FEREBUS: Highly Parallelized Engine for Kriging Training

Nicodemo Di Pasquale,[a] Michael Bane,[b] Stuart J. Davie,[a] and Paul L. A. Popelier*[a]

FFLUX is a novel force field based on quantum topological atoms, combining multipolar electrostatics with IQA intraatomic and interatomic energy terms. The program FEREBUS calculates the hyperparameters of models produced by the machine learning method kriging. Calculation of kriging hyperparameters ($\theta$ and $p$) requires the optimization of the concentrated log-likelihood $L(\theta, p)$. FEREBUS uses Particle Swarm Optimization (PSO) and Differential Evolution (DE) algorithms to find the maximum of $L(\theta, p)$. PSO and DE are two heuristic algorithms that each use a set of particles or vectors to explore the space in which $L(\theta, p)$ is defined, searching for the maximum. The log-likelihood is a computationally expensive function, which needs to be calculated several times during each optimization iteration. The cost scales quickly with the problem dimension and speed becomes critical in model generation. We present the strategy used to parallelize FEREBUS, and the optimization of $L(\theta, p)$ through PSO and DE. The code is parallelized in two ways. MPI parallelization distributes the particles or vectors among the different processes, whereas the OpenMP implementation takes care of the calculation of $L(\theta, p)$, which involves the calculation and inversion of a particular matrix, whose size increases quickly with the dimension of the problem. The run time shows a speed-up of 61 times going from single core to 90 cores with a saving, in one case, of $\approx$98% of the single core time. In fact, the parallelization scheme presented reduces computational time from 2871 s for a single core calculation, to 41 s for 90 cores calculation. © 2016 The Authors. Journal of Computational Chemistry Published by Wiley Periodicals, Inc.

DOI: 10.1002/jcc.24486

Introduction

Computer simulations of molecular systems are becoming essential tools in several scientific fields, from biology to engineering. Molecular Dynamics (MD) simulations describe the evolution of a group of atoms, and the accurate representation of the interaction energies between the different atoms represents a very challenging task. The exact description of interatomic interactions requires the solution of the Schrödinger equation. However, ab initio calculations are suitable only for small systems or short time scales, and larger systems or longer time scales can only be approximated through the use of a force field. Unfortunately, most force fields are not able to correctly reproduce a number of critical properties, in particular for biological molecules (hydrogen bonding, structure, $\pi$ stacking, amino acid pK$_a$ shifts$^{11-4}$). In recent years, several approaches have been proposed to overcome problems related to the poor description of classical force fields, by using machine learning to capture and then interpolate precomputed quantum mechanical data. Generally speaking, machine learning techniques are a set of different models that, starting from a set of observed data (known as training points), allow one to obtain a model that is able to both describe those training points and predict the value of the property for a previously unseen data point. Machine learning techniques are becoming increasingly popular in the computational chemistry community as a tool for obtaining a more realistic description of interactions than that obtained from classical force fields. For example, there is a rich literature$^{[5]}$ on the use of neural networks to construct potentials for a wide range of systems, from crystalline silicon$^{[6]}$ to the water dimer.$^{[7]}$ Genetic algorithms have also been used in connection with the fitting$^{[8]}$ of potential energy surfaces, for example, in the fitting$^{[9]}$ of the parameters of a Tersoff potential. Finally, Gaussian Approximation potentials have also been designed,$^{[10]}$ which are systematically improvable with more DFT data, successfully predict solid state properties, and were also applied to molecular and condensed water.$^{[11]}$

In this work, we consider kriging, a prediction method originally developed for geostatistical applications$^{[12]}$ and later applied to computer experiments by Sacks et al.$^{[13]}$ Kriging predictions are based on the intuitive notion that physical properties appear in space as smooth functions, and the value of a property in a point of space is highly correlated with neighboring points. In our framework, we obtain atomistic properties through expensive ab initio calculations for a certain number of configurations of an atomistic system. These properties are then used to build or train a kriging model, to enable the prediction of the same atomic properties but now for unseen data points. However, training of the kriging model requires the optimization, with respect to the hyperparameters within the model, of the so-called concentrated log-likelihood. This function is denoted $L(\theta, p)$ and is computationally expensive.

[a] N. Di Pasquale, S. J. Davie, P. L. A. Popelier
Manchester Institute of Biotechnology (MIB), 131 Princess Street, Manchester M1 7DN, Great Britain and School of Chemistry, University of Manchester, Oxford Road, Manchester M13 9PL, Great Britain
E-mail: paul.popelier@manchester.ac.uk

[b] M. Bane
Research IT, The University of Manchester and High End Compute, http://highendcompute.co.uk, Manchester, M13 0EL
© 2016 The Authors. Journal of Computational Chemistry Published by Wiley Periodicals, Inc.

This is an open access article under the terms of the Creative Commons Attribution License, which permits use, distribution and reproduction in any medium, provided the original work is properly cited.
Previous work from our group\textsuperscript{[14]} described how the optimization of the concentrated log-likelihood can be obtained through two heuristic algorithms, namely Particle Swarm Optimization (PSO)\textsuperscript{[15]} and Differential Evolution (DE)\textsuperscript{[16]} Both PSO and DE allow one to obtain the maximum of $L(\theta, p)$ in a reasonable number of iterations, even for highly multidimensional problems. However, the fact that $L(\theta, p)$ needs to be calculated several times at each iteration of the optimization is a serious problem to the speed of generation of the kriging model. For example, for a ten-dimensional problem, $L(\theta, p)$ needs to be calculated about 16 times with PSO and 100 times with DE, for each iteration. Specifically, the value of $L(\theta, p)$ is obtained through the calculation and inversion of a particular matrix, the $\mathbf{R}$ matrix, which is defined in “Kriging” section. The computational burden of inverting the $\mathbf{R}$ matrix is related to the number of training points considered and to the dimension of the problem. This inversion can easily become a huge task, as will be described in “Model Summary” section. Inversion of a matrix is usually a computationally expensive task and should be avoided if possible. However, in our case, the quantity of interest is the inverse of the $\mathbf{R}$ matrix, which needs to be explicitly computed in order to calculate $L(\hat{\theta}, \hat{p})$ and to obtain predictions with kriging.

The natural choice to reduce computation time is to resort to techniques of high performance computing. The particular problems we are addressing allow us to easily consider two well-known parallel paradigms, namely the Message Passing Interface (MPI)\textsuperscript{[17]} and Open Multi-Processing (OpenMP).\textsuperscript{[18]} In particular, we design an algorithm that is able to use both paradigms at the same time, in a hybrid parallel program.

High Performance Computing for MD simulations has become increasingly important over the last several years. Many attempts to present fast, efficient and reliable parallelized algorithms are reported for standard hardware architectures (CPUs)\textsuperscript{[19]} and also Graphic Process Units (GPU).\textsuperscript{[20]} The training of the kriging models is unrelated to the actual MD simulations, but fast and reliable software that allows one to quickly obtain (and therefore test) kriging models, without any limitations on the dimensionality of the training set or degrees of freedom, is essential in the generation of this new class of force fields.

The paper is structured as follows: in the first part we give a brief description of the problem at hand, and explain the concentrated log-likelihood $L(\hat{\theta}, \hat{p})$, as well as the PSO and DE algorithms. Then we describe the parallelization strategy that we employ in the in-house kriging program FEREBUS, and how MPI and OpenMP are embedded together. Finally, we demonstrate the performance results of FEREBUS and present our conclusions.

**Model Summary**

**Kriging**

Kriging represents a spatial regression technique that allows the prediction of properties of interest (e.g., atomic multipole moments). First used in geostatistics, Sacks et al\textsuperscript{[13]} later extended it to deterministic computer experiments. Under certain hypotheses, kriging represents the Best Linear Unbiased Predictor,\textsuperscript{[13]} and its use is based on the intuitive fact that physical quantities are correlated in space. In other words, two physical quantities located near each other are more likely to possess similar values than two quantities that are more separated. Here, we will briefly outline the main features of the kriging method, although more details can be found elsewhere.\textsuperscript{[13,14,21,22]}

Kriging predictions are obtained by finding the values of hyperparameters $\mathbf{p}$ and $\theta$ necessary to maximize the so-called “concentrated log-likelihood” $L(\hat{\theta}, \hat{p})$,

$$L(\hat{\theta}, \hat{p}) = -\frac{N_t}{2} \ln(\hat{\sigma}^2) - \frac{1}{2} \ln(|\mathbf{R}|)$$

where $\mathbf{R}$ is the correlation matrix, $|\mathbf{R}|$ its determinant, and $\hat{\sigma}^2$ is the variance of the kriging process estimated from the available data. By maximizing the concentrated log-likelihood we find the values of $\mathbf{p}$ and $\theta$ that maximize the probability of obtaining the true distribution of the data, given the input data available.

The matrix $\mathbf{R}$ is a $N_t \times N_t$ matrix, with $N_t$ is the number of training points, with elements defined as

$$R_{ij} = \exp \left( -\sum_{k=1}^{N_t} \theta_k |x_{ik} - x_{jk}|^p \right)$$

where $R_{ij}$ represents the correlation between two training points $i$ and $j$, and $N_t$ is the dimension of input (i.e., feature) space. The correlation function represented in eq. (2) is called the power exponential correlation function. When $p_x=2$ for every $k$ it will be called Gaussian correlation function.\textsuperscript{[23]}

The variance $\hat{\sigma}^2$ estimated from the data is defined as:

$$\hat{\sigma}^2 = \frac{(\mathbf{y} - \mathbf{1}\mu)^T \mathbf{R}^{-1} (\mathbf{y} - \mathbf{1}\mu)}{N_t}$$

where $\mathbf{y}$ is the column vector of the property evaluated at each of the $N_t$ training points, $\mathbf{1}$ is a column vector of ones, $(\cdot)^T$ is the transpose of the argument $\cdot$, while $\mu$ is the mean of the kriging process estimated from the data and is given by

$$\mu = \frac{\mathbf{1}^T \mathbf{R}^{-1} \mathbf{y}}{\mathbf{1}^T \mathbf{R}^{-1} \mathbf{1}}$$

When performed through an iterative algorithm, the maximization of the concentrated log-likelihood can be a formidable problem. The concentrated log-likelihood is a high-dimensional function, the calculation of which involves several matrix multiplications with a dimensionality dependent on the number of training points used in the model. Typical values for the $\mathbf{R}$ matrix are of the order of $10^6$ elements.\textsuperscript{[14]}

Two heuristic algorithms, namely PSO\textsuperscript{[14,24]} and DE\textsuperscript{[14]} were proposed to attack the problem of the maximization of the concentrated log-likelihood. In the next subsection, we will outline the principal features of both algorithms giving
references to more detailed analysis, in order to focus on the actual implementation of the parallelization strategy considered here.

Particle swarm optimization

PSO is a search algorithm, the main feature of which is to mimic the behavior of a swarm that converges towards the maximum of a given function. The swarm is composed of $S$ particles each of which represents a point in $\theta$-p space marked by a position vector $x_i$. At each iteration, the position of every particle is updated by a displacement $v_i$ called the velocity in the PSO community:

$$v_{t+1,i} = \omega v_{t,i} + c_1 r_{1,i}(b_{t,i} - x_{t,i}) + c_2 r_{2,i}(g_{t,i} - x_{t,i})$$

$$x_{t+1,i} = x_{t,i} + v_{t+1,i}$$

(5)

where $r_{1,i}$ and $r_{2,i}$ represent a vector of random real numbers, uniformly drawn from the interval (0,1). The dependence of $r_{1,i}$ and $r_{2,i}$ on iteration step $t$, indicates that a new set of random numbers is generated at every iteration. The symbol “$\omega$" represents the Hada-

mard product, $\omega$ is the inertia weight, while $c_1$ and $c_2$ are respectively the cognitive learning factor and the social learning factor (also known as acceleration coefficients). A discussion on the value of the inertia weight, and the cognitive and learning factors can be found in ref. [25,26]. In FEREBUS, $c_1$, $c_2$, and $\omega$ are user-defined in the input file. For the current work we adopted the literature-recom-

mended values of $\omega=0.729$ and $c_1=c_2=1.494$.

The size of the swarm, $S$, can be user-defined in the input file, or can be calculated [26] from the dimension of the problem through eq. (6),

$$S=10+2\sqrt{N_f}$$

(6)

Differential evolution

The Differential Evolution (DE) algorithm was first introduced by Storn and Price [27]. DE is an evolutionary algorithm in which, at every generation, each “parent” vector generates an “offspring.” The choice of allowing the parent or offspring to survive is based on the value of the concentrated log-likelihood obtained following a greedy criterion, i.e., if the value of the concentrated log-likelihood is higher for the new generation then the old one is discarded, otherwise the old generation is taken and the new one discarded. The structure of the DE algorithm follows four steps:

- **Mutation**: an offspring vector is created by randomly combining the difference between parent vectors. In the literature, several mutation strategies have been reported, based on how parent vectors are combined to generate offspring. FEREBUS implements five different mutation strategies to choose from in the input file.[16,28] These choices are

  0. **DE/best/1**:

  $$v_{G+1,i}=x_{G,best}+F(x_{G,a1}-x_{G,a2})$$

  (7)

  1. **DE/current-to-best/2**:

  $$v_{G+1,i}=x_{G,best}+F(x_{G,a1}-x_{G,a2})$$

  (8)

  2. **DE/rand/1**:

  $$v_{G+1,i}=x_{G,a1}+F(x_{G,a1}-x_{G,a2})$$

  (9)

  3. **DE/best/2**:

  $$v_{G+1,i}=x_{G,best}+F(x_{G,a1}-x_{G,a2}+x_{G,a3}-x_{G,a4})$$

  (10)

  4. **DE/rand/2**:

  $$v_{G+1,i}=x_{G,a5}+F(x_{G,a1}-x_{G,a2}+x_{G,a3}-x_{G,a4})$$

  (11)

where $x_{G,i}$ is the G-th generation of the i-th vector in the population; $a_1, a_2, a_3, a_4, a_5$ are the integer indices of the vectors of population chosen randomly over the interval $[1, N_f]$ with $a_i \neq a_j$, if $i \neq j$; $V$ is the number of vectors in the population; $F \in [0,2]$ is a parameter; and $x_{G,best}$ is the vector in the population at generation G with the highest value of the fitness function. The population size is given by,

$$V=AN_f$$

(12)

where A is a user-defined parameter. The default value in FEREBUS is $A=1$.[29] In the following, the five mutation strategies (MS) will be designated as MS0, MS1, MS2, MS3 and MS4.

- **Crossover**: The mutated vector generated in the previous step is crossed with itself prior the mutation (i.e. the original vector):

  \[
  u_{G+1,i,j} = \begin{cases} 
  v_{G+1,i,j} & \text{if} \text{rand} \leq CR \text{ or } j = l_{rand} \\
  x_{G,i,j} & \text{if} \text{rand} > CR \text{ and } j \neq l_{rand} 
  \end{cases}
  \]

  (13)

where rand $\in [0,1]$ is a random number drawn from the uniform distribution between zero and one, $CR \in [0,1]$ is the cross-over constant and $l_{rand} \in [1,N_f]$ is chosen randomly to allow that at least one element of the mutated vector $v_{G+1,i}$ enters in the new vector $u_{G+1,i}$ to avoid that population is not altered (i.e. to ensure $u_{G+1,i} \neq x_{G,i}$).

- **Selection**: FEREBUS then calculates the value of the concentrated log-likelihood in the point $u_{G+1,i}$, and applies the greedy criterion:

  $$x_{G+1,i} = \begin{cases} 
  u_{G+1,i} & \text{if } \mathcal{L}(u_{G+1,i}) > \mathcal{L}(x_{G,i}) \\
  x_{G,i} & \text{otherwise} 
  \end{cases}$$

  (14)

The parameters $F$ and $CR$ in FEREBUS are not user-defined but they are calculated through the self-adapting parameter control proposed by Brest et al.[30] shown in ref. [14].
L-BFGS-B

FEREBUS also implements the calculation of the derivative of the concentrated log-likelihood, and uses it for the L-BFGS-B algorithm. We reported\textsuperscript{[14]} the explicit form of this derivative previously. The L-BFGS-B is implemented through a library\textsuperscript{[11]} and it is not parallelized. The calculation of the derivative of the concentrated log-likelihood involves the building of the derivative of the \( R \) matrix, a process that is parallelized by OpenMP, which will be described for the construction of the “standard” \( R \) matrix described below. In our previous work,\textsuperscript{[14]} we showed how PSO and DE can be safely employed to optimize the concentrated log-likelihood. More precisely, we reported that sometimes a further refinement through L-BFGS-B may be needed to increase the accuracy in the localization of the maximum of the concentrated log-likelihood, but the gain in performance of the prediction, i.e. the error of the property predicted with respect to the correct one, can be considered negligible. For this reason, optimizing the concentrated log-likelihood through application of the L-BFGS-B is not relevant for the work presented here and thus we will not run any timings for it, although FEREBUS allows the user to do so.

Test Case: Water Decamer

As a test function, the net atomic charge on the oxygen of a central water molecule in a ten-molecule water cluster was taken as the property of interest. Using a spherical coordinate scheme centered on the atom of interest, such a system possesses \( 3N - 6 = 3(10 \times 3) - 6 = 84 \) geometric inputs, also known as features in machine learning language. The kriging problem thus exists in an 84-dimensional feature space. Accordingly, optimization of the concentrated log-likelihood occurs over a set of 84 \( \theta \) values and 84 \( p \) values. The details of the construction of the model for the decamer have been reported in our previous work.\textsuperscript{[32]} In particular, it was shown\textsuperscript{[32]} that a good compromise between accuracy of the prediction and dimension of the training set is obtained by using training sets with 1000 training points to predict the monopole, dipole and quadrupole moments of the atoms of the central molecule, as well as their atomic energies. The decamer system of reference\textsuperscript{[32]} will be used here as a test case for the parallelization algorithms implemented in this work. This system has been selected because its high-dimensional feature space and high conformational freedom make serial PSO and DE algorithms unfeasible. Throughout this work, decamer clusters will be written as DEC, supplemented by a number representing the number of training points used for that particular model. For example, DEC300 indicates the kriging model for a water decamer built using 300 training points.

Parallelization Strategy

Terminology and general remarks

Before describing in detail how parallelization is implemented we introduce our terminology. Because we are using both MPI and OpenMP parallel environments we will make use of the (respective) terms processes (from now on MPIp) and threads (from now on OMPt). Processes are associated with the chosen MPI environment and they represent the execution units that provide the resources needed by the program to perform all of the algorithm’s instructions. Threads, OMPt, represent the group of entities belonging to a parent MPI process that can be scheduled for execution, and they are associated with the OpenMP environment. Unlike processes, threads share the resources between them and exist in a fork-join manner\textsuperscript{[18]} (i.e., being created at the beginning of a specific parallel region and being destroyed at its exit). MPI processes, which exist for the lifetime of a running program, will communicate (update variables) by calls to a message-sending/receiving library, whereas OpenMP threads access the same (i.e. shared) memory on a physical node. OpenMP is limited to per-node parallelism whereas MPI can be used both between physical cores on the same node, and between nodes. As well as the hybrid MPI-OpenMP mode, FEREBUS may also be run in MPI-only and OpenMP-only parallel mode.

Threads and processes run on physical nodes, which are composed of a certain number of physical cores, hereafter just cores. In general, the association between processes, threads and cores is not one-to-one. Only in the case of single-process single-thread (MPIp = 1, OMPt = 1) (i.e., serial) execution does the number of processes equal the number of cores. Details about the numbers of cores that we consider for the benchmarking of the code will be given in later sections.

The configuration and placement of MPI processes and OpenMP threads, across available cores and discrete nodes, is determined by the batch scheduler, operating system and user flags. The term configuration refers to the number of MPI processes and OpenMP threads used. The term placement is related to the position of the MPI processes and OpenMP threads on a single node, as will be clarified in “Details of the Evaluation Environment” section. OpenMP threads will be associated with a given MPI process, and those associated with a given MPI process can only exist on the same physical node as the MPI process. When the OpenMP threads/MPI processes do not overlap on a single core, we speak of a nonoversubscribed state. In the opposite case, the oversubscribed state, the threads/processes compete for the resources of a core degrading the overall performances of the code. Typically, in the non-oversubscribed state, the number of OpenMP threads times the number of MPI process will not exceed the number of cores on a single node.

Figure 1 displays an example of the division of the work inside each node. In the case presented in Figure 1 each node has two MPI processes, and 12 OpenMP threads are associated to each MPI process.

In a typical problem, we need to train up to \( M \) different physical properties (e.g., an atomic multipole moment), that represent a \( N_{t} \)-dimensional problem, with \( N_{t} \) training points. FEREBUS is parallelized at two different levels:

1. Each of the DE and PSO algorithms uses \( P_{t} \) processes. A detailed description of the parallelization algorithm will be given in “The MPI Implementation of the PSO Approach" section for PSO and “The MPI Implementation
of the DE Approach" section for DE. The communications at this stage will focus on the MPI paradigm.

2. For each of the $P_1$ processes, $P_2$ parallel OpenMP threads are considered, which are used to build the $R$ matrix and its inverse, and to calculate the value of the concentrated log-likelihood at a given point. These computations are common to the DE and PSO algorithms.

In summary, the total number of cores, denoted $T_c$, required for a single calculation (in the non-oversubscribed state) is easily obtained as $T_c=P_1P_2$.

The MPI implementation of the PSO approach

A parallelization of the PSO algorithm was reported by Prasain et al.\cite{33} who described a hybrid MPI/OpenMP implementation. However, in that work, PSO was applied to optimize a function less computationally expensive to calculate than the one considered here. The difference in computational cost of the calculation of the function to optimize leads us to a different parallelization approach. In the approach of Prasain et al., $N$ particles are divided among $X$ nodes (using MPI). In each node, each particle is assigned to a different (OpenMP) thread, i.e., in each node $T=N/X$ threads are created. The global best value in each process is calculated, and the information sent to a central processor. The global best value for the entire swarm is calculated and broadcast to every node.

In the implementation presented here, the OpenMP threads are only used to calculate the value of the concentrated log-likelihood for each particle whereas the set of particle is evenly distributed among processes using MPI. While Prasain et al. assigned a single particle to each OpenMP thread, we decided to reserve the use of threads to the calculation of the fitness function (i.e., the concentrated log-likelihood), which as will be shown, represents the most computationally expensive part of the whole calculation. In Figure 2, we show the pseudocode for the PSO algorithm in order to show the critical operations performed by this algorithm.

At each iteration, the particle position is updated. Updating requires two pieces of information: the “particle best position” and the “global best position.” The former is specific for each particle and it represents the best position found by the particle during its search in $\theta - \rho$ space, whereas the latter is the best position found by the swarm in $\theta - \rho$ space.

For a given number of processes, $P_1$, the dimension, $S$, of a given swarm is split across $P_1$ processes in what we call the first level of parallelization (see “Particle Swarm Optimization” section). We consider that $\text{mod}(S, P_1)=0$, where $\text{mod}$ is the modulus function. We call $S_0$ the number of particles in each process, and according to our definition $S=P_1S_0$. If $S$ is not a multiple of $P_1$ we can just redefine the dimension of the swarm to be

$$S'=S-\text{mod}(S, P_1)+1$$

The choice to add particles instead of dealing with a value of $S$ that is not a multiple of $P_1$ avoids an un-even splitting of the particles between processes, with negligible cost in computational time. In fact, the number of particles added is at most $P_1-1$. Furthermore, increasing the number of particles actually increases the search performances of heuristic algorithms such as PSO.

From eq. (5) it follows that the only information each particle needs from the others is the global best position (gbest), which is checked for every iteration. For each MPI process, each particle moves independently from the others. After the swarm has moved, the new positions are calculated and the new particles best positions are updated. Next, the maximum of the concentrated log-likelihood for each process is found, and the highest value between those maxima is chosen. Finally, the maximum value of the log-likelihood, along with its position (i.e., values of hyperparameters) is sent back to all the processes. The modified pseudo-algorithm to consider these communications is depicted in Figure 3 where “Process $j$” indicates the $j$-th MPI process.

The MPI implementation of the DE approach

Strategic parallelization of DE requires more steps than that presented for the PSO algorithm. This fact is mainly due to the mutation section of the DE algorithm. The construction of a mutated vector requires a number of parent vectors randomly chosen from the population. Because the population is divided...
through the different processes, a vector located on another process may be required. Unlike the PSO algorithm, where the number of communications was fixed for each iteration, the number of communications among different processes is not defined \textit{a priori} in DE. This number ranges from a minimum of zero (i.e., all the parent vectors needed are already local to the process, and thus no cross-process communications are involved) to a maximum of $N_m^*V$, where $N_m$ can be an integer between 2 and 5 depending on how many parent vectors are required by the chosen mutation strategy. It clearly follows that $\text{DE/rand/2}$ (see eq. (11)) is the most expensive mutation strategy in terms of communications, because this construction method of the mutated vector requires five parents, which may all lie on other processes.

The parallelization strategy for DE consists of the following five steps: i) the population is divided between the different processes, applying the same strategy presented in “The MPI Implementation of the PSO Approach” section for the case when the population size is not an integer multiple of the number of processes, each process will then receive $V_p$ vectors; ii) at each generation, $N_m$ vectors of size $V$ are generated, each of which contains the random parents to be mixed in the mutation step to obtain the mutated vector. This step involves the generation of random numbers, which is handled with a non-parallelized random number generator. Use of a nonparallelized Random Number Generator (RNG) ensures that random numbers from the same sequence are chosen for each process. One problem that can arise for the generation of mutated vectors is that the “parents” do not lie in the same process of the mutated one. In the current implementation, each process acquires the parent vectors from other processes (if they lie in different processes) and at the same time it sends its own vectors to the processes that need them. Therefore, each process needs to know the full list of the parent vectors required by each vector. Pseudocode of the parallelization strategy for DE, depicting all the communications involved, is shown in Figure 4.

![Figure 4](image-url)

The calculation of $R$ matrix is computationally the most expensive part of the calculation of the concentrated log-likelihood for each particle or vector. In addition, at each iteration, $R$ must be re-calculated for each vector or particle. We take advantage of the fact that the $R$ matrix is symmetric by calculating only its upper half. Also, the diagonal is always nothing done. Otherwise, a local copy of the parent vector is created in the process where the mutated vector lies. Creating the local copy involves a communication step between the two processes; iv) after collection of all the parents, the mutated vector is built. The cross-over process and the selection are specific to each vector and do not need communication; v) at the end of the loop involving the vectors, the information about the best vector must be sent to all the processes. This is handled in the same way as already described for PSO (see “The MPI Implementation of the PSO Approach” section).

Overall parallelization strategy with OpenMP

OpenMP parallelization can be applied to both PSO and DE by using it to parallelize the construction of the $R$ matrix and the calculation of the concentrated log-likelihood. The parallelization of these steps is straightforward, since they involve iterative loops that can be easily handled by OpenMP. These loops are independent between MPI processes and the computation within each loop does not require access to any variable on another MPI process. That is, there is no communication required between MPI processes from inside the OpenMP parallel regions. The calculations considered in the OpenMP loops are given by eqs. (1–4).

Figure 5 displays a flowchart outlining the OpenMP parallel region considered in the calculation of the concentrated log-likelihood.

The calculation of the $R$ matrix is computationally the most expensive part of the calculation of the concentrated log-likelihood for each particle or vector. In addition, at each iteration, $R$ must be re-calculated for each vector or particle.
constant and filled with ones (see eq. (2)). The dimension of the \( \mathbf{R} \) matrix must also be taken into careful consideration when tasks are assigned to each computational unit. In fact, as the size of the \( \mathbf{R} \) matrix increases the required memory, it can eventually cause a detrimental effect on performance due to increasing number of elements having to be fetched from more distant elements of the memory hierarchy (on the same node). Despite the \( \mathbf{R} \) matrix being symmetric, each process requires the full \( \mathbf{R} \) matrix to be stored for each vector or particle in the process. In the calculation of the concentrated log-likelihood the complete inverse of the \( \mathbf{R} \) matrix is needed (see eqs. (1), (3), and (4)). However, the inversion of the \( \mathbf{R} \) matrix performed through the Cholesky decomposition\(^{34}\) implemented in NAG libraries\(^{36}\) requires only half of the \( \mathbf{R} \) matrix. Thus, the maximum value of the size of the \( \mathbf{R} \) matrices that will fit the RAM of a given node, can be estimated from \([\text{Nt x Nt x vector/particles in each process x MPIp per node x size in bytes of a double precision variable}]\).

In FEREBUS two distinct ways of calculating the entries of the \( \mathbf{R} \) matrix are allowed. In the first one the value of \( p_k \) in eq. (2) is kept constant for all \( k \) and for every iteration, only \( \theta \) values are optimized. The value of \( p_k \) can be user defined, but the value of 2 representing the Gaussian correlation function is one of the most common choices.\(^{23}\) The calculation with constant \( p_k = 2 \) will be indicated as Prefix.

After selection of Prefix we take advantage of the fact that the correlation function is a square and thereby avoid the expensive operations of raising to a power and taking the absolute value. Equation (2) can be recast in the following form:

\[
R_{ij} = \exp \left[ -\sum_{k=1}^{N_t} \theta_k (x_i^k - x_j^k) (x_i^k - x_j^k) \right]
\]

When optimization of \( p \) along with \( \theta \) is allowed, the option Popt is selected and the entries in \( \mathbf{R} \) matrix are calculated as shown in eq. (2).

**Details of the Evaluation Environment**

**General considerations**

In order to test the efficiency of the parallelization algorithm we considered the water decamer described in “Test Case: Water Decamer” section, with 3000 training points. FEREBUS was compiled with ifort 15.0.3, with Intel's MKL (version 11.2) used as the parallel random number generator for the optimization. In particular, a routine called “vdRngUniform” is used to generate random numbers drawn from a uniform distribution. The hardware used for the benchmarks in this test comprises a set of four nodes, each comprising two Intel(R) Xeon(R) CPU E5-2690 v3 “Haswell” chips with a nominal clock speed of 2.60 GHz, each with 12 cores. Each node is equipped with 128 GB RAM, and the nodes are connected by high speed interconnect (QDR Infiniband). MPI environment employs OpenMPI (version 1.8.3) implementation. All calculations will be performed by using the option “binding to socket” for OMPt. In case the number of OMPt for a single MPIp is greater than the actual number of cores on a single socket, the option “bind to node” will be used.

**Details of the input**

The number of particles/vectors (\( S \) for particles, \( V \) for vectors) set for the PSO/DE algorithms, respectively, is not the recommended value (see “Particle Swarm Optimization and Differential Evolution” sections) but it was hard-set at 1440. We used such a value for \( S \) and \( V \) to ensure an even distribution of vectors/particles among the processes and therefore an even workload among processes. In fact, as already discussed in “The MPI Implementation of the PSO Approach” section, if the initial dimension of the swarm \( S \) is not a multiple of the number of processes, more particles are added until \( \text{mod} (S', P_{\text{MPI}}) = 0 \), where \( S' \) is the new dimension of the swarm, leading to an artificial increase in the amount of work in some of the calculations. The same reasoning applies to population size for DE.

The runs were performed over 10 iterations plus 1 initialization step, considered as iteration 0, and the results averaged out

| Configuration | Time |
|---------------|------|
| 1 MPIp, 24 OMPt | 10343 s |
| 2 MPIp, 12 OMPt | 3300 s |
| 4 MPIp, 6 OMPt | 2573 s |

**Table 1.** Computational time needed by 10 iterations of a single run of FEREBUS for PSO with fixed \( p = 2 \) and 1440 particles
to obtain an average time per iteration. In the benchmarking of DE we should have a different run for each mutation strategy. However, as the only difference between different mutation strategies is the number of communications needed to build the mutated vector (see Figure 4), we only considered the case DE-MS4. In DE-MS4 each mutated vector requires 5 parents vector to be calculated, representing the most expensive mutation strategy computationally. The number of training points used in this work is \(N_t = 3000\). The computational time is taken as the time spent in DE and PSO, including the initialization of the particles/vector. The times shown in the Results section do not include the I/O time, which merely represents pre-calculation operations (reading the training points from an external file) and postcalculation operations (writing the results to a file).

**Results**

The effect of different sizes of the \(R\) matrix on the performance of FEREBUS can be detected by a simple experiment. There are two possible placements of the MPI processes and we first examine these to determine the effect of the size of the \(R\) matrix on the performance of FEREBUS. Placement 1 relates to placing both MPI processes on a single socket (see Figure 1) and Placement 2 relates to placing one MPI process on each of the two sockets.

For each of the two placements we run FEREBUS with 2 MPIp and 6 OMPt on a single node. For an \(R\) matrix of dimension 3000, the time taken by Placement 1 is 5230 seconds, and the time taken for Placement 2 is 4279 seconds. If the dimension of the \(R\) matrix is reduced to 300 training points, which corresponds to an \(R\) matrix two orders of magnitude smaller, then Placement 1 takes 16.2 s and Placement 2 takes 16.0 s.

As shown, the placement and dimension of the \(R\) matrix must be considered when a calculation is performed. We may then draw some indication about which placement can be considered optimal, i.e. the placement that does not lead to a degradation of performance. Whilst the choice of placement for small training data is not significant compared to the total time taken by FEREBUS, the choice for larger training data clearly points to the optimal placement being Placement 2.

Thus, henceforth, we place MPI processes on a per-socket basis (one for each socket until all sockets have one MPI process, then a second MPI process on each socket, and so on).

It is not easy to tell in advance which is the best choice of MPI processes and OpenMP threads for the job on the physical computational node, given we can vary either, as long as their product is less than the physical number of cores on a node, which is 24. Indeed, the best choice depends upon the size of the \(R\) matrix, and the number of particles/vector involved in the calculation. By using the same number of cores, but distributing them in different ways among MPI and OpenMP, different results are obtained, as reported in Table 1.

The best configuration is thus the one that favors higher MPI parallelization with respect to the OpenMP parallelization, which corresponds to “4 MPIp and 6 OMPt” in the example shown in Table 1. The worst result is obtained for “a single MPIp and 24 OMPt”. The reason for the behavior shown in Table 1 will be explained later.

Figure 6 reports the time taken by a single run of FEREBUS, per iteration. For clarity, Figure 6 only shows the case with one and with eight OpenMP threads as a function of the different numbers of MPI threads. Time for the intermediate OpenMP threads for DE-MS4 and PSO are reported in the Supporting Information Figure S1. Times reported are averaged over 10 iterations.

Our primary purpose is to show the performances of the two optimization algorithms under the implemented parallelization schemes rather than a comparison between them, mainly because direct comparison is not meaningful. In fact, DE and PSO are completely different algorithms, and in addition, DE has possibly more communications than PSO, but their number is variable at each iteration, depending on the positions of the parents vectors in the different processes with respect the position of the mutated vector (see “The MPI Implementation of the DE Approach” section and Figure 4). In Supporting Information Tables S1 and S2, the numerical value of the times for 1 OMPt and different values of MPIp of PSO and DE are reported. In serial configuration (1 OMPt and 1 MPIp) DE seems to perform slightly better that PSO. By increasing the number of MPIp the effect of the higher number of MPI communications for DE starts to become evident especially for intermediate number of MPI processes.

![Figure 6](wileyonlinelibrary.com)
While Figure 6 shows the time needed for a single iteration, it is important to note that a single optimization can take thousands of iterations. Thus, the need for a highly optimized program becomes also evident. Our results show that both parallelization configurations cause a significant reduction compared to the serial time (Figure 6), (i.e., 1 MPIp, 1 OMPt) of 2871 s. The configuration of 90 cores (i.e., 90 MPIp, 1 OMPt) gives a time of 47 s, which is a speed up of two orders of magnitude. Similar results were obtained for DE. The results for nonoptimized p display a similar behaviour, the only difference being the higher time per iteration, and are reported in Supporting Information Figure S2.

In addition to showing the time for each configuration, as per Figure 6, the scaling of the code can be represented by a scaling plot. The scaling (σp) is calculated as
\[
σ_p = \frac{t_1}{t_n}
\]
where p represents the processing units (MPIp or OMPt, with the other fixed), t1 is the time in the single core configuration and tn is the time in the n cores configuration. The scaling is reported in two ways. First, by considering the program running with only MPI or with only OpenMP, i.e., σp as a function of MPIp with OMPt = 1 and σp as a function of OMPt with MPIp = 1, along with the ideal scaling of the code. Ideal scaling represents the linear increase in speed with a corresponding linear increase in the number of cores used. Scaling is reported in Figure 7 for PSO and Figure 8 for DE-MS4. The scaling in the case of nonoptimized p is reported in Supporting Information Figures S3 and S4 for PSO and DE-MS4, respectively.

From Figures 7 and 8 it follows that MPI implementation for both PSO and DE follows the ideal scaling behavior roughly until 10 cores. From 10 up to 80 cores, the behavior starts to diverge from the ideal scaling and the plateau become clearly visible around 80 cores. However, even in the worst case, 90 MPIp, ~70% of the ideal scaling is retained (i.e., a speed-up of ~61 times). The sudden drop in the performance of MPI when it goes from 80 to 90 MPIp, can be explained by considering that MPI deals with particles/vectors only as shown in Supporting Information Figure S5. If the number of processes goes from one to two, the number of particles/vectors per process drops from 1440 to 720. When the number of processes increases from 80 to 90, the difference in the number of particles per process is only 2 (i.e. 18 particles/vectors per MPIp in the configuration with 80 MPIp and 16 particle/vector per MPIp in the configuration with 90 MPIp). That is to say that for MPI, the overheads of parallelization increasingly outweigh the savings of further dividing the available parallel work, as noted going from 80 to 90 processes.
In the case of OpenMP the behavior is again similar for both PSO and DE but the plateau is reached sooner, around 6 OMPt. For OpenMP the cause of this plateau is different. Our hybrid implementation generally divides work across MPI processes. We use OMP threads to parallelize R and log-likelihood functions. The speed-up for OpenMP is not expected to scale over a large number of threads since there remains much sequential work which, as described by Amdahl’s Law, puts a limit on the maximum speed up possible. Moreover, the dimension of the R matrix has a detrimental effect of the performances when the node on which the code is running is filled as shown in Table 1. Scaling for OMPt shows a ～60% improvement of time at 6 OMPt for a system of the size presented here.

From Figures 7 and 8, the behavior of the results shown in Table 1 becomes clearer. The configurations with a higher number of MPIp with respect to OMPt are preferred because the better scaling of MPI configurations with respect to OpenMP configurations, i.e., for the same number of cores the configurations with more MPI processes scales better than the configurations with more OpenMP threads. We reported a more detailed profiling in the supporting information (see Figures S6 and S7, Tables S3, S4 and S5). FEREBUS reaches a ～61 times speed-up but perhaps a higher gain can be reached with a better handling of the allocation of the memory in each node. Future developments of the code will be oriented towards this direction.

Conclusions
We have described the details of the parallelization strategy employed in the in-house code FEREBUS. This program uses kriging in conjunction with the PSO and DE algorithms. We have used MPI to parallelize across particles and OpenMP to parallelize the computationally intensive work per thread relating to the R matrix and the maximization of the concentrated log-likelihood function. This parallelization returns a speed-up of 61 times when run on 90 cores compared to the time on a single core with the water decamer now being solved in seconds.

We have outlined how each of the MPI and the OpenMP parallelization paradigms may independently decrease the time-to-solution and highlighted, as a critical factor, that the combination of these paradigms in an hybrid MPI-OpenMP implementation gives best overall performance.

MPI parallelization involves the splitting of the number of particles or vectors of the searching algorithms (namely PSO and DE) among different processes. Scaling of the pure MPI parallelization is near ideal up to a low number of processes with a 30% drop in parallel efficiency thereafter, due to the number of particles (or vectors) being constant in the calculation (see Supporting Information Figure S5).

OpenMP parallelization is used (within the MPI parallelization) to boost the performance of the calculation of some specific quantities that are computationally very expensive (and have memory locality for each particle) such as the R matrix and the value of the log-likelihood. Parallel efficiency for OpenMP degrades more quickly than for (pure) MPI because OpenMP is targeted only at specific parts of the code, leaving a significant serial proportion. Moreover, we have examined how the efficiency of OpenMP is affected by configuration choices on the physical computational nodes.

Despite limitations in MPI and in OpenMP parallelization, we have shown that when the two are combined together (i.e. with OpenMP as a boost for MPI) the overall efficiency increases (compared to either OMP or MPI by themselves) to give, for example on 3 MPIp and 4 OMPt, a speed-up of 9 times, which is a parallel efficiency of 75%.

As such, FEREBUS represents an invaluable tool in the development of the training of the force field under development called FFLUX. The training of kriging models with large molecules and/or clusters (from tens to hundreds of atoms), that need thousands of training points, is a problem that becomes easily intractable without efficiently parallelized and optimized code.

Acknowledgments
The authors thank the IT Services Research Infrastructure group at the University of Manchester for the exclusive access of nodes in order for us to undertake our benchmark simulations and for the useful computational advice. Gratitude is also expressed to the EPSRC for the award of an Established Career Fellowship to one of us (PLAP).

Keywords: kriging • machine learning • OpenMP • MPI • parallelization • particle swarm optimization • differential evolution • QTAIM • force field design • IQA

Additional Supporting Information may be found in the online version of this article.

How to cite this article: N. Di Pasquale, M. Bane, S. J. Davie, P. L. A. Popelier, J. Comput. Chem. 2016, 37, 2606–2616. DOI: 10.1002/jcc.24486
[17] E. Gabriel, G. E. Fagg, G. Bosilica, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, T. S. Woodall, Open MPI: Goals, Concepts, and Design of a Next Generation MPI Implementation In Proceedings, 11th European PVM/MPI Users’ Group Meeting; Springer Berlin Heidelberg, 2004, 97–104.

[18] OpenMP Application Program Interface OpenMP Architecture Review Board http://www.openmp.org.

[19] L. Lagardère, F. Lipparini, E. Polack, B. Stamm, M. Schnieders, E. Cances, P. Ren, Y. Maday, J. Piquemal, J Chem Theory Comput 2015, 11, 2589.

[20] S. Lindert, D. Bucher, P. Eastman, V. Pande, J. A. McCammon, J Chem Theory Comput 2013, 9, 4684.

[21] D. R. Jones, J Global Optim 2001, 21, 345.

[22] D. R. Jones, M. Schonlau, W. J. Welch, J Global Optim 1998, 13, 455.

[23] C. E. Rasmussen, C. K. I. Williams, In Gaussian Processes for Machine Learning; MIT Press: Cambridge, USA, 2006.

[24] D. J. J. Toal, N. W. Bressloff, A. J. Keane, C. M. E. Holden, Eng Optim 2011, 43, 675.

[25] R. C. Eberhart, Y. Shi, Proceedings of the 2000 Congress on Evolutionary Computation 2000, Vol. 84.

[26] B. K. Panigrahi, L. M. Hiot, Y. Shi, Handbook of Swarm Intelligence: Concepts, Principles and Applications, Vol. 8; Springer, Berlin Heidelberg, 2010.

[27] R. Storn, K. Price, J Global Opt 1997, 11, 341.

[28] S. Das, P. N. Suganthan, IEEE Trans Evol Comput 2011, 15, 4.

[29] Storn, R. On the Usage of Differential Evolution for Function Optimization; IEEE: Berkeley, CA, 1996.

[30] J. Brest, S. Greiner, B. Boskovic, M. Mernik, V. Zumer, Eng Optimiz, 2006, 10, 646.

[31] C. Zhu, R. H. Byrd, P. Lu, J. Nocedal, ACM Trans Math Software 1997, 23, 550.

[32] S. Davie, N. Di Pasquale, P. L. A. Popelier, J Comput Chem, 2016, DOI: 10.1002/jcc.24465.

[33] H. Prasain, G. K. Jha, P. Thulasiraman, R. Thulasiram, A Parallel Particle Swarm Optimization Algorithm for Option Pricing In IEEE International Symposium on Parallel & Distributed Processing, Workshops and PhD Forum. IEEE: Atlanta, GA, 2010; p. 1.

[34] J. J. Du Croz, N. J. Higham, IMA J Numer Anal 1992, 12, 1–19.

[35] NAG The Numerical Algorithms Group (NAG), Oxford, United Kingdom www.nag.com.

Received: 30 May 2016
Revised: 13 August 2016
Accepted: 20 August 2016
Published online on 21 September 2016