Cache Size in a Cost Model for Heterogeneous Skeletons

K.A. Armih
Heriot-Watt University, Edinburgh, Scotland, UK
kaa41@hw.ac.uk

G.J. Michaelson
Heriot-Watt University, Edinburgh, Scotland, UK
G.Michaelson@hw.ac.uk

P.W. Trinder
Heriot-Watt University, Edinburgh, Scotland, UK
P.W.Trinder@hw.ac.uk

Abstract

High performance architectures are increasingly heterogeneous with shared and distributed memory components. Programming such architectures is complicated and performance portability is a major issue as the architectures evolve.

This paper proposes a new architectural cost model that accounts for cache size and improves on heterogeneous architectures, and demonstrates a skeleton-based programming model that simplifies programming heterogeneous architectures. We further demonstrate that the cost model can be exploited by skeletons to improve load balancing on heterogeneous architectures. The heterogeneous skeleton model facilitates performance portability, using the architectural cost model to automatically balance load across heterogeneous components of the architecture. For both a data parallel benchmark, and realistic image processing program we obtain good performance for the heterogeneous skeleton on homogeneous shared and distributed memory architectures, and on three heterogeneous architectures. We also show that taking cache size into account in the model leads to improved balance and performance.

Keywords Parallel, Skeleton, Heterogeneous, Cost model

1. Introduction

With the advent of multicores, many high performance architectures are heterogeneous, that is they comprise clusters of multicore nodes. Load balancing is a major issue of such architectures where the optimisation of overall processing time depends on how to balance load across the nodes of these architectures. Hybrid parallel programming models like [19, 25, 29, 31] are used to exploit such architectures. These models are relatively complex as they typically combine a distributed-memory message-passing model like MPI [32] and a shared-memory model like OpenMP [9]. Moreover, performance portability becomes a major issue as programs must be rewritten as the shared and distributed memory characteristics of the heterogeneous architectures evolve, e.g. as the number of shared-memory cores skyrockets.

This paper proposes a new architectural cost model for load balance on heterogeneous architectures and demonstrates a skeleton-based programming model for programming heterogeneous architectures. The paper makes the following research contributions

- We present a new architectural cost model for heterogeneous architectures that characterise components of the architecture by number of cores, clock speed, and crucially the size of the L2 cache (Section 3).
- The heterogeneous skeleton model facilitates performance portability. The skeletons use the new cost model to automatically balance load across heterogeneous components of the architecture.
- We present the implementation of a data parallel heterogeneous skeletons, that are novel in supporting execution on heterogeneous architectures. The heterogeneous skeleton model simplifies parallel programming on heterogeneous architectures. The programmer has a single model, that of composing and parameterising skeletons rather than two models (Section 5).
- We demonstrate the performance portability by showing that the heterogeneous skeletons deliver good performance on shared memory, distributed memory, and four heterogeneous architectures (Section 6).

HPPL'11 September 18, 2011, Tokyo.

2. Background

Structured parallel programming (or algorithmic skeletons) was first introduced by Cole in [10]. Simply, algorithm skeletons are high-level parallel programming model that used to ease parallel programs development process by concealing most parallel coordination from the users(programmers). Much work [17, 21, 28, 30] has been done in the area of skeletal programming under various names, and for different parallel architectures. Different research groups have provided skeleton implementations. Some of the skeleton frameworks come as a library and others are provided as language constructs. Since the organisation of parallelism in skeletal programming is up to the skeleton implementation, algorithm skeletons can be classified either as distributed- or shared memory architecture. Many frameworks are provided as libraries that support distributed parallel computations which are implemented on distributed parallel architectures.

eSkel [5, 6, 11] is a library of C with MPI functions offering data parallel and task parallel skeletons. eSkel is used in distributed environments such as clusters and grids. SkeTo [27] is also using MPI to achieve parallel distributed computations on distributed parallel architectures. It is provided as a C++ library. Muesli [20] provides data parallel and task parallel skeletons as a C++ library. It works using MPI to achieve parallelism on distributed parallel architectures. Google's MapReduce [14] is a parallel programming model for distributed-memory environments developed by Google. It is proposed as library that can be implemented in C++. Google's MapReduce is designed to take advantage of large clusters by providing high-level abstractions for parallel algorithms. These abstraction are based on the concept of *map* and *reduce* primitives present in functional languages, where the *map* function processes

the input data and produces a set of intermediate key/value pairs and the *reduce* function merges the intermediate values that have the same key. Apache Hadoop [1] is a Java framework for processing large data set on large clusters. It provides a distributed file system(HDFS) that can store data on nodes in the cluster and implements the MapReduce paradigm.

One of the recent skeleton libraries that supports shared-memory architectures(*Multi-core systems*) is Skandium [3].
Skandium [22] is a Java library of shared memory algorithm skeletons. It was designed to works using the Fork/Join framework in Java to achieve parallelism in shared memory environments. Other frameworks such as TBB [2] provide high level parallel abstractions. TBB is a C++ library which provides various templates for parallel programming such as parallel_for, parallel_reduce, parallel_scan, and parallel_pipeline.

Many cost performance models [16] have been developed for parallel algorithmic skeletons on homogeneous architectures. Darlington [13] developed a cost model for a divide and conquer (DC) skeleton. This cost model calculates the execution time using the communication time between processors, the time needed to split the problem into two subproblem, and the time to solve the problem in one processor. Sreekantaswamy [33] developed a cost model for a farm skeleton. In this model three hardware parameters are used to describe the performance of parallel architectures: 1) communication time 2) computation time 3) the start up time for the skeleton.

Modern CPUs have multi-level memory which significantly affects performance. As discussed below, we think that cache size will be the dominating property. Compared with these cost performance models, we have developed a new cost model to optimise overall processing time for heterogeneous algorithmic skeleton on heterogeneous systems(*multi-core clusters etc*). The cost model uses three hardware properties to balance load across heterogeneous components of the architectures: 1) number of cores 2) CPU speed 3) size of L2 cache.

To experiment our cost model we developed a new skeleton library for heterogeneous systems. Our work is motivated by the development of eSkel in using MPI routines to achieve parallelism on distributed systems. Therefore our work takes advantage of MPI and OpenMP to provide a skeleton library named *HWSkel* for heterogeneous systems. *HWSkel* is a library of C functions running on top of MPI and OpenMP to achieve distributed- and shared parallelism respectively.

3. Cost Model

3.1 Related Cost Models

Several parallel computational models have been developed for parallel distributed systems[4, 12, 15, 34]. A good general survey of early research is given in [26]. In this section we discus the cost models that proposed for heterogeneous clusters and related to our cost model. HiHCoHP model[8] is realistic communication model for hypercluster with heterogeneous processors. It uses several parameters that reflect the heterogeneity of hyperclusters, such as latencies, link-bandwidth, capacities of network and processor,s message-processing time. An extension of logGP[4], HLogGP[7], has been proposed for heterogeneous clusters. It includes a number of parameters that capture the computational and network features namely *latency, overhead, gap between message, gap per byte and computational power*.

In contrast to the frameworks described above, our cost model provides the following features:

Complexity. As the degree of model complexity depends on numbers of parameters that need to be estimated, we are content with a simple model, with small numbers parameters.

Target Architectures. HiHCoHP and HLogGP models are designed for heterogeneous clusters in which both processors and network are heterogeneous. Nevertheless, our cost model targets homogeneous networked cluster where the nodes are heterogeneous. We focus on the optimisation of processing time that can be affected by the computational features of the nodes as discussed in section 3.

Skeleton-*based* Approach. The idea of associating cost models with algorithmic skeletons is not new. However, we integrate an architural cost model that accounts for cache size for load balance on heterogeneous clusters into a skeleton library for parallel implementations. Moreover, the cost model is used implicitly to guide the development and implementation of parallel programs.

3.2 Methodology

We wish to develop a cost model to optimise overall processing time for our skeleton on a heterogeneous system composed of networks of arbitrary numbers of nodes, each with an arbitrary number of cores sharing arbitrary amounts of memory. From our model, we seek *relative* measures of processing power to guide data distribution rather than *absolute* predictions of processing time. Thus, we are content with a simple model, with small numbers of easily instantiable parameters.

In constructing our model, we assume that:

- inter-node communication time is uniform;
- on an individual node, all the cores have the same processor characteristics;
- each core processes a distinct single chunk of data, without interruption, using the same algorithm as the other cores;

Hence, we focus the optimisation of processing time on distributing appropriately sized chunks of data to cores to balance processing.

For a first cut, we might base this distribution on the number of nodes, and for, each node, the speed of each core. Suppose node i has C_i cores each of speed S_i. Then, the total available processing power for n nodes is:

$$\sum_{i=1}^{i=n} C_i * S_i$$

Each node i might receive:

$$C_i * S_i / \sum_{i=1}^{i=n} C_i * S_i$$

of the data, so each core of node i might receive:

$$S_i / \sum_{i=1}^{i=n} C_i * S_i$$

Now, all processors have some memory hierarchy, from registers, via various levels of cache, to RAM and beyond. We assume that registers and on-core caches are private and operate at CPU speed. Shared cache, typically L2 or L3, is usually many orders of magnitude smaller than shared RAM, and, for many problems, RAM is sufficiently large for paging to be absent. Thus, we identify the size of top level shared cache as the most significant memory factor affecting overall performance.

We have assumed that all cores are running the same algorithm, which implies that they will have similar patterns of access to shared memory. In particular, each core will incur similar sequences of cache faults. Then, the number of cache faults will be determined by the size of the cache: for a larger cache it is more

likely that a required portion of the address space is already resident.

Thus, we refine our model to take into account the size of the cache, which we denote as $L2_i$ on node i, with a larger cache implying that a node should receive larger size data chunks. Then, the strength of a core on node i is given by:

$$C_i * S_i * L2_i$$

The overall power P of the system is given by:

$$P = \sum_{i=1}^{i=n} C_i * S_i * L2_i$$

For data size *D*, the chunk size for node i is:

$$(C_i * S_i * L2_i/P) * D$$

and each core processes:

$$(S_i * L2_i/P) * D$$

For a heterogeneous system, it is necessary to normalise the overall system power in order to predict maximum speedup and determine whether that has been achieved. We think it most principled to do so using *the core with the greatest power*.

So, to predict maximum speedup we:

- find the power of each core P_i and choose the greatest P_L:
- find the maximum predicted speedup by dividing the overall power by the greatest core power: P/P_l.

Then, to assess achieved speedup we:

- initially, measure the program on one core with that greatest power to provide a base line;
- subsequently, measure speedup relative to that base line measurement.

4. An Overview of *HWSkel*

The main design idea of the *HWSkel* library is to provide high-level parallel programming models(skeletons) that capture common parallel patterns, and execute them on heterogeneous systems, in particular on clusters of multi-core. In other words *HWSkel* is designed to abstract all the parallelism and communication involved in a program that will be executed on a multi-core cluster. The skeletons in *HWSkel* are implemented using a hybrid OpenMP/MPI model. This design is adaptable and hence *HWSkel* skeletons can be used for distributed-memory systems, shared-memory systems or both systems together as heterogeneous systems. For instance if the underlying system is distributed memory, the distributed parallel programming model will be automatically adopted. The *HWSkel* library has the following characteristics:

- The recent trend of designing algorithm skeletons is to present them as libraries to avoid adding any new syntax. Therefore, *HWSkel* is provided as a library of C that works using MPI and OpenMP to achieve the parallelisation on heterogeneous systems.
- *HWSkel* supports parallelism on heterogeneous architectures, and flexible parallelism on either shared or distributed memory architectures.
- Lower-level details of parallel programming are concealed from the users by our skeleton. Furthermore, the interaction between MPI and OpenMP introduces new communication such as data flow between theses models. These communication is implicitly defined by skeleton composition. Therefore, our skeleton can be used to develop parallel programs in a sequential fashion.
- To ensure a good load balance we integrate an efficient cost model for data-load distribution into our system. The cost model uses specific hardware properties to distribute work between processors.

4.1 Using The Cost Model in *HWSkel* Library

In the *HWSkel* library, the cost model is integrated in the skeletons to improve its parallel performance on heterogeneous multi-core clusters. In the hybrid programming model, load balance can be more easily achieved in the shared-memory model(*OpenMP*) than the distributed-memory model(*MPI*), hence the load balance is dependent on MPI distribution not on OpenMP[18]. Since the *hMapReduce* and *hMapReduceAll* skeletons are based on a hybrid programming model where we assume that all cores on the node have the same characteristics, we used the cost model only for data-load distribution over the cluster nodes. Since both skeletons use the SPMD model for distributed memory parallelism, therefore the master PE is responsible for applying the cost model. At the beginning of skeleton execution, the master PE collects and registers the hardware information for each PE that is needed by the cost model. This information is collected from the local system file "/proc/cpuInfo" of each PE in the cluster. After collecting the hardware information the master PE applies the cost model to distribute the data over the cluster.

5. Using *HWSkel* Skeletons

The prototype of the *HWSkel* library is implemented in C with MPI and OpenMP. Our framework enables the programmer to develop parallel programs in C in a sequential manner, where the skeleton can be written as a sequential function call in the program. The current specification of *HWSkel* defines a set of skeletons for data parallelisation. In this paper two skeletons are used:

- The *hMapReduce* skeleton that supports data parallel computation on a heterogeneous multi-core cluster, where the programmer must specify the input data and the operations(*worker operation, reduction operation*) that will be performed by the workers.

```
void* hMapReduce(void* dataList,int size,
    enum DataType dType, void* funcName,
     enum DataType rType,enum CombinOP opType);
```

where:

`dataList`	Specifies the starting address of the data.
`size`	Indicates the length of the data.
`dType`	Denotes the datatype of input data.
`funcName`	Specifies map function.
`rType`	Denotes the datatype of output data.
`opType`	Specifies the reduction operation.

- The *hMapReduceAll* skeleton that behaves like *hMapReduce*. The difference is that instead of splitting the data among the workers, all data are sent to all workers.

```
void* hMapReduceAll(void* dataList,int size,
    enum DataType dType, void* funcName,
     enum DataType rType,enum CombinOP opType);
```

5.1 *sum-Euler* Example

As an example program, *sum-Euler* calculates the sum of the totients between a lower and an upper limit, where the totient function of an integer (*n*) gives number of positive integers less than or equal

to (n) that are relatively prime to (n).

$$sumEuler = \sum_{n=lower}^{upper} \emptyset(n) \qquad \emptyset(n) = \sum_{i=1}^{n} euler(n)$$

Figure 1 presents the sequential `sumTotient` function that receives a list of integers.

```
int sumTotient(nt *datalist, int length)
{  int i,j,k;
   int sum = 0;
   for(i=0;i<length;i++)
      sum = sum+euler(datalist[i]);
   return sum;
}
```

Figure 1: Code for `sumTotient` function

```
int euler(int n)
{  int i, length=0;
   for(i=1;i<n;i++)
     if(relprime(n,i))
       length++;
   return length;
}
```

Figure 2: Code for `euler` function

Figure 2 shows the `euler` function that applies the Euler totient function to each element in the list, then the results are summed for all elements. In the parallel version, the list of integers is split into chunks using a split function which employs the cost model of load distribution, and then the `euler` function is mapped in parallel across each chunk. Finally, the results are summed sequentially for all elements in the main function(`sumTotient`).

Figure 3 presents the main *sum-Euler* program that uses the *hMapReduce* skeleton. The parameters of *hMapReduce* skeleton are `sumTotient` as the map function and the plus from Figure 4 as the reduction function.

```
int main(int argc, char **argv)
{ Init_HWSkel(argc,argv);
  result=hMapReduce(data,length,INT
                ,sumTotient,INT,plus);
  Terminate_HWSkel();
}
```

Figure 3: Main program for *sum-Euler*

```
int plus(int *arr,int size)
{ int i, result=0;
  for(i=0;i<size;i++)
      result+=arr[i];
  return result;
}
```

Figure 4: Code for `plus` function

5.2 Image Matching Example

Image Matching is fundamental aspect of many problems in computer vision including object recognition. Matching different images of object requires local image features that are unaffected by nearby clutter or partial occlusion [23]. The Scale Invariant Feature Transform (SIFT) is an approach used to transform image data into scale invariant coordinates relative to local features which has properties that make it suitable for image matching and recognition [24]. Therefore, image matching is performed by first extracting local features from the input image using SIFT algorithm and then these features are individually matched to sift features obtained from training images by using nearest-neighbour algorithm. In addition, to avoid expensive search that required for nearest-neighbour algorithm a modification of k-d tree algorithm called best-bin-first method is used [23].

We port the sequential program of object recognition application that was introduced by David Lowe at University of British Columbia. This application cosist of 26 C code files which approximately contains 9446 lines in aggregate. Basically, the sequetial algorithm of object recognition application is divided into two stages, first stages is SIFT keypoints detecting and secondly SIFT keypoints matching stage. The activity diagram in Figure 5 illustrate the original sequential algorithm of object recognition application.

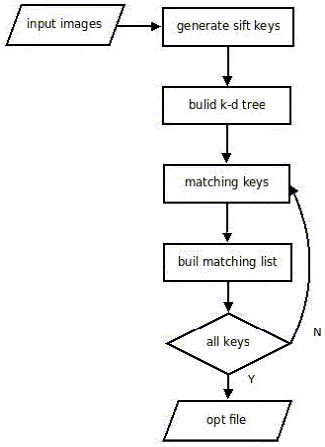


Figure 5: Activity Diagram of Sequential Image Matching Algorithm

In the parallel version, we parallelised the second stage (most time consumptive) of the application using *hMapReduceAll* skeleton. Therefore, the sequential algorithm of the entire application is parallelised by scheduling the sift keypoints to the processing elements and each processing element applies the BBF algorithm [24] in order to performs matching operation.

Figure 6 presents the main *Image Matching* program using the *hMapReduceAll* skeleton where the *matchKeys* is map function and the reduction function is *gatheringKeys*.

6. Evaluation

In this experiment we investigate the performance impact of both *hMapReduce* and *hMapReduceAll* skeletons on homogeneous shared memory architectures, and on different combinations of heterogeneous architectures using the cost model of load distribution. Moreover, we use this experiment to study the contribution of each hardware property that is used in the cost model. For our experiments, we have matched two input images, the size and the number of SIFT keypoints for each image are shown in Table 1. Since

```
int main(int argc, char **argv)
{ Init_HWSkel(argc,argv);
  // generating keypoints
   list=hMapReduceAll(listOfKeys,keysCount,
       ARRAY_LIST,matchKeys,ARRAY_LIST,gatheringKeys);
   Terminate_HWSkel();
}
```

Figure 6: Main program for *Image Matching*

the original sequential sum-Euler program generates irregular data granularity, for simplicity we assume that all the elements in the list have the same value by calculating the sum of the totients between 1 and 2,000,000 of an integer with fixed value of 10,000.

	Size	Keys
img1	1600x1200 (239,616)	71791
img2	1600x1200 (1,205,862)	12378

Table 1: Input Images for Image Matching Application.

6.1 Platform

We conduct our experiments on a heterogeneous cluster of five different parallel architectures located at Heriot-Watt University (Table 2): i)four 2-core (*linux*)machines consisting of Linux RedHat 4.1.2 workstations with a 2.4GHz Intel processor, using 2GB RAM and 2048KB L2 cache. ii) two 8-core Dell PowerEdge 2950 (*lxpara*) machines constructed from two quad-core Intel Xeon 5410 processors running Linux RedHat 5.5 at 2.3GHz with 6144 KB L2 cache and using 16GB RAM. iii) a 4-core (*amaterasu*) machine running Linux RedHat 4.1.2 at 2.93GHz with 8192 KB L2 cache and using 16GB RAM. iv) a 4-core (*brahma*) machine running Linux RedHat 4.1.2 at 3.06GHz with 512 KB L2 cache and using 4GB RAM. v) a 8-core (*jove*) machine running Linux RedHat 4.1.2 at 2.80GHz with 8192 KB L2 cache and using 16GB RAM. Throughout the evaluation section, they will be cited as *(speed/cache)*.

6.2 Heterogeneous Architectures

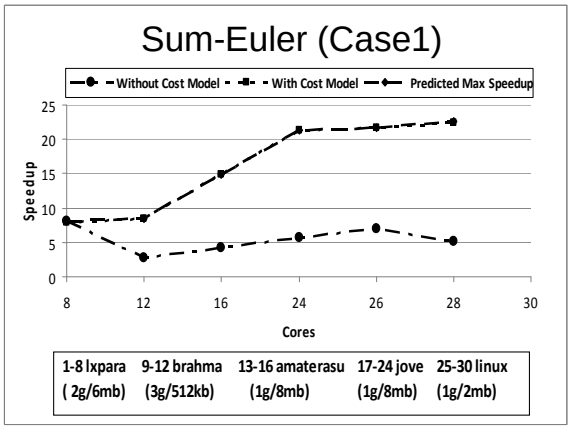
On homogeneous architectures both skeletons deliver linear speedup for *sum-Euler* and *Image Matching* programs.

On heterogeneous architectures, we run our skeletons on three different combinations of the architectures that are described in Section 6.1.

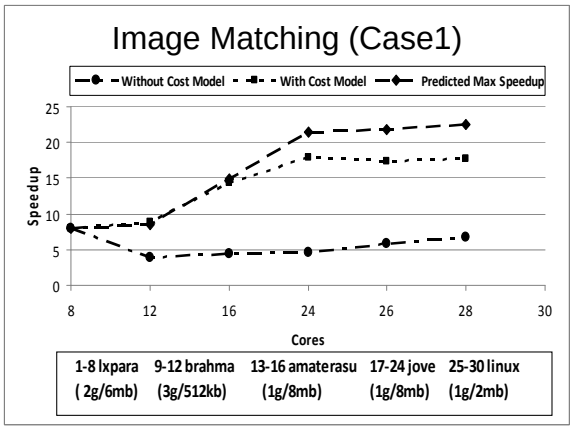
Firstly, Figure 7 plots two different speedup curves for our testbed parallel programs on case1 (*lxpara*, *brahma*, *amaterasu*, *jove* and 2x*linux*).

The results show that the implementation of *HWSkel* library without the cost model delivered worse scalability, where we achieved good speedup on the first fast machine (*lxpara*). The lower speedup curve falls as soon as we introduce heterogeneity by adding the slow machines. This is due to the naive load balancing mechanism which is based on naive technique that distribute load equally between the machines. The upper speedup curve shows the improved performance results for using a load distribution based on the cost model in Section 3. As anticipated, our results show better scalability for our skeletons with the cost model.

Secondly, we combine two 8-core shared-memory machine *lxpara* with 4-core shared-memory machine *brahma*, 4-core shared-memory machine *amaterasu* and two 2-core shared-memory machine *linux* (case2). The results show that in the first two machine (*lxpara1* and *lxpara2*) the performance of our implementations on



(a)



(b)

Figure 7: *hMapReduce* sum-Euler & *hMapReduceAll* image-matching Speedup with/without Cost Model on (Case1)

Predicted Speedup	Experimental-Speedup		Relative Error	
	Img-Match	Sum-Euler	Img-Match	Sum-Euler
8	7.99	7.99	0.125 %	0.125 %
8.51	8.694	8.492	-2.162 %	0.211 %
14.946	14.28	14.89	4.456 %	0.374 %
21.38	17.949	21.25	16.047 %	0.608 %
21.782	17.371	21.649	20.250 %	0.610 %
22.58	17.815	22.44	21.107 %	0.620 %

Table 3: Experimental Speedup and Predicted Maximum Speedup (Case1).

both *hMapReduce* and *hMapReduceAll* skeletons are slightly improved by using the cost model. This is due to the architectural similarity of these machines. As discused in the first combination, adding slow machines leads to poor performance due to the data-load distribution mechanism, again Figure 8 prove that the performance of our skeletons can be improved using the cost model.

Machine name	Architecture	CPU					
		Cores	MHz	L2 Cache	Case1	Case2	Case3
lxpara1	Xeon 4510	8	1998	6144KB	√	√	√
lxpara2	Xeon 4510	8	1998	6144KB		√	
brahma	Xeon(TM)	4	3065	512KB	√	√	√
amaterasu	Core(TM) i7	4	1199	8192KB	√	√	√
jove	Core(TM)i7	8	1200	8192KB	√		
linux01	2 Duo CPU	2	1200	2048KB	√	√	√
linux02	2 Duo CPU	2	1200	2048KB	√	√	√
linux03	2 Duo CPU	2	1200	2048KB			√
linux04	2 Duo CPU	2	1200	2048KB			√

Table 2: Experimental Architectures and Predicted Maximum Speedup.

Predicted Speedup	Experimental-Speedup		Relative Error	
	Img-Match	Sum-Euler	Img-Match	Sum-Euler
8	8	7.981	0 %	0.238 %
16	15.507	15.955	3.081 %	0.281 %
16.522	15.64	16.467	5.338 %	0.332 %
22.95	18.549	22.84	19.176 %	0.479 %
23.353	18.794	23.236	19.522 %	0.501 %
24.157	19.304	24.033	20.089 %	0.513 %

Table 4: Experimental Speedup and Predicted Maximum Speedup (Case2).

Predicted Speedup	Experimental-Speedup		Relative Error	
	Img-Match	Sum-Euler	Img-Match	Sum-Euler
8	8	7.981	0 %	0.237 %
8.51	8.694	8.492	-2.162 %	0.211 %
14.946	14.244	14.87	4.696 %	0.508 %
15.348	13.923	15.669	9.284 %	-2.091 %
16.153	13.967	16.067	13.533 %	0.532 %
16.957	15.495	16.864	8.621 %	0.548 %
17.762	13.152	17.661	25.954 %	0.568 %

Table 5: Experimental Speedup and Predicted Maximum Speedup (Case3).

Thirdly, Figure 9 shows the speedups for *sum-Euler* and *Image Matching* programs on a different heterogeneous architecture (case3) comprising (*lxpara*, *brahma*, *amaterasu*, and 4x*linux*). For this combination, the result looks similar to the first combination that shown in Figure 7. It shows that the performance of our skeleton is improved by using the cost performance model.

Finally, in order to assess the effectiveness and accuracy of the proposed cost model for our skeletons, we calculate the predicted maximum speedup as described in section 3.2 and compared with the experimental speedup for both programs. Tables [3,4,5] lists the predicted speedup, experimental speedup and the relative error on the given system configurations. In the three cases, the relative error for *Sum-Euler* program is very low where the experimental speedup is close to the maximum theoretical speedup limited by our cost model. However, as expected in the *Image-Matching* program the relative error is higher. Observe that the error is around 25.956 percent in the worst case. This is due to the characteristics of this program which suffers high overheads because of frequent communications. Figures 7, 8 and 9 plot the predicted maximum speedup for *Sum-Euler* and *Image-Matching* programs. Observe that the predicted and experimental speedup curves for *Sum-Euler* are identical.

6.3 Alternative Cost Models

Figure 10 shows different speedup results for the implementations of *HWSkel* library on case3 (*lxpara*, *brahma*, *amaterasu*, and 4x*linux*) using the cost model with different architecture properties which includes number of cores, CPU speed, and cache size. Although, the best result is achieved by using all CPU properties in the cost model, we can see that the cache size property has the most significant impact on the cost model performance.

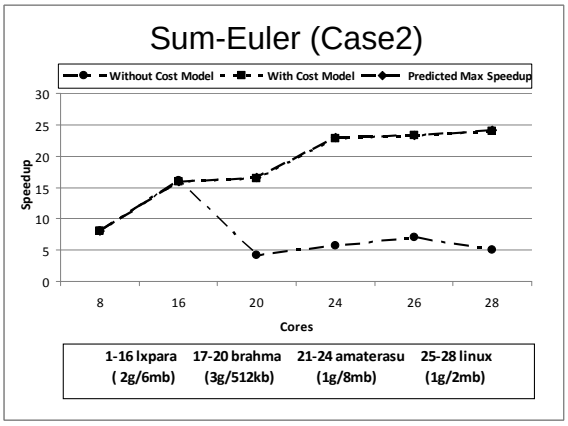
Therefore, we conclude that our skeletons can deliver good parallel performance and scalability on heterogeneous architectures using the static load-balancing mechanism based on architecture properties. On the architectures that are likely to be more heterogeneous the communication cost needs to be added to the cost model.

7. Conclusions and Future Work

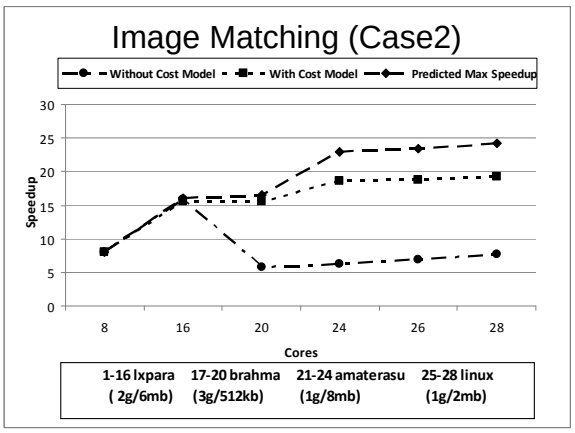
We propose a new architectural cost model for heterogeneous architectures.The cost model is used to determine the data-chunk size according to the number of cores, clock speed and crucially the cache size for each node over the cluster.

In addition, we present a new parallel skeleton library(*HWSkel*) for heterogeneous multi-core cluster architectures. This library is implemented in C on the top of MPI as a distributed-memory programming model and OpenMP for shared-memory parallelism. This means that the skeleton can take straightforward advantage of our cost model to be executed on distributed-memory systems, shared-memory systems or distributed shared memory systems. In particular, it provides data parallel skeletons *hMapReduce* and *hMapReduceAll*. These skeletons are similar to Google's MapReduce model. Moreover, since our skeletons need to be invoked within an MPI initialisation, the *HWSkel* library provides wrapper functions for some MPI routines to keep the user (programmer) away from using a new programming language within the skeletal programs.

Since the naive implementation of our skeletons on heterogeneous multi-core cluster deliver poor performance due to the difference of the nodes capability, we show that it is possible to obtain

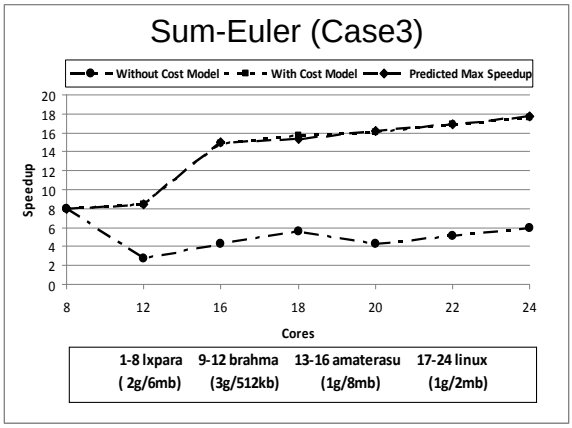


(a)

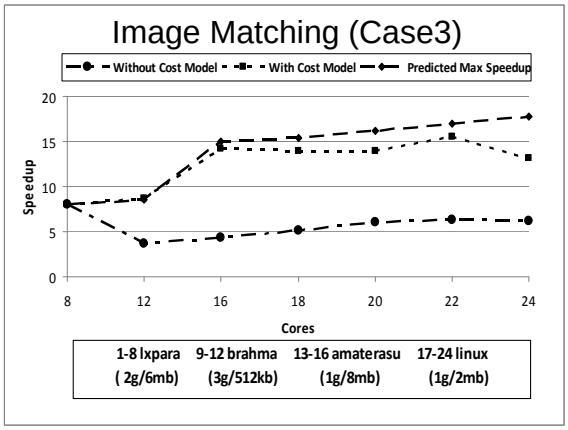


(b)

Figure 8: *hMapReduce* sum-Euler & *hMapReduceAll* image-matching Speedup with/without Cost Model on (Case2)



(a)



(b)

Figure 9: *hMapReduce* sum-Euler & *hMapReduceAll* image-matching Speedup with/without Cost Model on (Case3)

good performance using our cost model for data-load distribution. Our experiments show that the cache size has the most significant impact on the data-load distribution mechanism.

Finally, we anticipate delivering good parallel performance with our cost model in other data parallel skeletons as well as for task parallel skeletons.

In ongoing and future work, the *HWSkel* library will be extended to cover a wide variety of data parallel computations as well as task parallel computations. We anticipate improving the parallel performance of our skeletons by adding the network cost to our cost model.

Acknowledgments

We would like to thank Hans-Wolfgang Loidl for his collaborations on this paper.

References

[1] Apache Hadoop Project. http://hadoop.apache.org/, 2010.

[2] Intelthreading building blocks. http://www.threadingbuildingblocks.org/, 2010.

[3] Skandium. http://skandium.niclabs.cl/, Accessed : 2010.

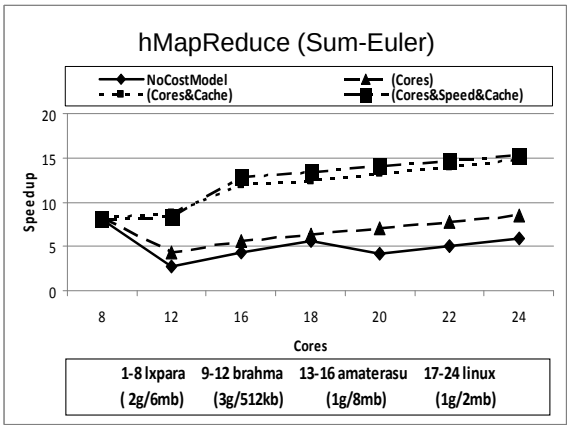
[4] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. Loggp: Incorporating long messages into the logp model — one step closer towards a realistic model for parallel computation. Technical report, Santa Barbara, CA, USA, 1995.

[5] Anne Benoit and Murray Cole. Two fundamental concepts in skeletal parallel programming. In *The International Conference on Computational Science (ICCS 2005) , Part II, LNCS 3515*, pages 764–771. Springer Verlag, 2005.

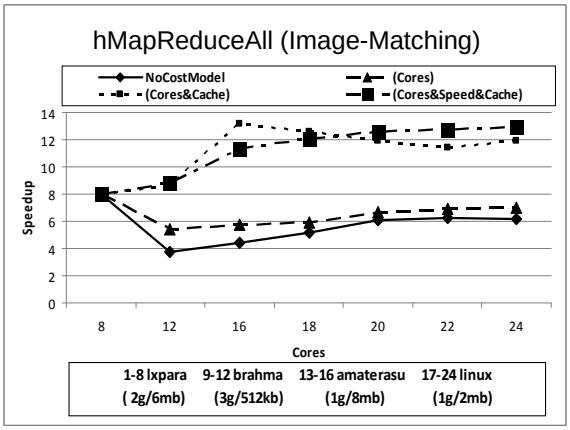
[6] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Flexible skeletal programming with eskel. In *In: 11th Intl Euro-Par: Parallel and Distributed Computing, vol. 3648 of LNCS, 761-770, Lisbona*, pages 761–770. Springer-Verlag, 2005.

[7] Jose Luis Bosque and Luis Pastor. A parallel computational model for heterogeneous clusters. *IEEE Trans. Parallel Distrib. Syst.*, 17:1390–1400, December 2006.

[8] Franck Cappello, Pierre Fraigniaud, Bernard Mans, and Arnold L. Rosenberg. Hihcohp: Toward a realistic communication model for hierarchical hyperclusters of heterogeneous processors. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, IPDPS '01, pages 42–, Washington, DC, USA, 2001. IEEE Computer Society.



(a)



(b)

Figure 10: *hMapReduce* sum-Euler & *hMapReduceAll* image-matching Speedup with Alternative Cost Models on (Case3)

[9] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, October 2007.

[10] M.I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. The MIT Press, Cambridge, MA, 1989.

[11] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, 2004.

[12] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: towards a realistic model of parallel computation. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '93, pages 1–12, New York, NY, USA, 1993. ACM.

[13] John Darlington, A. J. Field, Peter G. Harrison, Paul H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. In *PARLE '93: Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 146–160, London, UK, 1993. Springer-Verlag.

[14] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[15] Steven Fortune and James Wyllie. Parallelism in random access machines. In *Proceedings of the tenth annual ACM symposium on Theory of computing*, STOC '78, pages 114–118, New York, NY, USA, 1978. ACM.

[16] Mohammad M. Hamdan. A survey of cost models for algorithmic skeletons. Technical report, Heriot-Watt University,Dept of Computers and Electrical Engineering, 1999.

[17] Kevin Hammond and Greg Michelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, London, UK, 1999.

[18] D. S. Henty. Performance of hybrid message-passing and shared-memory parallelism for discrete element modeling. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Washington, DC, USA, 2000. IEEE Computer Society.

[19] Gabriele Jost, Haoqiang Jin, Dieter An Mey, and Ferhat F. Hatay. Comparing the OpenMP, MPI, and hybrid Programming Paradigms on an SMP Cluster 1, 2003.

[20] Herbert Kuchen. A skeleton library. In *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, Euro-Par '02, pages 620–629, London, UK, 2002. Springer-Verlag.

[21] Mario Leyton. *Advanced Features for Algorithmic Skeleton Programming*. PhD thesis, Universite de Nice - Sophia Antipolis – UFR Sciences, October 2008.

[22] Mario Leyton and Jose M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference*, pages 289–296, 2010.

[23] D. G. Lowe. Object recognition from local scale-invariant features. In *ICCV '99: Proceedings of the International Conference on Computer Vision-Volume 2*, page 1150, Washington, DC, USA, 1999. IEEE Computer Society.

[24] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, 2004.

[25] Ewing Lusk and Anthony Chan. Early experiments with the openmp/mpi hybrid programming model. In *IWOMP'08: Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, pages 36–47, Berlin, Heidelberg, 2008. Springer-Verlag.

[26] B. M. Maggs, L. R. Matheson, and R. E. Tarjan. Models of parallel computation: a survey and synthesis. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 61–, Washington, DC, USA, 1995. IEEE Computer Society.

[27] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang 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.

[28] Susanna Pelagatti. *Structured development of parallel programs*. Taylor & Francis, Inc., Bristol, PA, USA, 1998.

[29] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference*, pages 427–436, 2009.

[30] Fethi A. Rabhi and Sergei Gorlatch, editors. *Patterns and skeletons for parallel and distributed computing*. Springer-Verlag, London, UK, 2003.

[31] L. A. Smith. Mixed mode MPI/OpenMP programming. *UK High-End Computing Technology Report*, 2000.

[32] Marc Snir, Steve W. Otto, David W. Walker, Jack Dongarra, and Steven Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[33] H.V. Sreekantaswamy, S. Chanson, and A. Wagner. Performance prediction modeling of multicomputers. In *Distributed Computing Systems, 1992., Proceedings of the 12th International Conference on*, pages 278 –285, 9-12 1992.

[34] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.