A Case Study for a New Invasive Extension of Intel's Threading Building Blocks *

Martin Schreiber
Department of Computer Science / Mathematics,
University of Exeter
EX4 4QF Exeter, Great Britain
M.Schreiber@exeter.ac.uk

Tobias Weinzierl†
Department of Computer Science, Durham University
DH1 3LE Durham, Great Britain
tobias.weinzierl@durham.ac.uk

ABSTRACT

We study codes deploying multiple MPI ranks to one node where each rank is parallelised with TBB. A static assignment of cores to ranks here is disadvantageous if the load is not perfectly balanced, the runtime is subject to fluctuations or one MPI rank runs through phases with low concurrency. We propose an extension to TBB where developers manually annotate which code parts could exploit further cores. The cores are then dynamically associated with ranks. Our approach is decentralised, lightweight and minimally invasive w.r.t. code modifications. Some brief performance studies suggest that a flexible, permanently changing assignment of cores to compute ranks can outperform a static distribution, while greedily haggling over cores throughout a simulation might perform even better.

ACM Reference Format:
Martin Schreiber and Tobias Weinzierl. 2018. A Case Study for a New Invasive Extension of Intel's Threading Building Blocks . In *Proceedings of HiPEAC 2018—3rd COSH Workshop on Co-Scheduling of HPC Applications (HIPEAC'97).* ACM, New York, NY, USA, 6 pages. https://doi.org/arXiv:xxxx

1 INTRODUCTION

With clock rates in supercomputers plateauing, future generations of high-end computers will obtain their unprecedented capabilities from an increase of the number of cores that are integrated into every single node. Their nodes are small systems on chips. Though some roadmaps and funding calls demand for radical re-writes of our simulation software such that the codes exhibit omnipresent concurrency, we do believe that many supercomputer users will "simply" decompose machines with many cores per node logically into machines with many nodes and fewer cores. They will deploy multiple MPI ranks per node and assign each rank a subset of the available cores per node, as they want to continue to use their multi-decade legacy codes which cannot be rewritten from scratch. At the same time, supercomputing codes will retain code fragments with nested fork-join (NFJ) patterns as well as MPI synchronisation points. They will continue to run through phases with limited concurrency. Without novel node usage paradigms, many scientists will thus struggle to exploit the full potential of new supercomputers with their existing codes. A novel paradigm however has to come along with marginal code modifications. Otherwise, it will struggle to become accepted. We propose such a paradigm for Intel's Threading Building Blocks (TBB).

Classic load balancing uses cost models and runtime measurements to determine homogeneous workload distributions. It minimises runtime differences between ranks. This mitigates but does not eliminate inefficient node usage patterns which arise inevitably from thread-based NFJ and MPI synchronisation, i.e. once we assign each rank a fixed (and usually the same) number of cores to be exploited via multithreading. If sequential or low-concurrency phases per rank remain, even load-balanced codes continue to run into phases where few cores of a node are utilised. Temporal ill-balancing is amplified by algorithmic fragments with unpredictable workload (localised Newton-solves, e.g.), IO scattered among the nodes, or performance fluctuations caused by vector instructions, e.g. We focus on the classic time stepping from the ExaHyPE code [3] in this manuscript. ExaHyPE uses adaptive mesh refinement (AMR) with multipe MPI ranks, each parallelised with TBB, per compute node. Here, cores inevitably run into waits for other ranks per time step, while some internal program parts fork threads and join them again: Large fragments of each time step exhibit an enormous concurrency while the remainder scales only up to few cores. No matter how sophisticated the load balancing, a fixed association of cores to ranks is inappropriate at certain program phases.

The present paper follows a paradigm shift. Instead of assigning work to fixed rank-core assemblies, we make cores dynamically join the ranks that most need additional compute power. The core-rank association follows the workload, i.e. data distribution, and the core per rank ratio changes over time. From a programming model point of view, such a paradigm makes ranks invade cores when compute resources are needed and they retreat from these cores afterwards to free resources for other ranks. It falls into the class of invasive programming [9]. While we focus solely on one MPI code, all techniques also work if different applications run concurrently.

Our approach is minimalist: We disable features such as masking and make each rank running on one node occupy all hardware threads. If hyperthreading is disabled, physical threads correspond to cores. We thus use the terms as synonyms. For R ranks per node

*This work received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671698 (ExaHyPE). It made use of the facilities of the Hamilton HPC Service of Durham University.
†Corresponding author

HIPEAC'97, January 2018, Manchester, UK

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/arXiv:xxxx

with C cores, each core is overbooked as we launch in total $R \cdot C$ (logical) threads. All the ranks on one node agree through a shared memory region, a scratchpad, which cores may be used by which rank for their work. This establishes a lightweight communication channel between ranks. Once a rank has obtained c cores for work, it spawns $C - c$ lock tasks. Lock tasks make their designated threads sleep. This effectively releases the core for other ranks to perform their work. Invasion of cores in this model corresponds to a termination of the corresponding lock tasks.

Invasive programming and invasive code ingredients have been proposed before (see e.g. [2, 6, 8]). The present integration of a straightforward, lightweight, scratchpad-based, decentralised, and easy to use invasive extension of TBB into existing C++ codes is, to the best of our knowledge, however new. We believe that the proposed approach can help scientists to exploit manycore architectures more efficiently. For the very first time, standard load balancing is given a powerful and lightweight assistant: Our programming concept allows for an easy-to-use and fine granular adaption of computing resources throughout all phases. Communication overheads of a centralised resource manager are eliminated by the utilisation of a inter-program shared scratchpad.

The remainder is organised as follows: We start from a description of our minimalist application programming interface (Sec. 2) before we highlight the technical realisation. Both ingredients allow us to compare the proposed solution to existing approaches (Sec. 4). We quickly sketch the integration of our new techniques into an existing solver (Sec. 5) hereafter before some preliminary performance studies uncover the techniques' potential. A brief wrap up and outlook in Sec. 7 close the discussion.

2 APPLICATION PROGRAMMING INTERFACE

Our application programming interface (API) picks up idioms of TBB. The proposed API itself is implemented on top of TBB as an additional layer. An existing TBB parallelisation acts as starting point. We require the user to instantiate a class `SHMInvade` whenever additional cores would be of use, i.e. a user has to insert further code statements.

The construction of `SHMInvade` checks the scratchpad content for additional cores to "help out" on the computation. Here, the number of additionally requested cores is used as an upper bound for the new cores to be invaded. These additional cores are released automatically by the destructor of the `SHMInvade` object. Alternatively, users may manually free the cores before. The concept mirrors the realisation of TBB's `scoped_lock`.

```
foo(); // ran on default number of cores
/// try to get up to three additional cores
SHMInvade invade(3);
tbb::parallel_for ...{
  if (i==0) {
    // try to book up to two more cores
    SHMInvade invade(2);
    bar();
  } // release two more cores
  ...
}
```

Our approach

(1) neither requires the user to alter the original TBB code,
(2) nor does it require the user to keep track of how many cores actually have been grabbed (invaded)—this information is stored internally within the `SHMInvade` object,
(3) nor does it put any constraints on the invasion cardinality.

For expensive interior operations (`bar()` in the example above), it might be advantageous to book cores on short note and then to release them immediately, as other ranks running in SPMD mode might benefit from released compute resources. Overall, it however also is possible to work with very few `SHMInvade` objects held on an outer loop level: our demonstrator code for example can invade cores prior to the time stepping loop, and it releases those cores once this loop has terminated and the rank enters an MPI communication phase. The invasive infrastructure supports both granularity paradigms and any hybrid combination of them.

Our invasive realisation offers a robust, reliable resource management: Resources are physically allocated and freed by individual applications through instances of `SHMInvade`. As the ownership of compute data is encapsulated within the object, we can ensure that no core-to-rank assignment is corrupted. We also foster that programmers release resources on time and no instance behaves greedily. While our invasive realisation is minimally invasive w.r.t. programming—code is solely augmented—our work realises two design criteria which require developers to revise and rethink their solutions:

- Algorithm-aware programming: The application has to be aware where its algorithms run into arithmetically challenging phases. Such an awareness drives the sophisticated insertion of `SHMInvade` instances as it is not for free to instantiate this object.
- Resource-aware programming: Each application is responsible to take care of the resource demands of the other ranks running on the same node. A sophisticated application optimises the overall resource utilisation locally per rank under consideration of the performance and resource requirements of other ranks.

3 REALISATION

Once a rank starts up on a node, it issues TBB threads on all of the node's cores. In a second step, it establishes access to a scratchpad, a common memory region used by all threads. From hereon, the `SHMInvade` objects take over control as they argue through the scratchpad whether additional cores can be invaded, or they notify the other ranks through the scratchpad if cores become available.

3.1 Lock tasks and work stealing

We invert the actual invasion procedure: Ranks do not actively invade cores, but they actively retreat from cores. We assume that TBB's workstealing keeps all C threads busy by default. If a rank has to retreat from a core—it retreats from all cores besides its "master" one at startup—it spawns a lock task and marks, protected by a system mutex, the thread to be freed. Lock tasks are scheduled through TBB's `enqueue` command and thus are starvation resistant. If TBB's work stealing issues a lock task on a misfitting hardware thread, this lock tasks reschedules itself and yields immediately.

If a lock task is ran on its corresponding thread and the thread is marked to be freed, the tasks sends the executing thread to sleep. If a lock task is ran on its corresponding thread and the thread is to be used, the lock task terminates. Waking up threads is thus accomplished by unsetting the marker for a particular thread. This eventually terminates a lock task. Oversubscription of cores on purpose is not supported yet by our approach.

3.2 Scratchpad

We use a POSIX shared memory object as scratchpad for the ranks. Each process allocates a shared memory mapped region using `shm_open(...)`, `ftruncate(...)` and `mmap(...)`. Such shared memory objects behave as in-memory shared files between the MPI ranks. Using `ftruncate` to resize the memory object assures that the object is initialised with 0-values in case that it wasn't initialised before. An additional field `is_initialized` indicates that the data in the shared memory region was not initialised yet and allows the first rank creating the object to set up all data appropriately.

Besides a `spin_lock` used to lock the content of the shared memory block, the scratchpad holds some global properties (such as the number of known cores administered through it) plus a table of all cores on this machine and a table of all ranks (see Table 1).

At startup, each rank acquires a unique index within the shared memory region. The indices range from 0 to $R-1$.

```
(1) num_active_threads = 0
(2) lock()
(3)  reg_process_pids[counter_reg_processes] = PID
(4)  reg_process_cores[counter_reg_processes] = 0
(5)  user_data_per_process[counter_reg_processes] = {0,
     ..., 0}
(6)  counter_reg_processes++
(7) unlock()
```

Each rank with process ID `pid` corresponds to one entry within `reg_process_cores`. To invade a core, we lock the shared memory region, and we determine how many cores are globally available (`num_free_cores`). Once determining the number of cores to invade, the fields `num_free_cores` and `reg_process_cores` are updated. This is accomplished within the locking phase to avoid race conditions. After this, the access to the shared memory region is unlocked. All other TBB-related actions (see previous section) are executed after unlock the shared memory region. Retreating from a core works in a similar way.

Overall, the scratchpad avoids a centralised resource manager [2, 6, 8]. No master process exists and the applications have to bargain amongst themselves for the best resource allocation. This removes the overheads of communicating the master process as well as starting and scheduling the runtime of the master process.

We close this section with a brief discussion on the scalability limits of locking the shared-memory region. The scalability limitations for this depends on the overheads of the mutual exclusive access, the time to read/update the shared memory region and the number of processes which perform this in parallel. We used spinlocks for their low overheads compared to system mutices. This keeps the serial overheads low. We assure that the invasive operations are only used to invade/retreat cores and that user-space operations are only allowed to block-wise load and write data during a spinlock phase. Finally, we focus on commodity multicore chips, only. Additional hardware-related scalability limitations such as false-sharing and NUMA effect might arise but were not further investigated in the present work but might lead to unexpected behaviour.

3.3 User data exchange

Our approach technically relies on a greedy invasion of compute cores. We outsource a responsible and sustainable usage of cores to the users. As such, it is convenient to offer an additional table in the shared memory regions, which developers can use to exchange properties between the different ranks on one compute node. The unique index per node from $\{0, \ldots, r-1\}$ identifies which rank is allowed to write into which row of such a table, while all ranks may read all entries.

4 RELATED WORK AND CONTEXT

Tailoring and optimising the assignment of computing resources to processes is not new, dynamic, i.e. online approaches are rare though. We notably refer to work published under the umbrella of [9] for related work. Reacting to changing resource requirements, i.e. to invade resources and to retreat from resources, can for example be realised through a centralised [2, 6] or distributed [7] resource manager. Our approach abandons the idea of any resource manager and instead proposed a marketplace where ranks can grab cores in a first-come first-served manner. It allows us to require users to augment their codes but not changing them while it puts the responsibility for sustainable resource usage on them. In practice, we consider it to be a convenient strategy in HPC, as we (i) assume that multiple ranks running on the same node are not perfectly balanced anyway, (ii) have to synchronise in regular time intervals, (iii) have NFJ source code fragments and (iv) run into close-to-serial communication phases regularly. If one rank thus invades, at one point, an inappropriate number of cores, it finishes earlier and frees these resources earlier. The total balancing smooths out over given time intervals such as time steps.

The two omnipresent HPC standards MPI and OpenMP lack support for such lightweight dynamic resource assignment. For recent MPI, adding new computing resources is part of the standard. Despite recent investigation of fully malleable MPI [4], it remains unclear how "cheap" such MPI splits are, what implications for the data transfer (message passing) arise, and whether current MPI implementations and supercomputer job schedulers support an aggressive growth of MPI processes. In OpenMP, a dynamic resource management was investigated in [2, 6, 8], e.g. All approaches however only support resource reallocation in rather outer loops. It is not possible to invade additional cores while other logical threads of the application run concurrently as the scheduler has to be restarted. Finally, we note that projects alike [3] investigate into work stealing in-between MPI ranks. If such a technique is applied in-between ranks deployed to the same node, it smoothes out load imbalances per node, too. Yet, all paradigms start from the motivation to balance the work given a certain hardware configuration. Our approach tailors the hardware configuration towards the actual runtime behaviour.

Type	Identifier	Description
`mutex`	`spin_mutex`	Spin mutex to avoid race conditions
`bool`	`is_initialized`	True if this data structure is already initialized
`int`	`max_available_cores`	Maximum number of possible threads ($\leq C$ (max cores)
`int`	`num_free_cores`	Current number of free cores ($\leq C$ (max cores)
`int` (atomic)	`counter_reg_processes`	Number of registered processes using SHMInvade
`pid_t`	`reg_process_pids[]`	PIDs of registered programs
`int`	`reg_process_cores[]`	Number of used cores for each program
`void`	`user_data_per_process[][]`	User-specific data for each process

Table 1: Global data structures with data to be filled in. All variables are declared as `volatile`, hence are not cached in registers but always read from memory.

Figure 1: Two screenshots of the two-dimensional Euler equations applied to an initial energy distribution derived from the ExaHyPE logo.

5 INTEGRATION

For our proof-of-concept, we investigate two different approaches. In the first approach, we make each rank start per time step from only one core and rely on an on-the-fly invasion to balance out an appropriate distribution of cores to ranks. In the second approach, we make all ranks running on one node to minimise their total runtime in a joint afford. The ExaHyPE engine [3] acts as testbed. We simulate the two-dimensional Euler equations with the ADER-DG explicit time stepping scheme [5] on dynamically adaptive grids (Figure 1). The software base is interesting for multiple reasons:

- The code runs on a dynamically adaptive grid where the grid changes in each and every time step.
- The ADER-DG scheme is a predictor-corrector scheme where a computationally expensive predictor phase is followed by a reasonably cheap non-linear Riemann solve for the discontinuities plus a correction step.
- The predictor solves the underlying equations with Picard iterations per cell: the cost per cell thus depends heavily on the iteration steps for the nonlinear problem. It varies in each and every time step.

ExaHyPE is built upon the Peano sources which is an open-source spacetree code. We add all `SHMInvade` invocations there, i.e. our invasion approach is agnostic of the actual application domain.

5.1 Invade throughout computation

This approach does not hold any `SHMInvade` permanent over extensive time. Instead, we focus on the code's underlying nested parallel fors and tasks. Prior to its task spawning and the loops, we insert `SHMInvade` commands.

We note that the code uses the same computational infrastructure for all different algorithmic phases, i.e. if we augment one loop by invasion commands, this augmentation applies to both predictor, Riemann solve and correction. The invasion does not take the computational intensity into account. Furthermore, no rank does directly interact with any other rank to decide whether it should try to invade cores: We may assume that only the coarser levels of the nested parallelisation succeed in invading cores. Yet, once one rank frees cores on a rather coarse level, other, very fine-granular, invasion attempts might pass through. No fairness policy is in place.

5.2 High-level integration

For our fair alternative approach, we start from the assumption that each rank faces its own strong scaling challenge w.r.t. the cores available to it. A modified Amdahl's law [1]

$$t_r(c_r) = f_r \cdot t_r(1) + (1 - f_r)\frac{t_r(1)}{c_r} + s_r \cdot c_r \tag{1}$$

yields a reasonable description of the scaling behaviour (Fig. 2) global behaviour of one rank. Yet, the values f_r (serial code fraction), $t_r(1)$ (serial runtime) and s_r (startup cost of the cores/threads) differ per rank $r \in \{0, \ldots, R-1\}$. Here, $1 \leq c_r \leq C$ is the number of cores/threads available to the rank. Each rank runs the same code (SPMD) but we run dynamic AMR and thus obtain different performance characteristics per node. Our invasive strategy consists of two steps per iteration of the underlying solver:

(1) Runtime analysis: Each rank tracks its own compute time relative to the cores that are available to it. This allows the rank to calibrate its $(f_r, t_r(1), s_r)$ quantities to its own behaviour. It runs its own online machine learning algorithm.

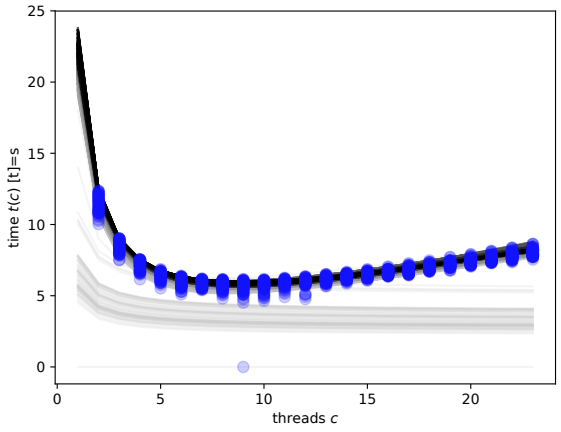

Figure 2: Scaling studies with varying core counts. The blue dots are the measurements, the black lines result from our non-linear weighted regression. The darker the stroke, the more data has been available to the regression.

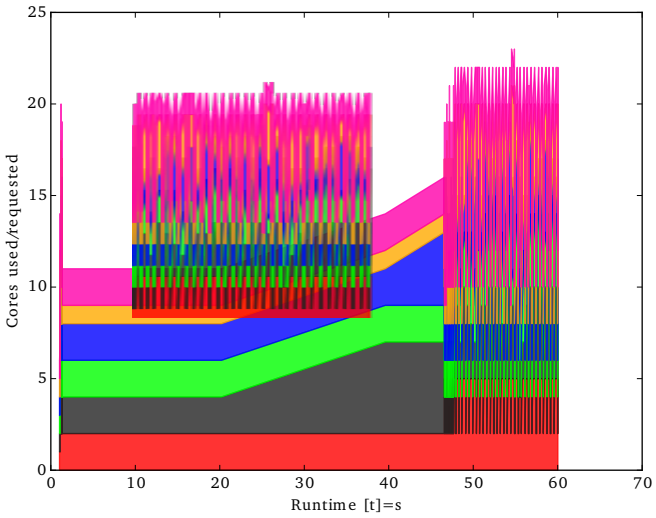

Figure 3: Example core per rank distribution incl. zoom-in for our global approach where the core-to-rank distribution is subject to a global optimisation, i.e. it is changed per simulation time step according to (2).

(2) Each rank notifies all other ranks about its $(f_r, t_r(1), s_r)$. A global optimisation to reduce the overall compute time then instructs the rank whether it should release some cores or try to grab more cores.
(3) Each rank tries to invade its optimal number of ranks prior to the time step kick off and retreats from the cores as soon as the traversal terminates and the rank continues with tidying up all MPI messages and waiting for the next time step instruction.

Runtime analysis. With N measurements $\left((c_r^{(1)}, t_r^{(1)}), \ldots, (c_r^{(N)}, t_r^{(N)})\right)$ of runtimes of a rank r for various (invaded) core numbers, we obtain an overdetermined non-linear data fitting problem to determine f_k, $t_r(1)$ and s_r. Let the first entry in this series, i.e. $(c_r^{(1)}, t_r^{(1)})$ is the most recent measurement. We formalise the calibration per node as

$$\forall r : \min_{f_r, t_r(1), s_r} \frac{1}{2} \sum_{n=1}^{N} q^n \| f_r \cdot t_r(1) + (1 - f_r) \frac{t_r(1)}{c_r^{(n)}} + s_r \cdot c_r^{(n)} - t_r^{(n)} \|^2$$

with a temporal weighting $q \in (0, 1)$ which makes more recent measurements more significant. $q = 0.9$ is used in the experiments.

The problem is a constrained, non-linear, weighted regression problem from machine learning. We solve it iteratively via Gauss-Newton shifts that we manually constrain after each Newton iteration such that $0 < f_r < 1$ and $t_r, s_r > 0$. Furthermore, updates to the three quantities are made sliding averages. Simpler schemes such as Picard experimentally did fail in our setups, so we assume that there is a lack of contraction properties. Given the temporal weighting, we may assume that the triple f_r, t_r, s_r yields a reasonable accurate description of a ranks runtime behaviour after few grid sweeps.

Global optimisation of core distribution. The global workload distribution now reads

$$\min_{c_r \geq 1} \max_r \; t(c_r) \quad \text{with} \quad \sum_r c_r \leq C,$$

which we smoothly approximate by the penalised

$$\min_{c_r} \frac{1}{2m} \sum_k t(c_r)^{2m} + \frac{\alpha}{2} \left(C - \sum_r c_r \right)^2 \tag{2}$$

with $\alpha > 0, m \in \mathbb{N}^+$. Setting the derivative zero yields the optimality condition.

Our code uses the scratchpad to exchange the (f_r, t_r, s_r) values determined from the measurements. As part of the machine learning process, each rank dumps its regression results and as each rank can read all entries from the scratchpad, each rank can determine the solution to the overall core optimisation problem locally. As (f_r, t_r, s_r) for most of our setups change only smoothly, these solves are not synchronised at all, i.e. some ranks might reuse calibration data from previous time steps, while other ranks already rely on updated data.

6 RESULTS

We ran experiments on 24 core Intel E5-2650V4 (Broadwell) nodes. The Broadwells run at 2.4 GHz and are connected by Omnipath. Intel's 2017.2 C++ compiler is used with the accompanying Intel MPI library. We deploy six MPI ranks on each single node, and pick out particularly interesting or characteristic results. All timings are real-time measurements sampled every time a time step terminates.

The global optimisation (Figure 3) starts with one core per ranks and remains quasi-stationary for the program's start-up phase where many system calls (memory allocations) are required. Not much concurrency is observed here. Once the actual computation

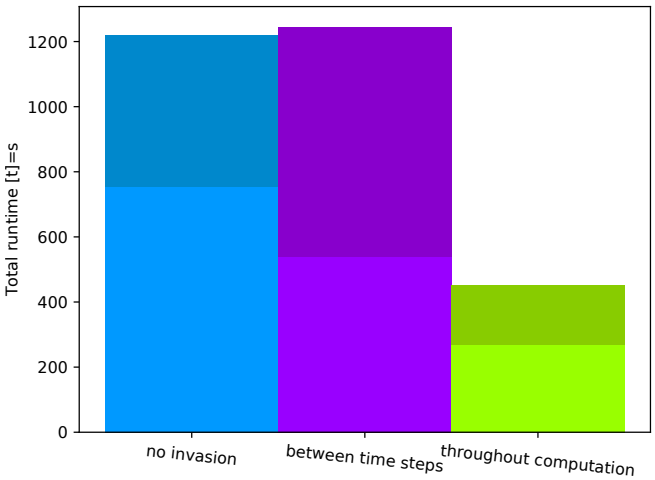

Figure 4: Global runtimes for a static core-to-rank association (TBB only) and the two different invasion approaches. The darker bars in the background track the total time-to-solution. The lighter bars in the foreground track only arithmetically intense solver steps.

starts, it yields a seismogram-like pattern. Each rank releases its cores after the traversal immediately but other ranks struggle to benefit from it as they are, for reasonably balanced MPI applications, already close to the time step completion, too. An amortised slack of 3–4 cores is observable which remain unused. A more aggressive invasion throughout the solve which does not try to fix the core count once per time step promises to fill in these slacks.

While our measurements (Fig. 2) suggest that the regression requires a few hundred steps to converge, the invasion seems not to suffer from this fact—if all rank estimates are off by the same ratio, the haggling for cores still seems to come up with reasonable core distributions. An exact study of this behaviour however is subject of future research.

We next compare invasion throughout the actual computation with the previous invasion in-between time steps and a non-invasive baseline with a homogeneous core distribution. Hereby, we track either the total runtime or the computationally intense code parts only. We see the global optimisation paying off if we focus only on the arithmetically intense code parts. If the computational intensity however is small, one invasion per time step yields worse performance even than a non-invasive approach. Our results suggest (not shown) that our simple Amdahl model in (1) does not hold anymore. The totally dynamic approach outperforms a per time step invasion robustly. If the computational intensity is high, the overhead induced by frequent invasion tries is negligible. If the computational intensity is low, any approach without fine-grain invasion is doomed to fail right from the start.

7 SUMMARY AND OUTLOOK

We propose an invasive extension of TBB which require users to insert only very few lines of code into their applications. It works without any centralised decision making algorithm (resource manager). The integration into a sophisticated simulation code suggests that a rank-global optimisation of resource usage is, counter-intuitively, not superior to an anarchic, greedy grabbing of resources.

Future work comprises, on the one hand, detailed studies on the invasion behaviour. Notably, we have to evaluate whether hybrids of global optimisation and greedy resource invasion can outperform all presented runs. On the other hand, locality- and affinity-aware assignment of invaded cores is not yet integrated into the invasion code base. Also, temporarily core overbooking might improve the performance for scenarios where cache thrashing plays a non-significant role, overheads induced by invasion are outperformed through non-exclusive core assignment or the program idles/stalls and the flow control is handed over to the operating system. Such features might provide a further invasion boost.

ACKNOWLEDGEMENTS

The authors appreciate support received from the European Union's Horizon 2020 research and innovation programme under grant agreement No 671698 (ExaHyPE). Thanks are due to all members of the ExaHyPE consortium who made this research possible, notably Dominic E. Charrier and Benjamin Hazelwood. This work made use of the facilities of the Hamilton HPC Service of Durham University. Both authors appreciate former funding through the Transregional Collaborative Research Centre 89—Invasive Computing (DFG funded).

REFERENCES

[1] G. M. Amdahl. 1967. 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))*. ACM, New York, NY, USA, 483–485. https://doi.org/10.1145/1465482.1465560

[2] Michael Bader, Hans-Joachim Bungartz, and Martin Schreiber. 2013. Invasive computing on high performance shared memory systems. In *Facing the Multicore-Challenge III*. Springer, 1–12.

[3] M. Bader, M. Dumbser, A.A. Gabriel, H. Igel, L. Rezzolla, and T. Weinzierl. 2017. ExaHyPE—An Exascale Hyperbolic PDE Engine. (2017). http://www.exahype.org

[4] Isaías Comprés, Ao Mo-Hellenbrand, Michael Gerndt, and Hans-Joachim Bungartz. 2016. Infrastructure and API Extensions for Elastic Execution of MPI Applications. In *Proceedings of the 23rd European MPI Users' Group Meeting*. ACM, 82–97.

[5] M. Dumbser, O. Zanotti, R. LoubÃÍre, and S. Diot. 2014. A posteriori subcell limiting of the discontinuous Galerkin finite element method for hyperbolic conservation laws. *J. Comput. Phys.* 278 (2014), 47–75.

[6] M. Gerndt, A. Hollmann, M. Meyer, M. Schreiber, and J. Weidendorfer. 2012. Invasive computing with iOMP. In *2012 Forum on Specification and Design Languages (FDL)*. IEEE, 225–231.

[7] S. Kobbe. 2015. *Scalable and Distributed Resource Management for Many-Core Systems*. Ph.D. Dissertation. Chair for Embedded Systems (CES), Department of Computer Science, Karlsruhe Institute of Technology (KIT).

[8] M. Schreiber, C. Riesinger, T. Neckel, H.-J. Bungartz, and A. Breuer. 2015. Invasive Compute Balancing for Applications with Shared and Hybrid Parallelization. *International Journal of Parallel Programming* 43, 6 (2015), 1004–1027.

[9] J. Teich et al. 2017. Transregional Collaborative Research Centre 89. (2017).