Invasive Computing in HPC with X10

Hans-Joachim Bungartz, Christoph Riesinger, Martin Schreiber
Technische Universität München
{bungartz,riesinge,schreibm}@in.tum.de

Gregor Snelting, Andreas Zwinkau
Karlsruhe Institute of Technology
{snelting,zwinkau}@kit.edu

Abstract

High performance computing with thousands of cores relies on distributed memory due to memory consistency reasons. The resource management on such systems usually relies on static assignment of resources at the start of each application. Such a static scheduling is incapable of starting applications with required resources being used by others since a reduction of resources assigned to applications without stopping them is not possible. This lack of dynamic adaptive scheduling leads to idling resources until the remaining amount of requested resources gets available. Additionally, applications with changing resource requirements lead to idling or less efficiently used resources. The invasive computing paradigm suggests dynamic resource scheduling and applications able to dynamically adapt to changing resource requirements.

As a case study, we developed an invasive resource manager as well as a multigrid with dynamically changing resource demands. Such a multigrid has changing scalability behavior during its execution and requires data migration upon reallocation due to distributed memory systems.

To counteract the additional complexity introduced by the additional interfaces, e. g. for data migration, we use the X10 programming language for improved programmability. Our results show improved application throughput and the dynamic adaptivity. In addition, we show our extension for the distributed arrays of X10 to support data migration.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming; D.3.3 [*Software*]: Language Constructs and Features—Patterns; D.4.1 [*Software*]: Process Management—Threads

Keywords Invasive Computing, Resource-aware Programming, X10 Programming Language, Multigrid, HPC

1. Introduction

Following the current trend of parallelization, the aspect of efficiency is gaining attention in high-performance computing (HPC). Optimizing for computational efficiency was so far achieved in several ways: Overcoming low instruction throughput due to evaluation of only a single instruction with single data (SISD) was e.g. achieved by pipelining [11]. Since many applications demand for linear algebra operations, single instruction multiple data (SIMD) extensions were also introduced to personal computers [15]. Due to limitations for increasing the core frequency, single CPUs are nowadays built with multiple cores overcoming the limitations by increasing the parallelism of the cores [3]. Current generations of HPC workstations as well as accelerator cards [7] are limited to core numbers far below one hundred, because there is no hardware support for cache- and memory-coherency among such a high number of cores. For HPC platforms, the way out of this parallelization issue is given by distributed memory systems and application developers being responsible for data exchange and synchronization among those distributed memory nodes. This demands for applications with high scalability especially on distributed memory systems. Among others, Amdahl's law [1] envisions that close-to-linear scalability cannot be guaranteed in general for HPC.

We consider *strong-scaling* by fixing the workload to a particular size while increasing the number of cores. This automatically leads to a decreased efficiency with a growing number of cores since an increase of cores leads to a smaller workload assigned to each core and thus additional time for synchronization relative to the overall computation time.

Above all, dynamical adaptive algorithms and their changing workload during the computation [18] have changing scalability behavior during run-time which we further refer to as different *phases*. Another one of the worst scalability issues is given by IO-bound phases, which is e. g. necessary for the simulation setup, for writing snapshots or output data to the file system. This leads to situations of reduced overall efficiency due to applications running on statically assigned resources without using them all.

1.1 Dynamic Behavior

The scalability issues presented above are related to different phases of an application. This can be improved with a dynamic (changing over time) approach including multiple applications and introduction of a resource manager. For example, a numeric application on a compute cluster could share its nodes with other applications during the initialization phase and allocate more nodes during the actual computation phase. Such an approach requires the other applications to be able to react to external resource requests. Vice versa, the numeric application itself must also be able to adopt to external resource requests. We call such applications *resource-aware*.

We propose applications providing hard constraints and soft hints to a resource manager, aiming for dynamic optimization of the overall system. With applications which have changing resource needs, like the common multigrid approach, the idle resources can be given to another application. In a system of such altruistic applications, the overall throughput should improve, although there might be downside for some of the applications.

X10'13, June 20, 2013, Seattle, Washington.
Copyright © 2013 ACM 978-1-4503-2157-0/13/06. . . $10.00

1.2 Contributions

In summary, our contributions in this paper are:

- a framework for resource-aware programming in X10, which supports adaptation at application-specific points
- a resource-aware multigrid algorithm providing the necessary constraints and hints for invasive computing
- an evaluation how dynamic behavior is exploited to improve throughput in a scenario with multiple applications

1.3 Outline

The remaining sections are structured in the following way. First, we give an introduction to the invasive computing paradigm in section 2 which we employ to handle the dynamic behavior. Then section 3 presents our X10 invasive framework. Our application with a dynamic behavior is a multigrid and is presented in section 4. Finally, our results in section 5 show the observation of the different phases of the application, necessary extensions to the X10 API for invasive computing as well as an improved application throughput.

2. Invasive Computing Paradigm

The dynamic behavior has to be expressed in some way as well as considered by applications. In this work, we employ the invasive computing paradigm [9, 20, 22]. This paradigm suggests a resource-aware programming model, where the program can dynamically adapt to the available resources, e.g., processing elements (PEs), memory and network connections. The request of such resources is further denoted as *invade*. Such an invasion of resources can be specified by constraints. This allows for example to request a certain number of resources or to target specific hardware. After the invasion is finished, the program *infects* the invaded resources, e. g. by using them for a certain computation. If the resources are not needed anymore, the program *retreats* from the resources.

In this work, we aim for improved efficiency through invasive computing. First, dynamic reallocation aims to improve the overall resource use compared to static resource allocation. Resources are shifted between or within applications with respect to the global state. Second, exposing resources more directly to the applications, instead of virtualizing them, removes overhead. For example, CPU cores are assigned to single applications, instead of multiplexing them which would lead to a sharing of lower cache levels and thus a severe slowdown for applications frequently accessing memory. Third, the invasion of the most suited hardware can improve efficiency. For example, not all processing elements in a heterogeneous system might support specific vector instructions. To exploit such heterogeneity, the hardware must be exposed to the programmer. Therefore, the invasive programming paradigm affects the application, the programming language, the compiler as well as the runtime system.

2.1 Demand on Applications

Invasive computing clearly leads to additional work for the programmer. In our case the application developers must extend their applications in three ways.

- First, the application must state its constraints and give hints to the resource manager.
- Second, resource adaptations must be supported by the application. While it is possible to tell the resource manager that adaptations must not occur, the application also gives up its chance to profit from other applications freeing resources.
- Third, the application must handle data migration, which results from resource adaptations. In shared memory systems as well as distributed memory systems with remote memory access, such resource adaptions typically lead to a changing locality of compute cores to associated memory locations. Since data structures are application-specific, this usually cannot be accomplished automatically.

2.2 Resource Manager

First proof-of-concept implementations of an invasive resource manager were made. Depending on the environment, the implementation of the resource manager can be a centralized or distributed resource manager.

A centralized resource management for small SoCs and shared memory systems can be used due to the limited numbers of resources and thus limited search spaces for optimization. The iPMO centralized resource manager presented in [2] offers decision making for applications running on a shared memory platform. This resource manager also accounts for particular phases of a fully-adaptive simulation.

For larger systems a centralized resource manager structure is expected to become a bottleneck and a distributed decision making should be taken. The resource manager presented in [10] is one example of such a distributed resource manager.

2.3 Data-locality and -migration

With dynamically changing assignments of cores to applications, the data dependencies have to be considered for efficient computations. Therefore, the resource manager has to consider whether it is worthwhile to change the resources due to data-migration overhead. Also the application has to consider changing resources since the application programmer has the best knowledge of expressing the necessary data redistributions.

3. The X10 Invasive Framework

For distributed memory systems, communication among computing nodes was traditionally done using MPI. Several libraries and language extensions were developed during the last two decades to enhance the programmability (ease of coding) of such distributed memory systems. Among others, Co-Array Fortran [13] extends the Fortran language itself, Global Arrays [12] offer data access and operations on distributed arrays and UPC [6] extends the C language.

However, such approaches also inherit the baggage of the programming language they build upon. In contrast, X10 [17] is a language which brings modern features to the field of scientific computing with parallelization considered from the very beginning. This includes well-known advantages like type safety, a module system, a partitioned global address space, generic programming, and integrated concurrency. Additionally, X10 also includes promising new features like dependent types and transactional memory (via `atomic` and `when`). Since this feature combination is not available with Fortran and C++, we decided to use X10 for the development of our invasive framework and resource-aware applications.

3.1 Hardware and Software Layers

Originally, the invasive framework is developed for a multiprocessor System on Chip, with custom hardware [16] and operating system [14]. An invasive application runs on the *invasive runtime support system (iRTSS)*, which includes an agent system [10] for global resource management. Within the agent system, each application is represented by one agent. If an application wants to invade additional resources, its agent checks whether and how the request can be fulfilled. In case of concurrent requests for the same resource, the corresponding agents are responsible for finding a

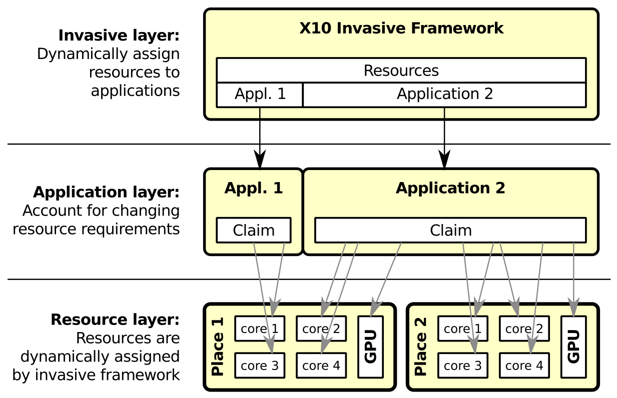

Figure 1. Our invasive architecture: The X10 invasive framework is responsible for the resource management. The applications are responsible to consider changing resources and account for appropriate execution of kernels on changed claims. With the second application requesting GPUs with appropriate constraints during the *invade*, the GPUs are assigned to this application without interfering with it. The hardware layer represents invadable resources assigned exclusively to an application.

suitable solution. However, for our evaluation we used a prototype version of the iRTSS, which is implemented in pure X10.

On the language level, we encapsulated the resource-aware features in an X10 library. We chose X10 because it already provides various features that make it suitable for programming invasive architectures. For example, the partitioned global address space (PGAS), which models a cluster network, fits quite naturally to a tiled cache-incoherent many-core architecture. The employed X10 concept is a *place*, such that each tile maps to a place. Essentially, activities within the same place have shared memory, whereas activities in different places must communicate with other means.

Programming with incoherent caches is challenging. One programming model is the partitioning of available memory, assigning a part of it to each tile. This PGAS model is mapped to X10 by representing each tile as a place. The exchange of information between tiles only happens on at expressions, so the runtime system can ensure a correct sequence of cache flushing and synchronization. Partitioning the address space inhibits sharing a memory cell between multiple tiles, although the architecture would allow this. However, it is possible [4] to take advantage of multiple readers to alleviate the costs of copying data between partitions.

A prototype variant of the framework is implemented in pure X10 which we use in this work for early evaluation as it can be executed on common HPC many-core and cluster systems. An exemplary sketch of an invasive architecture for HPC systems is given in Figure 1. In contrast to standard X10, places of an application can be created and destroyed conceptually depending on the resource allocations. This means that some features from the standard library must be removed, e. g. the list of all places. Destroying a place means all data there is lost and migrating an activity there would fail. The full safety implications for the programmer have yet to be investigated. In our prototype, places are merely reserved and no failure checks are performed.

3.2 Invade, Infect, Retreat & Claims

The simplest invasive program is shown in Figure 2. The concept of allocating, using and freeing resources is known from memory allocation. Invasive computing generalizes the concept to invade, infect and retreat under specific constraints. So far the framework supports the following reusable resources:

```
val ilet ← (id:IncarnationID) => {
  do_something(id);
};
claim ← Claim.invade(constraints)
claim.infect(ilet)
claim.retreat()
```

Figure 2. The basic idea of invasive programming: The ilet function provides an action to perform in parallel on all allocated processing elements; Invade allocates resources under specific constraints in competition with other applications; Infect uses those resources by letting iLets (basically compute kernels) run; Retreat frees allocated resources.

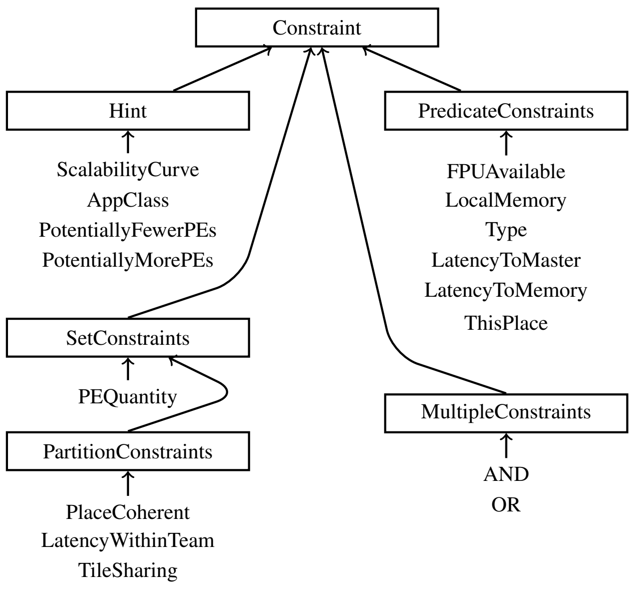

Figure 3. Constraint-hierarchy for invasion: The boxes represent abstract base classes. An arrow represents inheritance in classic object-oriented fashion.

Processing element (PE) which basically maps to a core. We assume exclusive rights to compute on a core, which means the operating system must restrict the respective activities to this core and not schedule other applications onto this core.

Place-local memory partitioning the physical memory of a shared memory domain for all the applications running there.

Inter-place network connections Applications can invade network connections to a certain extend, such that the network guarantees bandwidth or latencies.

Constraints for invasion are structured in a tree-like structure with constraints stored on each node. Each constraint type is implemented by a different class, which form a hierachy as shown in Figure 3. The most used constraint in practice is PEQuantity, which specifies number of PEs necessary to start and/or continue the computation. The class of predicate constraints is relatively simple, as they place a constraint on the requested PEs, like specific instruction set extensions. Partition constraints are complex in comparison, as they specify requirements for the whole set of the requested resources. For example, place coherence means all PEs share a memory domain. Effectively, they must be in the same place. The constraint hierarchy also includes AND and OR combinators to

```
claim.infect(ilet)
// optimize resource allocation:
val changed1 ← claim.reinvade()
claim.infect(ilet)
// respecify resource needs
val changed2 ← claim.reinvade(otherConstraints)
```

Figure 4. Resource-aware programming: The reinvade method without arguments gives the resource manager the opportunity to adapt resource allocation to a changed situation. This should be done, whenever an application has a global synchronization point to yield a maximum of flexibility. The reinvade method with new constraints arguments also allows to reallocate resources, but additionally the application provides a new specification. This should be done whenever an application knows that resource requirements changed.

construct complex constraints. This allows the programmer to provide multiple implementations for different types of processing elements. For example, the programmer might provide special code to exploit the vector instructions in addition to a slower fallback variant. At last there are hint constraints to specify non-functional aspects. For example, scalability curves contain information about the scalability of an application.

3.3 Resource Adaptation

The reinvade method is presented in Figure 4, which is one of the major mechanisms for resource-aware programming [2]. While there are more, we concentrate on the mechanism required to reproduce the results presented in this work.

Semantically, the reinvade method is equivalent to retreat followed by invade. However for the underlying resource manager, reinvade can be implemented more efficiently. Most of the time, a prepared resource adaption is performed and the application can continue without inter-application interaction.

While such a resource adaptation is not expected to speed up an algorithm, the goal is to speed up the system as a whole. For example, Speck et al.[19] implemented a sorting algorithm, which builds on invasive principles. While the management overhead seems to be negligible, the application throughput and thus efficiency improves on a loaded system, where common approaches rely on the thread switching of the operation system. An important property of the algorithm is that it is *malleable*, which means the amount of parallelism can be changed without significant overhead at arbitrary points in time.

The hard part of resource management is to balance applications against each other. If we have two equal applications, we could simply give half the cores to each application[8]. However, applications usually are not equal. For example, one application might not benefit from more than four cores, while the other one is embarrassingly parallel. A straightforward resource assignment would give four cores to the first application and the remaining cores to the second application. However, the performance improvement of three compared to four cores for the first application could be smaller than using one more core for the second application. Effectively, we could yield better overall performance, if we transfer one core from the first to the second application. To give the resource manager a chance to handle such cases, the applications must provide hints like a *scalability curve*. This scalability curve has to be determined either online or offline by the application developer. Such a curve would be linear for the embarrassingly parallel application and logarithmic for the first application, like you can see in Figure 5.

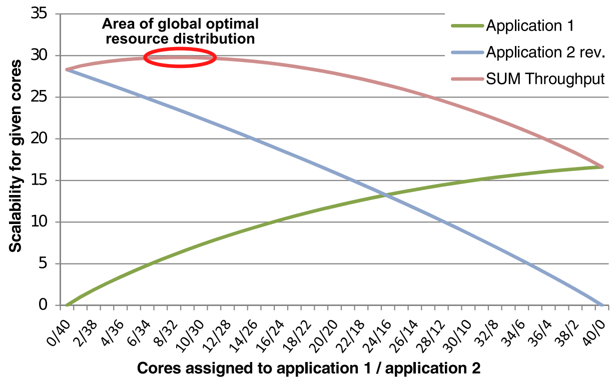

Figure 5. Scalability graphs of two applications and global throughput: The x-axis shows the resource partitioning between two applications. The y-axis the respective scalability of each application and the whole system throughput. The system throughput is directly proportional to the scalability value within our simplified model. Obviously, the global optimum is not optimal for each application.

3.4 Data Migration

The application itself may require data redistribution for load balancing reasons. If the number of cores per domain is changed by the resource manager, it can be beneficial to migrate data. As this is an algorithm-specific decision, a generic resource manager cannot perform this data migration.

With distributed arrays, X10 provides a convenient way of running computations on distributed data. While conventional distributed arrays are allocated and computations scheduled statically across processing elements, the invasive approach requires additional flexibility. For example, the data distribution must comply with the resource allocation, which is currently not available with the standard libraries of X10. When the resources are adapted while the program is running, the data must be redistributed accordingly.

Since the semantics of distributed data arrays are usually only known to the application developer, also the decision to redistribute them is part of the the application developer. For our implementation, such a request for a redistribution is currently triggered in case of a modifications of the current claim.

4. Application

To offer a realistic dynamic application, we implemented a time-dependent simulation of a laser engraving symbols on a metal plate. First of all, this leads to dynamic workload due to changing injected heat over time, e. g. with no heat during the first few time-steps. Second, we also get a dynamic behavior during computation of the solution of a single time-step due to the requirements of the multigrid solver which we use for this application.

We continue with a short description of the mathematical formulation of our problem as well as its discretization and refer to [5] and [21] for detailed information on multigrid solvers.

The heat distribution over time in an isotropic material is given by the following equation:

$$\frac{dT(x,y,t)}{dt} = \alpha\Delta T(x,y,t) + E(x,y,t), \quad (x,y) \in \Omega.$$

This equation describes the temperature distribution T, an external energy input E on our domain $\Omega = [0;1]^2$ our material is exposed to by the laser, and the thermal diffusivity coefficient α. We set the boundary conditions to 0 implementing homogeneous Dirichlet boundary conditions $T(x,y) = 0, (x,y) \in d\Omega$.

Basic variables:

N	problem size
x	solution
b	right hand side
r	residual
e	approximated error

Multigrid:

Nr	problem size for restricted level
rr	restricted residual
er	restricted approximated error

Claims:

nc	new claim after reinvade
nc2	updated claim after V-cycle for lower levels

```
vcycle(N, x, b):
    r ← computeResidual(N, x, b)
    while |r| > threshold:
        vcycleIteration(N, x, b)
        r ← computeResidual(N, x, b)


vcycleIteration(N, x, b):
    smoother(N, x, b) # pre−smooth
    r ← residual(N, x, b) # residual

    nc ← reinvade(N, claim) # reinvade claim

    Nr ← N/2 # restricted level
    rr ← restrict(N, r) # restrict residual to new claim
    er ← (Nr, 0) # setup error with 0 values

    nc2 ← vcycleIteration(Nr, er, rr) # vcycle

    # redistribute
    if (nc ≠ nc2):
        nc ← nc2
        x.redistribute(Nr, nc)
        b.redistribute(Nr, nc)

    e ← prolongate(Nr, er) # prolongate error
    x ← x + e # apply correction

    smoother(N, x, b) # post−smooth

    return nc # possibly modified claim
```

Figure 6. Invasive parallel multigrid in pseudo code

For spacial discretization we store the temperature values at $N \times N$ grid-points for each discrete grid-point (i, j) to a 2D array. The Laplace operator is discretized by the stencil approximating the 2nd partial derivatives

$$\Delta \approx \frac{1}{h^2} \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

with the mesh-width h. The stencil is applied by computing the discrete convolution using this stencil as the convolution kernel.

For discretization of the time-stepping, we approximate the temperature distribution with 1st order forward differences and use an implicit update scheme

$$\frac{T(x, y, t + \Delta t) - T(x, y, t)}{\Delta t} = \Delta T(x, y, t + \Delta t) + E(x, y, t).$$

This leads to the system of linear equations $\mathbf{A}\vec{x} = \vec{b}$ representing the implicitly discretized heat-equation. The external energy is handled by appropriate updates of $\vec{b}$. The approximated solution for $T(x, y, t)$ is then given in $\vec{x}$ after computation of an approximated solution of this system of equations.

Since solving such a systems of linear equations by using direct solvers —e. g. Gauss-elimination— would destroy the sparsity pattern of the matrix, we employ an iterative solver to compute an approximate solution.

The Jacobi-Solver is one of such iterative solvers which we employed here to solve our system of equations: The matrix A is formally decomposed into $\mathbf{A} = L + D + R$ with L the lower diagonal components, D the diagonal components and R the upper diagonal components. One solver-iteration is executed by

$$\vec{x}^{(i+1)} = D^{-1}(\vec{b} + (D - A)\vec{x}^{(i)})$$

with $\vec{x}^{(i)}$ storing the approximated solution of $\vec{x}$ after the i-th iteration.

With this iterative solver, we avoid storing the matrix A explicitly. We use the Euclidean norm on the residual $\vec{r}^{(i)} = \mathbf{A} \cdot \vec{x}^{(i)} - \vec{b}$ for the stopping criteria at the end of each V-cycle.

4.1 Geometric Multigrid Solver

Applying the iterative Jacobi solver for the heat equation directly would lead to elimination of high frequencies in error only while leaving low frequencies in error almost unchanged. Multigrid solvers were developed to account for elimination of lower frequencies by applying the solver on coarser grids as well. The scheme of a multigrid V-cycle running on different resolutions of the original problem is given in Figure 7. We use the error-correction scheme restricting the residual instead of the solution as it is the case for full-approximation scheme. This reduces our 2D problem size on each level by a factor of 4 which accounts for our *dynamic behavior* when solving for the next time-step. E. g., if we solve a heat equation with a size of 128 × 128, this leads to 7 levels for the up- and down-cycle. Exemplary results computed with this algorithm are given in Figure 8.

4.2 Parallelization

Three data arrays have to be stored for each multigrid level: The approximated solution $\vec{x}$ of the current level, its right side $\vec{b}$ and the residual $\vec{r}$ which is prolonged to the finer level. We used the slicing method for the domain decomposition distributing our domain to computation units by splitting it along the 2nd array dimension.

For our implementation in X10, all data arrays are stored in distributed arrays with appropriate invasive extensions discussed in subsection 5.1.

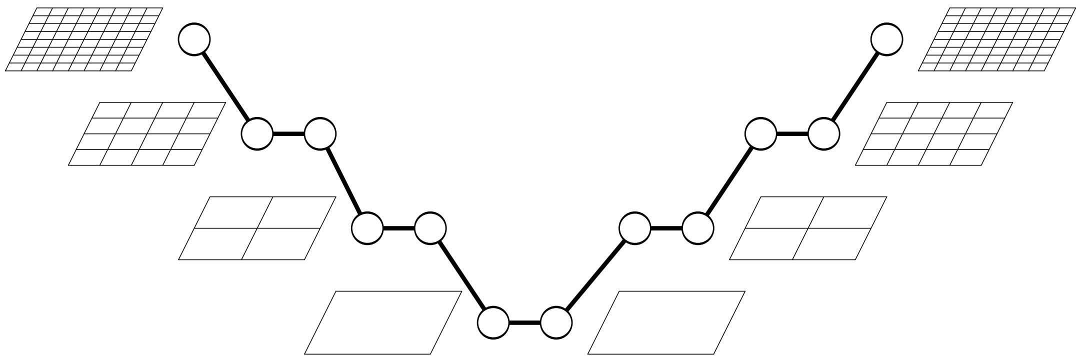

Figure 7. Schematic overview of multigrid V-cycle. Each level has different scalability behavior which results in dynamic behavior.

4.3 Invasive Parallel Multigrid

The parallel invasive multigrid program with a V-cycle is given in pseudo-code in Figure 6. Extensions for invasion are in line 12 and lines 21-24.

Obviously, multigrid levels with a problem size below the number of cores available in the system would not lead to any benefits from the cores unable to run computations on the level.

Running computations on levels with lower problem size, our multigrid also requests fewer compute resources. Once the resource manager redistributes this resources, two crucial aspects have to be considered for distributed memory systems present on our target platforms: data locality as well as data migration. While the responsibility of data locality is given to the resource manager which is described in the next section, the data migration has to be handled by the application itself. In case of a redistribution of resources, possible data migration has to be done to decrease the latency to access data stored at other places.

5. Results

5.1 Invasive Interfaces and Extension of X10 APIs

The finely-grained resource management within the invasive framework results in a dynamic heterogeneous environment. The amount of parallelism within a place varies from place to place and over time. Therefore we extended the distributed arrays in X10 to respect these differences for the data distribution and to work on the granularity of processing elements. A PEDist class inheriting from x10.array.Dist was implemented. The information from the resource manager must be considered when computing the PEDist.

Similarly, we needed to redistribute data, which is not supported by the usual DistArray class. Specifically, a helpful extension for distributed arrays would be a distarray.redistribute(dist) method, which changes to Dist of distarray to dist, such that the data is internally redistributed. This extension might also be interesting for non-invasive applications, so including this into the standard library of X10 should be worthwhile.

5.2 Programmability

Compared to other languages, the language features at of X10 lead to improved programmability. Among others, the flexibility of the data distribution and its access transparency to the application developer with the combination of distArrays, regions and at as well as synchronization barriers which are handled automatically across places after triggering an execution of a binary at another place.

For invasive computing, the invade and infect methods using lambda functions to express kernel routines provide a clear separation between claimed resources and actions executed on these resources. The processing resource is given as a parameter to the lambda function. In combination with out extension to the distributed arrays of returning the region assigned to each processing element, the corresponding regions of distributed arrays can be efficiently determined by each lambda function.

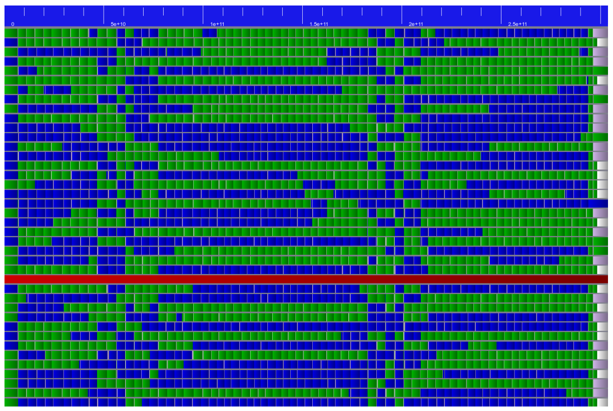

Figure 9. Two multigrid applications exchanging resources visualized using the ViTE tool. X-axis is time and each line on the Y-axis represents one processing element. The colors blue and green represent the ownership to the respective application. The red line is a coordinating master application occupying one core.

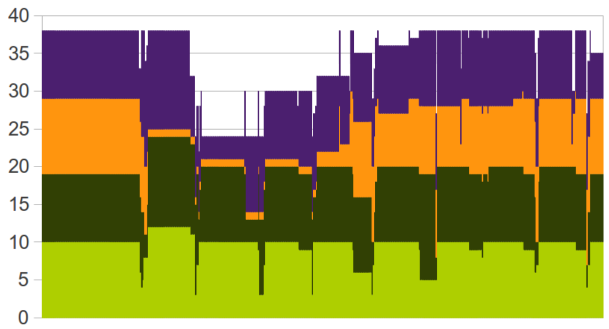

Figure 10. Four multigrid applications, each one drawn with a different color, with dynamic resource demands. Over a time span of 500 seconds they use a varying amount of up to 38 PEs. In between only 24 PEs are used, which means 37% of the cores idle.

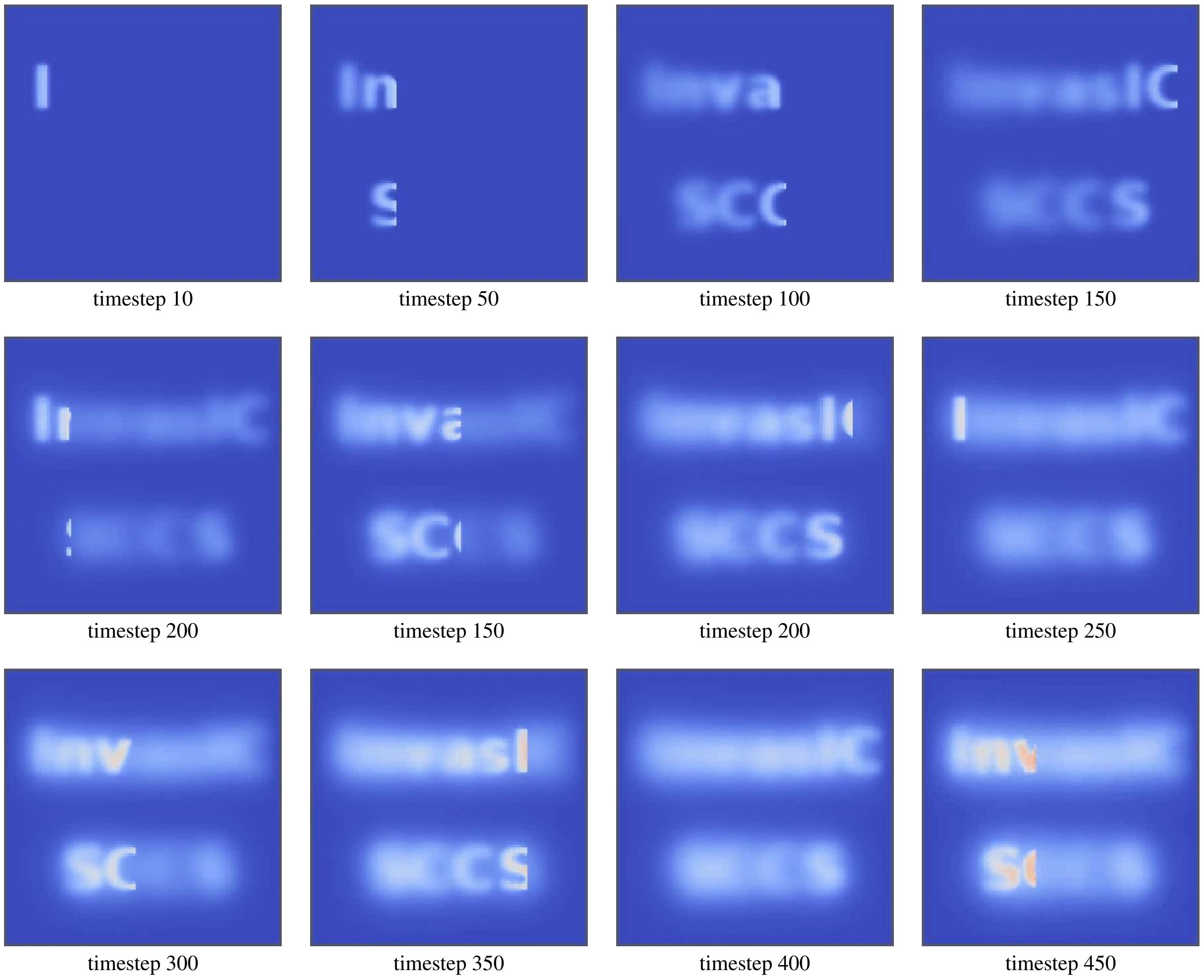

Figure 8. Simulation data for our multigrid method. The heat distribution of a metal plate during an engraving is simulated and visualized. The laser is moving from left to right.

5.3 Resource Adaption

In our multigrid case, the resources cannot be adapted at arbitrary program points. Adaptation is done, whenever the granularity is changed as illustrated in Figure 7. The programmer is responsible to provide an appropriate granularity of (re)invade use. A lower utilization of invade leads to lack of adaptivity and idle time; frequent invades lead to increased management overhead. While the resource manager tries to minimize overhead, the optimization is deferred to a background activity. However, there is no guarantee how long (re)invade takes as mentioned in subsection 3.3.

An example, with two applications is given in Figure 9. Likewise with four applications in Figure 10 These measurements were done on a shared memory system with a single place due to restrictions of the implementation. Therefore, no data migration was performed.

Since resource adaptation might require data migration, additional overhead is burdened on the programmer. While convenience methods offered by an invasive X10 API can simplify the task, the application developer must explicitly support knowledge of redistribution through constraints and callback mechanisms.

5.4 Higher Application Throughput

We evaluated our invasive application with our X10 invasive framework. The multigrid application requests only as many resources as there are array-columns available for our sliced domain decomposition (See subsection 4.2). Our benchmark was performed on a four-socket shared memory system, each socket equipped with an Intel(R) Xeon(R) CPU E7 4850 running at 2.00GHz. Since each CPU has 10 physical cores, this leads to 40 physical cores which were managed by the resource manager. The benchmark was started all multigrids at the same time. Our results are given in Figure 11 with different number of applications executing and a single application for our base line. Despite running the benchmark on a shared memory system, we still do a data migration required for execution on several distributed memory nodes by copying all data from the source array to the new array with respect to the new distribution of the processing elements using an owner-compute access. The results show an overall increased application throughput while

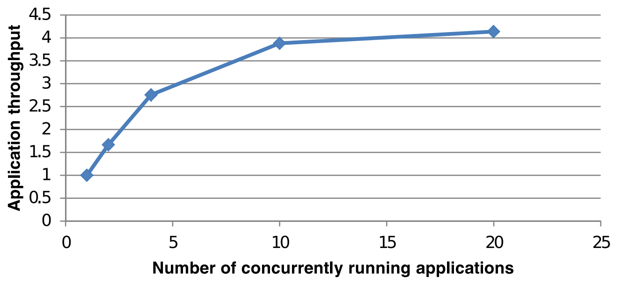

Figure 11. To exploit the dynamic behavior of an application for increased throughput, there must be another application available to use free resources. This plot show the relative application throughput when the number of parallel applications increases. Compared to a single application running alone, a second application can only increase the throughput by 80%. A value of 100% would indicate, that a single application is not scalable beyond 20 cores. With ten parallel applications the throughput is nearly four times as high.

considering the different applications phases as presented in the previous section.

6. Conclusions and Outlook

We demonstrated how the common multigrid HPC application can be made resource aware. This enables a resource manager to exploit the dynamic behavior and reallocate resources to optimize global throughput. Our current resource manager is not that clever. For example, it does not consider the cost of data migration or locality, yet.

The use of X10 with its modern language features improves programmability and counteracts the increased programmer burden of Invasive Computing.

With energy efficiency becoming more important in HPC, we aim to extend our invasive framework with energy aware features to reduce power consumption for cores not invaded by an application. We also plan to examine the class of malleable applications in more detail. When resources can be adapted at all times, They should be able to absorb all resources, which would otherwise idle. Also, malleable applications can shrink at the request of other applications at arbitrary points. Hence, malleability provides a maximum of flexibility for the resource manager.

7. Acknowledgments

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

References

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS ’67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[2] M. Bader, H.-J. Bungartz, and M. Schreiber. Invasive computing on high performance shared memory systems. In *Facing the Multicore-Challenge III*, volume 7686 of *Lecture Notes in Computer Science*, Sept. 2012.

[3] S. Borkar and A. A. Chien. The future of microprocessors. *Commun. ACM*, 54:67–77, 2011.

[4] M. Braun, S. Buchwald, M. Mohr, and A. Zwinkau. An X10 compiler for invasive architectures. Technical Report 9, Karlsruhe Institute of Technology, 2012.

[5] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial*. Society for Industrial Mathematics, 2000.

[6] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren. *Introduction to UPC and Language Specification*. 1999.

[7] J. Dokulil, E. Bajrovic, S. Benkner, S. Pllana, M. Sandrieser, and B. Bachmayer. Efficient Hybrid Execution of C++ Applications using Intel(R) Xeon Phi(TM) Coprocessor. *CoRR*, abs/1211.5530, 2012.

[8] M. Gerndt, A. Hollmann, M. Meyer, M. Schreiber, and J. Weidendorfer. Invasive computing with iomp. In *Specification and Design Languages (FDL)*, pages 225 –231, Sept. 2012.

[9] F. Hannig, S. Roloff, G. Snelting, J. Teich, and A. Zwinkau. Resource-aware programming and simulation of MPSoC architectures through extension of X10. In *Proceedings of the 14th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, pages 48–55. ACM Press, June 2011.

[10] S. Kobbe, L. Bauer, J. Henkel, D. Lohman, and W. Schröder-Preikschat. DistRM: Distributed resource management for on-chip many-core systems. In *Proceedings of the IEEE International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 119–128, Oct. 2011.

[11] K. Murakami, N. Irie, and S. Tomita. SIMP (Single Instruction Stream/Multiple Instruction Pipelining): A Novel High-speed Single-processor Architecture. *SIGARCH Comput. Archit. News*, 17(3):78–85, Apr. 1989.

[12] J. Nieplocha, R. Harrison, and R. Littlefield. Global arrays: A nonuniform memory access programming model for high-performance computers. *The Journal of Supercomputing*, 10:169–189, 1996.

[13] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998.

[14] 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-CoreArchitectures (SFMA)*, volume USB Proceedings of *Sixth International ACM/EuroSys European Conference on Computer Systems (EuroSys)*, pages 9–14. EuroSys, Apr. 2011.

[15] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *Micro, IEEE*, 16(4):42 –50, Aug. 1996.

[16] R. K. Pujari, T. Wild, A. Herkersdorf, B. Vogel, and J. Henkel. Hardware assisted thread assignment for RISC based MPSoCs in invasive computing. In *Proceedings of the 13th International Symposium on Integrated Circuits (ISIC)*, Dec. 2011.

[17] V. Saraswat, B. Bloom, I. Peshansky, O. Tardieu, and D. Grove. *X10 Language Specification Version 2.3*, Oct 2012.

[18] M. Schreiber, H.-J. Bungartz, and M. Bader. Shared memory parallelization of fully-adaptive simulations using a dynamic tree-split and -join approach. Puna, India, Dec. 2012. IEEE International Conference on High Performance Computing (HiPC), IEEE Xplore.

[19] J. Speck, P. Sanders, and P. Flick. Malleable sorting. In *International Symposium on Parallel and Distributed Processing*. IEEE Computer Society, May 2013.

[20] 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.

[21] U. Trottenberg, C. Oosterlee, and A. Schüller. *Multigrid*. Academic Press, 2001.

[22] A. Zwinkau. Resource awareness for efficiency in high-level programming languages. Technical Report 12, Karlsruhe Institute of Technology, 2012.