p4est: Scalable Algorithms for Parallel Adaptive Mesh Refinement

Carsten Burstedde and Johannes Holke
INS, University of Bonn

Description of the Code

We examine the scalability of the `p4est` code for parallel adaptive mesh refinement (AMR) [1]. This code implements several algorithms to create a dynamic distributed mesh data structure, to refine, coarsen, and 2:1 balance it (see also [2]), and to repartition it between the parallel processes. Additional algorithms may be called to obtain topological information about the mesh, such as to search or iterate through it [3], or to identify a so-called ghost layer of off-process neighbor elements and transfer data between them. Initial tests of the latter functionality, called ghost exchange, is discussed in this report, together with results of refinement and partitioning.

The basic meshing concept we follow is to divide the domain conformingly into one or more logically hexahedral blocks. One block is suitable for meshing a cube or a torus, and moderate numbers usually suffice to mesh shapes like the spherical shell with good aspect ratio [4]. Complex domains as shown in Figure 2 may be subdivided using mesh generators. This feature is strictly optional, but powerful when needed. (If a mesh generator creates tetrahedra, such as Tetgen [5], we divide each one into four cubes in a preprocessing step.) Each of the coarse blocks becomes an octree by subdividing it arbitrarily into octants. This data structure is fully distributed and dynamic, such that meshes can be modified during runtime.

The parallel arrangement of data is guided by a space filling curve; see Figure 1. This approach allows for fast dynamic repartitioning; see Figure 3 for recent results obtained on the JUQUEEN supercomputer.

The design of `Partition` contains one `MPI_Allgather` call on one integer per rank (or two calls if we use the extra feature to align the elements to allow for coarsening [6]) , $\mathcal{O}(N/P)$ memory traversal and movement, and $\mathcal{O}(1)$ point-to-point messages per rank of total length $\mathcal{O}(N/P)$ with known sender/receiver arrangements. Here, N/P is the number of elements per

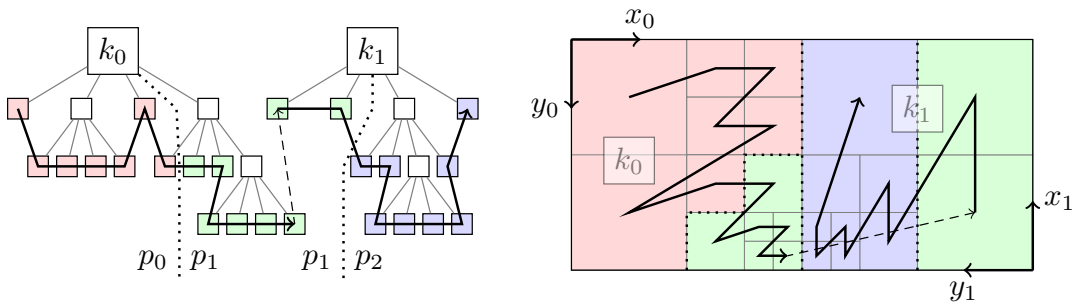

Figure 1: An example 2D mesh of two trees k_1 and k_2. It is partitioned between three processes p_0 through p_2 (color coded). The concept in 3D is analogous.

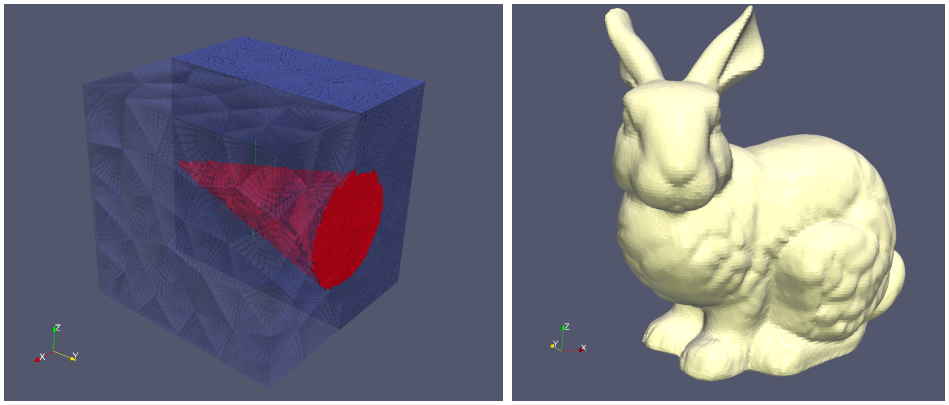

Figure 2: Left: The box mesh used in the 3D tests. Here we show a uniform refinement of level 2 (blue) with adaptive refinement to level 3 (red). Right: The Stanford bunny mesh from the Stanford University Computer Graphics Laboratory [10]. This version of the mesh consists of 495,511 tetrahedra.

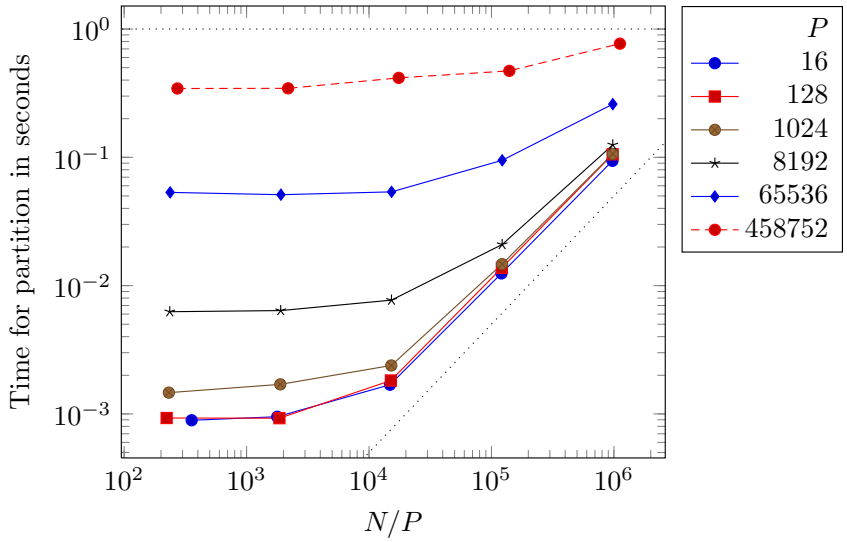

Figure 3: Time of `Partition` plotted against the number of elements N divided by MPI ranks P, on a mesh derived from six trees. Each line corresponds to varying N for a fixed P. All results are in between ideal strong scaling (diagonal line on the bottom right) and an absolute run time of under one second (top horizontal line). The largest run manages over $5 \cdot 10^{11}$ elements on the full size of JUQUEEN.
Strong scalability would be identified by keeping N constant and varying P. In the diagram, that means starting on the left, then moving one point to the right and one line down in each step. Towards the lower right, we approach the plotted diagonal since the lines for different P are on top of each other, indicating near-optimal scaling above 10^4 elements/rank and up to 1024 ranks. Weak scaling can be judged by looking vertically—keeping N/P constant should result in identical runtimes, which is satisfied by the lines with smaller P.
The results indicate that the timings become communication bound, which can be explained by the fact that the `Partition` algorithm has parts whose absolute run time depends on P, not N.

process. Thus, in contrast to say an explicit time step in a PDE solve, it is hard to determine which mechanism is dominant and what the ideal scaling would look like. The main statement that we would like to make is that our partition function is extremely fast in terms of absolute run time: below one second for $0.5 \cdot 10^{12}$ elements on the full size of JUQUEEN.

`p4est` is a portable code written in C using standard MPI. The basic functionality requires MPI version 1.1, with optional MPI file I/O. We link against zlib for compressing VTK output. Saving a mesh using `p4est_save` uses MPI I/O, while we use one file per rank when writing VTK pvtu/xml graphics.

`p4est` is free software and used in many applications, among them finite volume methods [7], higher order finite element [8] and spectral methods [9]. The latter two have been scaled to 1.57 and 3.14 million MPI ranks on Sequoia and Mira, respectively. `p4est` has been the meshing code demonstrated in ACM Gordon Bell Prize finalists in 2008, 2010 and 2012, and the prize winner for 2015 [8].

Results

In managing the mesh metadata, the `p4est` code handles an essential part of the numerical pipeline. The main requirement is that the parallel meshing algorithms do not slow down a simulation, thus we aim for small run times in absolute terms. Even on the biggest meshes, our algorithms require on the order of seconds to run, down to well below one second for realistic examples. Our main focus is thus to establish scalability to the largest possible problem sizes and to verify that the `p4est` algorithms contribute only a negligible fraction to a simulation's run time.

The test configurations

We describe briefly the tests that we planned to run during the Extreme Scaling workshop.

1. Construct a 3D coarse mesh of 4,580 trees from a tetrahedral mesh of a cube-shaped domain consisting of 1,145 tetrahedra. Create a load-balanced uniform refinement of this mesh at a given initial level (`New`) and then perform one adaptive refinement step (`Refine`). In this refinement step we refine those mesh cells that lie in a cone with tip in the middle of one side of the domain and base on the other, see Figure 2. As a last step we load-balance the refined mesh (`Partition`).

2. With the same configuration use the full JUQUEEN system to construct a big mesh of over $9.4 \cdot 10^{11}$ elements. This would be the largest mesh created with `p4est` so far.

3. Do a similar cone refinement pattern with a coarse mesh of $\sim 2 \cdot 10^6$ trees generated from the Stanford bunny mesh (see Figure 2). This mesh has an impractically large tree connectivity that currently has no relevance.

4. In 2D uniformly refine a coarse mesh of 5 trees modelling a Moebius band geometry to a given level, partition the mesh and run the `ghost_exchange` algorithm to exchange data between ghost elements. We used a data size of 4096 bytes per ghost element.

The first three configurations are designed to read the Tetgen [5] file format to preprocess the coarse mesh of octrees. This format is by design non-parallel, thus we opted for reading it on one processor and broadcast it to avoid loading the file system with redundant I/O. Given that the largest coarse mesh we used has under 500k trees, the total run time of reading and broadcasting the mesh was always below 0.1 seconds.

Realization of the tests

The first day of the workshop was used to set up the example applications and a short strong scaling test for test configuration 1. To fine tune the application and input parameters we did several test runs on one JUQUEEN rack using 128 to 32,768 MPI ranks (32 ranks per node).

After this initialization phase we scaled the test configuration 1 to 16 racks (524,288 MPI ranks) during the day and set up scaling runs on up to 24 racks over night (the full system was not available at this point). Results are shown in Table 1. We could run this configuration on the full JUQUEEN system later in the workshop.

On the second day we set up test configurations 2, 3, and 4. The mesh with $9.4 \cdot 10^{11}$ elements could be created successfully on 28 JUQUEEN racks with 32 MPI ranks per node. Results are shown in Table 2. When using smaller numbers of racks the application ran out of memory due to the size of the mesh.

Similar memory limitations were found when testing configuration 3 with the Stanford bunny. The coarse mesh seemed too big to fit into the 16 GiB memory of JUQUEEN nodes.

At the end of the second day and during the third day of the Extreme Scaling workshop we set up configuration 4 for testing our new `ghost_exchange` function. After first tests with smaller data sizes we set up strong and weak scaling runs using 4 kbytes of data per ghost element. During the workshop we ran on up to 16 racks using 32 ranks per node and in the week after the workshop we set up runs on the whole 28 racks.

The results in Table 3 show that the absolute run time of `ghost_exchange` is always below 62ms, even on meshes with $2.1 \cdot 10^{10}$ elements. These times are so small that a standard scaling plot would be dominated by measurement and execution noise.

Our tests with `ghost_exchange` in 3D ran into MPI errors that we will investigate more closely. This is somewhat puzzling since the code is mostly dimension independent.

Further Notes

During the workshop we faced several issues. In the second night several jobs crashed immediately after start, which was observed by other groups as well and seemed to be a transient issue as resubmitted jobs ran as expected.

Reading the Stanford bunny mesh with $2 \cdot 10^6$ trees did not work, since the application ran out of memory due to the size of the coarse mesh. Given more time we would have been able to generate a smaller mesh of the same input data to run our tests.

As described above before running jobs on 16 and more racks we tested our configurations with smaller refinement levels on 1 rack using between 128 and 32k ranks with and without debugging mode enabled (assertions and extra verification).

As a secondary project we would have liked to test to what extent the MPI-3 shared memory features can save memory when running more than one MPI process per node. We set up a small test program to create a shared memory array and measure memory usage. However, since the shared memory required by this particular test is on the order of 10^2 bytes and its measurement (using `Kernel_GetMemorySize`) displays the used memory on a scale of 10^4 bytes we could not obtain useful results. Due to a tight schedule we did not run more tests with other memory sizes, but we plan to further investigate the shared memory features in the future.

Table 1: Strong scaling run time results for the 4,580-tree box mesh from Figure 2 (left). We generate a distributed uniform level 8 mesh (`New`), refine once more according to the given cone shape (`Refine`) and load-balance (`Partition`). The mesh sizes are $7.68 \cdot 10^{10}$ elements before the final refinement and $1.18 \cdot 10^{11}$ afterwards.

Racks	MPI ranks	`New`	`Refine`	`Partition`
8	262,144	0.486s	2.25s	3.18s
14	458,752	0.722s	1.35s	3.28s
16	524,288	0.802s	1.12s	2.69s
20	655,360	0.980s	1.01s	3.11s
24	786,432	1.158s	0.89s	2.91s
28	917,504	1.366s	0.82s	2.82s

Table 2: We manage a mesh of over $9.4 \cdot 10^{11}$ elements. This mesh is created from 4,580 trees, first refined uniformly to level 9 and then refined once adaptively.

Racks	MPI ranks	# mesh Elements	`New`	`Refine`	`Partition`
28	917,504	940,642,225,005	1.64s	5.60s	14.2s

Table 3: Run time results for `ghost_exchange`. 'lvl' refers to the uniform refinement level of the mesh used. The total number of mesh elements is $5 \times 4^{\text{lvl}}$, ranging between $3.4 \cdot 10^8$ (level 13) to $2.1 \cdot 10^{10}$ (level 16).

racks	MPI ranks	`Exchange`	lvl
4	131,072	9.4ms	13
8	262,144	23.6ms	13
16	524,288	20.4ms	13
4	131,072	15.0ms	14
8	262,144	16.0ms	14
16	524,288	38.2ms	14
28	917,504	34.9ms	14
4	131,072	30.2ms	15
8	262,144	29.2ms	15
16	524,288	42.4ms	15
28	917,504	53.9ms	15
28	917,504	61.4ms	16

Conclusions

This workshop provided us with the opportunity to generate and publish latest results on scalability. While effective development of new features within the code was not possible given the fixed schedule of submitting jobs, we were able to obtain new information on routines and configurations that are not usually covered by the production usage of the code.

We have executed the functions `New`, `Refine`, `Partition`, and `ghost_exchange` implemented by the `p4est` AMR code. We have worked with coarse meshes on the order of 5k trees and created, refined, and partitioned meshes to sizes between 100 and 940 billion elements. Absolute run times of all meshing operations are between a few milliseconds and several seconds depending on the configuration.

References

[1] Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. `p4est`: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

[2] Tobin Isaac, Carsten Burstedde, and Omar Ghattas. Low-cost parallel algorithms for 2:1 octree balance. In *Proceedings of the 26th IEEE International Parallel & Distributed Processing Symposium*. IEEE, 2012. `http://dx.doi.org/10.1109/IPDPS.2012.47`.

[3] Tobin Isaac, Carsten Burstedde, Lucas C. Wilcox, and Omar Ghattas. Recursive algorithms for distributed forests of octrees. *SIAM Journal on Scientific Computing*, 37(5):C497–C531, 2015.

[4] Carsten Burstedde, Georg Stadler, Laura Alisic, Lucas C. Wilcox, Eh Tan, Michael Gurnis, and Omar Ghattas. Large-scale adaptive mantle convection simulation. *Geophysical Journal International*, 192(3):889–906, 2013.

[5] Hang Si. *TetGen—A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator*. Weierstrass Institute for Applied Analysis and Stochastics, Berlin, 2006.

[6] Hari Sundar, George Biros, Carsten Burstedde, Johann Rudi, Omar Ghattas, and Georg Stadler. Parallel geometric-algebraic multigrid on unstructured forests of octrees. In *SC12: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, UT, 2012. ACM/IEEE.

[7] Carsten Burstedde, Donna Calhoun, Kyle T. Mandli, and Andy R. Terrel. Forestclaw: Hybrid forest-of-octrees AMR for hyperbolic conservation laws. In Michael Bader, Arndt Bode, Hans-Joachim Bungartz, Michael Gerndt, Gerhard R. Joubert, and Frans Peters, editors, *Parallel Computing: Accelerating Computational Science and Engineering (CSE)*, volume 25 of *Advances in Parallel Computing*, pages 253 – 262. IOS Press, March 2014.

[8] Johann Rudi, A. Cristiano I. Malossi, Tobin Isaac, Georg Stadler, Michael Gurnis, Peter W.J. Staar, Yves Ineichen, Costas Bekas, Alessandro Curioni, and Omar Ghattas. An extreme-scale implicit solver for complex PDEs: highly heterogeneous flow in earth's mantle. In *Proceedings of the SC15 International Conference for High Performance Computing, Networking, Storage and Analysis*, article 5. ACM, 2015.

[9] Andreas Müller, Michal A. Kopera, Simone Marras, Lucas C. Wilcox, Tobin Isaac, and Francis X. Giraldo. Strong scaling for numerical weather prediction at petascale with the atmospheric model NUMA. http://arxiv.org/abs/1511.01561, 2015.

[10] The Stanford University Computer Graphics Laboratory. Stanford bunny dataset, 1994. `http://graphics.stanford.edu/data/3Dscanrep/`, last accessed Feb 16, 2016.

[11] James R. Stewart and H. Carter Edwards. A framework approach for developing parallel adaptive multiphysics applications. *Finite Elements in Analysis and Design*, 40(12):1599–1617, 2004.