ARTICLE IN PRESS

Computer Physics Communications ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Highly efficient spatial data filtering in parallel using the opensource library CPPPO☆

Federico Municchi[a,*], Christoph Goniva[b], Stefan Radl[a]

[a] Institute of Process and Particle Engineering, Graz University of Technology, Inffeldgasse 13/III, 8010 Graz, Austria

[b] DCS Computing GmbH, Industriezeile 35, 4020 Linz, Austria

ARTICLE INFO

Article history:
Received 16 September 2015
Received in revised form
18 May 2016
Accepted 26 May 2016
Available online xxxx

Keywords:
Multi-scale
Closure models
Parallel filtering
Post processing
MPI

ABSTRACT

CPPPO is a compilation of parallel data processing routines developed with the aim to create a library for "scale bridging" (i.e. connecting different scales by mean of closure models) in a multi-scale approach. CPPPO features a number of parallel filtering algorithms designed for use with structured and unstructured Eulerian meshes, as well as Lagrangian data sets. In addition, data can be processed on the fly, allowing the collection of relevant statistics without saving individual snapshots of the simulation state. Our library is provided with an interface to the widely-used CFD solver OpenFOAM®, and can be easily connected to any other software package via interface modules. Also, we introduce a novel, extremely efficient approach to parallel data filtering, and show that our algorithms scale super-linearly on multi-core clusters. Furthermore, we provide a guideline for choosing the optimal Eulerian cell selection algorithm depending on the number of CPU cores used. Finally, we demonstrate the accuracy and the parallel scalability of CPPPO in a showcase focusing on heat and mass transfer from a dense bed of particles.

Program summary

Program title: CPPPO

Catalogue identifier: AFAQ_v1_0

Program summary URL: http://cpc.cs.qub.ac.uk/summaries/AFAQ_v1_0.html

Program obtainable from: CPC Program Library, Queen's University, Belfast, N. Ireland

Licensing provisions: GNU Lesser General Public License, version 3

No. of lines in distributed program, including test data, etc.: 1043965

No. of bytes in distributed program, including test data, etc.: 11053655

Distribution format: tar.gz

Programming language: C++, MPI, octave.

Computer: Linux based clusters for HPC or workstations.

Operating system: Linux based.

Classification: 4.14, 6.5, 12.

External routines: Qt5, hdf5-1.8.15, jsonlab, OpenFOAM/CFDEM, Octave/Matlab

Nature of problem:
Development of closure models for momentum, species transport and heat transfer in fluid and fluid–particle systems using purely Eulerian or Euler–Lagrange simulators.

Solution method:
The CPPPO library contains routines to perform on-line (i.e., runtime) filtering and compute statistics on large parallel data sets.

☆ This paper and its associated computer program are available via the Computer Physics Communication homepage on ScienceDirect (http://www.sciencedirect.com/science/journal/00104655).

* Corresponding author.
E-mail address: fmunicchi@tugraz.at (F. Municchi).

http://dx.doi.org/10.1016/j.cpc.2016.05.026

Running time:
Performing a Favre averaging on a structured mesh of 128^3 cells with a filter size of 64^3 cells using one Intel Xeon(R) E5-2650, requires approximately 4 h of computation.

1. Introduction

Many relevant physical systems involve a wide spectrum of length scales that interact in a non-linear way. Hence, an accurate prediction of all relevant phenomena in these physical systems in engineering-scale equipment is challenging due to the inability to directly simulate certain small-scale phenomena. One example is dense fluid–particle flows, which are usually encountered in many industrial processes: details of the flow around each individual particle cannot be directly predicted, but are modeled instead, e.g., by a drag coefficient. In addition, the simulation of flows in engineering-scale equipment often necessitates the use of Eulerian models on comparably coarse computational grids, i.e., the continuum hypothesis has to be adopted, and small-scale information is lost. Consequently, closures have to be derived to account for a variety of phenomena, e.g., fluid–particle and particle–particle interactions, or unresolved turbulent motion. In order to accurately model such flow problems, a so-called multi-scale approach is often used [1,2]. The multi-scale approach consists in decomposing the original problem into various levels of description, each one involving a typical range of length scales. Then, simulations on the most detailed level (typically on the smallest length scales) are performed to extract quantities which can be used in coarse grained models. In a coarse-grained model, only coarse flow structures are resolved (where "coarse" means on the same order of the mesh size or larger). Transport processes occurring at smaller scales are considered by closures, e.g., filter-size dependent closures for scalar dispersion rates, inter-phase exchange rates, or effective stresses. Nowadays, these closures are often derived from simulations on a more detailed level, and not from experimental data. This process is normally referred to as "coarse-graining", and has become a major trend in a variety of scientific disciplines [3–6].

CPPPO (i.e., the "Compilation of fluid/Particle PostPrOcessing routines") has been developed as a flexible library that provides a collection of efficient algorithms to perform these coarse-graining operations. The main purpose of CPPPO is to act as a tool for "scale-bridging", regardless of the effective scale range, the model formulation, or the simulator used. CPPPO is designed to interact with any purely Eulerian, or mixed Eulerian–Lagrangian data set. This allows one to apply CPPPO for a number of different scientific and engineering applications. For example, this includes the verification of Large Eddies Simulation (LES) models based on differential filtering [7], anisotropic filtering of flow data [8], or the development of sub-grid stress tensors for LES. In what follows, however, we focus on a multi-scale scenario applied to study dense fluid–particle flows in order to outline how CPPPO can be used for scale bridging.

At the most fundamental level, Direct Numerical Simulations (DNS) are used to derive coefficients for heat, momentum and mass transfer in dense particulate systems [9–14]. Typically, a certain number of realizations for the case studied are needed [15,16] in order to derive statistically meaningful correlations. This approach requires to process data from large data sets in order to compute averaged (mean) quantities (which are needed to evaluate transfer coefficient), standard deviations, or other statistics like the distribution of the angle between two vector fields [17]. Also, time-averaged quantities are often used to evaluate transport coefficients in fluid–particle systems [18]. In case of non-equilibrium systems (like fluidized beds, in which instabilities are system-inherent [19]), the modeling of drag and stresses may require higher-order closures. Unfortunately, these models are difficult to develop [20]. Another example can be found in the field of granular materials: here the calculation of effective transport properties requires the evaluation of filtered fields and fluxes [14,21]. The same approach, i.e., considering statistical data of, e.g., the velocity fluctuations, can be used on intermediate length scales when deriving models for engineering applications. Typically, this results in an Eulerian "grid coarsening" approach, e.g., by deriving models for the sub-grid-scale fluid–particle agitation [3]. Favre averaging of relevant fluid variables (e.g., the fluid velocity), and fluid–particle interactions (e.g., the coupling force) is generally adopted to derive these closures [22–24]. In case an Euler–Lagrange approach is followed, the effect of "particle coarsening" (i.e., each simulated particle is a proxy for a prescribed number of particles named parcel) has to be taken into account as well [23]. All these examples demonstrate that spatial averaging operation on Eulerian and Lagrangian data sets is of key importance for multi-scale model development nowadays.

In principle, the application of an appropriate filtering strategy is straight forward once the fluid–particle flow simulator is available. However, filtering of scientific data and "coarse-graining" poses several challenges from the software point of view. For example, spatio-temporal averages have to be computed across different processors for the (typically large) filter sizes. Typically, filter sizes to be used when filtering DNS data of fluid–particle flows have a size of two to five particle diameters. Thus, filtering is typically performed over 20^3–50^3 Eulerian grid cells, often located on different processors. This requires an algorithm that can deal with parallel communication, and that does not require mirroring the full field information on every processor. The latter is of course a feasible approach, however, when aiming on large-scale simulations this would require an excessive use of RAM. At the same time, parallel communication of local field values requires significant network resources due to the large amount of data to communicate. Another issue is the amount of data generated during the simulation run: the implemented algorithms should be able to work "on the fly" in order to process (i.e., time-average) data from different time steps. Also, there should be a clear separation (in terms of namespaces and classes) between the simulator and the post-processing utility such that the latter can be linked to different simulators. Finally, the library should be modular in order to make the addition of new features as easy as possible.

In the present work we present the library CPPPO that addresses the above challenges. While most of the existing filtering algorithms documented in literature were developed for image processing applications [25,26], CPPPO is able to handle three-dimensional data sets in parallel. Specifically, CPPPO can process data sets from complex geometries and unstructured meshes. Another important difference with respect to image filtering is the type of data: images just deal with a limited set of scalar quantities represented by integers (i.e., the color intensity). In contrast,

Please cite this article in press as: F. Municchi, et al., Highly efficient spatial data filtering in parallel using the opensource library CPPPO, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.05.026

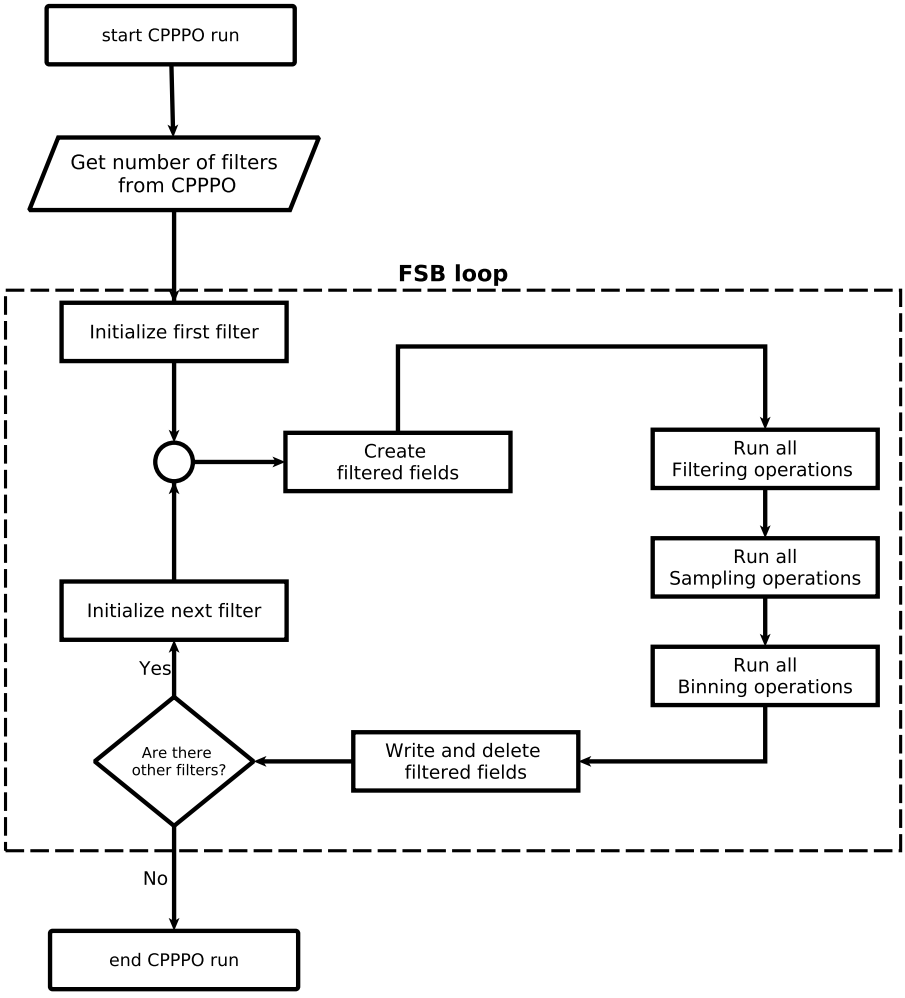

Fig. 1. Filtering–sampling–binning loop: the structure of a typical CPPPO run.

relevant simulation results are vectorial data that is represented by floating point values. Furthermore, CPPPO handles cell/particle selectors separately from filtering routines to allow an easier implementation of custom filtering kernels, and a higher flexibility in the choice of the algorithm. In summary CPPPO features a flexible code architecture tailored for scientific computations on high performance clusters. The library is designed to perform three kinds of operations on the data set:

- Filtering: Field volume averaging that can be performed on every cell (for the Eulerian filter option) or at specific user-defined locations (for the Lagrangian filter option). The user can customize the kernel function (see Section 3) by adding an arbitrary amount of weights (which have to be scalar fields) or by modifying the kernel's functional form.
- Sampling: This operation allows to take samples from the domain (results from the filtering operations may be sampled as well) and relate each sampled value with one or more markers. For example, CPPPO can sample a spatially-filtered fluid velocity field at every cell using the fluid phase fraction as marker.
- Binning: Data collected from sampling operations can be collapsed using binning operations. The marker field values are discretized according to the user input, and a conditional averaging calculation is performed on the sampled field. This data collapsing allows to reduce the amount of data that needs to be written to disk in case the user is only interested in correlations between the means of the sampled quantities and one (or more) markers.

For every user-specified filter (i.e., kernel function), the library performs these three operations in sequence (see Fig. 1). We will refer to this loop as the FSB loop (Filtering, Sampling and Binning).

Our paper is structured as follows. In Section 2, the basic data structure and the range of applicability of the library are discussed. In Section 3, the basics of filtering and Favre-averaging are introduced, as well as the novel *divergent* approach to parallel filtering implemented in CPPPO. Available routines for statistics calculation are described in Section 4, and the *sampling/binning* process is outlined. The implementation of the algorithms for cell selection and filtering is presented in Section 5, with emphasis on the parallelization strategy. In Section 6 several simple test cases for code verification are described, and in Section 7 we present a parallel scalability analysis for CPPPO. We also present a typical application of CPPPO in Section 8 for the evaluation of heat and mass transfer coefficients. Finally, conclusions are summarized in Section 9.

2. Library interface to simulators

Before detailing the algorithms available in CPPPO, it is worth describing the conditions that a simulator needs to meet in order to be linked to CPPPO. In the following, we will also describe the general set of data to which the algorithms can be applied.

2.1. Basic data structure

CPPPO can be coupled to simulators using finite volume, finite difference, finite elements, spectral, lattice Boltzmann or smoothed

Please cite this article in press as: F. Municchi, et al., Highly efficient spatial data filtering in parallel using the opensource library CPPPO, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.05.026

particle hydrodynamics methods. These methods normally consist in solving partial differential equations within a computational domain Ω_c of volume V_{Ω_c}. The library requires the simulator to provide the following data:

- A set of nodes (points) lying within Ω_c, each one identified with a set of three spatial coordinates.
- A set of scalars representing the measure of the spatial volume surrounding each node. Notice that, in order to correctly calculate spatial filtered quantities, the volumes must not overlap and their sum must be equal to the total computational volume V_{Ω_c}.
- A set of scalars representing field values (e.g., pressure, temperature, velocity components, species concentration, *et cetera*) at each node.

In the following, we will refer to the entity composed of a node and the associated volume as *cell*. The union of all cells is termed as *mesh* and field values associated with each node are named *cell values*. Mesh and cell values form the *Eulerian* data in CPPPO.

Notice that CPPPO does not require any information regarding cell shape or surfaces. The topological details of Ω_c or the original mesh are not considered and a cell is considered to lie within a certain region if its node is included in that region. Thus, filtering operations are affected by errors due to: (i) cell shape (or cell quality) and (ii) the ratio between cell size and filter size. However, this is not really an issue since (i) is generally controlled in the simulator in order to reduce numerical errors in the computation (before running CPPPO) and (ii) should always be low due to cell shape regularity required in (i) and the fact that filtering volumes are often much larger than smallest field structures (which normally require lumped nodes and thus, small cells, to be sufficiently resolved).

Additionally, the user can provide a set of *Lagrangian* (particle) data which may represent particle clouds or sampling probes. While probes are just defined by their position in Ω_c, particle clouds can be defined with several more properties (like particle diameter, velocity, torque, forcing terms, and scalars) which can be passed to CPPPO directly from the simulator. These properties can be used, for example, to calculate inter-phase transfer coefficients "on-the-fly".

Further information on data structure can be found in the CPPPO documentation.

2.2. General linking architecture

CPPPO has his own way of handling field, mesh and particle data which, in general, does not have to conform to any particular simulation software. In order to exchange data between CPPPO and a simulator, an *interface library* is required. The interface library is specific for every simulator, and it will typically rely on the simulator's classes and namespaces. CPPPO comes with an interface library for OpenFOAM®.

The role of the interface library is to get pointers to memory locations of all relevant data fields (e.g., holding mesh and particle information), and pass them to the core library in an appropriate format. For example, vector fields require pointers to the array of doubles containing each component. Similarly, for the mesh data pointers to coordinates and volumes of every cell are transferred. Some simulators store all the components of a vector or all the mesh point coordinates in just one array, one example of which is OpenFOAM®. CPPPO allows to specify a displacement between the component values (for example a displacement of 3 for three dimensional data) in order to automatically take into account this data structure. In addition, the interface library allocates space for the filtered fields (i.e., those fields that store the result of filtering operations) and registers them (i.e., pass the required pointers) into the CPPPO core library. Thus, the interface library performs additional storage operations using the simulator namespace. In this way, the resulting filtered fields can be saved and used in the simulator format, which positively contributes to the usability of CPPPO.

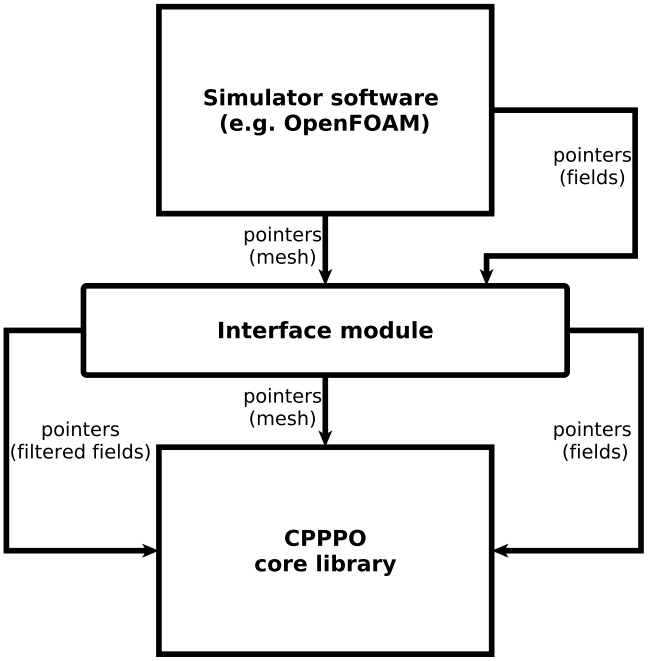

Fig. 2. Flow of information from the simulator to the interface and, finally, to the CPPPO core library.

The CPPPO core library performs filtering, sampling and binning operations using the memory allocated by the interface library and the simulator. The core library allocates heap memory for the filtering operations. For example every Eulerian filtering operation (i.e., where a filter is centered at every cell's center) requires an array of doubles with size equal to the number of cells to store intermediate values (see Section 3 for details). This data flow is summarized in Fig. 2.

2.3. Parallel data handling

The CPPPO core library represents the domain as a set of nodes which, in the case of the OpenFOAM® interface, correspond to the cell centers. CPPPO ignores the cell shape, but requires the interface to provide cell volumes. When a filtering or searching operation is performed, the cells with the closest cell center are selected.

CPPPO is designed for applications holding parallel-decomposed data (i.e., a physical domain is subdivided into smaller subdomains) where the domain subdivision is made of boxes whose faces are perpendicular to the corresponding Cartesian axis. This is an important requirement since it ensures that CPPPO exactly knows the position of every processor boundary. Every box can contain a different number of cells, or have different size in any direction. However, in order to keep a high parallel efficiency, it is recommended to keep the same number of cells for every processor.

CPPPO is parallelized using MPI [27], and can be run in parallel with any simulator that splits the computational domain in several box-shaped subdomains. Since CPPPO is designed for spatial filtering, data should be decomposed according to their position in space. This is true in almost the totality of currently available simulators for CFD either using a finite volume approach (e.g., OpenFOAM®, ANSYS FLUENT®, *Code_Saturne*®, STAR-CCM+®,

Please cite this article in press as: F. Municchi, et al., Highly efficient spatial data filtering in parallel using the opensource library CPPPO, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.05.026

AVL FIRE®, etc.) or not (e.g., Palabos®, Nektar++®, Nek5000®, etc.).

CPPPO requires each process to have its memory address space. Thus, simulators which rely on a GPU hardware architecture, or OpenMP may not be suitable for linking with CPPPO at the current stage. Also, RMA (remote memory access), and MPI one-sided communications in general, will most likely create problems in case of passive synchronization. This is because a processor would have to call `MPI_Lock` to himself in order to access local data inside an MPI window. In summary, we recommend careful testing of the interface routines when using CPPPO in connection with simulators that rely on RMA or one-sided communication. For the standard interface (to OpenFOAM®) we only require that:

- The global domain is decomposed in box-shaped subdomains,
- Each process holds only one subdomain, and that
- Accessibility of local data is ensured.

In case a user wants to link CPPPO to a software that does not meet the above requirements, a more careful design of the interface library is necessary. For example, we have implemented an interface to CSV data files that performs the domain decomposition, and does not require the input file to be already decomposed in parallel. Also, the above mentioned issue with not accessible local data can be circumvented by copying the shared data in separate arrays, thus creating a compact addresses set for the whole subdomain. We next detail on some practical aspects when implementing such a new interface library.

2.4. Coding an interface library

In case the user wants to link CPPPO with a software, he/she will have to use the existing OpenFOAM® or CSV interface module, or code a new interface library. For the latter, the OpenFOAM® interface library is a good template. In the following, we will refer to this library to illustrate the main steps needed to code an interface library.

Remember that all the functions that are called in an interface module are summarized in *core/c3po.h*.

- An interface should create an instance of the *c3po* class.
- An interface should be able to access pointers to mesh data and pass them to CPPPO, which then uses them to calculate and communicate other required quantities. An example can be found in the file *interface_OF/mesh_check.C*. Notice that the OpenFOAM® interface provides a public function that can be called in the simulator. This is done to track the evolution of dynamic meshes and ensure that pointers handed over to CPPPO are always valid.
- An interface should possess a *run* function which (i) registers (i.e., handles relevant pointers to) the required fields in CPPPO according to their data format, and (ii) starts the FSB loop of CPPPO. The *interface_OF/c3po_OF_interface.C* file provides an example therefore.
- During the FSB loop, the interface should be able to allocate heap memory for the required fields (e.g., filtered fields), and delete them when necessary. Note that CPPPO will already provide suitable names to label these new fields.

In general, the amount of time required to code a new interface can vary significantly with the architecture of the simulation software and the programmer's skills. Thus, it is useful to first study the architecture of the simulator and CPPPO, e.g., by using the training material available at http://www.tugraz.at/en/institute/ippt/downloads-software/.

3. Spatial filtering

Spatial filtering can be considered as a subset of the general operation [28]:

$$\overline{\phi}\left(\mathbf{x},t\right)=\int K\left(\mathbf{x}-\mathbf{x}',t-t'\right)\phi\left(\mathbf{x}',t'\right)d\mathbf{x}'dt' \tag{1}$$

where ϕ is a generic field, K is the kernel function, and the integration is performed over the whole space and time domain. An important property of the kernel function is normalization, thus:

$$\int K\left(\mathbf{x}-\mathbf{x}',t-t'\right)d\mathbf{x}'dt'=1. \tag{2}$$

CPPPO will automatically normalize your kernel function. In the case of spatial filtering, the kernel function is expressed as:

$$K\left(\mathbf{x}-\mathbf{x}',t-t'\right)=K\left(\mathbf{x}-\mathbf{x}'\right)\delta\left(t-t'\right). \tag{3}$$

Thus, the argument is integrated over space only:

$$\overline{\phi}\left(\mathbf{x},t\right)=\int K\left(\mathbf{x}-\mathbf{x}'\right)\phi\left(\mathbf{x}',t\right)d\mathbf{x}'. \tag{4}$$

The corresponding fluctuating field ϕ'' is defined as:

$$\phi''\left(\mathbf{x},t\right)=\phi\left(\mathbf{x},t\right)-\overline{\phi}\left(\mathbf{x},t\right). \tag{5}$$

CPPPO solves Eq. (4) at every cell center $\mathbf{x}$, or alternatively at predefined positions $\mathbf{r}$. The kernel used in this study is the top-hat kernel:

$$K\left(\mathbf{x}-\mathbf{x}'\right)=\prod_i\frac{\mathcal{H}\left(\frac{\Delta_i}{2}-|x_i-x_i'|\right)}{\Delta_i} \tag{6}$$

where Δ_i is the filter size in the ith direction of a Cartesian coordinate system and $\mathcal{H}$ is the Heaviside function. CPPPO also features a top-hat kernel in a spherical coordinate system. It has to be noticed that this kernel, while acting as a sharp cut-off in the physical space, features a smooth cut-off in the spectral space [29], resulting in a wave number overlap between $\overline{\phi}$ and ϕ''.

3.1. Favre filtering

The Favre averaging technique [30,31] consists in a decomposition of the flow field variables in terms of density-weighted variables:

$$\widetilde{\phi}=\frac{\overline{\rho\phi}}{\overline{\rho}}. \tag{7}$$

Favre averaging is often used for multiphase flows to derive filtered transport equations, and to decouple the phase fraction from the flow variables. In addition, variance and covariance calculations are of major importance to evaluate the components of the SGS (Sub Grid Scale) stress tensor, or SGS fluxes. CPPPO is able to perform Favre averaging and Favre variance and covariance calculation for every cell inside the domain, or at specific user-defined positions. To illustrate the equivalence between the variance (or covariance) and the components of the SGS stress tensor, we consider the definition of the latter as:

$$\tau_{ij}^{sgs}=\widetilde{u_iu_j}-\widetilde{u}_i\widetilde{u}_j. \tag{8}$$

Here u_i is the velocity of in the spatial direction i. For example, the diagonal elements of the tensor shown in Eq. (8) can be obtained

Please cite this article in press as: F. Municchi, et al., Highly efficient spatial data filtering in parallel using the opensource library CPPPO, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.05.026

from the Favre variance as follows:

$$\begin{aligned} Var(u_i(\mathbf{x})) &= \int G(\mathbf{x}'-\mathbf{x})\left(u_i(\mathbf{x}')-\widetilde{u}_i(\mathbf{x})\right)^2 d\mathbf{x}' \\ &= \int G(\mathbf{x}'-\mathbf{x})u_i^2(\mathbf{x}')d\mathbf{x}' + \widetilde{u}_i^2(\mathbf{x}) \\ &\quad - 2\widetilde{u}_i(\mathbf{x})\int G(\mathbf{x}'-\mathbf{x})u_i(\mathbf{x}')d\mathbf{x}' \\ &= \widetilde{u_i u_i} - \widetilde{u}_i\widetilde{u}_i = \tau_{ii}^{sgs} \end{aligned} \tag{9}$$

where $G(\mathbf{x}'-\mathbf{x})$ is a function representing the top-Hat kernel and the Favre averaging operation. The other components of τ_{ij}^{sgs} can be calculated in a similar manner using the Favre covariances. The same approach applies to evaluate SGS fluxes. Notice that CPPPO allows the user to define an arbitrary number of weighting fields for the kernel function, and hence offers the freedom to compute filtered quantities for virtually any application.

3.2. Convergent and divergent filtering algorithm

Since the calculation of filtered quantities implies long range interactions, processor communication has to be taken into account when designing an algorithm to numerically solve Eq. (4). In case Eq. (4) is projected into the discrete space (and when considering a top-Hat kernel), it can be written as:

$$\overline{\phi_i} = \frac{\sum_{j=0}^{j=N_f} v_j\phi_j}{V_f} \tag{10}$$

where the sum is over all N_f cells inside the filter region, v_i is the volume of the ith cell, and V_f is the total filter volume. The extension of the above equation to Favre averaging or arbitrary weighted averaging is obvious. The above calculation has to be performed for every cell i in order to compute a complete field of the filtered quantity. Considering Eq. (10) it is clear that, before the calculation can start, it is necessary to evaluate which cells are inside the filter.

The approach described by Eq. (10) is what we call the *convergent approach* for filtering. This is because, after the list of cells inside the filter is assembled, data from the neighboring cells is passed to the location where the filter is centered. This approach requires (for every location to filter) $N_f - 1$ summation and multiplication operations, and one division operation. The amount of multiplication operations could be reduced in case the multiplied field values are stored and then communicated. However, this would require additional memory. Also, communication of the cell list and values with other processors has to be performed. As shown in Fig. 3 (left panel), the convergent approach requires the communication of every required cell data owned by another processor. In our case three values need to be transferred from processor 2 to processor 1. The convergent approach is the most basic approach for spatial filtering, and most of the available filtering algorithms used for image processing are based on it.

In order to reduce the computational load and enhance parallel efficiency, we developed a novel approach named the *divergent approach*. The divergent algorithm does not evaluate the filtered value at any position sequentially, but updates the filtered fields at every step, and ends with a final division step. Specifically, every step consists of:

(i) Selecting a cell from the computational domain.
(ii) Creating a list of cells located in the region to be filtered around the selected cell.
(iii) Multiplying the field value at the selected cell with the required weight (i.e., cell volume or mass density). Note, when the filter size tends to the domain size only one cell value needs to be stored and communicated instead of having to allocate and communicate the whole field.
(iv) Communicating and adding the multiplied field values to all cells inside the cell list

The loop has to be repeated for every cell inside the domain. At the end, one last step is needed to divide the values of filtered fields by the filter volume (or by the summed weights in case of Favre averaging). This approach requires (for every cell) $N_f - 1$ additions, but only one multiplication. The number of divisions in the final step equals the total number of cells. Overall, less multiplication operations are required in the divergent approach compared to a convergent approach (without allocating memory for the multiplied fields as explained above). Most important, the key advantage of the divergent algorithm over the convergent algorithm is the amount of data that needs to be communicated. As shown in Fig. 3 (right panel), in the divergent algorithm the direction of communication is reversed, and just the field value at the current cell needs to be communicated once. The communicated value is then processed locally on the relevant processor (in our case processor 2), which does not involve any communication overhead any more.

It should be clear that the computational bottleneck for these kinds of algorithms is not the number of standard operations, but the number of MPI operations. While image filtering algorithms tend toward a reduction in the number of standard operations, the algorithms implemented in CPPPO have the reduction of the number of MPI operations as the main goal. Since field and mesh data can be very large in terms of the consumed memory, it is often not feasible to rely on massive data copying and thus, processor communications are rather frequent. The number of MPI communications in CPPPO can be of the same order of magnitude as the mesh size. More details on the parallel implementation will be given in Section 5.

There are several differences between CPPPO, and other tools for averaging like those provided in OpenFOAM® (or sub-modules such as swak4Foam [32]). These modules can just calculate averages over lines, faces and volumes using predefined lists (so-called "sets"). They cannot average at every cell, and cannot average around several moving Lagrangian objects (even if it could be possible to program the required utility). Also, OpenFOAM® does not feature a divergent algorithm to compute averages and variances. In general, other filtering utilities are based on the convergent algorithm, or on the improved convergent algorithm we describe in Section 5.

4. CPPPO statistics routines

CPPPO features a collection of sampling routines which allow to relate fields (named *sampled fields* in CPPPO) with other fields (named *markers* in CPPPO). The sampling utility will draw samples of the specified quantities of interest (according to the functions described in Section 4.1) at user-defined locations, or alternatively over the whole domain. Every sample will contain values of *sampled fields* and *markers*. *Sampled fields* are then binned according to the related *markers* following user-defined settings for discretization of the binning process. Every time a value is added to a bin, CPPPO will automatically update the mean value and variance related to that bin using a running statistics approach [33]. Therefore, CPPPO will also keep track of the number of values added to every bin. Following this procedure, a large dataset is reduced to a multidimensional array, in which each element contains a (conditional) average and variance with respect to the

Please cite this article in press as: F. Municchi, et al., Highly efficient spatial data filtering in parallel using the opensource library CPPPO, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.05.026

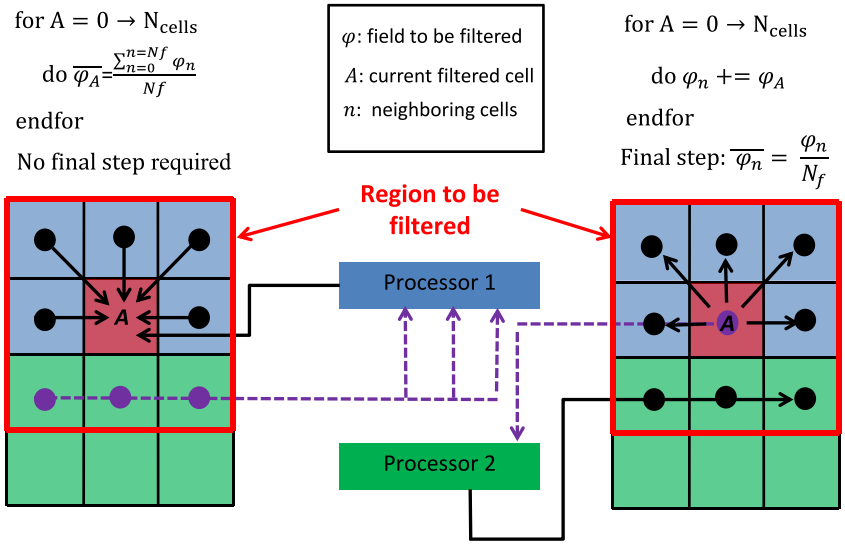

Fig. 3. Convergent approach (left) and divergent approach (right) for filtering. Continuous arrows represent intra-processor operations while dashed arrows indicate data exchange between processors. Processor domains are identified with the owner color, red cells represent the current cell to be filtered. Dots represent cell centers involved in local data operations (black) and parallel data operations (purple). In the picture N_{cells} is the total number of cells per processor (for simplicity we consider the same number N_{cells} in each processor) and N_f is the number of cells within the region to be filtered. In the divergent approach, field values at cell A are spread to the neighboring cells while the opposite occurs in the convergent algorithm. The divergent algorithm also requires the communication of less data. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

markers of the *sampled fields*. This multidimensional array is then written to disk in the form of one dimensional arrays. CPPPO allows to create new files (and thus, new statistics) at every time step, or to update the current files and statistics in order to collect a single (time-averaged) data set. This *sampling/binning* procedure has been developed to perform automatic correlation of quantities of interest during, or after a simulation run. In such a way, a user can quickly assess whether a simulation needs to be run longer, or can be aborted to save computational resources.

4.1. Available sampling operations

At the current state, the available sampling routines are:

- *General sampling*: This routine draws samples over the whole domain, or just a portion of it. *sampled fields* and *markers* are defined by the user. Also, *General sampling* allows the use of a formula parser implemented in CPPPO to draw samples of quantities which are not explicitly calculated in the simulator.
- *Angle vector–vector*: This routine can sample vector fields using the angle between the original and a second vector field as *marker*.
- *Two point correlation*: This routine will sample the value of the trace of the two point velocity correlation.

5. Parallel implementation

CPPPO has been designed to (i) maximize the speed of data averaging calculations, and (ii) to provide a flexible architecture for the future addition of new models and algorithms. For this reason, a separation between cell selectors and filters was needed. Despite performance could possibly be affected by this approach in a negative way, the philosophy behind CPPPO is to allow the user to code new filters without implementing a new cell selector. As we will discuss in the following, however, new algorithms have been developed in order to increase performance and minimize the number of parallel communications and the amount of communicated data.

A CPPPO run is initialized via the interface class which allocates memory for filtered fields. For every user-defined filter operation, the interface class can trigger an FSB loop. All these operations are encapsulated in the CPPPO core library. Note that filtered fields, while always available in the interface class, are mapped in the core library only once (and not for each filter, or filter size). Thus, fields created for a certain filter are not available when running CPPPO for another filter. This requires the interface and the core library to run at two different levels: while the interface class has pointers to quantities used over the whole run, CPPPO's core library has pointers to relevant quantities only for the current filter (with the exception of mesh data and source fields data). This allows an easier and more intuitive use of pointers in the core library when accessing filtered field data.

Parallel communication in CPPPO mainly relies on collective MPI operations, since most of the time all processors have to synchronize during the calculations. These MPI routines have shown excellent performance in many applications [34,35] on HPC hardware. The load partitioning is mainly a function of the domain decomposition, and the distribution of sampling locations, so that it is mostly user dependent. This is particularly true when using Lagrangian filtering operations (i.e., filtering is performed at pre-defined probing positions). This is because the user affects directly the processor load in such a situation. For example, in case all probes are positioned in a sub-domain belonging to the same processor, the calculation would be slow. Thus, all operations would be focused on just one processor.

In the following we will discuss the implementation of parallel selectors and filters in CPPPO.

5.1. Parallel selectors

At the current state CPPPO features two parallel cell selectors: general unstructured and IJK structured. The former can deal with any unstructured mesh, while the latter is designed for structured meshes whose cells are equal of size. Both selectors have a similar structure that can be summarized as follows:

(i) Evaluate the position of the current cell (or probe location) to filter.
(ii) Communicate this position to all the other processors.

(iii) For every position, calculate the filter size (according to the boundary conditions) and create a list of cells that reside within the filter. For every cell added, update the total filter volume (this volume calculation allows to deal with complex cell shapes).
(iv) Communicate the filter volume to all the other processors (optional, since the total volume is generally not necessary).

This algorithm does not calculate the complete cell list for a single cell or probe location, but every processor calculates the cell lists corresponding to the fraction of every filter residing within its boundary. The above algorithm has been used to optimize the run time of divergent and convergent filtering operations. A workflow which illustrates the main steps in the selecting operation is shown in Fig. 4.

CPPPO selectors also take periodic boundary conditions into account. In addition, and in case the processor sub domain is entirely inside the filter, all its cells are automatically added to a list with no further operations. Both parallel selectors that are currently implemented require one collective MPI operation, during which every processor communicates the coordinates of its currently filtered cell (i.e., in total $3n_p$ doubles where n_p is the number of processors).

The structured IJK selector takes advantage of the possibility to define a coordinate system using the grid axis. Thus, a one-to-one correspondence between the cell id and a location in the Cartesian reference frame can be obtained. In order to do that, we express the new cell center coordinates ζ_i in the form:

$$\zeta_i = \frac{c_i - \delta_i/2}{\delta_i} \tag{11}$$

where c_i is the non-IJK cell center coordinate and δ_i is the cell size in the ith spatial direction. This coordinate transformation allows us to immediately evaluate the cells inside a region and their id, consequently speeding up the calculation.

In contrast, the more general unstructured selector loops over all the cells in the processor subdomain and checks, for every cell, if its center lies within the filter region. Despite the fact that this algorithm is expensive in terms of computational time, it can deal with arbitrarily-shaped computational meshes. The latter are often used in engineering applications, and are also considered in the showcase detailed in Section 8.

5.2. Parallel filters

At the current state CPPPO features a top-Hat kernel filtering operation that can be run in Eulerian or Lagrangian mode depending if the filtering has to occur for every cell or at specific Lagrangian points. Since filtering operations are repeated for each cell/probe (see Fig. 4), in the following we will consider the parallel communications required to filter at just one location (Lagrangian mode) or one cell (Eulerian mode).

The Eulerian mode uses the divergent algorithm to update the filtered fields, and can be summarized as follows:

(i) Calculate weighted fields for the cell at the current step. Weights are defined by the user.
(ii) Communicate the values of weighted fields to the neighboring processors.
(iii) Update the filtered field.

Using this algorithm, just one `MPI_Allgather` operation is needed, and every processor exchanges a number of values equal to the total number of fields to filter. Clearly, in case vector fields are filtered, each spatial component has to be considered as a separate field when calculating the size of communicated data.

The Lagrangian mode uses an improved convergent algorithm, which can be summarized as follows:

(i) Calculate the locally-filtered value for the selected cell.
(ii) Communicate these filtered values, and calculate the final filtered value accounting for the locally-filtered values from all neighboring processors.

In the improved convergent algorithm, every processor instead of communicating the whole list of cell values, performs a local filtering calculation (i.e., performs the averaging using only the cells it owns). Thus, only locally-filtered values need to be communicated (see left panel in Fig. 5). This greatly reduces the number of communicated data. However, still this algorithm is less efficient than the divergent algorithm (see right panel in Fig. 5). In fact, the divergent algorithm requires the communication of just $n_f + 1$ values per processor (where n_f is the number of fields to filter and the additional one is the weight). In contrast, the improved convergent algorithm, requires the communication of $n_p(n_f + 1)$ values per processor (where n_p is the number of processors). This is due to the direction of the data flow which, in the convergent algorithm, points from the neighboring processors to the *central* one as shown in Fig. 3, and not *vice versa*. In this context, the term *central* refers to the processor owning the cell to be filtered. However, since the algorithm is running in parallel, n_p cells are filtered at the same time and thus, every processor represents the *central* processor with respect to the cell it owns. This means that the convergent approach requires the communication of, at least, $2n_p$ values so that the parallel efficiency will inevitably decrease with increasing number of cores. The variance calculation is another weak point of the improved convergent approach (even over the classic convergent algorithm). In fact, since the variance calculation requires the information on the filtered value (and not the partially-filtered value), additional communication is required to make filtered values available to every processor. In principle, the variance calculation follows an approach similar to the averaging step (i.e., partial variances are computed). This results in $n_p(2n_f + 1)$ data to be communicated (this calculation includes communication of the locally-computed variances). Clearly, the original convergent algorithm does not have this issue, since all the required values become available at the *central* processor after the first (and only) communication step. However, for the improved convergent algorithm, the number of communicated data scales linearly with n_p, while for the original convergent algorithm, this quantity is difficult to evaluate since each processor would need to communicate a different number of elements. That would require the use of less efficient collective operations like `MPI_Allgatherv`. Anyhow, the number of cells in the mesh is typically several orders of magnitude larger than n_p. Also, the filter size is, for the majority of applications, of the order of 10^{-1} times the domain length. Hence, we can conclude that, for almost any application, the number of exchanged data in the original convergent approach is much larger than in the improved convergent approach.

In contrast to the improved convergent algorithm, the divergent algorithm only requires the communication of $n_f + 1$ doubles, regardless of the fact whether variance calculation is performed or not. In terms of MPI collective operations, both algorithms require one operation per filtered cell for averaging. In case the variance is also computed, the convergent algorithm requires two additional MPI operations per cell.

6. Test calculations

The accuracy of CPPPO was tested by considering two well-known problems of fluid dynamics: Stokes flow and irrotational (i.e., potential) flow around a sphere. The main objective of these tests is to evaluate the accuracy of the parallel filter routines, and to illustrate the dependency of the results on the grid size. Therefore,

Please cite this article in press as: F. Municchi, et al., Highly efficient spatial data filtering in parallel using the opensource library CPPPO, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.05.026

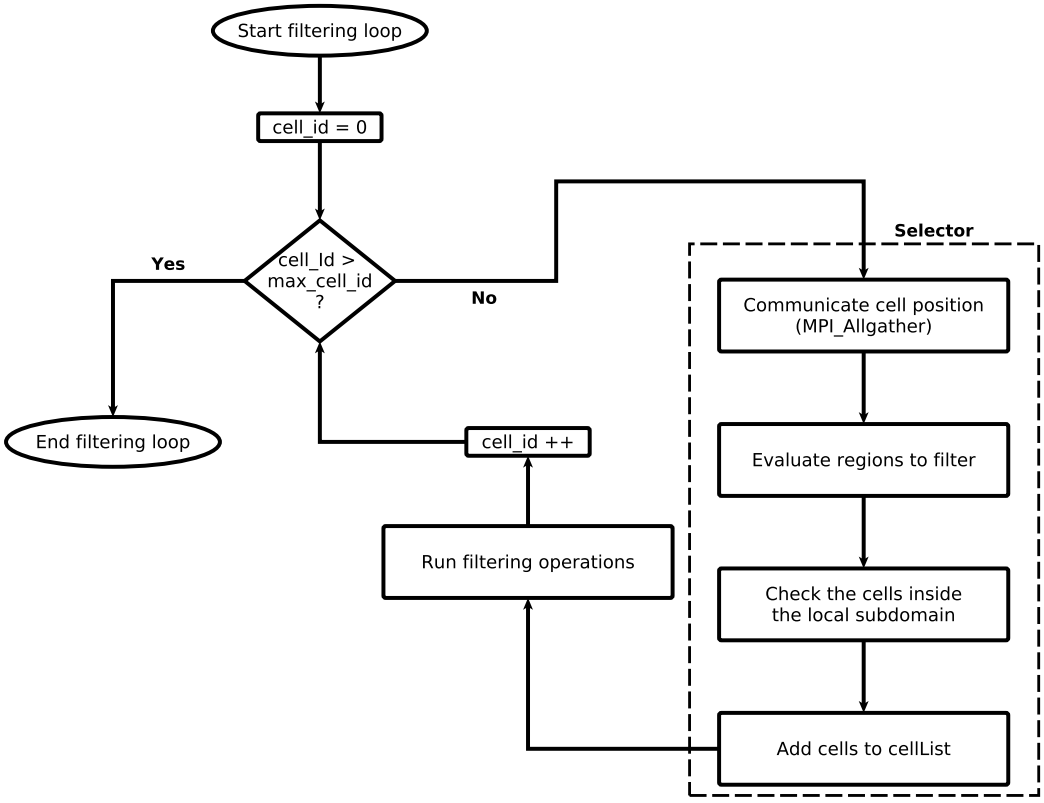

Fig. 4. Workflow illustrating the selector algorithm within the filtering loop. `max_cell_id` is the total number of cells on the processor. In case the filtering is carried over a set of probes, `max_cell_id` represents the number of probes on the processor.

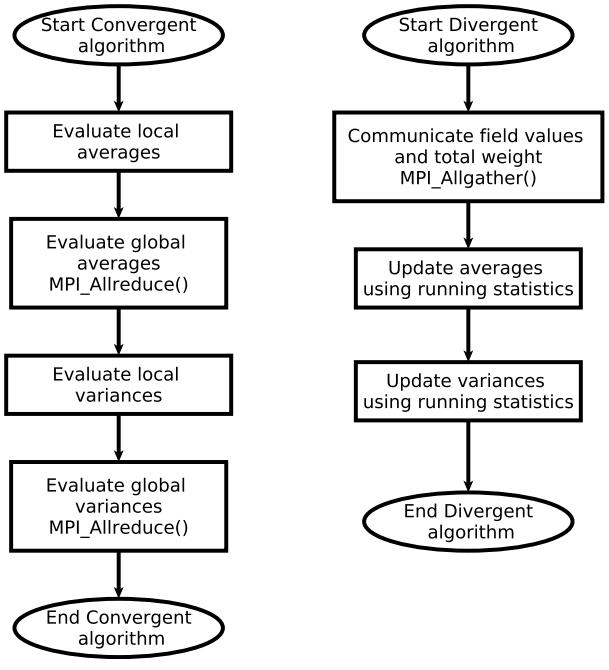

Fig. 5. Workflow for the convergent (left panel) and the newly proposed divergent algorithm (right panel). The convergent algorithm requires an additional `MPI_Allreduce` operation to calculate the variance. In addition, every `MPI_Allreduce` requires the exchange of more data than the `MPI_Allgather`.

we compared CPPPO results with analytical solutions of filtered quantities at the particle center. Recalling the analytical solution for Stokes flow (i.e., zero Reynolds number) around a sphere [36], and when considering the velocity component in the stream-wise (i.e., x-) direction, the flow field is described by:

$$u_x = U_\infty \left[\cos^2\theta \left(1 + \frac{R^3}{2r^3} - \frac{3R}{2r} \right) + \sin^2\theta \left(1 - \frac{R^3}{4r^3} - \frac{3R}{4r} \right) \right]. \tag{12}$$

Here θ and r describe the polar and radial positions in a spherical coordinate system (the solution is symmetric with respect to the azimuthal coordinate). U_∞ is the flow velocity far from the particle, and R is the particle radius. The corresponding solution for irrotational flow (i.e., a flow characterized by an infinitely large Reynolds number) past a sphere is:

$$u_x = U_\infty \left[\cos^2\theta \left(1 - \frac{R^3}{r^3} \right) + \sin^2\theta \left(1 + \frac{R^3}{2r^3} \right) \right]. \tag{13}$$

We now consider a spherical filter, and define a dimensionless filter size as:

$$\rho = \frac{R_f}{R} \tag{14}$$

where R_f is the filter radius. Integration of Eqs. (12) and (13) to obtain the mean and variance of the stream-wise velocity component leads to:

$$\widetilde{u}_x|_{Stokes} = \frac{\left(2\,\rho^2 - \rho - 1\right)\,U_\infty}{2\,\left(\rho^2 + \rho + 1\right)} \tag{15}$$

$$\tau_{xx}^{sgs}|_{Stokes} = \frac{\left(18\,\rho^5 - 32\rho^4 + 14\,\rho^3 - 3\rho^2 + 2\,\rho + 1\right)\,U_\infty^2}{\left(4\,\rho^4 + 8\,\rho^3 + 12\,\rho^2 + 8\,\rho + 4\right)\,5\,\rho^3} \tag{16}$$

$$\widetilde{u}_x|_{Irr} = U_\infty \tag{17}$$

$$\tau_{xx}^{sgs}|_{Irr} = \frac{U_\infty^2}{5\rho^3}. \tag{18}$$

Please cite this article in press as: F. Municchi, et al., Highly efficient spatial data filtering in parallel using the opensource library CPPPO, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.05.026

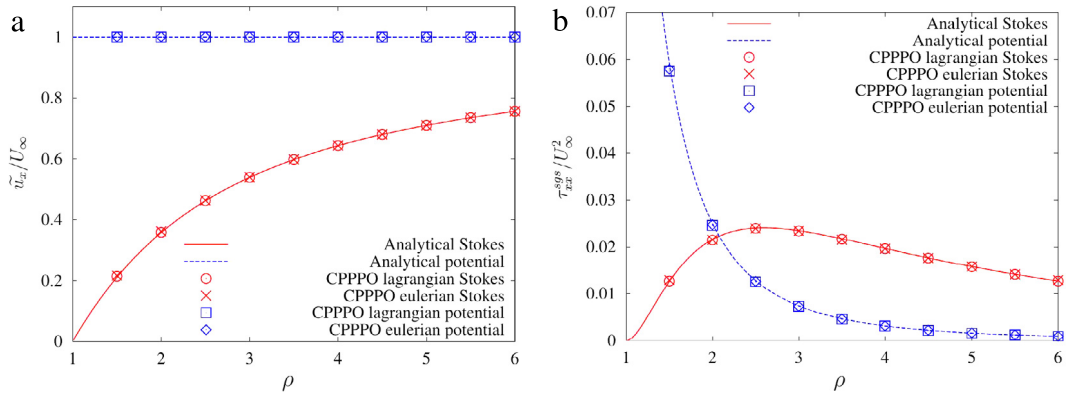

Fig. 6. Comparison between CPPPO's Lagrangian and Eulerian filtering tools with analytical results for the filtered quantities at the particle center. Results for the Favre average are shown in panel (a), while in panel (b) the Favre variance was calculated. The average velocity is normalized with the 'far field' velocity U_∞, while the variance with U_∞^2.

Table 1
Computed relative error for Stokes and potential flow test cases.

Case	Operation	Average relative error
Stokes Eulerian	Average	0.004877
Stokes Lagrangian	Average	0.002028
Potential Eulerian	Average	Below machine precision
Potential Lagrangian	Average	Below machine precision
Stokes Eulerian	Variance	0.006865
Stokes Lagrangian	Variance	0.003445
Potential Eulerian	Variance	0.011906
Potential Lagrangian	Variance	0.017841

The solutions in Eqs. (15)–(18) provide a set of cases to verify the filtering routines of CPPPO for the situation of a top-Hat filter kernel in spherical coordinates. However, Stokes and potential flows are not easily reproduced with standard CFD solvers unless the convective (or the viscous) term is removed from the equation. Even the use of specific solvers like *potentialFOAM* could induce some errors due to the discrete representation of the particle by an Eulerian mesh, or the finite size of the bounding walls. As a consequence, we preferred to impose the flow field rather than solving the governing equations with a CFD solver. The two newly implemented applications that impose the Stokes and the irrotational flow field are *stokesFilter* and *irrotationalFilter*, respectively. Two computational grids of $100 \times 100 \times 100$ and $160 \times 160 \times 160$ cells were used to evaluate the flow field. These resolutions resulted in a negligible effect of the mesh resolution on both the results for the mean and the variance. Test cases were run using 128 processes in order to assess the accuracy and speed of the parallel computation.

Results displayed in Fig. 6 show that CPPPO is able to correctly calculate the Favre average of a field both with Lagrangian and Eulerian filtering routines. Table 1 shows that the average relative error remains smaller than 1.7%, and that the variance may experience larger errors compared to the average.

The deviations between Lagrangian and Eulerian results can be explained considering that the two algorithms perform different algebraical operations and, thus, are subjected to different round-off errors.

In order to assess the runtime filtering routines, a low Reynolds number flow past a sphere was simulated using a computational grid of $120 \times 120 \times 120$ cells, and a computational domain of $10d_p \times 10d_p \times 10d_p$. CPPPO was linked to OpenFOAM®'s *pisoFoam* solver, and the simulation was run until a steady-state was obtained.

The results show that there is a difference in the values of the Favre averaged velocity field calculated by CPPPO with respect to the analytical results. This discrepancy is due to the incorrect velocity field computed by the solver as shown in Fig. 7. Specifically, this is caused by the finite domain size, as can be seen by the stronger deviations with increasing distance from the particle surface.

7. Parallel scalability and performance

In this section, we analyze the parallel scalability and performance of CPPPO. In particular, we compare the performance of the divergent and convergent algorithm, as well as the performance of the unstructured and IJK cell selector. Different metrics were used in order to quantitatively establish the performance of every algorithm, and to assess their preferred field of use.

Since CPPPO makes extensive use of MPI collective operations, individual processes are forced to synchronize often. This will result in acceptable parallel performance in case the load balance is uniform. However, the total time each processor takes to complete a certain task is subject to some fluctuations that are different for every run. For this reason, the average time τ_p (where p is the number of processes) and the time variance σ_p are used as main performance metrics in the following. Specifically, these metrics are defined as:

$$\tau_p = \frac{\sum_{n=1}^{p} t_{p,n}}{p} \tag{19}$$

$$\sigma_p = \sqrt{\frac{\sum_{n=1}^{p} \left(t_{p,n} - \tau_p\right)^2}{p}} \tag{20}$$

where $t_{p,n}$ is the time needed by process n to complete a certain task in case a total of p processes are used for the computation. In the following we refer to $\tau_p|_k$ as the average time taken by subroutine k when p processors are used. It should be noted that the wall time is $\max(t_{p,n})$. Furthermore, the standard deviation was, in general, observed to be small compared to the average time due to the frequent MPI barriers used for synchronization. Thus, σ_p is not discussed in further detail below.

The strong parallel efficiency is defined as:

$$\eta_s = \frac{\tau_1}{p\tau_p}. \tag{21}$$

In this study, we also define an advantage factor α_n^k to quantify the advantage, in terms of computational time, of using the subroutine n instead of the subroutine k. The advantage factor is then defined as:

$$\alpha_n^k = \frac{\tau_p|_k}{\tau_p|_n}. \tag{22}$$

Please cite this article in press as: F. Municchi, et al., Highly efficient spatial data filtering in parallel using the opensource library CPPPO, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.05.026

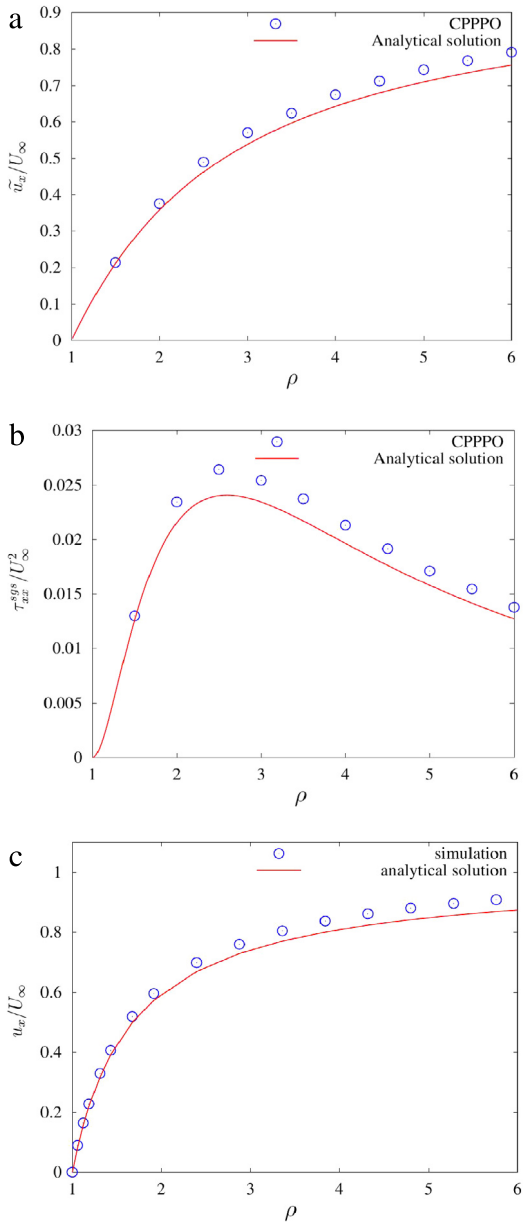

Fig. 7. Normalized Favre average (panel (a)) and variance (panel (b)) of the velocity field based on a simulation using OpenFOAM®'s *pisoFoam* solver. Fig. 7(c) shows the calculated velocity profile along the span-wise direction, i.e., $\theta = \pi$, and the corresponding analytical solution.

Table 2
Test cases for the parallel scalability analysis.

Filtering algorithm	Selector	Mesh size (cells)
Divergent	Unstructured	1×10^6
Divergent	Unstructured	2×10^6
Divergent	Unstructured	4×10^6
Divergent	IJK	1×10^6
Divergent	IJK	2×10^6
Divergent	IJK	4×10^6
Convergent	IJK	1×10^6
Convergent	IJK	2×10^6
Convergent	IJK	4×10^6

In particular, we will evaluate the advantage factor of the divergent filtering over the convergent filtering (α_d^c) and the advantage factor of the IJK selector over the unstructured selector (α_i^u).

In order to assess the parallel performance of the implemented algorithms, we run the *stokesFilter* test case introduced in Section 6 in order to evaluate the Favre averaged velocity field for every cell in the domain using a box filter. The filter size was approximately one quarter of the domain length in every direction. Nine test cases were run in total, using different routines and mesh size as reported in Table 2.

Fig. 8 shows average time and strong efficiency from the studied cases. It can be seen that the unstructured selector requires significantly more time compared to the IJK selector. However, the unstructured selector shows a far better parallel efficiency. Thus, the IJK selector shows significant advantages with respect to the unstructured selector in case the number of cores is small (e.g., when using a local workstation).

This fact is well represented by the advantage factors displayed in Fig. 9. When 128 cores are used, the IJK selector provides no more significant advantages in terms of computational time and α_i^u drops below unity. Fig. 9(a) shows that α_c^d is very close to unity when one processor is used, but increases rapidly with the number of cores. However, the effect on the total time is only moderate (see Fig. 8), since filtering operations are generally faster than selector operations. In summary, a divergent filtering approach and (surprisingly) an unstructured selector seem to be the optimal combination for a large number of cores.

Finally, the Vienna Scientific Cluster VSC-3 was used to test the library up to 1024 cores. The results generally showed a higher performance of VSC-3 with respect to TU Graz' dcluster (see Fig. 10). However, the parallel scalability was very similar to dcluster, showing the expected drop in performance for the smaller case involving 10^6 grid cells and when using less than approximately $4\,10^3$ grid cells per core. In summary, our benchmark calculations on VSC-3 confirmed the previously described excellent scalability of CPPPO.

8. Heat transfer in a dense particle bed

8.1. Transport in dense particle beds

Particle-resolved direct numerical simulations (PR-DNS) of flow through dense particle beds have become key instruments to develop closures for predicting momentum, heat and mass transfer rates in these systems [15,37]. Typically, these simulations require extremely large computational grids (with $O(10^7)$ cells) to resolve regions with large velocity, concentration, or temperature gradients. Furthermore, a large number of realizations (e.g., particle configurations in a channel) are needed to represent reality reasonably well. This naturally leads to large data sets, asking for on-the-fly data filtering and an automation of the post-processing workflow. In the following, we will show that such a workflow can be carried out efficiently and in a fully-automated fashion using CPPPO. Specifically, we study a situation similar to the one considered by [38,39], and limit our attention to the prediction of flow and a single inert scalar.

8.2. Governing equations and numerical solution

The open source library OpenFOAM® is used in order to solve the incompressible momentum transport equations, the continuity equation and the transport equation of an inert scalar. These equations can be re-written in their dimensionless form using the Einstein notation to arrive at:

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{23a}$$

$$\frac{\partial u_i}{\partial t} + \frac{\partial \left(u_j u_i\right)}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{1}{Re_p}\frac{\partial^2 u_i}{\partial x_j \partial x_j} \tag{23b}$$

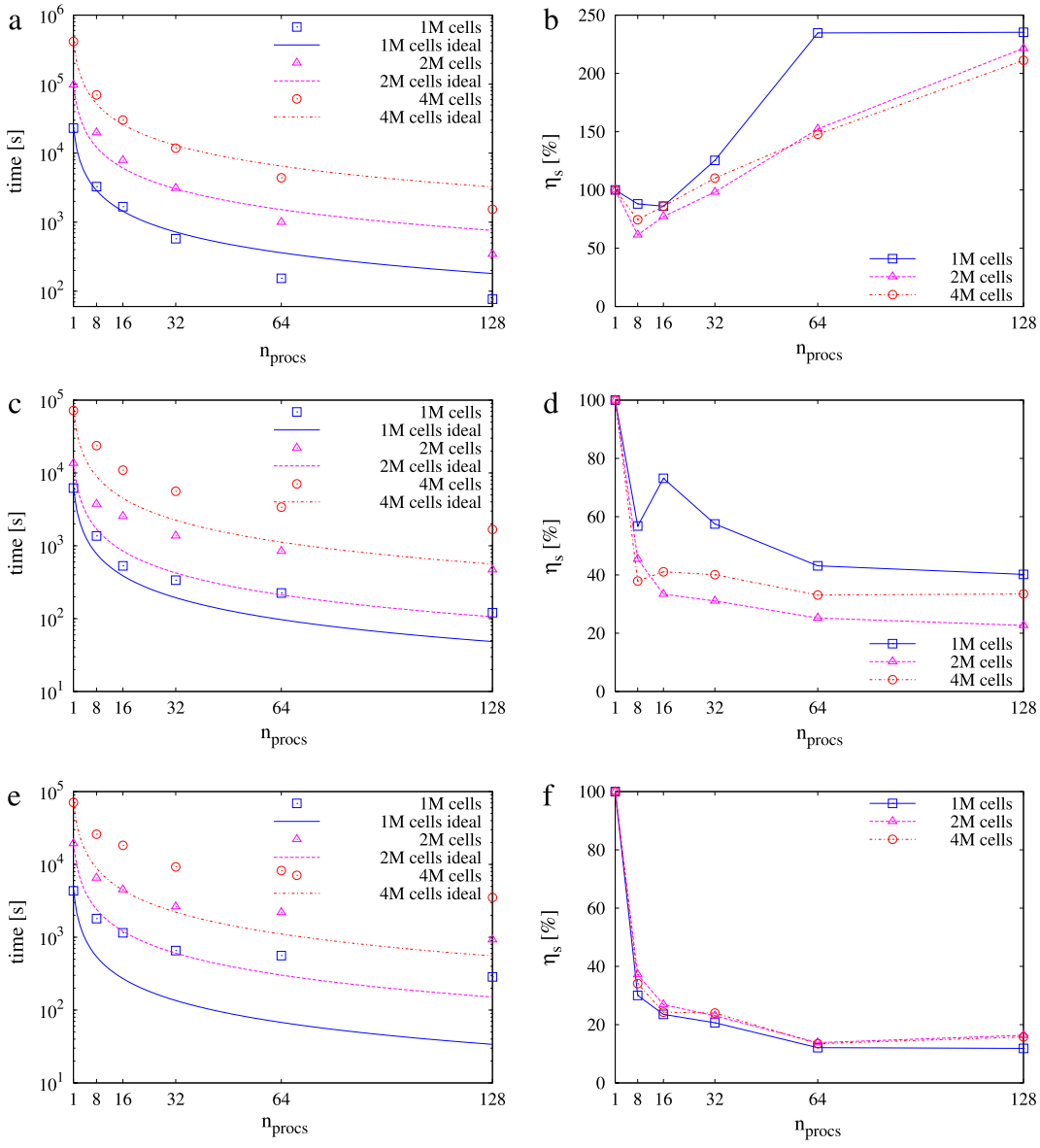

Fig. 8. Average computation time and strong parallel efficiency for the divergent filtering approach with unstructured selector (panels (a) and (b)), as well as the IJK selector (panels (c) and (d)). Panels (e) and (f) show average time and strong parallel efficiency for the convergent filtering approach with the IJK selector.

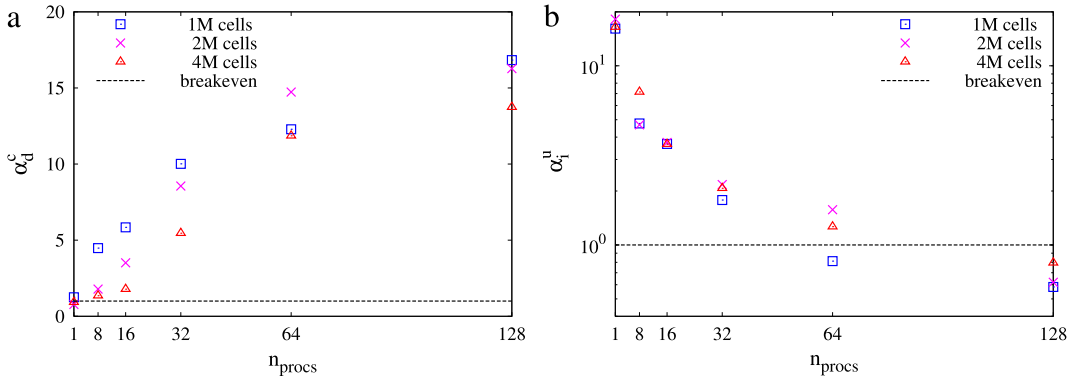

Fig. 9. Advantage factor of the divergent filtering approach over the convergent filtering approach (panel (a)), as well as advantage factor of the IJK selector over the unstructured selector (panel (b)).

Please cite this article in press as: F. Municchi, et al., Highly efficient spatial data filtering in parallel using the opensource library CPPPO, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.05.026

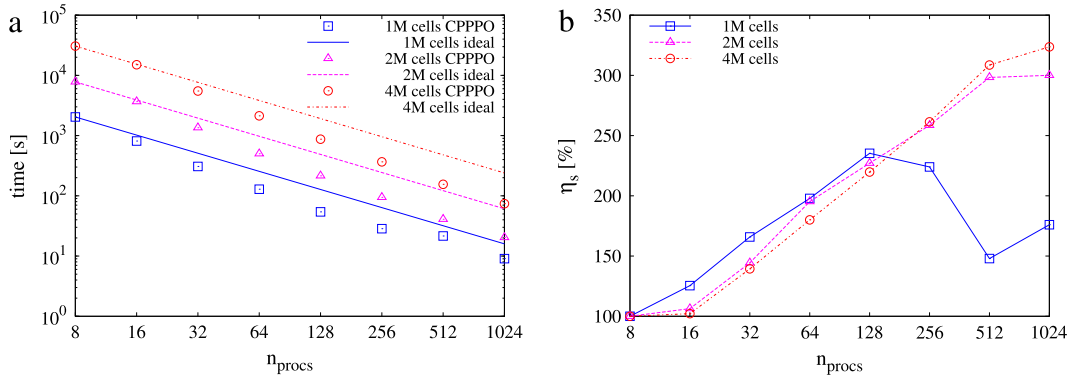

Fig. 10. Average time for the divergent filtering algorithm utilizing an unstructured grid selector on the VSC-3 cluster (panel (a)), as well as strong parallel efficiency (b).

$$\frac{\partial \phi}{\partial t} + \frac{\partial \left(u_i \phi\right)}{\partial x_i} = \frac{1}{Pe} \frac{\partial^2 \phi}{\partial x_i \partial x_i} \tag{23c}$$

where u is the velocity field, p is the pressure and ϕ is the scalar field. The scalar transport equation can model heat or mass transfer without additional source terms, e.g., due to chemical reactions. The relevant dimensionless group is therefore represented by the Peclet number Pe and the particle Reynolds number Re_p:

$$Re_p = \frac{U d_p}{\nu} \tag{24a}$$

$$Pe = \frac{U d_p}{\Gamma} \tag{24b}$$

where U is a typical flow speed (i.e., the superficial fluid velocity), d_p is the particle diameter, ν is the fluid kinematic viscosity, and Γ is the scalar's diffusion coefficient. For the present simulation we choose $Re_p = 10$ and $Pe = 20$. The equations were discretized using second order discretization schemes, and the PISO algorithm was adopted to solve the pressure equation.

The computational domain is a cylinder of radius $3d_p$ and a length of $16d_p$. A fixed particle bed having a void fraction of approximately 0.2 (involving 130 particles) was generated using the soft-sphere particle motion simulator LIGGGHTS® [40]. A body-fitted unstructured mesh was generated using the snappyHexMesh tool available in OpenFOAM®. Since constant velocity and zero gradient boundary conditions were used at the inlet and outlet surfaces, the particle bed was positioned between $x = 2d_p$ and $x = 14d_p$ in the cylinder's axial (i.e., x-) direction to reduce the effect of the above mentioned boundary conditions on the results. A no-slip boundary condition for the velocity field was imposed at the cylinder's and at the particles' surface. For the scalar field we imposed a zero gradient boundary condition at the cylinder wall, and a fixed value boundary condition at the particles' surface.

The final mesh consisted of approximately 7 million cells and featured a local grid refinement at the particles surface. The simulation was run until a steady-state solution was obtained.

8.3. Results and CPPPO post-processing

The case was run and post-processed using the Vienna Scientific Cluster (VSC-3). The computational domain was decomposed using 128 cores, and the execution of CPPPO's routines required approximately 1.5% of the total calculation time (i.e., 7.5 min out of 8.55 h). CPPPO applied four box filters of different lengths d_f to all the cells and the particles in the domain. In addition, the CPPPO *general sampling* utility was used to evaluate the probability distribution function of the filtered velocity field in the region between $x = 4.5d_p$ and $x = 12.5d_p$.

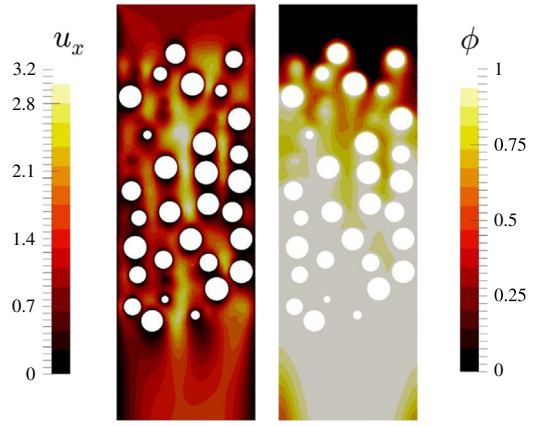

Fig. 11. Flow through a particle bed in a cylindrical channel: (unfiltered) velocity field in the axial direction (left panel), as well as scalar field (right panel; $\phi_p = 0.20$).

The resulting unfiltered fields are shown in Fig. 11. For every particle, a filter-size dependent bulk scalar field was defined:

$$\phi_b(d_f) = \frac{\int_{V_f} u_x \phi dV}{\int_{V_f} u_x dV} \tag{25}$$

where the filter volume V_f spans the region between the particle's surface and the filter radius. These quantities are calculated by CPPPO, and subsequently used to calculate a particle-based Sherwood number:

$$Sh_p = Pe \frac{q_p^{s,f}}{1 - \phi_b}. \tag{26}$$

Here $q_p^{s,f}$ is the dimensionless solid–fluid scalar flux for particle p, which is defined as:

$$q_p^{s,f} = \frac{1}{S_p\, Pe} \int_{S_p} \frac{\partial \phi}{\partial n} dS. \tag{27}$$

Values of Sh_p as a function of the axial position are shown in Fig. 12(a). The figure only displays values of $Sh_p < 20$, since some particles experience extremely small differences of the scalar quantity, i.e., $1 - \phi_b$, and hence would result in unrealistically large Sh_p values. However, Table 3 shows that most of the particles have a particle-based Sherwood number that is smaller than 20. Note, that the use of data from multiple realizations could reduce the standard deviation and result in a constant Sh_p number along the axis of the cylinder as has been already shown in literature [15,37]. In addition, the data shown in Table 3 reveals that increasing the filter size leads to a reduction in the data deviation.

Please cite this article in press as: F. Municchi, et al., Highly efficient spatial data filtering in parallel using the opensource library CPPPO, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.05.026

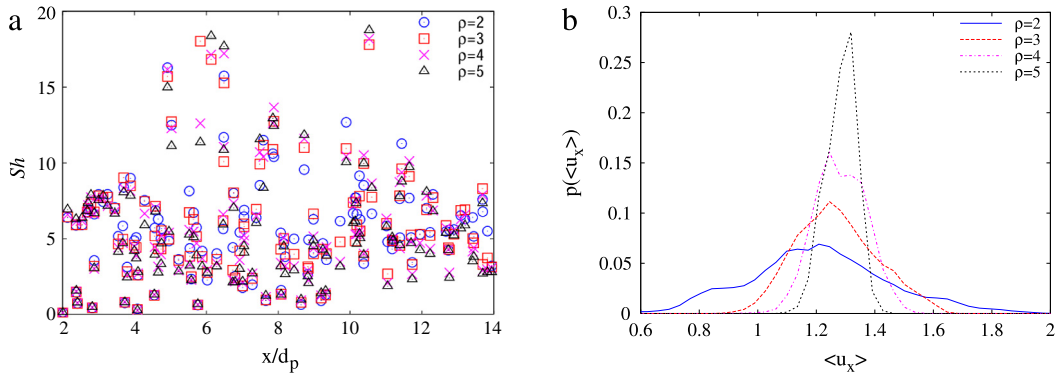

Fig. 12. Particle Sherwood number experienced by a dense particle ensemble in a cylindrical channel as a function of the filter size and the axial position (left panel). Probability distribution function of the filtered axial velocity experienced by the particles (right panel).

Table 3
Sh_p statistics for different filter parameters in the region $4.5 < x/d_p < 12.5$ (86 particles) and $\phi_b < 0.99$.

$\rho = d_f/d_p$	Average Sh_p	Standard deviation Sh_p	$\phi_b < 0.99$	$Sh_p < 20$
2	14.380	41.017	98.5%	87.3%
3	13.287	35.223	100%	86.0%
4	12.352	27.937	100%	86.0%
5	11.771	25.194	100%	86.0%

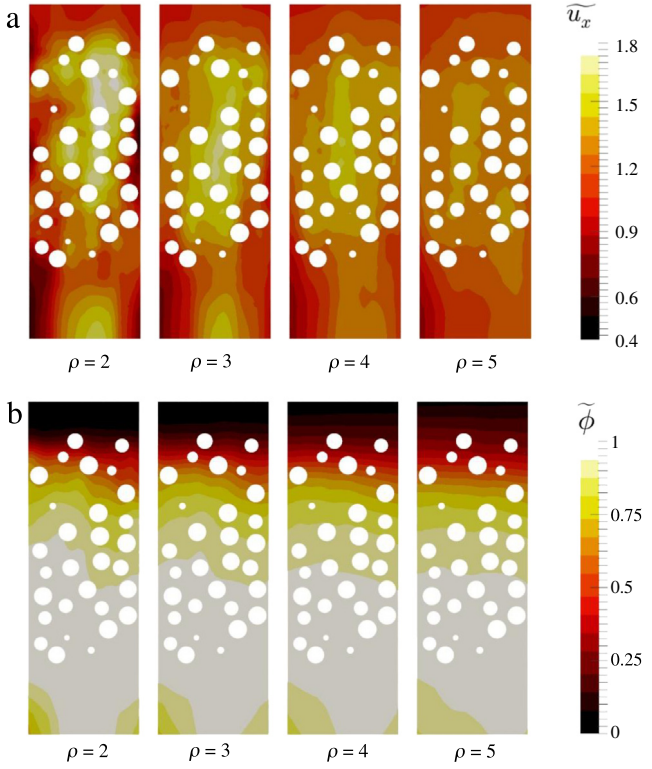

Fig. 13. Filtered velocity field in the axial direction and filtered scalar field.

Results obtained via CPPPO's *general sampling* module show that the larger the filter is, the more uniform the filtered velocity will be (see Fig. 12(b)).

Finally, we show the complete filtered velocity and scalar field in Fig. 13. These fields have been written by the CPPPO-OpenFOAM® interface and are automatically generated. Interestingly, while the filtered velocity field tends to become more uniform by increasing the filter size, the filtered scalar field maintains its dependence on the axial coordinate even for large filters. This is due to the fact that the scalar field is not statistically homogeneous in the axial direction.

9. Summary and conclusions

The aim of the CPPPO library is to provide a set of routines for efficient parallel data filtering and processing. These operations are meant to be performed "on the fly" during expensive numerical simulations running on large distributed memory clusters. In order to perform data filtering from parallel simulations on clusters, a novel approach to filtering named "*divergent*" was adopted. The divergent approach showed a linear increase of parallel efficiency with the number of cores, and a major reduction of computational time with respect to the standard convergent approach was demonstrated. Overall, the parallel scalability analysis of CPPPO showed promising results, demonstrating the computational efficiency of our library. Furthermore, the CPU time required by CPPPO was shown to be a small fraction (i.e., less than 2%) of the time required by a typical simulation in the field of dense fluid–particle systems. As recently shown in literature [17], more insight into the governing flow physics of dense particle beds can be gained from the analysis of individual-particle DNS data. We have demonstrated that the filter size should be considered when evaluating such individual-particle data, e.g., (average) fluid quantities experienced by the particles. In addition, the ability to perform variance calculations in CPPPO allows one to extract additional markers that can be helpful to correlate DNS data, and hence establish new closure models. What remains to be done is to develop relevant transport equations for predicting these markers in coarse-grained simulations. Then, we expect that a new generation of closure models, established with the help of tools like CPPPO, will help to refine our predictions for relevant fluid–particle systems in engineering simulations.

CPPPO allows a high flexibility in the filtering operations due to the easy customization of the filtering kernel. This can be achieved either by (i) including an arbitrary number of weights (which can be defined at runtime), or (ii) by implementing the desired kernel function (which requires some coding in C++, and recompilation of CPPPO).

CPPPO comes with instructions for compilation, as well as documentation covering input and usage of every module and sub-module. CPPPO also comes with examples on how to be coupled to OpenFOAM® or CFDEM® applications. A freely available version of the code can be downloaded from the CPC program library.

Please cite this article in press as: F. Municchi, et al., Highly efficient spatial data filtering in parallel using the opensource library CPPPO, Computer Physics Communications (2016), http://dx.doi.org/10.1016/j.cpc.2016.05.026

To download the up-to-date version of CPPPO and get additional documentation, the interested reader is referred to http://www.tugraz.at/en/institute/ippt/downloads-software/.

Acknowledgments

The authors acknowledge support by the European Commission through FP7 Grant agreement 604656 (NanoSim), and the NAWI Graz project by providing access to dcluster.tugraz.at. CFDEM® is a registered trademark of DCS Computing GmbH. The computational results presented have been achieved (in part) using the Vienna Scientific Cluster (VSC-3). OpenFOAM® is a registered trademark of OpenCFD.

References

[1] M. Van der Hoef, M. van Sint Annaland, N. Deen, J. Kuipers, Annu. Rev. Fluid Mech. 40 (2008) 47–70.
[2] M.A. van der Hoef, M. van Sint Annaland, J.A.M. Kuipers, China Particuol. 3 (2005) 69–77.
[3] Y. Igci, T.A.I. Arthur, S. Sundaresan, S. Pannala, T. O'Brien, AIChE J. 7 (2009) 405–410.
[4] D.L. Marchisio, R.O. Fox, Multiphase Reacting Flows: Modelling and Simulation, Springer, 2007.
[5] S. Schneiderbauer, S. Pirker, J. Comput. Multiph. Flows 6 (2014) 29–48. URL: http://multi-science.atypon.com/doi/10.1260/1757-482X.6.1.29.
[6] J. Li, W. Ge, W. Wang, N. Yang, X. Liu, L. Wang, X. He, X. Wang, J. Wang, M. Kwauk, From Multiscale Modeling to Meso-Science, Springer, 2013.
[7] M. Germano, Phys. Fluids 29 (1980) 1755–1757.
[8] L. Berselli, J. Math. Anal. Appl. 386 (2012) 149–170. URL: http://dx.doi.org/10.1016/j.jmaa.2011.07.044.
[9] N.G. Deen, S.H.L. Kriebitzsch, M.A. van der Hoef, J.A.M. Kuipers, Chem. Eng. Sci. 81 (2012) 329–344. URL: http://dx.doi.org/10.1016/j.ces.2012.06.055.
[10] N.G. Deen, E.A.J.F. Peters, J.T. Padding, J.A.M. Kuipers, Chem. Eng. Sci. 116 (2014) 710–724. URL: http://dx.doi.org/10.1016/j.ces.2014.05.039.
[11] N.G. Deen, J.A.M. Kuipers, Curr. Opin. Chem. Eng. 5 (2014) 84–89. URL: http://dx.doi.org/10.1016/j.coche.2014.05.005.
[12] Z.-G. Feng, S.G. Musong, Powder Technol. 262 (2014) 62–70. URL: http://linkinghub.elsevier.com/retrieve/pii/S0032591014003258.
[13] H. Tavassoli, S. Kriebitzsch, M.A. van der Hoef, E.A.J.F. Peters, J.A.M. Kuipers, Int. J. Multiph. Flow 57 (2013) 29–37. URL: http://linkinghub.elsevier.com/retrieve/pii/S0301932213001055.
[14] A.A. Zaidi, T. Tsuji, T. Tanaka, Advanced Powder Technol. 25 (2014) 1860–1871. URL: http://linkinghub.elsevier.com/retrieve/pii/S0921883114002015.
[15] S. Tenneti, B. Sun, R. Garg, S. Subramaniam, Int. J. Heat Mass Transfer 58 (2013) 471–479. URL: http://dx.doi.org/10.1016/j.ijheatmasstransfer.2012.11.006.
[16] S. Tenneti, S. Subramaniam, Annu. Rev. Fluid Mech. 46 (2014) 199–230. URL: http://www.annualreviews.org/doi/abs/10.1146/annurev-fluid-010313-141344.
[17] S.H.L. Kriebitzsch, M.A. Van der Hoef, J.A.M. Kuipers, AIChE J. 00 (2012) 1–9.
[18] J. Derksen, AIChE J. 60 (2014) 1202–12015.
[19] R. Jackson, The Dynamics of Fluidized Particles, in: Cambridge Monographs on Mechanics, 2000.
[20] J.J. Derksen, S. Sundaresan, J. Fluid Mech. 587 (2007) 303–336.
[21] H.W. Zhang, Q. Zhou, H.L. Xing, H. Muhlhaus, Powder Technol. 205 (2011) 172–183. URL: http://dx.doi.org/10.1016/j.powtec.2010.09.008.
[22] S. Radl, M. Girardi, S. Sundaresan, European Congress on Computational Methods in Applied Sciences and Engineering, ECCOMAS 2012, Vienna, Austria, 2012, pp. 1–15.
[23] S. Radl, S. Sundaresan, Chem. Eng. Sci. 117 (2014) 416–425. URL: http://dx.doi.org/10.1016/j.ces.2014.07.011.
[24] S. Sundaresan, S. Radl, C.C. Milioli, F.E. Milioli, The 14th International Conference on Fluidization—From Fundamentals to Products, 2013.
[25] S. Nakariyakul, J. Supercomput. 65 (2013) 262–273.
[26] C.-M. Tsai, Z.-M. Yeh, 2012 International Symposium on Computer, Consumer and Control, 2012, pp. 153–156. URL: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6228270.
[27] M.P.I. Forum, MPI: A Message-Passing Interface Standard, Version 3.0, High Performance Computing Center Stuttgart, 2012.
[28] P. Sagaut, Large Eddy Simulation for Incompressible Flows, Springer, 2006.
[29] G. De Stefano, O.V. Vasilyev, Phys. Fluids 14 (2002) 362–369.
[30] A. Favre, J. Appl. Mech. 32 (1965) 241–257.
[31] A. Favre, Studies in Turbulence, Springer-Verlag, 1992.
[32] B. Gschaider, swak4foam project, 2016. URL: https://openfoamwiki.net/index.php/Contrib/swak4Foam.
[33] B.P. Welford, Technometrics 4 (1962) 419–420.
[34] A.R. Mamidala, R. Kumar, D. De, D.K. Panda, Proceedings CCGRID 2008 - 8th IEEE International Symposium on Cluster Computing and the Grid, 2008, pp. 130–137.
[35] T. Ma, G. Bosilca, A. Bouteiller, J.J. Dongarra, J. Parallel Distrib. Comput. 73 (2013) 1000–1010. URL: http://dx.doi.org/10.1016/j.jpdc.2013.01.015.
[36] G. Batchelor, An Introduction to Fluid Dynamics, Cambridge University Press, 2000.
[37] H. Tavassoli, E.A.J.F. Peters, J.A.M. Kuipers, Chem. Eng. Sci. 129 (2015) 42–48. URL: http://linkinghub.elsevier.com/retrieve/pii/S000925091500130X.
[38] G.D. Wehinger, T. Eppinger, M. Kraume, Chem. Eng. Sci. 122 (2015) 197–209. URL: http://linkinghub.elsevier.com/retrieve/pii/S0009250914005016.
[39] M. Nijemeisland, A.G. Dixon, E.H. Stitt, Chem. Eng. Sci. 59 (2004) 5185–5191.
[40] C. Kloss, C. Goniva, A. Hager, S. Amberger, S. Pirker, Prog. Comput. Fluid Dyn. 12 (2012) 140–152.