Moldable Applications on Multi-Core Servers: Active Resource Management instead of Passive Resource Administration

Clemens Grelck

University of Amsterdam
Informatics Institute
System and Network Engineering Lab
Science Park 904
1098XH Amsterdam, Netherlands
`c.grelck@uva.nl`

Abstract. Malleable applications are programs that can, in principle, run with varying numbers of threads and thus on varying numbers of cores of a mult-core parallel system. Malleability is characteristic for many programming models from data-parallel to divide-and-conquer and streaming data flow where the actual amount of concurrency is application and data dependent and varies over time while the runtime system maps the actual concurrency to fixed number of kernel threads / cores.
We argue that such a fixed choice of kernel threads is suboptimal in two scenarios. Firstly, an application may temporarily expose less concurrency than the underlying hardware offers. In this case the cores waste energy. Secondly, the number of hardware cores effectively available to an application may dynamically change in multi-application and/or multi-user environments. This leads to an over-subscription of the available hardware by individual applications, costly time scheduling by the operating system and, as a consequence, to both waste of energy and loss of performance.
We propose an active resource management service developed in the context of the data parallel array language SAC and the streaming macro data flow coordination language S-Net. Both languages are examples of malleable runtime systems where the set of resources could be changed dynamically without affecting the consistency of a running application. A system-wide resource management service controls the computing resources and assigns them to applications on the basis of dynamically changing intra-application requirements as well as on dynamically changing inter-application scenarios in a near-optimal way.

1 Introduction

Malleable applications are programs that can, in principle, run with varying numbers of threads and thus on varying numbers of cores of a mult-core parallel system. Malleability is a characteristic feature of many parallel runtime systems.

For example, in data-parallel applications the number of iterations of a parallelised loop and thus the available concurrency typically exceeds the total number of cores in a system by several if not many orders of magnitude. Consequently, data-parallel applications typically scale down the structurally available concurrency in the application to the actually available concurrency of the execution platform. This is done by applying one of several available loop scheduling techniques, such as block scheduling, cyclic scheduling, block-cyclic scheduling, (guided) self scheduling or data locality aware variations of them.

The same even compiled binary application can in principle and within certain limits run on any number of cores. Typically, however, the number of cores/threads used is provided at application start through a command line parameter or an environment variable and then remains as set throughout the entire application life time. Dynamic malleability is usually not exploited. Common examples of such data-parallel runtime systems are OpenMP[1] or our own functional data-parallel array programming language Single Assignment C [2, 3].

The principle of malleable applications that do not exploit this property dynamically is not at all limited to the data-parallel scenario. In divide-and-conquer style applications written for instance in modern versions of OpenMP[4] using explicit task parallelism or in Cilk[5]. In either case the divide-and-conquer style parallelism, in beneficial scenarios, just like the data parallel approach exposes much higher levels of concurrency than general-purpose multi-core systems can exploit. The solution here in one way or another is to employ a fixed number of worker threads and work stealing techniques to balance the intra-application workload.

As a last example we mention streaming applications as for instance written in the declarative coordination language S-Net [6, 7]. S-Net defines the coordination behaviour of networks of asynchronous, stateless components and their orderly interconnection via typed streams. S-Net achieves a near-complete separation of concerns between the engineering of sequential application building blocks (i.e. *application engineering)* and the composition or orchestration of these building blocks to form a parallel application (i.e. *concurrency engineering).* S-Net effectively implements a macro data flow model where components represent non-trivial computations. Again the level of concurrency is not determined by the S-Net streaming application, but instead by characteristics of individual program runs. The S-Net runtime system [8] effectively maps the available concurrency to a number of threads that is determined upon program startup by the user and then remains fixed throughout the application's runtime.

To summarise, it is common across a wide range of concurrent programming models to expose concurrency to the underlying compiler and runtime system tool chain on a certain level of granularity that typically is finer-grained than what the execution platform effectively offers. Among others this choice is motivated by the fact that in many scenarios the exact properties and characteristics of the to be used execution platform are not known at compile time. The common solution is to map down the finer-grained concurrency exposed by some application to a fixed set of kernel worker threads launched at program startup and mapped to the actual computing resources such as processors (sockets), cores and hyperthreads (hardware execution contexts) by the operating system.

This immediately raises the question as to how many such worker threads to use. In practice, two solutions prevail: either an application determines the actual execution machinery it runs on and launches as many worker threads as it finds hardware execution units (the greedy approach) or an application simply asks the user (the clueless approach). The latter, of course, provides ample opportunity for exprimentation, but in this work we focus on the non-expert user who simply aims at making good use of available resources without extra effort.

We argue that both approaches are undesirable for a number of reasons. As soon as the user of an application is not its programmer at the same time, he or she may not be able to make an educated choice, in particular as the number of cores continues to rise and, thus, the design space. Furthermore, any fixed number of worker threads used throughout a program run is suboptimal for two reasons.

Firstly, we waste energy for operating all computing resources initially chosen as soon as the application effectively exposes less concurrency in certain phases of the program execution. Both the divide-and-conquer as well as the streaming model of parallel program organisation are characterised by ramp-up and fade-out phases of concurrency. In non-trivial applications it is fairly common that such phases occur repeatedly. Even in the data-parallel model non-trivial phases of low-concurrency execution appear in multi-scale method implementations.

Secondly, in typical multi-application or even multi-user environments we cannot expect any single application to have exclusive access to the hardware resources. Consequently, applications compete for resources in an uncontrolled and non-cooperative way as multiple applications start and stop at unpredictable and unforeseeable times. The operating system layer organises the resulting resource oversubscription in a correct but not in an efficient way, as we discuss in the following section.

2 Resource administration vs resource management

The operating system is the canonical layer that administrates computing resources and makes them available to running applications, as illustrated in Fig. 1. However, the operating system does not have any understanding of the internal organisation of concurrent applications. Neither does the operating system know the user's preferences regarding the placement of compute tasks with respect to the hierarchical memory organisation and the resulting opportunities for performance and energy saving.

In an undersubscribed system the operating system could follow essentially one of two policies. Compute tasks could be spread out as much as possible over the system. For instance, four tasks could be run on one core of each processor. As a consequence, each task would benefit from the entire cache memory and the whole memory bandwidth of the socket. Alternatively, the operating system could choose to concentrate all four tasks on a single socket. In this case the

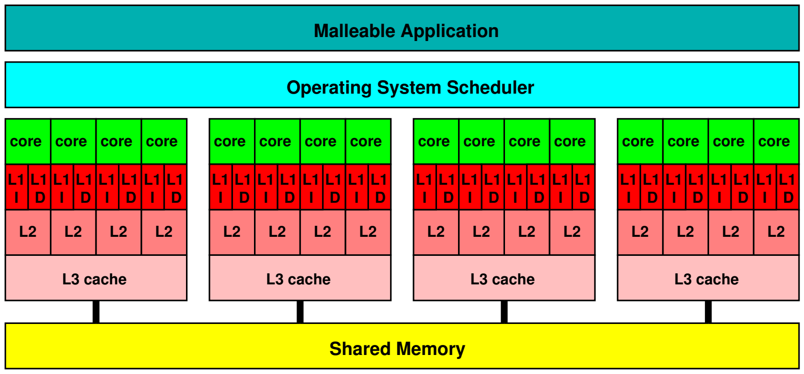

Fig. 1. Architectural model of a cache-coherent shared memory system with four sockets, each equipped with a quad-core processor, and a hierarchical memory organisation with shared L3 caches per processor and individual L2 and L1 (instruction and data) caches for each core

tasks must share the limited cache capacity and memory bandwidth, but on the other hand, the tasks could very efficiently communicate via the shared L3 cache and the other three sockets could be shut down for maximum energy savings in the absence of sufficient useful computing tasks.

In an oversubscribed system the situation changes profoundly. Now, all cores would be busy computing all the time, but in order to ensure fair progress of all tasks, despite the limited computing resources, the operating system resorts to pre-emptive scheduling and time slicing, i.e., an executable task is assigned to a core for execution for a bounded time interval only at whose end it is replaced by another task waiting for execution. This solution stems from the times when uni-processor systems were mimicking parallel systems where multiple interactive applications were supposed to all make progress and remain responsive. In our scenario of compute-bound applications time-slicing has a rather negative effect on performance.

With multiple compute-bound tasks time-slicing mainly causes overhead for stopping one task, saving its execution state and re-installing another task from the ready queue for execution. In addition to executing the necessary instructions we need to switch from user mode execution to kernel mode execution, which is particularly expensive. Moreover, a task over time is typically scheduled to different cores for execution. This has a detrimental effect on data locality as the task's data may still partially by available in core- or socket-local cache elsewhere, but after a context switch needs to be reloaded into a different part of the memory hierarchy.

To avoid costly over-subscription of resources we need runtime systems that specific to a certain concurrent execution model (e.g. streaming, divide-and-conquer or data-parallel) map the concurrency effectively exposed by an application to a fixed set of worker kernel threads as the common software abstraction of shared memory parallel systems. Fig. 2 illustrates the resulting system architecture. In this model the runtime system cooperates with the operating system such that the runtime system makes the educated decisions while it employs the runtime system to actully implement the descisions.

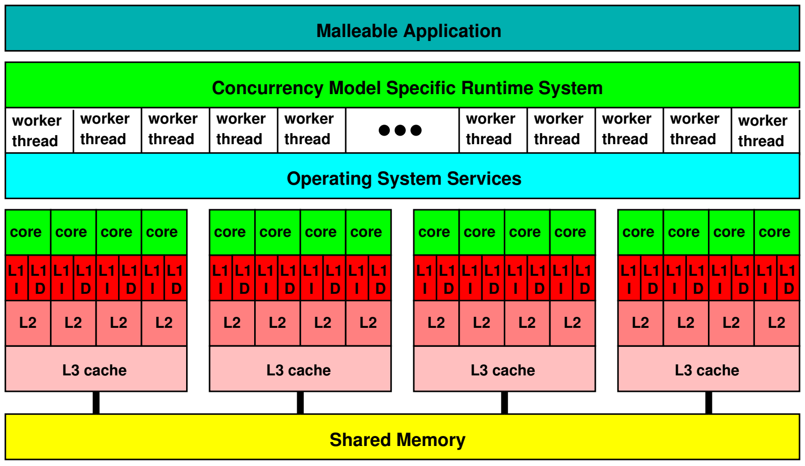

Fig. 2. Architectural model of a cache-coherent shared memory system, as in Fig. 1, with two layers of system software between applications and hardware: operating system and concurrency-model specific runtime system

A *resource management server* dynamically allocates execution resources to a running S-NET program. The (fine-grained) tasks managed by the runtime system are automatically mapped to the dynamically varying number of effectively available kernel threads. Their number is continuously adapted to the effective level of concurrency exposed by the running S-NET streaming network.

In this way, we actively control the energy consumption of a system and reduce the energy footprint of a resource management enabled application compared to greedy resource utilisation, assuming that the underlying operating system automatically reduces the clock frequency and potentially the voltage of underutilised processors and cores or switches them off entirely. Furthermore, we create the means to simultaneously run multiple independent and mutually unaware resource management enabled applications on the same set of resources by continuously negotiating resource distribution proportional to demands.

In contrast to an application-unaware operating system our approach has the advantage that the resource management server understands both sides: the available resources in the computing system **and** the parallel behaviour of the resource management aware running applications. This is why we expect to achieve better performance and less energy consumption compared to today's multi-core operating systems.

3 Managing resource under-subscription

With resource over-subscription effectively solved by a system of worker threads we now come back to a question raised earlier: how many worker threads to actually use in practice. Obviously, using more worker threads than cores leads to resource over-subscription and thus is undesirable. However, as argued earlier any fixed number of worker threads throughout the entire application live time is likely not to be ideal either as soon as applications expose varying levels of exploitable concurrency. If an application at certain times cannot make effective use of all resources, it would be very desirable to either shut down surplus resources for energy saving or, alternatively, make the resources available to other applications.

Active resource management is a runtime system service that dynamically allocates execution resources on demand. A dedicated resource server (thread) is responsible for dynamically spawning and terminating worker threads as well as for binding worker threads to execution resources like processor cores, hyperthreads or hardware thread contexts, depending on the architecture being used.

Upon program startup only the resource server thread is active; this is the master thread of the process. The resource server thread identifies the hardware architecture the process is running on. Next, the resource server sets up the static property graph, which is to be shared by all worker threads. Once the set up is completed, the resource server launches the first worker thread.

Creation (and termination) of worker threads is controlled by the resource service making use of two *resource level indicators*. The first one is the obvious number of currently active worker threads. This is initially zero. The second resource level indicator is a measure of *demand for compute power*. This reflects the number of work queues in the runtime system. The demand indicator is initially set to one. Both resource level indicators are restricted to the range between zero and the total number of hardware execution resources found in the system.

If the demand for computing resources is greater than the number of workers (i.e. the number of currently employed computing resources), the resource server spawns an additional worker thread. Initially, this condition holds trivially. The creation of an additional worker thread temporarily brings the (numerical) demand for resources into an equilibrium with the number of actively used resources. Before increasing the demand the new worker thread must actually find some work to do. Once doing productive work, the worker signals this to the resource server, and the resource server increments the demand level indicator, unless demand (and hence resource use) has already reached the maximum for the given architecture. This procedure guarantees a smooth and efficient organisation of the ramp up phase.

If an application exposes less concurrency work queues of workers may run empty. The worker signals this state to the resource server, which in turn reduces the demand level indicator by one. The worker thread does not immediately terminate because we would like to avoid costly repeated termination and re-creation of worker threads in not uncommon scenarios of oscillating resource demand. The worker thread, however, does effectively terminate with a configurable delay following an extended period of inactivity.

4 Multiple independent applications

The next step in advancing the concept of active resource management is to address multiple independent and mutually unaware applications (or instances thereof) that run at overlapping intervals of time on the same set of execution resources. Fig. 3 illustrates our approach with two applications. We effectively split our resource management service into two parts: a local resource service manages the worker threads within an application, whereas a system-wide resource service is in charge of the computing resources as a whole and effectively mediates these resources between multiple competing applications. This system resource service is started prior to any resource management enabled application process.

Whenever an application has reason to spawn one more worker thread, it first must contact the system resource service to obtain another execution resource. The system resource service either replies with a concrete core identifier or it does not reply at all. In the former case the aplication resource service spawns another worker thread and binds it to the given core. In the latter case the number of execution resources currently occupied by this application remains as is.

Fig. 3 illustrates the simulation of two malleable applications on an 8-core system. For simplicity we ignore any hierarchy in system architecture here. We begin with the start of application 1 on an idle system. Application 1 incrementally allocates all 8 cores via the system resource service. As application 1 apparently exposes sufficient concurrency internally that the application-level resource service actually decides to go this way. At some point application 1 runs concurrently on all 8 cores of the system.

Now we start application 2. Initially, there are no resources whatsoever to run application 2. Thus, application 2 merely requests one core from the system resource service. The system resource service currently has no resources to allocate, but it requests from application 1, more precisely from that application's resource service, to vacate one core. The runtime system of application 1 reacts to this request in an appropriate way and vacates one core at the earliest possible time. Once returned to the system resource service, the latter immediately assigns that core to application 2, which only now effectively begins to run.

Assuming both applications expose ample concurrency, the procedure repeats 3 times until both applications share the 8 cores in a fair way. At times applications run through phases of less concurrency. At some time application 1 deliberately returns a core to the system resource service, which is immediately given to application 2. In a later stage both applications can only make effective use of 3 cores each, and, thus, 2 cores remain empty and could be powered down if the hardware allows.

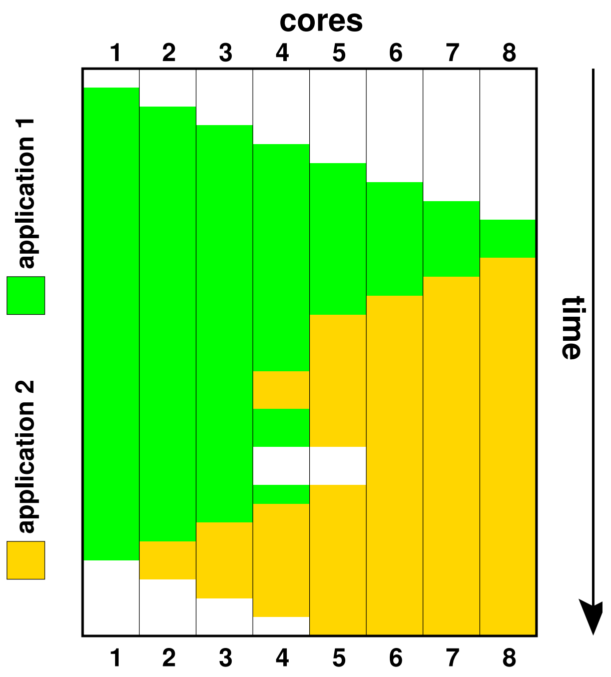

Fig. 3. Simulation of a fictive example with two independent applications

Eventually, application 1 approaches termination and its concurrency fades out. The vacant resources are immediately transferred to application 2 by the system resource service before also application 2 begins to fade out and step-by-step returns cores to the system resource service.

5 Related work

The work closest to our's is the concept of *invasive computing*, advocated by Teich et al [9, 10]. Here, application programs execute a cycle of four steps:

1. explore resources,
2. invade resources,
3. compute,
4. retreat / vacate resources.

Whereas these steps in one way or another can also be found in our proposal, the fundamental difference between their work and our's is the following: Teich et al demand every application to explicitly implement the above steps and provide an API to do so. In contrast, we develop a runtime system that automatically mediates between malleable but otherwise resource-unaware applications and a set of hardware resources that only become known at application start and are typically shared by multiple applications.

Other related work can be found in the general area of operating system process/thread scheduling. Operating systems have long had the ability to map dynamically changing numbers of processes (or kernel threads) to a fixed set of computing resources. However, operating systems do this in an application-agnostic way as they cannot affect the number of processes or threads created. They can merely admister them. As long as the number of processes is less than the number of resources, various mapping policies can be thought of like in our solution. As soon as the number of processes exceeds the number of resources, an operating system resorts to preemptive time slicing.

This all makes sense as long as one takes the resource demands of applications as fixed, but exactly that assumption does not hold for malleable applications. More precisely, malleable applications do have the freedom to adjust resources internally. Trouble is that the application programmer effectively can hardly make use of this opportunity as she or he has no indication of what a good policy could be at application runtime. The operating system, on the other hand, can only react on applications' demands, but not control or affect them in any way. This is exactly where our runtime system support kicks in.

6 Conclusion and future work

We presented active resource management for malleable applications. Instead of running an application on all available resources (or some explicitly defined subset thereof), our runtime system service dynamically adjusts the actually employed resources to the continuously varying demand of the application as well as the continuously varying system-wide demand for resources in the presence of multiple independent applications running on the same system.

Our motivation for this extension is essentially twofold. Firstly, we aim at reducing the energy footprint of streaming applications by shutting down system resources that at times we cannot make effective use of due to limitations in the concurrency exposed. Secondly, we aim at efficiently mediating the available resources among several S-Net streaming applications, that are independent and unaware of each other.

We are currently busy implementing the proposed runtime system techniques within the Front runtime system of S-Net, which is one of many variants of the model runtime system described earlier in the paper. As future work we plan to run extensive experiments demonstrating the positive effect on system-level performance of multiple applications as well as their accumulated energy footprint.

References

1. Dagum, L., Menon, R.: OpenMP: An Industry-Standard API for Shared-Memory Programming. IEEE Transactions on Computational Science and Engineering **5** (1998)

2. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multi-threaded execution. International Journal of Parallel Programming **34** (2006) 383–427
3. Grelck, C.: Shared memory multiprocessor support for functional array processing in SAC. Journal of Functional Programming **15** (2005) 353–401
4. Ayguade, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The Design of OpenMP Tasks. IEEE Transactions on Parallel and Distributed Systems **20** (2009) 404–418
5. Blumofe, R., Joerg, C., Kuszmaul, B., Leiserson, C., Randall, K., Zhou, Y.: Cilk: An Efficient Multithreaded Runtime System. Journal of Parallel and Distributed Computing **37** (1996) 55–69
6. Grelck, C., Scholz, S.B., Shafarenko, A.: A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. Parallel Processing Letters **18** (2008) 221–237
7. Grelck, C., Scholz, S., Shafarenko, A.: Asynchronous Stream Processing with S-Net. International Journal of Parallel Programming **38** (2010) 38–67
8. Gijsbers, B., Grelck, C.: An efficient scalable runtime system for macro data flow processing using s-net. International Journal of Parallel Programming **42** (2014) 988–1011
9. Teich, J., Henkel, J., Herkersdorf, A., Schmitt-Landsiedel, D., Schröder-Preikschat, W., Snelting, G.: Invasive computing: An overview. In Hübner, M., Becker, J., eds.: Multiprocessor System-on-Chip. Springer (2011) 241–268
10. Teich, J., Weichslgartner, A., Oechslein, B., Schröder-Preikschat, W.: Invasive computing — concepts and overheads. In: Forum on Specification and Design Languages (FDL 2012). Number 217–224, IEEE (2012)