Numer. Math. Theor. Meth. Appl.
doi: 10.4208/nmtma.m12si07

Vol. **5**, No. 1, pp. 110-130
February 2012

A Geometric Space-Time Multigrid Algorithm for the Heat Equation

Tobias Weinzierl[1,*] and Tobias Köppl[2]

[1] *Institut für Informatik, Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany.*
[2] *Institut für Mathematik, Technische Universität München, Boltzmannstr. 3, 85748 Garching, Germany.*

Received 29 November 2010; Accepted (in revised version) 05 May 2011

Available 21 December 2011

Abstract. We study the time-dependent heat equation on its space-time domain that is discretised by a k-spacetree. k-spacetrees are a generalisation of the octree concept and are a discretisation paradigm yielding a multiscale representation of dynamically adaptive Cartesian grids with low memory footprint. The paper presents a full approximation storage geometric multigrid implementation for this setting that combines the smoothing properties of multigrid for the equation's elliptic operator with a multiscale solution propagation in time. While the runtime and memory overhead for tackling the all-in-one space-time problem is bounded, the holistic approach promises to exhibit a better parallel scalability than classical time stepping, adaptive dynamic refinement in space and time fall naturally into place, as well as the treatment of periodic boundary conditions of steady cycle systems, on-time computational steering is eased as the algorithm delivers guesses for the solution's long-term behaviour immediately, and, finally, backward problems arising from the adjoint equation benefit from the the solution being available for any point in space and time.

AMS subject classifications: 65M50, 65M55, 65N50, 65N55, 65M22

Key words: Adaptive Cartesian grids, geometric multiscale methods, heat equation, octree, spacetree, space-time discretisation.

1. Introduction

We study the heat equation

$$\partial_t u - \kappa \Delta u = f \quad \text{with} \quad u : \Omega \times (0,T) \mapsto \mathbb{R}, \tag{1.1a}$$

$$u(t=0) = u_0 \quad \text{or} \quad u(t=0) = u(T) \tag{1.1b}$$

$$u|_{\partial\Omega} = g, \kappa \in \mathbb{R}^+ \tag{1.1c}$$

*Corresponding author. *Email addresses:* weinzier@in.tum.de (T. Weinzierl), koeppl@ma.tum.de (T. Köppl)

on the unit square $\Omega = (0,1)^d \subset \mathbb{R}^d$ with $d \in \{1,2,3\}$ for a fixed time interval $(0,T)$. f, g, and u_0 are sufficiently smooth. The simple equation is a building block of many sophisticated applications in science and engineering—ranging from nano-scale computational fluid dynamics to chemical diffusion processes in turbulent flows. Software for this type of problems typically discretises the time with a standard ordinary differential equation integrator, which leads to a cascade of elliptic problems in space, and uses a sophisticated linear equation system solver such as a geometric multigrid algorithm for the latter.

While both time integration and multigrid solver are comprehensively studied, the problem nevertheless is far from an old-fashioned challenge. As more and more massively parallel computers are released, the question arises how such a cascade of elliptic problems scales. Often, the size of one single elliptic subproblem is too small to exploit the full power of all processing units. This holds in particular for coarse grid subproblems of multigrid algorithms. More and more cores then do not enable one to handle longer and longer simulated time intervals in a given period due to the sequential character of the underlying challenge [10, 11]. Parareal-type algorithms [5] solving several spatial subproblems in parallel promise to resolve this problem due to a good prediction for the solution's time behaviour. Other approaches deploy the first p subsequent time slices to the p different nodes of a parallel computer [9, 19] and implement a chain of water carriers where the first computing node takes over the work of the $p+1$th time step as soon as the initial time step's solution has converged. However, on massively parallel systems, some processing units here might solve rather irrelevant equation systems. As many experimental settings exhibit small regions of interest (boundary or interface subdomains, e.g.) dynamic adaptivity becomes one key ingredient of sophisticated solvers. The adaptivity in space implies multiscale behaviour in time. Regions with small scale spatial behaviour also require an accurate time representation. Local time stepping approaches promise to take this insight into account. However, local time stepping makes load balancing challenging on massively parallel computers. As more and more solvers change from stand-alone applications to applications embedded into steering environments, coupled software component ecosystems, and problem solving environments [1], today's solvers for parabolic equations not only have to deliver an accurate approximation of the real-world behaviour. Good guesses of the long term behaviour have to be delivered fast to enable users and other components to interact with the solver immediately. Smaller and smaller time step sizes required to get better approximations however force the environment to wait longer for results. Sequences of computations with finer and finer time step sizes promise to deliver coarse grain solutions on time. However, they often neglect fine scale effects stemming from the initial conditions. As more and more challenges change from forward problems to optimisation and calibration tasks, algorithms today often have to solve parabolic equations forward and backward in time simultaneously [2, 3]. For this, one needs a representation of the solution all along the time interval. Sophisticated checkpointing strategies promise to deliver such snapshots of the solution at any time for a fair trade-off between computing resources and storage requirements [2]. However, with less and less memory per core available, it is not clear how these approaches scale on supercomputers—in particular for $d = 3$, where a full, regularly resolved space-time grid already reveals the curse of dimension [8]. Placed