*Farm Skeleton with Dynamic Task Allocation Using Task Mobility*

Turkey Alsalkini

Heriot-Watt University

Edinburgh, UK

ta160 @ hw.ac.uk

Gregory Michaelson

Heriot-Watt University

Edinburgh, UK

G.Michaelson @ hw.ac.uk

Abstract—**Parallel computing exploits the available resources in the network. The Grid has many resources, the location of which may be unknown. The Grid also has many users that can benefit from these resources, so the load is changeable and unpredictable. We need a mechanism to balance the load in the network locations during runtime by moving the computations between the network locations for better use of resources of the network. In this paper, we present a generic function by using algorithmic skeleton concept, a function called skeleton, that is able to move the task between processing items during runtime at the heterogeneous architecture; therefore, when network location is unable to complete the work, moving tasks will take place from source location by saving processing state to a new suitable location to continue processing the task.**

Keywords: Skeleton ,Parallel, Mobile, Grid, Computing, Load balancing.

# Introduction

In recent years, there has been a dramatic increase in the amount of available computing and storage resources, but special High-Performance Computers are expensive and rare resources. Emerging Multiprocessor Architecture techniques, which are one or more than one processor capable of communicating with each other, offer the opportunity to integrate enabled high-performance computers into a single high-performance computer. Using Multiprocessor Architecture entails several technical challenges: difficulty of effective utilization, high communication latency, and unpredictable effective speeds.

Researchers are investigating the possibility of exploiting the computational power and resources available in global networks. Mobile Computation is a way to use the resources available on both local and global networks. Mobile Computation gives the programmer control over the placement of code or active computations across a network to chart computational resources and better use the available resources. A mobile program can transport its state and code to another location where it resumes execution [27], so in the application that uses mobile computation, the program can move between locations for better utilisation of computational resources. By using load management systems, the program has a technique for distributing the worker location to achieve performance goals (balancing the load or minimising the execution time).

The main obstacle to the commercial uptake of parallel computing is the complexity and cost of the associated software development process. A promising way to overcome the problems of parallel programming is to exploit generic programs structures, called skeleton [17]. Skeletons capture common algorithms which can be used as components for building programs. The main advantage of the skeleton approach is that all the parallelism and communication are embedded in the set of skeletons.

# Background and related work

Mobility, which refers to the change of location achieved by system entities [10], is moving the computations among processors on a network which leads to a better use of resources and distributing the load over processors for a faster performance [25, x26]. Mobility has different forms: hardware and software mobility, process migration, mobile languages, weak and strong mobility. Hardware mobility means the mobility of devices, such as laptops and PDAs. Therefore, software mobility moves the computations from one location to another location [6]. Mobile languages and process migration are classification of software mobility. In process migration, the system decides the moving operation, i.e. MOSIX [4], which is operating system that supports process migration; nevertheless, in mobile languages, the system gives the programmer ability to decide the moving operation. Weak and strong mobility are forms of mobility defined by Fuggutta and Picco and Vigna [5]: Weak mobility is moving the code from one location to another. Strong mobility is moving the code and state information from one location to another and resuming the execution from the stop state [26]. Most mobile languages support weak and strong mobility such as JavaGo [2], but Java Voyager [3] supports only weak mobility, strong mobility is also known as transparent migration. Checkpointing is the main operation in mobile systems to move computations among processors in a network or cluster by snapshotting the state of application [14], an example of system using checkpoint is CONDOR [11].

A novel decentralised load management technique has been developed by Deng [x2], this technique is called AMP, Autonomous Mobile Program. The AMPs seek for better location and take a movement decision depending on whether the resource needs can be served locally or on another location. The movement decision on AMPs also depends on future resource needs, and if better to serve locally or move to another location.

The skeleton, by Cole, is an approach in parallel programming to abstract the complexities that exist in the parallel implementations[x12]. Algorithmic skeletons are common parallel programming patterns to avoid the parallel and communications details for the programmer so that he/she is not responsible for the synchronization between the application parts. The main advantages of using skeletons are having a higher order programming interface and general implementation for portability and efficiency.

The skeleton is closely related to functional languages, so higher order functional structures can be produced by using skeletons[9]. Each skeleton has implicit parallel implementation hidden from the application user; thus, the main advantage of skeletons is that the communication and parallelism details are embedded in the skeleton. The skeletons are polymorphic higher order functions, so that there are various kinds of skeletons to cover different programs over different data types [13]. These functions are implemented by libraries. Many implementations of computations on distributed and parallel architectures are supporting skeletal libraries. These libraries offer task parallel and data parallel skeletons. An example of C library with MPI function is eSkel[18], and an example of C++ library with MPI function is SkeTo [16].

Google developed a C++ library that offers parallel programming model, called MapReduce [12]. MapReduce skeleton is a programming model to process large sets of data. This model has an abstraction level where it is possible to perform computational operations while hiding communication and parallelism details, fault-tolerance, and data distribution. This model has two primitives *map & reduce*. *Map* operation applies the function on pairs of key/value to produce output key/value; *reduce* operation combines the shared key results to produce the final result. Map function is written by the user, and the user specifies the data sets with pairs. Similarly, reduce function is also written by the user. Apache Hadoop is a Java library used to process large data sets on distributed parallel architecture such as cluster [1].

A cost model is a performance model used to estimate the costs of programs such as time and space[24, 7]. Algorithmic skeletons involve the parallelism process, communication and synchronisation and cost complexity [8]. The cost models of algorithmic skeletons measure the computation cost and communication cost for skeletons. Many cost models have been developed for algorithmic skeletons on parallel architecture. Some models determine the task placement statically [15], while some models determine the whole skeleton placement dynamically[x19]. Our research proposes the dynamic task placement on skeletal programming.

We will propose a parallel skeleton implemented (*farm* skeleton) using C with MPI functions. This skeleton is able to move tasks between workers with preserving the execution state during moving operation. We have three approaches to implement the mobility on our skeleton: data mobility, data and state mobility, and data and state and code mobility.

Data mobility means moving the data between locations on a network[x10]. A Data and state mobility is a simple approach of mobility when the application can correctly save its state to a form and resume work from the saved point properly. Code mobility approach is moving the code, data of computation and the execution state to a different machine[x6]. Task mobility means moving data, state and code of computation to a different worker. This paper proposes the task mobility approach by moving the data and state between workers. The skeleton is implemented using C and MPI, but MPI clones the code to the workers so that no need for moving the code. Code mobility is difficult to implement in heterogeneous structure, and this is a future work for our research but now we implemented this skeleton as a first step in implementing a skeleton able to move code amongst machines on a network.

# An overview of hwFarm skeleton

The research aims to build high-level parallel programming model, known as skeleton that captures farm pattern which is a type of parallel patterns, and execute the skeleton on a high performance system such as cluster. Our skeleton has the name *hwFarm.*

In general, the main idea of skeleton is to abstract all the parallelism and communication details, but the *hwFarm* skeleton has another property which makes it able to move tasks amongst the workers during run-time.

The hwFarm skeleton has the following properties:

* *hwFarm* skeleton is a self-mobile skeleton which means that our skeleton is able to mobilize the task from one worker to another one during task execution when the overhead increases.
* *hwFarm* supports parallelism on distributed memory, high-performance architecture.
* Parallelism and communication details are implicit in the skeleton and hidden from the programmer.
* *hwFarm* skeleton presents a high-level function implemented using C and MPI [20].

##  Definition of hwFarm skeleton

The term task farming is used to describe parallel applications that have specific properties. Ordered and structured collection of data items, known as tasks, is processed by the same operation. Processing the task can be performed in parallel because the tasks are independent [19]. The static scheduling of tasks to a similar number of processes is poor load balancing. A task farm solves previous problems by implementing dynamic scheduling to ensure a better balance. The farmer represents the scheduler while the workers process the tasks assigned by the farmer. The h*wfarm* skeleton has the same characteristics but with the ability to make its task able to move amongst workers.

The implementation is divided into three steps:

1. *Implementing skeleton with workers without mobility*: This is a simple skeleton which contains farmer responsible for distributing the tasks to the workers, and workers executing these tasks, as shown in “Fig. 1”.
2. *Implementing skeleton where data has been sent between two workers*: This is the first step in making a task mobile, but the task will be processed by the worker from the beginning.See “Fig. 2”.
3. *Implementing skeleton where data and state have been sent between two workers*: The skeleton can move the data and state of the task between workers. The moved elements contain the processed data and unprocessed data, so the target machine will start processing not from the begging of the data but from the begging of the unprocessed data. An explanation of implementation will be given in the next section. See “Fig. 3”.



1. Standard skeleton

## How to move state?

The mobility process needs to save the state of execution to continue working form the stop state. The skeleton avoids the parallel and communication details for the programmer so that the programmer is not responsible for the synchronisation between application parts. The programmer has to write his own general function that should be executed in skeleton, this function has three arguments: input data that should be processed, output data which is the result and parameters of the user function. Array of parameters contains the variables **that the function needs to process the data. The state of** execution depends on the values of the parameters. During mobility process, the skeleton moves the input data that does not processed by the first worker, the subresult of the processed data and the array of parameters for the function at stop point. The new worker receives this data and continues processing these data from the stop point.



1. Skeleton with task mobility (data only)

##  Movement decision:

One of the biggest issues in parallel and distributed systems is developing techniques for distributing the processes on multiple locations[x19, 23]. The problem is how to distribute the processes amongst processing elements to minimize the execution time and increase performance. Our skeleton is balancing the load by depending on information collected from machines at run time to move the task from heavily loaded machines to lightly loaded machines.

The movable element in our skeleton is the task. The task is computing the function for specific data so that in movement operation we should move the function and the data. Since the function already exists in all workers, the movement operation is only moving the data and state between workers.

Moving decision on the skeleton has several polices:

* Information policy: Determines the load information to make a task placement or task mobility. This information is collected from the machines at runtime to know the load changes on these machines.
* Selection policy: Decides what task should be moved. The movement decision is taken by collaboration between master and workers. The worker decides if its task should be moved depending on its load and the load on free workers. When the worker decides to move its task, it sends a request to the master.



1. Skeleton with task mobility (data and state)
* Placement policy: Identifies to which a task should be transferred. The worker after deciding that it is unable to process its task, or becomes heavily-loaded, it decides on the best free worker available to process the task.

## Activities of hwFarm skeleton:

The activities that may happen during the execution of the skeleton:

* Getting the load of all workers in a system.
* Choosing the best workers where the number of workers is static. The system load in worker, also known as load average, is the measure of the amount of work that a computer system performs. The load average represents the average system load over a period of time. It appears in the form of three numbers which represent the system load during the last one, five-, and fifteen-minute period [21]. The hwFarm skeleton assumes that the load is first number (in last one-minute).
* Each worker will send load information to the master, and then the master will send the load information to all workers.
* Distributing the tasks to the worker chosen to execute the tasks.
* The master (farmer) awaits the results from workers and distributes new tasks.
* When the master receives a result, it will check the load for all free workers and then choose the best one to send the task to it.
* Depending on its load and percentage of increased load in this worker and the load on free workers, the worker decides if the task should be moved to a free worker or not.
* Moving the task from one worker to a new worker, and the destination worker continues executing the task from the stop point.

# Experiments

In these experiments, we evaluate the performance and the behaviour of the hwFarm skeleton on heterogeneous distributed memory architecture. We use Ray tracer program that generates the image for 100 rays for 150000 objects in the scene. Ray tracing algorithm is used to produce an image by imaginary rays of light from the viewer’s eye through pixels to the objects in the scene [22].

## Platform:

The hwFarm skeleton is tested with Beowulf cluster located at Heriot-Watt University. Beowulf cluster consists of 32 eight-core machines (8 quad-core Intel(R) Xeon(R) CPU E5504, running GNU/Linux at 2.00GHz with 4096 kb L2 cache and using 12GB RAM).

## Evaluation:

The skeleton is tested on 4 modes:

* *Static task allocation*: The skeleton is placing the tasks without using load information; we will refer to this mode as *Static*. See “Table 1”.
* *Dynamic task allocation*: The skeleton depends on load information collected from machines for placing the task but without mobility; we will refer to this mode as *Dynamic*. See “Table 2”.
* *Dynamic task allocation with load and no mobility*: The skeleton uses the load information for placing the task but without mobility. In this case, there are some loads that will be applied on different periods on some workers; we will refer to this mode as *Load*. See “Table 3”.
* *Dynamic task allocation with load and mobility*: The skeleton balances the load by moving the task from heavily loaded workers to lightly loaded workers where random loads are applied on the workers; we will refer to this mode as *Mobility*. See “Table 4”.

The results of these experiments presented in these tables:

1. Static task allocation time(sec)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  TasksWorkers | 1 | 2 | 3 | 4 | 5 |
| 1 | 186.363 | 184.034 | 188.476 | 187.443 | 188.626 |
| 2 | 186.681 | 97.948 | 121.437 | 97.304 | 110.800 |
| 3 | 187.031 | 97.580 | 68.674 | 88.795 | 74.177 |
| 4 | 186.276 | 97.411 | 69.121 | 50.945 | 71.665 |
| 5 | 186.321 | 97.707 | 68.602 | 51.323 | 43.105 |

This results in Static task allocation mode show that our skeleton has good speed up to solve Ray tracer program.

1. Dynamic task allocation time(sec)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  TasksWorkers | 1 | 2 | 3 | 4 | 5 |
| 1 | 193.168 | 186.973 | 185.559 | 187.703 | 185.978 |
| 2 | 193.227 | 96.788 | 120.203 | 93.854 | 107.920 |
| 3 | 193.664 | 97.462 | 70.462 | 87.707 | 74.052 |
| 4 | 194.076 | 98.460 | 69.216 | 53.102 | 71.751 |
| 5 | 193.043 | 98.474 | 69.074 | 52.977 | 43.500 |

This results in Dynamic task allocation mode show that our skeleton has good speed up but the difference in result comes from collecting and computing load information.

1. Dynamic task allocation with load and no mobility time(sec)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  TasksWorkers | 1 | 2 | 3 | 4 | 5 |
| 1 | 335.513 | 292.372 | 316.453 | 309.019 | 294.252 |
| 2 | 332.121 | 166.745 | 192.063 | 154.189 | 162.842 |
| 3 | 297.927 | 167.259 | 113.712 | 140.200 | 121.476 |
| 4 | 298.318 | 163.596 | 116.965 | 95.658 | 80.906 |
| 5 | 300.889 | 151.549 | 110.554 | 93.430 | 79.428 |

This results show that the skeleton has a speed up but the load on worker affected on the performance and the execution time became slower.

1. Dynamic task allocation with load and mobility time(sec)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  TasksWorkers | 1 | 2 | 3 | 4 | 5 |
| 1 | 329.445 | 315.098 | 312.860 | 311.969 | 291.501 |
| 2 | 195.449 | 137.300 | 131.514 | 135.443 | 121.701 |
| 3 | 196.673 | 127.245 | 103.468 | 93.554 | 94.028 |
| 4 | 197.781 | 109.565 | 92.112 | 71.096 | 73.512 |
| 5 | 197.454 | 105.680 | 83.069 | 69.257 | 62.329 |

This table shows that the performance in this mode is better than previous mode; therefore, mobility gives us better performance when some workers became heavily loaded, but the performance is worst than *static* and *dynamic* mode.

The graphs show the execution time of the program with different numbers of tasks on different numbers of workers.

1. 1-5 Workers, 1 Task

This graph shows the time for executing 1 task on 1 - 5 workers. The execution time in *static* and *dynamic* modes is approximately the same; the difference comes from the cost of computing and collecting load. The execution time in *load* mode depends on load applied on workers. In *mobility* mode the task will be moved to a free worker when the current worker becomes unable to process the task, so the execution time will be smaller. The task will not be moved when there are no free workers.

1. 1-5 Workers, 2 Task

This graph shows the time for executing 2 tasks on 1 - 5 workers. In *static* and *dynamic* modes the execution time is approximately the same. The improvement on *mobility* mode comes from moving the tasks from heavily loaded workers to lightly loaded workers.

1. 1-5 Workers, 3 Task

This graph shows the time for executing 3 tasks on 1 - 5 workers. The improvement of execution time on *mobility* mode is related to the availability of free workers.

1. 1-5 Workers, 4 Task

This graph shows the time for executing 4 tasks on 1 - 5 workers. The time in *mobility* mode approaches the time on *static* and *dynamic* mode.

1. 1-5 Workers, 5 Task

This graph shows the time for executing 3 tasks on 1 - 5 workers.

We can conclude from our experiments that:

* These experiments show a good speed up in executing the problem.
* There are no free workers; that means we will not get benefit from the mobility
* The execution time depends on the amount of load and the duration of load on workers.
* Moving process depends on load on the worker which process the task and the worker which is free and able to process the task in a faster time.

# Conclution and future work

We proposed a new type of skeleton for high performance, distributed memory architecture. This skeleton is implemented using C and MPI library. This skeleton is self-mobile which is able to move tasks from a heavily- overloaded worker to a lightly-overloaded worker.

The movement decision will be taken by collaboration between the master and workers. Some parameters for this skeleton are static. Our experiments showed that when applying load on a busy worker, the movement will happen to a best free worker, and this worker will continue executing the task from the stop point. Therefore, we anticipate having a good parallel performance with our skeleton.

In future work, the hwFarm skeleton will be extended to be able to move a code amongst processing units. In addition, we will define cost model to the skeleton to study the efficiency of moving the task on improving performance.

##### References:

1. Apache Hadoop Project. *http://hadoop.apache.org/*, 2010.
2. T. Sekiguchi. “JavaGo”, *http://homepage.mac.com /t.sekiguchi/javago/index.html*, May 2006.
3. “voyager user guide,” ”<http://www.recursionsw.com>”, 2005.
4. A. Barak, S. Guday, and R. G. Wheeler, *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1993.
5. A. Fuggetta, G. P. Picco, and G. Vigna, “Understanding code mobility,” *IEEE Trans. Softw. Eng*., vol. 24, no. 5, pp. 342–361, May 1998.
6. A. R. D. Bois, “Mobile Computation in a Purely Functional Language,” Ph.D. thesis, School of Mathematical and Computer Science, Heriot-Watt University, United Kingdom, Aug 2005.
7. A. Merlin and G. Hains, “A generic cost model for concurrent and data-parallel meta-computing,” *Electronic Notes in Theoretical Computer Science*, vol. 128, no. 6, pp. 3 – 19, May 2005.
8. D. K. G. Campbell, “Clumps: A Candidate Model Of Efficient, General Purpose Parallel Computation,” Ph.D. thesis, Department of Computer Science, University of Exeter, United Kingdom, Oct 1994.
9. F. A. Rabhi and S. Gorlatch, eds., Patterns and skeletons for parallel and distributed computing. London, UK: Springer-Verlag, 2003.
10. G. Cabri, L. Leonardi, and F. Zambonelli, “Weak and strong mobility in mobile agent applications,” in Proc. *2nd International Conference and Exhibition on The Practical Application of Java (PA JAVA 2000)*, Manchester (UK), April 2000.
11. J. Basney and M. Livny, *High Performance Cluster Computing: Architectures and Systems*, Volume 1, ch. Deploying a High Throughput Computing Cluster. Prentice Hall, 1999.
12. J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51,no. 1, pp. 107–113, January 2008.
13. J. Fischer, S. Gorlatch, and H. Bischof, Foundations of data-parallel skeletons, pp. 1–27. London, UK: Springer-Verlag, 2003.
14. J. G. Hansen, “VIRTUAL MACHINE MOBILITY WITH SELF-MIGRATION,” Ph.D. thesis, Department of Computer Science, University of Copenhagen, Apr 2009.
15. K. Armih, Greg Michaelson, and Phil Trinder, “Cache size in a cost model for heterogeneous skeletons,” In *Proc. fifth int. workshop on High-level parallel programming and applications (HLPP '11)*, 2011.
16. K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. "A library of constructive skeletons for sequential style of parallel programming,” In *InfoScale ’06: Proceedings of the 1st international conference on Scalable information systems*, New York, NY, USA, 2006. ACM. ISBN 1-59593-428-6.
17. M. Cole, *Algorithmic skeletons: structured management of parallel computation.* MIT Press, 1989.
18. M. Cole, “Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming,” *Parallel Comput.,* vol. 30, no. 3, pp. 389–406, 2004.
19. M. Cole, eSkel: The Edinburgh SKEleton Library, Tutorial Introduction. Internal Paper, School of Informatics, University of Edinburgh, 2002.
20. M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference.* MIT Press, Cambridge, MA, USA, 1995. ISBN 0262691841.
21. R. Walker, “Examining load average,” *Linux J*, vol. 2006, no. 152, pp. 5--, December 2006.
22. S. Schneider, *Concurrent and Real Time Systems: The CSP Approach (1st ed.).* John Wiley & Sons, Inc., New York, NY, USA, 1999
23. T. L. Casavant and J. G. Kuhl, “A taxonomy of scheduling in general-purpose distributed computing systems,” *IEEE Trans. Softw. Eng.*, vol. 14, no. 2, pp. 141–154, February 1988.
24. X. Y. Deng, “Cost-Driven Autonomous Mobility,” Ph.D. thesis, Heriot-Watt University, United Kingdom, May 2007.
25. X. Y. Deng, G. Michaelson, and P. Trinder, “Cost-driven autonomous mobility,” *Computer Languages. Systems and Structures*. vol. 36, no. 1, pp. 34 – 59, Apr 2010.
26. X. Y. Deng, G. Michaelson, and P. Trinder, “Autonomous mobility skeletons,” *Parallel Comput*., vol. 32, no. 7, pp. 463–478, September 2006.
27. Z. Kirli, “Mobile Computation with Functions,” Ph.D. thesis, University of Edinburgh, Laboratory for Foundations of Computer Science:Division of Informatics, 2001.

***Appendix****:*

**Using *hwFarm* skeleton**

The prototype of hwFarm skeleton is implemented in C and MPI. Our skeleton gives the programmer the ability to write his program in a sequential manner by C, and the programmer should specify the input data and call the high-ordered function which represents our skeleton to run the program with all data in an implicit, parallel manner. Our implementation uses MPI library to provide the communication, so we need to initialise the library before calling the skeleton.

 There are some constraints on the programmer in writing the function where the function has three values: the input data and its length, the output data and its length, and an array of parameters used in function, and these values should be initialised before function.

The main steps to write a parallel program using hwFarm skeleton are:

* Write the sequential code that should be executed in parallel on the data items. The user function is a generic function where the name of the function can be parameterized.
* Init MPI library.
* Init input data.
* Call the hwFarm skeleton.
* Finalise the MPI library.

The prototype of the main function of hwFarm skeleton:

void hwfarm(fp worker, int tasks,

 void \*input,int inSize,

 int inLen, MPI\_Datatype taskType,

 void\* output, int outSize,

 int outLen, MPI\_Datatype resultType, void\*FunPars, int parsSize, int procCount)

 {...}

Glossary of parameters:

worker: worker function.

tasks: total number of tasks.

input: array of input data.

inSize: size of input data type.

inLen: size of one task.

taskType: type of MPI input data.

output: array to output data.

outSize: size of output data type

outLen: size of data in one

resultType: type of MPI output data

FunPars: array of function parameters

parsSize: size of parameters

procCount: number of processors

We assume that chunk size and number of workers are static but may be made dynamic by the skeleton.

The prototype of the general function that user write which will be called from our hwFarm skeleton:

void doProcessing(

 void \*inputData,int inputLen,

 void \*result, int outputLen,

 void\* pars, int parsSize)

 {...}

Glossary of parameters:

inputData: input data.

inputLen: length of input data.

result: output data.

outputLen: length of input data.

pars: array of paremeters.

parsSize: length of parameters.

Each worker will execute his task by calling *doProcessing* function on his data chunk. All variable the function need should be parameterised, we can save the execution state of these function.