OpenCL based parallel algorithm for RBF-PUM interpolation

Roberto Cavoretto · Teseo Schneider · Patrick Zulian

Abstract We present a parallel algorithm for multivariate Radial Basis Function Partition of Unity Method (RBF-PUM) interpolation. The concurrent nature of the RBF-PUM enables designing parallel algorithms for dealing with a large number of scattered data-points in high space dimensions. To efficiently exploit this concurrency, our algorithm makes use of shared-memory parallel processors through the OpenCL standard. This efficiency is achieved by a parallel space partitioning strategy with linear computational time complexity with respect to the input and evaluation points. The speed of our approach allows for computationally more intensive construction of the interpolant. In fact, the RBF-PUM can be coupled with a cross-validation technique that searches for optimal values of the shape parameters associated with each local RBF interpolant, thus reducing the global interpolation error. The numerical experiments support our claims by illustrating the interpolation errors and the running times of our algorithm.

Keywords mesh-free approximation · partition of unity methods · radial basis functions · scattered data interpolation · parallel algorithms · opencl.

Mathematics Subject Classification (2000) 65D05 · 65D15 · 65Y05 · 65Y20 · 68W10

1 Introduction

Mesh-free methods are known to be powerful computational tools for solving approximation problems, which include either multivariate data interpolation or numerical resolution of partial differential equations (PDEs) [16]. In recent literature a very popular approach to tackle this kind of problems is based on the use of

R. Cavoretto
Department of Mathematics "G. Peano", University of Torino, via Carlo Alberto 10, 10123 Torino, Italy
E-mail: roberto.cavoretto@unito.it

T. Schneider · P. Zulian
Faculty of Informatics, Università della Svizzera italiana, via Giuseppe Buffi 13, 6904 Lugano, Switzerland
E-mail: teseo.schneider@usi.ch, patrick.zulian@usi.ch

radial basis functions (RBFs) or some particular change of the basis [10,11,15, 20,31,33]. The main benefits deriving from the mesh-free nature of RBF based methods are: the flexibility with respect to geometric problem, the simplicity for the implementation in higher dimensions, and the high convergence order of the approximation scheme [43].

However, in applications such as science, engineering or scientific computing, where one has to deal with very large scattered data interpolation problems, global RBF methods are not applicable due to the high computational cost associated with the solution of large linear systems. For this reason, researchers have focused their attention on local RBF methods such as the radial basis function partition of unity method (RBF-PUM). The basic idea of PUM consists of decomposing the domain into several sub-domains forming a covering of the original domain, then constructing a local RBF interpolant on each sub-domain. Originally, the RBF-PUM (also known as RBF-PU method) has been introduced in the context of PDEs [2,25], but now it is also an effective and very used tool in the field of approximation theory and its applications [7,8,9,22,36,39].

Although previous works study efficient techniques for the organization of scattered data in two and three dimensions, the common use of serial (sequential) algorithms does not allow to interpolate a large number (*e.g.*, many millions) of scattered data-points in high space dimensions in reasonable time. For this reason, we aim at constructing a parallel algorithm that is suitable for our purposes in any space dimension, also providing an avenue to effectively avoid the “curse of dimensionality” [28].

Therefore, in this paper we propose a new strategy which solves the interpolation problem with linear computational time complexity with respect to the interpolation and evaluation points. This is achieved by using a specific choice of space-indexing algorithm. There exist many of such algorithms and data-structures for various applications and have different run time complexities. One category of space-indexing consists of space partitioning trees, such as bounding volume hierarchies, *kd*-trees, octrees, and many others [14]. Unfortunately, an efficient parallel construction of such trees is not trivial, some effort has been made for GPUs [29] or cluster computing [24,40]. Trees are particularly suitable for non-uniformly distributed points due to their hierarchical structure, however in our particular application this is not the case. In fact, since we consider uniformly distributed points, all levels of the tree would result in regular grids. On the one hand, this construction results in redundant information generated by the hierarchy. On the other hand the run-time complexity for constructing the tree is $O(n \log n)$ and for querying the tree for a single point is $O(\log n)$, where n is the total number of data-points. In our case, we can directly construct the lowest level of the tree in the form of an uniform grid in linear time because the hierarchy is implicit. The first advantage of such construction is that a query for a single point can be performed in constant time through an hash-function. The second advantage is that the parallelization is easier [18].

The concurrent nature of the RBF-PUM together with the particular space partitioning strategy, allows exploiting parallel processors, which efficiently solves large interpolation problems. There exists two main parallel computing paradigms: message based and shared-memory based [1]. Message based parallelism is related to cluster computing where the data is physically separated among multiple compute nodes. This type of parallelism is usually realized through standards such as MPI [21] and its principal challenge is exploiting the concurrency of a computational problem such that the algorithm speeds-up when the number of cores is increased. The speed-up is usually achieved by using problem specific strategies that allow for a balanced computation across the different cores and as little as possible inter-node communication. In shared-memory based parallelism threads are dynamically spanned for parallelizing specific parts of an algorithm, such as loops, concurrently accessing the same (shared) memory. Commonly used thread libraries are OpenMP [30], p-threads [26], and boost-threads [38]. The last two decades witnessed the rise and evolution of new parallel computing hardware such as general purposes GPUs and multi-core processors. Consequently, programming language extensions such as Cuda [27] and standards such as OpenCL [41] have been developed for harnessing the full power of computational hardware. Both Cuda and OpenCL provide a convenient run-time compiler API which allows for the *just-in-time (JIT) compilation* of computational kernels, hence enabling code generation and optimizations based on the input data. In fact, the code can be tailored for a specific run (*e.g.*, trough macros) allowing the compiler to apply low-level optimizations (*e.g.*, loop unrolling, vectorization). Although OpenCL supports heterogeneous computing it requires specific implementations for the different architecture because of the memory handling. For the aforementioned reasons we implement our strategy with the OpenCL standard for exploiting share-memory parallel processors specifically for CPUs. The main reason for this choice is that the local RBF-PUM problem is dense and requires a significant amount of memory per thread and GPUs have smaller ratio between memory and number of threads. Nevertheless our code also runs on GPUs for smaller problems with comparable performance.

The speed of our algorithm allows tackling the critical choice of the shape parameters associated with the RBFs automatically. In fact, the selection of such parameters may greatly influence the accuracy of the global fit [16]. Consequently, in order to avoid inaccurate or unreliable results, it turns out to be essential to design a mesh-free approximation scheme that predicts optimal – or more probably reliable – values of the shape parameter via any error estimate [15]. For this reason we propose to couple the RBF-PUM with a *cross-validation* approach such as the *leave-one-out cross-validation* (LOOCV) [35]. This technique allows searching for optimal values of the shape parameters associated with each local RBF interpolant, thus reducing the global interpolation error. These effects are illustrated in our numerical experiments for bivariate, trivariate and spherical interpolation.

The paper is organized as follows. In Section 2 we recall some theoretical results for the RBF-PUM interpolation. Section 3 analyzes the LOOCV technique used to locally select the RBF shape parameters. In Section 4 we explain in all the different phases of our algorithm and compare their performances. In Section 5 we report numerical experiments devoted to point out interpolation errors and computational times of our parallel algorithm. Section 6 contains an application with real world data. Section 7 deals with conclusions and future work.

2 RBF based mesh-free methods

In this section we first review the main theoretical aspects concerning RBF interpolation, and we deal with the RBF-PU method based on a local use of RBF

RBF	$\phi_\varepsilon(r)$
Gaussian C^∞ (GA)	$\mathrm{e}^{-\varepsilon^2 r^2}$
Inverse MultiQuadric C^∞ (IMQ)	$(1+\varepsilon^2 r^2)^{-1/2}$
Matérn C^6 (M6)	$\mathrm{e}^{-\varepsilon r}(\varepsilon^3 r^3 + 6\varepsilon^2 r^2 + 15\varepsilon r + 15)$
Matérn C^4 (M4)	$\mathrm{e}^{-\varepsilon r}(\varepsilon^2 r^2 + 3\varepsilon r + 3)$
Matérn C^2 (M2)	$\mathrm{e}^{-\varepsilon r}(\varepsilon r + 1)$
Wendland C^6 (W6)	$(1-\varepsilon r)_+^8\,(32\varepsilon^3 r^3 + 25\varepsilon^2 r^2 + 8\varepsilon r + 1)$
Wendland C^4 (W4)	$(1-\varepsilon r)_+^6\,(35\varepsilon^2 r^2 + 18\varepsilon r + 3)$
Wendland C^2 (W2)	$(1-\varepsilon r)_+^4\,(4\varepsilon r + 1)$

Table 1 Examples of strictly positive definite RBFs, where $r = ||\cdot||_2$ is the Euclidean norm and $(\cdot)_+$ denotes the truncated power function.

interpolants. Both computational techniques are mesh-free and effectively works with scattered data-points, therefore they turn out to be flexible in terms of the geometry of the domain [5].

2.1 RBF interpolation

Given N distinct *data-points* or *nodes* $\boldsymbol{x}_1,\dots,\boldsymbol{x}_N \in \boldsymbol{X}_N$ in a domain $\Omega \subseteq \mathbb{R}^s$ and corresponding *data* or *function values* $f(\boldsymbol{x}_1),\dots,f(\boldsymbol{x}_N)$ obtained by possibly sampling any (unknown) function $f:\Omega\to\mathbb{R}$, the standard RBF interpolation problem consists of finding an interpolant $R:\Omega\to\mathbb{R}$ of the form

$$R(\boldsymbol{x}) = \sum_{i=1}^{N} c_i \phi_\varepsilon(||\boldsymbol{x}-\boldsymbol{x}_i||_2), \quad \boldsymbol{x}\in\Omega, \tag{1}$$

where c_i is an unknown real coefficient, $||\cdot||_2$ denotes the Euclidean norm, and $\phi:\mathbb{R}_{\geq 0}\to\mathbb{R}$ is a strictly positive definite RBF depending on a *shape parameter* $\varepsilon > 0$ such that $\phi_\varepsilon(||\boldsymbol{x}-\boldsymbol{z}||_2) = \phi(\varepsilon||\boldsymbol{x}-\boldsymbol{z}||_2)$, for all $\boldsymbol{x},\boldsymbol{z}\in\Omega$. For simplicity, from now on we refer to ϕ_ε as ϕ. In Table 1 we report a list of some strictly positive definite RBFs with their orders of smoothness. Note that Gaussian, Inverse MultiQuadric and Matérn functions are globally supported and strictly positive definite in $\mathbb{R}^s$ for any s, whereas Wendland functions are compactly supported – with support $[0,1/\varepsilon]$ – and strictly positive definite in $\mathbb{R}^s$ for $s\leq 3$ [43].

In order to determine the coefficients $c_1,\dots,c_N$, we enforce the interpolation conditions $R(\boldsymbol{x}_i) = f(\boldsymbol{x}_i)$, $i=1,\dots,N$, and, as a result, we obtain a symmetric linear system of equations

$$\boldsymbol{\Phi}\boldsymbol{c} = \boldsymbol{f}, \tag{2}$$

where $\boldsymbol{c} = (c_1,\dots,c_N)^T$, $\boldsymbol{f} = (f_1,\dots,f_N)^T$, and the interpolation matrix $\boldsymbol{\Phi}\in\mathbb{R}^{N\times N}$ is given by $\boldsymbol{\Phi}_{ki} = \phi(||\boldsymbol{x}_k-\boldsymbol{x}_i||_2)$, $k,i=1,\dots,N$. Since ϕ is a strictly positive function, the associated matrix $\boldsymbol{\Phi}$ is nonsingular and the RBF interpolation problem is well-posed, hence a solution to the problem exists and is unique [16].

Therefore, when the vector $\boldsymbol{c}$ is found, we can evaluate the RBF interpolant at any point $\boldsymbol{x}$ as

$$R(\boldsymbol{x}) = \boldsymbol{\phi}^T(\boldsymbol{x})\boldsymbol{c},$$

where $\boldsymbol{\phi}^T(\boldsymbol{x}) = (\phi(||\boldsymbol{x}-\boldsymbol{x}_1||_2), \ldots, \phi(||\boldsymbol{x}-\boldsymbol{x}_N||_2))$. In particular, the interpolant R is a function of the *native Hilbert space* $\mathcal{N}_\phi(\Omega)$ uniquely associated with the RBF, and, if $f \in \mathcal{N}_\phi$, then R is the $\mathcal{N}_\phi$-projection of f into the subspace $\mathcal{N}_\phi(\boldsymbol{X}_N) = \{\phi(||\boldsymbol{x}-\boldsymbol{x}_i||_2), \boldsymbol{x}_i \in \boldsymbol{X}_N\}$ [43].

2.2 RBF-PU approximation

Let $\Omega \subseteq \mathbb{R}^s$ be an open bounded domain, and let $\{\Omega_j\}_{j=1}^d$ be an open bounded covering of Ω satisfying some mild overlap condition among the sub-domains Ω_j and that

$$I(\boldsymbol{x}) = \{j : \boldsymbol{x} \in \Omega_j\}, \quad \text{card}(I(\boldsymbol{x})) \le K, \quad \forall \boldsymbol{x} \in \Omega.$$

In other words, the set $I(\boldsymbol{x})$ is uniformly bounded by the constant K (independent of d) on Ω, where $\Omega \subseteq \bigcup_{j=1}^d \Omega_j$. With the sub-domains we define a partition of unity $\{W_j\}_{J=1}^d$ subordinated to the covering $\{\Omega_j\}_{j=1}^d$ such that

$$\sum_{j=1}^d W_j(\boldsymbol{x}) = 1, \quad \boldsymbol{x} \in \Omega,$$

where the weight $W_j : \Omega_j \to \mathbb{R}$ is a compactly supported, nonnegative and continuous function with $\text{supp}(W_j) \subseteq \Omega_j$. Then, for each sub-domain we construct, similary to (1), a local RBF interpolant $R_j : \Omega_j \to \mathbb{R}$ of the form

$$R_j(\boldsymbol{x}) = \sum_{i=1}^{N_j} c_i^j \phi(||\boldsymbol{x}-\boldsymbol{x}_i^j||_2), \tag{3}$$

where N_j indicates the number of data-points in Ω_j (*i.e.*, the points $\boldsymbol{x}_i^j \in \boldsymbol{X}_{N_j} = \boldsymbol{X}_N \cap \Omega_j$) and then define the global PU interpolant

$$\mathcal{I}(\boldsymbol{x}) = \sum_{j=1}^d R_j(\boldsymbol{x})W_j(\boldsymbol{x}), \quad \boldsymbol{x} \in \Omega. \tag{4}$$

We remark that, if the functions R_j, $j = 1, \ldots, d$, satisfy the interpolation conditions

$$R_j(\boldsymbol{x}_i^j) = f(\boldsymbol{x}_i^j), \quad \boldsymbol{x}_i^j \in \Omega_j, \quad i = 1, \ldots, N_j, \tag{5}$$

then the global interpolant (4) inherits the interpolation property of the local interpolants [16]

$$\mathcal{I}(\boldsymbol{x}_i^j) = \sum_{j=1}^d R_j(\boldsymbol{x}_i^j)W_j(\boldsymbol{x}_i^j) = \sum_{j=1}^d f(\boldsymbol{x}_i^j)W_j(\boldsymbol{x}_i^j) = f(\boldsymbol{x}_i^j).$$

Solving the j-th interpolation problem (5) results in the linear system generated by local RBFs

$$\begin{pmatrix} \phi(||\boldsymbol{x}_1^j - \boldsymbol{x}_1^j||_2) & \cdots & \phi(||\boldsymbol{x}_1^j - \boldsymbol{x}_{N_j}^j||_2) \\ \vdots & \vdots & \vdots \\ \phi(||\boldsymbol{x}_{N_j}^j - \boldsymbol{x}_1^j||_2) & \cdots & \phi(||\boldsymbol{x}_{N_j}^j - \boldsymbol{x}_{N_j}^j||_2) \end{pmatrix} \begin{pmatrix} c_1^j \\ \vdots \\ c_{N_j}^j \end{pmatrix} = \begin{pmatrix} f_1^j \\ \vdots \\ f_{N_j}^j \end{pmatrix},$$

or simply

$$\boldsymbol{\Phi}_j \boldsymbol{c}_j = \boldsymbol{f}_j. \tag{6}$$

Note that existence and uniqueness of the solution and nonsingularity of the local matrix $\boldsymbol{\Phi}_j$ is guaranteed by the use of strictly positive definite functions ϕ [16].

Now, let us consider the following definition [42].

Definition 1 Let $\Omega \subseteq \mathbb{R}^s$ be a bounded set, and $\{\Omega\}_{j=1}^d$ be an open bounded covering of Ω. This means that all Ω_j are open and bounded and that Ω is contained in their union. A family of nonnegative functions $\{W_j\}_{j=1}^d$ with $W_j \in C^k(\mathbb{R}^s)$ is called a *k-stable partition of unity* with respect to the covering $\{\Omega_j\}_{j=1}^d$ if:

- $\mathrm{supp}(W_j) \subseteq \Omega_j$;
- $\sum_{j=1}^d W_j(\boldsymbol{x}) \equiv 1$ on Ω;
- for every $\beta \in \mathbb{N}_0^s$ with $|\beta| \leq k$ there exists a constant $C_\beta > 0$ such that

$$||D^\beta W_j||_{L_\infty(\Omega_j)} \leq \frac{C_\beta}{\delta_j^{|\beta|}}, \qquad j = 1, \dots, d,$$

where $\delta_j = \mathrm{diam}(\Omega_j) = \sup_{\boldsymbol{x},\boldsymbol{z} \in \Omega_j} ||\boldsymbol{x} - \boldsymbol{z}||_2$.

For each sub-domain Ω_j we can thus construct PU weight function W_j using the Shepard method as follows

$$W_j(\boldsymbol{x}) = \frac{\varphi_j(\boldsymbol{x})}{\sum_{k=1}^d \varphi_k(\boldsymbol{x})}, \quad j = 1, \dots, d, \tag{7}$$

with $\varphi_j(\boldsymbol{x})$ being a compactly supported function with support on Ω_j such as the W2 function (see Table 1). Such functions are scaled with a shape parameter σ to get $\varphi_j(\boldsymbol{x}) = \varphi(\sigma||\boldsymbol{x} - \boldsymbol{\xi}_j||)$, where $\boldsymbol{\xi}_j$ is the center of the weight function.

In order to be able to formulate error bounds, we define the *fill distance*

$$h_{\boldsymbol{X}_N,\Omega} = \sup_{\boldsymbol{x} \in \Omega} \min_{\boldsymbol{x}_i \in \boldsymbol{X}_N} ||\boldsymbol{x} - \boldsymbol{x}_i||_2 \tag{8}$$

and make some further assumptions on regularity of Ω_j [42].

Definition 2 Let $\Omega \subseteq \mathbb{R}^s$ be bounded and let $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N \in \boldsymbol{X}_N \subseteq \Omega$ be given. An open and bounded covering $\{\Omega_j\}_{j=1}^d$ is called regular for $(\Omega, \boldsymbol{X}_N)$ if:

- for each $\boldsymbol{x} \in \Omega$, the number of sub-domains Ω_j with $\boldsymbol{x} \in \Omega_j$ is bounded by a global constant M;
- each sub-domain Ω_j satisfies an interior cone condition [43];
- the local fill distances $h_{\boldsymbol{X}_{N_j},\Omega_j}$ are uniformly bounded by the global fill distance (8).

After defining the space $C_{\nu}^{k}(\mathbb{R}^s)$ of all functions $f \in C^k$ whose derivatives of order $|\alpha| = k$ satisfy $D^{\alpha} f(\boldsymbol{x}) = \mathcal{O}(||\boldsymbol{x}||_2^{\nu})$ for $||\boldsymbol{x}||_2 \to 0$, we consider the following convergence result, see [16, Theorem 29.1] and [43, Theorem 15.9].

Theorem 1 *Let $\Omega \subseteq \mathbb{R}^s$ be open and bounded and $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N \in \boldsymbol{X}_N \subseteq \Omega$. Let $\phi \in C_{\nu}^{k}(\mathbb{R}^s)$ be a strictly conditionally positive definite function of order m. If $\{\Omega_j\}_{j=1}^{d}$ is a regular covering for $(\Omega, \boldsymbol{X}_N)$ and $\{W_j\}_{j=1}^{d}$ is k-stable for $\{\Omega_j\}_{j=1}^{d}$, then the error between $f \in \mathcal{N}_{\phi}(\Omega)$ and its PU interpolant* (4) *is bounded by*

$$|D^{\alpha} f(\boldsymbol{x}) - D^{\alpha}\mathcal{I}(\boldsymbol{x})| \leq C h_{\boldsymbol{X}_N,\Omega}^{(k+\nu)/2-|\alpha|} |f|_{\mathcal{N}_{\phi}(\Omega)}, \quad \forall \boldsymbol{x} \in \Omega, \, |\alpha| \leq k/2.$$

Comparing this convergence result with the global error estimates in [43], we note that the PU preserves the local approximation order for the global interpolant (4). So we can efficiently solve a large problem by solving small RBF interpolation problems (possibly in parallel) and then glue them along with the PU weights $\{W_j\}_{j=1}^{d}$. Consequently, the PU approach turns out to be a simple and effective technique to decompose a large interpolation problem into many small problems, simultaneously ensuring that the accuracy obtained for the local fits is carried over to the global one.

3 Selection of ε via LOOCV

The accuracy of RBF based methods highly depends upon the shape parameter ε of the basis functions, which is responsible for the flatness of the functions. In particular, for smooth problems the best accuracy is typically achieved when ε is small, but then the condition number of the linear system becomes very large. Therefore, in order to get reliable approximation results, we need to find a technique that allows detecting a suitable value of ε for each PU sub-domain. In fact, since the RBF-PUM is based on the solution of d (usually small) linear systems of the form (6), the selection of shape parameters may greatly affect the accuracy of the global PU interpolant.

A good way to select a shape parameter ε is to use locally the LOOCV technique [35]. The idea behind LOOCV in the RBF-PU interpolation is to split the data of each sub-domain Ω_j, $j = 1, \dots, d$, into two different sets: a *training set* $\{f(\boldsymbol{x}_1^j), \dots, f(\boldsymbol{x}_{k-1}^j), f(\boldsymbol{x}_{k+1}^j), \dots, f(\boldsymbol{x}_{N_j}^j)\}$, and a *validation set* consisting of only the single value $f(\boldsymbol{x}_k^j)$ which was left out when creating the training set [17]. Now, for a fixed $k \in \{1, \dots, N_j\}$ and fixed ε, we define the partial RBF interpolant

$$R_j^{[k]}(\boldsymbol{x}) = \sum_{i=1,\ i \neq k}^{N_j} c_i^j \phi(||\boldsymbol{x} - \boldsymbol{x}_i^j||_2),$$

whose coefficients c_i^j are determined by interpolating the training data

$$R_j^{[k]}(\boldsymbol{x}_i^j) = f(\boldsymbol{x}_i^j), \qquad i = 1, \dots, k-1, k+1, \dots, N_j.$$

In order to measure the quality of this attempt, we define the error

$$e_k^j(\varepsilon) = f(\boldsymbol{x}_k^j) - R_j^{[k]}(\boldsymbol{x}_k^j) \tag{9}$$

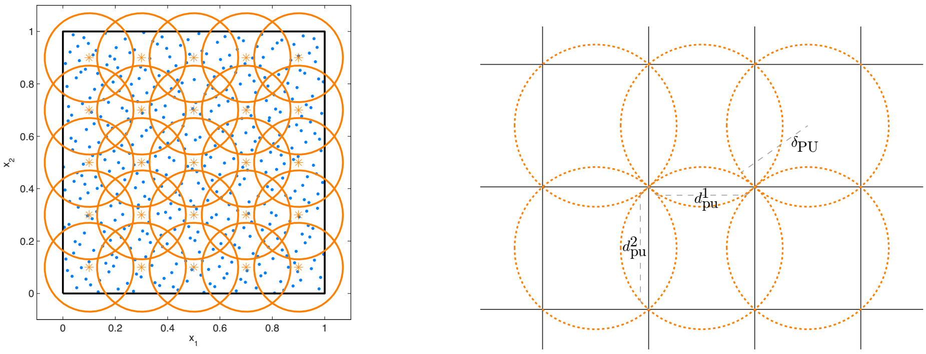


Fig. 1 Left: example of PU sub-domains (orange circles) and scattered node distribution (blue dots) in $\Omega = [0,1]^2 \subseteq \mathbb{R}^2$. Right: notation for the spatial hashing.

at the one validation point $\boldsymbol{x}_k^j$ not used to determine the interpolant. The "optimal" value of ε is found as

$$\varepsilon_{opt}^j = \mathrm{argmin}_\varepsilon ||\boldsymbol{e}_j(\varepsilon)||, \qquad \boldsymbol{e}_j = (e_1^j, \ldots, e_{N_j}^j)^T,$$

where $||\cdot||$ is any norm, however we use $||\cdot||_\infty$. We can now compute a vector $\boldsymbol{\varepsilon}_{opt} = (\varepsilon_{opt}^1, \ldots, \varepsilon_{opt}^d)^T$ containing an "optimal" value of ε, hence $\varepsilon = \varepsilon_{opt}^j$, for the RBF interpolant (3) defined on Ω_j, $j = 1, \ldots, d$.

The important thing is that we can find the error vector $\boldsymbol{e}_j$ without solving dN_j problems, each of size $(N_j - 1) \times (N_j - 1)$. In fact, instead of (9), the computation of the error components can be expressed in terms of the interpolation matrix $\boldsymbol{\Phi}_j$ in (6) as follows

$$e_k^j(\varepsilon) = \frac{c_k^j}{(\boldsymbol{\Phi}_j)_{kk}^{-1}},$$

where c_k^j is the k-th coefficient in the full RBF interpolant (3) and $(\boldsymbol{\Phi}_j)_{kk}^{-1}$ is the kth diagonal element of the inverse of the corresponding $N_j \times N_j$ interpolation matrix $\boldsymbol{\Phi}_j$ [17]. The LOOCV problem can be solved using the Brent's method [4].

4 Parallel algorithm

In the RBF-PU scheme the sub-domains can be of any (regular enough) geometric shape, such as $(s-1)$-spheres, s-cubes and s-orthotopes with $s \geq 2$. However, many other possible types of sub-domains are allowed. The usual requirement for all shapes of PU sub-domains is that they cover the domain $\Omega \subseteq \mathbb{R}^s$. In this paper we use circular sub-domains so that any (possibly also mild) overlap among the different sub-domains is ensured [36]. A typical example of PU sub-domains and node distribution in $\mathbb{R}^2$ is shown in Figure 1 left.

The PUM combined with RBF interpolation is suitable for shared-memory parallel computations for three main reasons. First, the computation of the coefficients c_i^j is completely independent across different partitions. Second, the evaluation of the global interpolant $\mathcal{I}$ can be performed separately for each evaluation point. Third, the intersection detection between the sub-domains Ω_j and either the data-points $\boldsymbol{X}_N$ or the evaluation points $\boldsymbol{y}_1, \dots, \boldsymbol{y}_{N_{eval}} \in \boldsymbol{Y}$ can be efficiently parallelized using space partitioning strategies.

The uniform distribution of the data-points allows for an *ad-hoc* space partitioning in a form of a structured grid. The grid is constructed by dividing each dimension k of the axis-align bounding-box $B = [\boldsymbol{B}_m, \boldsymbol{B}_M]$, in

$$d_{PU}^k = \left\lfloor \left\lceil \frac{1}{2} \left(\frac{N}{2} \right)^{1/s} \right\rceil \frac{B_M^k - B_m^k}{\displaystyle\min_{l=1,\dots,s} (B_M^l - B_m^l)} \right\rfloor$$

intervals which creates $d = \prod_{k=1}^{s} d_{PU}^k$ cells C_j, where B_m^k, B_M^k denote the k-th components of the respective vectors, $\lfloor \cdot \rfloor$ is the floor operator, and $\lceil \cdot \rceil$ is the ceiling operator. Note that the number of sub-domains d is proportional to the number of nodes N, that is $N/d \approx 2^{s+1}$ [6]. Nevertheless, in general, one could also study different PU strategies, suitably increasing (decreasing) the number of sub-domains to be used. From a computational point of view, we would expect to increase (reduce) the number of sub-domains, reducing (increasing) at the same time the sub-domain size. A change of this type may influence more or less significantly the approximation results in term of both accuracy and stability. In fact, a cover with small (large) partitions results in worse (better) approximations, even if it turns out to be computationally cheaper (more expensive); for further details we refer to [7,17].

The grid implicitly generates the space partitioning where the centre of each cell C_j corresponds to the centre of the hyper-spherical sub-domain Ω_j with radius

$$\delta_{PU} = \frac{\sqrt{2}}{\min_k d_{PU}^k} \min_k (B_M^k - B_m^k) \tag{10}$$

which entirely covers the volume of the cell as shown in Figure 1 right.

Given a point $\boldsymbol{x}$, for finding the cell $C_j = C_{j(\boldsymbol{x})}$ containing it we evaluate the hash-function

$$j(\boldsymbol{x}) = \sum_{k=1}^{s} \left(\left\lfloor (x^k - B_m^k)/(B_M^k - B_m^k) \cdot d_{PU}^k \right\rfloor \prod_{l=k+1}^{s} d_{PU}^l \right), \tag{11}$$

where x^k is the k-th coordinate of $\boldsymbol{x}$. By exploiting this function we build the inverse index which maps a cell to its contained points. Finally, for identifying the set of points within a sub-domain Ω_j we visit the corresponding cell C_j and its $3^s - 1$ neighbours and check if the contained points are within the radius δ_{PU} of Ω_j, see Section 4.1.

This condition, along with the value of the sub-domain radius (10), enables the algorithm to efficiently work for any space dimension s.

Once we have identified the points contained in each sub-domain we assemble the matrix $\boldsymbol{\Phi}_j$ and compute the local coefficients c_i^j by solving the interpolation system (6). Finally, for interpolating the function at a point $\boldsymbol{y}$, we again exploit the hash-function $j(\cdot)$ to identify all the intersecting cells and consequently the intersecting sub-domains so that we can build the global interpolant $\mathcal{I}$ in (4), as explained in Section 2.2.

Among shared-memory domain specific languages we opt for OpenCL [41]. The main advantage is that OpenCL is an open standard designed for heterogeneous computing, which allows exploiting different types of computational resources, such as CPUs, GPGPUs, FPGAs. Additionally, OpenCL is suitable for code-generation and JIT compilation, which allows *ad-hoc* memory management and compile-time optimization (for instance loop unrolling and private memory allocation based on problem dimensions). However, the flexibility of JIT code generation requires compiling the code whenever the algorithm is used, adding some neglectable overhead.

The adoption of the parallel paradigm of standards such as OpenCL, implies a special memory handling due to the limitations of graphics hardware. Such hardware requires the explicit handling of the different memory hierarchies, and precludes dynamic memory-allocation within compute kernels. Here, we do not address the memory-hierarchy issue since we focus on CPU-based architectures, however we tackle the problem concerning storing and solving the local interpolation problems, as explained in Section 4.2.

4.1 Space indexing

In order to efficiently index the input data-points $\boldsymbol{x}_i \in \boldsymbol{X}_N$ with respect to the sub-domains Ω_j we exploit a compressed storage indexing format. This format $\Gamma = \{\boldsymbol{a}^{\mathrm{idx}}, \boldsymbol{a}^{\mathrm{ptr}}\}$ consists of two arrays, the first one $\boldsymbol{a}^{\mathrm{idx}}$ contains the indices pointing to the actual data, and the second $\boldsymbol{a}^{\mathrm{ptr}}$ stores the offsets to access specific ranges within the index array $\boldsymbol{a}^{\mathrm{idx}}$, see Figure 2 middle. We exploit the compressed storage format in our two-stage algorithm, whose overview is illustrated in Algorithm 1.

The goal of the first stage is computing the mapping Γ_C from each cell C_j to its contained points. To this end, we first use the hash-function (11) to compute in parallel the vector $\boldsymbol{J}$ which stores the association $j_i = j(\boldsymbol{x}_i)$ from $\boldsymbol{x}_i$ to cell $C_{j(\boldsymbol{x}_i)}$, see Figure 2 left. Then, for improving caching when accessing memory, we rearrange the data-points, using Γ_C, such that the points belonging to the same cell are stored contiguosly in memory, which allows for 20% speed-up. Then, in serial, we employ a variation of the bucket-sort algorithm (see Algorithm 2), for creating the compressed-storage index Γ_C storing the association from cell to its contained points. We use the initial part of the bucket-sort algorithm where a cell C_j corresponds to the j-th "bucket", and the elements are the point indices i. Note that this stage has linear time computational complexity, since we do not need to sort the elements once they are within each bucket.

The goal of the second stage is creating the mapping Γ_Ω from each sub-domain Ω_j to its contained points. We first count in parallel the number of points in the hyper-spherical sub-domains Ω_j exploiting Γ_C and store them in $\boldsymbol{q}_\Omega$. The vector $\boldsymbol{q}_\Omega$, whose entries are the number of points N_j in the corresponding sub-domain Ω_j, is necessary for allocating the required memory. For each cell in parallel, we visit all its $3^s - 1$ neighbouring cells, and count the number of points that are within the sub-domain radius δ_{PU}, see Figure 2 right. Then, in serial, we create the sub-domain pointer by computing a cumulative sum of $\boldsymbol{q}_\Omega$. Finally, we fill the compressed storage index with the indices of intersecting points, which are selected the same way as for the computation of $\boldsymbol{q}_\Omega$. Note that duplicate indices may appear across different sets since the points might belong to multiple sub-

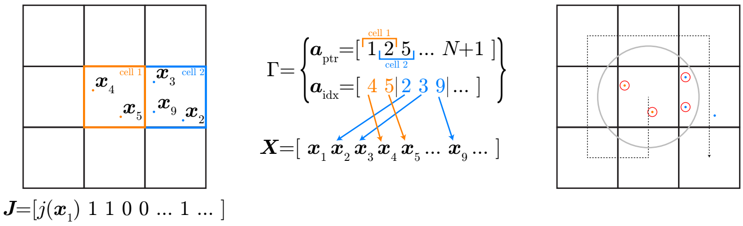

cell 1
cell 2
x_4
x_5
x_3
x_9
x_2
J=[$j(x_1)$ 1 1 0 0 ... 1 ...]
Γ=
a_{ptr}=[1 2 5 ... N+1]
a_{idx}=[4 5|2 3 9|...]
X=[x_1 x_2 x_3 x_4 x_5 ... x_9 ...]

Fig. 2 Overview of the space partitioning data-structures and algorithm.

Algorithm 1: Overview of the space indexing algorithm.

```
Data: $x_i \in X_N$
Result: $\Gamma_\Omega$

  First stage
    $J$ ← in parallel, compute the hash function for all $x_i$;
    $\Gamma_C$ ← in serial, count the number of points contained in each cell $C_j$ and create
      the mapping from cell to contained points applying a partial sorting, using $J$;
  Second stage
    $q_\Omega$ ← in parallel, count the number of points contained in each $\Omega_j$ to determine
      memory size, using $\Gamma_C$;
    $\Gamma_\Omega$ ← in serial, create the sub-domain pointer array and, in parallel, fill the index
      array with the mapping from $\Omega_j$ to its contained points, using $\Gamma_C$ and $q_\Omega$.
```

domains. This compressed storage index is used both when computing the local coefficients and when evaluating the interpolant $\mathcal{I}$.

4.2 Computation of the local coefficients

The dense matrix $\boldsymbol{\Phi}_j$ of the interpolation system (6) is symmetric positive definite. However, the system may become ill-conditioned when data-points are close to each other or even semi-definite because of numerical truncation. For all forementioned reasons we employ the Cholesky decomposition $\boldsymbol{\Phi}_j = \boldsymbol{LDL}^T$, which allows for efficiently and accurately computing the solution $\boldsymbol{c}_j$ by the forward-

Algorithm 2: Construction of the compressed storage index.

```
Data: $J \in \mathbb{N}^N, q \in \mathbb{N}^d, I \in \mathbb{N}^d$
Result: $\Gamma = \{a^{\mathrm{idx}} \in \mathbb{N}^{d+1}, a^{\mathrm{ptr}} \in \mathbb{N}^N\}$

Count points in cells:
for $k = 1, \dots, N$ do
    $q_{J_k} \leftarrow q_{J_k} + 1$
end

Create cell pointer:
$a_1^{\mathrm{ptr}} \leftarrow 1$
for $k = 1, \dots, d$ do
    $a_{k+1}^{\mathrm{ptr}} \leftarrow a_k^{\mathrm{ptr}} + q_k$
end

Create cell to point index:
for $k = 1, \dots, d$ do
    $I_k \leftarrow 0$
end

for $k = 1, \dots, N$ do
    $i \leftarrow J_k, \quad j \leftarrow a_i^{\mathrm{ptr}}$
    $a_{j+I_i}^{\mathrm{idx}} \leftarrow k$
    $I_i \leftarrow I_i + 1$
end
```

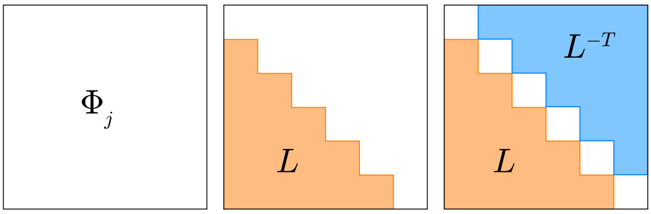


Fig. 3 Visualization of the memory layout $\boldsymbol{A}$ for different stages of the decomposition algorithm. Initially $\boldsymbol{A} = \boldsymbol{\Phi}_j$, then we compute $\boldsymbol{L}$ and store it in the lower triangular region of $\boldsymbol{A}$, finally we invert $\boldsymbol{L}$ and save it in the upper triangular part of $\boldsymbol{A}$.

substitution $\boldsymbol{L}^{-1}\boldsymbol{f}_j$ followed by the pseudo-inverse diagonal scaling $\boldsymbol{D}^{-1}(\boldsymbol{L}^{-1}\boldsymbol{f}_j)$, and the transposed backward-substitution $\boldsymbol{L}^{-T}(\boldsymbol{D}^{-1}\boldsymbol{L}^{-1}\boldsymbol{f}_j)$.

As previously mentioned, the main challenge of computing the coefficients c_i^j for the local interpolants R_j in (3) is the memory management. The interpolation systems (6) may become very large for $s > 2$, which precludes the use of *private* memory (*i.e.*, static allocation on the stack). This limitation force us to dynamically allocate memory either using *local* or *global* address spaces. This implies that the total amount of memory grows with the number of interpolation systems solved in parallel.

For avoiding unnecessary memory consumption, we adapt the Cholesky decomposition to be computed and stored completely in place. Let $\boldsymbol{A} \in \mathbb{R}^{n\times n}$ be the n^2 size storage for both the input matrix $\boldsymbol{\Phi}_j$ and the triangular matrix $\boldsymbol{L}$. We first fill $\boldsymbol{A}$ with the entries of $\boldsymbol{\Phi}_j$ (Figure 3 left), then we overwrite them with the entries of $\boldsymbol{L}$ as shown in Algorithm 3 and Figure 3 middle. Note that this procedure also computes the vector $\boldsymbol{d} = \mathrm{diag}(\boldsymbol{D}) \in \mathbb{R}^n$ which does not requires special memory handling since it grows linearly; the same holds for the auxiliary storage vectors $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^n$.

Algorithm 3: In place Cholesky factorization of $\boldsymbol{\Phi}_j$.

```
Data: $\boldsymbol{A} = \boldsymbol{\Phi}_j \in \mathbb{R}^{n\times n}, \boldsymbol{d} \in \mathbb{R}^n$, $\boldsymbol{v} \in \mathbb{R}^n$ , $\boldsymbol{w} \in \mathbb{R}^n$
Result: $\boldsymbol{A} = \boldsymbol{L} + \mathbf{Id}, \boldsymbol{d} = \mathrm{diag}(\boldsymbol{D})$

$v_1 \leftarrow A_{1,1},\ d_1 \leftarrow A_{1,1},$
  $A_{1,1} \leftarrow 0$

for $i = 2, \dots, n$ do
    $A_{i,1} \leftarrow A_{i,1}/v_1$
end

for $j = 2, \dots, n-1$ do
    for $k = 1 \dots, j-1$ do
        $v_k \leftarrow A_{j,k}\, d_k$
    end

    $v_j \leftarrow A_{j,j}$
    for $k = 1, \dots j-1$ do
        $v_j \leftarrow v_j - A_{j,k}\, v_k$
    end

    $d_j \leftarrow v_j$
end

for $j = 2, \dots, n-1$ do
    for $k = j+1, \dots, n$ do
        $w_k \leftarrow 0$
        for $l = 1, \dots, j-1$
         do
            $w_k \leftarrow$
              $w_k + A_{k,l}\, v_l$
        end
    end

    for $k = j+1, \dots n$ do
        $A_{k,j} \leftarrow$
          $(A_{k,j} - w_k)/v_j$
    end
end

for $k = 1, \dots n-1$ do
    $v_k \leftarrow A_{n,k}\, d_k$
end

$v_n \leftarrow A_{n,n}$
for $k = 1, \dots n-1$ do
    $v_n \leftarrow v_n - A_{n,k}\, v_k$
end

$d_n \leftarrow v_n$
for $k = 1, \dots n$ do
    $A_{i,i} \leftarrow 1$
end
```

Algorithm 4: In place computation of the inverse transpose of lower triangular matrix $\boldsymbol{L}$.

```
Data: $\boldsymbol{A} = \boldsymbol{L} + \mathbf{Id} \in \mathbb{R}^{n \times n}$
Result: $\boldsymbol{A} = \boldsymbol{L} + \boldsymbol{L}^{-T} + \mathbf{Id}$

for $k = 1, \ldots, n$ do
    for $i = k + 1, \ldots, n$ do
        $A_{k,i} \leftarrow 0$
        for $j = k, \ldots, i - 1$ do
            $A_{k,i} \leftarrow A_{k,i} - A_{i,j}\, A_{k,j}$
        end
        $A_{k,i} \leftarrow A_{k,i} / A_{i,i}$
    end
end
```

Algorithm 5: Simplified computation of the diagonal of $\boldsymbol{\Phi}_j^{-1}$.

```
Data: $\boldsymbol{A} = \boldsymbol{L} + \boldsymbol{L}^{-T} + \mathbf{Id} \in \mathbb{R}^{n \times n}$,
      $\boldsymbol{d} = \mathrm{diag}(\boldsymbol{D}) \in \mathbb{R}^n, \boldsymbol{g} \in \mathbb{R}^n$
Result: $\boldsymbol{g} = \mathrm{diag}(\boldsymbol{\Phi}_j^{-1})$

for $k = 1, \ldots, n$ do
    $g_k \leftarrow 0$
end
for $k = 1, \ldots, n$ do
    for $i = 1, \ldots, k$ do
        $g_k \leftarrow g_k + (A_{i,k})^2 / d_k$
    end
end
```

4.2.1 Computation of the local error estimator

For computing the optimal ε as introduced in Section 3, we need to evaluate the error $\boldsymbol{e}_j$, which requires the computation of the diagonal of the inverse of $\boldsymbol{\Phi}_j$. For this purpose, we exploit the quantities $\boldsymbol{L}$ and $\boldsymbol{d}$ which are already computed as explained in the previous section. Again, we need to exploit the allocated storage $\boldsymbol{A}$ and avoid allocating new memory. At this stage of the algorithm the matrix $\boldsymbol{\Phi}_j$ is not needed any more, hence we store $\boldsymbol{L}^{-T}$ on the upper triangular part of $\boldsymbol{A}$ as shown in Figure 3 right. The procedure for computing $\boldsymbol{L}^{-T}$ can be interpreted as the forward-substitution $\boldsymbol{L}^{-1}\mathbf{Id}$ as illustrated in Algorithm 4. Having $\boldsymbol{L}^{-T}$, allows computing $\boldsymbol{g} = \mathrm{diag}(\boldsymbol{\Phi}_j^{-1})$ by simplifying the operations in $\mathrm{diag}(\boldsymbol{L}^{-T}\boldsymbol{D}^{-1}\boldsymbol{L}^{-1})$ as shown in Algorithm 5.

4.3 Performance

The algorithms described in the previous sections contain both serial and parallel parts (*e.g.*, the interpolation is fully parallel while the construction of the compressed storage index is serial). In a shared-memory context, some computations

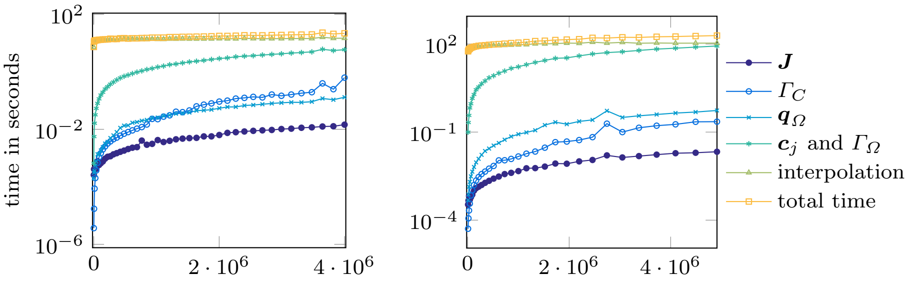


Fig. 4 Computational timings for the different parts of the algorithm with respect to the number N of data-points, evaluating the interpolant (4) at 9 and 8 million points for 2D and 3D respectively. Results for 2D on the left and for 3D on the right.

	$\boldsymbol{J}$	Γ_C	$\boldsymbol{q}_\Omega$	$\boldsymbol{c}_j, \Gamma_\Omega$
2D	0.2%	3.6%	1.8%	94.4%
3D	0.03%	0.2%	0.6%	99.2%

Table 2 Percentages of the average computational times excluding the evaluation phase.

are more efficient in serial than in parallel since they may require heavy synchronization or replicated memory. For instance, creating the compressed-storage index in parallel would involve either synchronizing the increment of each entry of the cell pointer $\boldsymbol{a}^{\text{ptr}}$ (see Algorithm 2), or replicating $\boldsymbol{a}^{\text{ptr}}$ for each thread.

Figure 4 shows the times of different computational phases. We see that the influence of the serial computation of the compressed storage index Γ_C on the overall time is negligible. In fact, it is comparable with the times for computing the hash-function $\boldsymbol{J}$ and counting the points within the sub-domains $\boldsymbol{q}_\Omega$. For efficiency reason, we combine the computation of the local coefficients $\boldsymbol{c}_j$ and the compressed storage Γ_Ω since for both we need to search for points contained in a sub-domain Ω_j. Note that, without considering the evaluation of $\mathcal{I}$, most of the time is employed in assembling and solving the local interpolation problems for computing the coefficients $\boldsymbol{c}_j$, as shown in Table 2. The evaluation time of the global interpolant $\mathcal{I}$ is independent from the number of data-points. In our experiments the number of evaluation points is larger than the number of data-points which results in longer computation times in the interpolation phase.

Note that, as mentioned in the previous sections, the overall algorithm is linear with respect to the number of data-points and evaluation points, which is confirmed in the numerical experiments shown in Figure 5.

We perform strong scaling studies on a cluster with an Intel Xeon E5-2695 at 2.1 GHz processor with 2 x 18 cores where we measure performance as the ratio between the serial experiment and the parallel experiment with different number of cores. The results depicted in Figure 6 show that the speed-up is linear with respect to number of cores and is close to the ideal one.

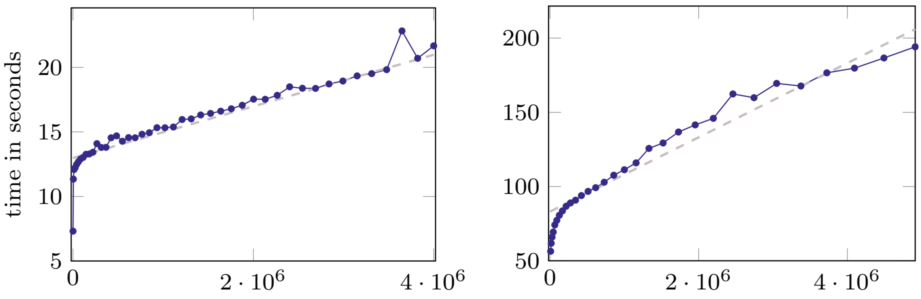
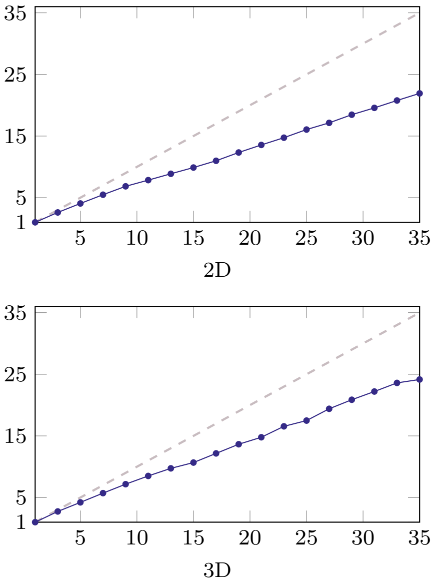


Fig. 5 Linear trend of the total time with respect to the number N of data-points, evaluating the interpolant (4) at 9 million points in 2D and 8 million points in 3D. Results for 2D on the left and for 3D on the right.



#cores	time (sec)		speed-up	
	2D	3D	2D	3D
1	91.65	960.89	–	–
3	35.20	350.81	2.60	2.74
5	22.57	228.03	4.06	4.21
7	16.72	168.19	5.48	5.71
9	13.39	134.11	6.84	7.16
11	11.68	113.01	7.85	8.50
13	10.32	98.71	8.88	9.73
15	9.27	90.02	9.89	10.67
17	8.34	79.02	10.99	12.16
19	7.43	70.41	12.33	13.65
21	6.76	65.02	13.56	14.78
23	6.22	58.05	14.74	16.55
25	5.71	54.95	16.06	17.49
27	5.35	49.53	17.14	19.40
29	4.96	46.05	18.47	20.87
31	4.68	43.28	19.57	22.20
33	4.41	40.71	20.77	23.60
35	4.18	39.78	21.92	24.15

Fig. 6 Strong scaling experiment for two and three dimensional RBF-PUM interpolation problems using M4 as local RBF function. The x-axis represents the number of threads while the y-axis shows the speed-up and the dashed line illustrates the ideal speed-up. In 2D we use 4 000 000 points and 9 000 000 evaluation points, while in 3D we use 4 913 000 points and 8 000 000 evaluation points.

5 Numerical experiments

In this section we report the performance of our parallel algorithm which is measured through numerical experiments, illustrated in some tables and figures. All these experiments have been carried out on a Macbook Pro laptop with an Intel Core i7 2.3GHz processor and 16GB RAM.

In the following we focus on a wide series of experiments, which usually concern very large interpolation problems in different dimensions and with uniformly distributed data-points. Though our study is generally devoted to solve "standard" 2D and 3D problems on scattered data, *i.e.*, problems where the domain Ω is contained in $\mathbb{R}^2$ or $\mathbb{R}^3$, we consider particular situations in which the domain Ω is the surface of a sphere such as the 2-sphere in $\mathbb{R}^3$ [19,23], and we also comment on experiments in higher dimensions. Note that, for simplicity, we do not report thorough quantitative analysis in higher dimensions ($s > 3$), because from a numerical standpoint the experiments carried out for $s \leq 3$ confirm both the theoretical analysis outlined in Section 2 and the practical study provided in Section 4.

We show the results obtained by applying the RBF-PU method using some of the RBFs contained in Table 1 as local approximants. More precisely, our analysis is based on considering the family of Matérn functions – essentially M2 and M4 – as basis functions in (3), and the compactly supported W2 as localizing function of Shepard's weight in (7). In this way, we apply the RBF-PUM for computing the interpolation errors for various fixed values of ε, as well as using the LOOCV approach locally as described in Section 3. In fact, the target of our study is two-fold: i) analyzing the efficiency expressed in terms of CPU times of the proposed algorithm; ii) verifying accuracy of the RBF-PU method with and without employing the LOOCV based technique for selecting ε.

In the various experiments we thus analyze the performance of the parallel algorithm taking the data values by three test functions. The former two are known in literature as bivariate and trivariate Franke's functions [6,34], whose analytic expressions are

$$\begin{aligned} f_2(x_1,x_2) = {} & \frac{3}{4}\mathrm{e}^{-\frac{(9x_1-2)^2+(9x_2-2)^2}{4}} + \frac{3}{4}\mathrm{e}^{-\frac{(9x_1+1)^2}{49}-\frac{9x_2+1}{10}} \\ & + \frac{1}{2}\mathrm{e}^{-\frac{(9x_1-7)^2+(9x_2-3)^2}{4}} - \frac{1}{5}\mathrm{e}^{-(9x_1-4)^2-(9x_2-7)^2} \end{aligned}$$

and

$$\begin{aligned} f_3(x_1,x_2,x_3) = {} & \frac{3}{4}\mathrm{e}^{-\frac{(9x_1-2)^2+(9x_2-2)^2+(9x_3-2)^2}{4}} + \frac{3}{4}\mathrm{e}^{-\frac{(9x_1+1)^2}{49}-\frac{9x_2+1}{10}-\frac{9x_3+1}{10}} \\ & + \frac{1}{2}\mathrm{e}^{-\frac{(9x_1-7)^2+(9x_2-3)^2+(9x_3-5)^2}{4}} - \frac{1}{5}\mathrm{e}^{-(9x_1-4)^2-(9x_2-7)^2-(9x_3-5)^2}. \end{aligned}$$

The latter is the s-variate function [16]

$$g_s(\boldsymbol{x}) = 4^s \prod_{i=1}^{s} x_i(1-x_i), \qquad \boldsymbol{x} = (x_1,\dots,x_s) \in \Omega.$$

Note that the bivariate and trivariate functions f_2 and f_3 as well as the multivariate function g_s are commonly used in approximation processes to test and validate new methods and algorithms, then making them usable in the field of applications.

In order to investigate accuracy of the interpolation method, we compute the Root Mean Square Error (RMSE), whose formula is given by

$$\mathrm{RMSE} = \sqrt{\frac{1}{N_{eval}} \sum_{i=1}^{N_{eval}} |f(\boldsymbol{y}_i) - \mathcal{I}(\boldsymbol{y}_i)|^2}. \tag{12}$$

Therefore, in the sequel we consider more in detail the two aforementioned cases, *i.e.*, hyper-cubical interpolation in Section 5.1 and 3D interpolation on the 2-sphere in Section 5.2.

5.1 Results for hyper-cubical interpolation

In this section we first show numerical results acquired from tests carried for 2D and 3D interpolation, then we comment results for 4D and 5D problems. As interpolation nodes, we take some sets of N uniformly random Halton data-points contained in the unit hypercube $\Omega = [0,1]^s \subset \mathbb{R}^s$, with $s = 2,\dots,5$, (see Figure 7 left) which are generated by using the Matlab program `haltonseq.m` [16]. This node distribution is a typical example of scattered data-point set. Moreover, the computation of interpolation errors is carried out on a grid of N_{eval} evaluation points.

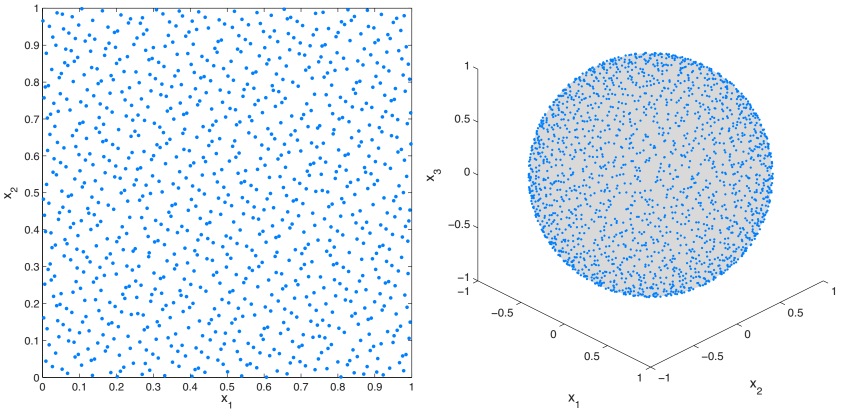


Fig. 7 Example of Halton data-point sets on the plane (left) and on the sphere (right).

In Table 3, we deal with bivariate interpolation taking five sets of Halton points with the number N of interpolation nodes going from 289 to 66 049 for the function f_2. In such experiments we apply the RBF-PUM by locally using the M4 and M2 interpolants. Then, we compute the RMSEs through the use of the LOOCV scheme – denoted by opt – and, as a comparison, for fixed values of $\varepsilon = 10, 15, 20$. In particular, in these tests the cross-validation approach based on an error prediction results on average in an improvement of about an half order of magnitude (compared to the best result found for ε fixed). In both tables we observe a similar behavior of the errors: on the one hand, accuracy is greater for small values of the shape parameter, while it tends to decrease when ε grows; on the other hand, errors decrease as the number N and the RBF smoothness increase. Hence, these tests also confirm from a numerical viewpoint the theoretical results presented in Section 2.2, Theorem 1.

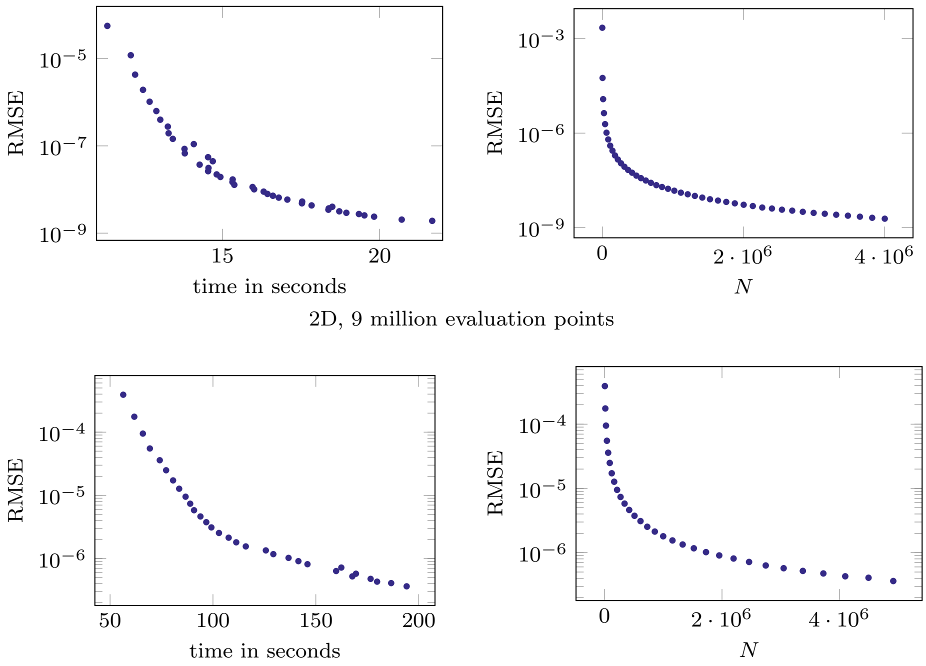
	$\varepsilon \backslash N$	289	1 089	4 225	16 641	66 049
M2	10	1.00E − 2	2.60E − 3	6.01E − 4	1.15E − 4	3.58E − 5
	15	2.05E − 2	5.73E − 3	1.33E − 3	3.23E − 4	7.79E − 5
	20	3.41E − 2	1.01E − 2	2.36E − 3	5.74E − 4	1.38E − 4
	opt	3.02E − 3	6.14E − 4	1.31E − 4	3.20E − 5	7.38E − 6
M4	10	3.40E − 3	4.73E − 4	5.98E − 5	7.70E − 6	9.25E − 7
	15	8.30E − 3	1.36E − 3	1.80E − 4	2.27E − 5	2.83E − 6
	20	1.59E − 2	2.96E − 3	4.09E − 4	5.21E − 5	6.58E − 6
	opt	1.95E − 3	1.75E − 4	2.00E − 5	2.34E − 6	1.97E − 7

Table 3 RMSEs computed on Halton points by applying the RBF-PUM with the M2 and M4 as local interpolant for f_2. Interpolation errors are obtained by using the LOOCV scheme (opt) and, as a comparison, for various values of ε. The interpolant (4) is evaluated at 90 thousand points.

$\varepsilon\backslash N$	4 913	35 937	274 625	2 146 689	16 974 593
10	6.68E − 4	6.93E − 5	7.03E − 6	7.98E − 7	9.38E − 8
15	1.52E − 3	1.76E − 4	1.87E − 5	2.12E − 6	2.47E − 7
20	2.97E − 3	3.81E − 4	4.19E − 5	4.77E − 6	5.53E − 7
opt	3.02E − 4	2.99E − 5	2.83E − 6	3.36E − 7	−−

Table 4 RMSEs computed on Halton points by applying the RBF-PUM with the M4 as local interpolant for f_3. Interpolation errors are obtained by using the LOOCV scheme (opt) and, as a comparison, for various values of ε. The interpolant (4) is evaluated at 9 million points.

In Table 4 we instead focus on trivariate interpolation considering other five sets of Halton points, where N is now included in the range between 4 913 and 16 974 593. The experiments for interpolating the function f_3 follow the same guidelines outlined earlier. In particular, we test our OpenCL algorithm with the M4 as local RBF interpolant by varying the shape parameter, *i.e.*, taking $\varepsilon = 10, 15, 20$, and also applying the LOOCV approach to find an optimal value of ε for each PU sub-domain. From this analysis we can extend our previous remarks also to 3D interpolation, thus drawing similar conclusions. Note that we omit error computation with LOOCV in the case of more than 16 million points because – even if computable – it is quite expensive from a computational time viewpoint.



2D, 9 million evaluation points

3D, 8 million evaluation points

Fig. 8 Interpolation error against total computational time (left) and data-point number (right) computed on 2D Halton points by using the RBF-PUM with the M4 as local interpolant for f_2. Errors and times are obtained by taking $\varepsilon = 10$ for different evaluation points.

		RMSEs		CPU times			
s	N	OpenCL	opt	OpenCL	Matlab	opt	speed-up
	9 216	2.63E − 5	3.78E − 6	3.05	45.05	3.73	14.8
2	250 000	1.50E − 7	1.53E − 7	3.58	240.46	22.39	67.2
	1 000 000	1.93E − 8	1.36E − 9	4.59	879.74	73.20	191.7
	19 683	3.94E − 4	8.18E − 5	23.70	197.63	40.98	8.3
3	110 592	6.56E − 5	1.08E − 5	26.65	277.47	121.41	10.4
	884 736	7.43E − 6	1.11E − 6	42.13	715.67	1030.69	17.0

Table 5 RMSEs and CPU times (in seconds) computed on Halton points by using the RBF-PUM with the M4 as local interpolant for g_s, $s = 2, 3$. Errors and times are obtained by using the LOOCV scheme (opt) and taking $\varepsilon = 10$; in the latter case, the speed-up between our OpenCL algorithm and Matlab implementation [6] is given. The 2D and 3D interpolants (4) are evaluated at 2 250 000 and 3 375 000 points, respectively.

Moreover, a more extensive and detailed analysis on the behavior of computational error against computational time and interpolation data-point number is shown in Figure 8 for 2D and 3D, left to right, respectively. From such plots it is evident how fast the RMSEs decrease when increasing the number of points interpolated, although the CPU times do not suffer from a significant growth as N becomes very large (for instance, $N > 10^6$). These pictures refer to numerical experiments shown in Figure 5, and therefore complete the study started in Section 4.3.

Finally, in Table 5 we further test our parallel algorithm with the multivariate test function g_s, for $s = 2, 3$. In both dimensions, we report the results computed with the M4 and for three sets of Halton data-points. Specifically, here we show RMSEs and CPU times obtained via LOOCV and fixing $\varepsilon = 10$. In the latter case, errors and times are labelled by OpenCL, so as to emphasize the use of our new code. In fact, we also compare our algorithm with the only – at the best of our knowledge – Matlab software [6] so far implemented for RBF-PUM interpolation in $\mathbb{R}^s$, for any $s \geq 2$. We note that this implementation uses a different algorithm, in particular the search for the cells is performed trough a kd-tree which has run-time complexity of $O(n \log n)$. In particular, we highlight the great speed-up between the two algorithms, furthermore underlining as this gap tends to be more and more remarkable when the number of interpolation nodes increases. Note that, despite the fact that the Matlab code is not parallel, Matlab parallelizes many parts thus exploiting the four cores of our machine.

For higher dimensions the results are similar. In fact, we performed two experiments in 4D for 531 441 and 5 308 416 data-points evaluated at 12 960 000 points, which took 12.7 and 36.2 minutes (Matlab took 69.9 minutes and about 5 hours) with RMSE 1.91E − 4 and 3.04E − 5 respectively. In 5D, we considered an experiment with 1 048 576 data-points and 14 348 907 evaluation points, which lasted for 148 minutes with a RMSE of 6.31E − 4.

	$\varepsilon \backslash N$	16 641	66 049	263 169	1 050 625
M2	5	$2.83\mathrm{E}-5$	$5.76\mathrm{E}-6$	$1.00\mathrm{E}-6$	$1.88\mathrm{E}-7$
	10	$6.75\mathrm{E}-5$	$1.32\mathrm{E}-5$	$2.23\mathrm{E}-6$	$4.21\mathrm{E}-7$
	15	$1.43\mathrm{E}-4$	$2.70\mathrm{E}-5$	$4.48\mathrm{E}-6$	$8.46\mathrm{E}-7$
	20	$2.56\mathrm{E}-4$	$4.76\mathrm{E}-5$	$7.79\mathrm{E}-6$	$1.47\mathrm{E}-6$
	25	$4.10\mathrm{E}-4$	$7.54\mathrm{E}-5$	$1.22\mathrm{E}-5$	$2.29\mathrm{E}-6$
	30	$6.04\mathrm{E}-4$	$1.11\mathrm{E}-4$	$1.79\mathrm{E}-5$	$3.33\mathrm{E}-6$
M4	5	$2.05\mathrm{E}-6$	$2.42\mathrm{E}-7$	$1.76\mathrm{E}-8$	$1.20\mathrm{E}-5$
	10	$5.40\mathrm{E}-6$	$6.77\mathrm{E}-7$	$4.84\mathrm{E}-8$	$7.37\mathrm{E}-9$
	15	$1.51\mathrm{E}-5$	$1.88\mathrm{E}-6$	$1.27\mathrm{E}-7$	$1.57\mathrm{E}-8$
	20	$3.46\mathrm{E}-5$	$4.08\mathrm{E}-6$	$2.81\mathrm{E}-7$	$3.10\mathrm{E}-8$
	25	$6.68\mathrm{E}-5$	$7.55\mathrm{E}-6$	$5.37\mathrm{E}-7$	$5.59\mathrm{E}-8$
	30	$1.15\mathrm{E}-4$	$1.26\mathrm{E}-5$	$9.26\mathrm{E}-7$	$9.33\mathrm{E}-8$

Table 6 RMSEs computed on spherical Halton points by applying the RBF-PUM with the local M2 and M4 interpolant for f_3. Interpolation errors are obtained for various values of ε. The interpolant (4) is evaluated at 1 million points.

5.2 Results for spherical interpolation

In this section we present some of the numerical experiments we made to test our parallel algorithm for data-point interpolation on the sphere. For this purpose, as interpolation nodes we consider some sets of N Halton data-points [44], which are uniformly random distributed on the unit sphere $\mathbb{S}^2 \subset \mathbb{R}^3$ (see Figure 7 right). Instead, interpolation errors are evaluated on a nearly uniform distribution of N_{eval} spiral points [37], which uniformly fill up the sphere by tracing out an imaginary spiral from the south pole to the north pole.

To test the OpenCL algorithm for spherical interpolation, we take four sets of Halton points with N varying from 16 641 up to 1 050 625 for f_3. In Table 6, we apply the RBF-PUM by using local M2 and M4 interpolants, respectively. In particular, we report the errors committed for various values of ε fixed, *i.e.*, $\varepsilon = 5, 10, 15, 20, 25, 30$. Analyzing the results, we can observe good RMSEs in both cases, even if as expected the M4 turns out to be roughly more accurate (from one to two orders of magnitude) than the M2. Moreover, though the errors gradually increase as ε grows, for $N = 1\,050\,625$ and $\varepsilon = 5$ we point out a sudden loss of accuracy due to ill-conditioning of some of the local interpolation matrices in (6).

6 Application

In this section we consider an application to Earth's topography, which consists of interpolating with our OpenCL algorithm a set of real world data, *i.e.* the Maunga Whau volcano data-points, available in [32]. The data represents 5307 elevation measurements obtained from Maunga Whau (Mt. Eden) in Auckland, New Zealand, sampled on a grid made by 10 m × 10 m cells.

$\varepsilon\backslash$ RBF	M2	M4	M6
10	0.73	0.83	1.14
15	0.84	0.83	1.12
20	1.07	0.84	1.09
opt	0.73	0.83	1.09

Table 7 RMSEs in meters computed on 5 200 Maunga Whau volcano points by using the RBF-PUM with M2, M4, M6 as local interpolants. Interpolation errors are obtained via LOOCV scheme (opt) and for various values of ε. The interpolant (4) is evaluated at 107 points.

From a geographical viewpoint, Maunga Whau is a scoria cone in the Mount Eden suburb of Auckland. It is a dormant volcano whose summit is at 196 meters above sea level, and is one the most prominent volcanic cones remaining in the Auckland region. Erupting more than 10 thousand years ago from three overlapping scoria cones, it formed a huge scoria mound with a central crater from the last eruption. Lava flowed out from the base of the mound, and in some places the lava is more than 60 meters thick.

After this brief historical and geographical footnote, we come back now to our interpolation/application purpose. Hence, from [32] we have 5 307 volcano data-points, which we subdivide into two subsets: first, we randomly select $N = 5\,200$ nodes for the interpolation process; second, we reserve the remaining $N_{eval} = 107$ evaluation points for the cross-validation. The latter technique is commonly used in applications to assess reliability of approximation results and, accordingly, to verify the accuracy provided by the RBF-PUM, by comparing the predicted values with the original ones. In particular, in Table 7 we report the RMSEs in meters computed by using M2, M4 and M6 (as local RBF interpolants) and varying the value of the shape parameter ε, *i.e.* for $\varepsilon = 10, 15, 20$. These interpolation errors are also compared with ones obtained by applying the LOOCV approach. Note that, although such errors are larger than the ones shown in Section 5, they turn out to be consistent with the previous results; in fact, in this real case, the error in (12) is computed in absolute value and measured in meters.

From this study we can observe that the M2 provides in general the best results, while it seems that an increase of RBF smoothness leads to a gradual accuracy decrease. Moreover, the use of a cross-validation approach for selecting suitable values of ε confirms the results obtained for ε fixed, thus showing our algorithm turns to be effective also in real-life applications.

7 Conclusions and future work

In this paper we proposed a new parallel algorithm for multivariate interpolation of scattered data-points via the RBF-PUM. Exploiting the concurrent nature of the PUM and its applicability in high space dimensions, we efficiently implemented an OpenCL algorithm that makes use of shared-memory parallel processors. This strategy allowed us to reach linear computational time complexity with respect to the number of data-points. Additionally, we combined the RBF-PUM with a LOOCV based technique that effectively finds optimal values of the RBF shape

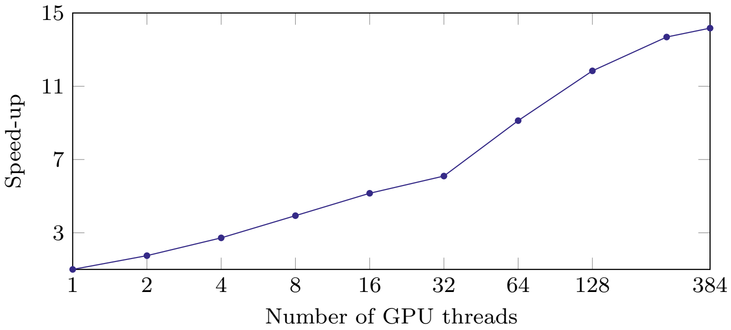


Fig. 9 GPU based two-dimensional experiment with 4 225 data-points and 5 000 evaluation points performed on a Macbook Pro laptop with a GeForce GT 650M. The base experiment with one thread lasts for 10.9 seconds.

parameters. The latter also provided a way to reduce the global interpolation error as illustrated in our numerical experiments. Finally, we applied our OpenCL algorithm to real-life data.

While our algorithm is tailored for CPU-based computations, we plan to extend it for GPUs. As explained, our algorithm requires dedicated memory for each thread for storing the local interpolation system (6). For CPUs this is not a problem since usually the number of cores is small (typically less than 100) and the memory large. GPUs usually have smaller memory and many computational units. To overcome this limitation we require work-group synchronization and proper handling of memory hierarchies. This problem dramatically affects scaling performance as shown in Figure 9. In fact, when exploiting the full resources of the GPU, the measured performance is similar to the CPU runs.

As future work we propose a new algorithm that allows us to consider subdomains with variable shape and size. This approach turns out to be particularly meaningful when strongly non-uniform data-points are considered as in [3]. Moreover, further extensions could concern the parallel implementation of new schemes for Hermite-Birkhoff interpolation [12] and based on rescaled RBFs [13].

For large number of data-points the required amount of memory becomes very large and it can exceed the capacity of the physical memory of one machine. To overcome this limitation, standards such as MPI allow exploiting most modern super-computers which have dedicated memory for each computational node. This means that the available memory grows with the number of computational units used, therefore enabling the treatment of very large data-sets. We envision to adapt our shared-memory algorithm to exploit these parallel computing architecture.

The OpenCL kernels described in this paper, a Python script, and a C++ code can be found at `https://bitbucket.org/zulianp/opencl-rbf-pum`.

Acknowledgements The first author acknowledges financial support from the GNCS-INdAM and sincerely thanks professor Kai Hormann for the invitation to visit the USI in Lugano. This work was supported by the SNF under project number 200020_156178. The method presented in this paper is implemented within the MOONoLith software library. Finally, we are grateful to the Swiss national supercomputing centre for providing us with the computational resources and the OpenCL SDK on their system for the scaling experiments.

References

1. Anderson, R.J., Snyder, L.: A comparison of shared and nonshared memory models of parallel computation. Proceedings of the IEEE **79**(4), 480–487 (1991). DOI 10.1109/5.92042
2. Babuška, I., Melenk, J.M.: The partition of unity method. Internat. J. Numer. Methods Engrg. **40**, 727–758 (1997)
3. Bozzini, M., Lenarduzzi, L., Rossini, M.: Polyharmonic splines: An approximation method for noisy scattered data of extra-large size. Appl. Math. Comput. **216**, 317–331 (2010)
4. Brent, R.P.: Algorithms for minimization without derivatives. Courier Corporation (2013)
5. Buhmann, M.D.: Radial Basis Functions: Theory and Implementation, *Cambridge Monographs on Applied and Computational Mathematics*, vol. 12. Cambridge University Press, Cambridge (2003)
6. Cavoretto, R.: A numerical algorithm for multidimensional modeling of scattered data points. Comput. Appl. Math. **34**, 65–80 (2015)
7. Cavoretto, R., De Rossi, A.: A trivariate interpolation algorithm using a cube-partition searching procedure. SIAM J. Sci. Comput. **37**, A1891–A1908 (2015)
8. Cavoretto, R., De Rossi, A., Perracchione, E.: Efficient computation of partition of unity interpolants through a block-based searching technique. Comput. Math. Appl. **71**, 2568–2584 (2016)
9. Cavoretto, R., De Rossi, A., Perracchione, E., Venturino, E.: Robust approximation algorithms for the detection of attraction basins in dynamical systems. J. Sci. Comput. **68**, 395–415 (2016)
10. Cavoretto, R., Fasshauer, G.E., McCourt, M.: An introduction to the Hilbert-Schmidt SVD using iterated Brownian bridge kernels. Numer. Algorithms **68**, 393–422 (2015)
11. De Marchi, S., Santin, G.: Fast computation of orthonormal basis for rbf spaces through krylov space methods. BIT **55**, 949–966 (2015)
12. Dell'Accio, F., Di Tommaso, F.: Complete Hermite-Birkhoff interpolation on scattered data by combined shepard operators. J. Comput. Appl. Math. **300**, 192–206 (2016)
13. Deparis, S., Forti, D., Quarteroni, A.: A rescaled localized radial basis function interpolation on non-cartesian and nonconforming grids. SIAM J. Sci. Comput. **36**, A2745–A2762 (2014)
14. Ericson, C.: Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3D Technology). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2004)
15. Fasshauer, G., McCourt, M.: Kernel-based Approximation Methods using Matlab, *Interdisciplinary Mathematical Sciences*, vol. 19. World Scientific, Singapore (2015)
16. Fasshauer, G.E.: Meshfree Approximation Methods with Matlab, *Interdisciplinary Mathematical Sciences*, vol. 6. World Scientific, Singapore (2007)
17. Fasshauer, G.E.: Positive definite kernels: Past, present and future. Dolomites Res. Notes Approx. **4**, 21–63 (2011)
18. Fernando, R.: GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics. Pearson Higher Education (2004)
19. Fornberg, B., Flyer, N.: A Primer on Radial Basis Functions with Applications to the Geosciences. SIAM, Philadelphia (2015)
20. Fornberg, B., Larsson, E., Flyer, N.: Stable computations with gaussian radial basis functions. SIAM J. Sci. Comput. **33**, 869–892 (2011)
21. Forum, M.P.I.: MPI: A Message-Passing Interface Standard Version 3.0 (2012). Chapter author for Collective Communication, Process Topologies, and One Sided Communications
22. Heryudono, A., Larsson, E., Ramage, A., von Sydow, L.: Preconditioning for radial basis function partition of unity methods. J. Sci. Comput. **67**, 1089–1109 (2016)
23. Hubbert, S., Le Gia, Q., Morton, T.: Spherical Radial Basis Functions, Theory and Applications. SpringerBriefs in Mathematics. Springer, London (2015)
24. Krause, R., Zulian, P.: A parallel approach to the variational transfer of discrete fields between arbitrarily distributed finite element meshes. SIAM J. Sci. Comput. **38**, C307–C333 (2016)
25. Melenk, J.M., Babuška, I.: The partition of unity finite element method: Basic theory and applications. Comput. Methods. Appl. Mech. Engrg. **139**, 289–314 (1996)
26. Nichols, B., Buttlar, D., Farrell, J.P.: Pthreads Programming. O'Reilly & Associates, Inc., Sebastopol, CA, USA (1996)
27. Nickolls, J., Buck, I., Garland, M., Skadron, K.: Scalable parallel programming with CUDA. Queue **6**(2), 40–53 (2008)

28. Novak, E., Woźniakowski, H.: Tractability of Multivariate Problems Volume 1: Linear Information. No. 6 in EMS Tracts in Mathematics. European Mathematical Society (2008)
29. NVIDIA: CUDA Samples (2013)
30. OpenMP Architecture Review Board: OpenMP application program interface version 3.0 (2008). URL `http://www.openmp.org/mp-documents/spec30.pdf`
31. Pazouki, M., Schaback, R.: Bases for kernel-based spaces. J. Comput. Appl. Math. **236**, 575–588 (2011)
32. R Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2014)
33. Rashidinia, J., Fasshauer, G.E., Khasi, M.: A stable method for the evaluation of gaussian radial basis function solutions of interpolation and collocation problems. Comput. Math. Appl. **72**, 178–193 (2016)
34. Renka, R.: Multivariate interpolation of large sets of scattered data. ACM Trans. Math. Softw. **14**, 139–148 (1988)
35. Rippa, S.: An algorithm for selecting a good value for the parameter c in radial basis function interpolation. Adv. Comput. Math. **11**, 193–210 (1999)
36. Safdari-Vaighani, A., Heryudono, A., Larsson, E.: A radial basis function partition of unity collocation method for convection-diffusion equations arising in financial applications. J. Sci. Comput. **64**, 341–367 (2015)
37. Saff, E., Kuijlaars, A.: Distributing many points on a sphere. Math. Intelligencer **19**, 5–11 (1997)
38. Schling, B.: The Boost C++ Libraries. XML Press (2011)
39. Shcherbakov, V., Larsson, E.: Radial basis function partition of unity methods for pricing vanilla basket options. Comput. Appl. Math. **71**, 185–200 (2016)
40. Speck, R., Gibbon, P., Hofmann, M.: Efficiency and scalability of the parallel Barnes-Hut tree code PEPC. In: B. Chapman, F. Desprez, G.R. Joubert, A. Lichnewsky, F.J. Peters, T. Priol (eds.) Parallel Computing: From Multicores and GPU's to Petascale, *Advances in Parallel Computing*, vol. 19, pp. 35–42. IOS Press (2010)
41. Stone, J.E., Gohara, D., Shi, G.: Opencl: A parallel programming standard for heterogeneous computing systems. IEEE Des. Test **12**(3), 66–73 (2010). DOI 10.1109/MCSE.2010.69. URL `http://dx.doi.org/10.1109/MCSE.2010.69`
42. Wendland, H.: Fast evaluation of radial basis functions: methods based on partition of unity. In: C.K. Chui, L.L. Schumaker, J. Stöckler (eds.) Approximation Theory X: Wavelets, Splines, and Applications, pp. 473–483. Vanderbilt University Press (2002)
43. Wendland, H.: Scattered Data Approximation, *Cambridge Monographs on Applied and Computational Mathematics*, vol. 17. Cambridge University Press, Cambridge (2005)
44. Wong, R., Luk, W., Heng, P.: Sampling with Hammersley and Halton points. J. Graphics Tools **2**, 9–24 (1997)