Active Resource Management for Declarative Data-Flow Processing

Grelck, C.; Gijsbers, B.

Publication date
2014

Document Version
Final published version

Published in
Preproceedings of the 15th Symposium on Trends in Functional Programming (TFP2014)

Link to publication

Citation for published version (APA):
Grelck, C., & Gijsbers, B. (2014). Active Resource Management for Declarative Data-Flow Processing. In J. Hage (Ed.), *Preproceedings of the 15th Symposium on Trends in Functional Programming (TFP2014)* (pp. [283-292]). Utrecht University, Department of Information and Computing Sciences.

Preproceedings of the 15th Symposium on Trends in Functional Programming (TFP2014)

Jurriaan Hage (editor)

Active Resource Management for Declarative Data-Flow Processing (Extended Abstract)

Clemens Grelck[1] and Bert Gijsbers[2,3]

[1] University of Amsterdam
Amsterdam, Netherlands
c.grelck@uva.nl
[2] Ghent University
Ghent, Belgium
bert.gijsbers@ugent.be

Abstract. S-Net is a declarative asynchronous data-flow coordination language. Like many other high-level multi-core programming approaches, the S-Net runtime system makes use of light-weight task abstractions that are automatically mapped to a set of heavy-weight kernel threads for execution. The number of kernel threads is typically motivated by the number of cores in the hardware. 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-approximation 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 layer for S-Net that effectively overcomes these issues.

1 Introduction

S-Net [1, 2] is a declarative coordination language whose design thoroughly avoids the intertwining of computational and organizational aspects. S-Net achieves a near-complete separation of the concern of writing sequential application building blocks (i.e. *application engineering)* from the concern of composing these building blocks to form a parallel application (i.e. *concurrency engineering).* S-Net defines the coordination behaviour of networks of asynchronous, stateless components and their orderly interconnection via typed streams. We deliberately restrict S-Net to coordination aspects and leave the specification of the concrete operational behaviour of basic components, named *boxes*, to conventional programming languages, primarily C and the purely functional, data-parallel array language SaC [3].

[3] This work was performed while Bert still worked at the University of Amsterdam.

An S-Net box is connected to the outside world by two typed streams, a single input stream and a single output stream. Boxes execute fully asynchronously: as soon as data is available on the input stream, a box may start computing. The operational behaviour is determined by a stream transformer function that maps a single data item from the input stream to a (possibly empty) sequence of data items on the output stream. Boxes are free of internal state to facilitate dynamic reconfiguration, which facilitates dynamic reconfiguration and resource mapping, including elastic box replication. S-Net effectively implements a macro data flow model, *macro* because boxes do not normally represent basic operations but rather individually non-trivial computations.

While the original S-Net runtime system [4] merely served as a proof of concept for the macro data flow approach as such, we recently developed the novel Front runtime system [5] that achieves very competitive runtime performance [6]. Whereas the original proof-of-concept design more or less directly implements the operational semantics of S-Net [7] (i.e. a system of asynchronous components, each implemented by a kernel thread, that communicate via bounded buffers), the Front runtime system employs a fixed number of kernel threads as a software abstraction of the underlying hardware, in practice one thread per core. As a rule of thumb the number of kernel threads used typically equals the number of cores in the system or a deliberately chosen subset thereof.

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).

The contribution of this paper is the extension of the Front runtime system by active resource management. A *resource management server* dynamically allocates execution resources to a running S-Net program. The (fine-grained) tasks managed by the Front 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 an S-Net 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 S-Net applications on the same set of resources by continuously negotiating resource distribution proportional to demands.

2 The FRONT runtime system

The FRONT runtime system[5] is characterised by a fixed number of worker kernel threads that run S-NET components (*boxes*) which are activated by the presence of input data. Instead of costly roaming the dynamic streaming network of asynchronous components on the search for such activated boxes, a FRONT worker thread maintains a private queue of such cases that it continuously processes. Execution of a box yields a stream of output data that the worker thread inserts into its own work queue. When the worker is confronted with an empty work queue, it starts to actively look for work elsewhere. One place to do so is the global input stream of the S-NET streaming network. Another option is to steal work from other worker threads, similar to other work stealing runtimes as pioneered by Cilk [8]. We employ a hierarchical organisation of worker threads in order to accelerate detection of absence of work across all or many workers.

3 Resource management server

The *resource management server* is a system service that dynamically allocates execution resources to running programs, more precisely to S-NET streaming networks of asynchronous components on demand. Whereas the FRONT runtime system originally is based on a configurable but dynamically constant number of kernel threads, we now relax this restriction and make the number of worker threads variable over the entire program runtime. 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. We illustrate our system architecture in Fig. 1.

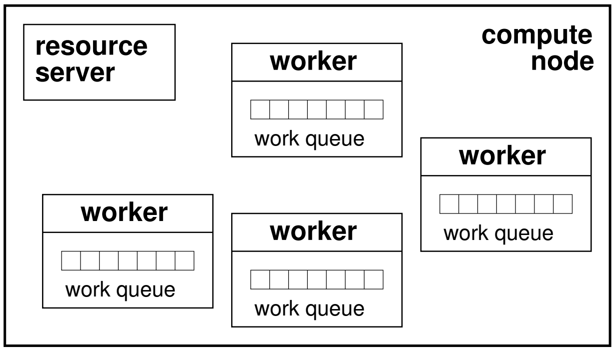

Fig. 1: Resource server architecture for FRONT runtime systems

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. Optionally, the number of cores or hardware threads to be effectively used can be restricted by the user; this is primarily meant as a means for experimentation, not for production use. 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.

The worker thread executes its standard work stealing procedure. In the presence of an obviously empty work queue it reads the global input stream and, thus, creates the first data item in the system. This is afterwards processed as usual in Front, which triggers the creation of further data items to be put into the local work queue.

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 systems. This is not the same as the number of threads because we have a very special further work queue not associated with any of the workers: the global input of the S-Net streaming network. 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 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. In particular during the startup phase of an S-Net streaming network, this usually happens by reading another item from the global input stream. 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. As a standard scenario we assume the availability of considerable input data on the global input stream as well as a non-trivial amount of initial computation on each data item. In this case, we effectively overlap (the overhead of) worker thread creation with reading data from global input and its processing. Moreover, only one worker thread at a time attempts to read the global input stream, which avoids costly synchronisation upon accessing this device.

Executing the Front work stealing model potentially leads worker threads to states of unemployment. With the local work queue being empty, no new data items on the global input and nothing to steal from other workers, there is nothing left to do for a worker thread. 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 Energy consumption

Effective application-level software control over energy consumption parameters such as clock frequency and voltage is still in its infancy. While such features exist on some architectures, e.g. Intel's Single Chip Cloud Computer (SCC) [9, 10], portability in the availability of features and their control are still to come.

As a consequence, we decided for indirect control over energy consumption and anticipate corresponding support in the operating system for automatic clock frequency and potentially voltage scaling. Most commonly used architectures and operating systems do support this today. Originally motivated by the needs of battery-powered devices like laptops and notebooks server installations likewise use these features nowadays to avoid wasting energy when compute power is temporarily unrequested.

We make use of these 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 within the S-Net streaming network, 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 first tries to get hold of the input device and import new data items from the global input stream. If that fails, the worker 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 energysaving 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.

5 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 S-Net-related 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. 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

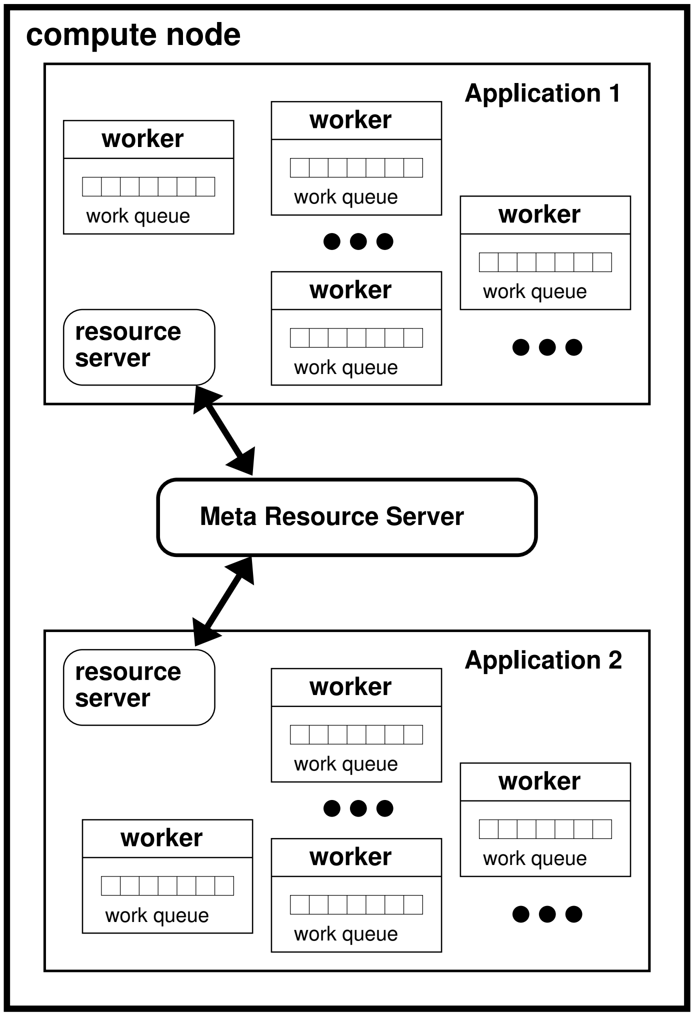

Fig. 2: Resource server architecture for multiple independent applications

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

```
for (i = 0; i < num_clients; ++i) {
  client = all[i];
  if (client->local_workload >= 1) {
    ++num_positives;
    total_load += client->local_workload;
    portions[i] = 1;
  } else portions[i] = 0;
  remains[i] = 0.0;
}
assert(host->nprocs < total_load);
for (i = 0; i < num_clients; ++i) {
  client = all[i];
  if (client->local_workload >= 2) {
    portions[i] += (client->local_workload - 1)
                 * (host->nprocs - num_positives)
                 / (total_load - num_positives);
    remains[i] = ((double) ((client->local_workload - 1)
                          * (host->nprocs - num_positives))
                          / ((double) (total_load - num_positives)))
               - (double) (portions[i] - 1);
  }
  num_assigned += portions[i];
}
while (num_assigned < host->nprocs) {
  p = 0;
  for (i = 1; i < num_clients; ++i) {
    if (remains[i] > remains[p]) p = i;
  }
  if (remains[p] > 0) {
    portions[p] += 1;
    num_assigned += 1;
    remains[p] = 0.0;
  } else break;
}
```

Fig. 3: Algorithm to divide resources between independent applications proportionally to their resource demand

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. Fig. 3 shows an excerpt of the relevant algorithm to ensure proportional resource distribution among applications.

6 Related work

still missing

7 Conclusion and future work

We presented the extension of the Front runtime system for asynchronous stream processing by explicit resource servers. They ensure an active management of execution resources, i.e. processors, cores, hyperthreads of a given system. Instead of running an S-Net streaming network on all available resources (or some explicitly defined subset thereof), we dynamically adjust the actually employed resources to the continuously varying demand of the S-Net streaming application.

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.

In the future we plan to run extensive experiments demonstrating the positive effect on both combined performance of multiple applications and energy footprint.

While the concrete motivation of our work is fueled by S-Net component coordination and asynchronous streaming networks, the general ideas if not even certain parts of the implementation can relatively straightforwardly be carried over to a range of runtime systems for other high-level parallel languages that share with Front the common idea to use a fixed set of kernel threads as an abstraction of the available multi-core hardware, and thus suffer from exactly the same potential shortcomings with respect to energy consumptions and multi-application coordination.

References

1. 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
2. Grelck, C., Scholz, S., Shafarenko, A.: Asynchronous Stream Processing with S-Net. International Journal of Parallel Programming **38** (2010) 38–67
3. Grelck, C., Scholz, S.B.: SAC: A functional array language for efficient multi-threaded execution. International Journal of Parallel Programming **34** (2006) 383–427
4. Grelck, C., Penczek, F.: Implementation Architecture and Multithreaded Runtime System of S-Net. In Scholz, S., Chitil, O., eds.: Implementation and Application of Functional Languages, 20th International Symposium, IFL'08, Hatfield, United Kingdom, Revised Selected Papers. Volume 5836 of Lecture Notes in Computer Science., Springer-Verlag (2011) 60–79
5. Gijsbers, B., Grelck, C.: An efficient scalable runtime system for macro data flow processing using s-net. International Journal of Parallel Programming (2014)
6. Zaichenkov, P., Gijsbers, B., Grelck, C., Tveretina, O., Shafarenko, A.: A case study in coordination programming: Performance evaluation of Concurrent Collections vs. S-Net. In: 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS'14) Workshops, Phoenix, USA, IEEE Computer Society (2014)
7. Penczek, F., Grelck, C., Scholz, S.B.: An Operational Semantics for S-Net. In Chapman, B., Desprez, F., Joubert, G., Lichnewsky, A., Peters, F., Priol, T., eds.: Parallel Computing: From Multicores and GPU's to Petascale. Volume 19 of Advances in Parallel Computing. IOS Press (2010) 467–474
8. Blumofe, R., Leiserson, C.: Scheduling Multi-Threaded Computations by Work Stealing. In: 35th Annual Symposium on Foundations of Computer Science (FOCS'94), Santa F/'e, New Mexico, USA. (1994) 356–368
9. Mattson, T., van der Wijngaart, R., Riepen, M., Lehnig, T., Brett, P., Haas, W., Kennedy, P., Howard, J., Vangal, S., Borkar, N., Ruhl, G., Dighe, S.: The 48-core scc processor: the programmerâĂŹs view. In: Conference on High Performance Computing Networking, Storage and Analysis (SC'10), New Orleans, USA 2010, IEEE (2010)
10. Howard, J., Dighe, S., Hoskote, Y., Vangal, S., 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, IEEE (2010) 108–109