Architecture-Aware Cost Modelling for Parallel Performance Portability

Evgenij Belikov\textsuperscript{1}, Hans-Wolfgang Loidl\textsuperscript{2}, Greg Michaelson\textsuperscript{2}, and Phil Trinder\textsuperscript{2}

\textsuperscript{1} Institut für Informatik, Humboldt Universität zu Berlin, Germany
belikov@informatik.hu-berlin.de
\textsuperscript{2} School of Mathematical and Computer Sciences, Heriot-Watt University, UK
\{hwloidl,greg,trinder\}@macs.hw.ac.uk

Abstract. In this paper we explore and quantify the impact of heterogeneity in modern parallel architectures on the performance of parallel programs. We study a range of clusters of multi-core machines, varying in architectural parameters such as processor speed, memory size and interconnection speed. In order to achieve high performance portability, we develop several formal, architecture-aware cost models and use characteristics of the target architecture to determine suitable granularity and placement of the parallel computations. We demonstrate the effectiveness of such cost-model-driven management of parallelism, by measuring the performance of a parallel sparse matrix multiplication, implemented in C+MPI, on a range of heterogeneous architectures. The results indicate that even a simple, static cost model is effective in adapting the execution to the architecture and in significantly improving parallel performance: from a speedup of 6.2, without any cost model, to 9.1 with a simple, static cost model.

1 Introduction

To fully harness the potential power offered by rapidly evolving and increasingly heterogeneous and hierarchical parallel architectures, parallel programming environments need to bridge the gap between expressiveness and portable performance. To achieve the latter, the dynamic behaviour of the application needs to adapt to the architectural characteristics of the machine, rather than tying it to one particular configuration. Our long term goal is to automate this process of adaptation by using, on implementation level, architectural cost models and sophisticated run-time environments to coordinate the parallel computations, and, on language level, high-level abstractions such as algorithmic skeletons [7] or evaluation strategies [25]. In this paper, we focus on one concrete application, study the impact of heterogeneity on its performance, and demonstrate performance improvements due to the use of architectural cost models.

Parallel programming has proved to be significantly more difficult than sequential programming, since in addition to the need to specify the algorithmic solution to a problem the programmer also has to specify how the computation is coordinated, taking into account architectural specifics such as processor
and interconnection speed. This is difficult, but manageable on homogeneous high-performance architectures. However, novel architectures are increasingly heterogeneous, being composed of different kinds of processors and using hierarchical interconnections. Therefore, current assumptions that all processors have roughly the same computational power or that communication between any two processors is equally fast, no longer hold.

We contend that architecture-aware cost models can be beneficially applied to narrow the aforementioned gap and facilitate performance portability, which is critical, since hardware architectures evolve significantly faster than software applications. We develop four simple cost models, characterising basic system costs on a heterogeneous network of multi-cores, and demonstrate how the use of these cost models improves parallel performance of a parallel, sparse matrix multiplication algorithm, implemented in C+MPI across a range of parallel architectures.

We continue by presenting general background in Section 2, followed by a discussion of our simple static architectural cost model in Section 3. Subsequently, we present and evaluate experimental results of the application of cost-model-based approach to parallel sparse matrix multiplication in Section 4 and mention related research efforts in Section 5. Finally, we conclude in Section 6 and suggest further research to address the limitations of present work.

2 Background

This section surveys research in parallel architectures, parallel computational models, and high-level parallelism, focusing on the contributions to the adaptation of the execution to massively parallel, heterogeneous architectures.

2.1 Parallel Architectures

An architecture defines a set of interconnected processing elements (PEs), memory units and peripherals. In general, it is intractable to derive absolute prediction of run times of parallel applications, yet it is often possible to use only the most relevant parameters, such as the speed of the processors, the available cache and RAM that characterise computational power of a PE, as well as latency and bandwidth that describe communication costs, to acquire a relative prediction that is sufficient to drive adaptation.

According to Flynn’s taxonomy [11], most of the currently available architectures fit into the MIMD category that can be further subdivided into shared memory architectures that require synchronisation, which may severely limit available parallelism, and distributed memory architectures that suffer the overhead of message buffering. As a higher-level abstraction, distributed shared memory aims at providing the shared address space view on the possibly distributed memory to facilitate location-transparent parallel programming. Furthermore, new architectures are increasingly hierarchical including several memory and network levels and increasingly heterogeneous in respect to the computational
power of the PEs and networking capabilities. To guarantee efficiency, it is re-
required to integrate architectural parameters, due to their impact on performance,
however, to provide expressive abstractions it is also required to hide the low-
level details within a supporting parallel programming environment [23].

2.2 Parallel Computational Models and High-Level Parallelism

Ideally, a model of parallel computation would accomplish what the von-Neumann
model does for sequential computation – it provides a simple way to design
architecture-independent algorithms that would nevertheless efficiently execute
on a wide range of uni-processor machines. On the one hand, such a model should
hide the architecture-specific low-level details of parallel execution and provide
high-level language constructs, thus facilitating parallel programming. On the
other hand, the model should allow for cost estimation and efficient implementa-
tion on a broad range of target platforms, to support performance portabil-
ity [23]. Clearly, there is a tension between these two goals, and historically the
parallel computing community has focused on exploiting architecture-specific
characteristics to optimise runtime.

Occam [16] has demonstrated the advantages of tailoring a parallel program-
ning environment to an architecture to achieve high performance. However, in
this model low-level details are exposed to the programmer, thus decreasing pro-
grammability. Furthermore, tight coupling to a single target platform severely
limits portability.

One of the most prominent cost models for parallel execution is the PRAM
model [12], an idealised analytical shared memory model. The major drawback
of PRAM is that, because of its simplicity, efficient PRAM algorithms may turn
out to be inefficient on real hardware. This is due to the optimistic assumptions
that all communication is for free and that an infinite number of PEs with
unbounded amount of memory are available.

Another prominent cost model is the Bulk Synchronous Parallel model (BSP) [27],
which restricts the computational structure of the parallel program to achieve
good predictability. A BSP program is composed of a sequence of super-steps
consisting of three ordered sub-phases: a phase of local computation followed
by the global communication and then by a barrier synchronisation. Although
appropriate for uniform memory access (UMA) shared-memory architectures,
BSP assumes unified communication, thus renouncing locality as a performance
optimisation, rendering BSP unsuitable for applications that rely on data loca-
ity for performance. The parameters used by the BSP model are the number of
PEs $p$, the cost of global synchronisation $l$, global network bandwidth $g$, and the
speed of processors that determines the cost of local processing. Benchmarks are
used to measure these parameters for any concrete machine.

The LogP model [8] is another well-established cost model for distributed
memory architectures, based on the message passing view of parallel computa-
tion and emphasising communication costs. LogP uses four parameters: $L$, an
upper bound for the latency when transmitting a single word; $o$, the sending and
receiving overhead; $g$, gap per byte for small messages, with $\frac{1}{g}$ representing the
bandwidth; and the number of PEs $P$. The model has been extended to account for long messages (LogGP) [1], for heterogeneity (HLogGP) [5] that uses vector and matrix parameters instead of scalar parameters in LogGP, and for hierarchy (Log-HMM) [20].

The aforementioned models provide a good selection of parameters, which are likely to be relevant for parallel performance on distributed memory machines. However, no sufficiently accurate parallel computation model that accounts for both heterogeneity and hierarchy exists to date.

An all-explicit approach to parallelism, although providing highest performance on one platform, is non-portable and error-prone. An all-implicit approach [18] can provide performance guarantees only for severely restricted forms of parallelism. Ultimately, we envision the almost-implicit high-level parallelism [26] as the most promising approach for parallel programming that provides expressiveness and ease-of-use without sacrificing portable performance.

### 2.3 Cost Modelling

In general, predicting computational costs is mostly intractable, and therefore some qualitative prediction is used in practice to successfully guide adaptation. Abstract models, developed for algorithm design, are usually architecture-independent by assigning abstract unit cost to the basic operations. For more accurate prediction, the cost model needs to be parametrised with the experimentally determined characteristics of the target architecture. On the one hand, static cost models incur less overhead than the dynamic cost models, however, they do not take dynamic parameters such as system load and network contention into account, which may significantly affect performance of the executing program. On the other hand, dynamic models suffer from additional overhead that can cancel out the benefits of more accurate performance prediction.

In the compositional cost model associated with BSP, the cost of a program is the sum of costs of all super-steps, where the cost of a super-step is calculated as the maximum of the sum of the costs for local computation, inter-process communication, and global synchronisation. The costs of global synchronisation, the network bandwidth, and the speed of PEs are empirically determined by running a benchmark on a given target platform.

Cost models can be used at design time to help choosing suitable algorithms and avoiding restricting the parallelism too early. At compile time, cost models are useful for performance debugging and for automatic optimising code transformations based on static analysis techniques such as type inference and cost semantics. Ultimately, at run-time, cost models can guide dynamic adaptation.

Higher-level cost models are commonly provided for algorithmic skeletons [9], however, they do not incorporate architecture information. Bischof et al. [4] claim that cost models can help developing provably cost optimal skeletons. Another conceivable use is for improving coordination or for determining granularity and mapping according to a custom goal function such as achieving highest utilisation of resources, maintaining highest throughput, lowest latency or achieving best performance to cost ratio [24].
Cost models have been successfully applied to drive decentralised dynamic load-balancing for autonomous mobile programs [10], which periodically use a cost model to decide where in the network to execute. If the cost of execution on the current location is higher than predicted communication and execution cost on another location, then migration is justified.

Hardware models provide the most concrete and accurate cost models, which can be used to determine worst case execution time for safety-critical systems [28]. Ultimately, cost models provide a way for a program, to a certain extent, to predict its own performance, which is a prerequisite for successful self-optimisation [17].

3 Architecture-Aware Cost Modelling

In this section we motivate and discuss the cost models that we are using in our experiments to adapt the behaviour of the program to underlying hardware characteristics. As opposed to characteristics of the application and dynamic runtime parameters, this work focuses solely on static architectural parameters. We devise and refine a simple cost model that is used to adapt the execution to a new target platform by determining suitable granularity and placement that affect static load-balancing for a chosen application. As mentioned above, we identify the number of PEs, the speed of PEs, the amount of L2 cache and RAM, as well as latency as the most relevant parameters.

Typically, the only parameters used in the parallelisation of algorithms are the number of available PEs $P$ and the problem size $N$. The naive approach, which serves as a baseline case, is to distribute chunks of work of equal size of $\frac{N}{P}$ among the PEs. This is sufficient for perfect load balancing for regular applications on regular data, if run on homogeneous and flat architectures. However, this simple strategy results in poor performance for applications that aim at exploiting irregular parallelism, operate on irregular data or run on hierarchical, heterogeneous systems, since it is necessary to distribute work in a way that accounts for the computational power of each PE and for the communication overhead. This intuition is captured by the following equation: $C_i = \frac{S_i}{S_{\text{all}}} \cdot N$, where $S_i$ denotes the computational power of a node characterised by the CPU frequency and $S_{\text{all}}$ is the overall computational power of the system calculated as $\sum_{i=0}^{n} S_i$. Our results in Section 4 underline this point.

In the baseline case we set $S_i = 1$, whereas in the initial cost model (CM0) we start with a single additional parameter – the CPU speed of a PE, thus $S_i = CPU_i$. At the first glance it is a primary attribute that characterises the computational power of a PE and can be used in addition to $P$ and $N$ to determine the chunk size $C_i$. However, practitioners report the importance of increasing the cache hit ratio [15]. A larger cache size results in fewer main memory accesses, thus improving the performance of an application.

Therefore, the next step is to include the L2 cache size in our cost model, resulting in the cost model (CM1) with $S_i = aCPU_i \cdot bL2_i$, where $L2_i$ denotes the cache size of a node (in kilobytes) and $a$ as well as $b$ are scaling factors that can be tuned to prioritise one parameter over the other. Some potentially
important aspects, that are not yet covered in our model are cache hierarchies and application characteristics such as data sharing patterns.

Since very large problems may exceed RAM size, we include RAM as a binary threshold parameter in the next cost model – we require the problem size to fit into RAM if performance is critical. Moreover, we can incorporate the knowledge of the size of the RAM of each node in the cache-to-RAM ratio that further characterises the computational power of the node and facilitates adaptation to heterogeneous architectures. The resulting cost model (CM2) defines $S_i$ as follows: $S_i = \frac{a\cdot CPU_i \cdot bL^2}{c\cdot RAM_i}$. In addition to CM1, $RAM_i$ denotes the size of the main memory (in megabytes) and $c$ is the corresponding scaling factor.

Finally, we refine our model once again and include the latency as additional parameter that characterises the interconnection network. On the one hand, latency can be used as a binary parameter to decide whether to send work to a remote PE. If the latency is too high we would not send any work, otherwise we can use latency in the cost model to determine the chunk size that would be large enough to compensate for higher communication overhead. Currently, we simply use latency to avoid slowdown by not sending any work to exceedingly far remote PEs. In ongoing work we aim to quantify the relationship among the benefit of offloading work and the communication overhead that can cancel out the benefits.

Generally, developing a cost model is an iterative process, where one discovers a seemingly important parameter and integrates it in the cost model, then the model is tested empirically and the performance is compared to that of other models. After the evaluation phase the model can be further refined and tuned during a further iteration if necessary.

4 Experimental Results and Evaluation

This section presents the experimental design, describes the chosen application and the available target architectures, and evaluates the experimental results obtained on different target architectures by comparing the baseline implementation without any cost model to the cost-model-enhanced versions.

4.1 Experimental Design

The experiments include running versions of the sparse matrix multiplication (discussed in Section 4.2) that differ in the used cost model and thus in the way the work is distributed on different target platforms. We measure the run times for each architecture, using PE numbers in the range 1..16 for a fixed data size of 8192 × 8192 with sparsity of 1% for both matrices to be multiplied. To minimise the impact of interactions with the operating system and other applications, the median of the run times of five runs is used for each combination.

In our experiments, we vary heterogeneity by adding different machines from different subnets and by using different numbers of the available cores. We also study a separate experiment, adding a slow machine (linux01) to the network.
We study two policies of process allocation to PEs: in the unshuffled setup we send chunks to a host as many times as there are cores to be used and then continue with the next host, whereas in the shuffled setup we distribute the work in a round robin fashion among all multi-core machines. Thus, a 4 processor execution on 4 quad-cores uses all 4 cores on one quad-core in an unshuffled setup but uses 1 core of each quad-core in a shuffled setup. This variation aims at investigating the predictability of performance, i.e. the change of scalability as new nodes are introduced and sending data to hosts on other subnets is required.

In order to assess the influence of the interconnection network, we study configurations with varying communication latencies: an all-local setup with fast connections is compared to a setup with $\frac{1}{8}$ remote machines. We expect the most sophisticated cost model to provide best performance improvements in the most heterogeneous and hierarchical case, whereas in the homogeneous and the less heterogeneous cases we expect the cost models to have no negative impact on performance.

4.2 Parallel Sparse Matrix Multiplication

We have chosen sparse matrix multiplication as an application that is representative for a wide range of high-performance applications that operate on irregular data [3]. A matrix is termed sparse if it has a high number of zero-valued elements. Computationally, sparseness is important because it allows to use a more space-efficient representation. Algorithms operating on such a representation will automatically ignore zero-entries, at the expense of some administrative overhead, and will therefore be more time-efficient. In practice, we consider large matrices with less than 30% of the elements being non-zero.

Matrix multiplication for two given matrices $A \in \mathbb{Z}^{m \times n}$ and $B \in \mathbb{Z}^{n \times l}$, written $C = A \ast B$, is defined as $C_{i,j} = \sum_{k=0}^{n} A_{i,k} \ast B_{k,j}$. Here, the focus is on how sparse matrices are represented in memory and how they can be distributed across the available PEs efficiently allowing for parallel calculations. Our cost model is used to improve parallel performance and performance portability by determining the size of the chunk of the result matrix to be calculated by each processor, thus matching the granularity of the computation to the processor’s computational power.

Although low-level and rather difficult to use, we have decided to use C and MPI for parallel programming, since this combination proved to facilitate development of efficient yet portable software. The focus in this paper is on system level implementation of strategies to achieve performance portability, rather than on language level abstractions or automated support for parallelisation. In particular, we use the MPICH1, since only this implementation of the MPI standard for message passing supports mixed mode 32/64-bit operation. The `MPI_Wtime()` function facilitates accurate run time measurements.

We follow the Partitioning - Communication - Agglomeration - Mapping (PCAM) methodology advocated by Foster [13], also partly inspired by the Pattern Language for Parallel Programming suggested by Mattson et al. [21], and...
employ the load balancing scheme recommended by Quinn for static irregular problems [22, p. 72].

In the Partitioning phase we identify all the available parallelism of finest possible granularity, which is in our case by domain decomposition a single vector-vector multiplication of the corresponding components of a row of $A$ and of a column of $B$. However, as we recognise in the Communication phase, exploiting parallelism of such fine granularity requires an obviously too large volume of communication (for each multiplication send both elements and receive one resulting element). Since communication is responsible for a significant part of the parallel overhead, we reduce communication during the Agglomeration phase. First, we combine all the operations that contribute to the same element of the result matrix to the same task, hence reducing the communication overhead from $O(n^4)$ to $O(n^2)$ (we send less but larger messages). Moreover, since the problem size is known in advance, we can further increase the grain size by combining several tasks to a larger task that is associated with larger data size, so that each PE receives the whole matrix $A$ and a column of $B$, thus further reducing the number of messages to be sent. By now we have one chunk of data for each PE, therefore, Mapping is trivial. Note that by using an architecture-aware cost model, better static load balancing is achieved, since more powerful PEs receive proportionally more work. The straightforward algorithms are presented in the listings below.

**Listing 1.1.** Master process

| generate matrix A                      |
| generate matrix B                      |
| gather parallel architecture descriptions |
| broadcast matrix A to all PEs          |
| determine chunk sizes of the PEs based on architecture information |
| scatter chunks of matrix B among the PEs |
| perform local computation (matrix–vector multiplication) |
| gather partial results from the PEs    |
| combine partial results                |

We decided to use collective operations, specifically we use `MPI_Scatterv()`, `MPI_Gatherv()`, and `MPI_Bcast()`, instead of a loop using the lower-level send and receive operations for possibly higher performance, following the convincing recommendation made by Gorlatch in [14]. At this point we rely on an efficient implementation provided by the MPICH library.

A matrix is stored as an array of $(row, column, value)$ triplets to maintain a high level of efficiency, since the elements are sorted in consecutive memory and allow for fast access. However, such an approach increases the coupling between the representation and the functions that use it which requires significant code alterations in case the compression scheme is to be exchanged. We randomly
Listing 1.2. Worker process

- receive matrix A and one chunk of matrix B from master
- perform local computation (matrix–vector multiplication)
- send results (chunk of matrix C) back to master

generate two sparse matrices with a specified sparseness, for flexibility, and a seed parameter, for repeatability. Thus, this first phase of the algorithm is local and performed only by the master and is not included in the run time.

The structure of the matrix multiplication algorithm is fairly standard, and we focus here on the architecture-specific aspects, in particular on gathering the architecture descriptions and determining granularity and placement based on these descriptions. After the two matrices are generated, the master gathers the information about the available architecture, in particular processor speed, RAM size, cache size and latency. All of our cost models are static, and we therefore do not monitor dynamic aspects of the computation, thus avoiding additional overhead. In the next step the sizes of the chunks are determined using a given cost model, and the data is distributed across the available PEs. After receiving the work, each PE, including the master, performs matrix-vector multiplications locally and sends the results back to the master process.

4.3 Target Architectures

All machines available at Heriot-Watt University are local, hence remote latency is emulated using the netem utility that adds an additional queueing discipline to the operating system kernel and allows to artificially delay packets at the interface. In different setups some of the available machines are combined to architectures that range from homogeneous flat architectures to heterogeneous and hierarchical ones.

The Beowulf machines are 64-bit Dual Quad-Core Intel Xeon E5504 machines with each of its 8 cores running at 2.0 GHz, sharing 2x4 MB L2 cache and 6 GB RAM. The 32-bit Dual Quad-Core Intel Xeon E5410 multicore (lxpara) machines with each of its 8 cores running at 2.33 GHz, share 2x6 MB L2 cache and 8 GB RAM. The dual-core linux lab machines are 32-bit Pentium D machines with two cores at 3.4 GHz each, sharing 2x2 MB L2 cache and 3 GB RAM. The slower linux01 machine is a 32-bit Intel Core 2 Duo E4600 machine with two cores at 2.4 GHz each, sharing 2 MB L2 cache and 2 GB RAM, with a legacy network interface card that can handle only 10/100 Mbit connections. All systems run a version of CentOS 5.5 and are connected via switched Gigabit Ethernet as part of a respective subnet of similar nodes.

Different setups are represented by a list of integers denoting the number of participating nodes and the number of corresponding cores followed by additional information such as the type of the machines, and concrete setup, i.e. with

\[\text{http://www.linuxfoundation.org/collaborate/workgroups/networking/netem}\]
linux01 vs without linux01, all-local or partly remote. For example, 1x8 1x4 1x2 2x1 (beowulf, lxpara, linux01, linux lab) describes a configuration with five participating nodes of which one is a beowulf machine using all of its cores, another is a lxpara multicore machine using four of its cores, another node is linux01 using both cores, and the remaining two nodes are the linux lab machines using one core each. By default we refer to the all-local unshuffled experiments without linux01, where the x8 and x4 machines are beowulf or multicore machines, and the x2 and x1 are the linux lab machines.

4.4 Results and Evaluation
In this section the results of the experiments are presented. First, we show results for different unshuffled all-local setups. Then we compare selected shuffled vs unshuffled, all-local configurations. Subsequently, we compare the cost models for the most heterogeneous configuration: shuffled, all-local, 2x4 2x2 4x1. Moreover, we investigate the impact of latency for a partly remote configuration and suggest a means to avoid the slowdown.

![Graph showing speedups for selected unshuffled all-local setups](image)

**Fig. 1.** Speedups for selected unshuffled all-local setups

*Different All-Local Configurations:* We begin with a homogeneous (all nodes have the same computational power) and flat (all nodes are on the same subnet) architecture. Figure 1 shows the speedups for up to 16 processors, using configurations of varying heterogeneity. The speedup is almost linear on one multicore
machine and decreases due to communication overhead once we leave the node (more than 8 processors). Further decrease in performance can be observed for configurations that include machines from a different subnet (\texttt{linux lab}), due to the increased communication overhead. In summary, the graphs in Figure 1 depict the decrease in performance with increasing heterogeneity of the setup. This observation is the starting point for our work on using architectural cost models to drive the parallel execution.

![Graph showing speedups for selected unshuffled versus shuffled all-local setups](image)

**Fig. 2.** Speedups for selected unshuffled versus shuffled all-local setups

\textit{All-Local, Heterogeneous and Hierarchical Case:} The $2 \times 4$ $2 \times 2$ $4 \times 1$ configuration in Figure 1 represents the most heterogeneous case, where instances of each available architecture are used (\texttt{beowulf}, \texttt{lxpara}, \texttt{linux lab} and the slow \texttt{linux01}). This configuration exhibits significant performance degradation: compared to the homogeneous case, with a speedup of almost 10.0 on 16 processors, the speedup drops to 5.0. While some of this drop is inevitable, due to the hardware characteristics, we show below that this result can be significantly improved by exploiting information from an architectural cost model.

Figure 2 demonstrates the effect of the placement strategy on predictability. For a small number of PEs it is beneficial to keep the execution local, fully exploiting one multi-core before placing computations on a different multi-core.
This strategy is tuned to minimise communication cost. However, if we are more concerned with steady and predictable (but slower) increase in performance, we should use a shuffled setup.

The results on all-local configurations suggest that our cost model, discussed in Section 3, improves performance on heterogeneous setups and does not impede performance on homogeneous setups. However, if a network hierarchy is introduced we need to account for additional communication overhead for remote nodes.

![Graph showing speedup comparison](image)

**Fig. 3.** Speedups for the most heterogeneous configuration using different cost models.

Figure 3 presents a comparison of different cost models, that are all performing better than the baseline case: on 16 processors, the speedup improves from 6.2, without cost model, to 9.1, with cost model CM0. With the current settings for the scaling parameters, the more advanced cost models do not achieve further improvements in performance, and the simple cost model CM0 performs best. Presumably, other cost models can be further calibrated to further increase the performance. Moreover, application-specific and dynamic system parameter should be taken into account, since they are likely to influence the importance of the available architectural parameters.

**Partly Remote, Heterogeneous and Hierarchical Case:** To investigate the impact of latency on the parallel performance, we use a partly remote setup where two nodes are emulated to have high latency interconnect. Figure 4 illustrates the importance of taking the latency into account. All the cost model that do not use
latency as a parameter send work to remote PEs leading to a parallel slowdown, which can be avoided by latency-aware models by simply not using the remote PEs.

In ongoing work we aim to devise a more sophisticated cost model, that would send off the work only in case it is beneficial to achieve additional speedup.

5 Related Work

HeteroMPI [19] is an attempt to integrate cost modelling with MPI in order to exploit heterogeneous clusters more efficiently by automating the optimal selection and placement of a group of processes according to both architecture and application characteristics. It defines an additional abstraction layer on top of MPI and requires the user to provide a performance model that incorporates the number of PEs, their relative speed and the speed of communication links in terms of latency and bandwidth, as well as the volume of computations, the volume of data to be transferred between each pair of processes in the group, and the order of execution of computations and communications. However, to create such a model the user needs intricate knowledge of the target architecture and of the application. Additionally, familiarity with a special performance model definition language is required, steepening the initial learning curve. Since HeteroMPI employs only a partially dynamic model, it may not be suitable to
support applications that exhibit irregular parallelism. In contrast, our work focuses on an architectural cost model and aims to improve performance without additional knowledge about the application structure.

Another recent framework that facilitates exploitation of heterogeneous platforms is hwloc \cite{6}. It uses sysfs to gather hardware information about processors and the memory hierarchy of the target architecture, and exposes it to the application and run-time environment. The latter in turn adapts the placement and communication strategies used in running the application. For instance, threads that share data shall be placed so that they share a common cache.

The design of our cost model is based on results in \cite{2}, which achieves good speedups using a simple cost model that does not take into account RAM and latency on a heterogeneous architecture for a rather artificial application\footnote{computing the Euler totient sum}. While the focus in this work is on hybrid programming models for algorithmic skeletons, we study the behaviour on a standalone parallel application.

6 Conclusion

This paper makes a case for using a high-level modelling approach to achieve performance portability on modern parallel hardware. More specifically, formal, architecture-aware, static cost models are used to enhance parallel performance on a range of clusters of multi-core machines – an emerging class of parallel architectures. We quantify the impact of heterogeneity by comparing the performance of a parallel sparse matrix multiplication application on a range of clusters of multi-core machines: on 16 processors the speedup in the most heterogeneous setting is only about half of the speedup in a homogeneous setting.

By making decisions on the size and placement of computations based on a simple, static, architectural cost model, the application doesn’t have to be changed when moving to a different parallel machine. Our example program achieves good performance portability even on the most heterogeneous configuration: the speedup on 16 processors increases from 6.2 (without any cost model) to 9.1 (using a simple, static cost model). Although based on one application, these results on a range of configurations with increasing heterogeneity indicate that using a simple cost model, incorporating information on processor speed, memory size and cache size, can already achieve a high degree of performance portability.

Furthermore, this static cost model has only negligible overhead in the still important homogeneous, flat case. Our results also show to a user, who values predictability over performance, that a shuffled placement strategy, which evenly distributes computations across multi-cores rather than fully exploiting individual multi-cores, significantly improves predictability, while moderately impacting overall performance. This observation is relevant, considering that with increasing numbers of cores per machine, shared memory access will become a bottleneck, and therefore fully exploiting all cores on one machine might
no longer be an optimal strategy. Moreover, our results affirm that latency is critical for load balancing decisions on partly remote configurations.

**Limitations:** The main limitation of our approach to cost modelling is that we do not incorporate application specific information and no dynamic load information. Although this design decision reduces the overhead associated with the cost model, it limits the flexibility of the system by missing opportunities of dynamic reconfiguration of the parallel computation, which could improve performance in highly dynamic environments. Moreover, we rely on heuristics and emphasise the need of a suitable high-level parallel computational model to allow for more systematic cost modelling.

From the implementation point of view, it would be beneficial to use a common architecture description format (and an architecture description language) to enable automated discovery of an architecture. Packaging such functionality as a library would further enhance the practicability of our approach.

**Future Work:** Although we have demonstrated the feasibility of using architecture-aware cost models, more work is needed to generalise the results to other application domains. Including further architectural parameters as well as application specific ones (e.g. cache hit patterns, communication patterns) and dynamic load information in the cost model has the potential for more accurate prediction, which pays off especially in heterogeneous and hierarchical setups. An interesting direction is also the investigation of the use of more sophisticated cost models within a run-time system or as part of algorithmic skeletons in accordance with a semi-implicit approach to parallel programming.

We advocate developing an open and standardised discovery and composition mechanism that will enable automated collection of relevant architectural information at run-time and might use some sort of a context dissemination middleware. This mechanism would benefit from vendors providing accurate architecture descriptions of their products in a portable format, facilitated by the use of a suitable architecture description language. Ultimately, devising a unified model of parallel computation that would account for both heterogeneity and hierarchy would enable a more principled approach to cost estimation facilitating the development of algorithms that are provably efficient on a wide range of parallel architectures.

**Acknowledgements**

We thank Artur Andrzejak and Khari Armih for inspiring discussions and helpful comments. The first author has been partly supported by an ERASMUS scholarship.
References


