Architecture-Aware Cost Modelling for Parallel Performance Portability

Evgenij Belikov, Hans-Wolfgang Loidl, Greg Michaelson, Phil Trinder

School of Mathematical and Computer Sciences
Heriot-Watt University
EH14 4AS Edinburgh
{eb120,hwloidl,greg,trinder}@hw.ac.uk

Abstract: Languages for efficient parallel programming need to achieve high performance portability in order to harness the power offered by rapidly evolving parallel architectures. We use a combination of high-level architecture-aware cost modelling with a low-level, explicit control of coordination as a programming model to improve performance portability. We explore and quantify the impact of heterogeneity in modern parallel architectures on the performance of parallel programs on a range of clusters of multi-cores, varying in architectural parameters such as processor speed, memory size and interconnection speed. Additionally, we develop several formal cost models and automatically use these architectural characteristics to determine suitable granularity and work placement. The effectiveness of such cost-model-driven management of parallelism on common-place cluster hardware is demonstrated by measuring the performance of a parallel sparse matrix multiplication, implemented in C+MPI, on a range of heterogeneous architectures. On a cluster with 16 cores, the speedup increases from 6.2, without any cost model, to 9.1, indicating that even a simple, static cost model is effective in adapting the execution to the target architecture and in significantly improving parallel performance and scalability with negligible overhead.

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 [Col89] or evaluation strategies [THLJ98]. 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.

The approaches to parallel programming range from high-level implicit, where the coordination is completely hidden from the programmer, over semi-explicit, to lower-level explicit models, where every aspect of coordination is exposed to the programmer [ST98]. While fully implicit approaches proved intractable for unrestricted parallelism, explicit approaches were historically favoured for optimisation capabilities. However, manual optimisations are prone to error and often result in non-portable code. Therefore, following the principle of separation of concerns, we use an architecture-aware cost model to automatically partition the work, adapting to the target architecture, hence improving load balancing on heterogeneous architectures and raising the level of abstraction, as well as increasing performance portability. Moreover, we use a message-passing programming model, since we cannot assume a heterogeneous and hierarchical architecture to be a shared-memory architecture and wish to avoid the complexities of implementing application-level virtual shared-memory abstraction. This eliminates the synchronisation overhead of shared-memory models, since there is no shared state, at the cost of packing and buffering. To avoid deadlocks we rely on a deadlock-free library implementation of collective operations. Furthermore, due to the simple communication pattern and data layout of the chosen application, it is possible to ensure that no race conditions can occur. By keeping the cost model encapsulated within a single function, minimal code changes are required to exchange the model without affecting the application code. Provided the necessary language and communication library support, the need for manual changes of code while migrating to another target architecture is eliminated altogether.

We contend that architecture-aware cost models can be beneficially applied to connect languages and architectures, narrowing the aforementioned gap, and in particular to 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 and MPI, across a range of parallel architectures of varying heterogeneity with negligible overhead.

We continue by presenting general background in Section 2, followed by a discussion of our simple static architectural cost model and its integration within the application 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

An architecture defines a set of interconnected processing elements (PEs), memory units and peripherals. According to Flynn's taxonomy [Fly66], most of the currently available architectures fit into the MIMD category but also may have SIMD components. Moreover, 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 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 [Ski94]. We use the explicit low-level message-passing model that is commonly used to implement portable HPC applications on homogeneous and flat distributed-memory machines, and aim at enhancing performance portability for a variety of architectures by adding an architecture-aware cost model to guide the adaptation.

2.1 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 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 implementation on a broad range of target platforms, to support performance portability [Ski94]. Clearly, there is a tension between these two goals, and historically the parallel computing community has focused on exploiting architecture-specific characteristics to optimise run time.

One of the most prominent cost models for parallel execution is the PRAM model [FW78], 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) [Val90], which restricts the computational structure of the parallel program to achieve fairly good predictability. A BSP program is composed of a sequence of super-steps, each consisting of three ordered sub-phases: a phase of local computation followed by the global communication phase and then by a barrier synchronisation phase. Although appropriate for uniform memory access shared-memory architectures, BSP assumes unified communication, thus renouncing locality as a performance optimisation, rendering BSP unsuitable for applications that rely on data locality 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.

MultiBSP is a recent update of the BSP model that aims primarily at modelling computations on multi-core architectures with cache hierarchies [Val11]. Benchmarks are used to determine the model parameters for each target architecture.

The LogP model [CKP+93] is another well-established cost model for distributed-memory architectures, based on the message-passing view of parallel computation 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 subsequently extended to account for long messages (LogGP [AISS95]), for heterogeneity (HLogGP [BP06]) that uses vector and matrix parameters instead of scalar parameters in LogGP, and for hierarchy (Log-HMM [LMR95]).

These models provide a good selection of parameters that are relevant for performance on distributed-memory architectures. However, no sufficiently accurate unified parallel computation model that accounts for both heterogeneity and hierarchy exists to date.

2.2 Cost Modelling

In general, predicting computational costs proved 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 a 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 ones, however, they do not take dynamic parameters such as system load and network contention into account, which may severely affect performance. On the other hand, dynamic models suffer from additional run-time overhead that can cancel out the benefits of more accurate performance prediction.

Cost models can be used at design time to help choosing suitable algorithms and to avoid 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 [Dar93], but they do not incorporate architecture information. Bischof et al. [BGK03] 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 [TCH+11].

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

3 Architecture-Aware Cost Modelling

As opposed to characteristics of the application and dynamic run-time 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. Rather than for run time prediction, these parameters are used by the cost model to estimate relative computational power of the available PEs which determines the partitioning of work.

3.1 Analytic, Static Cost Models

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:

$$C_i = \frac{S_i}{S_{all}} \cdot N, \text{where } S_{all} = \sum_{i=1}^{P} S_i$$

with C_i being the chunk size that represents the work assigned to PE i depending on its computational power S_i in relation to the overall computational power of the system S_{all}. Our results in Section 4 underline this point. In the baseline case we set $S_i = 1$.

CPU-aware Cost Model (CM0): In the initial cost model we start with a single additional parameter — the *CPU speed* of a PE. 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. This results in the following cost model by changing the way relative computational power is estimated:

$$S_i = CPU_i$$

where CPU_i denotes the speed of PE i. However, there is consensus amongst practitioners on the high importance of increasing the cache hit ratio. A larger cache size results in fewer main memory accesses, thus improving the performance of an application [AMT11].

CPU-Cache-aware Cost Model (CM1): Thus the next step is to include the *L2 cache size* in our cost model, resulting in the following change to CM0 to obtain CM1:

$$S_i = aCPU_i \cdot bL2_i$$

where $L2_i$ denotes the cache size associated with a PE (in kilobytes) and a as well as b are scaling factors that can be tuned to prioritise one parameter over the other. However, the cache miss ratio for our application is fairly low[1], which explains why cache does not appear to be a good predictor in the untuned CM1 that performs worse than CM0. Due to time constraints, we have not tuned the scaling factors to further improve performance.

Based on probability theory, large estimated values for S_i suggest that both CPU speed and cache size are large, therefore a multiplicative rather than additive model is a natural choice in this case. This helps to avoid specific large values dominating all the other parameters, which would increase the amount of required tuning. Moreover, we display two distinct scaling factors for flexibility and to emphasise that each parameter can be separately tuned. However, all scaling factors can be combined into a single factor in the actual implementation.

CPU-Cache-RAM-aware Cost Model (CM2): 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 ratio reflects the intuition that larger caches and larger RAM size generally positively affect performance. However, the strength of the respective effect depends on the particular application. The resulting change to CM1 to obtain CM2 is as follows:

$$S_i = \frac{aCPU_i \cdot bL2_i}{cRAM_i}$$

where RAM_i denotes the size of the main memory (in megabytes) and c is the corresponding scaling factor.

CPU-Cache-RAM-latency-aware Cost Model (CM3a): Finally, we refine our model to include the *latency* as an additional parameter that characterises the interconnection network, resulting in the following change to CM2 to obtain CM3:

$$S_i = \begin{cases} \frac{aCPU_i \cdot bL2_i \cdot dL_i}{cRAM_i}, \text{if } L_i < t \\ 0 \qquad\qquad , \text{otherwise} \end{cases}$$

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 L_i exceeds a threshold t (both in ms), we do not send any work, otherwise we use latency with a corresponding scaling factor d in the cost model to determine the chunk size that is large enough to compensate for higher communication

[1] 0.03%, measured for the sequential run with cachegrind

overhead. Currently, we simply use latency to avoid slowdown by not sending any work to exceedingly far remote PEs. In ongoing work (CM3b), we aim to quantify the relationship among the benefit of offloading work and the communication overhead that can cancel out the benefits, as well as to automate the choice of a suitable threshold value.

3.2 Embedding within an Application

Although a cost-model is language-agnostic, it needs to be expressed in some language and embedded in the application code within a single function that determines the granularity of work packages for given problem size and is informed by the description of the target architecture. In our case, the cost model is implemented directly in the host language and it influences the behaviour of the parallel execution by determining the partitioning of work, which is distributed to the workers by using collective operations. Process management is handled implicitly by the communication library that places one process on each used PE. Though performance varies for different process placement policies that are described in Section 4.1, the use of cost models decreases this variation.

To introduce a static, analytic cost model, almost no changes are required, since partitioning is performed in any case. One refactoring step suffices to separate the partitioning from other coordination and computation concerns. In a well-designed modular application this step would be unnecessary. Instead, merely the partitioning function would be affected by introducing a cost model. Furthermore, depending on the host language, exchanging a cost model can be accomplished during the run-time of the application, facilitating the creation of adaptive components.

Moreover, it is conceivable to create an abstract partitioning function, that is then instantiated with the specifics of the data structure that represents work. We have studied such generic partitioning functions in the context of parallel Haskell before [LTB01] and demonstrated their advantages in terms of code re-usability. Hence, the cost model is neither tied to a specific language, nor to a data structure, nor to a specific algorithm.

Since we use a low-level coordination model, the well-known disadvantages remain: a careless programmer may introduce deadlocks and non-portable optimisations. At this point we rely on a highly tuned and deadlock-free communication library and ensure the absence of race conditions by using non-overlapping buffers. Ideally, the library would avoid the packing and buffering overhead if the work is sent to a PE on the same node.

4 Experimental Results and Evaluation

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.

4.1 Experimental Design

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 explore application behaviour under two process allocation policies: in the `unshuffled` setup we send chunks to a PE 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. 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.

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 [Asa06]. 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 $C = A*B$, is defined as $C_{i,j} = \sum_{k=0}^{n} A_{i,k} * B_{k,j}$. 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 apply, 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 supports mixed-mode $32/64$-bit operation.

We follow the *Partitioning - Communication - Agglomeration - Mapping* (PCAM) design methodology [Fos95] and employ the load balancing scheme for static irregular problems [Qui04, p. 72]. In our case mapping is trivial, since we partition the work statically in exactly as many chunks as there are available PEs. To minimise communication overhead we broadcast the whole matrix A and scatter disjunct chunks of columns of the matrix B to the workers. Note that by using an architecture-aware cost model, better static load balancing is achieved, since more powerful PEs receive proportionally more work. We use highly tuned collective operations (`MPI_Scatterv`, `MPI_Gatherv`, and `MPI_Bcast`) for potentially higher performance [Gor04].

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. We randomly 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

The available machines run under CentOS 5.5 and are local[2], hence remote latency is emulated using the `netem`[3] utility that adds an additional queueing discipline to the operating system kernel and allows to delay packets at the interface. In different setups some of the available machines (as summarised in Table 1) are combined to architectures that range from homogeneous flat architectures to heterogeneous and hierarchical ones.

Table 1: Machines available for combination to different architectural configurations

alias	arch (bit)	PEs	speed (GHz)	L2 cache (MB)	RAM (GB)	cache/RAM
linux_lab	32	2	3.40	2x2	3	1.33
linux01	32	2	2.40	1x2	2	1.00
lxpara	32	8	2.33	2x6	8	1.50
beowulf	64	8	2.00	2x4	6	1.33

[2]using Gigabit Ethernet, except for `linux01` that has a 100Mbit interface
[3]http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

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 `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 nodes of which one is a `beowulf` machine using all of its cores, another is a `lxpara` multi-core 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` are `beowulf` or `lxpara` machines, and the `x2` and `x1` are `linux lab` machines.

4.4 Results and Evaluation

First, we present results for different unshuffled all-local configurations and for shuffled vs unshuffled, all-local configurations. Next, we compare the cost models for the most heterogeneous configuration: shuffled, all-local, `2x4 2x2 4x1` and investigate the impact of latency for a partly remote configuration.

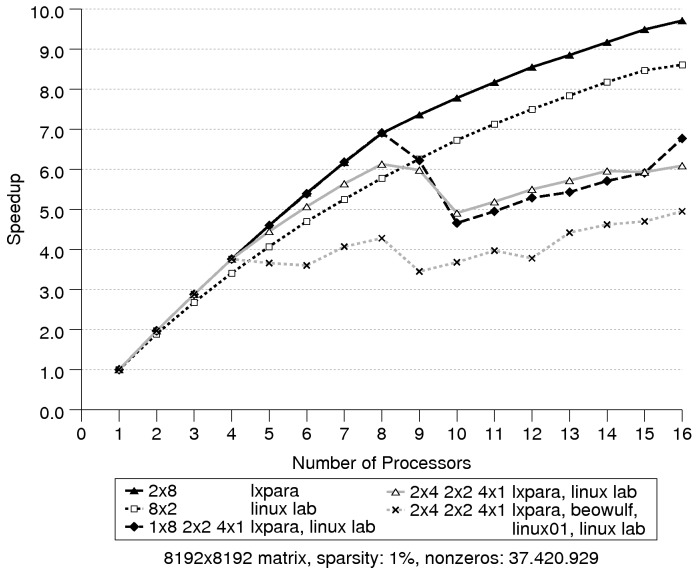

Figure 1: Speedups for unshuffled all-local setups with *varying heterogeneity*

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 PEs, using configurations of varying heterogeneity.

The speedup is almost linear on one multi-core machine and decreases due to communication overhead once we leave the node (e.g. more than 8 PEs for the `2x8 lxpara` setup). Further decrease in performance can be observed for configurations that include machines from a different subnet (`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.

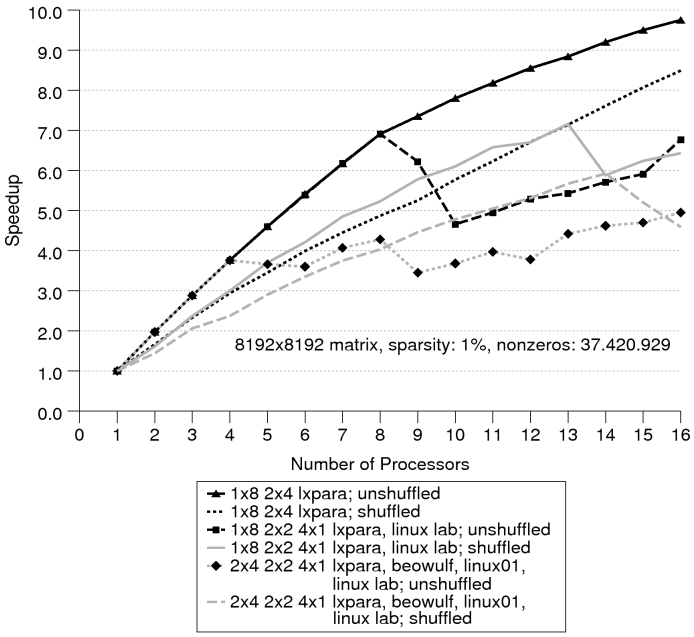

Figure 2: Speedups for selected *unshuffled versus shuffled* all-local setups

All-Local, Heterogeneous and Hierarchical Case: The `2x4 2x2 4x1` configuration in Figure 1 represents the most heterogeneous case, where instances of each available architecture are used (`beowulf`, `lxpara`, `linux lab` and the slow `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 architecture-aware cost model.

Figure 2 depicts the effect of the placement strategy on predictability by showing the graphs for different `unshuffled` (ticked) and `shuffled` (unticked) placement on different configurations. For a small number of PEs it is beneficial to keep the execution local by fully exploiting one multi-core before placing computations on another multi-core.

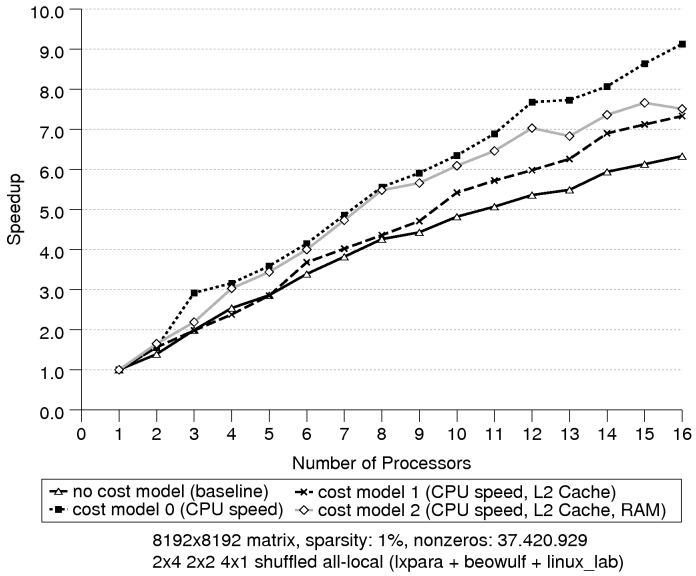

Figure 3: Speedups for the most heterogeneous configuration using *different cost models*

This strategy is tuned to minimise communication cost. However, if we are more concerned with steady and predictable (speedup curve is more smooth) but slightly slower increase in performance on the heterogeneous setups, 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.

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, thus nearly achieving the speedup reached on the homogeneous and flat setup, as observed in the Figure 1 above. With the current settings for the scaling factors, the more advanced cost models do not achieve further improvements in performance, and the simple cost model CM0 performs best. This reflects the intuition that PE speed is the primary performance predictor for computationally intensive applications on all-local setups. Additionally, low variation of the cache-to-RAM ratio leads to the low impact of the respective parameters (cf. last column of Table 1 in Section 4.3). The number of parameters makes the more sophisticated models, which can be tuned to increase performance, more flexible and potentially employable for a wider range of applications. Moreover, application-specific and dynamic system parameters should be taken into account, since they are likely to influence the impact of the architectural parameters.

Partly Remote, Heterogeneous and Hierarchical Case: To investigate the *impact of latency* on 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 models 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 sophisticated cost model that would send off the work only if it would increase the speedup.

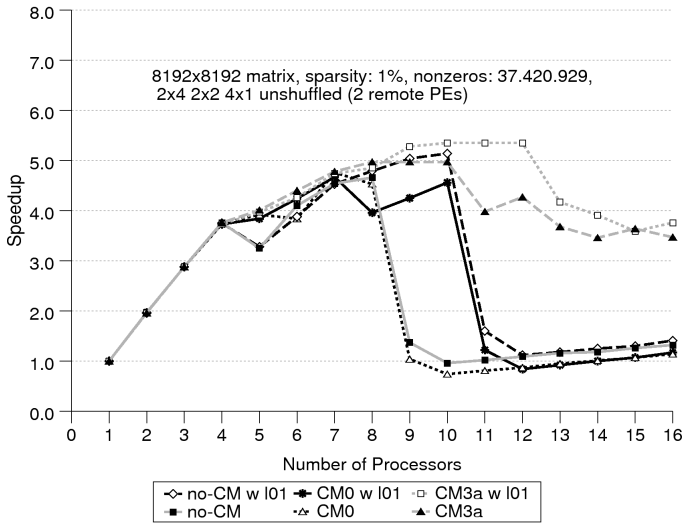

Figure 4: Speedups for the partly remote heterogeneous configuration using *different cost models*

5 Related Work

HeteroMPI [LR06] integrates cost modelling with MPI 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 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 modelling language is required, steepening the initial learning curve. Since HeteroMPI employs a partially static model, it may not be suitable to support applications with irregular parallelism. By contrast, we focus on using an architecture-aware but application-agnostic cost model to improve performance portability.

The design of our cost model is based on results in [AMT11], which achieves good speedups using a simple cost model that does not take into account RAM and latency for two applications[4] expressed in terms of skeletons and run on a heterogeneous architecture. While the focus of that work is on hybrid parallel programming models for algorithmic skeletons, we study the behaviour on a standalone parallel application.

6 Conclusion

This paper investigates the feasibility and performance of a programming model that combines high-level cost modelling with low-level, explicit control of coordination using C and MPI 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 and important class of parallel architectures. We quantify the impact of heterogeneity by comparing the performance of parallel sparse matrix multiplication on a range of clusters of multi-cores: on the most heterogeneous setup with 16 PEs the speedup is merely about half of the speedup on a homogeneous setup. By making decisions on the size and placement of computations using an architecture-aware 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 PEs increases from 6.2 (without any cost model) to 9.1 (using a simple, static cost model), thus nearly matching the performance on a homogeneous architecture. Based on one application, these results on a range of configurations with increasing heterogeneity indicate that using a simple cost model parameterised by PE speed, memory size and cache size, can already achieve a high performance portability. Although measured only for up to 16 PEs, the results indicate that the scalability is improved beyond the available number of PEs on the all-local setups. Since there is no need for monitoring or time-consuming calculations, our static cost model has negligible overhead in the still important homogeneous, flat case. Our results also show that using a shuffled placement strategy, which evenly distributes computations across multi-cores rather than fully exploiting individual multi-cores, provides higher predictability whilst moderately impacting performance. Given the trends towards large-scale many-core architectures, this observation is relevant, considering that with increasing numbers of cores per machine, shared-memory access will become a bottleneck, and therefore exploiting all the available cores on one machine may no longer be an optimal strategy. Additionally, our results affirm that latency is critical for load balancing decisions on partly remote setups.

In summary, our findings suggest that architecture-aware cost modelling facilitates the transition to high-level parallel programming improving performance portability by transferring the responsibility for specific policies controlling parallelism from the programmer to an informed cost model. The need of manual code alterations is eliminated as target architecture is exchanged.

[4] computing the Euler Totient sum and the SIFT image processing algorithm

Limitations: The main limitation of our approach to cost modelling is that we do not incorporate any application-specific information or 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.

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. A further interesting direction is 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 on the more heterogeneous architectures comprising clusters of GPU-enabled multi-cores.

Acknowledgements

We thank Khari Armih, Artur Andrzejak, and Miroslaw Malek for inspiring discussions and the anonymous reviewers for the helpful comments.

References

[AISS95] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP model. In *Proceedings of the 7th ACM Symposium on Parallel Algorithms and Architectures*, pages 95–105, July 1995.

[AMT11] Khari Armih, Greg J. Michaelson, and Phil W. Trinder. Cache Size in a Cost Model for Heterogeneous Skeletons. In *Proceedings of the 5th ACM SIGPLAN Workshop on High-Level Parallel Programming and Applications*, September 2011.

[Asa06] Krste Asanovic et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.

[BGK03] Holger Bischof, Sergei Gorlatch, and Emanuel Kitzelmann. Cost Optimality and Predictability of Parallel Programming with Skeletons. *Parallel Processing Letters*, 13(4):575–587, 2003.

[BP06] Jose Luis Bosque and Luis Pastor. A Parallel Computational Model for Heterogeneous Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 17(12):1390–1400, December 2006.

[CKP+93] D. E. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the 4th ACM Symposium on Principles and Practice of Parallel Programming*, pages 1–12, May 1993.

[Col89] Murray I. Cole. *Algorithmic Skeletons: Structural Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[Dar93] John Darlington et al. Parallel Programming Using Skeleton Functions. In *Proceedings of the 5th International Conference on Parallel Architectures and Languages*, volume 694 of *LNCS*, pages 146–160. Springer, June 1993.

[Fly66] Michael J. Flynn. Very High-Speed Computing Systems. In *Proceedings of the IEEE*, volume 54, pages 1901–1909, December 1966.

[Fos95] Ian Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley, 1995.

[FW78] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the Symposium on the Theory of Computing*, pages 114–118, 1978.

[Gor04] Sergei Gorlatch. Send-Receive Considered Harmful: Myths and Realities of Message Passing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):47–56, January 2004.

[LMR95] Zhiyong Li, Peter H. Mills, and John H. Reif. Models and Resource Metrics for Parallel and Distributed Computation. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 133–143. IEEE Computer Society Press, 1995.

[LR06] Alexey Lastovetsky and Ravi Reddy. HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. *Journal of Parallel and Distributed Computing*, 66(2):197–220, 2006.

[LTB01] H-W. Loidl, P.W. Trinder, and C. Butz. Tuning Task Granularity and Data Locality of Data Parallel GpH Programs. *Parallel Processing Letters*, 11(4), December 2001. Selected papers from HLPP'01 — International Workshop on High-level Parallel Programming and Applications, Orleans, France, 26-27 March, 2001.

[Qui04] M. J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill, 2004.

[Ski94] David B. Skillicorn. *Foundations of Parallel Programming*, volume 6 of *Cambridge International Series on Parallel Computation*. Cambridge University Press, 1994.

[ST98] David B. Skillicorn and Domenico Talia. Models and Languages for Parallel Computation. *ACM Computing Surveys*, 30(2):124–169, June 1998.

[TCH+11] Phil W. Trinder, Murray I. Cole, Kevin Hammond, Hans-Wolfgang Loidl, and Greg J. Michaelson. Resource Analysis for Parallel and Distributed Coordination. *Concurrency: Practice and Experience*, 2011. Accepted for publication.

[THLJ98] Phil W. Trinder, Kevin Hammond, Hans-Wolfgang Loidl, and Simon L. Peyton Jones. Algorithm + Strategy = Parallelism. *J. Func. Prog.*, 8(1):23–60, January 1998.

[Val90] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of ACM*, 33(8):103–111, August 1990.

[Val11] Leslie G. Valiant. A Bridging Model for Multi-Core Computing. *Journal of Computer and System Sciences*, 77(1):154 – 166, 2011.