juliacon

Distributed Parallelization of xPU Stencil Computations in Julia

Samuel Omlin[1], Ludovic Räss[2, 3], and Ivan Utkin[2, 3]

[1]Swiss National Supercomputing Centre (CSCS), ETH Zurich, Lugano, Switzerland
[2]Laboratory of Hydraulics, Hydrology and Glaciology (VAW), ETH Zurich, Zurich, Switzerland
[3]Swiss Federal Institute for Forest, Snow and Landscape Research (WSL), Birmensdorf, Switzerland

ABSTRACT

We present a straightforward approach for distributed parallelization of stencil-based xPU applications on a regular staggered grid, which is instantiated in the package `ImplicitGlobalGrid.jl`. The approach allows to leverage remote direct memory access and enables close to ideal weak scaling of real-world applications on thousands of GPUs. The communication costs can be easily hidden behind computation.

Keywords

Julia, Distributed Parallelization, xPU, GPU, Supercomputing, Stencil Computations, Staggered Grid

1. Introduction

In light of the high pace of hardware evolution since the dawn of the 21st century, the HPC community has identified the 3 "P"s - (scalable) Performance, (performance) Portability and Productivity - as fundamental requirements for today's and tomorrow's software development. The approach and package development presented in this paper responds to each of the 3 "P"s. We present an approach for automatic and architecture-agnostic distributed parallelization of stencil-based xPU applications on a regular staggered grid (with xPU we refer simultaneously to GPU and CPU in this paper).

2. Approach

The here presented approach renders the distributed parallelization of stencil-based xPU applications on a regular staggered grid almost trivial. We have instantiated the approach in the Julia package `ImplicitGlobalGrid.jl`. A highlight in the design of ImplicitGlobalGrid is the automatic implicit creation of the global computational grid based on the number of processes the application is run with (and based on the process topology, which can be explicitly chosen by the user or automatically defined). As a consequence, the user only needs to write a code to solve his problem on one xPU (local grid); then, as little as three functions can be enough to transform a single xPU application into a massively scaling multi-xPU application: a first function creates the implicit global staggered grid, a second function performs a halo update on it, and a third function finalizes the global grid. ImplicitGlobalGrid does not have any requirements on the packages used to obtain good per xPU performance (shared memory parallelization and optimisations can, e.g., be performed with ParallelStencil [3] or any other package that might suit this task). Fig. 1 shows a stencil-based 3-D heat diffusion xPU solver, where distributed parallelization is achieved with the three ImplicitGlobalGrid functions mentioned (lines 23, 38 and 43) plus some additional functions to query the size of the global grid (lines 24-26; in this example ParallelStencil [3] is used to obtain high per xPU performance).

ImplicitGlobalGrid relies on `MPI.jl` [2] to perform halo updates close to hardware limits. For GPU applications, ImplicitGlobalGrid leverages remote direct memory access when CUDA- or ROCm-aware MPI is available and, otherwise, uses highly optimized asynchronous data transfer routines to move the data through the hosts. In addition, pipelining is applied on all stages of the data transfers, improving the effective throughput between GPU and GPU. Low level management of memory, CUDA streams, ROCm queues and signals permits to efficiently reuse send and receive buffers and streams or queues and signals throughout an application without putting the burden of their management to the user. Moreover, all data transfers are performed on non-blocking high-priority streams or queues, allowing to overlap the communication optimally with computation. `ParallelStencil.jl`, e.g., can do so with a simple macro call (Fig. 1, line 36).

Asymmetrical halos that could result from staggered grids and would represent a considerable complexity are fully circumvented in our approach: a field will only have halos in a given dimension if the corresponding overlap between the local fields is at least two cells wide; no halos are created if the overlap is only one cell wide (redundant computation is done instead).

ImplicitGlobalGrid is fully interoperable with `MPI.jl`. By default, it creates a Cartesian MPI communicator, which can be easily retrieved together with other MPI variables. Alternatively, an MPI communicator can be passed to ImplicitGlobalGrid for usage. As a result, ImplicitGlobalGrid's functionality can be seamlessly extended using `MPI.jl`.

The modular design of ImplicitGlobalGrid, which heavily relies on multiple dispatch, enables adding support for other hardware with little development effort (including new kind of accelerators as soon as they become programmable with Julia). Support for AMD GPUs using the recently matured `AMDGPU.jl` package [5] could be implemented shortly after as a result (Nvidia GPUs are supported using `CUDA.jl` [1]). ImplicitGlobalGrid supports at present distributed parallelization for CUDA- and ROCm-capable GPUs as well as for CPUs.

```julia
using ImplicitGlobalGrid
using ParallelStencil
using ParallelStencil.FiniteDifferences3D
@init_parallel_stencil(CUDA, Float64, 3)

@parallel function step!(T2,T,Ci,lam,dt,dx,dy,dz)
    @inn(T2) = @inn(T) + dt*(
        lam*@inn(Ci)*(@d2_xi(T)/dx^2 +
                      @d2_yi(T)/dy^2 +
                      @d2_zi(T)/dz^2 ) )
    return
end

function diffusion3D()
    # Physics
    lam       = 1.0           #Thermal conductivity
    c0        = 2.0           #Heat capacity
    lx=ly=lz  = 1.0           #Domain length x|y|z

    # Numerics
    nx=ny=nz  = 512           #Nb gridpoints x|y|z
    nt        = 100           #Nb time steps
    me,       = init_global_grid(nx, ny, nz)
    dx        = lx/(nx_g()-1) #Space step in x
    dy        = ly/(ny_g()-1) #Space step in y
    dz        = lz/(nz_g()-1) #Space step in z

    # Initial conditions
    T  = @ones(nx,ny,nz).*1.7 #Temperature
    T2 = copy(T)              #Temperature (2nd)
    Ci = @ones(nx,ny,nz)./c0  #1/Heat capacity

    # Time loop
    dt = min(dx^2,dy^2,dz^2)/lam/maximum(Ci)/6.1
    for it = 1:nt
        @hide_communication (16, 2, 2) begin
            @parallel step!(T2,T,Ci,lam,dt,dx,dy,dz)
            update_halo!(T2)
        end
        T, T2 = T2, T
    end

    finalize_global_grid()
end

diffusion3D()
```

Fig. 1: Stencil-based 3-D heat diffusion xPU solver implemented using ImplicitGlobalGrid (and ParallelStencil).

3. Results

We here report the scaling achieved with the 3-D heat diffusion xPU solver (Fig. 1) on up to 2197 Nvidia Tesla P100 GPUs on the Piz Daint Supercomputer at the Swiss National Supercomputing Centre (Fig. 2; 17^3, i.e., 2197 nodes is the biggest cubic node topology that can be submitted in the normal queue of Piz Daint). We observe a parallel efficiency of 93% on 2197 GPUs. Moreover, we have employed ImplicitGlobalGrid (and ParallelStencil) for the parallelization of a solver for nonlinear 3-D poro-visco-elastic two-phase flow and have also conducted a weak scaling experiment on Piz Daint (Fig. 3). We observe a parallel efficiency of over 95% on up to 1024 GPUs. As a performance reference, the solver implemented in Julia achieved 90% of the per-node performance of the respective original solver written in CUDA C using MPI.

4. Conclusions

We have shown that ImplicitGlobalGrid enables scalable performance, performance portability and productivity and addresses the 3 "P"s in all of its aspects. In addition, we have demonstrated the effectiveness and wide applicability of our approach within geosciences. Our approach is naturally in no sense limited to geosciences as distributed parallelization based on halo updates is employed in many scientific disciplines. We illustrated this in a recent contribution, where we showcased a quantum fluid dynamics solver using the nonlinear Gross-Pitaevski equation implemented with ImplicitGlobalGrid (and ParallelStencil) [4].

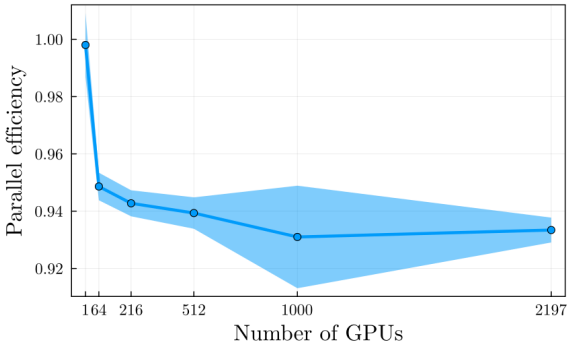

Fig. 2: Parallel weak scaling of the 3-D heat diffusion solver (Fig. 1) from 1 to 2197 (17^3) Nvidia P100 GPUs on Piz Daint at CSCS (problem size per GPU is 512^3). The blue surface visualizes the 95% confidence interval of the reported medians (20 samples). Per-node performance is discussed in [3]. The raw data and plotting script are available in `github.com/omlins/ImplicitGlobalGrid.jl/tree/master/paper`.

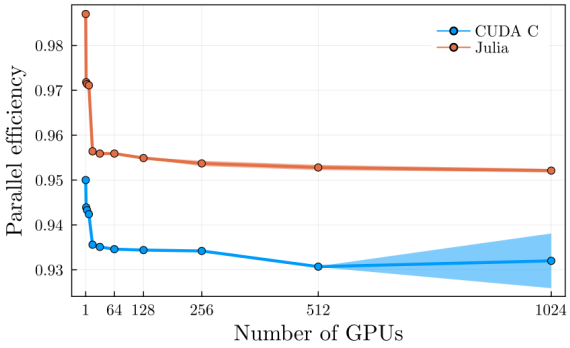

Fig. 3: Parallel weak scaling of the nonlinear 3-D poro-visco-elastic two-phase flow solver from 1 to 1024 Nvidia P100 GPUs on Piz Daint at CSCS (problem size per GPU is 382^3). The blue and orange surfaces visualize the 95% confidence interval of the reported medians (20 samples). For reference, the Julia solver achieved 90% of the per-node performance of the corresponding CUDA C solver. The raw data and plotting script are available in `github.com/omlins/ImplicitGlobalGrid.jl/tree/master/paper`.

5. Acknowledgments

We would like to thank Julian Samaroo (MIT) for his pro-active support for enabling AMDGPU in ImplicitGlobalGrid. This work was supported by a grant from the Swiss National Supercomputing Centre (CSCS) under project ID c23 through the Platform for Advanced Scientific Computing (PASC) program.

6. References

[1] T. Besard, C. Foket, and B. De Sutter. Effective extensible programming: unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):827–841, 2018. doi:10.1109/TPDS.2018.2872064.

[2] S. Byrne, L. C. Wilcox, and V. Churavy. MPI.jl: Julia bindings for the Message Passing Interface. In *Proceedings of the JuliaCon Conferences*, volume 1, page 68, 2021. doi:10.21105/jcon.00068. `https://github.com/JuliaParallel/MPI.jl`.

[3] S. Omlin and L. Räss. High-performance xPU Stencil Computations in Julia. *Proceedings of the JuliaCon Conferences*, 6(64):138, 2024. doi:10.21105/jcon.00138.

[4] S. Omlin, L. Räss, N. Keepfer, G. Kwasniewski, B. Malvoisin, and Y. Y. Podladchikov. Solving Nonlinear Partial Differential Equations on GPU Supercomputers Using Julia. PASC21 conference, 2021.

[5] J. Samaroo, T. Besard, V. Churavy, D. Lin, and other contributors. AMDGPU.jl: AMD GPU (ROCm) programming in Julia. `https://github.com/JuliaGPU/AMDGPU.jl`, 2013. doi:10.5281/zenodo.10040461.