***	***16***	***32***	***64***	***128***	***256***	***512***
212	0.33	0.16	0.09	0.05	0.05	0.05	0.05	0.05	0.06	0.06
530	16.93	8.53	4.34	2.28	2.26	2.29	2.40	2.41	2.43	2.54
848	146.29	78.59	43.61	31.64	24.63	19.25	19.47	19.69	19.86	20.09

Table 2. Runtime (in minute) of Half-Half and Fill-Spill using 2 to 512 threads on 2 compute machines

Threads	**Problem size (locations*)***					
	212		***530***		***848***	
	h-h	**f-s**	**h-h**	**f-s**	**h-h**	**f-s**
2	0.17	0.16	8.57	8.53	73.81	78.59
4	0.09	0.09	4.32	4.34	37.01	43.61
8	0.05	0.05	2.2	2.28	18.65	31.64
16	0.02	0.02	1.13	1.13	9.6	9.6
32	0.02	0.02	1.13	1.13	9.72	9.46
64	0.03	0.03	1.16	1.17	9.85	9.97
128	0.03	0.03	1.18	1.18	9.88	10.05
256	0.03	0.03	1.23	1.12	10.04	10.22
512	0.03	0.04	1.23	1.21	10.18	10.33

From Table 1 and Table 2 speedup and efficiency of the execution can be plotted. The results are as shown in Fig. 10, Fig. 11 and Fig. 12.

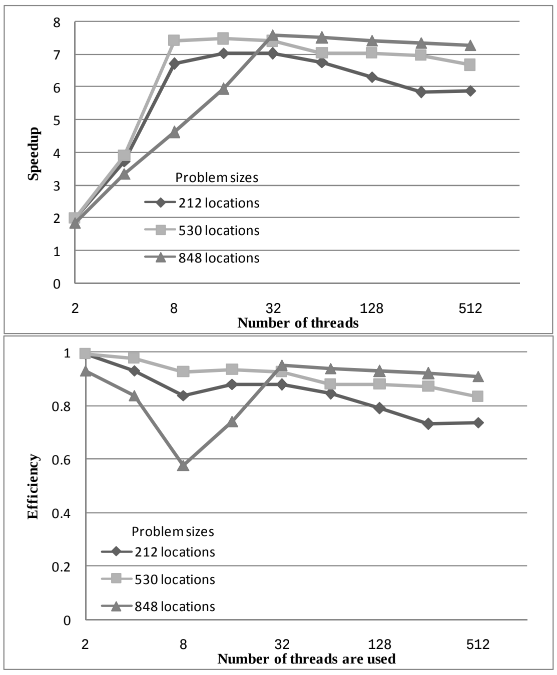

Figure 10. Speedup and efficiency derived from average runtime on 1 machine

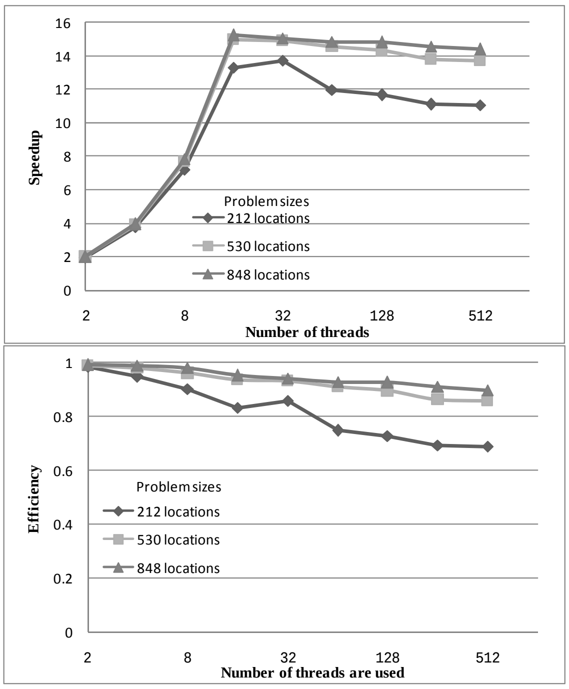

Figure 11. Speedup and efficiency derived from average runtime on 2 machines (half-half)

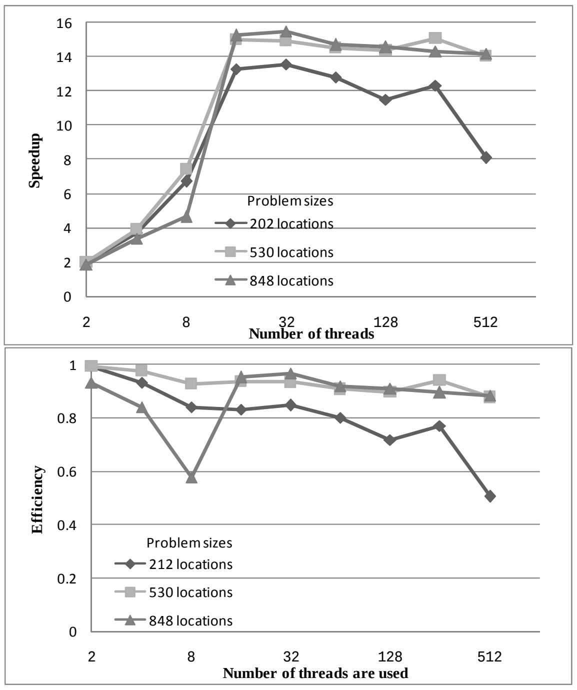

Figure 12. Speedup and efficiency derived from average runtime on 2 machines (fill-spill)

Fig. 10, Fig. 11 and Fig. 12 depicts the speedup compute from the execution time obtained in Table 1 and Table 2. First, from Fig. 10, it can be seen that our implementation shows a substantially better speed up as the number of threads increases and the problem size increases. For single machine, we can get speed up close to 8 which is a number of hardware core on a single machine. The 80%-90%

efficiency is also achieved. Hence, the implementation can run well on a multicore/single machine. The efficiency decreases slower for larger scale due to additional parallelism in larger problem.

Fig. 11 and Fig. 12 shows the speedup of half-half and fill-spill strategy. The result shows that half-half strategy seems to perform better than fill-spill strategy. One interesting fact that number of threads that we can obtain fastest speed is usually much greater than number of hardware core. This is the results of operating system scheduling. We suspect that if number of threads is not large enough, other background tasks is still getting a lot of time from CPU. As number of threads increases, majority of running process will belong to our application. Therefore, the result is the increases in performance. Anyway, as we increase the number of threads pass that point, the speed will be lower due to the additional thread overhead.

5.2. Experimental Results on Teraflop Machine

For the experimental setup, we have tested on Windows HPC at Thai National Grid Center (TNGC), we have setup and tested the parallel program and sequential program on a multicore machine. We used 17 machines in this works. 1 frontend and 16 compute machines, each machine consist of 4 processors, 8 GB of main memory. Each machine is equipped with Microsoft Windows HPC Server 2003 operating system, Microsoft Visual Studio 2008, Microsoft .NET framework SDK and Microsoft Robotics Developer Studio 2008. In our implementation, we write both the parallel and sequential program in C# language and use the CCR and DSS library for parallel computing. We use data from [8] as a test problem for our implementation.

To evaluate the work, we divide the experiment into 5 parts. First, we ran a series of tests on 1 compute machine, using 1 to 1024 threads. Second, we ran a series of tests on 2 compute machines, using 2 to 1024 threads. Third, we ran a series of tests on 4 compute machines, using 4 to 2048 threads. Fourth, we ran a series of tests on 8 compute machines, using 8 to 4096 threads. Finally we ran a series of tests on 16 compute machines, using 16 to 2048 threads.

We measured both the total execution time of sequential program and parallel program for PDPTW problem. We repeated each experiment at least 3 times and use the average value as the results. The results are as listed in Table 3, Table 4, Table 5, Table 6 and Table 7.

Table 3. Runtime (in minute) using 1 to 1024 threads on 1 compute machine

Problem Sizes (locations)	**Number of threads**										
	1	***2***	***4***	***8***	***16***	***32***	***64***	***128***	***256***	***512***	***1024***
212	0.42	0.21	0.11	0.11	0.11	0.11	0.11	0.12	0.18	0.21	0.24
530	19.68	9.88	5.01	5.04	5.05	5.05	5.16	5.32	5.37	5.41	5.47
848	158.72	79.7	40.73	40.34	40.38	40.39	40.7	40.91	41.21	41.34	41.21

Table 4. Runtime (in minute) using 2 to 1024 threads on 2 compute machines

Problem Sizes (locations)	**Number of threads**									
	2	***4***	***8***	***16***	***32***	***64***	***128***	***256***	***512***	***1024***
212	0.22	0.11	0.06	0.06	0.06	0.06	0.06	0.06	0.09	0.10
530	9.98	5.00	2.54	2.54	2.56	2.58	2.68	2.73	2.71	2.74
848	79.96	40.23	20.32	20.36	20.37	20.41	20.46	20.55	20.84	20.85

Table 5. Runtime (in minute) using 4 to 2048 threads on 4 compute machines

Problem Sizes (locations)	**Number of threads**									
	4	***8***	***16***	***32***	***64***	***128***	***256***	***512***	***1024***	***2048***
212	0.11	0.06	0.03	0.03	0.03	0.03	0.04	0.04	0.03	0.03
530	5.10	2.56	1.31	1.30	1.30	1.31	1.35	1.39	1.42	1.43
848	40.65	20.37	10.3	10.29	10.29	10.34	10.42	10.48	10.53	10.61

Table 6. Runtime (in minute) using 8 to 4096 threads on 8 compute machines

Problem Sizes (locations)	**Number of threads**									
	8	***16***	***32***	***64***	***128***	***256***	***512***	***1024***	***2048***	***4096***
212	0.06	0.03	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.03
530	2.65	1.34	0.68	0.68	0.68	0.69	0.70	0.71	0.73	0.76
848	20.88	10.67	5.29	5.29	5.30	5.31	5.35	5.39	5.47	5.44

Table 7. Runtime (in minute) using 16 to 2048 threads on 16 compute machines

Problem Sizes (locations)	**Number of threads**							
	16	***32***	***64***	***128***	***256***	***512***	***1024***	***2048***
212	0.04	0.02	0.01	0.01	0.01	0.02	0.01	0.01
530	1.44	0.73	0.37	0.37	0.37	0.38	0.38	0.4
848	11.02	5.54	2.83	2.81	2.80	2.82	2.88	2.82

From Table 3, 4, 5, 6 and 7, speedup and efficiency of the execution are plotted in Fig. 13-18.

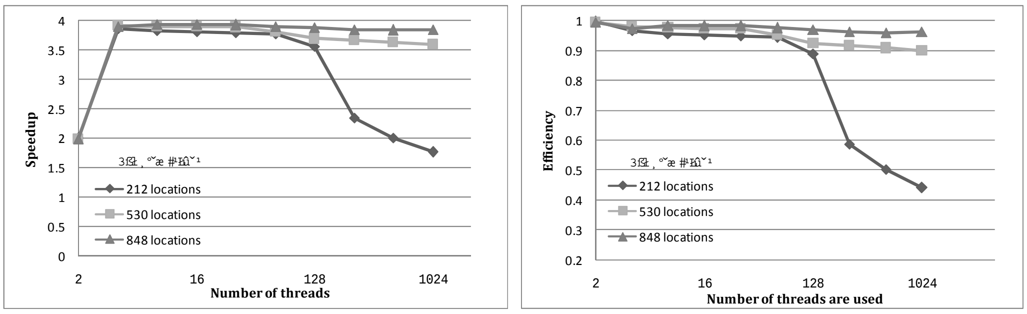

Figure 13. Speedup and efficiency derived from average runtime on 1 machine (4 cores)

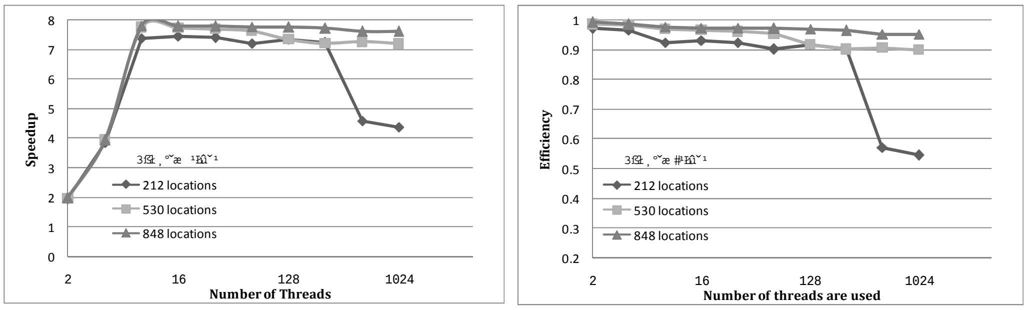

Figure 14. Speedup and efficiency derived from average runtime on 2 machines (8 cores)

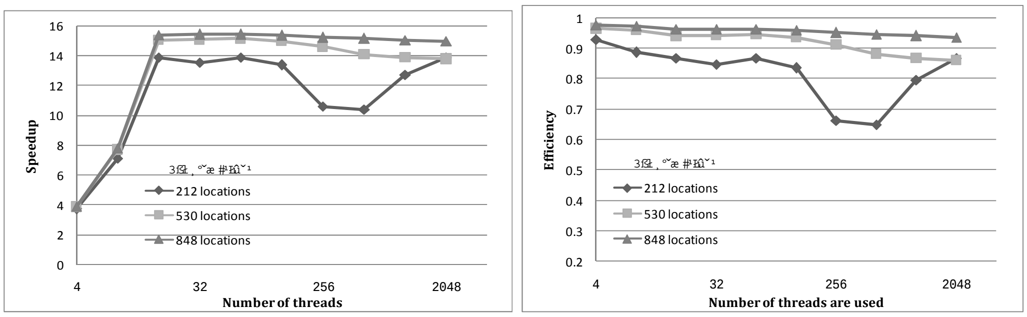

Figure 15. Speedup and efficiency derived from average runtime on 4 machines (16 cores)

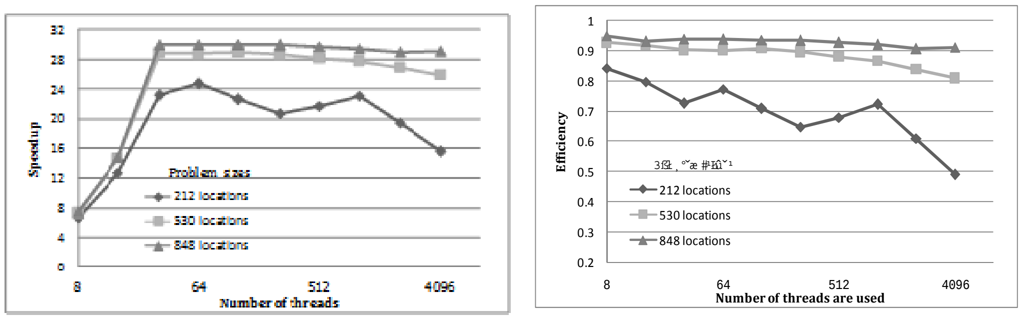

Figure 16. Speedup and efficiency derived from average runtime on 8 machines (32 cores)

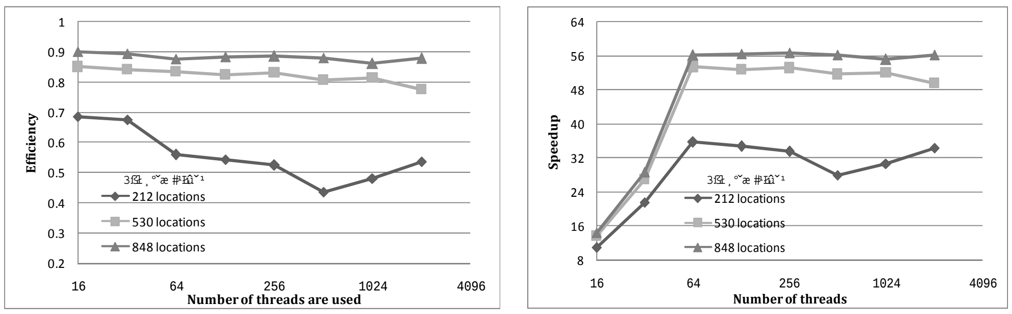

Figure 17. Speedup and efficiency derived from average runtime on 16 machines (64 cores)

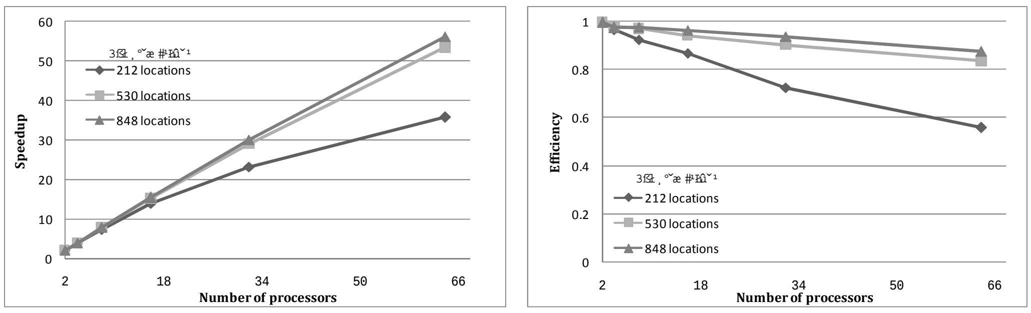

Figure 18. Speedup and efficiency derived from average runtime on 1, 2, 4, 8 and 16 machines

From Fig. 13, it can be seen that our implementation shows a substantially better speed up as the number of threads increases and the problem size increases. For a single machine, we can get a speed up close to 4 which is a number of hardware core on a single machine. The efficiency of 80%-90% is also achieved. Hence, the implementation can run well on a multicore machine.

One interesting things is that the same program that execute well on a single multicore machine can be executed in a cloud environment with a minimal change. As the number of core increases from 4 to 64 cores, the speed of the application is still scale well with the system size. In these experiments, we try to create more number of threads than number of hardware core exists. We found that the maximum attainable speedup happens when number of threads are equal to number of hardware cores except for a small problem size that speedup can be slightly increase. This may caused by the interference from operating system scheduling and external factors. As the system size grows from one to 16 machines, efficiency decreases. This is caused by more communication on the interconnection network that is slower than internal memory bus. Anyway, from Fig. 18, we can see that the system still maintain an efficiency higher than 80% on 64 distributed core for large problem. Hence, the implementation is scale well for this type of problem.

6. Conclusion and future work

In this paper, we propose an architecture and software approach that can be used to built a *Service Oriented Cloud Computing System (SOCCS)* using Microsoft CCR/DSS system. We also show that the business application that is built according to this concept can scale well from a single multicore computer to multiple computers. The experimental results demonstrate the execution speed that is substantially faster for the target application which is a high performance Pickup and Delivery Problem with Time Windows (PDPTW). The use of Microsoft CCR/DSS to develop parallel application has many advantages such as more robustness, seamless integration with many windows based application development model. Moreover, a CCR/DSS application can potentially be deployed on a large clustering environment with a minimal change from multicore implementation.

In the future, more scalable and better *Cloud Service management (CSM)* is planned. This part is still a single point of failure and performance degradation. Thus, some distributed scheme must be applied to improve this vital component.

7. References

[1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg and I. Brandic. (2009, Jun). Cloud computing and emerging IT platforms: Version, hype, and reality for delivering computing as the 5th utility. 25(6), pp. 599-616. Available: http://www.sciencedirect.com/science/journal/1067739X

[2] G. Chrysanthakopoulos and S. Singh,“An Asynchronous Messaging Library for C#,” presented at Synchronization and Concurrency in Object-Oriented Languages(SCOOL) at OOPSLA 2005 Workershop. October 16.

[3] J. Dongarra, D. Gannon, G. Fox and K. Kennedy. (2007, Feb). The Impact of Multicore on Computational Science Software. CTWatch Quarterly[Online]. Available: http://www.ctwatch.org/quarterly/articles/2007/02/the-impact-of-multicore-on-computational-science-software/

[4] H. F. Nielsen and G. Chrysanthakopoulos. (2007,July). Decentralized Software Services Protocol-DSSP/1.0. Microsoft Corp. [Online]. Available: http://purl.org/msrs/dssp.pdf

[5] X. Qiu, G. Fox, and A. Ho,”Analysis of Concurrency and Coordination Runtime CCR and DSS,” Anabas,Inc., Tech. Rep. Jan. 21, 2007.

[6] X. Qiu, G. C. Fox, H. Yuan, S. H. Bae, G. Chrysanthakopoulos and H. F. Nielsen, “High Performance Multi-Paradigm Messaging Runtime Integrating Grids and Multicore Systems,” in 3rd IEEE International Conference on e-Science and Grid Computing Conf., 2007, pp.407-414.

[7] J. Richter. (2008, May. 15). Concurrent Affairs: Concurrency and Coordination Runtime. MSDN Mag. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc163556.aspx

[8] H. Li and A. Lim. (2004, Feb.). Benchmarks Vehicle Routing and Travelling Salesperson Problems. [Online]. Available: http://www.top.sintef.no/vrp/benchmarks.html

[9] H. Li and A. Lim, “A Metaheuristic for the Pickup and Delivery Problem with Time Windows, “in 13th IEEE Conference on Tools with Artificial Intelligence Conf., ICTAI-2001,pp.160-170.

[10] S. Ropke and D. Pisinger,(2006, Nov). An Adaptive Large Neighborhood Search Heuristic for the Pickup and Delivery Problem with Time Windows. In Transportation Science,40(4), pp. 455-472.

[11] M. J. Quinn, Parallel Programming in C with MPI and OpenMP, international Ed. New York: McGraw-Hill, 2003.

[12] L. M. Vaquero, L. Rodero-Merino, J. Caceres and M. Lindner, “A Break in the Clouds: Towards a Cloud Definition”,vol 39, pp.50-55, Jan. 2009.

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

[14] I. Foster, Yong. Zhao, I. Raicu and S. Lu, “Cloud Computing and Grid Computing 360-Degree Compared”, Grid Computing Environment Workshop, Nov, 2008.