Dynamic Farm Skeleton Task Allocation Through Task Mobility

Turkey Alsalkini[1], Greg Michaelson[1]
[1]Computer Science Department, Heriot-Watt University, Edinburgh, UK
(ta160, G.Michaelson) @ hw.ac.uk

Abstract—*Demand for multi-process resource invariably outstrips supply and users must often share some common provision. Where batch-based, whole processor allocation proves inflexible, user programs must compete at runtime for the same resource so the load is changeable and unpredictable. We are exploring a mechanism to balance the runtime load by moving computations between processors to optimize resource use. In this paper, we present a generic algorithmic farm skeleton which is able to move worker tasks between processors in a heterogeneous architecture at runtime guided by a simple dynamic load model. Our experiments suggest that this mechanism is able to effectively compensate for unpredictable load variations.*

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

PDPTA'12

1 Introduction

In recent years, there has been a dramatic increase in the amount of available computing and storage, but dedicated High-Performance Computers are expensive and rare resources. Emerging multiprocessor architecture techniques offer the opportunity to integrate individual high-performance computers into a unitary high-performance system. This 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 and better use the available computational resources. A mobile program can transport its state and code to another location where it resumes execution [27], so in an application that uses mobile computation, the program can move between locations for better utilisation of computational resources. By using load management techniques, the program has a mechanism for distributing the tasks to worker locations 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.

We are exploring a mechanism to balance the runtime load by moving computations between processors to optimize resource use. In this paper, we present a generic algorithmic farm skeleton which is able to move worker tasks between processors in a heterogeneous architecture at runtime guided by a simple dynamic load model. Our experiments suggest that this mechanism is able to effectively compensate for unpredictable load variations.

2 Background and related work

Mobility, which refers to the change of location achieved by system entities [10], involves moving computations amongst processors on a network to distribute the load, giving better use of resources and a faster performance [25, 27]. 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. In contrast, software mobility moves the computations from one location to another location [6], typically through process migration or mobile languages. In process migration, the system determines load movement e.g. MOSIX [4], which is an operating system that supports process migration. In contrast, in mobile languages, the system gives the programmer the ability to control load movement. Weak and strong mobility are alternative forms of mobility defined by Fuggutta and Picco and Vigna [5]. Weak mobility involves moving the code from one location to another. Strong mobility involves moving the code and state information from one location to another and resuming the execution from the stop state [26]. Strong mobility is also known as transparent migration. Many mobile languages support weak and strong mobility, e.g. JavaGo [2], but Java Voyager [3] supports only weak mobility. Checkpointing is the main operation in mobile systems

to move computations amongst processors in a network or cluster by snapshotting the state of application [14], e.g. CONDOR [11].

A novel Autonomous Mobile Program (AMP) decentralised load management technique has been developed by Deng [25]. AMPs seek to execute on "better" locations and take movement decisions depending on whether the resource needs can be served locally or on another location. This movement decision also depends on future resource needs, and whether it is better to continue locally or to move to another location.

Algorithmic skeletons offer an approach in parallel programming to abstract the complexities that exist in the parallel implementations [17]. They are common parallel programming patterns that avoid the parallel and communications details for the programmer so that they are not responsible for the synchronization between the application parts. Skeletons are closely related to functional languages, so higher order functional structures can be produced by using skeletons [9]. Each skeleton has an implicit parallel implementation hidden from the application user. The main advantages of using skeletons are having a higher order programming interface and a general implementation for portability and efficiency.

Skeletons are polymorphic higher order functions, so that there are various kinds of skeletons to cover different program classes over different data types [13]. These functions are implemented by libraries. Many implementations of computations on distributed and parallel architectures support skeletal libraries which offer task parallel and data parallel skeletons. An example of a C library with MPI functions is eSkel [18], and an example of a C++ library with MPI functions is SkeTo [16].

Google developed a C++ library that offers parallel programming model, called MapReduce [12]. The MapReduce skeleton is a programming model for processing 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* and *reduce*. The *map* operation applies a function to pairs of key/value to produce output key/value; the *reduce* operation combines the shared key results to produce the final result. The mapped function is written by the user, and the user specifies the data sets with pairs. Similarly, the reduced function is also written by the user. The closely related open-source Apache Hadoop is a Java library used to process large data sets on distributed parallel architecture such as cluster [1].

A cost or performance model may be used to estimate the costs of programs such as time and space [24, 7]. While algorithmic skeletons involve the parallelism process, communication and coordination [8], their cost models typically measure the computation and communication cost. Many cost models have been developed for algorithmic skeletons on parallel architecture. Some models determine the task placement statically [15], while others determine the whole skeleton placement dynamically [24].

Our approach is based on dynamic task placement for skeletal programming. We have developed a parallel farm skeleton using C with MPI functions which is able to move tasks between workers while preserving the execution state during moving operation. We have explored three approaches to implementing mobility in our skeleton: data mobility, data and state mobility, and data, state and code mobility.

Data mobility involves moving the data between locations on a network [14]. For state mobility, the program can correctly save its state and resume work from the saved point properly. Code mobility involves moving the whole program code, as well as the data and execution state, to a different machine [5]. This paper proposes as task mobility approach of moving the data and state for a sub-computation between processors, rather than the whole program. Our skeleton is implemented using C and MPI, but MPI clones the code to the workers so there is no need for moving the code. Code mobility is difficult to implement in heterogeneous structures, and this remains future work for our research: our work is a first step in implementing a skeleton fully able to move arbitrary code amongst machines on a network.

3 An overview of hwFarm skeleton

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 is also able to move tasks amongst the worker processors at run-time.

The hwFarm skeleton:

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

3.1 Definition of hwFarm skeleton

The term task farming is used to describe parallel applications that have specific properties. Ordered and structured collections of data items, known as tasks, are each processed by the same operation. Processing the task can be performed in parallel because the tasks are independent [19]. In general, the static scheduling

of tasks to a similar number of processes gives poor load balancing. A task farm solves this by implementing dynamic scheduling to ensure a better balance. The farmer acts as 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 move its tasks amongst workers.

The implementation is divided into three steps:

1) *Implementing skeleton with workers without mobility*: This is a simple skeleton which contains a farmer responsible for distributing the tasks to the workers executing these tasks, as shown in "Fig. 1".

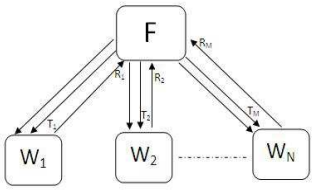

Figure 1. Standard skeleton

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".

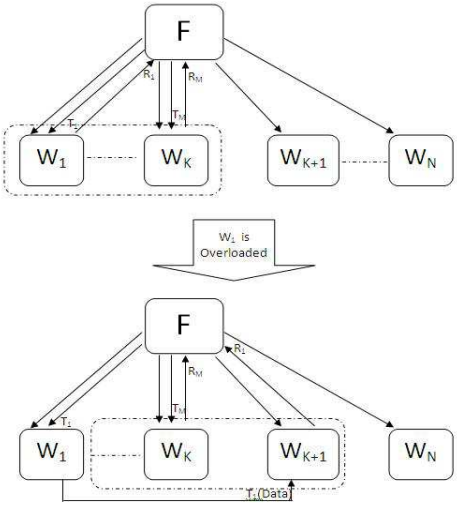

Figure 2. Skeleton with task mobility (data only)

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 beginning of the data but from the beginning of the unprocessed data. A fuller explanation of the implementation will be given in the next section. See "Fig. 3".

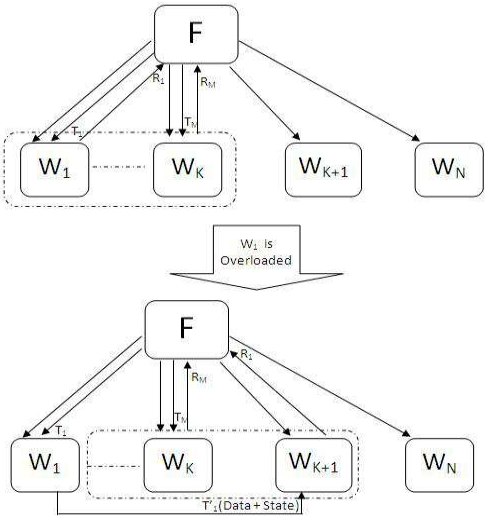

Figure 3. Skeleton with task mobility (data and state)

3.2 How to move state?

The mobility process needs to save the state of execution to continue working from the stop state. As noted, the skeleton hides the parallel and communication details from the programmer so that the programmer is not responsible for the synchronisation between application parts. The programmer has to write their own general function that should be executed by the skeleton. This function has three arguments: the input data to be processed, the output data which is the result, and parameters for the user function. An array of parameters contains the variables that the function needs to process the data. The state of execution depends on the values of these parameters. During mobility, the skeleton moves the input data that was not yet processed by the first worker, the sub-the result of the processed data and the array of parameters for the function at stop point. The new worker receives this data and continues processing from the stop point.

3.3 Movement decision:

One of the biggest issues in parallel and distributed systems is developing techniques for distributing processes to multiple locations [24, 23], to minimize the execution time and increase performance. Our skeleton balances load by using information collected from machines at run time to move a task from heavily loaded processors to lightly loaded processors.

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

A movement decision by a skeleton depends on several polices:

- Information policy: Determines the load information to make a task placement or task mobility. This information is collected from the processors at runtime to know their load changes.
- Selection policy: Decides what task should be moved. The movement decision is taken by collaboration between the master and workers. A 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.
- Placement policy: Identifies where a task should be transferred. The worker, after deciding that it is unable to process its task, or becomes heavily-loaded, determines the best free worker available to process the task.

3.4 Activities of hwFarm skeleton:

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

- The load of all workers in a system is acquired;
- The best workers are chosen where the number of workers is static. The system load in a worker, also known as its 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 is most easily available from a host operating system 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 for the last one-minute period;
- Each worker sends load information to the master, and then the master sends the load information to all workers;
- The tasks are distributed to the chosen workers;
- 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 next task to;
- Depending on the 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.
- The task may then be moved from one worker to a new worker, and the destination worker continues executing the task from the stop point.

4 Experiments

In these experiments, we evaluate the performance and the behaviour of the hwFarm skeleton on heterogeneous distributed memory architecture. We use a ray tracer program that generates the image for 100 rays for 150000 objects in the scene. A 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].

4.1 Platform:

The hwFarm skeleton is tested with a Beowulf cluster located at Heriot-Watt University. The 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).

4.2 Evaluation:

The skeleton is tested in 4 modes:

- *Static task allocation*: The skeleton places 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 [processors for placing the tasks 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 tasks but without mobility. In this case, additional loads will be applied in different periods to 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 tasks from heavily loaded workers to lightly loaded workers where additional loads are applied to workers; we will refer to this mode as *Mobility*. See "Table 4".

The results of these experiments presented in the following tables:

Table 1. STATIC TASK ALLOCATION TIME(SEC)

Tasks / Workers	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

Table 1 shows that our skeleton gives good speed with the ray tracer program and static task allocation.

Table 2. DYNAMIC TASK ALLOCATION TIME(SEC)

Tasks / Workers	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

Table 2 shows show that our skeleton retains good speed with some modest difference in result from collecting and computing load information.

Table 3. DYNAMIC TASK ALLOCATION WITH LOAD AND NO MOBILITY TIME(SEC)

Tasks / Workers	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

Table 3 shows that the skeleton may retain speed when there is additional external load but worker performance deteriorates and execution time became slower.

Table 4. DYNAMIC TASK ALLOCATION WITH LOAD AND MOBILITY TIME(SEC)

Tasks / Workers	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

Table 4 shows that the performance is improved with task mobility when local loads change. However, the performance is still worse than for the *static* and *dynamic* modes.

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

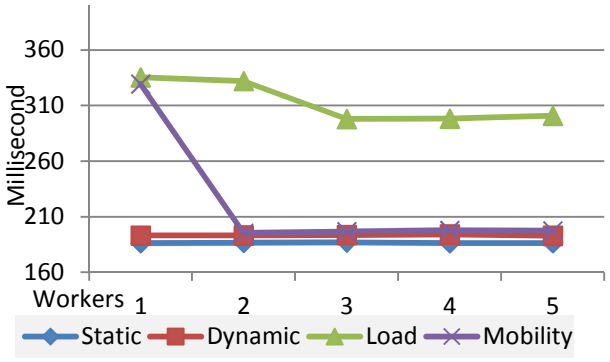

Figure 4. 1-5 Workers, 1 Tasks

Figure 4 shows the time for executing one task on 1 - 5 workers. The execution time in the *static* and *dynamic* modes is approximately the same; the difference comes from the cost of computing and collecting load. The execution time in the *load* mode depends on the load applied to the workers. In the *mobility* mode, the task is moved to a free worker when the current worker becomes unable to effectively process the task, so the execution time will be smaller. The task will not be moved when there are no free workers.

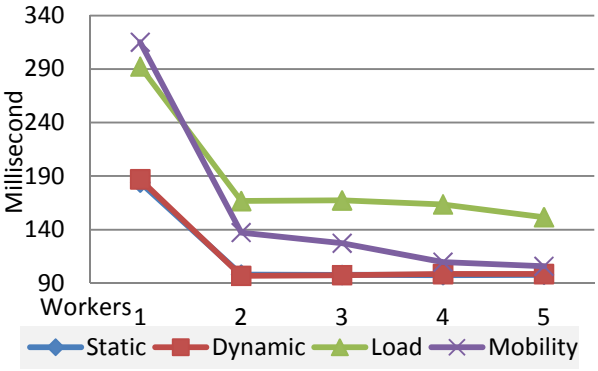

Figure 5. 1-5 Workers, 2 Tasks

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

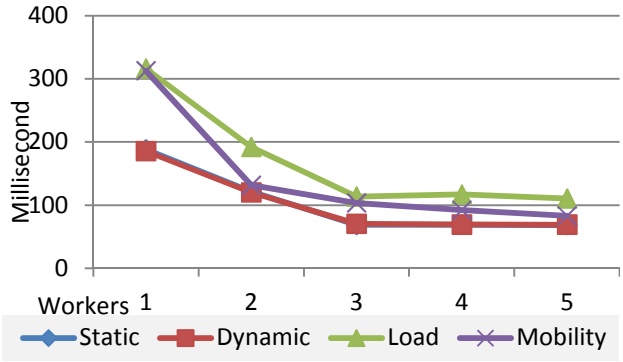

Figure 6. 1-5 Workers, 3 Task

Figure 6 shows the time for executing three tasks on 1 - 5 workers. The improvement of execution time in the *mobility* mode is related to the availability of free workers.

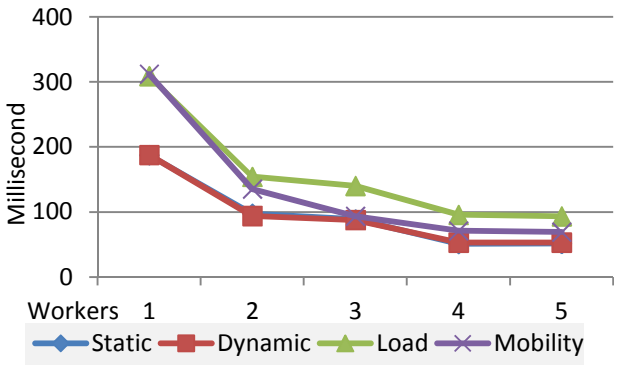

Figure 7. 1-5 Workers, 4 Tasks

Figure 7 shows the time for executing four tasks on 1 - 5 workers. The time in the *mobility* mode approaches the time in the *static* and *dynamic* modes.

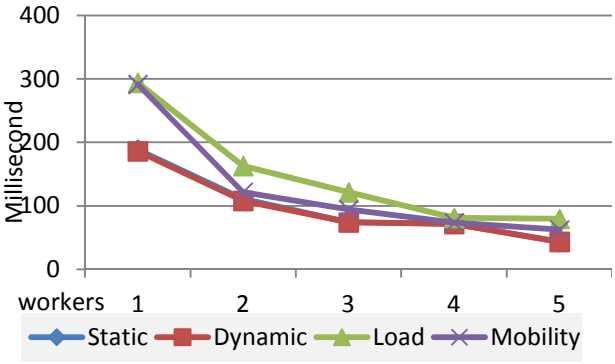

Figure 8. 1-5 Workers, 5 Tasks

Figure 8 shows the time for executing five tasks on 1 - 5 workers. The executing time in *mobility* mode is better than the executing time in *load* mode but is still worse than the *static* and *dynamic* modes.

5 Conclution and future work

We have 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 and able to move tasks from a heavily- loaded to a lightly-loaded worker. Our experiments show that, for the ray tracer program with small numbers of processors, the hwFarm skeleton is able to mitigate the performance effects of external load on individual processors by dynamically moving tasks across processors

We next intend to conduct considerably larger scale experiments, on much larger numbers of processors, with a variety of applications, systematically exploring mobile task behaviour in the presence of different patterns of external load. To aid this, we propose to construct a "load skeleton" which runs alongside an hwFarm application program to apply additional loads in predictable ways.

In future work, the hwFarm skeleton will be extended to be able to move code, as well as data and state, amongst processing units. In addition, we will define a richer cost model for the skeleton to take account of heterogeneity in the processing environment.

6 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 their program in a sequential manner in C . They should specify the input data and identify the high-ordered function which represents our skeleton to run the program with all data in an implicit, parallel manner. Our implementation uses the 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 which must have six parameters: the input data and its length, the output data and its length, and an array of parameters used in the function and its length, 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 as a parameterised function above.
- Initialise the MPI library.
- Initialise the input data.
- Call the hwFarm skeleton.
- Finalise the MPI library.

The prototype of the main function of hwFarm skeleton is :

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

The prototype of the general function that the user writes to be called from the hwFarm skeleton is:

```
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 their task by calling the *doProcessing* function on their data chunk. All variables the function need should be parameterised so we can save the execution state of the function.