myjournal manuscript No.
(will be inserted by the editor)

Invasive Compute Balancing for Applications with Shared and Hybrid Parallelization

Martin Schreiber · Christoph Riesinger · Tobias Neckel · Hans-Joachim Bungartz · Alexander Breuer

Received: date / Accepted: date

Abstract Achieving high scalability with dynamically adaptive algorithms in high-performance computing (HPC) is a non-trivial task. The invasive paradigm using *compute migration* represents an efficient alternative to classical data migration approaches for such algorithms in HPC. We present a core-distribution scheduler which realizes the migration of computational power by distributing the cores depending on the requirements specified by one or more parallel program instances. We validate our approach with different benchmark suites for simulations with artificial workload as well as applications based on dynamically adaptive shallow water simulations, and investigate concurrently executed adaptivity parameter studies on realistic Tsunami simulations. The invasive approach results in significantly faster overall execution times and higher hardware utilization than alternative approaches. A dynamic resource management is therefore mandatory for a more efficient execution of scenarios similar to our simulations, e.g. several Tsunami simulations in urgent computing, to overcome strong scalability challenges in the area of HPC. The optimizations obtained by invasive migration of cores can be generalized to similar classes of algorithms with dynamic resource requirements.

Keywords Invasive Computing · compute migration · high-performance computing · hybrid parallelization · dynamic adaptive mesh refinement

1 Introduction

In many applications modeled with partial differential equations (PDE) the current trend is to use dynamic-adaptive mesh refinement (DAMR). Adaptivity in general accounts for feature-rich areas by refining the grid, if this area

M. Schreiber, C. Riesinger, T. Neckel, H.-J. Bungartz, A. Breuer
Technische Universität München, Fakultät für Informatik
Boltzmannstraße 3, 85748 Garching, Germany
E-mail: {martin.schreiber, riesinge, neckel, bungartz, breuera}@in.tum.de

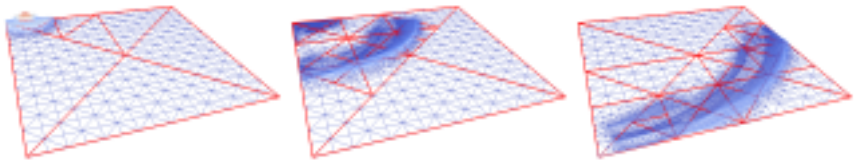

Fig. 1 Visualization of dynamically changing triangular grid created by dynamical adaptive simulation with 1st order basis functions. Blue cells represent the mesh with their height the water surface elevation. The red borders indicate the partitions.

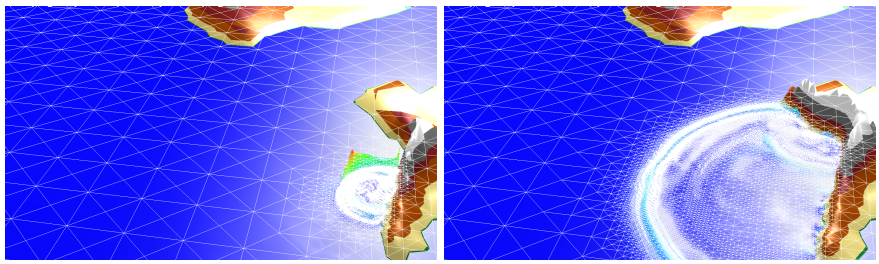

Fig. 2 Visualization of selected time steps of a Tsunami simulation executed on a dynamic adaptive triangular grid based on displacement datasets for the Chile earthquake 2010. The water is colored with a rainbow map according to its displacement relative to sea surface and the water surface elevation is scaled up for enhanced visualization. Note the highly increased grid resolution which is directly related to the changing computational workload before and after the propagating Tsunami wave fronts.

significantly contributes to the final result, and coarsening the grid in case that the result is not highly dependent on this area [5, 11, 30, 46]. In comparison to regularly resolved simulations, such simulations aim e.g. for the highest possible science-per-flops-ratios [25]. This approach leads to significantly shorter run times while keeping the order of accuracy of highly refined regular grids.

Fig. 1 shows an example of a shallow water simulation which uses dynamical h-adaptive grids also applicable to Tsunami simulations [3] (see Fig. 2).

Feature-rich areas near the wave fronts are higher resolved than other areas. However, realizing dynamical h-adaptivity involves additional demands not only on the underlying grid and data management but also on providing subgrid-migration for parallelization with distributed-memory concepts.

Due to the computational intensity and the memory bandwidth requirements, a parallelization of adaptive PDE algorithms is mandatory. Currently, such parallelizations are usually achieved via threading libraries such as OpenMP, via MPI, or via hybrid approaches combining distributed- and shared-memory parallel programming to obtain benefits of both implementations [16].

The trend for modern CPU architectures is clearly towards many-core processors with shared-memory domains (e.g. Intel Xeon Phi). A purely threaded parallelization can lead to several overheads such as *increased management of structures and thread synchronization* [37], *false sharing*[10] and *resource sharing of a single program*[29].

All the beforementioned overheads can be damped by using a lower number of threads in each program context. Here, we evaluate two different parallelization methods for concurrently running programs: First, a pure *shared-memory parallelization* with resource-competing and concurrently-running applications and second a *hybrid parallelization* with a single application executed on multiple multi-threaded program instances on each MPI rank. With our requirements of running more than one thread in a program context, pure distributed-memory parallelization methods are not considered in the discussion. For the following sections, we use the MPI and OpenMP terminology for the distributed and shared-memory parallelization, respectively.

With parallelization models for distributed- and hybrid-memory systems, state-of-the art simulation software for dynamically changing grids has to cope with load imbalances to provide a scalable simulation. These imbalances are typically tackled with a *data migration* approach. This sets up demands to the application developer either to extend interfaces of meshing tools supporting dynamical remeshing and load balancing or to manually implement the load balancing and data migration in the simulation software. Tackling load balancing so far was only resolved by explicit (developer-provided), or implicit (framework-provided) *workload migration*, thus requiring extensions for redistributing data. This typically involves severe programmability (see e.g. required interfaces in [11, 46]) and migration-latency overheads (see [15]). Additionally,spontaneous and typically non-predictable load imbalances can occur such as for computations with local-state depending number of instructions [21] and computations executed only on subsets of the overall grid, e.g. for local-residual corrections [34] and simulation output of data only laying in a fast moving field of view [17]. These effects can lead to frequent data migration of larger chunks. Here, *dynamically changing resources* provide a potential solution to (a) programmability by clearly defined interfaces and programming patterns and (b) data-migration overheads by using compute migration; furthermore, spontaneous workload imbalances can be handled more efficiently by fast compute migration.

2 Existing work and Invasive Computing

Handling changing demands for resources during run time was investigated in different areas in the last two decades: the basic principles for scheduling of multi-programmed applications originate from embedded systems, especially in the context of real-time applications (c.f. [33]), which need special scheduling algorithms (c.f. [45]) that deal with the inherent dynamics. However, these algorithms do not consider hybrid parallelization or HPC systems. Embedded systems often consist of heterogeneous multiprocessors where the single cores have different capabilities such as additional floating-point units, exclusive caches, etc. (c.f. [6, 2]). In our work, we focus on homogeneous multicore processors. Current HPC systems also utilize a memory protection, preventing inter-application work stealing with today's HPC threading libraries

(e.g. OpenMP, TBB) due to separated address spaces. This leads to additional constraints for the HPC architectures considered in this work.

An additional challenge arises when several multi-programmed applications with changing demand on resources run on the same node in parallel and compete for congested resources. Bhadauria et al. [8] tackle this problem and optimize the thread throughput of all running applications in a global view. For that, information on the scalability of the single applications is required. This information is collected by a software-based performance monitoring unit (PMU) during run time. Corbalan et al. [13, 14] are using a similar approach. The scalability information is gathered by the *SelfAnalyzer* during run time, the scheduling itself is done by the Performance-Driven Processor Allocation (PDPA) policy. In contrast, our approach can also use information such as scalability graphs and workload information based on the explicit knowledge of the application developer. This results in more recent (e.g. using the number of workload of the current time step) rather than over-time derived performance information. In addition, we are pursuing for maximal global throughput while Corbalan et al. try to fulfill certain given target efficiencies.

Hybrid parallelization is indispensable when exploiting modern HPC clusters (c.f. [16, 27]) but little investigation has been done in combination with changing demand of resources during run time. Garcia et al.'s approach (c.f. [20]) tackles the issue in an interesting manner, but loses flexibility due to limitations of OpenMP and SMPSuperscalar and does not offer the opportunity to provide scalability information on the application by the developer.

The approach considered by Hsieh [24] is closely related to our approach: Instead of migrating data, computational resources are migrated when they are needed which avoids data-migration overheads. He also uses distributed shared-memory systems as target platform. However, his approach is not based on standard HPC programming models (OpenMP/TBB, e.g.), hence requires significant changes in the application, and thus does not evaluate dynamic resource scheduling for applications developed with standard HPC programming models.

The invasive computing paradigm was originally introduced to be applied on embedded systems (see [42] for an overview) targeting to optimize dynamically scheduled resources by developing InvasIC-enabled hardware and software. The paradigm is currently subject of research in the InvasIC TCRC 89[1] with its main focus on embedded systems. For HPC systems, this paradigm covers all issues involved in compute migration:

From the application developer's point of view, resources assigned to an application are dynamically changing during run time. Resources which are not used by an application can be assigned to another application. Applications themselves behave in a resource-aware manner, offering information to a resource manager or multiple cooperating resource managers to optimize the resource distribution over all applications and providing computing resources if demanded by the resource manager. Three clear basic interfaces are sug-

[1] http://invasive-computing.de

gested in the context of Invasive Computing, which can be directly applied to our *compute migration* issue: *invade*, *retreat* and *infect*. With *invade*, resources are requested depending on particular application-specific requirements. Free resources matching the requirements are then returned to a so-called *claim*. Computations on the granted resources are then started by executing kernels on resources made available in the claims, also described by *infect*ing resources. Resources can be finally released by using *retreat* on the owned resources.

In contrast to auto tuning, Invasive Computing puts its focus on application-supported optimization, thus moving the input for optimizations to the responsibility of the application developer. Optimizations are then achieved by a centralized [4] or decentralized [26] resource manager.

When applying this invasive paradigm in reality, several extensions are required, such as asynchronous invades [4] to overcome scheduling latencies and iOMP as an extension to OpenMP [22].

3 Contribution

Our contribution is the exploration and optimization with the Invasive Computing paradigm applied to compute migration for simulations with shared- and hybrid-parallelization on dynamically adaptive grids in the context of PDE simulations.

These simulations (see Sec. 4) lead to dynamically changing application requirements regarding computational resources and, thus, extensions in the invasive resource manager for dynamical compute balancing. We then present the realization of *compute balancing* with Invasive Computing for shared- and hybrid-parallelized application scenarios based on a resource manager (Sec. 5). The benefits of *compute balancing* for this class of applications are then shown for several different benchmark suites (Sec. 6).

4 Simulations with dynamic adaptive mesh refinement

The shallow water equation (SWE) Tsunami simulations described below are based on a dynamically adaptive triangular grid: In each time step, the grid is refined by triangle bisection in grid areas with a large contribution to the result we are interested in and coarsened in grid areas with a low contribution. Triangles are chosen as basic elements assembling the domain to run computations on conforming grids which clearly would not be possible with h-adaptive Cartesian grids. Running simulations on such dynamically adaptive grids typically leads to a higher science-per-flop ratio, but introduces load-imbalances due to the dynamically changing grid and thus workload.

On shared-memory systems, a parallelization of spatial meshes can be tackled in a variety of ways:

(a) One approach is storing patches in each cell: Instead of storing only the data for a single cell in one dynamically adaptive grid cell, regular grid

structures (patches) containing multiple cells are stored in each patch. Parallelization within each patch by executing operations on the patch concurrently [31] leads to low scalability for small patch sizes.

(b) Another way is ordering all cells, e.g. based on one-dimensional space-filling curves (SFC) index projections [36, 7], and using the one-dimensional representation for partitioning. Here, the communication meta information is stored per-cell or for each hyper face shared among different partitions. However, such meta information typically only allows single-threaded processing of each partition (see e.g. [28] with the parallelization from [44]). (c) Cluster-based parallelization strategies provide an alternative to the previously mentioned parallelization strategies and we continue with a description of this new alternative: They split the domain into a bulk of connected grid cells with consideration of spatial locality, e.g. by using space-filling curves, but contrary to (b), this approach uses a different meta-information scheme and software design: we demand the ability of efficient cluster-based local-time stepping (C-LTS) [12]. An efficient software design of such a C-LTS yields requirements of replicated interfaces [40] between each clusters and communication schemes with run-length encoding for efficient communication in a multi-threaded and multi-node environment [38]. The resulting software design directly yields efficient DAMR simulations with shared and hybrid parallelization.

For this work, our cluster generation is based on tree splits of SFC-induced spacetrees: clusters are then split and joined depending on local or global information on the grid [39]. Since this algorithm offers high scalability as well as performance boosts via cluster-based optimizations and is applicable to Tsunami simulations [3], this provides a solid base line for the evaluation of our invasive compute-balancing strategies with realistic applications.

Our major target application is given by concurrently executed Tsunami simulations. Instead of running a three-dimensional flow simulation, one may apply a frequently used and well established approximation based on the assumption of shallow water in the regions of interest. This allows a simplification of the three-dimensional Navier-Stokes equations to the two-dimensional shallow water equations (SWE). Furthermore, we use a discontinuous Galerkin (DG) method for the spatial discretization (see e.g. [1]). We consider the homogeneous form given by the conservation law of hyperbolic equations

$$\frac{\partial U(x,y,t)}{\partial t} + \frac{\partial G(U(x,y,t))}{\partial x} + \frac{\partial H(U(x,y,t))}{\partial y} = 0 \ , \tag{1}$$

or in shorthand form

$$U_t + G_x(U) + H_y(U) = 0$$

with $U = (h, hu, hv)^T$ and

$$G(U) = \begin{pmatrix} hu \\ hu^2 + \frac{1}{2}gh^2 \\ huv \end{pmatrix} \qquad H(U) = \begin{pmatrix} hv \\ huv \\ hv^2 + \frac{1}{2}gh^2 \end{pmatrix}.$$

The conserved quantities $(h, hu, hv)^T = (\text{height, x-momentum, y-momentum})^T$ of the water are given by $U(x, y, t)$ with parameters dropped for sake of clarity.

The particle velocity components as part of the primitive variables along the unit vector $\mathbf{e}_i$ in direction i are given by $(u, v)^T$ and can be directly computed by $(\frac{hu}{h}, \frac{hv}{h})^T$ using the conserved quantities. The so-called flux functions $G(U)$ and $H(U)$ describe the change of the conserved quantities U over time by the possible interplay of each conserved quantities U.

By multiplying eq. (1) with a test function φ_i and applying the divergence theorem, this yields the weak form

$$\underbrace{\int_T U_t \varphi_i}_{\text{mass term}} - \underbrace{\int_T G(U) \cdot \frac{\partial \varphi_i}{\partial x} + H(U) \cdot \frac{\partial \varphi_i}{\partial y}}_{\text{stiffness term}} + \underbrace{\oint_T \mathcal{F}(U)\varphi_i \cdot \mathbf{n}}_{\text{flux term}} = 0$$

with T representing a triangular grid cell and $\mathbf{n}(x, y)$ the outward pointing normal at the boundary of the grid cell. Next, we approximate the solution U in each cell by N ansatz functions: $U(x, y, t) \approx \tilde{U}(x, y, t) = \sum_{j=1}^{N} \tilde{U}_j(t)\ \varphi_j(x, y)$. Furthermore, let $\mathcal{F}$ be a solver for discontinuity on the the nodal points used for the Lagrange reconstruction of the flux polynomial on each edge. Such a flux solver can be e.g. the Rusanov flux solver. We can then rearrange the equations to matrix-matrix and vector-matrix operations. Using an explicit Euler time stepping, this yields

$$\tilde{U}_i^{t+\Delta t} = \tilde{U}_i^t + \Delta t \mathcal{M}^{-1} \left(\mathcal{S}_x \tilde{U}(t) + \mathcal{S}_y \tilde{U}(t) + \mathcal{F}(\tilde{U}^-(t), \tilde{U}^+(t)) \right).$$

This represents the very basic implementation of the DG method, see e.g. [23] for enhanced versions.

For the benchmarks with a hybrid parallelization, we used the Rusanov flux solver [35] and a discretization based on a constant basis function on each cell support, thus a finite volume discretization.

For the simulations used in the Tsunami parameter studies, varying underwater depth (bathymetry) data has to be considered. Here, we used the computationally more intensive Augmented Riemann solver [21] and multi-resolution sampled GEBCO [9] bathymetry datasets.

We use these simulations as a realistic basis for an application scenario with varying workload. In particular for hyperbolic simulations, a changing workload leads to varying efficiency which cannot be considered with a static resource allocation.

This changing efficiency information can be provided in different ways to a resource manager which then optimizes the current resource distribution. The next Section presents such a solution of a dynamic resource allocation.

5 Realizing Invasive Computing

We first introduce our point of view on hybrid-parallelized applications (Sec. 5.1) and the challenges in the context of concurrently executed shared-memory parallelized applications (Sec. 5.2). Afterwards, a generic view on the optimization algorithm in the resource manager is given (Sec. 5.3) and the interface requirements between the resource manager and the applications to distribute

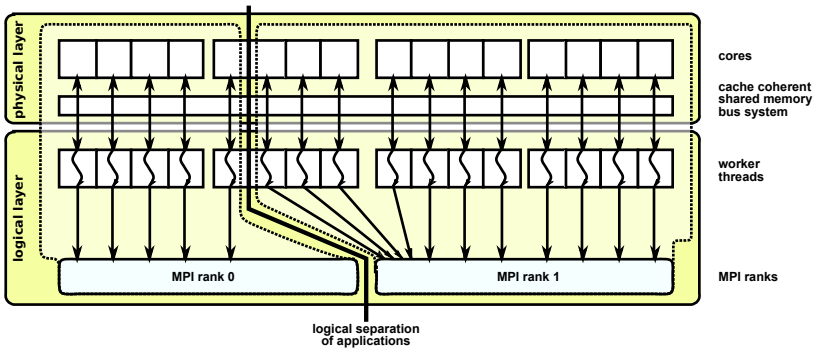

Fig. 3 Overview of the dynamical assignment of cores to ranks with hybrid parallelization. The top layer represents the physical cores available for computations with each processor providing 4 cores for computations. The bottom layer associates the physical cores with threads executing instructions and threads being dynamically assigned to MPI ranks.

computational resources to MPI ranks and other concurrently running applications are presented (Sec. 5.4). Finally, we explain a resource manager as the core component of the Invasive Computing approach which is responsible to distribute resources to applications (Sec. 5.5).

5.1 Hybrid parallelization

Invasive Computing for hybrid parallelized applications involves a mix of shared and distributed memory parallelization strategies. Fig. 3 gives an overview of the dynamically changing resource layers of a hybrid simulation on a single shared-memory HPC system. The top *physical layer* describes the physical resources including the cache-coherent bus system. For operating systems, such a cache-coherency is required internally whereas for applications pinned to cores, this is typically not required between different MPI ranks due to separated address space. The lower *logical layer* maps resources to the physical components. Each worker thread operates on a single core and in case of using MPI, one MPI rank usually contains several worker threads. The number of worker threads per MPI rank in combination with the pinning to cores is static over run time with existing standard parallelization models. This leads to compute or work-imbalances among MPI ranks due to refining and coarsening (coarse grids cause less computational load, fine grids cause more) with the compute imbalance being research of this paper. Therefore, we suggest a dynamically changing number of worker threads and dynamic pinning of threads to cores.

5.2 Concurrently running applications

Considering concurrently running applications on cache-coherent memory systems, the typical way of parallelization is accomplished with a threading library

Symbol	Description
R	Number of system-wide available computing resources
N	Number of concurrently running processes
$\mathbf{A}$	List of running applications or MPI processes
ϵ	Placeholder for "no application"
$\mathbf{C}$	State of resource assignments to applications
$\mathbf{D}_i$	Optimal resource distribution assigning D_i cores to application A_i
$\mathbf{P}_i$	Optimization information (scalability graphs, e.g.) for application i
$\mathbf{T}_i$	Optimization targets (throughput, energy, etc.) for each application
$\mathbf{G}_i$	Number of resources currently assigned to application i
$\mathbf{F}_i$	List of free resources
$\mathbf{W}_i$	Workload for application i
$T(c)$	Throughput for c cores
$S_i(c)$	Scalability graph for application i.

Table 1 Symbols representing the data structures used by the resource manager.

such as OpenMP and TBB. However, once running applications concurrently, resource conflicts can lead to a severe slowdown in performance due to frequently executed context switches on shared cores. This results in overheads induced by cache thrashing and costs of context switches. Furthermore, originally load-balanced computations suffer from load-imbalances due to computing delays introduced by the before-mentioned issues.

In the next Section, we introduce a resource manager which assigns resources dynamically to applications, focusing to avoid the beforementioned issues.

5.3 Resource manager

The resource manager (RM) itself is implemented as a separate process running in background on one of the cores utilized by the simulation software. Its responsibility is to optimize the resource distribution. This optimization is achieved by utilizing the information provided by the applications through their developers. Such information can be scalability graphs to optimize for non-linear workload-to-scalability and range-constraints requesting resources within a specific range such as "1 to 6 cores".

The communication to the RM is achieved via IPC message queues [18] due to their low overhead compared to TCP/IP or other socket-based communication. Thus, the RM provides a service bound to a particular message queue ID and each process has to subscribe to the service by a handshake protocol. With the utilization of message queues, the addresses of the processes and the RM are made unique by tagging messages to the RM with ID 0 and those to each process by the unique process id.

For sake of clarity, Table 1 contains an overview of the symbols introduced in the following. For the management of the cores, the RM uses the vector $\mathbf{C}$ with each entry representing one of the $R = |\mathbf{C}|$ physical cores. In case of core

i being assigned to a process, the process id is stored to the entry $\mathbf{C}_i$ and ϵ otherwise.

5.3.1 Scheduling information

Next, we discuss our algorithm for optimizing the resource distribution for concurrently executed applications. Let R be the number of system-wide available compute resources, N be the number of concurrently running applications, ϵ be a marker for a resource not assigned to any application and $\mathbf{A}$ be a *list of identifiers* of concurrently running applications, with $|\mathbf{A}| = N$. We then distinguish between uniquely *system-wide* and *per-application* stored management data.

System-wide data: We define the system-wide management data given by the resource assignment which is done by the RM and the optimization target such as maximizing application throughput or minimizing for energy efficiency. The current state on the resource assignment is given by

$$\mathbf{C} \in (\{\epsilon\} \cup \mathbf{A})^R,$$

uniquely assigning each compute resource to either an application $a \in \mathbf{A}$ or to none ϵ. The *optimal resource distribution* is given by

$$\mathbf{D} \in \{0, 1 \ldots, R\}^N$$

with each entry $\mathbf{D}_i$ storing the number of cores to be assigned to the i-th application $\mathbf{A}_i$. To *avoid oversubscription* of these resources to the applications, we further demand

$$\sum_i \mathbf{D}_i \leq R. \tag{2}$$

This subscription constraint avoids assignment of more resources than there are available on the system, whereas the explicit assignment of resources in an exclusive way via the vector $\mathbf{C}$ avoids *resource collision* per se. For enhanced releasing of cores, the cores currently assigned are additionally maintained in a list for each application.

Per-application data: The data $\mathbf{P}_i$ stored for each application $\mathbf{A}_i$ consists of the currently specified constraints which were sent to the RM with a (non-)blocking invade call. These constraints provide the basis for the optimizations with different optimization targets available and discussed in Section 5.3.3.

5.3.2 Optimization loop

After its setup, the RM processes messages from applications in a loop. Updates of resource distributions are then based on messages processed in a way optimizing the *current resource distribution* $\mathbf{C}$ towards the optimal *target resource distribution* $\mathbf{D}$. We can separate the optimization loop into the following three parts:

– *Computing target resource distribution* $\mathbf{D}$*:* For setup, shutdown and in particular invade messages, new parameters for computing the target resource distribution are handed over to the RM via the constraints. This triggers execution of the optimization function, in its general form given by

$$(\mathbf{D}^{(i+1)}, \mathbf{C}^{(i+1)}) := f_{optimize}(\mathbf{D}^{(i)}, \mathbf{C}^{(i)}, \mathbf{P}, \mathbf{T}) \tag{3}$$

with $\mathbf{T}$ being a vector of optimization targets for each application such as improved throughput or load distribution, $\mathbf{P}$ the application constraints, the current core-to-application distribution $\mathbf{C}^{(i)}$ and the optimizing function $f_{optimize}$ as input parameters. $f_{optimize}$ computes the quantitative target resource distribution $\mathbf{D}^{(i+1)}$ to which the computing cores have to be updated to. The superscript (i) annotates the i-th execution of the optimization function in the RM.
In its generic form, also the core-to-application assignment is returned in $\mathbf{C}^{(i+1)}$. We expect this to get beneficial in case of accounting for non-uniform memory access (NUMA) prone applications and mandatory, once extending the RM to distributed memory systems. So far, this direct core-to-application assignment is not considered in computation of the target resource distribution and we continue solely using the quantitative optimization given in $\mathbf{D}^{(i+1)}$.
– *Optimizing current resource distribution* $\mathbf{C}$*:* With an optimized resource distribution $\mathbf{D}^{(i+1)}$ at hand, the current resource distribution in $\mathbf{C}$ has to be successively updated. During this resource reassignment, resources can be assumed to be only immediately releasable under special circumstances. Such circumstances are e.g. that the cores have to be released for the application for which the optimization process with a blocking (re)invade call is currently processed. Otherwise, the RM has to send a message with a new resource distribution to an application. Only as soon as the application replies with its updated resource distribution, these resources can be assumed to be released. Hence, *resources cannot be assumed to be directly released in general* during the optimization process inside the RM, e.g. after sending a release message. This results in a delay in resource reassignment, hence *idling time*, which has to be compensated by the benefits of core-migration.
The *resource redistribution step* then iterates over the list $\mathbf{A}$ of applications. For each application $\mathbf{A}_i$, either the resources stay unmodified, are released or assigned from or to the application. Let $\mathbf{G}_i := |\{j|\mathbf{A}_i = \mathbf{C}_j, \forall j \in \{1, \ldots, R\}\}|$ be the number of resources currently assigned to application $\mathbf{A}_i$ and a list of free resources $\mathbf{F}$ with $\mathbf{C}_{\mathbf{F}_j} = \epsilon$. The redistribution process then iterates over all applications:
  – $\mathbf{G}_i = \mathbf{D}_i$: No update
    The number of resources assigned to an application equals the currently assigned resources. Therefore, there are no resources to update for this application.
  – $\mathbf{D}_i < \mathbf{G}_i$: Release resources
    In case of *less resources* to be assigned to the application, a message

is sent to the application. This message is either send directly to the application in case of a non-blocking communication or as a response message to a blocking (re)invade call. In every case, the message includes only a shrank set of resources, $\mathbf{G}_i - \mathbf{D}_i$ cores less than currently assigned to the application.
Note that the current resource distribution $\mathbf{C}$ is not updated yet. Otherwise, those resources could be assigned to other applications, leading to resource conflicts.

- $\mathbf{D}_i > \mathbf{G}_i$: Add resources
  Assignment of *additional resources* is accomplished by searching for resources in the list of free resources and assigning up to $k \leq \mathbf{D}_i - \mathbf{G}_i$ of them to the application with

$$\forall j \in \{\mathbf{F}_1, \ldots, \mathbf{F}_k\} : \mathbf{C}_j := \mathbf{A}_i.$$

- *Client-side resource update messages:* As soon as the change of resource utilization is assured, e.g. by an application responding with the resources currently used, the RM tests for further optimizations in case of released resources. We then apply the same operations for the standard optimization of the resource distribution $\mathbf{C}$ since this also accounts for assigning recently released resources by adding resources in case of $\mathbf{D}_i > \mathbf{G}_i$.

5.3.3 Scheduling decisions

We use the previously introduced data structures to compute our optimized target resource distribution $\mathbf{D}$ depending on the specified optimization target $\mathbf{T}$ and per-application specified information $\mathbf{P}$.

Recapitulating our original optimization function (3), we drop the core dependencies $\mathbf{C}$ yielding an optimization function

$$\mathbf{D}^{(i+1)} := f_{optimize}(\mathbf{D}^{(i)}, \mathbf{P}, \mathbf{T}) \tag{4}$$

with a reduced set of parameters. We then apply optimizations based on the constraints given for all applications in $\mathbf{P}$ and depending on the optimization target $\mathbf{T}$.

Requirements on constraints: Resource-aware applications are expected to forward information on their state as well as their requirements via constraints to the RM, which keeps this information in $\mathbf{P}$. Depending on the optimization target $\mathbf{T}$, the RM then schedules resources based on these constraints. We further distinguish between *local* and *global constraints*, respectively, depending on their capability of optimization per application or for all applications.

Local constraints: With constraints such as a range of cores, an application can always request between 1 and the maximum number of cores available on the system. Such constraints do not yield a way of adopting the application's resources under consideration of other concurrently running applications without knowledge on the state (FLOP/s, throughput, etc.) of these applications. Therefore we refer to such constraints as local ones.

Global constraints: With global constraints, we refer to constraints to be evaluated by the optimization function leading to a global cooperative way:

- *Application's workload*: Under the assumption of similar applications, load balancing related values such as the *workload* can be used to schedule resources. Using this constraint, our target function then distributes R compute resources to N applications with workload $\mathbf{W}_i$ for each application i:

$$\mathbf{D}_i := \left\lceil \frac{R \cdot \mathbf{W}_i}{\sum_j \mathbf{W}_j} \right\rceil - \alpha_i, \quad \alpha_i \in \{0, 1\}$$

  This assigns $\mathbf{D}_i$ resources to application $\mathbf{A}_i$. α has to be chosen in a way to avoid over-subscription (see Eq. (2)).
  Only considering the assigned resources $\mathbf{D}_i$, we take a different point of view leading to alternative global scheduling: Each application has a perfect strong scalability $S(c)$ for c cores within the range $[1; \mathbf{D}_i]$: $S(c) := \min(c, \mathbf{D}_i)$. The cores are then assigned to the applications until their scalability does not yield any performance improvement. Obviously, such a strong scalability graph represents only an approximation of the real scalability graph which we discuss next:
- *Application's scalability graph*: We consider applications messaging *strong scalability graphs* to the RM. Such scalability graphs are linearly dependent on the application's workload throughput via the strong scalability:
  We compare the throughput $T(c)$ depending on the number of cores c. With the throughput for a number of cores given by the fraction of the time taken to compute a solution and the fixed problem size $w = \mathbf{W}_i$, we compute the throughput improvement with the baseline set at the throughput with a single core:

$$\frac{\frac{w}{T(c)}}{\frac{w}{T(1)}} = \frac{T(1)}{T(c)} =: S(c)$$

  yielding the scalability graph $S(c)$. Therefore this *justifies relating the scalability graph to the application's throughput.* Moreover, a scalability graph also yields a way to optimize for throughput for different application types due to the normalization $S(1) = 1$ for a single core.
  Given a scalability graph $S_i(c)$ with subindex i for the i-th application, we further demand each graph to be monotonously increasing $S_i(c) - S_i(c-1) \geq 0$, and concave $S_i(c+1) - S_i(c) \leq S_i(c) - S_i(c-1)$, thus assuming no super-linear speedups with the concavity property.
  We can then search for combinations in $\mathbf{D}$ maximizing the global throughput by formulating our optimization target as a maximization problem: $\max_{\mathbf{D}} \left(\sum_i S_i(\mathbf{D}_j)\right)$ with the side constraint avoiding over-subscription $\sum_j \mathbf{D}_j \leq R$. Hence, we get a multivariate optimization problem with $\mathbf{D}_j$ the search space.
  A sketch of this optimization is given in Fig. 4. Here, we consider two applications, each one providing a scalability graph. The scalability graph for the first application is given with increasing number of resources (red solid

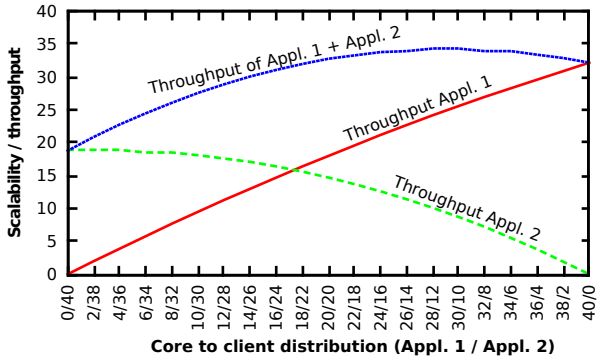

Fig. 4 Examples for scalability graphs: The scalability graph for the first application is given with increasing number of cores from left to right and for the second application vice versa. The global throughput is given with different assignments of all cores to both applications with the maximum throughput our optimization target.

line) and the second scalability graph (greed dashed line) with increasing numbers of resources from right to left. Then the theoretical optimal resource distribution is given by the global maximum of the sum of normalized throughput of both applications (blue dotted line) for different valid resource constellations.
Using our assumptions of strictly monotonously increasing and concave scalability graphs, we can solve this maximization problem for more than two applications with an iterative method similar to the steepest descent solver [19]:

Initialization: We introduce the iteration vector $\mathbf{B}^{(k)}$ assigning $\mathbf{B}_i$ computing cores to application i in the k-th iteration. Since each application requires at least one core to continue its execution, we start with $\mathbf{B}^{(0)} := (1, 1, \ldots, 1)$, assigning each application a single core at the start.

Iteration: With our optimization target aiming for improved application throughput, we then compute the *throughput improvement* for application i if a single core is additionally assigned to this application

$$\Delta S_i := S_i(\mathbf{B}_i + 1) - S_i(\mathbf{B}_i). \tag{5}$$

and determine the application n, which yields the maximum throughput improvement $\Delta S_n := \max_j \{\Delta S_j\}$.
The resource distribution is then updated by

$$\mathbf{B}_i^{(k+1)} := \mathbf{B}_i^{(k)} + \delta_{i,n} \tag{6}$$

with the Kronecker delta δ.

Stopping criterion: We stop the iterative process, as soon as all resources are distributed, thus if $\sum_i \mathbf{B}_i^{(k)} = R$. The target resource distribution $\mathbf{D}^{(k+1)}$ is then given by the last iteration vector $\mathbf{B}$.

5.4 Resource manager and hybrid-parallelized applications interplay

In order to apply Invasive Computing for applications with hybrid parallelization, two additional extensions have been realized: (a) an extension to the RM to start it on the first MPI rank with appropriate synchronizations for contacting the RM by the other MPI ranks and (b) a dead-lock free implementation due to intermixing the communication of the RM with MPI synchronization barriers with the dead-lock free implementation discussed next.

We start with an example of such a deadlock which can occur during the initialization phase of Invasive Computing for a better understanding of the challenges of core migration: We assume (without loss of generality) only two MPI ranks being executed in parallel. All cores are initially assigned to the first MPI rank which starts the computations in parallel whereas the second rank is waiting for resources. This waiting is due to avoidance of oversubscription of resources, see Eq. (2). During the computations, an MPI reduce operation is executed - e.g. to compute the maximum allowed time step width. However, the second MPI rank is not allowed to start any computations since all resources are already reserved by the first MPI rank which is executing the barrier. Due to the blocking barrier, this MPI rank is not able to free resources and make them available for other MPI ranks to call the barrier. The solution is given by *non-blocking invasive requests* during the setup phase. These non-blocking invased are executed until at least one computational resource is assigned to each MPI rank, implemented with an MPI reduce.

We conclude that non-blocking invasive interfaces during the setup phase are *mandatory for Invasive Computing for our hybrid parallelized applications.* The simulation loop itself can be executed deadlock-free with blocking or non-blocking invasive commands. With the (MPI-)setup phase being provided by the invasive framework layer, our deadlock-free initial resource assignment is hidden from the application developer.

5.5 Owning computation resources

The pinning of threads to cores is frequently used in HPC to assure locality of computation cores to data on the memory hierarchy. With OpenMP, changing the number of cores is only available out of the scope of a parallel region. Also with TBB [32], the task scheduler has to be deleted and reallocated to change the number of cores used for our simulation. However, dynamically changing and pinning of computational resources during a simulation are not considered by current standard threading libraries such as OpenMP and TBB.

We extended OpenMP and TBB to allow for changing the number of active threads and their pinning during runtime and continue describing one of our approaches for TBB. Before the setup phase of the simulation, as many threads as cores are available on the system are started. A list of mutices with each mutex assigned to one of the available threads is used to enable and disable threads of doing work stealing with work stealing initially disabled. Then, for all but the first master thread, tasks are enqueued with setting affinities to the corresponding threads requesting a lock to one of the mutices. Clearly, no spin-lock may be used in this circumstances since the thread really has to idle to make it available to other applications. Otherwise this would lead to resource conflicts as discussed in Sec. 5.2.

For requesting a different number of resources, we distinguish between an increase or a decrease in the requested number of cores. If the number of cores has to be increased, work-stealer tasks can be enabled directly by unlocking the corresponding mutex. For decreasing the number of cores, tasks requesting the mutex are enqueued, leading to worker-threads with an idling state.

We are now able to change the number of cores used for running computations for each program context. To consider the memory hierarchy, we describe our pinning method of threads to cores, which avoids memory conflicts. Without pinning, our invasive applications would not be able to work cooperative with other applications due to violating exclusive resource agreements. The information on which cores an application should run on is dynamically given to the application during run time. After changing the number of resources, we set the thread-to-core affinities by enqueuing tasks, which set affinities of the currently executing thread to the desired core. This assures that threads, which continue running computations on particular cores, don't conflict with others.

To improve the programmability for MPI parallelized applications, the RM is started on the first rank and running in background on an additional thread. Contrary to the exclusive pinning of threads to cores for the application's threads, we rely on the operating system scheduler to use pre-emptive scheduling for the RM.

6 Results

All experiments presented in this section were conducted on an Intel Westmere EX machine with 4 Intel Xeon CPUs (E7-4850@2.00GHz) and 256 GB memory totally available on the platform. This gives 4×10 physical cores plus 4×10 additional hyper threading cores, with the latter ones not used during the benchmarks.

All the benchmarks used in this section use a changing workload in each memory context, hence leading to the requirements of coping with the load imbalances induced by the changing workload. While the non-invasive benchmarks do not use core-migration between each time steps, the invasive benchmarks allow changing the thread-to-core assignment between time steps if

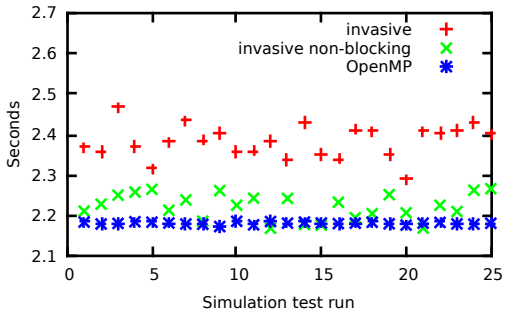

Fig. 5 Invasive message passing and processing overheads: 25 identical simulation runs execute on a single MPI rank. We compare a purely threaded simulation, a non-blocking invasive execution and a blocking execution.

triggered by the resource manager. Regarding the programmability issues of data migration, we purely focus on core migration only and do not consider data migration approaches.

For sake of reproducibility of our results, the source code is released at `http://www5.in.tum.de/sierpinski/`, `http://www.martin-schreiber.info/sierpinski/` (mirror) and `https://github.com/schreibm/ipmo`

6.1 Micro benchmark for invasive message passing and processing

Invasive execution of our applications typically involves a message passing to the resource manager. This leads to overheads due to the message passing and the response latency. We measure this overhead with a micro benchmark based on a very small single-process SWE scenario (with a regularly refined spacetree resulting in 128 grid cells), ignoring the influence of other applications. The size of this setup is just large enough to obtain significantly measurable times for communication overheads.

The tests were conducted in three different variants: (a) a *pure threaded* execution, (b) a scenario sending requests to *resource manager in a blocking way*, thus waiting and forcing cores to idle until the resource manager responds, and (c) invasive requests to the *resource manager using a non-blocking communication*. Such a non-blocking communication sends new requirements to the resource manager, tests for and processes resource-update messages from the resource manager and immediately continues the simulation in case of messages left in the message queue to the resource manager. Results for multiple runs of identical scenarios are given in Figure 5. These micro benchmark scenarios show in general small overheads of Invasive Computing due to the communication with the resource manager. Compared to the non-invasive execution, the blocking communication to the resource manager leads to additional and scattered overhead of up to 15%. With non-blocking communication, the maximum

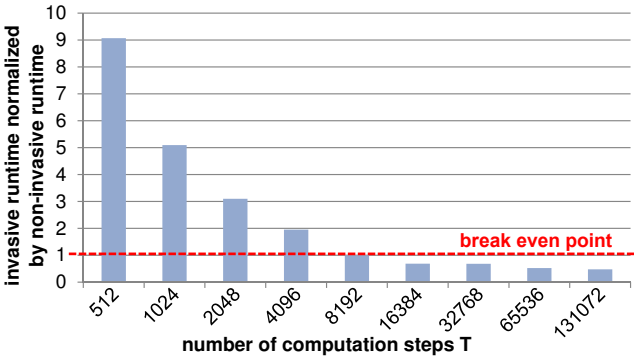

Fig. 6 Invasive vs. non-invasive execution of the artificial benchmark. For different problem sizes (e.g. representing different initial grid refinement) depending on T, the time of the invasive execution is given normalized to the non-invasive run time.

overhead and its scattering is reduced to 5% providing a robust improvement for invasive executions.

6.2 Artificial hybrid-parallelized load-imbalance benchmark

Our next benchmark suite for invasive executions involves more than a single program context and is based on artificial workload executed with two MPI ranks to show the general applicability, but also restrictions of our approach.

Let r be the MPI rank of a single simulation run. We execute our test application with artificial load simulating an application with T computation steps and each computation step denoted with $t \in [0; T-1]$. Then, the workload for rank r at computation step t is given with $L_r(t)$. For our artificial test case executed on two MPI ranks, we chose linear functions creating the workloads $L_0(t) := T - t$ for rank 0 and for rank 1 $L_1(t) := t$.

Our artificial workload is simulated by L_r^2 square roots computed for each rank and for each simulation time step. Thus, MPI rank 0 starts with a workload of 0 quadratically increased to T and MPI rank 1 vice versa. This artificial workload represents the changing number of grid cells and a barrier is executed after each artificial workload L_r^2 e.g. to account for similar parallel communication pattern of hyperbolic simulations.

Our artificial load imbalance benchmark scenarios are conducted for different numbers of time-steps T and, thus, workload sizes. We compare the benchmark setup for our invasive implementation, which allows compute balancing via the RM, with the non-invasive counterpart. This assigns the resources equally distributed to all MPI ranks at simulation start. All other simulation parameters (such as adaptivity by refining and coarsening) are identical in both variants. For a better comparison, we use the ratio of the time for the invasive execution $\mathcal{T}_{inv}$ to the time for the non-invasive execution $\mathcal{T}_{default}$.

The results are given in Fig. 6 with the break-even point at 1 representing the normalized run time of the non-invasive application. For small prob-

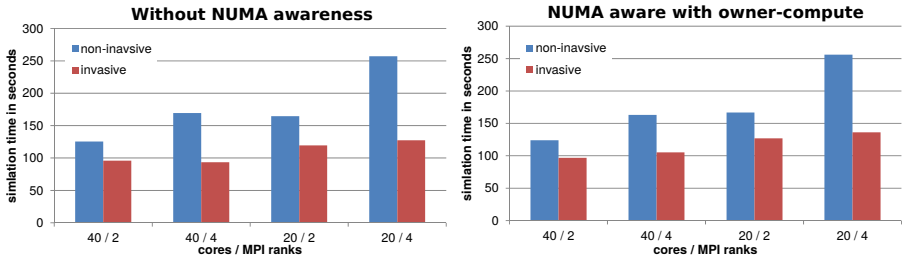

Fig. 7 Invasive vs. non-invasive execution times with default TBB affinities for different combinations of number of cores and MPI ranks. See text for further information on NUMA aware execution.

lem sizes, Invasive Computing has a clear overhead compared to the non-invasive execution. However, this soon improves starting with still relatively small problem sizes with $T = 8192$ and remains a robust optimization for larger problem sizes. Comparing the run times for our largest test simulation ($\mathcal{T}_{default} = 6057.24$, $\mathcal{T}_{inv} = 2870.82$), the run time was improved by 53%. Therefore, the realization of the invasive paradigm, even though including the overheads determined with the micro benchmarks, really pays off.

To discuss the applicability of our results to other simulations, we consider the threshold of the break-even point at the relatively small workload of 8192 taking 8.43 seconds for the non-invasive and 8.39 seconds for the invasive execution. Our simulations with dynamically changing resource requirements typically yield larger workloads, hence we expect robust performance improvements for typical DAMR simulations with similar workload changes.

6.3 Shallow water simulation benchmark

For our simulation based on a dynamically adaptive mesh refinement, we used the shallow water equations explained in Sec. 4. The scenarios are conducted with an initial refinement depth of 14, thus creating $(2 \times 2)^{14}$ bisection-generated initial grid-cells for a triangulated domain setup by two initial triangles and a relative refinement depth of 10. The domain was initially split up along the diagonals assigning the computations for each quarter to an MPI rank. This assignment to MPI ranks is kept over the entire simulation run time, thus without data migration, with each MPI rank being able to split its subregion to improve local load balancing by massive splitting in combination with threaded parallelization as proposed in [38]. The adaptivity criterion was chosen to refine and coarsen based on the relative water-surface displacement to the horizon.

The results for the benchmark are given in Fig. 7. A robust improvement of simulation run time for all different utilized cores and MPI ranks can be observed. For the scenario using 20 cores and 4 MPI ranks, an increased load imbalance can be shown. The computational efficiency of this scenario was

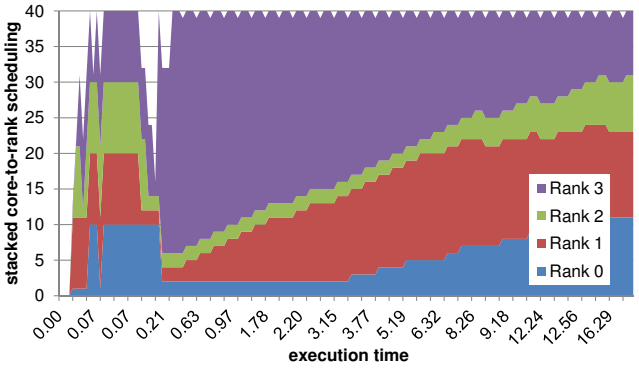

Fig. 8 Overview of core-to-MPI rank distribution. The cores are given in a stacked representation depending on the real time in which the scheduler is reassigning the resources.

mostly improved by invasive core migration whereas for the non-invasive scenario, the lost computation time was due to idling cores.

Due to different results depending on using owner-compute for scheduling on simulations with a longer run time [39], we decided to run additional simulations with owner-compute cluster scheduling to improve awareness of NUMA effects: instead of generating a task for each cluster, we assign one or more clusters to a thread. The results for a simulation with the owner-compute scheme enabled are visualized in the right image in Fig. 7 with a performance similar to the default simulation.

The owner-compute simulations are even slightly slower compared to the default work stealing - in particular the 40/4 combination. We account for this by NUMA effects of the underlying memory architecture: due to the changing core distribution, core-to-NUMA domain relations are frequently changing whereas our owner-compute scheduling aims to compensate NUMA effects under the assumption of a core-to-cluster locality over time. However, the owner-compute scheme does not allow for work-stealing, e.g. to compensate dynamic resource distributions introducing additional NUMA effects, therefore leading to longer run time. We emphasize, that this is contrary to the *results obtained for the static resource assignment* with the owner-compute scheme yielding improved runtime compared to the work stealing [39].

An overview of the scheduling is given in Fig. 8. We executed our simulation with a severely reduced problem size in order to get a better survey on the distribution of the workload: the initial depth was set to 10, the adaptive depth leading to the dynamical grid to 8 and the simulation was executed for 1000 time steps taking 19.3 seconds to compute the simulation. A radial breaking dam is initialized with the gravitation-induced wave leading to a dynamically adaptive grid refinement, see Fig. 1 for an example. The first phase between 0 to 0.2 seconds is used for the setup. Afterwards, the simulation itself is executed, starting at 0.2 seconds. Rank 3 was initially assigned the most computation cores which is due to the initial radial dam break created mainly in the grid assigned to this rank, leading to a severely higher workload at rank 3. During

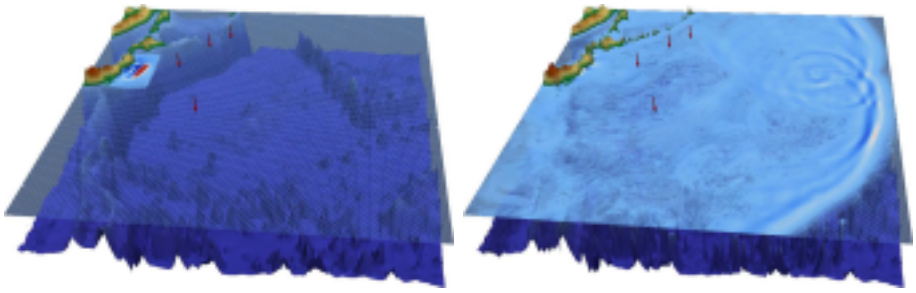

Fig. 9 Screenshots with visualization of the dynamic adaptive grid underlying to a 2011 Tohoku Tsunami simulation used for the Tsunami parameter study benchmark. We use a highly-refined grid at the wave front and a coarser grid in the other areas.

wave propagation, the workload in rank 1 dominates successively and, thus, obtains more and more resources from rank 3. At the end of the simulation run, the grid resolution and, hence, workload for rank 2 is increased relatively to other ranks, and it consequently reassigns resources from other ranks. This dynamic resource distribution fits to the underlying dynamic adaptive mesh refinement.

Without synchronization for core reassignment, idling processing time is automatically introduced during the scheduling of taking away a core to initialize migrating this core to another MPI rank. This overhead is visible at the top of the graph by the small white gaps. Despite this core-scheduling-idling overhead, the overall reduced idling time due to compute imbalances is severely reduced, thus improving the applications efficiency.

6.4 Tsunami parameter study benchmark

The final benchmark suite using several concurrently executed Tsunami simulations represents an example of a parameter study for the 2011 Tohoku Tsunami event. The simulation first loads the bathymetry data [9], preprocesses it to a multi-resolution format and then sets up the initial simulation grid iteratively by inserting edges close to the earthquake-generated displacement data [41], see Fig. 9. Such parameter studies are e.g. required to identify adaptivity parameters such as the net-flux crossing each edges and/or the minimum and maximum refinement depth.

The resource optimization constraint we used in these studies is based on the workload, e.g. current number of cells, in each parameter study.

We compare three different ways to execute such parameter studies:

– *Invasive Computing*:
  The application is started as soon as it is enqueued. A short period after its start, the application is waiting until a message from the RM provides at least a single resource. This is important to avoid any conflicts with other concurrently running applications.

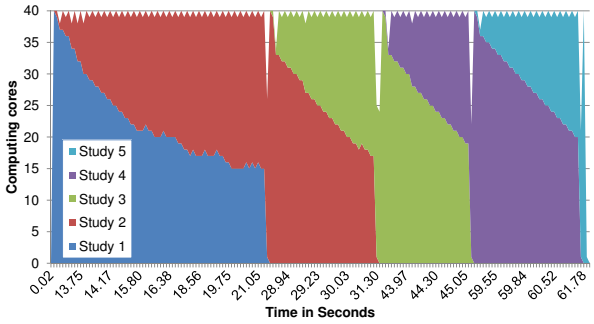

Fig. 10 Visualization of a typical resource redistribution to invasified applications executed with *different parameters* and started at *different points in time*. The spikes at the top of the resource distribution represent the idle time of computing cores until they are rescheduled to another application.

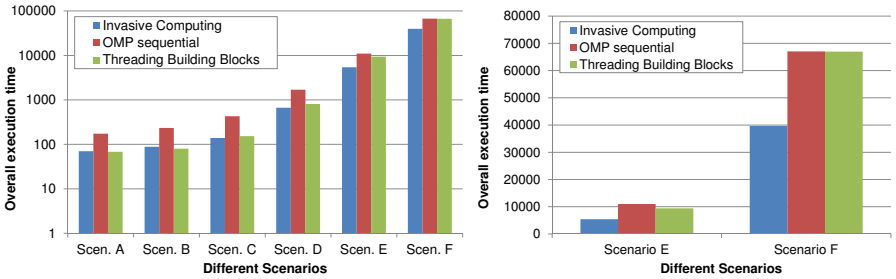

Fig. 11 Left image: The problem size was successively increased per execution scenario (left to right) via the adaptivity parameters. Comparing our Invasive Computing approach with scenarios using typical OpenMP and TBB parallelized applications, Invasive Computing results in a robust optimization for larger scenarios. Right image: Scenarios E and F with linear scaling for improved comparison.

– *OpenMP sequential*:
  Using OpenMP scheduling, starting each application directly after its enqueuing would result in resource conflicts, and thus severely slowing down the execution. Therefore, this execution policy starts the enqueued application only if the execution of the previous applications was finished.
– *Threading Building Blocks (TBB)*:
  Using TBB, we start each application as soon as it is enqueued to our system. TBB has features which automatically circumvent resource conflicts in case of idling resources, hence setting up a perfect baseline to compare our Invasive approach with another optimization method.

A typical dynamic resource distribution with Invasive Computing for such parameter studies is given in Fig. 10. We successively increase the problem size for the scenarios by increasing the maximum allowed refinement-depth parameter, resulting in higher workload for each scenario. Five applications with slightly different adaptivity parameters are enqueued to the system with a delay of a few seconds. Such a delay represents a more realistic, user-driven-

like enqueuing of applications to the computing system compared to starting all applications at once.

The results of the three approaches for different parameter studies are given in Fig. 11 with increasing workload from left (scenario A) to right (scenario F). The OpenMP sequential execution always yields the longest run time. For smaller workloads, the TBB implementation is competitive to the Invasive Computing approach whereas for larger workloads, it approaches the longer OpenMP sequential run times. Regarding our Invasive Computing approach, the costs for rescheduling resources are compensated for larger simulations (scenario C to F). For the scenarios E and F with a larger workload, the invasive execution of such larger problem sizes yields an optimization of 45%.

For our simulations, the effects of the underlying NUMA architectures did not have a significant impact on the performance. We account for this e.g. by the computational intensive solvers, the cache-optimized grid traversal and efficient stack-based data exchange as well as the underlying hardware which, in case of accessing a non-local NUMA domain, requires only a single HOP. We expect higher impact on latencies on larger systems with more than 1K cores, e.g. with NUMA domain data access requiring page-wise migration.

7 Conclusions and future work

We presented a new approach in the context of Invasive Computing as performance improving solution for (a) dynamically changing resource requirements of concurrently running applications on cache-coherent shared-memory systems and for (b) load imbalances for applications with hybrid parallelization. Due to the clear interfaces and easy extension of applications with this compute-balancing strategy, this leads to improved programmability while accounting for changing resource requirements and load imbalances.

We conducted experiments based on four different benchmarks on a cache-coherent shared-memory HPC system. The results show robust improvements in performance for realistic PDE simulations executed on NUMA domains with hybrid parallelization. With the efficiency of Tsunami parameter studies considerably improved by 45%, the Invasive Computing approach is very appealing for concurrently executed applications with changing resource demands and time-delayed points of execution.

The presented optimizations with Invasive Computing are not only applicable to DAMR simulations but can also be applied to other applications with dynamically changing resource requirements in general.

Our current work is on systems with run time configurable cache-coherency protocols and cache-levels [43] to further enhance the performance and programmability of parallel applications on such systems. With our simulations conducted on an HPC shared-memory NUMA domain which requires only 1 additional hop to each domain, thus hiding the NUMA domain effects very efficiently, we expect that evaluation of the invasive concepts leads to additional requirements on larger scale NUMA domains.

Acknowledgements

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. Aizinger V (2002) A discontinuous Galerkin method for two-dimensional flow and transport in shallow water. Advances in Water Resources
2. Al Faruque MA, Krist R, Henkel J (2008) ADAM: run-time agent-based distributed application mapping for on-chip communication. In: Proceedings of the 45th annual Design Automation Conference, ACM, New York, NY, USA, DAC '08, pp 760–765
3. Bader M, Breuer A, Schreiber M (2012) Parallel Fully Adaptive Tsunami Simulations. In: Facing the Multicore-Challenge III, Institut für Informatik, Technische Universität München, Springer, Heidelberg, Germany, Lecture Notes in Computer Science, vol 7686
4. Bader M, Bungartz HJ, Schreiber M (2012) Invasive Computing on High Performance Shared Memory Systems. In: Facing the Multicore-Challenge III, Springer, Lecture Notes in Computer Science, vol 7686, pp 1–12
5. Bangerth W, Hartmann R, Kanschat G (2007) deal.II – a General Purpose Object Oriented Finite Element Library. ACM Trans Math Softw
6. Becchi M, Crowley P (2006) Dynamic thread assignment on heterogeneous multiprocessor architectures. In: Proceedings of the 3rd conference on Computing frontiers, ACM, New York, NY, USA, CF '06, pp 29–40
7. Behrens J (2012) Efficiency for Adaptive Triangular Meshes: Key Issues of Future Approaches. In: Earth System Modelling-Volume 2, Springer
8. Bhadauria M, McKee S (2010) An approach to resource-aware co-scheduling for CMPs. In: Proceedings of the 24th ACM International Conference on Supercomputing, ACM, ICS '10, pp 189–199
9. BODC (2013) Centenary Edition of the GEBCO Digital Atlas
10. Bolosky WJ, Scott ML (1993) False sharing and its e ect on shared memory performance. In: 4th Symposium on Experimental Distributed and Multiprocessor Systems, pp 57–71
11. Burstedde C, Wilcox LC, Ghattas O (2011) `p4est`: Scalable Algorithms for Parallel Adaptive Mesh Refinement on Forests of Octrees. SIAM Journal on Scientific Comp 33(3):1103–1133, DOI 10.1137/100791634
12. Castro C, Käser M, Toro E (2009) Space–time adaptive numerical methods for geophysical applications. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences
13. Corbalán J, Martorell X, Labarta J (2000) Performance-driven processor allocation. In: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4, pp 5–5

14. Corbalan J, Martorell X, Labarta J (2005) Performance-driven processor allocation. Parallel and Distributed Systems, IEEE Transactions on 16(7):599–611
15. De Grande R, Boukerche A (2011) Dynamic load redistribution based on migration latency analysis for distributed virtual simulations. In: Haptic Audio Visual Environments and Games (HAVE), 2011 IEEE International Workshop on, pp 88–93, DOI 10.1109/HAVE.2011.6088397
16. Drosinos N, Koziris N (2004) Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters. In: Parallel and Distributed Processing Symposium, 2004, IEEE
17. Falby JS, Zyda MJ, Pratt DR, Mackey RL (1993) NPSNET: Hierarchical data structures for real-time three-dimensional visual simulation. Computers & Graphics 17(1):65–69
18. Fleisch BD (1986) Distributed system V IPC in LOCUS: a design and implementation retrospective. In: ACM SIGCOMM Computer Communication Review, ACM, vol 16, pp 386–396
19. Fletcher R, Powell MJ (1963) A rapidly convergent descent method for minimization. The Computer Journal 6(2):163–168
20. Garcia M, Corbalan J, Badia Maria R, Labarta J (2012) A dynamic load balancing approach with SMPSuperscalar and MPI. In: Keller R, Kramer D, Weiss JP (eds) Facing the Multicore-Challenge II
21. George D (2008) Augmented Riemann solvers for the shallow water equations over variable topography with steady states and inundation. Journal of Computational Physics 227(6):3089–3113
22. Gerndt M, Hollmann A, Meyer M, Schreiber M, Weidendorfer J (2012) Invasive computing with iOMP. In: Specification and Design Languages (FDL)
23. Hesthaven JS, Warburton T (2008) Nodal discontinuous Galerkin methods: algorithms, analysis, and applications. Springer
24. Hsieh WCY (1995) Dynamic Computation Migration in Distributed Shared Memory Systems. PhD thesis, MIT
25. Keyes DE (2000) Four horizons for enhancing the performance of parallel simulations based on partial differential equations. In: Euro-Par 2000 Parallel Processing, Springer, pp 1–17
26. Kobbe S, Bauer L, Lohmann D, Schröder-Preikschat W, Henkel J (2011) DistRM: Distributed resource management for on-chip many-core systems. In: Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, ACM, pp 119–128
27. Li D, De Supinski B, Schulz M, Cameron K, Nikolopoulos D (April) Hybrid MPI/OpenMP power-aware computing. In: Parallel Distributed Processing (IPDPS), 2010, pp 1–12
28. Meister O, Rahnema K, Bader M (2011) A Software Concept for Cache-Efficient Simulation on Dynamically Adaptive Structured Triangular Grids. In: PARCO, pp 251–260
29. Michael MM (2004) Scalable lock-free dynamic memory allocation. In: ACM SIGPLAN Notices, ACM, vol 39, pp 35–46

30. Neckel T (2009) The PDE Framework Peano: An Environment for Efficient Flow Simulations. Dissertation, Institut für Informatik, Technische Universität München
31. Nogina S, Unterweger K, Weinzierl T (2012) Autotuning of Adaptive Mesh Refinement PDE Solvers on Shared Memory Architectures. In: PPAM 2011, Springer-Verlag, Heidelberg, Berlin, Lecture Notes in Computer Science, vol 7203, pp 671–680
32. Reinders J (2010) Intel threading building blocks: outfitting C++ for multi-core processor parallelism. O'Reilly Media, Inc.
33. Rosu D, Schwan K, Yalamanchili S, Jha R (1997) On adaptive resource allocation for complex real-time applications. In: Proceedings of the 18th IEEE Real-Time Systems Symposium, IEEE Computer Society, Washington, DC, USA, RTSS '97, pp 320–, DOI 0-8186-8268-X
34. Rüde U (1993) Fully adaptive multigrid methods. SIAM Journal on Numerical Analysis 30(1):230–248
35. Rusanov VV (1962) Calculation of interaction of non-steady shock waves with obstacles. NRC, Division of Mechanical Engineering
36. Sagan H (1994) Space-filling curves, vol 18. Springer-Verlag New York
37. Schmidl D, Cramer T, Wienke S, Terboven C, Müller M (2013) Assessing the performance of openmp programs on the intel xeon phi. In: Wolf F, Mohr B, Mey D (eds) Euro-Par 2013 Parallel Processing, Lecture Notes in Computer Science, vol 8097, Springer Berlin Heidelberg, pp 547–558
38. Schreiber M, Bungartz HJ, Bader M (2012) Shared Memory Parallelization of Fully-Adaptive Simulations Using a Dynamic Tree-Split and -Join Approach. IEEE International Conference on High Performance Computing (HiPC), IEEE Xplore, Puna, India
39. Schreiber M, Weinzierl T, Bungartz HJ (2013) Cluster optimization of parallel simulations with dynamically adaptive grids. In: EuroPar 2013, Aachen, Germany
40. Schreiber M, Weinzierl T, Bungartz HJ (2013) SFC-based Communication Metadata Encoding for Adaptive Mesh. In: Proceedings of the International Conference on Parallel Computing (ParCo)
41. Shao G, Li X, Ji C, Maeda T (2011) Focal mechanism and slip history of the 2011 Mw 9.1 off the Pacific coast of Tohoku Earthquake, constrained with teleseismic body and surface waves. Earth, planets and space 63(7)
42. Teich J, Henkel J, Herkersdorf A, Schmitt-Landsiedel D, Schröder-Preikschat W, Snelting G (2011) Invasive computing: An overview. In: Multiprocessor SoC, Springer, pp 241–268
43. Tradowsky C, Schreiber M, Vesper M, Domladovec I, Braun M, Bungartz HJ, Becker J (2014) Towards Dynamic Cache and Bandwidth Invasion. Springer
44. Vigh CA (2012) Parallel Simulations of the Shallow Water Equations on Structured Dynamically Adaptive Triangular Grids. Dissertation, Institut für Informatik, Technische Universität München
45. Vuchener C, Esnard A (2012) Dynamic Load-Balancing with Variable Number of Processors based on Graph Repartitioning. In: Proceedings

of High Performance Computing (HiPC 2012), pp 1–9
46. Weinzierl T (2009) A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids. Dissertation, Institut für Informatik, Technische Universität München, München