OpenFPM for scalable particle-mesh codes on CPUs and GPUs

Pietro Incardona[1,2], Ivo F. Sbalzarini[1,2,3]

[1] Chair of Scientific Computing for Systems Biology, Faculty of Computer Science, TU Dresden, Germany
[2] MOSAIC Group, Center for Systems Biology Dresden, Germany
[3] Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany

Introduction:

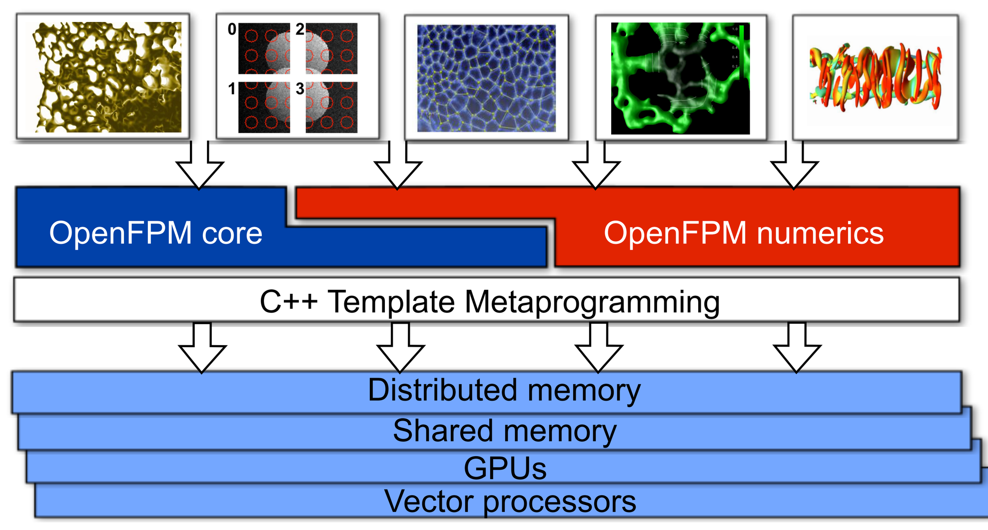


Fig 1: OpenFPM [1] is a general-purpose software platform for scalable parallel numerical simulations using particle and particle-mesh methods. Client codes (top row) from various application domains, from biology over image processing to fluid mechanics, access the distributed data structures of the OpenFPM core and the numerical solvers of OpenFPM-numerics via standardized APIs. C++ Template Meta Programming is used for compile-time code targeting to different hardware platforms.

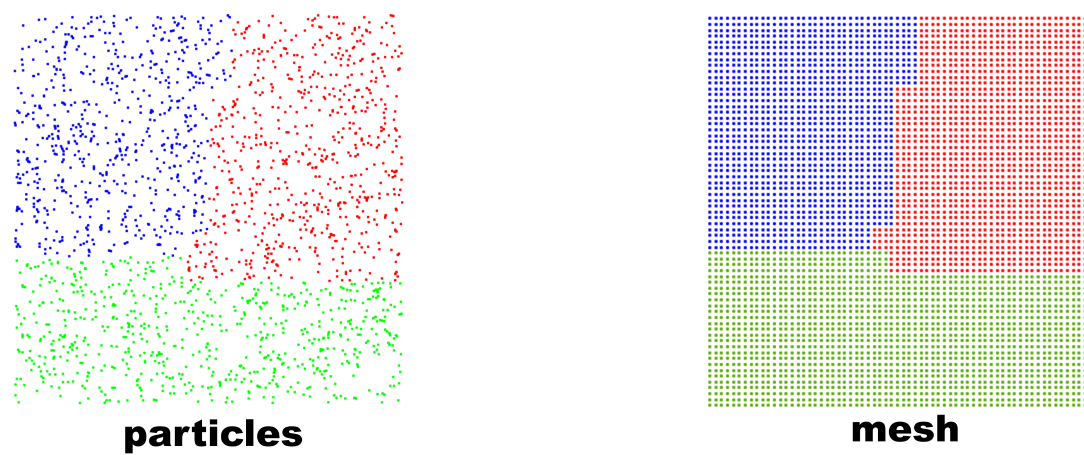


Fig 2: Example of particle (left) and mesh (right) data structures distributed across 3 processes in 2D.

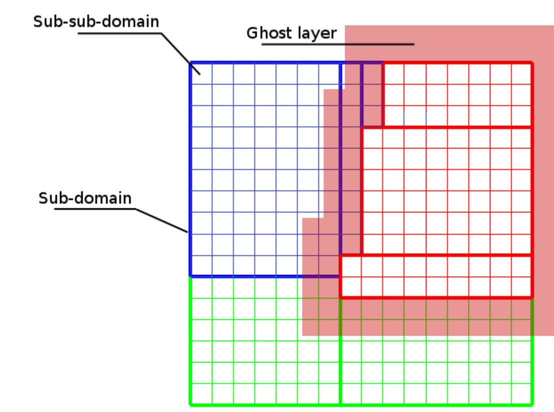


Fig 3: Sketch of the domain decomposition of the data structures from Fig. 2. In OpenFPM, the simulation domain is first divided into **sub-sub-domains** (small squares). From that, a communication adjacency graph is created and partitioned into subgraphs using the METIS graph-decomposition library. Sub-sub-domains are then merged to larger **sub-domains** (bold lines) in order to reduce data fragmentation. Communication between neighboring processes is handled by ghost layers (a.k.a. halo layers) around each set of subdomains assigned to the same processor (shaded areas shown for the red processor).

Project Goals:

1) Reduce development times for parallel numerical simulations from years to days [4].
2) Make HPC more accessible to computational scientists without parallel programming expertise [4].
3) Improve portability and reproducibility of numerical simulations in scientific computing [5].
4) Provide a successor for the classic PPM library (Fortran95, [2]) that uses modern software engineering principles and relaxes PPM's most salient limitations.
5) Enable transparent accelerator programming using domain-level abstractions [4].

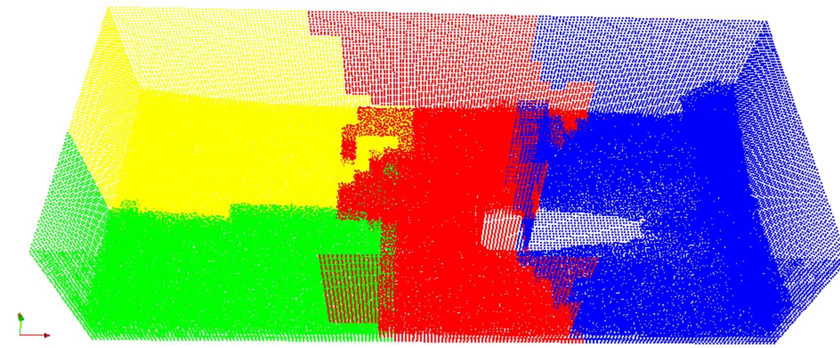


Fig 4: Runtime adaptive domain decomposition continuously adjusts the data distribution (example: SPH dam break, see below).

NEW: Multi-(CPU/GPU) support:

Non-kernel based: (Multi-CPU)

```
Box<2,double> domain({0.0,0.0},{1.0,1.0});
Ghost<2,double> ghost(0.1);
size_t bc[2] = {NON_PERIODIC,NON_PERIODIC};
vector_dist<2,double,aggregate<double[2]>> vd(1000,domain,bc,ghost);

auto it = vd.getDomainIterator()
while (it.isNext())
{
    auto p = it.get();
    vd.getPos(p)[0] = rand() / RAND_MAX;
    vd.getPos(p)[1] = rand() / RAND_MAX;
    ++it;
}
vd.map();
vd.ghost_get<>();

auto NN = vd.getCellList(0.1);

auto it = vd.getDomainIterator()
while (it.isNext())
{
    auto p = it.get();
     Point<2,double> xp = vd.getPos(p);
     Point<2,double> force_tot({0.0,0.0});

     auto NN_it = NN.getNNIterator(NN.getCell(xp))
     while (NN_it.isNext())
     {
            auto q = NN_it.get();

            Point<2,double> xq = vd.getPos(q);
            force_tot += (xp - xq) / norm2(xq - xp) * exp( - norm2(xq - xp) / 0.03)

            ++NN_it;
     }

        vd.getProp<0>(p)[0] = force_tot.get(0);
        vd.getProp<0>(p)[1] = force_tot.get(1);

    ++it;
}
```

kernel based: (Multi-CPU/GPU)

```
vd.deviceToHostPos()
vd.map(RUN_ON_DEVICE);
vd.ghost_get<>(RUN_ON_DEVICE);

auto NN = vd.getCellListDevice(0.1);
auto it = vd.getDomainIteratorDevice()
KERNEL_LAUNCH(calc_forces,ite.wthr,ite.thr,vd.toKernel(),NN.toKernel())
```

```
        template<typename vector_type, typename NN_type>
        __global calc_forces(vector_type vd, NN_type NN)
        {
                auto p = GET_PARTICLE(vd);

                Point<2,double> xp = vd.getPos(p);
                Point<2,double> force_tot({0.0,0.0});

                auto NN_it = NN.getNNIterator(NN.getCell(xp))
                while (NN_it.isNext())
                {
                        auto q = NN_it.get();

                        Point<2,double> xq = vd.getPos(q);

                        force_tot += (xp - xq) / norm2(xq - xp) * exp( - norm2(xq - xp) / 0.03)

                        ++NN_it;
                }

                vd.getProp<0>(p)[0] = force_tot.get(0);
                vd.getProp<0>(p)[1] = force_tot.get(1);
        }
```

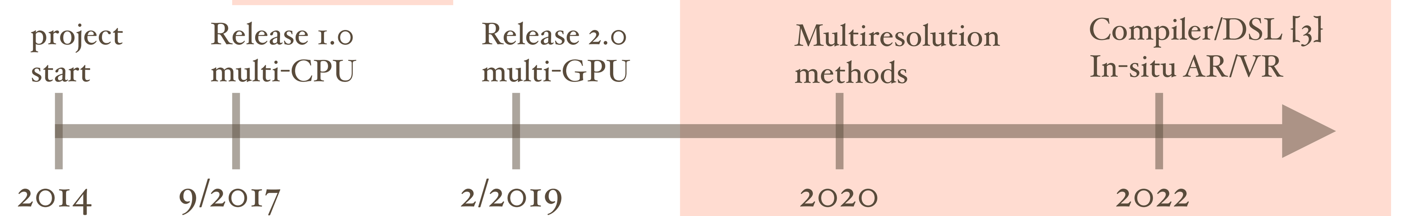
Vectorized (like Matlab, numpy): CPU/GPU

```
vd.deviceToHostPos()
vd.map(RUN_ON_DEVICE);
vd.ghost_get<>(RUN_ON_DEVICE);

auto NN = vd.getCellListDevice(0.1);

auto kernel = [](Point<2,double> & p, Point<2,double> & q) -> Point<2,double>
{return (xp - xq) / norm2(xq - xp) * exp( - norm2(xq - xp) / 0.03)}
auto Force = getV<0>();
Force = applyKernel_in(vd,NN,kennel);
```

Methodology / Approach:

OpenFPM provides **scalable distributed and parametric data-structures** using **C++ Template Meta Programming**. Code targeting to different hardware platforms **(CPU/GPU/ Accelerators)** is transparently supported for particle and mesh methods (see examples on the left). All data-structures can be used in **arbitrary dimensions** and particles/meshes (Fig. 2) can store scalars, vectors, and tensors of any rank and any custom data type (i.e., any C++ class). OpenFPM automatically distributes the data structures across multiple machines by decomposing the simulation domain (Fig. 3). It provides transparent communication abstractions, remote information query, and **dynamic runtime load balancing** (Fig 4).

Timeline / Roadmap:

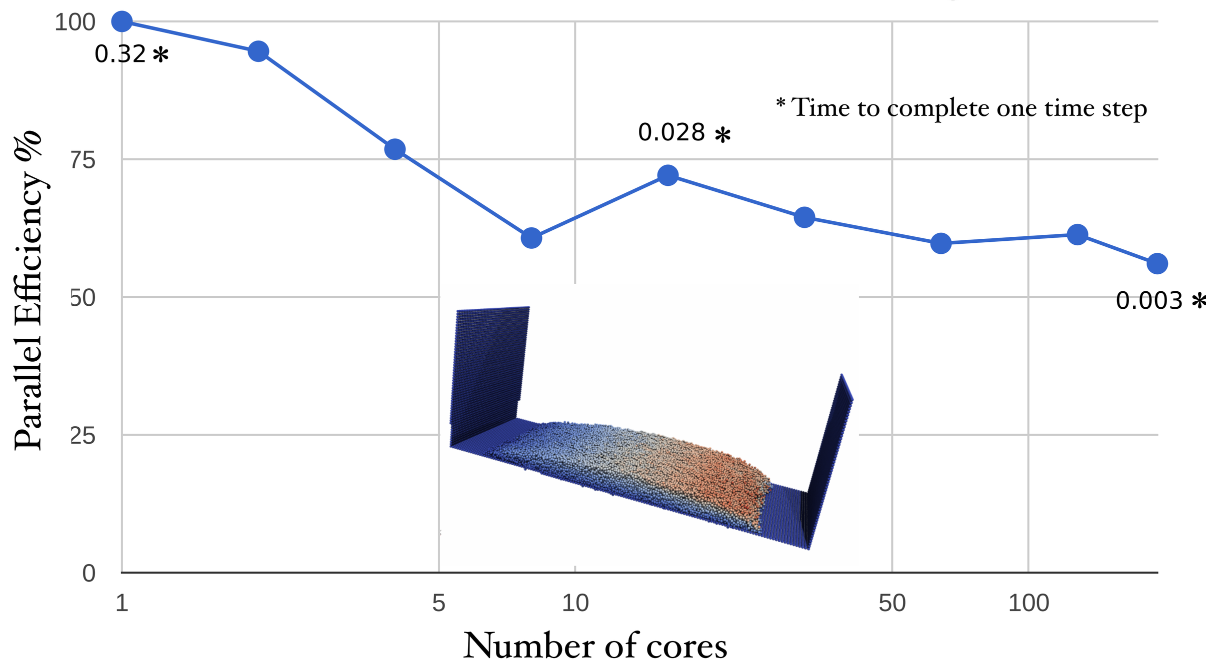


LEFT: Code example to show a particle calculation on different types of hardware. The first loop initializes the particles at random positions using a particle domain iterator. The colored part of the code is shown in three variants: distributed-memory parallelism without kernels (multi-CPU), kernel-based code akin to CUDA, and a kernel-free accelerator version using lambda expressions and template parsing to enable direct mathematical input instead of loops. Color codes are according to the legend below:

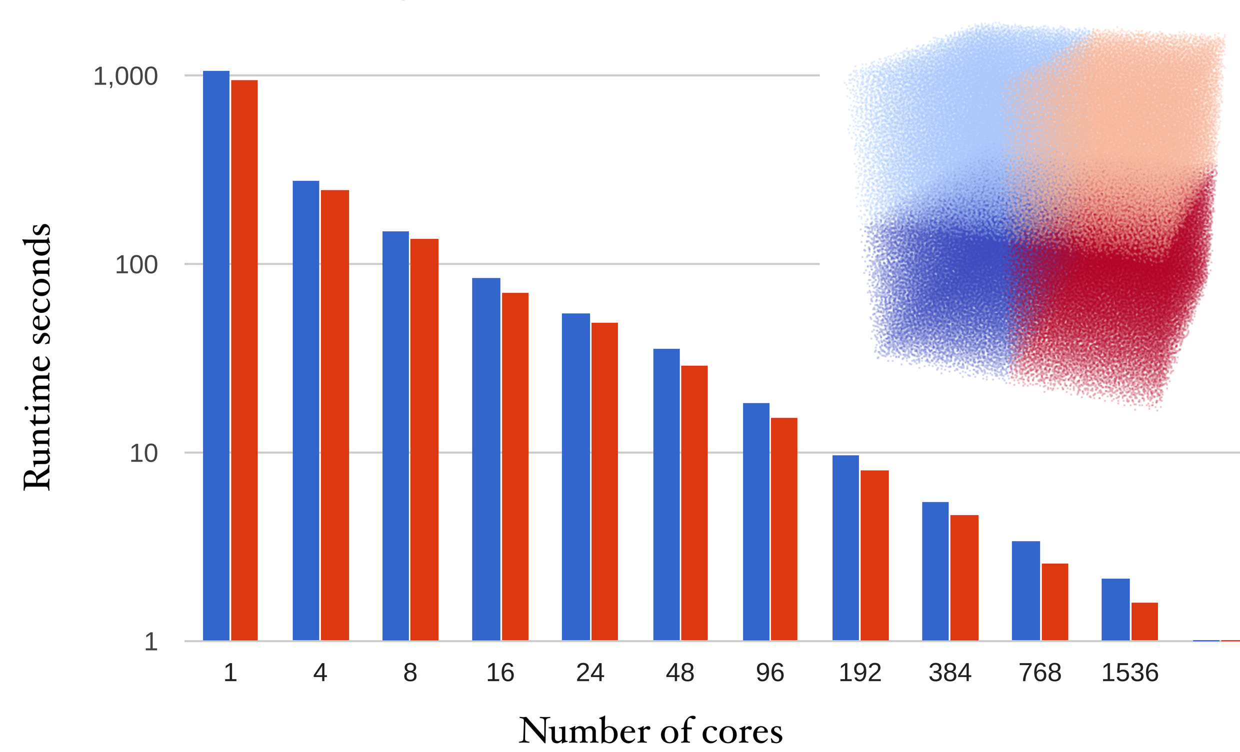
- Serial/algorithm instructions
- Parallelization multi-CPU/GPU instructions
- Accelerator/GPU instructions

Benchmarks:

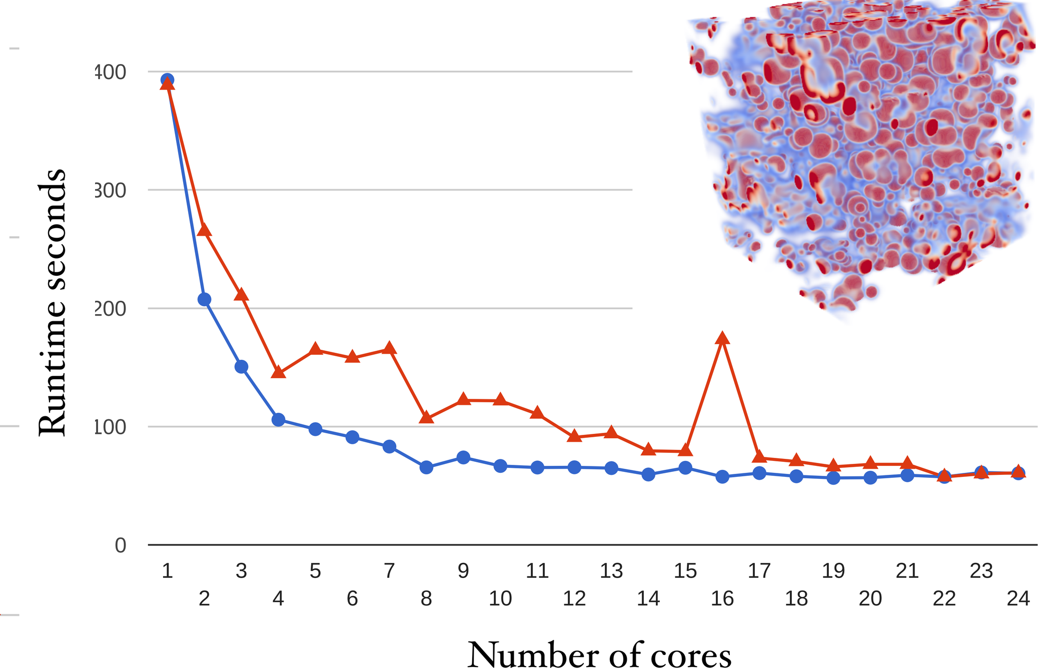
Discrete element simulation, Silbert grain model



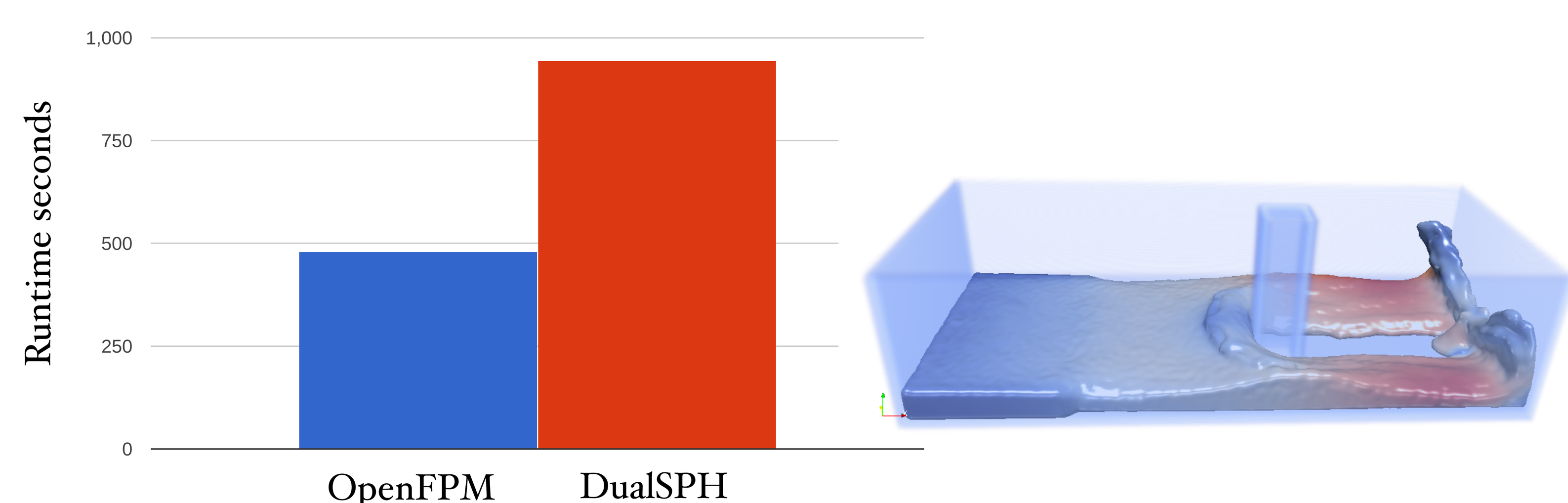
Molecular Dynamics, Lennard-Jones potential



Gray-Scott Reaction-Diffusion model
Finite difference stencil code



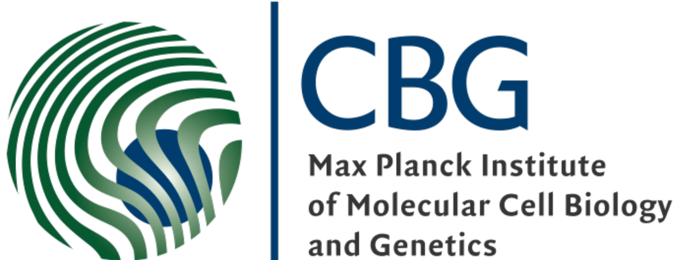
Smoothed-particle hydrodynamics, Dam break simulation



In **blue** we show the performance of OpenFPM, in **red** we compare with other frameworks. top-middle: LAMMPS, top-right: AMReX, bottom-left: DualSPH

The project is open source and available at:

openfpm.mpi-cbg.de



References:

1. P. Incardona, A. Leo, Y. Zaluzhnyi, R. Ramaswamy, I. F. Sbalzarini, OpenFPM: A scalable open framework for particle and particle-mesh codes on parallel computers, *arXiv*:1804.07598, *Comput. Phys. Commun. (in print)*, 2019.
2. I. F. Sbalzarini, J. H. Walther, M. Bergdorf, S. E. Hieber, E. M. Kotsalis, and P. Koumoutsakos. PPM – A Highly Efficient Parallel Particle-Mesh Library for the Simulation of Continuum Systems, *J. Comput. Phys.* 215(2):566-588, 2006.
3. S. Karol, T. Nett, P. Incardona, N. Khouzami, J. Castrillon, I. F.Sbalzarini, A language and development environment for parallel particle methods, in: *International Conference on Particle-based Methods – Fundamentals and Applications*, Hanover, Germany, 2017, pp. 1–12.
4. I. F. Sbalzarini. Abstractions and middleware for petascale computing and beyond. *Intl. J. Distr. Systems & Technol.*, 1(2):40–56, 2010.
5. S. Karol, T. Nett, J. Castrillon, and I. F. Sbalzarini. A domain-specific language and editor for parallel particle methods. *ACM Trans. Math. Softw.*, 44(3):34, 2018.