Branching and Bounding Improvements for Global Optimization Algorithms with Lipschitz Continuity Properties

Coralia Cartis[*], Jaroslav M. Fowkes[*] and Nicholas I. M. Gould[†]

April 29, 2014

Abstract

We present improvements to branch and bound techniques for globally optimizing functions with Lipschitz continuity properties by developing novel bounding procedures and parallelisation strategies. The bounding procedures involve nonconvex quadratic or cubic lower bounds on the objective and use estimates of the spectrum of the Hessian or derivative tensor, respectively. As the nonconvex lower bounds are only tractable if solved over Euclidean balls, we implement them in the context of a recent branch and bound algorithm (Fowkes et al., 2013) that uses overlapping balls. Compared to the rectangular tessellations of traditional branch and bound, overlapping ball coverings result in an increased number of subproblems that need to be solved and hence makes the need for their parallelization even more stringent and challenging. We develop parallel variants based on both data- and task-parallel paradigms, which we test on an HPC cluster on standard test problems with promising results.

Keywords: global optimization, lipschitzian optimization, parallel branch and bound, nonconvex programming.

1 Introduction

In many applications one encounters the global optimization problem

$$\min_{x\in\mathcal{D}} f(x), \tag{1.1}$$

where $f\colon \mathcal{D} \subset \mathbb{R}^n \to \mathbb{R}$ is smooth and in general non-convex and $\mathcal{D}$ is a compact, convex set. It has been shown that this problem is NP-hard (Kreinovich and Kearfott, 2005) and requires global information to be solved efficiently (Stephens and Baritompa, 1998). Branch and bound algorithms are a traditional way to solve (1.1) (see for example, Horst and Tuy, 1996, Pintér, 1996 and Neumaier, 2004). Such algorithms work by recursively splitting (branching) the domain $\mathcal{D}$ into subregions and bounding the objective function f over each subregion until the global minimum is found.

In order for such algorithms to be efficient, one requires accurate and efficiently computable lower bounds on f over each subregion (upper bounds are typically taken to be the function evaluated at some point or the outcome of a local solver). Global information in the form of

[*]School of Mathematics, University of Edinburgh, The King's Buildings, Edinburgh, EH9 3JZ, Scotland, UK. Emails: coralia.cartis@ed.ac.uk; jaroslav.fowkes@ed.ac.uk.

[†]Computational Science and Engineering Department, Rutherford Appleton Laboratory, Chilton, Oxfordshire, OX11 0QX, England, UK. Email: nick.gould@stfc.ac.uk.

a Lipschitz constant is often used to construct such lower bounds. The case where the lower bound is based on a Lipschitz constant of the objective function f has the immediate form $f(x) \geq f(x_{\mathcal{B}}) - L_f(\mathcal{B})\|x - x_{\mathcal{B}}\|$ for some point $x_{\mathcal{B}}$ in a subregion $\mathcal{B}$ and a Lipschitz constant $L_f(\mathcal{B})$ for f over $\mathcal{B}$; this case has been well studied in the global optimization literature (see Evtushenko, 1971; Piyavskii, 1972; Shubert, 1972; Pardalos, Horst and Thoai, 1995; Pintér, 1996; Strongin and Sergeyev, 2000; Neumaier, 2004; Kvasov and Sergeyev, 2012b; Sergeyev, Strongin and Lera, 2013, and references therein).

A more accurate lower bound using a Lipschitz constant of the gradient of the objective function $g = \nabla_x f$ can be derived using Taylor's theorem to first order

$$f(x) \geq q_{\mathcal{B}}(x) := f(x_{\mathcal{B}}) + (x - x_{\mathcal{B}})^T g(x_{\mathcal{B}}) - \frac{L_g(\mathcal{B})}{2}\|x - x_{\mathcal{B}}\|_2^2 \tag{1.2}$$

for some point $x_{\mathcal{B}}$ in a subregion $\mathcal{B}$ and the gradient's Lipschitz constant $L_g(\mathcal{B})$ for g over $\mathcal{B}$, and has, together with refinements, also been well studied (see Breiman and Cutler, 1993; Baritompa and Cutler, 1994; Sergeyev, 1998; Kvasov and Sergeyev, 2009, 2012a; Evtushenko and Posypkin, 2011, 2013; Lera and Sergeyev, 2013; Fowkes, Gould and Farmer, 2013). Evtushenko (1971); Baritompa and Cutler (1994); Evtushenko and Posypkin (2011, 2013) have replaced $-L_g(\mathcal{B})$ in (1.2) with a lower bound on the spectrum of the Hessian. Baritompa and Cutler (1994) also go a step further and replace the simple quadratic term in (1.2) by a quadratic form that lower bounds f.

The case where one goes even further and uses second order Taylor's theorem to obtain a cubic lower bound using a Lipschitz constant for the Hessian $H = \nabla_{xx} f$ was considered in Fowkes et al. (2013)

$$f(x) \geq c_{\mathcal{B}}(x) := f(x_{\mathcal{B}}) + (x - x_{\mathcal{B}})^T g(x_{\mathcal{B}}) + \frac{1}{2}(x - x_{\mathcal{B}})^T H(x_{\mathcal{B}})(x - x_{\mathcal{B}}) - \frac{L_H(\mathcal{B})}{6}\|x - x_{\mathcal{B}}\|_2^3 \tag{1.3}$$

for some point $x_{\mathcal{B}}$ in a subregion $\mathcal{B}$ and the Hessian's Lipschitz constant $L_H(\mathcal{B})$ for H over $\mathcal{B}$.

In this paper, we propose new bounding techniques using refinements of the bounds (1.2) and (1.3). We show that for the first order bound (1.2) it is possible to obtain tighter results by using existing lower bounds on the spectrum of the Hessian, some of which have not been previously used, as far as we are aware, in the context of Lipschitz based global optimization. Additionally, we extend one of these approaches to the second order bound (1.3) by replacing the Lipschitz constant estimate with a novel lower bound on the spectrum of the third order derivative tensor. We test the new proposals in the Overlapping Branch and Bound (oBB) framework proposed in Fowkes et al. (2013) which allows efficient global solution of the non-convex lower bounding subproblems

$$\min_{x \in \mathcal{B}} l_{\mathcal{B}}(x) \tag{1.4}$$

where $l_{\mathcal{B}}(x)$ is either $q_{\mathcal{B}}(x)$ in (1.2) or $c_{\mathcal{B}}(x)$ in (1.3), over each subdomain $\mathcal{B}$, by letting $\mathcal{B}$ be a Euclidean ball which makes (1.4) tractable.

In greater detail, oBB is a Lipschitz derivative based approach that uses an overlapping covering of balls rather than rectangular partitions. The main idea behind oBB is to recursively split an initial ball covering the domain $\mathcal{D}$ into sub-balls until we find a ball (or balls) of sufficiently small size containing the global minimiser of f over $\mathcal{D}$. Using (1.4), oBB is able to obtain lower bounds on the minimum of f over each ball which can then be used to discard balls that cannot contain the global minimiser, i.e. balls whose lower bound is greater than the smallest upper bound. Each ball of radius r is split into 3^n overlapping sub-balls of half-radius $r/2$ centred at the vertices of a hypercubic tessellation of edge-length $r/\sqrt{n}$ around the centre of the ball. This ensures that the sub-balls entirely cover the original ball with a constant amount of overlap irrespective of the original ball's radius. A detailed description of oBB is given as Algorithm 3.1 when the latter is run on one (master) processor core.

As one would expect, testing the proposed bounds in oBB we find that in general, due to additional problem information being employed in the model, second order models yield better lower bounds on the objective function compared to first order models and hence can potentially lead to better branch and bound algorithms for Lipschitz optimization.

In general, Lipschitz-based lower bounding subproblems are non-convex, and branch and bound algorithms require their global solution (an NP-hard problem over boxes, see Theorem 2 in Kreinovich and Kearfott, 2005). This is usually achieved using techniques such as vertex enumeration, interval arithmetic, convexification or problem specific constructs (Neumaier, 2004) but these may not be flexible enough or suitably scalable for the purpose of generic problem solvers. The approaches in Evtushenko (1971) and Evtushenko and Posypkin (2011, 2013) and in oBB allow global solution for both first and second order models by minimizing these models over Euclidean balls rather than boxes, which ensures that the subproblems can be solved in polynomial time. Evtushenko (1971); Evtushenko and Posypkin (2011, 2013) propose the use of a non-uniform mesh and employ the global solutions over balls to exclude elements in such a non-uniform rectangular partition. While their algorithm is already in a parallel framework it only uses first order models. Gaviano and Lera (2008) devise a similar algorithm which also excludes elements from a non-uniform rectangular partition using zeroth order models. oBB uses second order models on overlapping balls leading to an overlapping covering of the domain (as opposed to the rectangular partition used in Evtushenko, 1971; Evtushenko and Posypkin, 2011, 2013). Thus the oBB approach leads to more computational effort as well as potential doubling of work so parallelisation is both crucial and challenging for obtaining good performance.

Similarly to other branch and bound algorithms, there is also the curse of dimensionality which is made worse by the high number of balls in each oBB covering. At each iteration, oBB splits a ball into 3^n smaller sub-balls (with constant overlap) whereas traditional branch and bound splits a box into only two larger sub-boxes. In the worst case both algorithms can be said to perform comparably, with each ball in oBB being split into 3^n sub-balls, compared to each box being split into 2^n sub-boxes for traditional branch and bound. However, as both algorithms use disparate coverings with subdomains of different sizes, it is difficult to compare them directly in general.

Due to the curse of dimensionality, many parallel branch and bound algorithms over boxes have been proposed in the literature (see Ananth, Kumar and Pardalos, 1993; Crainic, Le Cun and Roucairol, 2006; Alba et al., 2006; Casado et al., 2008; Evtushenko, Posypkin and Sigal, 2009; Paulavičius, Žilinskas and Grothey, 2011 and the survey by Gendron and Crainic, 1994). Gendron and Crainic (1994); Crainic et al. (2006) have classified the main approaches into two classes: Type I and Type II parallelism that correspond to forms of data parallelism and task parallelism respectively. In Type I parallelism operations on subproblems (e.g. bounding) are conducted in parallel whereas the branch and bound tree is explored in serial (i.e. by one processor). In Type II parallelism by contrast, the tree itself is explored in parallel by many processors. It should be noted that while branch and bound algorithms are conceptually thought of as exploring a tree, for reasons of efficiency they are often implemented numerically as a priority queue (Crainic et al., 2006). In order to parallelise oBB, we develop parallel algorithms using both data parallel (Type I) and task parallel (Type II) paradigms. Our main contribution here is to develop an effective task parallel variant of oBB using novel hashing techniques that enable efficient communication, essentially removing the doubling of work entirely. Additionally, we address the problem of balancing the load between processors by implementing an effective load balancing strategy.

The layout of the paper is as follows. First order lower bound estimates are given in Section 2.1 and second order lower bound estimates in Section 2.2, with numerical results presented in Section 2.3. We then consider the two main paradigms for parallelising the oBB algorithm, data parallel (bounds in parallel) in Section 3.1 and task parallel (tree in parallel) in Section 3.2 with numerical results in Section 3.3. Finally, we draw conclusions in Section 4.

2 Improving Lipschitz lower bounds

Let us first consider devising more accurate lower bound estimates for Lipschitz based branch and bound algorithms. We will therefore begin this section by looking at improved estimates for the first order lower bound (1.2) and then extend some of these ideas to the second order lower bound (1.3). It should be noted that the first order lower bound (1.2) and refinements are well known in the Lipschitz derivative optimization literature, see for example, Evtushenko (1971); Breiman and Cutler (1993); Baritompa and Cutler (1994); Sergeyev (1998); Kvasov and Sergeyev (2009, 2012a); Evtushenko and Posypkin (2011, 2013); Lera and Sergeyev (2013); Fowkes et al. (2013) but the use of the cubic lower bound (1.3) within a global optimization context is recent (to the best of our knowledge).

2.1 First order lower bounds

As far as we are aware, there are two principal approaches in the literature which provide suitable estimates for the first order lower bound (1.2) and we will briefly describe these before discussing alternative approaches. The approach taken to estimate the gradient Lipschitz constant in Fowkes et al. (2013) was to bound the norm of the Hessian over a suitable domain using interval arithmetic. Evtushenko and Posypkin (2011, 2013), amongst others, replace the negative Lipschitz constant by a lower bound on the spectrum of the Hessian, $\lambda_{\min}(H(x))$, for x in some interval, which they claim yields a more accurate estimate. They approximate $\lambda_{\min}(H(x))$ using Gershgorin's Theorem, but other approximations to $\lambda_{\min}(H(x))$, for x in some domain, have been proposed in the literature. Floudas (1999, Section 12.4) provides a useful summary of such approximations to convexify the objective function in the context of his branch and bound algorithm. In this section, we show that some of the estimates from Floudas (1999) are more accurate than the Lipschitz constant estimates considered in Fowkes et al. (2013) and estimates using Gershgorin's Theorem in Evtushenko and Posypkin (2013).

We assume the following about problem (1.1) throughout this section:

AF 1. *The objective function $f\colon \mathcal{C} \to \mathbb{R}$ is twice continuously differentiable, where $\mathcal{C} \subset \mathbb{R}^n$ is a sufficiently large open set containing the convex, compact domain $\mathcal{D}$.*[1]

Let us start by showing why lower bounds on $\lambda_{\min}(H(x))$ can be used in place of the gradient Lipschitz constant $-L_g$ in (1.2). To this end, define for some compact domain $\mathcal{B}$

$$\lambda_{\min}^{\mathcal{B}}(H) := \min_{\xi\in\mathcal{B}} \lambda_{\min}(H(\xi)). \tag{2.1}$$

Lemma 2.1 (Evtushenko and Posypkin, 2013)**.** *Let AF 1 hold. Suppose $\mathcal{B} \subset \mathcal{C}$ is a convex, compact subdomain and $x_{\mathcal{B}} \in \mathcal{B}$.*[2] *Then, for any $x \in \mathcal{B}$ we have*

$$f(x) \geq f(x_{\mathcal{B}}) + (x - x_{\mathcal{B}})^T g(x_{\mathcal{B}}) + \frac{\lambda_{\min}^{\mathcal{B}}(H)}{2}\|x - x_{\mathcal{B}}\|_2^2. \tag{2.2}$$

[1] Note that we need a larger set here as the balls in our overlapping covering extend outside the domain during the initial subdivisions.

[2] Note that $\mathcal{B}$ does not need to be convex provided all line segments from $x_{\mathcal{B}}$ to x are contained in $\mathcal{B}$, i.e. if $\mathcal{B}$ is star-convex with star-centre $x_{\mathcal{B}}$.

Proof: For all $x, x_{\mathcal{B}} \in \mathcal{B}$ and some $\xi(x) \in \mathcal{B}$ the first order Taylor expansion with the Lagrange form for the remainder gives

$$\begin{aligned} f(x) &= f(x_{\mathcal{B}}) + (x - x_{\mathcal{B}})^T g(x_{\mathcal{B}}) + \frac{1}{2}(x - x_{\mathcal{B}})^T H(\xi)(x - x_{\mathcal{B}}) \\ &= f(x_{\mathcal{B}}) + (x - x_{\mathcal{B}})^T g(x_{\mathcal{B}}) + \frac{1}{2}\frac{(x - x_{\mathcal{B}})^T H(\xi)(x - x_{\mathcal{B}})}{(x - x_{\mathcal{B}})^T (x - x_{\mathcal{B}})}(x - x_{\mathcal{B}})^T (x - x_{\mathcal{B}}) \\ &\geq f(x_{\mathcal{B}}) + (x - x_{\mathcal{B}})^T g(x_{\mathcal{B}}) + \frac{\lambda_{\min}(H(\xi))}{2}\|x - x_{\mathcal{B}}\|_2^2 \\ &\geq f(x_{\mathcal{B}}) + (x - x_{\mathcal{B}})^T g(x_{\mathcal{B}}) + \frac{\lambda_{\min}^{\mathcal{B}}(H)}{2}\|x - x_{\mathcal{B}}\|_2^2 \end{aligned}$$

where the last two inequalities follow from the fact that the Rayleigh quotient reaches its minimum at the smallest eigenvalue and from (2.1), respectively. □

We can therefore use any lower bound on $\lambda_{\min}^{\mathcal{B}}(H)$ in place of $-L_g(\mathcal{B})$ in (1.2). In particular, we consider the following possible lower bounds on $\lambda_{\min}^{\mathcal{B}}(H)$ from Floudas (1999, Section 12.4), which all require the following bounds on the Hessian.

Definition 2.1. *Let AF 1 hold. Let $h_{ij}(\xi)$ denote the elements of the Hessian matrix $H(\xi)$ of f. Furthermore, let $\underline{H} = (\underline{h}_{ij})_{1\leq i,j\leq n}$, $\overline{H} = (\overline{h}_{ij})_{1\leq i,j\leq n}$ be such that for all $i, j = 1, \dots, n$*

$$\underline{h}_{ij} \leq h_{ij}(\xi) \leq \overline{h}_{ij} \tag{2.3}$$

for all ξ in a convex, compact subdomain $\mathcal{B}$.

Such elementwise lower and upper bounds (2.3) can be obtained, for example, using interval arithmetic.

Theorem 2.2 (Floudas, 1999)**.** *Let AF 1 hold. Given the elementwise bounds $\underline{h}_{ij}, \overline{h}_{ij}$ and corresponding matrices $\underline{H}, \overline{H}$ in* (2.3)*, the following lower bounds for $\lambda_{\min}^{\mathcal{B}}(H)$ in the bound* (2.2) *hold:*

i) Gershgorin's Theorem (**Ger**)*:*

$$\lambda_{\min}^{\mathcal{B}}(H) \geq \min_i \left[\underline{h}_{ii} - \sum_{j\neq i} \max\left\{|\underline{h}_{ij}|, |\overline{h}_{ij}|\right\}\right] \tag{2.4}$$

ii) E-Matrix Diagonal (**Ediag**)*:*

$$\lambda_{\min}^{\mathcal{B}}(H) \geq \lambda_{\min}(H_M) - \rho(\Delta H) \tag{2.5}$$

where $\lambda_{\min}(H_M)$ denotes the smallest eigenvalue of the midpoint matrix $H_M := \frac{\overline{H}+\underline{H}}{2}$ and $\rho(\Delta H)$ the spectral radius of the radius matrix $\Delta H := \frac{\overline{H}-\underline{H}}{2}$.

iii) E-Matrix Zero (**E0**)*:*

$$\lambda_{\min}^{\mathcal{B}}(H) \geq \lambda_{\min}(\widetilde{H_M}) - \rho(\widetilde{\Delta H}) \tag{2.6}$$

where the modified radius matrix $\widetilde{\Delta H}$ is ΔH with zeros on the diagonal and the modified midpoint matrix $\widetilde{H_M}$ is H_M with $\underline{h}_{ii}$ on the diagonal.

iv) Lower Bounding Hessian (**lbH**)*:*

$$\lambda_{\min}^{\mathcal{B}}(H) \geq \lambda_{\min}(L) \tag{2.7}$$

where the lower bounding Hessian $L = (l_{ij})$ is defined as

$$l_{ij} = \begin{cases} \underline{h}_{ii} + \sum_{k \neq i} \frac{\underline{h}_{ik} - \overline{h}_{ik}}{2} & \text{if } i = j \\ \frac{\underline{h}_{ij} + \overline{h}_{ij}}{2} & \text{if } i \neq j \end{cases}$$

v) Hertz's Method (**Hz**)*:*

$$\lambda_{\min}^{\mathcal{B}}(H) = \min_k \{\lambda_{\min}(H_k)\} \tag{2.8}$$

where the vertex matrices H_k are defined as follows: Let $x \in \mathbb{R}^n$, then there are 2^{n-1} possible combinations for the signs of the $x_i x_j$ products ($i \neq j$). For the k-th such combination, define the vertex matrix $H_k = (h_{ij}^k)$ where

$$h_{ij}^k = \begin{cases} \underline{h}_{ii} & \text{if } i = j, \\ \underline{h}_{ij} & \text{if } x_i x_j \geq 0, i \neq j \\ \overline{h}_{ij} & \text{if } x_i x_j < 0, i \neq j \end{cases}$$

Proof: See Floudas (1999, Section 12.4) for proofs of the above lower bounds (2.4)–(2.8). □

We also consider a lower bound on the best $-L_g(\mathcal{B})$ in (1.2), given in the following Theorem.

Theorem 2.3 (Norm of the Hessian (**Norm**)). *Let AF 1 hold. Suppose $\mathcal{B} \subset \mathcal{C}$ is a convex, compact subdomain and $x_{\mathcal{B}} \in \mathcal{B}$. Then, for any $x \in \mathcal{B}$, the first order lower bound* (1.2) *holds.*[3] *Furthermore, a lower bound for the best $-L_g(\mathcal{B})$ in* (1.2) *is given by*

$$-L_g(\mathcal{B}) \geq -\sqrt{\sum_{ij} \max\left\{|\underline{h}_{ij}|, |\overline{h}_{ij}|\right\}^2} \tag{2.9}$$

where the elementwise bounds $\underline{h}_{ij}, \overline{h}_{ij}$ are defined in (2.3).

Proof: (1.2) is a well-known consequence of first order Taylor expansions; see for example Theorem 3.1.4 in Conn, Gould and Toint (2000). Note that $\|M\|_2 \leq \|M\|_F$ for any matrix M. We have from Taylor's theorem to first order and Cauchy-Schwarz that for any $x, y \in \mathcal{B}$

$$\begin{aligned} \|g(x) - g(y)\|_2 &\leq \left\| \int_0^1 H\left(y + \tau(x-y)\right)(x-y) d\tau \right\|_2 \\ &\leq \max_{0 \leq \tau \leq 1} \|H\left(y + \tau(x-y)\right)\|_2 \|x-y\|_2 \\ &\leq \max_{0 \leq \tau \leq 1} \|H\left(y + \tau(x-y)\right)\|_F \|x-y\|_2 \\ &= \max_{0 \leq \tau \leq 1} \left(\sum_{ij} [H\left(y + \tau(x-y)\right)]_{ij}^2 \right)^{1/2} \|x-y\|_2 \\ &\leq \left(\sum_{ij} \max\left\{|\underline{h}_{ij}|^2, |\overline{h}_{ij}|^2\right\} \right)^{1/2} \|x-y\|_2 \\ &= \left(\sum_{ij} \max\left\{|\underline{h}_{ij}|, |\overline{h}_{ij}|\right\}^2 \right)^{1/2} \|x-y\|_2. \end{aligned}$$

[3] Note that if $\mathcal{B}$ is not assumed to be compact, then (1.2) still holds provided the gradient g is Lipschitz continuous on the convex subdomain $\mathcal{B}$ and $f \in C^1(\mathcal{B})$.

Thus the gradient g is Lipschitz continuous on a compact domain $\mathcal{B}$ with ℓ^2-norm Lipschitz constant $\sqrt{\sum_{ij} \max\{|\underline{h}_{ij}|, |\overline{h}_{ij}|\}^2}$. In particular, this means that for the best gradient Lipschitz constant $L_g(\mathcal{B})$, we have for all $x \in \mathcal{B}$

$$L_g(\mathcal{B}) \leq \sqrt{\sum_{ij} \max\left\{|\underline{h}_{ij}|, |\overline{h}_{ij}|\right\}^2}. \qquad \square$$

If we look at the computational cost of the estimation approaches given in (2.4)–(2.8) and (2.9) (and exclude the cost of calculating the Hessian bounds $\underline{h}_{ij}, \overline{h}_{ij}$) we can show that **Ger** is an $\mathcal{O}(n^2)$ method (i.e. it requires $\mathcal{O}(n^2)$ floating point operations). **Eidag**, **E0**, **lbH** require the calculation of one or two extreme eigenvalues and **Hz** requires 2^{n-1} leftmost eigenvalues. Assuming standard methods for calculating all the eigenvalues of a matrix, **Eidag**, **E0**, **lbH** would all be $\mathcal{O}(n^3)$ methods and **Hz** would be an $\mathcal{O}(2^{n-1}n^3)$ method. In practice, extreme eigenvalues of dense matrices are usually obtained in $\mathcal{O}(n^{2+v})$ flops, where $v < 1$, e.g. using the power method. Calculating **Norm** requires squaring n^2 entries and so is an $\mathcal{O}(n^2)$ method.

2.2 Second order lower bounds

In Section 2.1, we considered replacing the gradient Lipschitz constant in the first order lower bound (1.2) by an estimate of the smallest eigenvalue of the Hessian. In this section we will show that, to an extent, a similar approach is also possible for the second order lower bound (1.3) and we can replace the Hessian Lipschitz constant by an estimate of the smallest eigenvalue of the derivative tensor. Before we describe this in detail we need to introduce some tensor eigenvalue notation.

Let $T \in \mathbb{R}^{n\times n\times n}$ denote a third order tensor, which being a generalisation of a matrix to three indices, is a 3-dimensional array. As with matrices, t_{ijk} denotes the (i, j, k)-th component (i.e. element in the array) of the tensor T. Furthermore, a tensor T is called symmetric (sometimes supersymmetric) if $t_{\sigma(i)\sigma(j)\sigma(k)} = t_{ijk}$ for any permutation σ of the indices (i, j, k). This is the natural generalisation of a symmetric matrix to tensors. For a vector $x \in \mathbb{R}^n$, the multiplication of a tensor T three times on the right by x is denoted by

$$Tx^3 := \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} t_{ijk}x_ix_jx_k.$$

Let $\|T\|_F$ denote the Frobenius norm for the tensor T defined as

$$\|T\|_F^2 = \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n} t_{ijk}^2.$$

We have from Lim (2005) that the *multilinear Rayleigh quotient* for the ℓ^3-norm is given by

$$\frac{Tx^3}{\|x\|_3^3}$$

where $\|\cdot\|_3$ is the ℓ^3-norm for vectors. The ℓ^3-eigenvalues (or H-eigenvalues) of T are then defined as the stationary points of the multilinear Rayleigh quotient. In particular, this means that the smallest ℓ^3-eigenvalue of T, $\lambda^{\ell^3}_{\min}(T)$ is given by[4]

$$\lambda^{\ell^3}_{\min}(T) = \min_{x\neq 0} \frac{Tx^3}{\|x\|_3^3}. \tag{2.10}$$

We assume the following about problem (1.1) throughout this section:

AF 2. *The objective function $f\colon \mathcal{C} \to \mathbb{R}$ is thrice continuously differentiable, where $\mathcal{C} \subset \mathbb{R}^n$ is a sufficiently large open set containing the convex, compact domain $\mathcal{D}$.*

We are now in a position to show why lower bounds on the spectrum of the derivative tensor can be used in place of the Hessian Lipschitz constant L_H in (1.3). To this end, let

$$T(x) := \nabla_{xxx} f(x)$$

denote the third order derivative tensor of $f(x)$ and note that it is symmetric by construction. Define for some compact domain $\mathcal{B}$

$$\lambda^{\ell^3,\mathcal{B}}_{\min}(T) := \min_{\xi\in\mathcal{B}} \lambda^{\ell^3}_{\min}(T(\xi)). \tag{2.11}$$

Lemma 2.4. *Let AF 2 hold. Suppose $\mathcal{B} \subset \mathcal{C}$ is a convex, compact subdomain and $x_{\mathcal{B}} \in \mathcal{B}$. Then, for any $x \in \mathcal{B}$ we have*

$$\begin{aligned} f(x) \geq f(x_{\mathcal{B}}) + (x - x_{\mathcal{B}})^T g(x_{\mathcal{B}}) + \frac{1}{2}(x - x_{\mathcal{B}})^T H(x_{\mathcal{B}})(x - x_{\mathcal{B}}) \\ + \begin{cases} \frac{\lambda^{\ell^3,\mathcal{B}}_{\min}(T)}{6}\|x - x_{\mathcal{B}}\|_2^3 & \text{if } \lambda^{\ell^3,\mathcal{B}}_{\min}(T) \leq 0, \\ \frac{\lambda^{\ell^3,\mathcal{B}}_{\min}(T)}{6} n^{-1/2}\|x - x_{\mathcal{B}}\|_2^3 & \text{if } \lambda^{\ell^3,\mathcal{B}}_{\min}(T) > 0. \end{cases} \end{aligned} \tag{2.12}$$

Proof: First of all, in order to use ℓ^3-eigenvalues in (2.12) we require relations between the ℓ^2 and ℓ^3 vector norms. It is a standard result that for any $p > r > 0$

$$\|x\|_p \leq \|x\|_r \leq n^{(1/r-1/p)}\|x\|_p$$

for any $x \in \mathbb{R}^n$ and in particular this means that

$$\begin{aligned} \|x\|_3 &\geq n^{-1/6}\|x\|_2, \\ \|x\|_3 &\leq \|x\|_2 \end{aligned} \tag{2.13}$$

for any $x \in \mathbb{R}^n$.

Now, for $x = x_{\mathcal{B}}$ the claim in the theorem is trivial, so w.l.o.g. assume $x \neq x_{\mathcal{B}}$. Then for all $x, x_{\mathcal{B}} \in \mathcal{B}$ and some $\xi(x) \in \mathcal{B}$, the second order Taylor expansion with the Lagrange form for

[4] Note that one can instead use the alternative definition of ℓ^2-eigenvalues that are the stationary points of the multilinear Rayleigh quotient for the ℓ^2-norm, $Tx^3/\|x\|_2^3$ (Lim, 2005) and then the smallest ℓ^2-eigenvalue of T, $\lambda^{\ell^2}_{\min}(T)$ would be given by $\lambda^{\ell^2}_{\min}(T) = \min_{x\neq 0} Tx^3/\|x\|_2^3$. However, we will not use ℓ^2-eigenvalues here for reasons that will become clear later.

the remainder gives

$$\begin{aligned}
f(x) &= f(x_{\mathcal{B}}) + (x - x_{\mathcal{B}})^T g(x_{\mathcal{B}}) + \frac{1}{2}(x - x_{\mathcal{B}})^T H(x_{\mathcal{B}})(x - x_{\mathcal{B}}) + \frac{1}{6}T(\xi)(x - x_{\mathcal{B}})^3 \\
&= f(x_{\mathcal{B}}) + (x - x_{\mathcal{B}})^T g(x_{\mathcal{B}}) + \frac{1}{2}(x - x_{\mathcal{B}})^T H(x_{\mathcal{B}})(x - x_{\mathcal{B}}) + \frac{1}{6}\frac{T(\xi)(x - x_{\mathcal{B}})^3}{\|x - x_{\mathcal{B}}\|_3^3}\|x - x_{\mathcal{B}}\|_3^3 \\
&\geq f(x_{\mathcal{B}}) + (x - x_{\mathcal{B}})^T g(x_{\mathcal{B}}) + \frac{1}{2}(x - x_{\mathcal{B}})^T H(x_{\mathcal{B}})(x - x_{\mathcal{B}}) + \frac{\lambda_{\min}^{\ell^3}(T(\xi))}{6}\|x - x_{\mathcal{B}}\|_3^3 \\
&\geq f(x_{\mathcal{B}}) + (x - x_{\mathcal{B}})^T g(x_{\mathcal{B}}) + \frac{1}{2}(x - x_{\mathcal{B}})^T H(x_{\mathcal{B}})(x - x_{\mathcal{B}}) + \frac{\lambda_{\min}^{\ell^3,\mathcal{B}}(T)}{6}\|x - x_{\mathcal{B}}\|_3^3 \\
&\geq f(x_{\mathcal{B}}) + (x - x_{\mathcal{B}})^T g(x_{\mathcal{B}}) + \frac{1}{2}(x - x_{\mathcal{B}})^T H(x_{\mathcal{B}})(x - x_{\mathcal{B}}) \\
&\quad + \begin{cases} \frac{\lambda_{\min}^{\ell^3,\mathcal{B}}(T)}{6}\|x - x_{\mathcal{B}}\|_2^3 & \text{if } \lambda_{\min}^{\ell^3,\mathcal{B}}(T) \leq 0, \\ \frac{\lambda_{\min}^{\ell^3,\mathcal{B}}(T}{6} n^{-1/2}\|x - x_{\mathcal{B}}\|_2^3 & \text{if } \lambda_{\min}^{\ell^3,\mathcal{B}}(T) > 0 \end{cases}
\end{aligned}$$

using (2.10), (2.11) and (2.13) respectively. □

We can therefore use any (suitably scaled) lower bound on $\lambda_{\min}^{\ell^3,\mathcal{B}}(T)$ in place of $-L_H(\mathcal{B})$ in (1.3).[5] In Section 2.1, Theorem 2.2 (2.4)–(2.8) and Theorem 2.3 (2.9) give several different approaches to obtain lower bounds on the smallest eigenvalue in the case of a Hessian matrix. We will now show which of these estimation approaches generalises to the case of a third order derivative tensor. While there are ℓ^3-eigenvalue algorithms that are guaranteed to converge to the smallest eigenvalue, these are only applicable to tensors with non-negative (or equivalently non-positive) entries (Kolda and Mayo, 2011). Unfortunately, the tensor generalisations of the matrices required for the lower bounding strategies presented in (2.5)–(2.8), namely the E-matrix, Lower bounding Hessian and Hertz method have both positive and negative entries in general. However, the generalisation of Gershgorin's Theorem (Qi, 2005) does not require an eigenvalue algorithm and we can therefore generalise Theorem 2.2 (2.4) to tensors. We first need the following definition before we can give the generalised theorem.

Definition 2.2. *Let AF 2 hold. Let $t_{ijk}(\xi)$ denote the elements of the third order derivative tensor $T(\xi)$. Furthermore, let $\underline{T} = (\underline{t}_{ijk})_{1\leq i,j,k\leq n}$, $\overline{T} = (\overline{t}_{ijk})_{1\leq i,j,k\leq n}$ be such that for all $i, j, k = 1, \ldots, n$*

$$\underline{t}_{ijk} \leq t_{ijk}(\xi) \leq \overline{t}_{ijk} \tag{2.14}$$

for all ξ in a convex, compact subdomain $\mathcal{B}$.

Once again, the elementwise lower and upper bounds (2.14) can be obtained using interval arithmetic.

Theorem 2.5 (Gershgorin's Theorem for the derivative Tensor (**Ger T**))**.** *Let AF 2 hold. Assuming the elementwise bounds $\underline{t}_{ijk}, \overline{t}_{ijk}$ in* (2.14)*, $\lambda_{\min}^{\ell^3,\mathcal{B}}(T)$ in* (2.12) *can be bounded below by*

$$\lambda_{\min}^{\ell^3,\mathcal{B}}(T) \geq \min_i \left[\underline{t}_{iii} - \sum_{k\neq j\neq i} \max\left\{|\underline{t}_{ijk}|, |\overline{t}_{ijk}|\right\}\right]. \tag{2.15}$$

[5] Note that an analogous result holds for ℓ^2-eigenvalues. Unfortunately, to the best of our knowledge, there are no known eigenvalue algorithms that are guaranteed to converge to the smallest ℓ^2-eigenvalue but it is possible to use generalisations of the power method using multiple starting points (Kolda and Mayo, 2011; Zhang, Qi and Ye, 2012). However, this is not reliable as (1.3) requires a bound on the smallest eigenvalue and using multiple starting points does not guarantee this. Furthermore, there is no generalisation of Gershgorin's Theorem for ℓ^2-eigenvalues, which is what we propose next for ℓ^3-eigenvalues.

Proof: Let $\xi \in \mathcal{B}$ be arbitrary. We have from Qi (2005) that Gershgorin's Theorem for tensors applied to the third order derivative tensor $T(\xi)$ gives

$$\begin{aligned}\lambda_{\min}^{\ell^3}(T(\xi)) &= \min_i \left[\underline{t}_{iii} - \sum_{k\neq j\neq i} |t_{ijk}(\xi)| \right] \\ &\geq \min_i \left[\underline{t}_{iii} - \sum_{k\neq j\neq i} \max\left\{ |\underline{t}_{ijk}|, |\overline{t}_{ijk}| \right\} \right]\end{aligned}$$

for any $\xi \in \mathcal{B}$. As $\lambda_{\min}^{\ell^3,\mathcal{B}}(T) = \min_{\xi\in\mathcal{B}} \lambda_{\min}^{\ell^3}(T(\xi))$ from (2.11), the result follows. □

Additionally, we also have a bound on the Hessian Lipschitz constant in (1.3), an extension of the **Norm** bound (2.9) from Theorem 2.3.[6]

Theorem 2.6 (Norm of the derivative tensor (**Norm T**))**.** *Let AF 2 hold. Suppose $\mathcal{B} \subset \mathcal{C}$ is a convex, compact subdomain and $x_{\mathcal{B}} \in \mathcal{B}$. Then, for any $x \in \mathcal{B}$, the second order lower bound* (1.3) *holds.*[7] *Furthermore, a lower bound for the best $-L_H(\mathcal{B})$ in* (1.3) *is given by*

$$-L_H(\mathcal{B}) \geq -\sqrt{\sum_{ijk} \max\left\{ |\underline{t}_{ijk}|, |\overline{t}_{ijk}| \right\}^2} \tag{2.16}$$

where the elementwise bounds $\underline{t}_{ijk}, \overline{t}_{ijk}$ are defined as in (2.14)*.*

Proof: (1.3) is a well-known consequence of second order Taylor expansions; see for example Theorem 3.1.5 in Conn et al. (2000). Note that as in the matrix case, $\|T\|_2 \leq \|T\|_F$ for any tensor T (see Lemma 6.1 in Fowkes et al., 2013, for a proof). We have from Taylor's theorem to first order and Cauchy-Schwarz that for any $x, y \in \mathcal{B}$

$$\begin{aligned}\|H(x) - H(y)\|_2 &\leq \left\| \int_0^1 T\left(y + \tau(x-y)\right)(x-y)d\tau \right\|_2 \\ &\leq \max_{0\leq\tau\leq1} \|T\left(y+\tau(x-y)\right)\|_F \|x-y\|_2 \\ &= \max_{0\leq\tau\leq1} \left(\sum_{ijk} \left[T\left(y+\tau(x-y)\right)\right]_{ijk}^2 \right)^{1/2} \|x-y\|_2 \\ &\leq \left(\sum_{ijk} \max\left\{ |\underline{t}_{ijk}|, |\overline{t}_{ijk}| \right\}^2 \right)^{1/2} \|x-y\|_2.\end{aligned}$$

Thus the Hessian H is Lipschitz continuous on a compact domain $\mathcal{B}$ with ℓ^2-norm Lipschitz constant $\sqrt{\sum_{ijk} \max\{|\underline{t}_{ijk}|, |\overline{t}_{ijk}|\}^2}$. In particular, this means that for the best Hessian Lipschitz constant $L_H(\mathcal{B})$, we have

$$L_H(\mathcal{B}) \leq \sqrt{\sum_{ijk} \max\left\{ |\underline{t}_{ijk}|, |\overline{t}_{ijk}| \right\}^2}.$$ □

Looking at the computational cost of the second order estimation approaches **Ger T** and **Norm T** given in (2.15), (2.16) (and excluding the cost of calculating the tensor bounds $\underline{t}_{ijk}, \overline{t}_{ijk}$) we can see that they are $\mathcal{O}(n^3)$ methods since each requires summing or squaring n^3 elements.

[6]Note that this bound appears in Section 6.2 of Fowkes et al. (2013) but it is incorrect there.

[7]Note that if $\mathcal{B}$ is not assumed to be compact, then (1.3) still holds provided the Hessian H is Lipschitz continuous on the convex subdomain $\mathcal{B}$ and $f \in C^2(\mathcal{B})$.

2.3 Numerical results

The overlapping branch and bound algorithm (oBB), namely Algorithm 2.1 from Fowkes et al. (2013), is especially suited to testing the first and second order estimation approaches from Section 2.1 and Section 2.2. As oBB is exactly Algorithm 3.1 from Section 3.1 in which all the worker calculations are performed by the master, we will only briefly outline it here and postpone a detailed description to Section 3.1. The algorithm is structured in much the same way as most standard branch and bound algorithms: It starts with a ball covering the domain and recursively subdivides it into overlapping balls, bounding each ball and discarding balls that cannot contain a global minimiser until the global minimum is located. The branching subdivides each ball into 3^n half-sized overlapping sub-balls that cover the original ball and have a fixed amount of overlap. The bounding uses the first and second order Lipschitz-based lower bounds (1.2), (1.3) but can also accommodate the eigenvalue-based lower bounds (2.2), (2.12) all of which it can solve in polynomial due to its use of overlapping balls.

We test the first and second order estimation approaches on test sets of

1) Random polynomials
2) Random radial basis functions (RBFs)

which we will describe in turn. The aim of the numerical experiments is to test which estimation approach gives the best oBB performance in terms of runtime. This gives an indirect indication of the accuracy of the estimation approach.

Random Polynomials (Evtushenko and Posypkin, 2013) This is a collection of bound constrained global optimization problems with polynomial objective functions and randomly generated coefficients. The polynomial objective functions used are of the form

$$f(x) = \sum_{i=1}^{n} 10x_i^m + \sum_{p\in\mathcal{P}} a_p x_{i_1}^{p_1} \dots x_{i_n}^{p_n} \tag{2.17}$$

where m is an even polynomial degree and $\mathcal{P} = \{(p_1, \dots, p_n) : p_i \in \mathbb{Z}_+, \sum_{i=1}^n p_i \le m-1\}$ is the set of n-tuples corresponding to the powers of the monomials. The randomly generated coefficients a_p are uniformly distributed in $[0, 10]$. Let $|\mathcal{P}| = \binom{m-1+n}{n}$ be the number of n-tuples in $\mathcal{P}$. Evtushenko and Posypkin (2013) then observe that for an even m, the global optimiser lies in the box $[-|\mathcal{P}|, |\mathcal{P}|]^n$ and this is therefore taken to be the search domain. Following Evtushenko and Posypkin (2013), we set the following following values for m and n in (2.17):

- Series 1: $n = 3, m = 4$
- Series 2: $n = 3, m = 6$ (2.18)
- Series 3: $n = 4, m = 4$

and test 10 realisations of (2.17) for each series. The bounds required by our estimation approaches, namely $\underline{h}_{ij}, \overline{h}_{ij}$ in (2.3) on the Hessians and $\underline{t}_{ijk}, \overline{t}_{ijk}$ in (2.14) on the derivative tensors of the polynomials are calculated using our own implementation of standard interval arithmetic (see e.g. Section 11 of Neumaier, 2004).

Random RBFs This is a RBF test set similar to the one above for random polynomials. We will use cubic spline RBF objective functions of the form

$$f(x) = \mu_0 + \sum_{i=1}^{n} \mu_i x_i + \sum_{j=1}^{N(m,n)} \lambda_j \|x - x^j\|_2^3 \tag{2.19}$$

where μ_i, λ_j are coefficients of the linear and radial terms respectively and $N(m,n) := \binom{m-1+n}{n}$ is a given number of centres $x^j \in \mathbb{R}^n$. As before, we let the coefficients μ_i, λ_j be random, that is uniformly distributed in the interval $[0, 10]$. We will use the same values for m and n in (2.19) as those for random polynomials given in (2.18) and test 10 realisations of (2.19) for each series. Note that we choose $N(m,n) = \binom{m-1+n}{n}$ so that we have the same number of terms in the RBFs as in the polynomials above (up to a constant). We also take the box $[-N(m,n), N(m,n)]^n$ as the search domain so that the search regions for the RBFs are the same as for the polynomials. The bounds $\underline{h}_{ij}, \overline{h}_{ij}$ in (2.3) on the Hessians and $\underline{t}_{ijk}, \overline{t}_{ijk}$ in (2.14) on the derivative tensors of the RBFs are calculated using a more accurate interval arithmetic type approach (see Section 6.2 of Fowkes et al., 2013, for details).

As we are interested in the relative performance of the first and second order estimation approaches, we will look at runtime performance profiles for both the random polynomial and RBF test sets described above (for the definition of performance profiles, see Dolan and Moré, 2002; see also Strongin and Sergeyev, 2000, p. 203, for the similar notion of operating characteristics first proposed by Grishagin, 1978). To this end, we ran a Python-based serial implementation of oBB to an absolute tolerance of 10^{-6} for the global minimum using one of the estimation approaches given in (2.4)–(2.8) and (2.9). If the algorithm did not complete a run in 24 hours then that run was considered a failure. The hardware used was part of the ECDF Eddie cluster using a single 2.4GHz Intel Xeon E5645 processor core with 2GB of RAM for each random polynomial or RBF realisation.

2.3.1 First order lower bounds

Let us first consider the numerical performance of the first order estimation approaches **Ger**, **Eidag**, **E0**, **lbH**, **Hz** and **Norm** given in (2.4)–(2.8) and (2.9). Figure 2.1 below shows performance profiles of the total runtime for the first order estimation approaches on the random polynomials. For clarity we consider two ranges of the performance ratio so we can clearly see

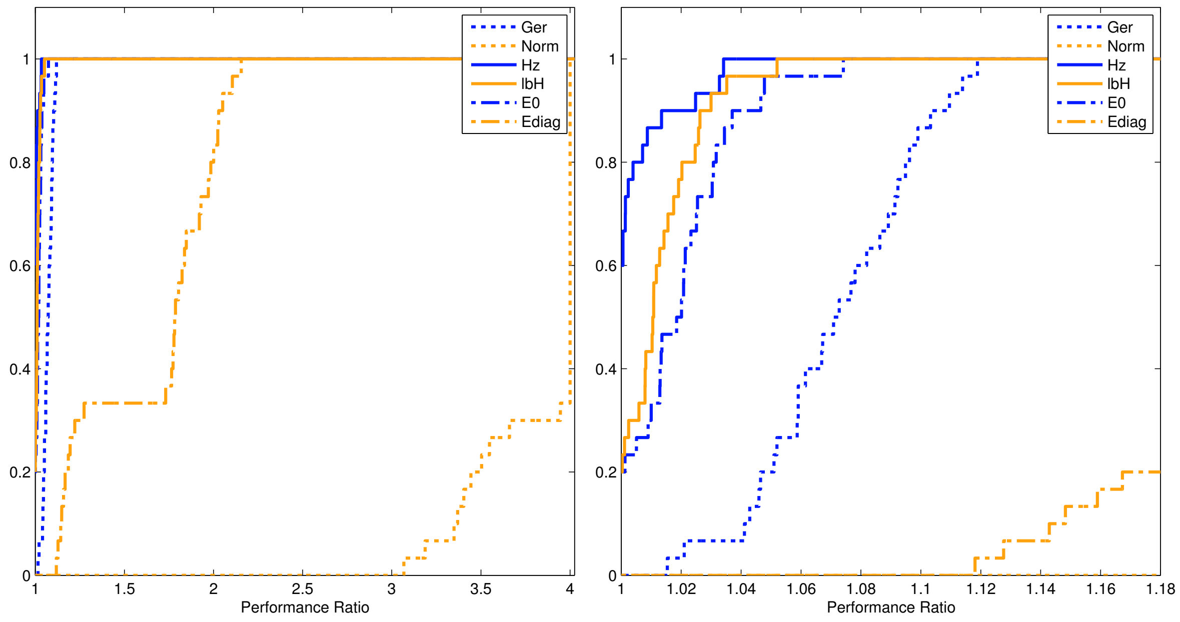

Figure 2.1: *Random polynomial runtime performance profiles (left) with a close up of the left-hand figure (right) for the first order estimation approaches given in* (2.4)–(2.8) *and* (2.9).

the poorer estimates (**Norm** and **Ediag**) in the left-hand side of Figure 2.1 and the better estimates (**Hz**, **lbH**, **E0** and **Ger**) in the right-hand side of Figure 2.1, which is a close-up of the left-hand figure. From the left-hand side of Figure 2.1 we can see that **Norm** is by far the weakest approach, in fact the algorithm only finds the global minimum within 24 hours in a third of the problems tested. For all the other estimates, the global minimum is always found, although the **Ediag** approach also performs poorly. Looking at the better performing approaches in the right-hand side of Figure 2.1, we can see that **Hz** is the best, presumably because it always calculates exactly the smallest eigenvalue of the Hessian $H(\xi)$ for $\underline{H} \leq H(\xi) \leq \overline{H}$. However, this necessitates calculating the eigenvalues of 2^{n-1} matrices and while this is practical for two and three dimensional polynomials this will clearly be an issue in higher dimensions. With this in mind, the best approaches seem to be **lbH** and **E0** which perform similarly well, followed by **Ger** which does not appear to be quite as good but nonetheless still shows reasonable performance.

Figure 2.2 below shows performance profiles of the total runtime for the first order estimation approaches on the random RBFs. We can see from Figure 2.2 that the performance profiles for

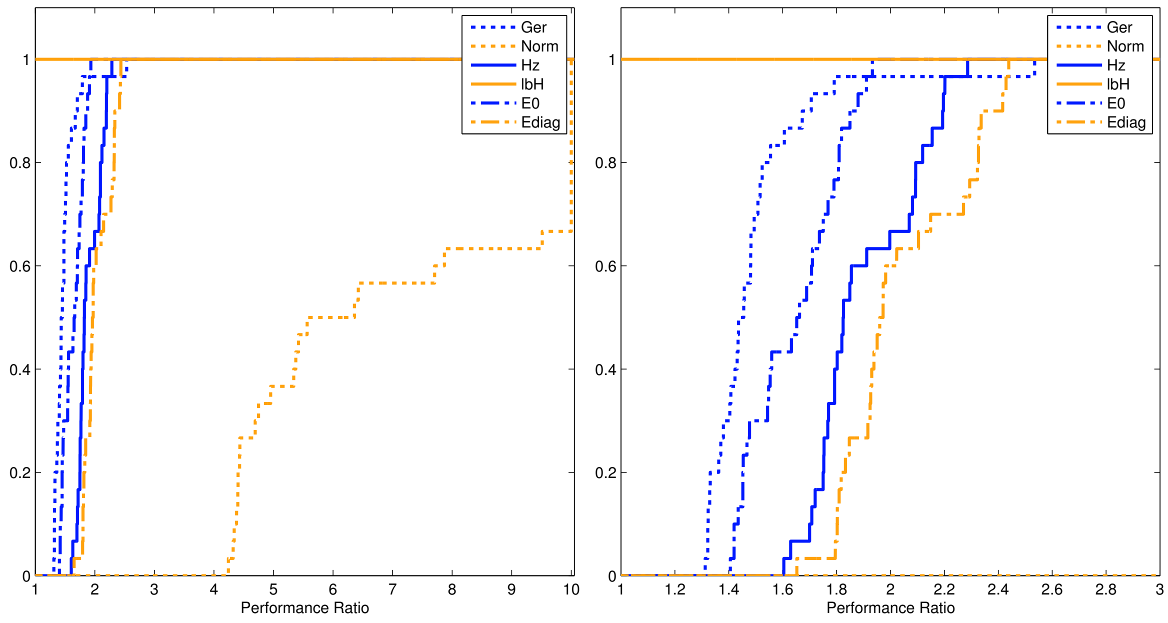

Figure 2.2: *Random RBF runtime performance profiles (left) with a close up of the left-hand figure (right) for the first order estimation approaches given in* (2.4)–(2.8) *and* (2.9).

random RBFs are very different from those for the random polynomials obtained above in Figure 2.1. In particular, **lbH** significantly outperforms all the other estimation approaches. **Ger**, **E0**, **Hz** and **Eidag** all perform similarly well and this is especially surprising as the **Eidag** approach showed poor performance on the random polynomials. The **Norm** approach is once again the weakest, although it performs somewhat better on the random RBFs.

It is evident from these numerical experiments that there is a need for several different estimation approaches as no single approach is superior. In particular, as the computational cost of the estimation approach is generally negligible compared to the cost of computing the bounds $\underline{h}_{ij}, \overline{h}_{ij}$, it is possible to have an adaptive algorithm that computes several estimates and uses the best one.

2.3.2 Second order lower bounds

We will now look at the numerical performance of the second order tensor Gershgorin and Norm approaches given in (2.15) and (2.16), respectively, on the random polynomial and RBF test sets. Figure 2.3 below shows performance profiles of the total runtime for the second order estimation approaches on the random polynomials and RBFs. We can see in the left-hand side

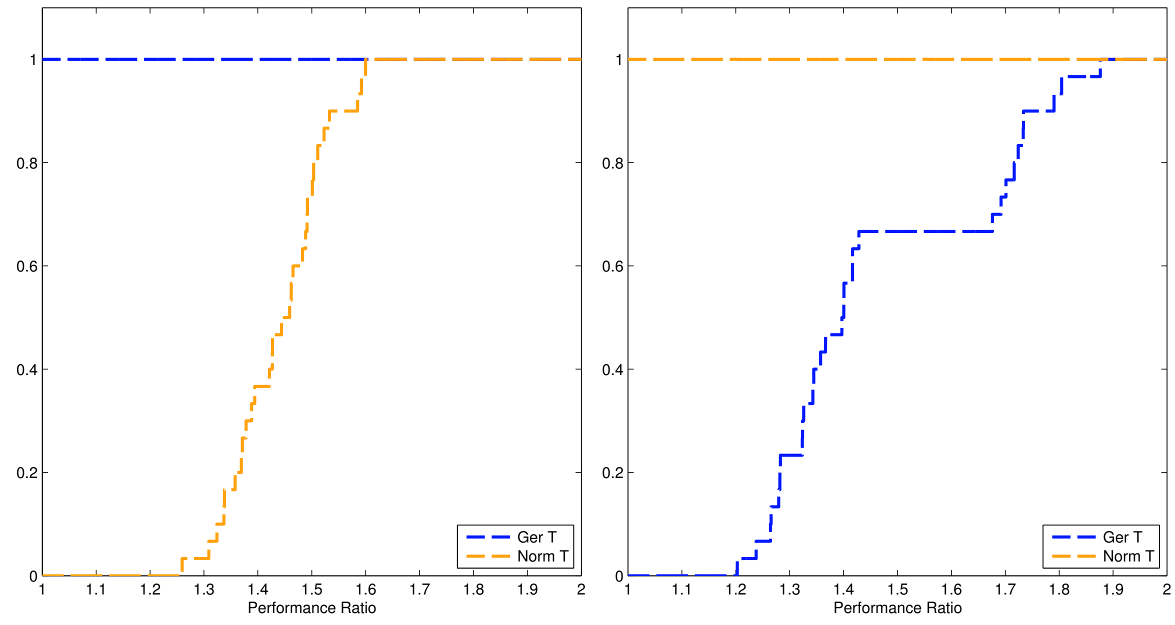

Figure 2.3: *Random Polynomial (left) and RBF (right) runtime performance profiles for the second order estimation approaches given in* (2.15) *and* (2.16)*, respectively.*

of Figure 2.3 that the Gershgorin based estimate consistently outperforms the tensor norm approach for random polynomials. Although the algorithm always finds the global minimum using these estimates, the Gershgorin estimate yields faster and more accurate second order lower bounds. The situation, however, is completely reversed for random RBFs as we can see from the performance profiles in the right-hand side of Figure 2.3. In this case the tensor norm based estimate outperforms the Gershgorin estimate and yields faster and more accurate bounds. Once again, this emphasises the need to compute several estimation approaches and use whichever is best.

2.3.3 Comparison of first versus second order bounds

Finally, to wrap up the discussion of finding better bounds, we compare both first and second order lower bounds in Figure 2.4 by recalculating performance profiles of the total runtime for both. One can clearly see from the left-hand side of Figure 2.4 that for random polynomials the second order lower bounds significantly outperform the first order ones, with the tensor Gershgorin approach clearly superior. This is perhaps not surprising as the second order lower bounds (2.12),(1.3) utilising second order derivative information are likely to be more accurate than the first order lower bounds (2.2),(1.2) which can only make use of first order information. However, the situation is not quite so simple for the random RBFs as we can see from the right-hand side of Figure 2.4 where the first order lower bounding Hessian estimation approach actually outperforms the second order tensor Gershgorin approach. This is encouraging since it shows that in some cases first order bounds which are significantly cheaper to compute can

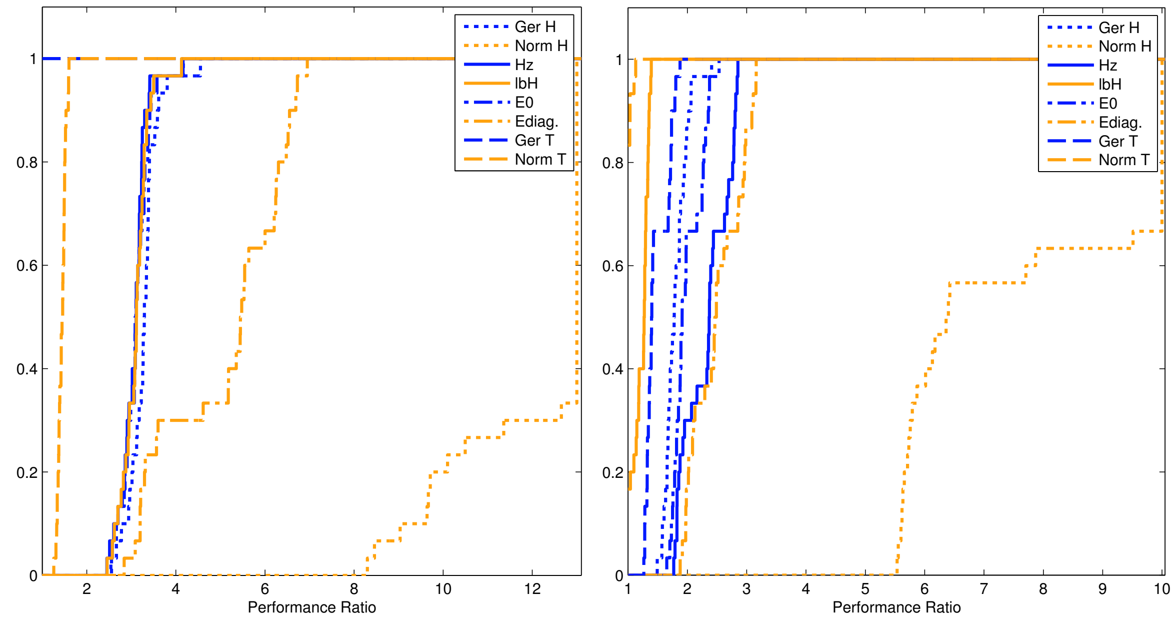

Figure 2.4: *Random Polynomial (left) and RBF (right) runtime performance profiles for the first order (dotted, solid and dash-dotted lines) and second order (dashed lines) estimation approaches.*

be competitive with the more expensive second order bounds. The tensor norm approach is, however, evidently the best for random RBFs, outperforming all other approaches.

In conclusion, there is no single first or second order bound that is clearly superior across different objective functions. There are even instances where first order bounds outperform second order ones. The best strategy in our opinion is therefore to implement all the first and second order bounds within a branch and bound algorithm and adaptively choose which is best. For example, for the first few subproblems all possible bounds could be computed and the best two or three used throughout the rest of the computation. As the computational cost of calculating the lower bounds is negligible compared to the cost of calculating the bounds $\underline{h}_{ij}, \overline{h}_{ij}$ or $\underline{t}_{ijk}, \overline{t}_{ijk}$, such adaptive strategies are feasible and indeed recommended to maximise performance.

3 Parallelising Overlapping Branch and Bound (oBB)

In Section 2 we considered improving the bounding in Lipschitz based branch and bound global optimization algorithms and tested our findings using an implementation of oBB, Algorithm 2.1 from Fowkes et al. (2013). In this section we will consider this implementation of oBB and show how it can be speeded up through parallelism. As mentioned in the introduction, there are two main approaches to parallelising branch and bound algorithms: data parallel, namely performing the bounding operations in parallel and task parallel, traversing the branch and bound tree in parallel (Gendron and Crainic, 1994; Crainic et al., 2006). We consider applying these in turn to oBB in the following sections.

3.1 Data Parallelism: Bounds in Parallel

The idea behind data parallelism of a branch and bound algorithm is to share the computational burden of calculating the bounds amongst many processor cores. Our implementation of this is a very straightforward master/worker approach. The master processor core runs the entire algorithm except for the calculations involved in obtaining bounds on each subdomain, which are (roughly) evenly divided amongst itself and the worker processors. It is immediately obvious that this type of parallelism will only be successful if there are many bounding calculations that can be performed independently at the same time and if these calculations are relatively expensive compared to the rest of the algorithm.

The oBB algorithm uses Euclidean balls as the subdomains since this allows the lower bounding subproblem (1.4) to be solved in polynomial time. However, this comes at a cost, as the rigorous variant of oBB requires that each ball is split into 3^n sub-balls which can very quickly become prohibitively large as the dimension n increases. Nevertheless, this lends itself well to data parallelism since at each step of the algorithm we have to bound around 3^n balls and these bounding operations can of course be done in parallel. This is the basis of the data parallel version of oBB, given below as Algorithm 3.1.

The algorithm solves (1.4) to obtain a lower bound $\underline{f}(\mathcal{B})$ on the objective function f over the subdomain $\mathcal{B}$, that is

$$\underline{f}(\mathcal{B}) := \min_{x \in \mathcal{B}} l_{\mathcal{B}}(x) \tag{3.1}$$

where $l_{\mathcal{B}}(x)$ can be any of the first or second order lower bounds given in (1.2), (2.2) and (1.3), (2.12), respectively. The upper bound $\overline{f}(\mathcal{B})$ on f over $\mathcal{B}$ is simply the objective function f evaluated at a feasible point $x_F \in \mathcal{B}$, that is

$$\overline{f}(\mathcal{B}) := f(x_F). \tag{3.2}$$

It is important to note that if we run this algorithm on one master processor core, we recover the serial version of oBB.

Algorithm 3.1. Data Parallel Branch and Bound Algorithm
Master Processor

```
0. Initialisation:
    (a) Set $k = 0$ and $t_{max}$ to be the maximum runtime.
    (b) Let $\mathcal{B}_0$ be a ball with centre $x_{\mathcal{B}} \in \mathcal{D}$ of sufficiently large radius to cover $\mathcal{D}$.
    (c) Let $\mathcal{L}_0 = \{\mathcal{B}_0\}$ be the initial set of balls.
    (d) Let $U_0 = \overline{f}(\mathcal{B}_0)$ be the initial upper bound for $\min_{x \in \mathcal{B}_0} f(x)$.
    (e) Let $L_0 = \underline{f}(\mathcal{B}_0)$ be the initial lower bound for $\min_{x \in \mathcal{B}_0} f(x)$.
1. While $U_k - L_k > \varepsilon$ and the runtime $< t_{max}$, repeat the following procedure:
    (a) Pruning: Remove from $\mathcal{L}_k$ balls $\mathcal{B} \in \mathcal{L}_k$ such that $\underline{f}(\mathcal{B}) > U_k$.[8]
    (b) Branching: Choose $\mathcal{B} \in \mathcal{L}_k$ such that $\underline{f}(\mathcal{B}) = L_k$. Split $\mathcal{B}$ into $3^n$ overlapping sub-balls
        $\mathcal{B}_1, \ldots, \mathcal{B}_{3^n}$ according to the splitting rule in Section 2.2 of Fowkes et al. (2013) and
        discard any sub-balls that lie entirely outside of $\mathcal{D}$. Let $\mathcal{R}_k$ denote the set of remaining
        sub-balls and let $\mathcal{L}_{k+1} := (\mathcal{L}_k \setminus \{\mathcal{B}\}) \cup \mathcal{R}_k$.
    (c) Bounding: Partition $\mathcal{R}_k$ into $P$ subsets $\mathcal{R}_k^p$ for $p \in \{1, \ldots, P\}$ and distribute them
        amongst the $P$ worker processors for bounding. Wait until all the bounds $\underline{f}(\mathcal{B}), \overline{f}(\mathcal{B})$
        for $\mathcal{B} \in \mathcal{R}_k$ are received back.
    (d) Set $U_{k+1} := \min_{\mathcal{B} \in \mathcal{L}_{k+1}} \overline{f}(\mathcal{B})$.
```

[8] As correctly pointed out by an anonymous referee (and as used in Algorithm 3.2) one can optionally use $\underline{f}(\mathcal{B}) > U_k - \varepsilon$ as the condition for pruning which may allow the algorithm to discard more redundant balls.

```
    (e) Set $L_{k+1} := \min_{\mathcal{B} \in \mathcal{L}_{k+1}} \underline{f}(\mathcal{B})$.
    (f) Set $k = k + 1$.
2. Send termination signal to worker processors.
3. Return $U_k$ as the estimate of the global minimum of $f(x)$ over $\mathcal{D}$.
```

Worker Processor p

```
1. Repeat the following procedure until termination signal is received:
    (a) Wait for a set of balls $\mathcal{R}_k^p$ from the master processor.
    (b) When the set is received, calculate bounds $\underline{f}(\mathcal{B}), \overline{f}(\mathcal{B})$ for each ball $\mathcal{B} \in \mathcal{R}_k^p$ and send
        the bounds back to the master processor.
```

In step 1b, the algorithm splits each ball $\mathcal{B}$ into 3^n overlapping sub-balls of half-radius $r(\mathcal{B})/2$ centred at the vertices of a hypercubic tessellation of edge length $r(\mathcal{B})/\sqrt{n}$ around the centre of the ball; see Fowkes et al. (2013, Section 2.2) for further details of this splitting rule.

3.2 Task Parallelism: Tree in Parallel

In task parallelism of a branch and bound algorithm, the focus is on exploring the branch and bound tree in parallel. Conceptually, a branch and bound algorithm running on an arbitrary problem can be thought of as generating a tree. The nodes of the tree represent the subregions and the edges denote the regions they are split from. One can then think of having several processor cores generating different sections of the tree starting from different subregions. In the case of traditional branch and bound using boxes this is conceptually straightforward to implement as each subregion forms a distinct partition of the domain and any subregions split from it are also contained within that partition. All that is required is that the processor cores communicate the best upper bound found so far and balance the load, namely make sure the work is evenly distributed amongst the processor cores.

However, we are interested in parallelising oBB which uses overlapping balls rather than rectangular partitions. This makes the parallelisation more difficult since the balls do not form natural partitions. As such several processor cores can end up bounding and splitting the same promising ball, arrived at by repeatedly splitting different initial balls and so doubling of work can occur. Our solution to this problem is to essentially eliminate the doubling entirely through efficient communication. That is, the master processor keeps a list of all the balls created so far and any new balls created by the worker processors are cross-checked against this list to see if they already exist. Of course, such an approach relies heavily on the ability to efficiently communicate centres and radii from the workers to the master. Sending the centre and radius of each ball would be prohibitively expensive, but if we instead send an integer hash (Knuth, 1998, Section 6.4) of each centre and radius this greatly decreases the cost of communication. In fact, every time a worker processor splits a ball it needs to check whether at most 3^n balls of the same radius exist. Thus, we only need to send a hash of one radius and at most 3^n balls, so $3^n + 1$ integers in total. Of course, the hashes are not guaranteed to be unique and there is a chance that the algorithm will occasionally discard a ball that does not already exist. However, such an event is extremely rare, likely to have a very small effect on the resulting minimum and can be easily corrected for by running a local solver at the end of the algorithm. If we combine this approach with task parallelism ideas, we obtain a suitable parallel branch and bound version of oBB with hashing.

An important performance consideration is the order in which the master processor deals with the incoming hashes and we have found two different approaches to be suitable. The first approach is perhaps the most obvious, the master processes the hashes one at a time as they are received and the workers simply wait for confirmation of which balls already exist before bounding them (see One-at-a-time Hashing in Section 3.2.1). While this approach is suitable in situations where the balls are inexpensive to bound relative to the cost of communicating the hashes, it does not perform as well when they are not. This is because the workers tend to spend a significant amount of time waiting for a response from the master. The second approach therefore tries to address these issues by getting the master to process the hashes from all the workers in one go while the workers start bounding the balls in the background (see Synchronous Hashing in Section 3.2.1). This is indeed advantageous if bounding the balls is expensive relative to the communication cost.

Another performance improvement to oBB we implement is the use of a priority queue to store the subproblems. A priority queue is simply an ordered list where each element is ranked according to a specified order. In our case we order the list of balls according to the lower bound $\underline{f}(\mathcal{B})$, with the smallest lower bound included first in the list. This enables us to restructure oBB so that we do not need to find or communicate the smallest lower bound, resulting in a more efficient algorithm in standard numerical form (Crainic et al., 2006).

We also need a strategy to balance the load between processor cores, i.e. the number of balls, or equivalently the number of subproblems, on each processor core. After due consideration and testing of the underlying hardware topology, we implemented a two tier strategy. This is because most modern HPC clusters consists of a large number of nodes (i.e. sets of processors which share the same memory) interconnected by gigabit ethernet or infiniband switches. It therefore makes sense to load balance both within each node where communication via shared memory will be very efficient and across different nodes where communication via gigabit ethernet or infiniband will be relatively slow. We will describe this load balancing strategy in detail in Section 3.2.2. The complete task parallel branch and bound algorithm is given below, with the lower and upper bounds calculated as before in (3.1),(3.2). Details of the hashing and load balancing are presented later in Sections 3.2.1 and 3.2.2.

Algorithm 3.2. Task Parallel Branch and Bound Algorithm
Master Processor

0. *Initialisation*
   - (a) *Set t_{max} to be the maximum runtime of the algorithm.*
   - (b) *Let $\mathcal{B}$ be a ball with centre $x_{\mathcal{B}} \in \mathcal{D}$ of sufficiently large radius to cover $\mathcal{D}$.*
   - (c) *Split $\mathcal{B}$ into 3^n overlapping sub-balls according to the splitting rule in Section 2.2 of Fowkes et al. (2013) and discard any sub-balls that lie entirely outside of $\mathcal{D}$. Partition the remaining sub-balls into P subsets and distribute them amongst the p worker processors as sets $\mathcal{L}^p$ for $p \in \{1,\dots,P\}$.*
   - (d) *Let $\mathcal{R} = \emptyset$ be the initial ordered list of hashes of radii.*
   - (e) *Let $\mathcal{C} = \emptyset$ be the initial ordered list of sets of hashes of centres with the same radius.*
1. *While $\mathcal{L}^p \neq \emptyset\ \forall p$ and the runtime $< t_{max}$, repeat the following procedure:*
   - (a) *Asynchronously receive U_p and the size $|\mathcal{L}^p|$ of the set $\mathcal{L}^p$ from all $p \in \{1,\dots,P\}$ worker processors.*
   - (b) *Asynchronously send $U := \min_{p\in\{1,\dots,P\}} U_p$ to all P worker processors.*
   - (c) *Hashing: Process lists of hashes received from worker processors, updating $\mathcal{R}$, the list*

of radius hashes,[9] *and $\mathcal{C}$, the list of ball-centre hashes, and inform the workers of any duplicate entries (see Section 3.2.1).*

   (d) *Perform load balancing across nodes (see Section 3.2.2).*

   (e) *Perform load balancing within nodes (see Section 3.2.2).*

2. *Send termination signal to worker processors.*
3. *Return U as the estimate of the global minimum of $f(x)$ over $\mathcal{D}$.*

Worker Processor p

1. *Initialisation*

   (a) *Receive workload $\mathcal{L}^p$ from master processor.*

   (b) *Calculate bounds $\underline{f}(\mathcal{B})$ and $\overline{f}(\mathcal{B})$ as defined in (3.1) and (3.2), respectively, for each ball $\mathcal{B} \in \mathcal{L}^p$ and convert $\mathcal{L}^p$ into a priority queue w.r.t. $\underline{f}(\mathcal{B})$.*

   (c) *Set $U_p := \min_{\mathcal{B}\in\mathcal{L}^p} \overline{f}(\mathcal{B})$.*

   (d) *Asynchronously send U_p and $|\mathcal{L}^p|$ to master processor.*

   (e) *Asynchronously receive U from master processor.*

2. *Repeat the following procedure until termination signal is received:*

   (a) *Pruning: Remove from the priority queue $\mathcal{L}^p$ balls $\mathcal{B}$ such that $\underline{f}(\mathcal{B}) > U - \varepsilon$.*

   (b) *Branching: Let $\mathcal{B}$ be the first element in the priority queue $\mathcal{L}^p$.*[10] *Split $\mathcal{B}$ into 3^n overlapping sub-balls $\mathcal{B}_1, \ldots, \mathcal{B}_{3^n}$ according to the splitting rule in Section 2.2 of Fowkes et al. (2013) and discard any sub-balls that lie entirely outside of $\mathcal{D}$. Let $\mathcal{R}$ denote the list of remaining sub-balls.*

   (c) *Hashing: Generate an integer hash for each ball in $\mathcal{R}$ and an integer hash for the radius. Send the integer hashes to master processor to see if any of the balls already exist. (Synchronised hashing only: Start bounding $f(x)$ for each ball in $\mathcal{R}$ until the master processor sends the results of the check back). Receive an ordered integer list from the master processor that contains either 1 or 0 depending on whether each ball exists and update $\mathcal{R}$ accordingly.*

   (d) *Bounding: Calculate bounds $\underline{f}(\mathcal{B}), \overline{f}(\mathcal{B})$ according to (3.1),(3.2), for each ball $\mathcal{B} \in \mathcal{R}$ if not already bounded.*

   (e) *Remove the split ball $\mathcal{B}$ from the priority queue $\mathcal{L}^p$ and add the list of remaining sub-balls $\mathcal{R}$ to $\mathcal{L}^p$.*

   (f) *Set $U_p := \min_{\mathcal{B}\in\mathcal{L}^p} \overline{f}(\mathcal{B})$.*

   (g) *Load Balancing: Asynchronously send the requested number of subproblems from the current workload to the required processor(s) as instructed by the master processor and update $\mathcal{L}^p$ accordingly. If more subproblems are requested than in the current workload, send as many as possible. Send confirmation to the master processor once the send has completed.*

   (h) *Load Balancing: Asynchronously receive subproblems from other processors and update $\mathcal{L}^p$ accordingly.*

   (i) *Asynchronously send U_p and $|\mathcal{L}^p|$ to master processor.*

   (j) *Asynchronously receive U from master processor.*

[9] Note that $\mathcal{R}$ is always a finite set. As the radius is halved each time a ball is split, there can only be a finite number of radii before numerical underflow occurs.

[10] Note that since $\mathcal{L}^p$ is a priority queue w.r.t. $\underline{f}(\mathcal{B})$, $\mathcal{B}$ has the smallest lower bound $\underline{f}(\mathcal{B})$ of all balls in $\mathcal{L}^p$.

There are a number of strategies we have tried in an attempt to further improve Algorithm 3.2 and we will briefly mention them here. One may be tempted to think that breaking down the communication of the hashes into smaller messages would be more efficient, as this would allow the algorithm to solve some of the subproblems during sending/receiving. However, sending several small messages can cause the communication latency to dominate the overall cost of communication. Our tests have indicated that it is indeed more efficient to send all the hashes in one large message as this minimises latency and increases the effective communication bandwidth. Another aspect we have looked at is whether the cost of solving each subproblem has a significant effect on the speedup observed. To test this we artificially extended the cost of solving each subproblem by one second on the random polynomials and tested Algorithm 3.2 on this modified test set. However, we observed little difference in the speedup when compared to Algorithm 3.2 on the original test set.

3.2.1 Hashing

In this section we will describe the hashing process used in Algorithm 3.2 in more detail, primarily the role of the master processor. In order to be able to process the hashes efficiently, the master processor keeps a list $\mathcal{R}$ containing hashes of the radius and a corresponding list $\mathcal{C}$ of sets of hashes of centres of balls with that radius (see step 0d and step 0e for the master processor in Algorithm 3.2). For example if $\mathcal{R} = \{\#r_1, \#r_2\}$ and $\mathcal{C} = \{\{\#x_{\mathcal{B}_1}, \#x_{\mathcal{B}_2}\}, \{\#x_{\mathcal{B}_3}\}\}$ then balls $\mathcal{B}_1, \mathcal{B}_2$ have radius r_1 and $\mathcal{B}_3$ has radius r_3. Every time a ball is split into sub-balls by a worker processor in the algorithm, each sub-ball has exactly the same radius and so the worker only has to send one radius hash along with the hashes of the centres. When the master processor receives the radius hash and corresponding centre hashes, it can quickly determine the radius of the split balls since hashes of radii are stored in a separate list $\mathcal{R}$ which can be efficiently searched.

As for calculating the hashes themselves, hashing the radius is straightforward, we simply multiply by a suitably large number (e.g. we use 10^8) and convert to a 32-bit integer. For hashing the centre of each ball, we use a variant of the hash function from Section 4.1 of Teschner, Heidelberger, Mueller, Pomeranets and Gross (2003). That is, for $x \in \mathbb{R}^n$, a collection of large primes $p_1, \ldots, p_n$ and a resolution r, the hash is

$$\#x = \left\lfloor \frac{x_1}{r} \right\rfloor p_1 \veebar \left\lfloor \frac{x_2}{r} \right\rfloor p_2 \veebar \cdots \veebar \left\lfloor \frac{x_n}{r} \right\rfloor p_n$$

where $\veebar$ denotes a *bitwise xor* (i.e. an *exclusive or* on the binary digits). The hash is then converted to a 32-bit integer which ensures that the communication is as efficient as possible. In our implementation we use a collection of arbitrarily chosen 8-digit primes along with a resolution r of 10^{-5}, so any $\|x\|_\infty < 10^{-5}$ hashes to zero.

As we have mentioned previously, there are two possible ways for the master to process the hashes: one-at-a-time or synchronised (namely, all together). We describe how in Algorithm 3.2, step 1c the master processor handles either of the two approaches below.

One-at-a-time Hashing

Master Processor: *(Step 1c of Algorithm 3.2)*

1. *If a worker processor p wants to check if a set of balls of the same radius already exists, receive a list containing an integer hash $\#c_i$ of the centre of each ball $\mathcal{B}_i$ and an integer hash $\#r$ of the radius.*
2. *Check if $\#r$ is in $\mathcal{R}$. If it is not, append $\#r$ to $\mathcal{R}$, append the set of $\#c_i$'s to $\mathcal{C}$ since they cannot already be present in $\mathcal{C}$. If $\#r$ is in $\mathcal{R}$, check if any of the $\#c_i$'s are already present in the corresponding set in $\mathcal{C}$. Add any $\#c_i$'s that are not present to the corresponding set in $\mathcal{C}$.*

3. *Set $\mathcal{E}$ to be an ordered list that contains either* 1 *or* 0 *for each i depending on whether $\#c_i$ is present in the corresponding set in $\mathcal{C}$ or not and send $\mathcal{E}$ to worker processor p.*

Synchronised Hashing

Master Processor: *(Step 1c of Algorithm 3.2)*

1. *Receive from all worker processors p, a list containing integer hashes of the radius $\#r^p$ and centres $\{\#c_i^p\}$ of each ball $\mathcal{B}_i^p$ on processor p wanting to be checked.*
2. *For each p, check if $\#r^p$ is in $\mathcal{R}$. If it is not, append $\#r^p$ to $\mathcal{R}$, append the set of $\#c_i^p$'s to $\mathcal{C}$ since they cannot already be present in $\mathcal{C}$. If $\#r^p$ is in $\mathcal{R}$, check if any of the $\#c_i^p$'s are already present in the corresponding set in $\mathcal{C}$. Add any $\#c_i^p$'s that are not present to the corresponding set in $\mathcal{C}$.*
3. *For each p, set $\mathcal{E}_p$ to be an ordered list that contains either* 1 *or* 0 *for each i depending on whether $\#c_i^p$ is present in the corresponding set in $\mathcal{C}$ or not and send $\mathcal{E}_p$ to worker processor p.*

3.2.2 Load Balancing

As we have discussed previously, our load balancing scheme in Algorithm 3.2 takes into account the underlying topology of modern HPC clusters by alternately balancing across and within the underlying physical nodes of the cluster. In this section we describe the two load balancing approaches, starting with load balancing within nodes as it forms the basis of our strategy for load balancing across nodes. We will use $\mathcal{N}$ throughout to denote a node.

Load balancing across processors within a node

For each node, we balance the load across processors within the node as follows: At each load balancing step the master processor takes a snapshot of the load on the node and works out how many subproblems each processor within that node should have in order to be balanced. It then assigns the shortfall from the processor with the largest load to the one with the smallest, updates the snapshot and repeats until all processors in the node have a load that does not differ by more than 10%. That is, for all processors $p_1, p_2 \in \mathcal{N}$

$$\frac{|S^{p_1} - S^{p_2}|}{\max\{\min\{S^{p_1}, S^{p_2}\}, 1\}} > 0.1 \tag{3.3}$$

where S^p denotes the load (i.e. the number of subproblems) on processor p. The scheme for the master processor in Algorithm 3.2, step 1e is given in more detail below.

Master Processor: *(Step 1e of Algorithm 3.2) For each node $\mathcal{N}$, repeat the following procedure: Let S^p be a snapshot of the load $|\mathcal{L}^p|$ on each worker processor p in $\mathcal{N}$, i.e. a local copy of the load that we will work with. Let $I = \left\lfloor \sum_{p\in\mathcal{N}} S^p / |\mathcal{N}| \right\rfloor$ be the ideal load on each processor in $\mathcal{N}$. We would like all processor loads to be as close as possible to the ideal load I. Set $k = 0$. While* (3.3) *holds and $k < |\mathcal{N}|$, repeat the following procedure:*

1. *Let $S^{p_{\min}}$ and $S^{p_{\max}}$ be the smallest and largest loads in the node $\mathcal{N}$ on processors $p_{\min}$ and $p_{\max}$ respectively.*
2. *Calculate $I - S^{p_{\min}}$ as the load we need to add to $S^{p_{\min}}$ so that it has ideal load.*

3. *If any previous send has reached its destination, instruct processor $p_{\max}$ to asynchronously send $I - S^{p_{\min}}$ subproblems to processor $p_{\min}$.*
4. *Update snapshots: subtract $I - S^{p_{\min}}$ from $S^{p_{\max}}$ and add $I - S^{p_{\min}}$ to $S^{p_{\min}}$ so that the previously smallest load increases to I and the previously largest load decreases to $S^{p_{\max}} + S^{p_{\min}} - I > S^{p_{\min}}$ (unless $S^{p_{\max}} = I$ in which case it is already balanced).*
5. *Set $k = k + 1$.*

Load balancing across nodes

We apply a similar scheme for load balancing across nodes as follows: At each load balancing step the master processor takes a snapshot of the overall load on each node and works out how many subproblems each node should have in order to be balanced. It then assigns a fraction of the shortfall from the processor with the largest load and distributes it as evenly as possible to all processors on the node with the smallest load. The snapshot of the total load on each node is then updated and the process repeats until all the nodes have a load that does not differ by more than 10%. That is, for all nodes $j_1, j_2 = 1, \dots, N$

$$\frac{|T^{j_1} - T^{j_2}|}{\max\{\min\{T^{j_1}, T^{j_2}\}, 1\}} > 0.1 \tag{3.4}$$

where T^j denotes the total load on node $\mathcal{N}_j$ for $j = 1, \dots, N$. Of course by the time the subproblems are actually transmitted the nodes are unbalanced again and the whole process starts over at the next load balancing step. The scheme for the master processor in Algorithm 3.2, step 1d is given in more detail below.

Master Processor: *(Step 1d of Algorithm 3.2) Let S^p be a snapshot of the load $|\mathcal{L}^p|$ on each worker processor $p = 1, \dots, P$, i.e. a local copy of the load that we will work with. Let $T^j = \sum_{p \in \mathcal{N}_j} S^p$ denote the total load on each node $\mathcal{N}_j$ for $j = 1, \dots, N$. Set $k = 0$. While* (3.4) *holds and $k < P$, repeat the following procedure:*

1. *Let $I = \left\lfloor \sum_{j=1}^{N} T^j / N \right\rfloor$ be the ideal node load. We would like all node loads to be as close as possible to the ideal node load I.*
2. *Let $T^{j_{\min}}$ and $T^{j_{\max}}$ be the smallest and largest node loads, present on the nodes $\mathcal{N}_{j_{\min}}$ and $\mathcal{N}_{j_{\max}}$ respectively. Calculate $I - T^{j_{\min}}$ as the node load we need to add to node $\mathcal{N}_{j_{\min}}$ so that it has ideal node load.*
3. *Let $S^{p_{\min}}$ and $S^{p_{\max}}$ be the smallest and largest processor loads on nodes $\mathcal{N}_{j_{\min}}$ and $\mathcal{N}_{j_{\max}}$ respectively. Ideally, we would like to take $I - T^{j_{\min}}$ subproblems from processor $p_{\max}$ and distribute them evenly across all processors in node $\mathcal{N}_{j_{\min}}$. However, this may deplete processor $p_{\max}$ so we lower the amount we take by $S^{p_{\min}}$ and do not take more than $[S^{p_{\max}}/3]$, where $[\cdot]$ denotes rounding to the nearest integer. This gives the actual amount A to take from $p_{\max}$ as*

$$A = \begin{cases} \max\{I - T^{j_{\min}} - S^{p_{\min}}, 0\} & \text{if less than } [S^{p_{\max}}/3], \\ [S^{p_{\max}}/3] & \text{otherwise.} \end{cases}$$

4. *If $A > 0$ and any previous send has reached its destination, instruct processor $p_{\max}$ to asynchronously send $[A/|\mathcal{N}_{j_{\min}}|]$ subproblems to each of the processors on node $\mathcal{N}_{j_{\min}}$.*
5. *Update snapshots: add $[A/|\mathcal{N}_{j_{\min}}|]$ to S^p for all processors p in node $\mathcal{N}_{j_{\min}}$ and subtract A from $S^{p_{\max}}$ so that the previously smallest node load increases by A and the previously largest node load decreases by A.*

6. *Recompute the total node load* $T^j = \sum_{p \in \mathcal{N}_j} S^p$ *on each node* $\mathcal{N}_j$ *for* $j = 1, \ldots, N$.
7. *Set* $k = k + 1$.

In order to get an idea of how the load on a processor core varies throughout the computation, we have included in Appendix B diagrams of the load on an arbitrarily chosen processor for RBF approximations to *biggsc4, ex2_1_4* and *brownden* with average, poor and excellent speedup, respectively. Each figure depicts the load against time on processor core no. 12 for the task parallel algorithm, Algorithm 3.2, running on $24, 36, 48$ and 60 processor cores. One can see from the figures that while the load is somewhat erratic at times, in general the problems with better speedup (*biggsc4* and *brownden*) have a load that is better balanced. It should be noted that good load balancing is very difficult to obtain for task parallel branch and bound algorithms because entire sub-trees within the branch and bound tree regularly disappear as they become fathomed, i.e. can no longer contain the global minimum as they have a lower bound greater than the smallest upper bound. This requires large transfers of data to smooth out the sudden load imbalance, which the communication hardware struggles to cope with.

3.3 Numerical Results

We will now test the parallel performance of our data parallel and task parallel algorithms, namely Algorithm 3.1 and Algorithm 3.2, respectively, on the random polynomials and RBFs from Section 2.3 along with RBF approximations to a selection of problems from the COCONUT benchmark (Shcherbina et al., 2003). We will also run the serial code, namely Algorithm 3.1 on one processor core, so that we can compare the parallel performance against the serial. In order to do this we will calculate the speedup S_P of the parallel algorithm on P processor cores over the serial defined as

$$S_P = \frac{T_1}{T_P} \tag{3.5}$$

where T_1 is the runtime of the serial algorithm and T_P the runtime of the parallel algorithm on P processors.[11] The hardware used is part of the ECDF Eddie cluster where each node consists of two six-core 2.4GHz Intel Xeon E5645 processors with 2GB of RAM per core and the nodes communicate via Gigabit Ethernet.

3.3.1 Random Polynomials and RBFs

Let us begin by looking at the parallel performance of our data and task parallel algorithms on the random polynomials and RBFs from Section 2.3. To this end, we will run a parallel Python-based MPI implementation of both Algorithm 3.1 and Algorithm 3.2 with one-at-a-time hashing to an absolute tolerance of 10^{-6}, i.e. we set $\varepsilon = 10^{-6}$ in step 1 of Algorithm 3.1 and in step 2a of Algorithm 3.2. As the first two series of problems described in Section 2.3 are very fast to solve in serial, there is little to be gained from running them in parallel and we will therefore focus on series three only. We will test the same ten realisations of series three from Section 2.3 in both serial and parallel for 12, 24, 36, 48 and 60 processor cores. Based upon the results of Section 2.3 we will use cubic underestimators (2.12) with the second order tensor Gershgorin estimation approach given in (2.15) for the random polynomials and the second order tensor norm approach given in (2.16) for the random RBFs.

[11]Note that another approach to test the efficiency of parallelism is to calculate the redundancy defined as $(F_P - F_1)/F_P$ when $F_P > F_1$ and 0 otherwise, where F_1 is the number of function evaluations of the serial algorithm and F_P the number of function evaluations of the parallel algorithm on P processors (see p. 324 of Strongin and Sergeyev, 2000).

Let us first consider the parallel performance of our data parallel algorithm, Algorithm 3.1. In the left-hand side of Figure 3.1 we can see the average speedup with confidence intervals for

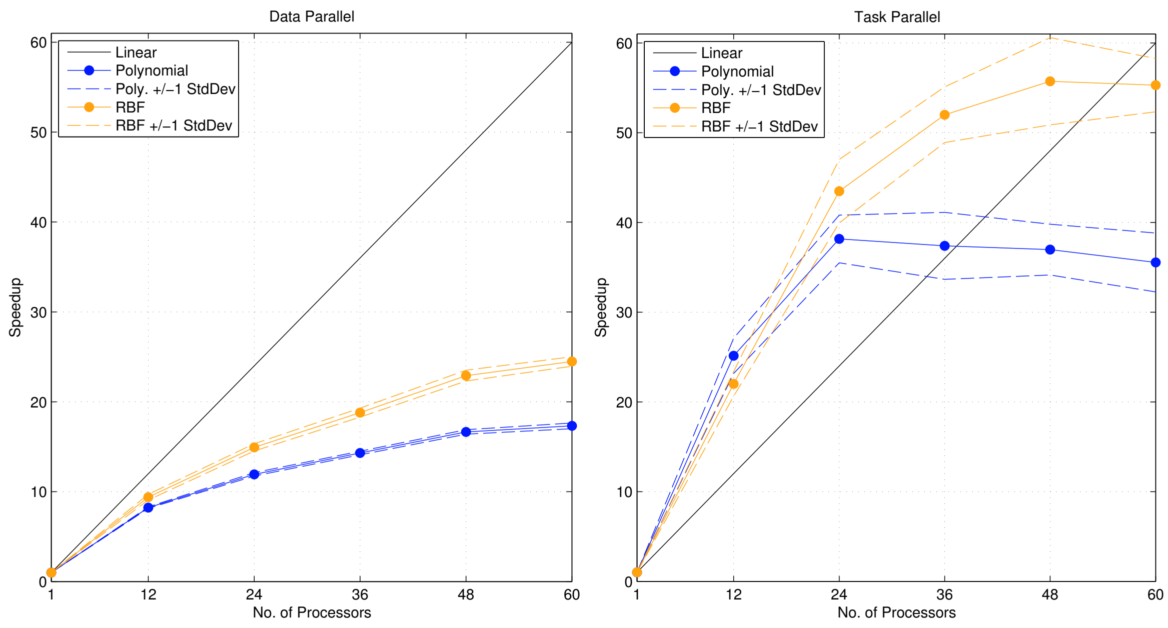

Figure 3.1: *Average speedup* (3.5) *with confidence intervals for Algorithm 3.1 (Data Parallel, left) and Algorithm 3.2 (Task Parallel, right) over random polynomial and RBF series 3.*

Algorithm 3.1 over random polynomial and RBF series three as the number of processor cores is increased. We can see that we get rather poor sublinear speedup for both with a maximum of 17 times average speedup for the random polynomials and 24 times average speedup for the random RBFs on 60 processor cores. The reason for the better performance of the random polynomials is that calculating the elementwise lower and upper bounds is more demanding as it uses a more sophisticated interval arithmetic approach (see Section 6.2 of Fowkes et al., 2013, for details) and therefore the worker processors are better utilised.

Let us now look at the performance of our task parallel algorithm, Algorithm 3.2 with one-at-a-time hashing. We have found that one-at-a-time hashing significantly outperforms synchronised hashing for both random polynomials and RBFs because the subproblems are inexpensive to solve relative to the cost of communicating the hashes. We can immediately see from the right-hand side of Figure 3.1 that our task parallel algorithm performs significantly better than the data parallel algorithm. Random RBFs exhibit the best performance with superlinear speedup until 48 processor cores with a maximum of 55 times average speedup before levelling off. Random polynomials do not perform as well with superlinear speedup until 24 processor cores with a maximum of 36 times average speedup before dipping slightly, nonetheless the performance is still much better than that achieved with the data parallel algorithm. The poorer performance of the random polynomial problems is due to the fact that they are very quick to solve, taking only around four hundred seconds on 12 processor cores (see Table A.2), which coupled with the fact that the subproblems themselves are fast to solve, leaves little scope for improvement by adding additional processor cores.

3.3.2 COCONUT Benchmark

For a more thorough numerical evaluation we will test the parallel performance of our data parallel and task parallel algorithms on radial basis function approximations to a selection of 31 problems[12] from the COCONUT benchmark whose dimension varies from 4 to 6 (see Shcherbina et al., 2003, for details of the benchmark). Table A.3 in Appendix A gives a brief overview of the test functions we have approximated. We chose to approximate the COCONUT problems using RBFs as they enable us to calculate the tensor bounds (2.14) used in the estimation approach for the lower bound (3.1) efficiently using a more accurate interval arithmetic type approach (see Section 6.2 of Fowkes et al., 2013). As before, we will use cubic underestimators (2.12) with the second order tensor norm approach given in (2.16).

The RBF approximations are fitted to a maximin Latin Hypercube sample of $10n$ scattered sample points in $\mathbb{R}^n$ and use the cubic spline objective function we have previously used for the random RBFs in (2.19) with a weighted norm (see Chapters 3 and 4 of Fowkes, 2012, for details). Once again, we will run a parallel Python-based MPI implementation of both Algorithm 3.1 and Algorithm 3.2 but this time with synchronous hashing. This is because we have found that synchronous hashing leads to significantly better performance for our approximation to the COCONUT benchmark since in general the subproblems are expensive to bound relative to the cost of communicating the hashes. So that we can test both easier and harder problems we will run each problem to the absolute tolerance it achieved in 12 hours on the serial code. We will test all 31 problems in both serial and parallel for 12, 24, 36, 48 and 60 processor cores. We will use the tensor norm approach given Theorem 2.6 (2.16) as we have shown in Section 2.3 that it performs better for RBF approximations.

As before, let us start by looking at the performance of our data parallel algorithm, Algorithm 3.1, on our approximation to the COCONUT benchmark. In the left-hand side of

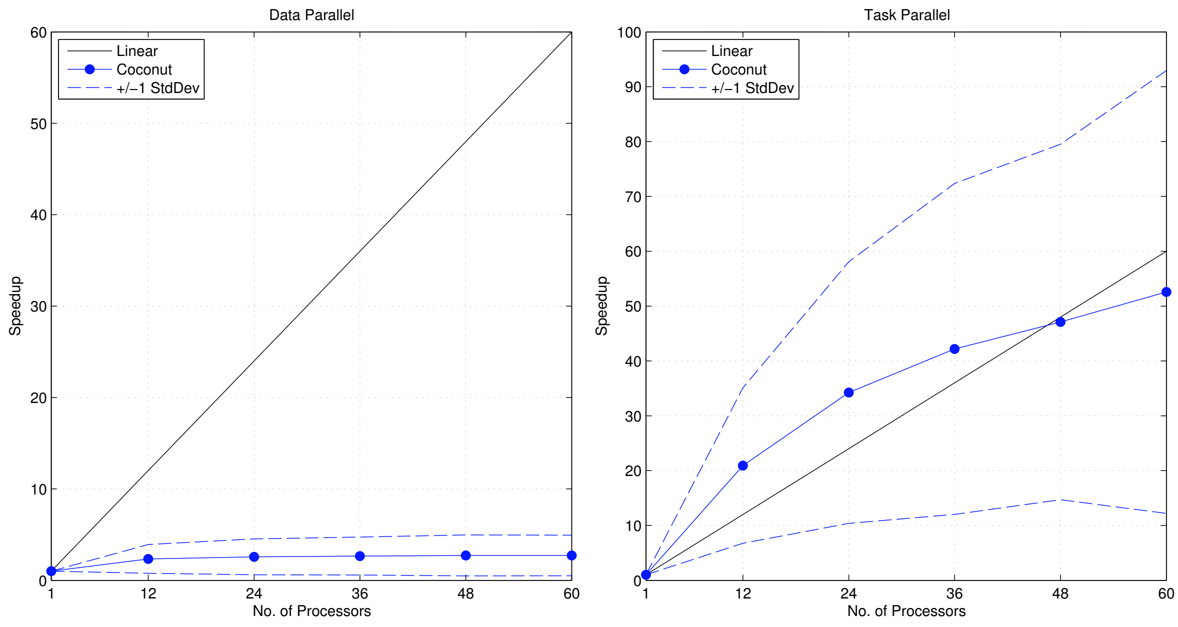

Figure 3.2: *Average speedup* (3.5) *with confidence intervals for Algorithm 3.1 (Data Parallel, left) and Algorithm 3.2 (Task Parallel, right) over RBF approximations to selected functions from the COCONUT test set.*

[12]Note that the majority of problems in the COCONUT benchmark have nonlinear constraints that our algorithms cannot handle at present. This rather limited the number of problems we could actually test.

Figure 3.2 we can see that the performance is in fact very poor with an average speedup of around three times all the way through from 12 to 60 processors. This is very disappointing but not unexpected as bounding subproblems is not where the majority of the work in the algorithm takes place but it is in exploring the branch and bound tree.

Looking at the performance of our task parallel algorithm, Algorithm 3.2 with synchronous hashing in the right-hand side of Figure 3.2, we can see significantly better speedup. In fact, we are able to achieve superlinear speedup on average up to 36 processor cores, past which the speedup continues to increase, albeit remaining slightly sublinear, up to a maximum of 52 times average speedup.

In conclusion, we can clearly see from the numerical results that a task parallel approach leads to a very efficient parallel algorithm on average which exhibits good speedup. The data parallel algorithm on the other hand performs rather poorly, especially on our approximation to the COCONUT test set. This is due to the fact that the subproblem bounding which is parallelised in the data parallel algorithm does not account for majority of the computational work in exploring the branch and bound tree. Overall, we have shown that it is indeed possible to devise an efficient parallel overlapping branch and bound algorithm albeit after overcoming some underlying difficulties.

4 Conclusions

We have presented branching and bounding improvements for global optimization algorithms with Lipschitz continuity properties and implemented our findings by improving a recent serial branch and bound algorithm presented in Fowkes et al. (2013). We have shown that it is possible to significantly improve upon the bounding strategies used in Lipschitz based global optimization algorithms by drawing upon a variety of both existing and novel bounds. Our numerical results indicate that no single bound is optimal across all types of objective function, although our novel second order bounds exhibit the best performance in general. As these bounds are inexpensive to calculate for small to medium-scale problems compared with the cost of the rest of the algorithm it is feasible to implement all of them in a branch and bound algorithm and adaptively choose the best bound at runtime. Clearly, the latter would be the best way to maximise the efficiency of Lipschitz based branch and bound algorithms.

Our second avenue of investigation considered improving the branching used in a Lipschitz based global optimization algorithm through the use of parallelism. We investigated two standard parallel programming paradigms, namely data parallelism and task parallelism. We found that our data parallel approach which focused on parallelising the bounding operations within the algorithm performed poorly. However, our task parallel approach which focused on parallelising the branch and bound tree itself and proved to be a real challenge to realise, proved very successful once implemented and exhibited excellent average speedup on a large number of varied test problems. Our use of hashing within the task parallel algorithm was essential to obtaining good performance and we identified two main strategies for processing the hashes, namely one-at-a-time and synchronous hashing. Once again, adaptively choosing between these two strategies in the algorithm would maximise its efficiency over a large variety of problems.

A critical challenge remains: scaling up the problem dimension so that we can solve problems of greater practical interest. Scaling up the code to larger parallel machines seems like an immediate solution but may not yield results as good as one might expect due to the increased communication overhead this would bring. In the context of our approach, a better remedy perhaps lies in finding more efficient coverings that would still allow us to use non-convex bounding procedures within the algorithm.

Acknowledgements

The work of the first and second authors was supported by EPSRC grants EP/I028854/1 and NAIS EP/G036136/1 and the work of the third author by EP/I013067/1. We are also grateful to NAIS for funding computing time.

Bibliography

Alba, E., Almeida, F., Blesa, M., Cotta, C., Díaz, M., Dorta, I., Gabarró, J., León, C., Luque, G., Petit, J., Rodríguez, C., Rojas, A. and Xhafa, F. (2006) 'Efficient Parallel LAN/WAN Algorithms for Optimization. The MALLBA Project'. *Parallel Computing*, vol. 32, no. 5-6, pp. 415 – 440. http://dx.doi.org/10.1016/j.parco.2006.06.007.

Ananth, G. Y., Kumar, V. and Pardalos, P. M. (1993) 'Parallel Processing of Discrete Optimization Problems'. In Kent, A. and Williams, J. (eds.) *Encyclopedia of Microcomputers*, vol. 13. Dekker. ISBN 978-0-8247-2711-6, pp. 129–147. http://books.google.co.uk/books?id=Rx3hqGdXcooC.

Baritompa, W. and Cutler, A. (1994) 'Accelerations for Global Optimization Covering Methods using Second Derivatives'. *Journal of Global Optimization*, vol. 4, no. 3, pp. 329–341. http://dx.doi.org/10.1007/BF01098365.

Breiman, L. and Cutler, A. (1993) 'A Deterministic Algorithm for Global Optimization'. *Mathematical Programming*, vol. 58, no. 1-3, pp. 179–199. http://dx.doi.org/10.1007/BF01581266.

Casado, L. G., Martìnez, J. A., Garcìa, I. and Hendrix, E. M. T. (2008) 'Branch-and-Bound Interval Global Optimization on Shared Memory Multiprocessors'. *Optimization Methods & Software*, vol. 23, no. 5, pp. 689–701. http://dx.doi.org/10.1080/10556780802086300.

Conn, A. R., Gould, N. I. M. and Toint, P. L. (2000) *Trust Region Methods*. MPS-SIAM Series on Optimization. SIAM. ISBN 978-0-89871-460-9. http://books.google.co.uk/books?id=5kNC4fqssYQC.

Crainic, T. G., Le Cun, B. and Roucairol, C. (2006) 'Parallel Branch-and-Bound Algorithms'. In Talbi, E. (ed.) *Parallel Combinatorial Optimization*, Wiley Series on Parallel And Distributed Computing. Wiley. ISBN 978-0-471-72101-7, pp. 1–28. http://books.google.co.uk/books?id=rYtuk_sm23UC.

Dolan, E. D. and Moré, J. J. (2002) 'Benchmarking Optimization Software with Performance Profiles'. *Mathematical Programming*, vol. 91, no. 2, pp. 201–213. http://dx.doi.org/10.1007/s101070100263.

Evtushenko, Y. G. (1971) 'Numerical Methods for Finding Global Extrema (Case of a Non-Uniform Mesh)'. *USSR Computational Mathematics and Mathematical Physics*, vol. 11, no. 6, pp. 38–54.

Evtushenko, Y. G. and Posypkin, M. A. (2011) 'An Application of the Nonuniform Covering Method to Global Optimization of Mixed Integer Nonlinear Problems'. *Computational Mathematics and Mathematical Physics*, vol. 51, no. 8, pp. 1286–1298. http://dx.doi.org/10.1134/S0965542511080082.

Evtushenko, Y. G. and Posypkin, M. A. (2013) 'A Deterministic Approach to Global Box-Constrained Optimization'. *Optimization Letters*, vol. 7, no. 4, pp. 819–829. http://dx.doi.org/10.1007/s11590-012-0452-1.

Evtushenko, Y. G., Posypkin, M. A. and Sigal, I. (2009) 'A Framework for Parallel Large-Scale Global Optimization'. *Computer Science - Research and Development*, vol. 23, no. 3-4, pp. 211–215. http://dx.doi.org/10.1007/s00450-009-0083-7.

Floudas, C. (1999) *Deterministic Global Optimization: Theory, Methods and Applications*. Nonconvex Optimization and Its Applications. Springer. ISBN 978-0-7923-6014-8. http://books.google.co.uk/books?id=qZSpq27TsOcC.

Fowkes, J. M. (2012) *Bayesian Numerical Analysis: Global Optimization and Other Applications*. Ph.D. thesis, Mathematical Institute, University of Oxford. http://ora.ox.ac.uk/objects/uuid:ab268fe7-f757-459e-b1fe-a4a9083c1cba.

Fowkes, J. M., Gould, N. I. M. and Farmer, C. L. (2013) 'A Branch and Bound Algorithm for the Global Optimization of Hessian Lipschitz Continuous Functions'. *Journal of Global Optimization*, vol. 56, no. 4, pp. 1791–1815. http://dx.doi.org/10.1007/s10898-012-9937-9.

Gaviano, M. and Lera, D. (2008) 'A Global Minimization Algorithm for Lipschitz Functions'. *Optimization Letters*, vol. 2, no. 1, pp. 1–13. http://dx.doi.org/10.1007/s11590-006-0036-z.

Gendron, B. and Crainic, T. G. (1994) 'Parallel Branch-and-Bound Algorithms: Survey and Synthesis'. *Operations Research*, vol. 42, no. 6, pp. 1042–1066. http://dx.doi.org/10.1287/opre.42.6.1042.

Grishagin, V. A. (1978) 'Operating Characteristics of Some Global Search Algorithms'. *Problems of Stochastic Search*, vol. 7, pp. 198–206 (In Russian).

Horst, R. and Tuy, H. (1996) *Global Optimization: Deterministic Approaches*. Springer. ISBN 978-3-540-61038-0. http://books.google.co.uk/books?id=usFjGFvuBDEC.

Knuth, D. (1998) *The Art of Computer Programming: Sorting and Searching*, *The Art of Computer Programming*, vol. 3. Addison-Wesley. ISBN 978-0-201-89685-5. http://books.google.co.uk/books?id=ePzuAAAAMAAJ.

Kolda, T. and Mayo, J. (2011) 'Shifted Power Method for Computing Tensor Eigenpairs'. *SIAM Journal on Matrix Analysis and Applications*, vol. 32, no. 4, pp. 1095–1124. http://dx.doi.org/10.1137/100801482.

Kreinovich, V. and Kearfott, R. (2005) 'Beyond Convex? Global Optimization is Feasible Only for Convex Objective Functions: A Theorem'. *Journal of Global Optimization*, vol. 33, no. 4, pp. 617–624. http://dx.doi.org/10.1007/s10898-004-2120-1.

Kvasov, D. E. and Sergeyev, Y. D. (2009) 'A Univariate Global Search Working with a Set of Lipschitz Constants for the First Derivative'. *Optimization Letters*, vol. 3, no. 2, pp. 303–318. http://dx.doi.org/10.1007/s11590-008-0110-9.

Kvasov, D. E. and Sergeyev, Y. D. (2012a) 'Lipschitz Gradients for Global Optimization in a One-Point-Based Partitioning Scheme'. *Journal of Computational and Applied Mathematics*, vol. 236, no. 16, pp. 4042 – 4054. http://dx.doi.org/10.1016/j.cam.2012.02.020.

Kvasov, D. E. and Sergeyev, Y. D. (2012b) 'Univariate Geometric Lipschitz Global Optimization Algorithms'. *Numerical Algebra, Control and Optimization*, vol. 2, no. 1, pp. 69–90. http://dx.doi.org/10.3934/naco.2012.2.69.

Lera, D. and Sergeyev, Y. D. (2013) 'Acceleration of Univariate Global Optimization Algorithms Working with Lipschitz Functions and Lipschitz First Derivatives'. *SIAM Journal on Optimization*, vol. 23, no. 1, pp. 508–529. http://dx.doi.org/10.1137/110859129.

Lim, L.-H. (2005) 'Singular Values and Eigenvalues of Tensors: A Variational Approach'. In *Computational Advances in Multi-Sensor Adaptive Processing, 2005 1st IEEE International Workshop on*. pp. 129 –132. http://dx.doi.org/10.1109/CAMAP.2005.1574201.

Neumaier, A. (2004) 'Complete Search in Continuous Global Optimization and Constraint Satisfaction'. *Acta Numerica*, vol. 13, pp. 271–369. http://dx.doi.org/10.1017/S0962492904000194.

Pardalos, P. M., Horst, R. and Thoai, N. V. (1995) *Introduction To Global Optimization*, *Nonconvex Optimization and its Applications*, vol. 3. Springer. ISBN 978-0-7923-3556-6. http://www.springer.com/mathematics/book/978-0-7923-3556-6.

Paulavičius, R., Žilinskas, J. and Grothey, A. (2011) 'Parallel Branch and Bound for Global Optimization with Combination of Lipschitz Bounds'. *Optimization Methods and Software*, vol. 26, no. 3, pp. 487–498. http://dx.doi.org/10.1080/10556788.2010.551537.

Pintér, J. D. (1996) *Global Optimization in Action*, *Nonconvex Optimization and its Applications*, vol. 6. Springer. ISBN 978-0-7923-3757-7. http://www.springer.com/mathematics/book/978-0-7923-3757-7.

Piyavskii, S. A. (1972) 'An Algorithm for Finding the Absolute Extremum of a Function'. *USSR Computational Mathematics and Mathematical Physics*, vol. 12, no. 4, pp. 57 – 67. http://dx.doi.org/10.1016/0041-5553(72)90115-2.

Qi, L. (2005) 'Eigenvalues of a Real Supersymmetric Tensor'. *Journal of Symbolic Computation*, vol. 40, no. 6, pp. 1302 – 1324. http://dx.doi.org/10.1016/j.jsc.2005.05.007. http://www.sciencedirect.com/science/article/pii/S0747717105000817.

Sergeyev, Y. D. (1998) 'Global One-Dimensional Optimization using Smooth Auxiliary Functions'. *Mathematical Programming*, vol. 81, no. 1, pp. 127–146. http://dx.doi.org/10.1007/BF01584848.

Sergeyev, Y. D., Strongin, R. G. and Lera, D. (2013) *Introduction to Global Optimization Exploiting Space-Filling Curves*. SpringerBriefs in Optimization. Springer. ISBN 978-1-4614-8041-9. http://books.google.co.uk/books?id=IqyYnAEACAAJ.

Shcherbina, O., Neumaier, A., Sam-Haroud, D., Vu, X.-H. and Nguyen, T.-V. (2003) 'Benchmarking Global Optimization and Constraint Satisfaction Codes'. In Bliek, C., Jermann, C. and Neumaier, A. (eds.) *Global Optimization and Constraint Satisfaction*, *Lecture Notes in Computer Science*, vol. 2861. Springer Berlin Heidelberg. ISBN 978-3-540-20463-3, pp. 211–222. http://dx.doi.org/10.1007/978-3-540-39901-8_16.

Shubert, B. (1972) 'A Sequential Method Seeking the Global Maximum of a Function'. *SIAM Journal on Numerical Analysis*, vol. 9, no. 3, pp. 379–388. http://dx.doi.org/10.1137/0709036.

Stephens, C. P. and Baritompa, W. (1998) 'Global Optimization Requires Global Information'. *Journal of Optimization Theory and Applications*, vol. 96, no. 3, pp. 575–588. http://dx.doi.org/10.1023/A:1022612511618.

Strongin, R. G. and Sergeyev, Y. D. (2000) *Global Optimization with Non-Convex Constraints: Sequential and Parallel Algorithms*. Nonconvex Optimization and Its Applications. Springer. ISBN 978-0-7923-6490-0. http://books.google.co.uk/books?id=xh_GF9Dor3AC.

Teschner, M., Heidelberger, B., Mueller, M., Pomeranets, D. and Gross, M. (2003) 'Optimized Spatial Hashing for Collision Detection of Deformable Objects.' In *Proceedings of Vision, Modeling, Visualization VMV'03*. pp. 47–54.

Zhang, X., Qi, L. and Ye, Y. (2012) 'The Cubic Spherical Optimization Problems'. *Mathematics of Computation*, vol. 81, no. 279, pp. 1513–1525. http://dx.doi.org/10.1090/S0025-5718-2012-02577-4.

A Tables of Results

Function	12 pr.	24 pr.	36 pr.	48 pr.	60 pr.	Initial Gap	Final Gap
Polynomial	8.23	11.93	14.30	16.65	17.33	9.67e+08	1.00e-06
RBF	9.39	14.93	18.80	22.91	24.49	4.78e+08	1.00e-06

Table A.1: *Average speedup* (3.5) *for Algorithm 3.1 (Data Parallel) on Random Polynomial and RBF series three. Also included are the average initial and final tolerances.*

Function	12 pr.	24 pr.	36 pr.	48 pr.	60 pr.	Initial Gap	Final Gap
Polynomial	25.14	38.16	37.38	36.97	35.54	9.67e+08	1.00e-06
RBF	22.01	43.48	52.00	55.73	55.29	4.78e+08	1.00e-06

Table A.2: *Average speedup* (3.5) *for Algorithm 3.2 (Task Parallel) on Random Polynomial and RBF series three. Also included are the average initial and final tolerances.*

Problem	Objective Type	Constraint Type	Problem Type	n	m	nnl
biggsc4	Quadratic	Linear	Academic	4	7	4
biggs5	Sum of Squares	Fixed Variables	Academic	6	1	5
brownden	Sum of Squares	Unconstrained†	Academic	4	0	4
bt3	Sum of Squares	Linear	Academic	5	3	5
ex2_1_1	Quadratic	Linear	Academic	5	1	5
ex2_1_2	Quadratic	Linear	Academic	6	2	5
ex2_1_4	Quadratic	Linear	Academic	6	5	1
ex6_2_10	Other	Linear	Real Application	6	3	6
ex6_2_13	Other	Linear	Real Application	6	3	6
expfita	Other	Linear	Academic	5	22	5
expfitb	Other	Linear	Academic	5	102	5
expfitc	Other	Linear	Academic	5	502	5
hatflda	Sum of Squares	Bound Constrained	Academic	4	0	4
hatfldb	Sum of Squares	Bound Constrained	Academic	4	1	4
hatfldc	Sum of Squares	Bound Constrained	Academic	4	3	4
hatfldh	Quadratic	Linear	Academic	4	7	4
hong	Other	Linear	Academic	4	1	4
hs038	Other	Bound Constrained	Academic	4	0	4
hs041	Other	Linear	Academic	4	5	3
hs045	Other	Bound Constrained	Academic	5	0	5
hs048	Sum of Squares	Linear	Academic	5	2	5
hs049	Other	Linear	Academic	5	2	5
hs051	Quadratic	Linear	Academic	5	3	5
hs052	Quadratic	Linear	Academic	5	3	5
hs053	Quadratic	Linear	Academic	5	3	5
hs054	Other	Linear	Academic	6	1	6
hs055	Other	Linear	Academic	6	6	2
hs086	Other	Linear	Academic	5	10	5
hs268	Quadratic	Linear	Academic	5	5	5
kowosb	Sum of Squares	Unconstrained†	Modelling	4	0	4
lsnnodoc	Other	Linear Network	Academic	5	4	5

Table A.3: *Selected problems from the COCONUT test set with summary statistics: n - number of variables; m - number of constraints; nnl - number of nonlinear variables.* †*Note that we have imposed suitable bounds on these problems so that we can test them using our algorithms.*

Function	12 pr.	24 pr.	36 pr.	48 pr.	60 pr.	Initial Gap	Final Gap	$\|x^p - x^*\|$
biggsc4	1.96	2.20	2.07	2.20	2.18	2.25e+03	1.63e-02	6.22e-04
biggs5	1.40	1.30	1.35	1.32	1.33	3.26e+07	3.33e+05	1.30e+02
brownden	1.67	1.63	1.68	1.69	1.69	1.56e+12	1.02e+09	3.46e+01
bt3	1.58	1.67	1.61	1.74	1.64	3.56e+04	7.69e-02	5.49e-02
ex2_1_1	1.62	1.57	1.64	1.64	1.68	6.27e+03	5.89e+01	1.41e+00
ex2_1_2	4.19	4.79	5.03	5.31	5.26	4.07e+05	5.31e+01	0.00e+00
ex2_1_4	1.43	1.40	1.51	1.43	1.45	9.94e+04	1.17e+02	3.99e-01
ex6_2_10	1.50	1.41	1.40	1.41	1.42	5.06e+02	6.38e+00	4.80e-02
ex6_2_13	1.26	1.29	1.28	1.32	1.37	9.14e+02	1.60e+00	2.50e-01
expfita	2.83	3.23	3.28	3.35	3.52	9.52e+06	8.14e+03	1.61e+01
expfitb	1.22	1.22	1.19	1.23	1.23	2.07e+09	2.09e+06	2.76e+01
expfitc	1.21	1.18	1.24	1.24	1.24	2.60e+11	3.24e+08	1.49e+01
hatflda	4.93	6.04	6.37	6.67	6.76	4.60e+03	1.52e-01	1.68e+00
hatfldb	2.23	2.42	2.47	2.62	2.59	5.74e+03	7.75e-01	1.22e+00
hatfldc	2.18	2.11	2.27	2.24	2.20	4.40e+06	1.11e+03	4.36e+00
hatfldh	2.07	2.17	2.14	2.17	2.12	2.25e+03	1.62e-02	6.22e-04
hong	8.38	9.77	10.02	11.05	10.71	1.01e+06	1.17e-04	2.00e-01
hs038	1.40	1.40	1.47	1.43	1.46	2.43e+09	1.79e+06	1.32e+01
hs041	1.40	1.46	1.45	1.43	1.40	2.20e+02	7.93e-03	1.47e-02
hs045	1.32	1.40	1.36	1.39	1.31	1.13e+03	1.52e+01	1.25e-02
hs048	5.10	6.49	7.23	7.31	7.60	2.40e+04	2.88e-02	2.70e-01
hs049	1.20	1.13	1.17	1.14	1.23	8.95e+08	3.92e+05	7.45e+00
hs051	2.32	2.51	2.49	2.39	2.44	3.56e+04	3.30e-03	1.43e-01
hs052	1.87	1.96	1.97	1.93	1.88	2.36e+05	9.40e-02	4.49e-01
hs053	1.73	1.83	1.91	1.92	1.87	1.44e+05	7.83e-02	3.68e-02
hs054	2.51	2.74	2.83	2.80	3.02	1.13e+05	1.33e+01	7.07e-01
hs055	1.87	1.80	1.95	1.96	1.87	3.24e+04	1.16e-01	4.24e-06
hs086	4.39	5.19	5.45	5.49	5.14	3.46e+05	1.50e-01	3.92e-01
hs268	1.43	1.42	1.48	1.49	1.51	5.84e+08	5.46e+05	5.61e+00
kowosb	3.13	3.49	3.71	3.63	3.76	9.99e+07	5.39e+04	9.13e+00
lsnnodoc	1.29	1.37	1.30	1.39	1.28	1.78e+08	1.89e+04	1.44e+00

Table A.4: *Speedup* (3.5) *for Algorithm 3.1 (Data Parallel) on RBF approximations to selected problems from the COCONUT test set. Also included are the initial tolerance, final tolerance and the distance of the obtained RBF solution to the best known solution for the original problem. All problems were run to the absolute tolerance they achieved in 12 hours on the serial code.*

Function	12 pr.	24 pr.	36 pr.	48 pr.	60 pr.	Initial Gap	Final Gap	$\|x^p - x^*\|$
biggsc4	21.56	36.69	46.10	52.46	53.88	2.25e+03	1.63e-02	6.22e-04
biggs5	11.53	21.22	25.69	30.82	32.79	3.26e+07	3.33e+05	1.30e+02
brownden	24.91	47.94	65.61	78.69	88.87	1.56e+12	1.02e+09	3.46e+01
bt3	15.41	22.54	27.28	29.75	31.59	3.56e+04	7.69e-02	5.49e-02
ex2_1_1	17.58	23.23	24.12	26.06	26.20	6.27e+03	5.89e+01	1.41e+00
ex2_1_2	13.87	19.27	20.21	23.23	24.45	4.07e+05	5.31e+01	0.00e+00
ex2_1_4	8.52	12.51	15.84	15.18	17.02	9.94e+04	1.17e+02	3.99e-01
ex6_2_10	12.69	17.93	21.55	20.19	21.59	5.06e+02	6.38e+00	4.80e-02
ex6_2_13	11.10	14.06	15.21	15.77	17.38	9.14e+02	1.60e+00	2.50e-01
expfita	15.16	24.12	29.68	33.03	36.90	9.52e+06	8.14e+03	1.61e+01
expfitb	14.27	24.18	28.58	34.35	33.77	2.07e+09	2.09e+06	2.76e+01
expfitc	12.95	22.33	27.63	31.91	34.63	2.60e+11	3.24e+08	1.49e+01
hatflda	24.62	47.01	60.92	74.38	83.00	4.60e+03	1.52e-01	1.68e+00
hatfldb	39.05	73.73	95.85	108.16	137.10	5.74e+03	7.75e-01	1.22e+00
hatfldc	17.84	31.74	41.63	47.93	51.83	4.40e+06	1.11e+03	4.36e+00
hatfldh	20.86	36.30	45.91	54.21	56.37	2.25e+03	1.62e-02	6.22e-04
hong	17.88	31.82	40.81	48.55	50.59	1.01e+06	1.17e-04	2.00e-01
hs038	24.19	44.38	59.98	71.19	79.05	2.43e+09	1.79e+06	1.32e+01
hs041	72.57	123.27	155.45	159.58	206.15	2.20e+02	7.93e-03	1.47e-02
hs045	15.66	29.06	33.67	40.41	49.38	1.13e+03	1.52e+01	1.25e-02
hs048	12.00	19.76	25.96	29.37	32.83	2.40e+04	2.88e-02	2.70e-01
hs049	12.08	16.10	17.72	17.49	18.37	8.95e+08	3.92e+05	7.45e+00
hs051	19.64	31.01	38.03	40.68	45.44	3.56e+04	3.30e-03	1.43e-01
hs052	14.03	22.08	25.27	28.21	29.60	2.36e+05	9.40e-02	4.49e-01
hs053	16.90	25.65	30.56	32.43	35.52	1.44e+05	7.83e-02	3.68e-02
hs054	13.26	17.26	20.95	23.66	23.73	1.13e+05	1.33e+01	7.07e-01
hs055	45.09	60.30	61.25	63.20	61.10	3.24e+04	1.16e-01	4.24e-06
hs086	12.81	20.67	25.18	30.05	32.14	3.46e+05	1.50e-01	3.92e-01
hs268	10.86	18.42	22.81	24.18	28.26	5.84e+08	5.46e+05	5.61e+00
kowosb	20.53	34.97	47.75	55.27	61.99	9.99e+07	5.39e+04	9.13e+00
lsnnodoc	59.22	91.96	110.59	120.18	128.07	1.78e+08	1.89e+04	1.44e+00

Table A.5: *Speedup* (3.5) *for Algorithm 3.2 (Task Parallel) on RBF approximations to selected problems from the COCONUT test set. Also included are the initial tolerance, final tolerance and the distance of the obtained RBF solution to the best known solution for the original problem. All problems were run to the absolute tolerance they achieved in 12 hours on the serial code.*

B Diagrams of Processor Load

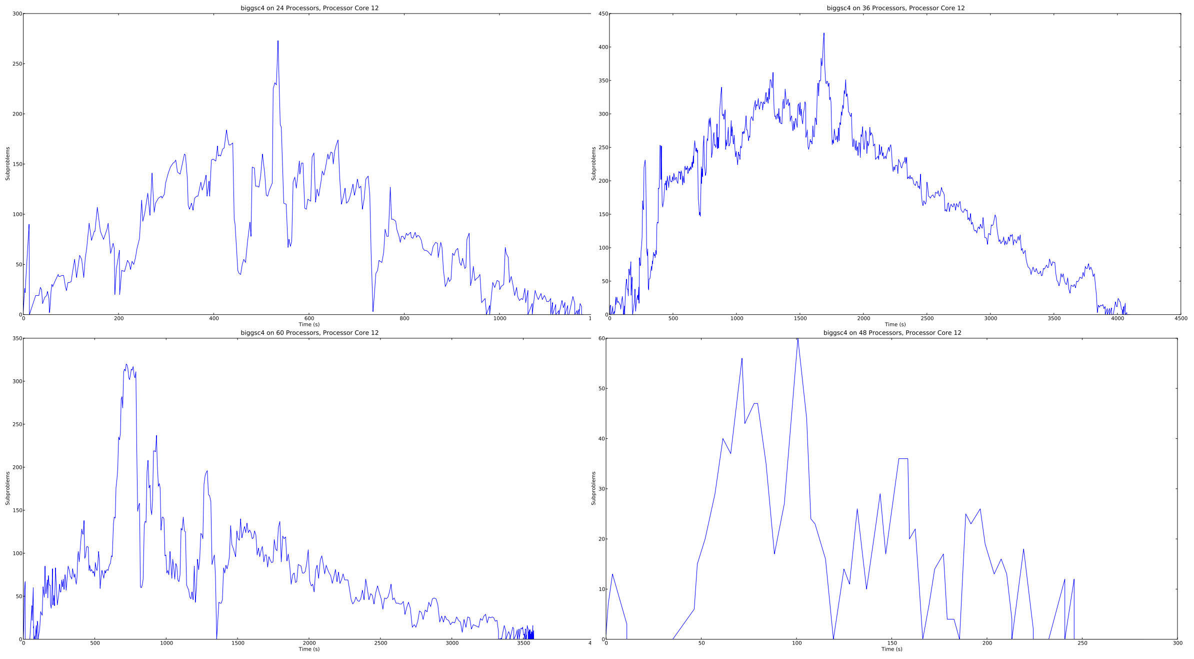

Figure B.1: *Plots of the load against time for processor core no. 12 of the RBF approximation to problem biggsc4 on, clockwise from top left, 24, 36, 48 and 60 processor cores.*

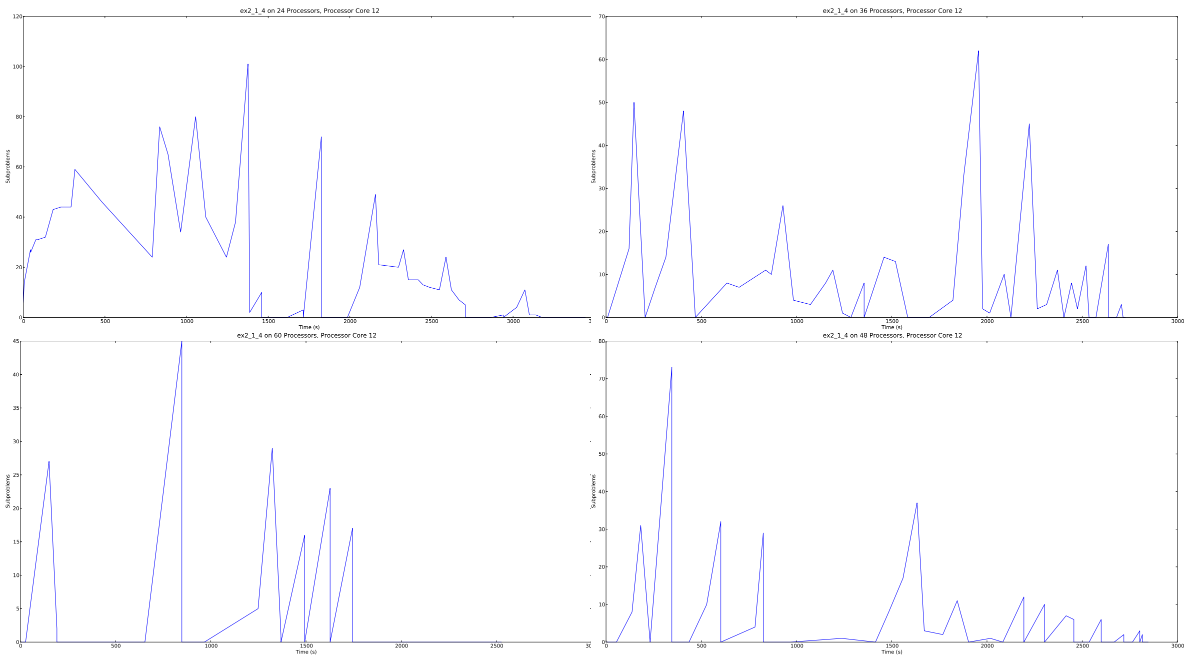

Figure B.2: *Plots of the load against time for processor core no. 12 of the RBF approximation to problem ex2_1_4 on, clockwise from top left, 24, 36, 48 and 60 processor cores.*

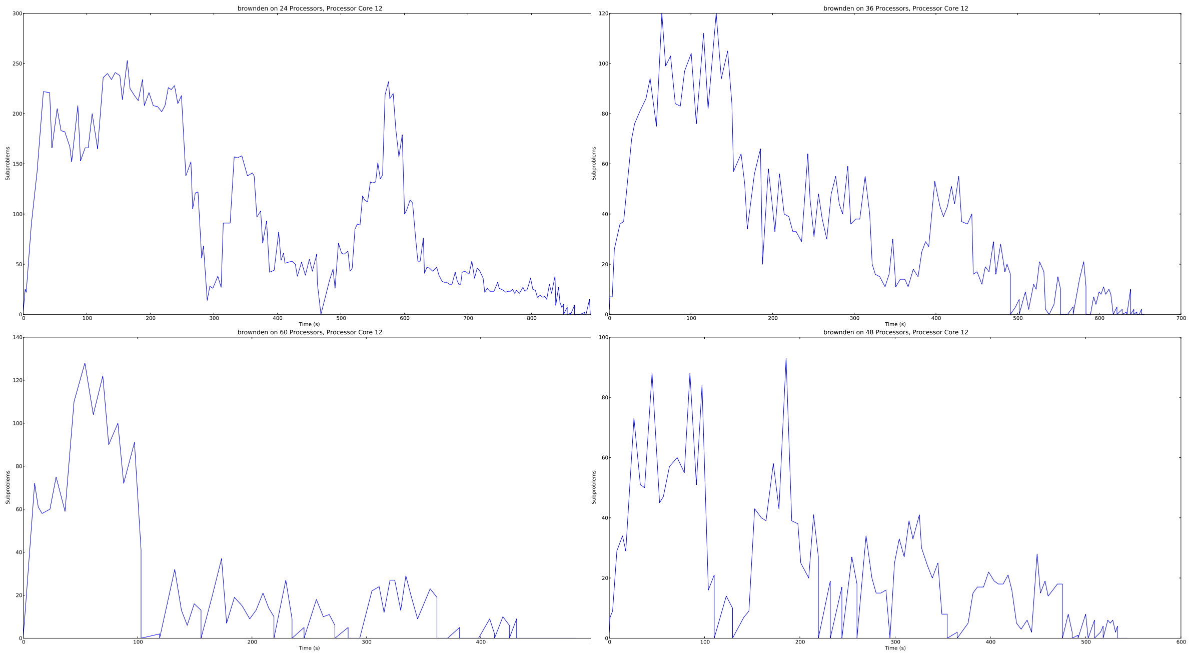

Figure B.3: *Plots of the load against time for processor core no. 12 of the RBF approximation to problem brownden on, clockwise from top left, 24, 36, 48 and 60 processor cores.*