Active Resource Management for Multi-Core Runtime Systems Serving Malleable Applications

Clemens GRELCK
System and Network Engineering Lab
University of Amsterdam
Amsterdam, Netherlands

Abstract. Malleable applications are programs that may run with varying numbers of threads and thus on varying numbers of cores because the exact number of threads/cores used is irrelevant for the program logic and typically determined at program startup.

We argue that any fixed choice of kernel threads is suboptimal for both performance and energy consumption. Firstly, an application may temporarily expose less concurrency than the underlying hardware offers, leading to waste of 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 that continuously mediates betwen dynamically changing intra-application requirements as well as on dynamically changing system load characteristics in a near-optimal way.

Keywords. resource management, multicore computing, runtime systems

Introduction

Malleable applications are programs that may run with varying numbers of threads and thus on varying numbers of cores without recompilation. Malleability is characteristic for many programming models from data-parallel to divide-and-conquer and streaming data flow. The actual amount of concurrency is application and data dependent and may vary over time. It is the runtime system's task to map the actual concurrency to a fixed number of kernel threads / cores.

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 or (guided) self scheduling.

The same compiled binary application can within certain limits run on any number of cores. Typically, the number of 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 not exploited. Common examples of such data-parallel runtime systems are OpenMP[4] or our own functional data-parallel array programming language Single Assignment C [9,7].

Malleable applications can also be found in the domain of divide-and-conquer applications, for instance written in modern versions of OpenMP[1] using explicit task parallelism or in Cilk[2]. 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 [10,8]. 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 [5] 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.

All these examples share the common property that a generally high degree of concurrency is mapped down to a much smaller and per application run fixed number of kernel threads and thus execution units. We argue that any fixed number of kernel threads used throughout a program run is suboptimal for two reasons. Firstly, we waste energy for operating all computing resources whenever the application effectively exposes less concurrency than the execution architecture provides. 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. This leads to time slicing in the operating system and thus to suboptimal performance of each application (assuming non-interactive, compute-oriented applications as they prevail in parallel computing). In this work we propose active resource management for malleable applications.

1. Malleable runtime system model

In order to have a wider impact we do not look into individual applications, but rather into common runtime system scenarios for parallel execution. For practical reasons we do not focus our work on any specific programming language, compiler or runtime system, but instead look at a simple, idealised task-oriented, work-stealing runtime system. Our model runtime system consists of a number of kernel worker threads. Each worker thread has a double ended task queue. If the task queue is not empty, a worker thread removes the first task from the head of the queue and executes it. Task execution may lead to the creation of further tasks that are would be added at the end of the local task queue. If a worker thread completes execution of a task, it continues with the following task from the queue. Should the queue become empty, the worker thread switches into work stealing mode and systematically checks other worker thread's task queues for work. If successful, the worker steals the work and continues as above. If the workers fails to find stealable tasks, it waits a while and tries again.

This model runtime system matches a large variety of concrete existing runtime system for a diversity of programming languages and models. While the ways tasks are spawned and synchronised, etc, may be very different, the relevant properties as far as our current work is concerned are surprisingly homogeneous. Further common characteristics are that the runtime system works with any number of worker threads. With a single worker thread we produce overhead due to the internal parallel organisation of the application program without having any any benefits from parallel execution, but the program logic itself is not compromised. Should the number of worker threads exceed the actual concurrency exposed by an application program that does not create sufficiently many independent tasks, worker threads unsuccessfully search for work. Again, we waste resources and produce overhead, but the correct execution of the application is not challenged.

This observation already sets the margins of our current work. Neither too few nor too many worker threads make good use of execution resources and thus do not lead to favourable relationship between computing resources invested and performance obtained. Typically, concrete runtime systems either greedily grab all computing resources provided by the hardware or rely on a fixed number of cores to be assigned at application start via a command line parameter or an environment variable or possibly some user input.

2. Active resource management

We argue that any fixed choice of kernel threads is suboptimal for both performance and energy consumption. Firstly, an application may temporarily expose less concurrency than the underlying hardware offers, leading to waste of 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.

In our proposed solution a *resource management server* dynamically allocates execution resources to a running application program and continuously monitors the adequacy of the resources both with respect to application demands as well as with respect to system load characteristics. The (fine-grained) tasks managed by

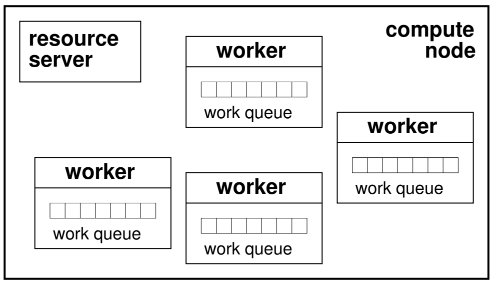

Figure 1. Resource server architecture for malleable runtime systems

the runtime system are automatically mapped to the dynamically varying number of effectively available worker threads (or cores).

In this way, we actively control the energy consumption of a system and reduce the overall energy footprint. Unused computing resources and entire areas of silicon may first run on a reduced clock frequency assuming new work becomes available shortly. After some while of inactivity individual cores and entire sockets may reduce their voltage and could eventually be powered down 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. Resource management server

The *resource management server* is a system service that dynamically allocates execution resources to running programs 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 platform being used. We illustrate our system architecture in Fig. 1.

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 by means of the `hwloc` utility [3,6]. Optionally, the number of cores or generally hardware resources can be restricted by the user. This option is primarily meant as a means for experimentation, not for production use. Next, the resource server initialises the runtime system, creates a work queue, adds the initial task into the queue and turns into a worker thread. We assume here that the application somewhat specifies an initial task. As in the case of S-Net this could likewise mean to read data from standard input or the like.

The worker thread executes its standard work stealing procedure and begins execution of the initial task. Creation (and termination) of worker threads is controlled by the resource server making use of two counters, or better *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 process. Thus, 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 units 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. In general, the new thread could alternatively steal existing work from other threads. In any case, 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.

Worker threads may reach states of unemployment when all local work is exhausted (empty task queue) and no tasks are ready to be stolen from other workers. 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. At first, the kernel thread is put to sleep effectively reducing its resource overhead to near zero, before effectively terminating the thread with a configurable delay following an extended period of inactivity.

4. Multiple independent applications

The next step in advancing the concept of resource management servers is to address multiple independent and mutually unaware applications (or instances thereof) running at overlapping intervals of time on the same set of execution resources. Fig. 2 illustrates our approach with two applications. The role of the resource management server as introduced in the previous section is split into two disjoint parts: a local resource server per application (process) manages the worker threads of the S-Net runtime system and adapts the number and core-binding of the workers as described before.

The second part of multi-application resource management servers lies with a separate process that we coined *meta resource server*. This meta resource server is started prior to any resource management enabled application process. It is in exclusive control of all hardware execution resources of the given system. We deliberately ignore the underlying operating system here as well as potentially running further applications unaware of our resource management model.

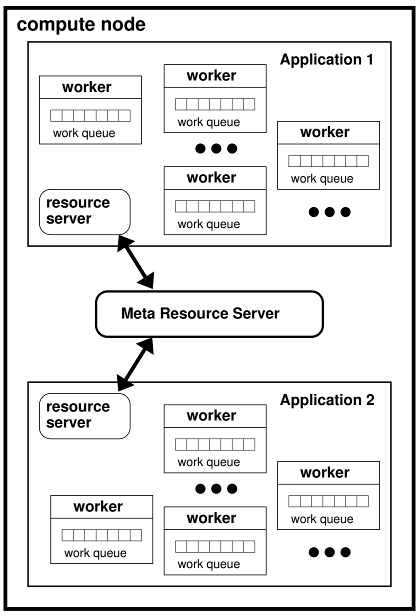

Figure 2. Resource server architecture for multiple independent applications

Whenever a local resource server has reason to spawn another worker thread, in the current multi-application scenario, it first must contact the meta resource server to obtain another execution resource. The meta server either replies with a concrete core identifier or it does not reply at all. In the former case the local resource server of the corresponding application spawns another worker thread and binds it to the given core. In the latter case the local resource server simply does nothing, which means that the number of execution resources currently occupied by this application remains unmodified.

As said before, the meta resource server is in control of all execution resources and decides which application can make use of which cores. With a single application (instance) the system behaves almost exactly as described in the previous section. The local resource server, assuming that the application exposes ample concurrency, incrementally obtains all available resources on the compute node. Only the additional inter-process communication marginally slows down this process.

Let us look at the more interesting scenario of two applications that both expose sufficient concurrency to make use of the entire compute server by themselves. One is started first and obtains one core after the other until it occupies the entire system.

Now, we start the other application. To do this we must first admit that the meta resource server as well as the local resource servers are scheduled preemptively by the operating system. In other words they are not in possession of an exclusive core. And neither are the worker threads. While we guarantee that no two worker threads are bound to the same core at the same time, resource management servers may well interfere with worker execution. With large numbers of cores it may prove more suitable in the future to reserve particular cores for resource management, but the still fairly low core counts representative today, we choose the above solution in order to avoid wasting considerable computing resources. Our general underlying assumption here is that time spent on any form of resource management is negligible compared with the actual computing.

Coming back to our example, all cores are in “exclusive” use by the first application when we start the second application. Hence, we effectively only start the second application’s local resource server, which in turn contacts the meta resource server via inter-process communication to ask for a computing core. Since the meta resource server has no such core at hand, it first needs to get one back from another application. To determine the relative need for computing resources the meta resource server compares two numbers for each application:

a) the number of currently allocated cores;
b) the demand for cores, i.e. how many cores the application has asked for.

The quotient between the latter and the former determines the relative need for cores.

In our running example and assuming an 8-core system, the first application has a demand quotient of $\frac{9}{8}$ because it currently occupies all eight cores but asked for one more core (we assume ample internal concurrency). The second application has a demand quotient of $\frac{1}{0}$ which we interpret as infinitely high. Thus, a new application that has been started but does not yet have any execution resources has a very high relative demand. The meta resource server goes back to the first application and withdraws one the cores previously allocated to it. The local resource server decides which worker thread to terminate and empties that threads work queue, which is simply appended to another work queue. The worker thread is not preemptively terminated but we wait until it finishes its current box computation. After that the worker thread tries to retrieve the next read license from its work queue, but finds its work queue removed. The thread, thus, signals the local resource server its end and terminates. The local resource server immediately communicates the availability of the corresponding core back to the meta resource server. The meta resource server allocated that core to the second application, which now starts to ramp up its execution.

Assuming that the second application likewise exposes ample concurrency, it will soon ask the meta resource server for another threads. The meta resource server, by means of the demand quotients, step-by-step takes execution resources away from the first application and gives them to the second application until an equilibrium is reached. In order to avoid moving resources back and forth uselessly, the meta resource server makes sure that moving one execution resource from one application to another does not invert relative demands.

If the first application terminates at some point in time while the second is still running, all vacated resources will be moved over to the second application.

In our work we focus on long running, compute-intensive applications. Conseqently, adaptations of resources allocated to one or another application are rather infrequent. Furthermore, the number of independent such applications running simultaenously on the same system naturally remains limited. Thus, we do not expect the meta resource server to become anything like a performance bottleneck, despite its central role in the proposed software architecture.

5. Managing Energy Consumption

Intel's Single Chip Cloud Computer (SCC) [12,11] pioneered application-level control of energy consumption by organising its 48 cores into 24 pairs, whose clock frequency can be adjusted, and into 8 voltage islands of 6 cores each, for which the voltage can be adjusted. In other cases, the operating system may adjust voltage and frequency based on load. What the example of the Intel SCC architecture teaches us is that changing the voltage is fairly time-consuming and that voltage and frequency cannot efficiently be controlled on the finest level of execution units, but rather in groups of units of varying size.

We make use of such facilities by creating worker threads step-wise in a demand-driven manner and bind these threads to run on hardware resources as concentrated as possible. For example, on a dual-processor, quad-core, twice hyperthreaded system we would start at most 16 worker threads. While ramping up the number of active worker threads we first fill the hyperthreads of one core, then the cores of one processor, and only when the number of workers exceeds eight, we make use of the second processor. This policy allows the operating system to keep the second processor at the lowest possible clock frequency or even to keep it off completely until we can indeed make efficient use of it.

While we only ramp up the number of worker threads on-demand as computational needs grow, we also reduce the number of workers when computational needs decrease. This fits well with our work stealing based runtime system organisation. If a worker runs out of private work, i.e. its work queue becomes empty, it turns into a thief and tries to obtain work from other workers' work queues. If that also fails, it must be concluded that there is at least currently no useful work to do and the worker terminates. By doing so the worker releases the corresponding hardware resource and, thus, gives the operating system the opportunity to reduce its energy consumption by reducing clock frequency and/or voltage or by shutting it down entirely.

While it is fairly straightforward during worker thread creation to incrementally invade the available hierarchical execution resources, worker thread termination as described above is bound to result in a patchwork distribution of active workers over hardware resources over time. This would render the energy-saving capacities of the operating system largely ineffective. To overcome this shortcoming, the resource server continuously monitors the allocation of worker threads to hardware resources and rebinds the workers as needed.

6. Related work

The work closest to our's is the concept of *invasive computing*, advocated by Teich et al [13,14]. 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 administer 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.

7. 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] E. Ayguade, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(13):404–418, 2009.

[2] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. *Journal of Parallel and Distributed Computing*, 37(1):55–69, 1996.

[3] F. Broquedis, J. Clet Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa Italie, 02 2010.

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

[5] B. Gijsbers and C. Grelck. An efficient scalable runtime system for macro data flow processing using s-net. *International Journal of Parallel Programming*, 42(6):988–1011, 2014.

[6] B. Goglin. Managing the Topology of Heterogeneous Cluster Nodes with Hardware Locality (hwloc). In *International Conference on High Performance Computing & Simulation (HPCS 2014)*, Bologna, Italy, July 2014. IEEE.

[7] C. Grelck. Shared memory multiprocessor support for functional array processing in SAC. *Journal of Functional Programming*, 15(3):353–401, 2005.

[8] C. Grelck, S. Scholz, and A. Shafarenko. Asynchronous Stream Processing with S-Net. *International Journal of Parallel Programming*, 38(1):38–67, 2010.

[9] C. Grelck and S.-B. Scholz. SAC: A functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[10] C. Grelck, S.-B. Scholz, and A. Shafarenko. A Gentle Introduction to S-Net: Typed Stream Processing and Declarative Coordination of Asynchronous Components. *Parallel Processing Letters*, 18(2):221–237, 2008.

[11] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *IEEE International Solid-State Circuits Conference (ISSCC'10), San Francisco, USA*, pages 108–109. IEEE, 2010.

[12] T. Mattson, R. van der Wijngaart, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core scc processor: the programmers view. In *Conference on High Performance Computing Networking, Storage and Analysis (SC'10), New Orleans, USA 2010*. IEEE, 2010.

[13] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. Invasive computing: An overview. In M. Hübner and J. Becker, editors, *Multiprocessor System-on-Chip*, pages 241–268. Springer, 2011.

[14] J. Teich, A. Weichslgartner, B. Oechslein, and W. Schröder-Preikschat. Invasive computing — concepts and overheads. In *Forum on Specification and Design Languages (FDL 2012)*, number 217–224. IEEE, 2012.