Dynamic X10

Resource-Aware Programming for Higher Efficiency

Matthias Braun Sebastian Buchwald Manuel Mohr Andreas Zwinkau

Karlsruhe Institute of Technology

{matthias.braun,sebastian.buchwald,manuel.mohr,andreas.zwinkau}@kit.edu

Abstract

Static resource allocation, as common on supercomputers, decreases the overall system efficiency because most applications exhibit a varying degree of parallelism. Dynamic exclusive resource allocation can remedy this issue. For an X10 application, this means that the number of places can change dynamically at run-time. In this paper, we propose Dynamic X10, an X10 extension to support a changing number of places. We present an X10 framework for resource-aware programming, where resources are managed explicitly by the programmer. We show the necessary modifications to X10's runtime and standard library as well as an implementation of the proposed framework tailored to tiled many-core architectures. The required adaptions to existing X10 code are evaluated using applications from numerical mathematics.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent programming; D.3.3 [*Software*]: Language Constructs and Features—Concurrent programming structures, frameworks

Keywords X10 Programming Language, Dynamic X10, Invasive Computing, Resource-Aware Programming

1. Introduction

To run a High-Performance Computing (HPC) application on a supercomputer, a user must reserve a time slot with a specific number of compute nodes. Colloquially, the user "stakes a claim" for an application. These claims essentially represent an exclusive and static resource allocation where two applications can never use the same node at the same time. Most applications do not use all their claimed resources at every point during their run-time. For example, an application using multigrid methods needs fewer resources while restricted to coarser grids [2]. However, due to the exclusive and static allocation, unused cores just idle. To improve the efficiency of the system as a whole, the unused resources could be reallocated to other claims [2]. Moreover, with an ever growing number of cores on a single chip, proposed many-core architectures share the characteristics and thus also the problems of supercomputers.

On shared memory systems, e.g., within a compute node, CPU cores are allocated neither exclusively nor statically. Instead, the operating system (OS) offers virtualized cores called threads to the programmer. Threads are transparently multiplexed onto the physical cores, so programmers need not care for their number. However, this implies overhead due to context switching and cache thrashing, especially when there are more threads than cores. Additionally, many HPC algorithms rely on barrier synchronization, so if one thread is interrupted the other threads cannot make progress. Exclusive core allocation can improve efficiency in such cases [4]. This could be realized by using supercomputer concepts in the small, which leads to the same problems as described above. Instead, the OS should dynamically reallocate cores exclusively to applications or claims. This applies to both distributed supercomputers and shared memory systems.

Dynamic exclusive core allocation can be performed within an application using the standard Linux API, e.g., through thread pinning [9] or cpusets [3]. However, an application-internal approach cannot optimize well in a multi-application scenario. An application lacks information about other claims and global resource management is therefore the task of the OS. However, current OSs do not have enough application-specific information. For example, the number of runnable threads is not enough information to partition cores between two applications, as nothing about their scaling behavior is known. While embarassingly parallel applications can use an unlimited amount of cores, most applications exhibit a logarithmic scalability curve. Necessarily, we need an API for application programmers to provide such information to the OS.

If claims are a concept to represent a set of resources, they implicitly are a resource isolation mechanism. So if claims isolate resources, this poses the question of how claims interact if multiple claims exist within the same application. Another problem with programming with claims is that it requires the application to adapt to changing resources. For example, data must be redistributed whenever compute nodes and their local memory are added to or removed from a claim. Unless the application does not rely on persistent distributed data, it also follows that claims cannot change at arbitrary points in time. There are moments where a data redistribution is possible, and application-specific synchronization is necessary to exploit them.

For an X10 application, the resource-changing environment manifests as places appearing and disappearing. Therefore, it breaks one of X10's principles from the language specification [8]: *"The set of places available to a computation is determined at the time that the program is started, and remains fixed through the run of the program."* This has implications on various concepts. For example, how should `DistArray`s behave when a place disappears? What are the semantics of a place belonging to another claim? To support resource-aware programming in X10, some changes to the language semantics and standard library are necessary.

In this work we present:

- An X10 framework for programming with claims and communicating resource requirements to the OS.
- An implementation of this framework on a custom OS, which provides a suitable resource management interface. We especially discuss necessary changes to the X10 language semantics, runtime system, and standard library.
- An evaluation of the application changes for dynamic exclusive allocation on a tiled many-core architecture, which has similar characteristics as supercomputers.

We present the X10 framework in section 2 and the OS and architecture we implemented it on in section 3. Finally, section 4 includes case studies that show how existing X10 programs have to be modified in order to use our proposed framework.

2. Claim Framework

In this section, we will describe how claims can be integrated into the existing X10 language. The framework [11, 12] was developed within the Invasive Computing project [10], which is driven by the question of how heterogeneous many-core architectures can scale to hundreds or thousands of cores on a single chip. Invasive Computing considers many resources types like communication links [5] or memory. However, for the purpose of this section, we will limit ourselves to processor cores as computing resources. We will first present the integration of claims on a higher level and then give a more detailed technical explanation of the impact of claims on X10 and its runtime library.

2.1 Basic Claim Handling

A claim is a set of processor cores. At every point in time, all existing claims form a partition of the set of cores in a system, i.e., each core belongs to exactly one claim. Cores that are not contained in a claim belonging to a running application are part of a system claim that serves as a pool for currently available cores. Every non-empty subset of the set of cores is a valid claim. Hence, claims are allowed to span multiple shared-memory domains, i.e., nodes in a supercomputer.

From the viewpoint of the application running inside a claim, the usual X10 semantics hold. However, the application's view is restricted to the resources that are contained in the claim. Thus, if a claim consists of three cores belonging to a shared-memory domain, the application only sees a single place. Similarly, if a claim consists of one core on each of the supercomputer nodes, the X10 program sees as many places as there are nodes.

Hence, traditional X10 can be embedded naturally into X10 with claims: A traditional X10 program behaves exactly like an X10 program that runs inside a claim containing all cores in the system. Both the operating system and the X10 runtime system must be claim-aware, so that activities created by an X10 application are only executed on the cores belonging to the application's claim (see subsection 3.2).

Each application runs inside its own claim, initially consisting of one core. Should the need for more resources arise, one way to acquire more resources is to enlarge the existing claim. We call the function to modify an existing claim *reinvade*. Hence, `reinvade()` marks program points where the application's view of the resources can change: if the modified claim contains a core on a shared-memory domain that previously was not part of the claim, a new place becomes visible. If a whole shared-memory domain is removed from the claim, its associated place disappears, too. For example, a multigrid application can use `reinvade()` whenever the grid is made coarser (in the restrict step) or finer (in the interpolate step) as shown in Figure 1.

2.2 Multiple Claims and Isolation Boundaries

Instead of modifying existing claims, it is also possible to acquire more resources by creating entirely new claims. Having separate claims inside one application can be useful, because claims provide *isolation* with respect to resource changes. For example, suppose that an application needs to execute two compute-intensive parallel tasks. Both tasks are implemented in a resource-aware manner and call `reinvade()` to inform the system of their resource usage. If both tasks operated on the same claim, this would cause multiple problems. First, concurrent updates of the same claim make reasoning about the code very hard. From the perspective of one task, resource changes can now not only happen on its own `reinvade()` calls, but at arbitrary program points because of a concurrent `reinvade()` call by the other task. And second, the type of resources needed by the two tasks might be different, so contradicting `reinvade()` calls might be executed.

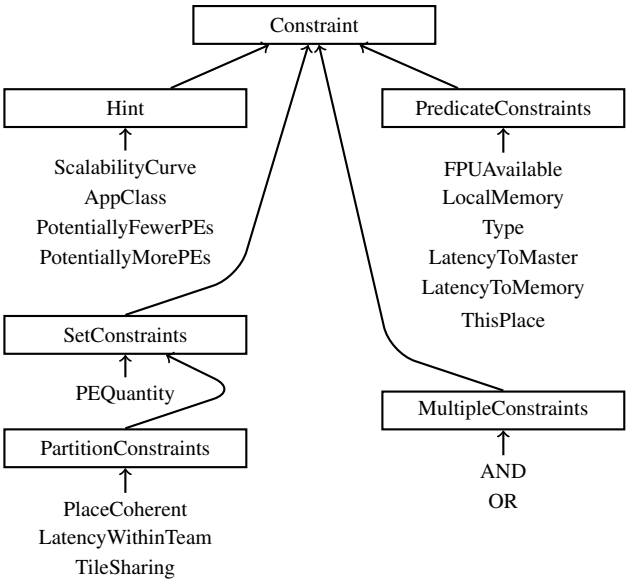

Figure 2. Constraint hierarchy for describing resources: the boxes represent abstract base classes, an arrow represents inheritance in classic object-oriented fashion.

We call the function that creates and returns a new claim *invade*. For the X10 programmer, a claim is an X10 object of type `Claim`. The class `Claim` offers the static method `invade()` for constructing new claim objects, as well as non-static methods `reinvade()` and `retreat()` for modifying and destroying exisiting claims. Hence, `invade()` acquires new resources and returns them as an object of type `Claim`. Conversely, `retreat()` releases the associated resources of a claim.

Additionally, `Claim` has a non-static method *infect*. By calling `infect()` on a claim object C, the programmer can cross claim boundaries and execute an activity on C's resources. Thus, the `infect()` method behaves analogously to X10's `at` construct but switches between claims instead of places. It takes a closure as an argument and has the same deep copy semantics as `at`. `infect()` executes the closure as a new activity on one core inside the new claim. Upon execution, the activity runs in the context of the new claim and thus only sees the resources that belong to this claim. It can then use X10's `at` and `async` constructs to switch places inside the claim or start new activities, respectively. A call to `infect()` blocks until execution in the other claim terminates locally.

2.3 Resource Awareness

As mentioned before, the OS needs application-specific information to globally optimize resource usage for maximum efficiency in a scenario with multiple applications. Hence, both `invade()` and `reinvade()` take a description of the requested resources as an argument. We call these resource descriptions *constraints* and model them as a class hierarchy in X10, where all constraints inherit from an abstract base class `Constraint`. The class hierarchy is shown in Figure 2.

The most simple and at the same most important constraint is `PEQuantity`, which specifies a range for the number of "process-

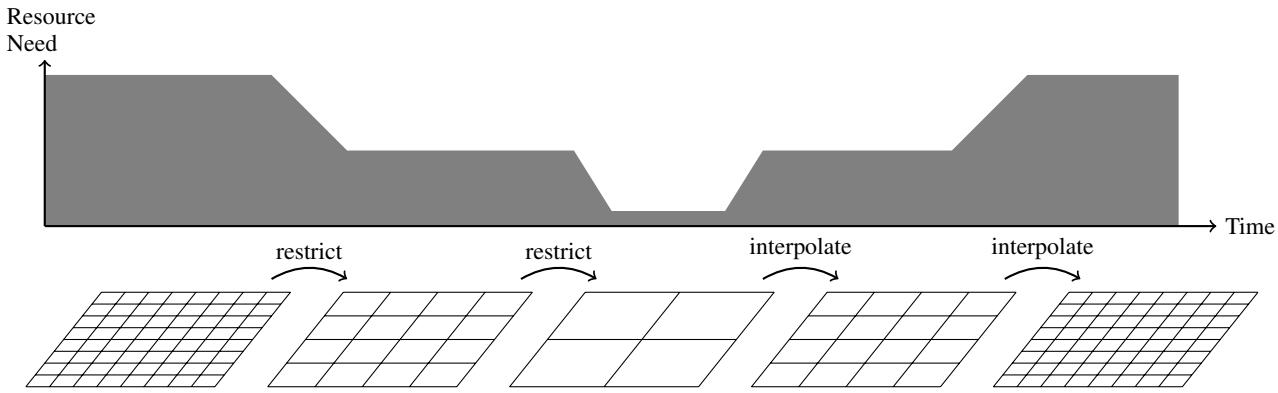

Figure 1. The multigrid approach means that the grid is restricted to coarser versions and then interpolated again to the finer variants. Processing the coarse version takes fewer resources, so during one v-cycle resources are free for other applications. Each simulation step consists of at least one such cycle.

ing elements", i.e., processor cores. Several other constraints are available to further describe resources. For example, `LocalMemory` requires a specific amount of local memory to be available per core. Multiple constraints can be combined using overloaded operators `&&` and `||` for complex constraints.

From a resource management perspective, the most interesting constraint is `ScalabilityCurve`. With this constraint, the programmer can pass application-specific knowledge about the application's scaling behavior to the system. The scaling behavior is described as approximated speedup depending on the number of cores in the claim. As shown in Figure 3, the basic constructor of `ScalabilityCurve` takes a list of speedup values (in percent), where the list index (starting at 1) is the number of processor cores. One way to obtain a scalability curve is by running experiments on the target platform and hard-code the measured speedup values. An alternative is to leverage monitoring infrastructure in order to tune the speedup values at runtime. Combinations of both approaches are also possible.

```
val c = new PEQuantity(1, 3) &&
        new ScalabilityCurve([100, 130, 150]);
val claim  = Claim.invade(c);
// ... perform work ...
claim.retreat();
```

Figure 3. Example use of `invade()` and resource constraints.

In the case of multiple concurrent resource requests, the system can exploit the information passed via constraints to decide which resource distribution is optimal for global efficiency. For example, if two applications request an additional core via `reinvade()`, the system can look at their scaling behavior and give the core to the application that benefits most. Moreover, constraints enable the system to choose the most suitable cores on a heterogeneous architecture. So a compute-bound simulation gets high-performance cores, while a memory-bound sorting application gets slower cores instead.

With the inclusion of logical disjunction and range-based constraints like `PEQuantity`, there are multiple possible outcomes of a resource query. Hence, upon return from `invade()` or `reinvade()`, the programmer has to inspect the returned claim and find out which resources were granted by the system. If the system could not fulfill the resource request at all, `invade()` and `reinvade()` throw a `NotEnoughResources` exception.

2.4 Necessary X10 Runtime Adaptations

To support a dynamically changing number of places, the X10 runtime library has to be adapted at various points. First, all static fields that encode information about the number of places have to be removed and replaced with property functions. Examples include `Place.MAX_PLACES` and `PlaceGroup.WORLD`, which must both query the application's current claim. Second, also some method implementations, like `Place.places()` to retrieve a sequence of places, have to be changed. Overall, only few explicit changes are necessary and all of them are trivial like the presented examples. More fragile is source code which implicitly assumes a constant number of places.

An important issue is the interaction of X10's `DistArray`s with `reinvade()`. As `reinvade()` can lead to new places appearing and also possibly disappearing, it is initially not clear how this affects a `DistArray` that was created before `reinvade()` was called. If a new place appears, existing `DistArray`s can continue to function normally, as their associated distributions do not map any `Point`s to the new place. If a program wants to make use of new places, it has to redistribute its data. Hence, the programmer must create a new `DistArray` with an updated distribution and move the data to the corresponding places. Further details on data redistribution are described in subsection 2.5.

Handling disappearing places is more difficult. The application can find out that a place disappeared only *after* the call to `reinvade()`. At this point, it has already lost access to its data on this place and cannot move the data over to another place anymore. One solution to this is to forbid the loss of places: once a claim contains a core on a shared-memory domain D, the system guarantees that the claim will continue to contain at least one core from D after all subsequent calls to `reinvade()`. We call this property *sticky* claims. If the number of cores per shared-memory domain is high and the maximum number of concurrent applications (and therefore claims) is low, sticky claims can be a viable solution. The advantage of sticky claims is that the programmer never has to deal with disappearing places. However, their downside is that they severely restrict the reallocation of resources, therefore possibly decreasing efficiency.

The alternative is to give the application the opportunity to redistribute its data before actually removing the cores from the claim. One possibility to realize this is to split `reinvade()` into a request step and a confirmation step. After the request step, the application's claim contains both the old and the new cores. With additional information about which cores the system wants to

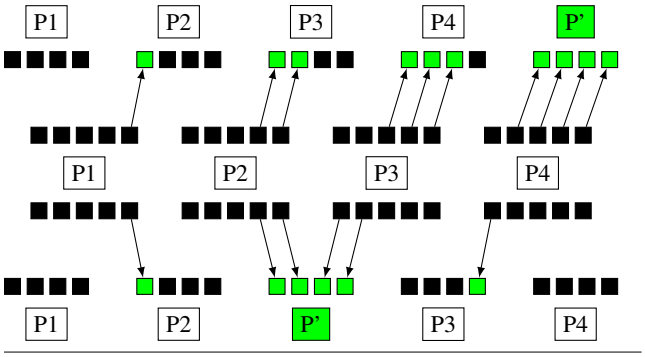

Figure 4. Redistribution of a `DistArray` after a new place P' was added. White boxes represent places, black boxes represent `DistArray` elements. Elements highlighted in green are transferred between places.

remove from the claim, the application can now redistribute its data if necessary. After the redistribution, the confirmation step signals the system that it is now safe to actually update the claim. Another way of giving the application the chance to redistribute data is by passing an X10 closure as an additional argument to `reinvade()`. The closure is called if the system decides to remove cores that would result in at least one place disappearing. Upon invocation, the closure is supplied the necessary information about the old and the new claim and can redistribute the data accordingly. To be able to do this, the programmer has to capture references to all relevant `DistArray`s in the closure body.

2.5 Communication Reduction on Data Redistribution

If the number of places changes, data saved in existing `DistArray`s may need to be redistributed. In case a new place appears, data needs to be moved to this place in order to exploit its additional processing power. Correspondingly, before a place is removed its data must be distributed to other places. As this process is generic and mechanical, it makes sense to offer it as part of a library instead of replicating the code in each application. Moreover, because communication is potentially expensive, in the following we will present ways to reduce the amount of necessary communication.

We will focus on new places appearing, but the dual case of disappearing places can be handled similarly, where the direction of all communication operations is reversed. Figure 4 shows a simple example of a `DistArray` with a `BlockDist` distribution, initially distributed over 4 places. If a new place appears, the position of the new place relative to the existing places is not fixed yet. It might make sense to choose the mapping from place ids to physical nodes in a cluster so that neighboring nodes also have consecutive place ids. However, in general, the new place can be assigned an arbitrary id in the allowed range. As we will show in the following, it does make a difference in terms of communication overhead which place id is chosen.

We will assume that all communication operations have equal cost, independent of source and destination place, and we will only focus on the number of `DistArray` elements that have to be transferred. Let p be the number of places before a new place is added and m the number of `DistArray` elements per place. For ease of presentation, let us assume that m is also divisible by $p+1$, so all array elements mp can also be evenly distributed over $p+1$ places. The amount of data that has to be moved per place is $d := m - \frac{mp}{p+1}$. If the new place is added at the end of the place list, as shown in the upper half of Figure 4, the first place has to copy one element, the second two elements, etc. Thus, the total communication costs are

$$c = d \cdot \Sigma_{i=1}^{p} i = d\frac{p(p+1)}{2} = d\frac{p^2+p}{2}.$$

As the number of places p approaches infinity, c can be approximated by the term $\frac{dp^2}{2}$.

However, if the new place is added in the middle as shown in the lower half of Figure 4, the new place receives half its data from the left $\frac{p}{2}$ places and half its data from the right $\frac{p}{2}$ places (assuming p is even). Following the same reasoning as above, we find that

$$\frac{c'}{2} = d \cdot \Sigma_{i=1}^{\frac{p}{2}} i = d\frac{\frac{p}{2}(\frac{p}{2}+1)}{2}.$$

Hence, $c' = d\frac{p^2+2p}{4}$, which approaches $\frac{dp^2}{4}$ as p approaches infinity. This shows that asymptotically, inserting the new place in the middle can save up to 50% of communication operations.

3. Hardware Architecture and Operating System

This framework was developed within the Invasive Computing Project [10] for a specific OS [7] and hardware architecture [6]. Invasive Computing is driven by the question of how many-core architectures can scale to hundreds or thousands of cores on a single chip and how these chips can be programmed. For scalability reasons, invasive computing looks at *tiled* many-core architectures.

3.1 Hardware Architecture

Tiled many-core architectures are a way to arrange a large number of cores on a single chip. They provide a traditional multi-core architecture within a tile and a simple scalable interconnection between tiles.

Figure 5 shows an example of a tiled many-core architecture. The building block of such an architecture is a *tile*. In our prototype hardware, a tile is a group of 4 processor cores that share some resources, like an L2 cache or possibly a small tile-local memory. Most importantly, cache coherence is guaranteed between the per-processor L1 caches inside a tile. Hence, from a programmer's point of view, a single tile behaves exactly like a common shared-memory multiprocessor system. Multiple such tiles can be combined to create a tiled many-core architecture, where the tiles are connected via a network-on-chip. The tiles are also allowed to be heterogeneous and can, for example, contain memory instead of processor cores.

It is long known that cache coherence protocols do not scale beyond a certain number of participants. Therefore, in such a tiled many-core architecture, *no* cache coherence is provided between different tiles. This makes partitioning the global memory and thus the PGAS model the most viable method to program the system. Hence, X10 is a perfect fit and maps naturally to these architectures: each tile is presented as a separate place to the programmer. As X10 semantics demand that data must be serialized and deserialized when switching places via `at`, the missing cache coherence across tile boundaries does not cause problems.

3.2 Operating System

The operating system used within Invasive Computing, Invasive Run-time Support System (iRTSS), promotes the claim to a central data structure of the OS. Hence, X10 claim objects are just thin wrappers of OS claims, which fully support the operations `invade()`, `infect()` and `retreat()`. For a discussion of the overhead of these operations, see [7]. The iRTSS scheduler is claim-aware and takes care that newly created activities are only executed on cores that belong to the respective claim. Furthermore, its thread model is designed to provide very lightweight threads. Hence, X10's activities can be mapped directly to OS threads, without any need for a user-level scheduler as part of the X10 runtime

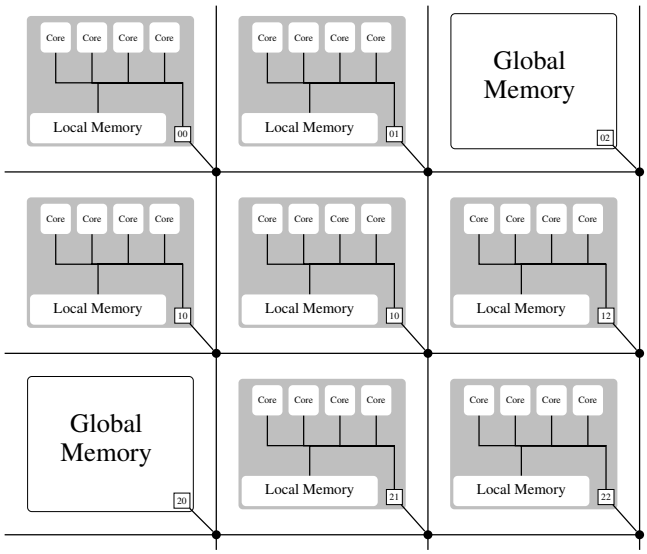

Figure 5. An example of a tiled many-core architecture. The shown example has 9 tiles, arranged in a 3×3 mesh and connected by a network-on-chip. 7 of the tiles contain processor cores and 2 tiles contain external memory.

library. Additionally, the iRTSS includes a resource manager that can make use of the resource constraints provided by applications.

4. Evaluation

We have adapted two existing X10 applications, a heat simulation and a numerical integration application, to the resource-aware programming style presented in this paper. Both applications exhibit a varying degree of parallelism that depends on the input data, making it natural to use `reinvade()` to inform the system of the application's current resource needs. In this section, we will focus on software engineering aspects and evaluate the necessary modifications to the applications. Hence, we will not present runtime measurements; see [1, 2] for discussions of performance and efficiency.

4.1 Heat Simulation

This application simulates a laser engraving text onto a metal plate and computes the heat distribution on the plate using a multigrid solver. It comprises about 2500 lines of code and is described in more detail in [2]. Compared to the normal version, the version using Dynamic X10 required two changes. The first change is to add `reinvade()` calls during restriction and interpolation steps (see Figure 1). The second change is to redistribute the data according to changed resources.

If `reinvade()` calls are added to an application, this requires the programmer to consider the constraints to use. Our application is not optimized for any specific hardware, so the constraints are essentially the desired number of cores and the scalability curve. The desired number of cores is computed from the dimensions of the grid that the solver uses, i.e., more cores for a fine grid and fewer cores for a coarse grid. The scalability curve was initially guessed as slightly sub-linear and will be refined according to actual measurements. As the multigrid approach itself determines the points of changing resource needs, choosing the correct program points to insert `reinvade()` calls was straightforward. There was no need for additional resource isolation inside the application, thus besides resizing the initial claim, no additional claims are created.

After each `reinvade()`, the grid data, which is held in `DistArrays`, must potentially be redistributed. In our prototype, we used sticky claims, so the code did not have to cope with disappearing places. We plan to support place removal in our framework with one of the alternatives presented in subsection 2.4 and plan to investigate the necessary programming effort. The actual redistribution code was written specifically for this application. However, it is planned to generalize it and move into the framework as there is nothing application-specific about the code. All in all, modifying the multigrid application to support Dynamic X10 required less than 50 lines of additional code.

4.2 Numerical Integration

This application approximates a definite integral of a given function. The integration range is split into smaller intervals, which can be viewed as "jobs" that can be processed independently. Depending on the behavior of the function, a varying number of jobs is created and thus also the size of the claim is adjusted dynamically via `reinvade()`. Since a job can be described using two `Double` values and the result of a job is another `Double` value, there is no need for using `DistArrays` and data redistribution. However, from the viewpoint of global resource management, this application is interesting because it is both purely compute-bound and embarassingly parallel. Hence, it can serve to "fill gaps" in case resources would be otherwise unused.

5. Conclusion

In this paper, we showed that a resource-aware programming style using dynamic exclusive core allocation is feasible in X10. We introduced the concept of claims as sets of exclusively granted computing resources and presented an X10 framework for programming with claims. To cope with changing claims during run-time, we extended X10 to support a dynamically changing number of places and presented the necessary modifications to runtime and standard library. We argued that optimizing the global resource distribution is only possible with application-specific information and showed how this information can be transported and exploited in our framework. We presented one implementation of our framework tailored to tiled many-core architectures. An evaluation using suitable HPC applications showed that Dynamic X10 is practical and that it has moderate impact on programming effort.

Acknowledgments

This work was supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Invasive Computing" (SFB/TR 89).

References

[1] M. Bader, H.-J. Bungartz, and M. Schreiber. Invasive computing on high performance shared memory systems. In R. Keller, D. Kramer, and J.-P. Weiss, editors, *Facing the Multicore-Challenge III*, volume 7686 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-35892-0. . URL `http://dx.doi.org/10.1007/978-3-642-35893-7_1`.

[2] H.-J. Bungartz, C. Riesinger, M. Schreiber, G. Snelting, and A. Zwinkau. Invasive computing in HPC with X10. In *Proceedings of the third ACM SIGPLAN X10 Workshop*, X10 '13, pages 12–19, New York, NY, USA, 2013. ACM. . URL `http://doi.acm.org/10.1145/2481268.2481274`.

[3] S. Derr, P. Jackson, C. Lameter, P. Menage, and H. Seto. Cpusets, 2004. URL `https://www.kernel.org/doc/Documentation/cgroups/cpusets.txt`.

[4] P. Flick, P. Sanders, and J. Speck. Malleable sorting. *IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 418–426, May 2013.

[5] J. Heisswolf, A. Zaib, A. Zwinkau, S. Kobbe, A. Weichslgartner, J. Teich, J. Henkel, G. Snelting, A. Herkersdorf, and J. Becker. CAP: Communication Aware Programming. In *Design Automation Conference (DAC), 2014 51th ACM / EDAC / IEEE*, 2014. Accepted.

[6] J. Henkel, A. Herkersdorf, L. Bauer, T. Wild, M. Hubner, R. Pujari, A. Grudnitsky, J. Heisswolf, A. Zaib, B. Vogel, V. Lari, and S. Kobbe. Invasive manycore architectures. In *17th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 193–200, Jan 2012. .

[7] B. Oechslein, J. Schedel, J. Kleinöder, L. Bauer, J. Henkel, D. Lohmann, and W. Schröder-Preikschat. OctoPOS: A Parallel Operating System for Invasive Computing. In R. McIlroy, J. Sventek, T. Harris, and T. Roscoe, editors, *Proceedings of the International Workshop on Systems for Future Multi-Core Architectures (SFMA'11)*, volume USB Proceedings, pages 9–14, 2011. URL `http://www4.cs.fau.de/~benjamin/documents/octopos_sfma2011.pdf`.

[8] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. X10 language specification. Technical report, IBM, February 2014. URL `http://x10.sourceforge.net/documentation/languagespec/x10-latest.pdf`.

[9] O. Tardieu, B. Herta, D. Cunningham, D. Grove, P. Kambadur, V. Saraswat, A. Shinnar, M. Takeuchi, and M. Vaziri. X10 and APGAS at petascale. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 53–66, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2656-8. . URL `http://doi.acm.org/10.1145/2555243.2555245`.

[10] 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 – Hardware Design and Tool Integration*, pages 241–268. Springer, Berlin, Heidelberg, 2011.

[11] A. Zwinkau. Resource awareness for efficiency in high-level programming languages. Technical Report 12, Karlsruhe Institute of Technology, 2012. URL `http://digbib.ubka.uni-karlsruhe.de/volltexte/1000028712`.

[12] A. Zwinkau, S. Buchwald, and G. Snelting. Invadex10 documentation v0.5. Technical Report 7, Karlsruhe Institute of Technology, 2013. URL `http://pp.info.uni-karlsruhe.de/~zwinkau/invadeX10-0.5/manual.pdf`.