OPTIMISING DATA-PARALLEL PERFORMANCE WITH A COST MODEL IN THE PRESENCE OF EXTERNAL LOAD

Turkey Alsalkini
Department of Computer Science, Heriot-Watt University, Edinburgh, UK
ta160@hw.ac.uk

Greg Michaelson
Department of Computer Science, Heriot-Watt University, Edinburgh, UK
G.Michaelson@hw.ac.uk

ABSTRACT

Computational environments, such as clusters and grids, provide a cost-effective platform for running computationally-intensive and data-intensive parallel applications. When such computational environments are shared, the demand for resources is irregular and so the load is unpredictable.

We have been exploring a task mobile skeleton guided by a dynamic cost model, which encapsulates self-aware mobile control for the Master/Worker pattern of data-parallel computations and is able to move running tasks amongst the available processors.

In this paper, we propose a dynamic scheduler guided by a performance cost model. This model enables a skeleton to anticipate future resource needs, be sensitive to the run-time loads and decide whether it would be better to serve the tasks elsewhere. Our experiments show that the skeleton is effective in dynamically relocating tasks in the presence of varying external loads to decrease the overall processing time on shared multi-core environment.

KEYWORDS

Cost Model; Mobility; Skeleton; Load Balancing

1. BACKGROUND AND RELATED WORK

Mobility, which is sometimes termed migration or rescheduling, refers to relocating the computations during run-time amongst the processing elements for distributing the load and giving better use of shared resources (Cabri 2000). Some well-known examples of distributed operating systems that manage the load using task migration are Mach (Baron1985) and MOISX (Barak 1998). Also, Charm++ (Kale 1993), a parallel programming language implementation, supports task migration in distributed memory environments. Other work can be found in (Milojicic 1999).

Skeletal programming, an approach introduced by Cole (Cole 1989), is used to overcome the problems of coordination in parallel programming by exploiting generic program structures. Algorithmic skeletons are high-level parallel programming constructs that embed parallel coordination over sets of locations. Much work has been carried out on skeletal programming for different data types for various parallel architectures. For example, eSkel (Benoit 2005;Benoit 2005B) and Muesli (Kuchen 2002) are libraries that offer data parallel and task parallel skeletons in distributed environments. A widely used parallel programming model for distributed memory architectures is MapReduce developed by Google (Dean 2008). Furthermore, some libraries support shared memory architectures, such as Skandium (Leyton 2010) and TBB (Reinders 2007).

Performance cost models are used to estimate the resource consumption of a program such as execution time or memory consumption (Deng2007; Merlin 2005). Foundational work includes that of Cohen and Zuckerman, who consider cost analysis of Algol-60 programs (Cohen1974), Wegbreit (Wegbreit1975; Wegbreit1976), and Ramshaw (Ramshaw 1979). Some of these models statically determine task placement (Armih 2011) while others dynamically decide optimal task location (Deng 2007). Several cost models have been developed for algorithmic skeletons and related parallel models on shared and distributed memory environments. The BSP (Bulk Synchronous Parallel) cost model is associated with the BSP model to estimate the cost of a superstep and the cost of the program. A cost calculus for the Bird-Meertens Formalism (BMF)

has been developed by Skillicorn and Cai (Skillicorn 1992). The HOPP (High-Order Parallel Programming) model is also based on BMF (Rangaswami 1996). A recent survey is available in (Trinder 2013). Recently, Deng (Deng 2010) developed a novel Autonomous Mobile Program (AMP) that uses a decentralised load balancing technique to choose a location to run on. Depending on future resource needs, AMPs decides either to continue executing locally or to move to a better location.

2. COST MODEL DESIGN AND IMPLEMENTATION

2.1. Task Mobile Skeleton

We propose a dynamic cost model that can be used within our skeleton (Alsalkini 2012). This skeleton is supported with a mobility mechanism to reallocate its running tasks. In this work, we are optimising the mobility decisions through a dynamic scheduler supported with a performance cost model. This skeleton is enhanced with multiple agents distributed across the available locations. These agents cooperate with each other to exchange the most recent dynamic load information among the workers, estimate the cost of executing the tasks, and reallocate tasks to better resources. In this paper, we discuss how the cost model uses the load information from shared resources to take mobility decision. Details of task mobility in our mobile skeleton can be found in (Alsalkini 2012).

To effectively locate tasks at runtime, a mobile skeleton has to be aware of the load changes in the environment. Consequently, each worker has *load* and *estimator* agents for the cost model in addition to *mobile* agents to perform the mobility. The Load Agents, LA, are responsible for collecting the load information from the current location and send them on request. The Estimator Agents, EA, check periodically the progress of the current tasks based on cost model and provide a movement report for moving selected tasks to the chosen workers. The Mobility Agents, MA, are responsible for moving the tasks between the source worker and the destination worker. See Figure 1. Note that the load management agents run asynchronously in order to equalise the system load.

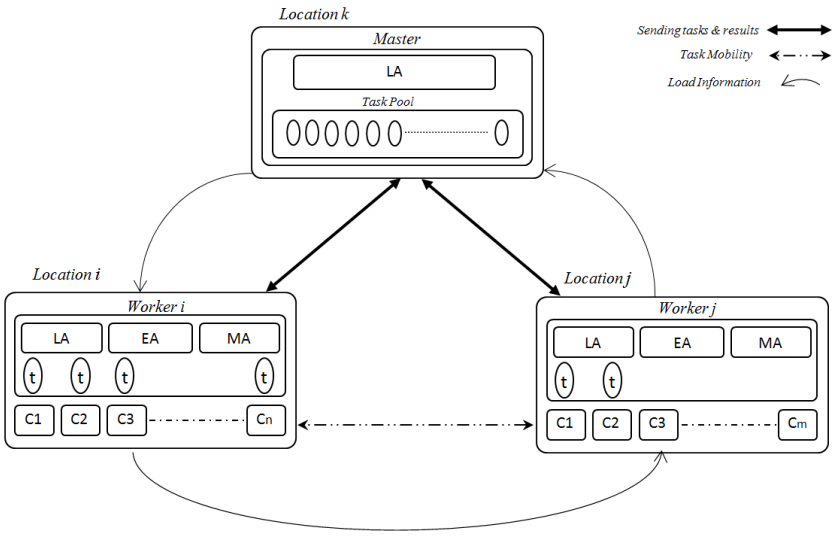

Figure 1: Mobile skeleton design.

2.2. Cost Model Design

For measuring location resource usage, the cost model needs some metrics as parameters. Some metrics are static such as the number of cores and CPU frequency, while others are dynamic and can be obtained from the `/proc` virtual file system. The dynamic metrics used in the cost model are CPU utilisation, the load average and the number of running processors. The static information will be collected once at the beginning while the dynamic metrics will be obtained periodically at runtime.

The cost model estimates the continuation time of the running tasks based on the static and dynamic load information collected from local and the remote locations, as well as from progress information acquired from the running tasks. To gain accurate estimations, the dynamic information needs to be continually updated to reflect the current utilisation of the location. If the utilisation is less than 100%, then the cores are not fully assigned and there is plenty of time for processing other jobs.

Once the CPU utilisation becomes 100% or greater, the processes on this location will compete to use the cores and the scheduler will try to distribute fairly the core time slots to the processes. In this case, the processes will receive less of the computing power than they need. Experimentation shows that CPU utilisation is not sufficient to characterise accurately the processing power that the process may get, so we use two further metrics: the load average and the number of running processes. Depending on these two metrics, we can calculate the relative processing power for a process when the CPU utilisation is 100%.

The cost model uses instrumentations from past iterations to predict future needs. Hence, it is capable of modelling the cost of the current iteration and the cost of the reminder of the program. The cost model is used by the estimator agent to inform the decision of movement. To improve the decision accuracy, the model has to be instantiated with the load information at the current location and other locations.

Our cost model is based on the generic model developed by Deng et al (Deng 2007) for their Autonomous Mobility Skeleton (AMS).

$$P^i = S^i C^i \quad (1)$$

$$R^i = \frac{P^i}{n^i} \quad (2)$$

$$T^h = \frac{W^l R^e T^e}{W^d R^h} \quad (3)$$

$$T^n = \frac{W^l R^e T^e}{W^d R^n} \quad (4)$$

$$T^h > T^{mobility} + T^n \quad (5)$$

S^i: The CPU speed at location *i*
C^i: Number of cores at location *i*
P^i: The total processing power at location *i*
n^i: The number of running processes at location *i*
R^i: The relative processing power at location *i*
R^e: The relative processing power at the current location for the elapsed time
R^h: The relative processing power here
R^n: The relative processing power at the next location
T^h: The estimated time to finish the task here
T^e: The elapsed time here
T^n: The estimated time to finish the task at next location
$T^{mobility}$: The time spent in moving the task from a location to another
W^l: The work left
W^d: The work done

Figure 2: Mobile skeleton Cost Model.

Figure 2 shows the cost model for the mobile skeleton. This model will be used by the skeleton scheduler to estimate the behaviour of the running tasks. Then, the scheduler might take decision to reschedule some tasks to faster locations. (1) gives the total processing power at location *i*. The processing power depends on the number of cores available at that location and the speed of its cores. (2) shows the relative power that a process can have at that location. (3) shows the estimated continuation time for the task here. The relative processing power R^e for the previous time can be taken by recording the load status during the task lifetime at the current location. (4) gives the estimated continuation time for a task at the remote location. (5) shows the condition under which the tasks will move if the time to complete the task in the current location is more than the time to complete in the remote location.

All cost model estimations and decisions are made under the assumption that the load on the system will not change dramatically immediately after a task moves. Thus, prediction of future system load is not addressed in this work.

3. EVALUATION

We have implemented our skeleton using the C programming language. We use MPI for a distributed memory environment (Snir 1998) and the POSIX library for shared memory architectures (Bradford 1996). In this implementation, we also use some features of the Linux kernel (2.6 or later).

We have explored two types of computations: regular and irregular. For regular problems, we use a simple Matrix Multiplication application. In contrast, we are testing a simple Raytracer as an example of irregular computations.

The skeleton with its cost model was tested in 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(2.6.32) at 2.00GHz with 4096 kb L2 cache and using 12GB RAM).

3.1. Cost Model Validation

3.1.1. Regular Computations

Regular computations have iterations where each consumes the same amount of processing time under the same load. We use a simple Matrix Multiplication as a regular application to validate the estimations of our cost model:

```
for(i=0;i<n;i++)                    //n:row count in M1
  for(j=0;j<m;j++)                  //m:col count in M2
    for(k=0;k<c;k++)                //c:col count in M1 = row count in M2
      M3[i][j]=mul(M1[i][k],M2[k][j);
```

Table 1 refers to the results of running the problem with different sizes while Table 2 shows the times of each task, both when running a 4000*4000 Matrix Multiplication problem. These results show the accuracy of the estimated time with a maximum error 2.67%.

Table 1. The estimated and actual times of Matrix Multiplication problem

Size	Estimated Time (Sec)	Actual Time (Sec)	Av-Error	Perf (%)	St-Dev
1000*1000	3.095	3.095	0.006	0.194	0.00042
1200*1200	5.349	5.341	0.009	0.169	0.01118
1400*1400	8.472	8.464	0.010	0.118	0.01040
1800*1800	17.972	17.936	0.036	0.201	0.03707
2000*2000	24.629	24.598	0.032	0.130	0.05363

Table 2. The estimated and actual times of each task in 4000*4000 Matrix Multiplication

Task	Estimated Time (sec)	Actual Time (Sec)	Ave-Error	Perf (%)	St-Dev
1	24.931	24.531	0.401	1.635	0.42224
2	24.935	24.537	0.403	1.642	0.43074
3	24.939	24.528	0.414	1.688	0.44459
4	24.944	24.526	0.422	1.721	0.45783
5	26.968	27.115	0.613	2.261	0.49469
6	26.933	27.128	0.725	2.673	0.52169
7	26.694	26.931	0.693	2.573	0.48356
8	30.248	30.480	0.529	1.736	0.27621

Table 3. The sampled estimation times during running of task 1 of 2000*2000 Matrix Multiplication

Sample point (Sec)	Est-Time (Sec)	Act-Time (Sec)	Diff	Diff %
4	24.644	24.599	0.045	0.183
8	24.629	24.599	0.030	0.122
12	24.625	24.599	0.026	0.106
16	24.603	24.599	0.004	0.016
20	24.606	24.599	0.007	0.028
24	24.608	24.599	0.009	0.037

We also measure the estimations while the tasks are running to check the accuracy of the model results. Table 3 shows sampled estimated times of task 1 of 2000*2000 Matrix Multiplication problem composed of one task. Moreover, Table 4 shows sampled estimated times of task 8 of 4000*4000 Matrix Multiplication problem composed of eight tasks. These results show the accuracy of the estimations produced by the cost model in comparable with the actual execution times.

Table 4. The sampled estimation times during running of task 8 of 4000*4000 Matrix Multiplication

Sample point (Sec)	Est-Time (Sec)	Act-Time (Sec)	Diff	Diff %
6	30.703	29.667	1.036	3.492
12	29.480	29.667	0.187	0.630
18	29.799	29.667	0.132	0.445
24	29.993	29.667	0.326	1.099
30	29.792	29.667	0.125	0.421

3.3.2. Irregular Computations

In irregular computations, each iteration may need a different amount of processing time depending on the data. Here we use a simple Raytracer application where each ray has to process different numbers of objects in a scene:

```
rays=generateRays(rays_count,coordinates);
scene=loadObjects();
foreach ray in rays
    imp=firstImpact(ray,scene);
    imps=addImpact(imp);
showImpacts(imps,rays_count);
```

Table 5 shows the estimation times when running the Raytracer problem with different sizes, where each instance is composed of one task. Table 6 refers to the times of executing Raytracer with 100 rays where the problem is divided into 8 tasks. Tables 7 and 8 show the sampled estimated times in comparable with the actual times.

Table 5. The estimated and actual times of Raytracer problem

Size (Rays)	Estimated Time (Sec)	Actual Time (Sec)	Av-Error	Perf (%)	St-Dev
20	6.836	6.814	0.059	0.873	0.03079
30	15.188	15.329	0.400	2.608	0.35025
40	27.216	27.585	0.830	3.008	0.68579
50	42.844	43.431	1.285	2.959	1.01699

Table 6. The estimated and actual times of each task in Raytracer with 100 rays

Task	Estimated Time (sec)	Actual Time (Sec)	Ave-Error	Perf (%)	St-Dev
1	20.418	20.717	0.328	1.585	0.34545
2	20.119	20.328	0.235	1.157	0.16234
3	26.109	26.172	0.789	3.015	1.04277
4	39.525	32.863	6.662	20.271	4.64025
5	38.298	32.471	5.827	17.945	4.07632
6	32.070	27.700	4.370	15.777	3.29234
7	22.355	21.277	1.078	5.069	1.24830
8	20.339	20.204	0.203	1.003	0.25247

Table 7. The sampled estimation times during running of task 1 of Raytracer with 40 rays

Sample point (Sec)	Est-Time (Sec)	Act-Time (Sec)	Diff	Diff %
6	25.796	27.585	1.789	6.485
12	27.230	27.585	0.355	1.287
18	28.317	27.585	0.732	2.654
24	27.927	27.585	0.342	1.240

Table 8. The sampled estimation times during running of task 8 of task 1 of Raytracer with 100 rays

Sample point (Sec)	Est-Time (Sec)	Act-Time (Sec)	Diff	Diff %
5	19.870	20.328	0.458	2.253
10	20.421	20.328	0.093	0.457
15	20.181	20.328	0.147	0.723
20	19.965	20.328	0.363	1.786
25	19.944	20.328	0.384	1.889
30	20.174	20.328	0.154	0.758
35	20.281	20.328	0.047	0.231

The previous results illustrate that the estimation is not as accurate as in the regular example, with the error reaching 20% from the actual time. Nonetheless, the decisions made by the cost model reduced the overall execution time because the continuation cost is affected on the highly loaded workers. Therefore, the rescheduling using mobility will help to execute the task faster on the lightly loaded workers.

3.2. Experimental Evaluation

We next evaluate the behaviour and the performance of our skeleton with its cost model on a shared/distributed memory architecture. For evaluation purposes, we need to explore large experiments on a larger number of processors in the presence of different patterns of external loads. To generate such a load, we use our load function (Alsalkini 2014) to dynamically and systematically apply a load of a known pattern alongside the mobile skeleton. The load function has minimal overhead and makes no appreciable difference to experimental times. We again explore Matrix Multiplication and Raytracer as examples of regular and irregular computations.

3.2.1. Mobility Behaviour

To investigate if mobility behaves as we expect, we ran a Matrix Multiplication problem with 8 tasks on 3 locations. Figure 3.a shows the changes on the load over these locations. It can be seen in Figure 3.b that the tasks are moving with behaviour inverse to the load to achieve balance. For the Raytracer problem, see Figure 4. In both problems, it can be observed that the skeleton responds quickly to load changes.

(a) Load pattern (b) Mobility behaviour

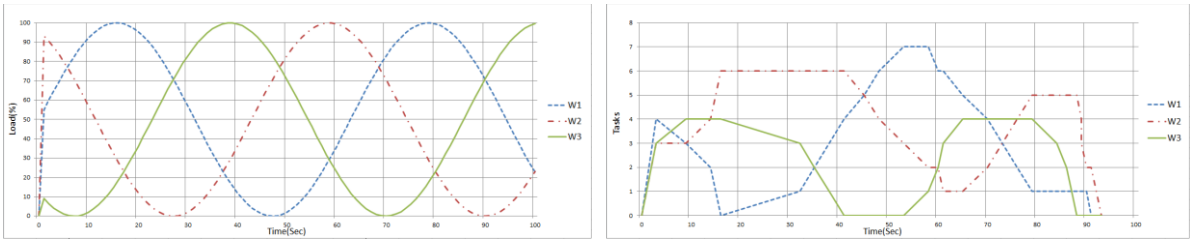

Figure 3: Mobility behaviour of 10 tasks on 3 workers (Matrix Multiplication)

(a) Load pattern (b) Mobility behaviour

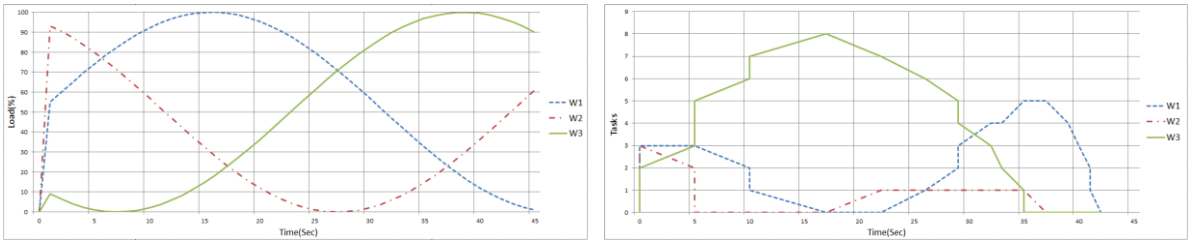

Figure 4: Mobility behaviour of 8 tasks on 3 workers (Raytracer)

3.2.2. Mobility Performance

Finally, we explore the improvement in performance for a wide range of problem sizes with different numbers of tasks. Table 9 shows the results of Matrix Multiplication. We can see the improvement in the total execution time after applying our cost model and how the model compensates for unpredictable load variations. In the table, *Orig*: is the original execution time without any load applied; *L-on*: is the execution time with a load pattern applied during the run-time; *M-on*: is the execution time and activated mobility. *Diff*: is the difference between two modes. *Abs-impr*: is the absolute improvement we get after applying mobility to the delay time due to the load. *Rel-impr*: is the relative improvement we get after applying mobility to the execution time in the presence of load. The best absolute improvement is 57% and the worst is 12%. In contrast, the best relative improvement is 19% and the worst is 3%.

For the Raytracer problem, see Table 10. Note that for this irregular problem, load compensation is weaker because the estimation is less accurate. Note also that while the estimation is less accurate, load compensation is comparable to that for the regular problem.

Table 9: Improvement in performance in the presence of external load (Matrix)

Matrix size	Tasks/ Workers	Orig (s)	L-on & M-off (s)	L-on & M-on (s)	Diff(L-on & M-off (s))	Diff(L-on & M-on (s))	Abs-impr (%)	Rel-impr (%)
6000*6000	12/3	60.98	86.83	81.69	25.85	20.71	19.91	5.93
7200*7200	12/3	102.52	143.58	138.12	41.06	35.6	13.31	3.81
4800*4800	12/3	31.08	47.28	42.15	16.2	11.08	31.66	10.85
5600*5600	14/3	42.74	61.46	57.83	18.72	15.09	19.41	5.91
7700*7700	14/3	108.9	165.09	149.35	56.19	40.45	28.01	9.54
6000*6000	10/3	73.81	104.08	94.42	30.27	20.60	31.92	9.28
3600*3600	6/3	26.65	40.35	32.47	13.7	5.8	57.52	19.53

Table 10: Improvement in performance in the presence of external load (Raytracer)

Raytracer (rays)	Tasks/ Workers	Orig (s)	L-on & M-off (s)	L-on & M-on(s)	Diff(L-on & M-off (s))	Diff(L-on & M-on (s))	Abs-impr (%)	Rel-impr (%)
90	6/3	24.66	35.31	32.10	10.65	7.44	30.10	9.08
100	5/3	36.82	49.06	41.82	12.24	5.01	59.09	14.74
120	8/3	32.62	47.51	42.22	14.89	9.60	35.55	11.14
140	8/3	49.21	66.99	63.21	17.78	14.00	21.26	5.64
150	9/3	44.55	58.59	54.65	14.05	10.10	28.08	6.73
150	10/3	40.38	55.75	48.47	15.37	8.09	47.34	13.05
150	15/3	28.01	39.11	35.58	11.10	7.56	31.84	9.04
200	8/4	94.25	121.37	107.35	27.12	13.11	51.68	11.55
300	16/4	105.17	143.87	134.62	38.70	29.45	23.91	6.43

4. CONCLUSION

We have presented a dynamic cost model to manage data-parallel skeleton tasks in the presence of external loads. The cost model can be used to estimate the continuation completion time for running tasks at the current location and remote locations. Depending on conditions related to the cost of moving and network delay, tasks can be transferred to faster locations for better performance. The estimation process can be triggered periodically or when the worker becomes heavily loaded. Here we address only the loaded worker initiated method.

Our experiments show that the cost model gives accurate decisions in a regular computation whilst the decision is less accurate in an irregular computation. However, in the latter case, the decision made by the model helps in balancing the load and improving the performance.

We have not considered network characteristics which can also have an impact on mobility decisions. We next intend to conduct larger scale experiments on remote clusters to investigate the effect of network delay

on mobility. We also want to extend our work to heterogeneous architectures including GPUs and co-processors.

REFERENCES

Alsalkini T. and Michaelson G., 2012. Dynamic Farm Skeleton Task Allocation through Task Mobility. *In: 18th International Conference on Parallel and Distributed Processing Techniques and Applications*. Las Vegas, USA, pp. 232-238.

Alsalkini T. and Michaelson G., 2014. Generating Artificial Load Patterns on Multi-Processor Platforms. *In: 11th International Conference Applied Computing*. Porto, Portugal, pp. 77-84.

Armih K. et al, 2011. Cache Size in a Cost Model for Heterogeneous Skeletons. *In Proc. Fifth int. workshop on High-level parallel programming and applications (HLPP '11*). New York, NY, USA, pp 3-10.

Barak A. and La'adan O., 1998. The MOSIX Multicomputer Operating System for High-Performance Cluster Computing, *Future Generation Computer Systems.* Vol. 13, No. 4-5, pp 361-372.

Baron R. et al, 1985. Mach-1: An Operating Environment for Large-Scale Multiprocessor Applications. *IEEE Software*. Vol. 2, No. 4, pp 65-67.

Benoit A. and Cole M., 2005. Two Fundamental Concepts in Skeletal Parallel Programming. *In The International Conference on Computational Science (ICCS 2005)*, Springer-Verlag, pp. 764-771.

Benoit A. et al, 2005. Flexible Skeletal Programming with eSkel. *In Proceedings of the 11th international Euro-Par conference on Parallel Processing (Euro-Par'05)*, Springer-Verlag. Berlin, Heidelberg. pp. 761-770.

Bradford N. et al, 1996. *Pthreads Programming*. O'Reilly Associates, Inc., Sebastopol, CA, USA.

Cabri G. et al, 2000. 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).

Cohen J. and Zuckerman C., 1974. Two Languages for Estimating Program Efficiency. *Commun. ACM*. Vol. 17, No. 6, pp 301-308.

Cole M., 1989. *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press, Cambridge MA.

Dean J. and Ghemawat S., 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*. Vol. 51, No. 1, pp 107-113.

Deng X. Y., 2007. *Cost-Driven Autonomous Mobility*, Ph.D. thesis, Heriot-Watt University, United Kingdom, May.

Deng X. Y. et al, 2010. Cost-driven Autonomous Mobility, *Computer Languages, Systems and Structures*. Vol. 36, No. 1, pp 34-59, April.

Kale L. V. and Krishnan S., 1993. Charm++: A Portable Concurrent Object Oriented System Based on C++. *SIGPLAN Not.* Vol. 28, No. 10, pp. 91-108.

Kuchen H., 2002. A Skeleton Library. *In Proceedings of the 8th International Euro-Par Conference on Parallel Processing (Euro-Par '02)*, Springer-Verlag. London, UK, pp 620-629.

Leyton M. and Piquer J. M., 2010. Skandium: Multi-core Programming with Algorithmic Skeletons. *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pp 289-296.

Merlin A. and Hains G., 2005. A Generic Cost Model for Concurrent and Data-Parallel Meta-Computing. *Electronic Notes in Theoretical Computer Science*. Vol. 128, No. 6, pp 3-19.

Milojicic D. et al, 1999. *Mobility: Processes, Computers, and Agents*. Addison-Wesley, Reading, MA, USA.

Rangaswami R., 1996. *A Cost Analysis for a Higher-Order Parallel Programming Model*, Ph.D. thesis. Department of Computer Science, Edinburgh University.

Ramshaw L. H., 1979. *Formalization the Analysis of Algorithms*, Ph.D. thesis. Stanford University, Department of Computer Science.

Reinders J., 2007. *Intel Threading Building Blocks (First ed.)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA.

Skillicorn B. D., 1992. *Parallelism and the Bird-Meertens Formalism*, Department of Computing and Information Science, Queen's University, Kingston, Ontario.

Snir M. et al, 1998. *MPI-The Complete Reference. Volume 1: The MPI Core*, MIT Press, Cambridge, MA, USA.

Trinder P. W. et al, 2013, Resource analyses for parallel and distributed coordination. *Concurrency and Computation: Practice and Experience*. Vol. 25, No. 3, pp 309-348.

Wegbreit B., 1975. Mechanical Program Analysis, *Commun. ACM*. Vol. 18, No. 9, pp 528-539.

Wegbreit B., 1976. Verifying Program Performance, *Journal of ACM*. Vol. 23, No. 4, pp 691-699.