A parallel approach to bi-objective integer programming

W. Pettersson[1] M. Ozlen[2]

Received 23 January 2017; revised 5 October 2017

Abstract

The real world applications of optimisation algorithms often are only interested in the running time of an algorithm, which can frequently be significantly reduced through parallelisation. We present two methods of parallelising the recursive algorithm presented by Ozlen, Burton and MacRae [*J. Optimization Theory and Applications*; 160:470–482, 2014]. Both new methods utilise two threads and improve running times. One of the new methods, the Meeting algorithm, halves running time to achieve near-perfect parallelisation, allowing users to solve bi-objective integer problems with more variables.

DOI:10.21914/anziamj.v58i0.11724, Published 2017-10-13, as part of the Proceedings of the 18th Biennial Computational Techniques and Applications Conference. ISSN 1445-8810. (Print two pages per sheet of paper.)

Contents

1 Introduction

Integer programming (IP) requires either one single measurable objective, or a pre-existing and known mathematical relationship between multiple objectives. If such a relationship, often called a 'utility function', is known then one can optimise this utility function [1, 6]. However, if the utility function is unknown, we must instead identify the complete set of nondominated solutions for the Bi-Objective Integer Programming (BOIP) problem. A decision maker can then more easily see the trade-offs between different options, and therefore make a well informed decision.

Algorithms that determine this complete set can take exact approaches [2], or utilise other techniques (including evolutionary techniques [9]) to approximate

the solution. For further details on multi-objective optimisation see the book by Ehrgott [5].

The performance of BOIP algorithms, and algorithms in general, is often based on the computational time taken to find the solution. This allows algorithms to be compared without needing to use expensive or speciality hardware (as long as comparisons are made on identical hardware setups), but does not take into account real world computing scenarios. Modern computing hardware usually includes multiple cores allowing several threads to run simultaneously. Given this, we consider how to best utilise parallel processing in BOIP algorithms. We constrain ourselves to bi-objective problems to demonstrate the feasibility of our approach.

Existing parallel algorithms in exact bi-objective integer optimisation algorithms partition the solution space based on heuristics or meta-calculations, and then iteratively find solutions inside each partition [3, 4, 10]. In this paper we look at a BOIP algorithm from Ozlen, Burton and MacRae [7] which calculates a solution by recursively solving constrained lexicographic IP problems. We parallelise this by considering different lexicographic orderings simultaneously. Our new parallel algorithm achieves near-ideal parallelisation, calculating a solution in half the running time without incurring any additional computational time. Comparisons with other algorithms show that this is an effective technique across problems of all sizes.

Section 2 of this paper gives a background in optimisation. A brief outline of the recursive algorithm we build upon is given in Section 3. Section 4 describes our parallel computing approach. Section 5 details our software implementation and gives running time comparisons between the original algorithm, the original algorithm with CPLEX parallelisation, and our parallelisations.

2 Background

In an IP problem, we are given a set of variables and a set of linear inequalities called *constraints*. An assignment of an integer value to each variable is called feasible if it satisfies all constraints, and an assignment which does not satisfy all constraints is called infeasible. The set of all feasible vectors we will call the *feasible set*, and can be defined as follows.

Definition 1. The *feasible set* of an IP problem is given by

$$X = \{\mathbf{x} \in \mathbb{Z}^n | A\mathbf{x} = \mathbf{b}, x_j \geqslant 0 \text{ for } j \in \{0, 1, \ldots, n-1\}\}$$

where A is an n-by-n matrix and $\mathbf{b}$ is an n-by-1 matrix that together represent the linear constraints of the problem.

Inequalities can be converted to equalities with the introduction of slack variables (see the book by Papadimitriou and Steiglitz [8] or any introductory linear programming text).

Given a feasible set X and an objective function f, the goal of an IP problem is to find the solution $\mathbf{x} \in X$ that optimises $f(x)$. In this paper we assume that all objective functions are to be minimised. In some scenarios the goal is to maximise a given objective function. Such a problem can trivially be converted into an equivalent problem where the objective is to be minimised. We will denote an IP problem (and various derived problems) as $\mathcal{P}$.

In a multi-objective integer programming problem (MOIP) we do not have a single objective function but rather a set of objective functions. The goal then is to determine all *nondominated* (or *Pareto optimal*) solutions.

Definition 2. Given a pair f_1, f_2 of objective functions $f_i : X \to \mathbb{R}$, a solution $\mathbf{x} \in X$ is considered *nondominated* (or *Pareto optimal*) if there does not exist an $\mathbf{x'} \in X$ with $\mathbf{x'} \neq \mathbf{x}$ such that $f_i(\mathbf{x'}) \leqslant f_i(\mathbf{x})$ for all $i \in \{1, 2\}$.

A *bi-objective integer programming* problem then involves the calculation of the set of all nondominated feasible solutions.

Given a set of objective functions, one of the simpler methods of generating a related single-objective IP problem is to apply an ordering to the objective functions, and compare solution vectors by considering each objective in order. That is, each objective function is considered in turn, with objective functions that appear earlier in the ordering being given a high priority. We will call such a problem a *lexicographic bi-objective integer programming* (LBOIP) problem on k objectives.

Our parallel algorithm will use different orderings of a set of objective functions to determine the solution set. We therefore introduce the following notation to refer to different lexicographic variants of a BOIP problem with a given set of objectives.

Notation 3. If a lexicographic version of the BOIP problem $\mathcal{P}$ will order objectives according to the ordered set (f_1, f_2), we will write $\mathcal{P}^{(1,2)}$.

The optimal solution for a LBOIP problem will be part of the solution set for the related BOIP problem, but there is no guarantee that it will be the only solution for the BOIP problem. Indeed, it will often not be the only nondominated solution. To determine all nondominated solutions, the algorithm described in Section 3 utilises *constrained lexicographic multi bi-objective linear programming* (CLBOIP). A CLBOIP problem is simply a LBOIP problem with a constraint on the second objective function. These constraints limit the solution space to some given bound.

Notation 4. Given an LBOIP problem $\mathcal{P}^{\mathbf{s}}$, if the upper bound on the second objective is l_k we will denote the CLBOIP problem by $\mathcal{P}^{\mathbf{s}}(< l_k)$.

Example 5. Given a BOIP problem $\mathcal{P}$ with objective functions f_1 and f_2, the CLBOIP problem $\mathcal{P}^{(1,2)}(< 15)$ is then the problem

$$\begin{aligned} \text{Minimise} \quad & f_2(x) \\ \text{s.t} \quad & f_1(x) = \min\{f_1(x) | x \in X \text{ and } f_2(x) < 15\} \\ \text{and} \quad & f_2(x) < 15 \end{aligned}$$

whereas the CLBOIP problem $\mathcal{P}^{(2,1)}(< 18)$ is the problem

$$\begin{aligned} \text{Minimise} \quad & f_1(x) \\ \text{s.t} \quad & f_2(x) = \min\{f_2(x) | x \in X \text{ and } f_1(x) < 18\} \\ \text{and} \quad & f_1(x) < 18 \ . \end{aligned}$$

3 The algorithm of Ozlen, Burton and MacRae

The full recursive algorithm given by Ozlen, Burton and MacRae [7] is suitable for problems with an arbitrary number of objective functions. Algorithm 1 describes a bi-objective version of the algorithm. For a complete introduction to the algorithm, see [7].

The correctness of this algorithm is readily shown by induction. For a formal proof of the correctness of this algorithm, see [7].

Algorithm 1: A simple overview of the bi-objective version of the algorithm of Ozlen, Burton and MacRae.

Data: A BOIP $\mathcal{P}$ with objective functions f_1 and f_2
Result: The nondominated solutions to the BOIP
Let $S = \{\}$ be an empty set to which we will add all nondominated solutions;
Let $l_2 = \infty$;
while $\mathcal{P}^{(1,2)}(< l_2)$ *is feasible* **do**
  Let $\boldsymbol{\nu} = (\nu_1, \nu_2)$ be the optimal vector for the CLBOIP problem $\mathcal{P}^{(1,2)}(< l_2)$;
  Add $\boldsymbol{\nu}$ to S;
  Set $l_2 = \nu_2$;
end

4 Parallelisation

We utilise parallelisation to reduce the elapsed running time of optimisation algorithms. Here the term *thread* denotes a single computational core performing a sequence of calculations. In a parallel algorithm, multiple *threads* perform multiple calculations simultaneously. In this paper we investigate improvements gained by utilising two threads.

4.1 Range splitting

When solving $\mathcal{P}$, it is clear that the maximum and minimum values of $\mathtt{f}_2(\mathbf{x})$ can easily be determined by minimising $\mathtt{f}_1$ and $\mathtt{f}_2$ respectively. One naïve method of distributing this problem across $\mathtt{t}$ threads would be to split this range into $\mathtt{t}$ equal sized pieces, and then adding an upper and lower limits on $\mathtt{f}_2$ to the specific sub-problem solved by each thread. These results can be combined in the obvious manner to give the solution. We will refer to this algorithm as the *Splitting* algorithm, as the range of $\mathtt{f}_2$ is split up so that each thread gets a single section. The proof of correctness of this algorithm is trivial. Implementation and timing results are detailed in Section 5.

4.2 Efficient parallelisation

Whilst Algorithm 2 is parallel, there is no guarantee that all threads will perform an equal (or near-equal) amount of work. Indeed, it is easy to visualise problems where one thread will perform far more work than another. Instead we use an algorithm which dynamically adapts itself as the solution set is found.

Recall that in Algorithm 1 we used the specific ordering $\mathcal{P}^{(1,2)}$. We could also solve $\mathcal{P}^{(2,1)}$ and obtain the same result. This idea forms the basis of our work. We show below how the bound $\mathtt{l}_2$ obtained from $\mathcal{P}^{(1,2)}(< \mathtt{l}_2)$ is able

Algorithm 2: Our range-splitting algorithm, utilising the MOIP algorithm from [7].

Data: A BOIP problem $\mathcal{P}$ with objective functions f_1 and f_2, and T representing the number of threads to use
Result: The nondominated solutions to the BOIP
Calculate $u = \min\{f_1(x)|x \in X\}$ and $l = \min\{f_2(x)|x \in X\}$;
Let $step = (u - l)/2$;
foreach $t \in \{1, \ldots, T\}$ **do**
  Let $LB = l + (t-1) * step$ and $UB = l + t * step$;
  Create $\mathcal{P}'$ as a copy of $\mathcal{P}$;
  Add constraints $f_2(x) < UB$ and $f_2(x) > LB$ to $\mathcal{P}'$;
  Solve $\mathcal{P}'$ in a new thread using the algorithm of [7] in a separate thread;
end
return *The union of all solutions returned by all threads.*

to be shared with the problem $\mathcal{P}^{(2,1)}(< l_1)$. This allows the two problems to be solved simultaneously, which almost halves the running time of our new algorithm when compared with the original.

Theorem 6. *If we have the complete set S of nondominated solutions for $\mathcal{P}^{(1,2)}$ with $x_2 \geqslant k$, the complete set S' of nondominated solutions for $\mathcal{P}^{(2,1)}$ with $x_1 \geqslant l$, and we also have the nondominated solution (l, k), then the union $S \bigcup S' \bigcup \{(l, k)\}$ is the complete set of nondominated solutions to $\mathcal{P}$.*

Proof: Assume that $\boldsymbol{\nu} = (\nu_1, \nu_2)$ is a nondominated solution to $\mathcal{P}$ that is not in either S nor S'. If $x_1 > l$ then $\boldsymbol{\nu} \in S'$, a contradiction. Similarly, if $\nu_2 > k$ then $\boldsymbol{\nu} \in S$ which is also a contradiction. Therefore $\nu_1 \leqslant l$ and $\nu_2 \leqslant k$, but then as (l, k) is nondominated, the only solution is $(\nu_1, \nu_2) = (l, k)$. ♠

The CLBOIP problems $\mathcal{P}^{(1,2)}$ and $\mathcal{P}^{(2,1)}$ can be solved independently by Algorithm 1, and (l, k) will be found as a solution to these problems. Given

Algorithm 3: Our Meeting algorithm, which is a parallel version of the algorithm from [7]. In the line marked *, the value $\mathtt{l}_{\mathtt{t}'}$ is shared between the two threads.

Data: A BOIP problem $\mathcal{P}$ with objective functions $\mathtt{f}_1$ and $\mathtt{f}_2$
Result: The nondominated solutions to $\mathcal{P}$
Let $\mathtt{t} \in \{1, 2\}$, and let $\mathtt{t}'$ be the unique value in $\{1, 2\} \setminus \{\mathtt{t}\}$;
Let $\mathtt{s}_1 = (2, 1)$ and $\mathtt{s}_2 = (1, 2)$;
Let $\mathtt{S}_1 = \mathtt{S}_2 = \{\}$ be empty sets;
Let $\mathtt{l}_1 = \mathtt{l}_2 = \infty$;
foreach *thread* $\mathtt{t}$ **do**
  while $\mathcal{P}^{\mathtt{s}_\mathtt{t}}(< \mathtt{l}_\mathtt{t})$ *is feasible* **do**
    Let $\boldsymbol{\nu} = (\nu_1, \nu_2)$ be the solution for the CLBOIP problem $\mathcal{P}^{\mathtt{s}_\mathtt{t}}(< \mathtt{l}_\mathtt{t})$;
    Add $\boldsymbol{\nu}$ to $\mathtt{S}_\mathtt{t}$;
    Set $\mathtt{l}_\mathtt{t} = \nu_\mathtt{t}$;
    Add $\nu_{\mathtt{t}'} < \mathtt{l}_{\mathtt{t}'}$ as a constraint to $\mathcal{P}^{\mathtt{s}_\mathtt{t}}(< \mathtt{l}_\mathtt{t})$ *;
  end
end
return $\mathtt{S}_1 \cup \mathtt{S}_2$

this result, we propose the parallel algorithm, Algorithm 3, for computing the solution to BOIP problems.

We refer to this algorithm as the *Meeting* algorithm, as the two threads meet in the middle to complete the calculations. The correctness of the Meeting algorithm follows from Theorem 6 and the correctness of Algorithm 1.

5 Implementations and running times

5.1 Implementation

Our algorithms were implemented in C++, and are available at https://github.com/WPettersson/moip_aira. All calculations were run on the NCI supercomputing cluster Raijin, on nodes consisting of two Intel Sandy Bridge E5 2670 processors and 32GB of RAM. Code was compiled with GCC 4.9, using no special optimisation controls beyond `-O2`. We compared the elapsed running time (and not CPU time) of the original algorithm (with both one thread allocated to CPLEX, and two threads allocated to CPLEX), along with the *Splitting* algorithm and the *Meeting* algorithm. These tests utilised assignment and knapsack problems to give a broad overview of the performance of our algorithms. These running times are summarised in Table 1.

Allocating a second thread to CPLEX reduced running times slightly, with an average reduction to 95% of the running time of the original algorithm of Ozlen, Burton and MacRae [7]. It is not surprising that CPLEX does not parallelise efficiently in this manner, as CPLEX cannot take advantage of the full details of the algorithm used. The Splitting algorithm was more impressive, showing an average reduction to 76% of the original running time.

Our Meeting algorithm is the clear outlier, running twice as fast as the original algorithm of Ozlen, Burton and MacRae [7] on all problems tested. We expect similar performance increases on all problem types. On some problems our Meeting algorithm was more than twice as fast as the original algorithm. This occurs when a given solution can be found faster by first minimising f_2 and then f_1, rather than first minimising f_1 and then f_2. This behaviour seemed to be more common in the knapsack problems.

Table 1: Elapsed running timing comparisons of the four algorithms. We ran ten different random versions of each sized problem and averaged the results.

Assignment problems				
# tasks	Ozlen et al.	CPLEX	Splitting	Meeting
40	1.1×10^1	1.1×10^1	9.1×10^0	5.7×10^0
60	3.4×10^1	3.2×10^1	2.9×10^1	1.8×10^1
80	6.8×10^1	5.8×10^1	5.6×10^1	3.6×10^1
100	1.2×10^2	1.1×10^2	9.6×10^1	6.3×10^1
200	5.2×10^2	4.5×10^2	4.0×10^2	2.8×10^2
500	3.3×10^3	3.5×10^3	2.3×10^3	1.7×10^3
Knapsack problems				
# items	Ozlen et al.	CPLEX	Splitting	Meeting
50	1.0×10^0	1.1×10^0	6.7×10^{-1}	5.3×10^{-1}
100	5.0×10^1	4.8×10^1	3.6×10^0	2.6×10^0
200	2.2×10^1	2.1×10^1	1.6×10^1	1.2×10^1
400	7.4×10^1	7.2×10^1	5.8×10^1	3.6×10^1
1000	3.4×10^2	3.5×10^2	2.6×10^2	1.5×10^2
2000	1.2×10^3	1.1×10^3	9.1×10^2	5.3×10^2

6 Conclusion

We successfully implemented two parallel algorithms to solve bi-objective optimisation problems. Both improved performance, with one showing ideal performance increase. For bi-objective problems (and potentially multi-objective problems) this faster algorithm allows solutions to be found in half the time. Ongoing work in this field will look at various ways of utilising more threads in parallel to further improve running times for IP problems with three or more objectives. This could utilise elements from the symmetric group S_n (where $n > 2$) to allocate the different orderings of the objective functions to various threads, but this will make the sharing of information

more complicated and care must be taken to ensure that any new algorithm does exactly determine all nondominated solutions.

Acknowledgements This study is supported by the Australian Research Council under the Discovery Projects funding scheme (project DP140104246).

References

[1] M. Abbas and D. Chaabane. Optimizing a linear function over an integer efficient set. *European Journal of Operational Research*, **174**:1140–1161, 2006. doi:10.1016/j.ejor.2005.02.072 C70

[2] H. P. Benson. An outer approximation algorithm for generating all efficient extreme points in the outcome set of a multiple objective linear programming problem. *Journal of Global Optimization*, **13**:1–24, 1998. doi:10.1023/A:1008215702611 C70

[3] N. Boland, H. Charkhgard and M. Savelsbergh. The triangle splitting method for biobjective mixed integer programming. In J. Lee and J. Vygen editors *Integer Programming and Combinatorial Optimization 2014*, Lecture Notes in Computer Science, **8494**. Springer, Cham, 2014, pp. 162–173. doi:10.1007/978-3-319-07557-0_14 C71

[4] C. Dhaenens, J. Lemesre and E. G. Talbi. K-PPM: A new exact method to solve multi-objective combinatorial optimization problems. *European Journal of Operational Research*, **200**:45–53, 2010. doi:10.1016/j.ejor.2008.12.034 C71

[5] M. Ehrgott. *Multicriteria Optimization*, Lecture notes in economics and mathematical systems, Springer, 2005. doi:10.1007/3-540-27659-9 C71

[6] J. M. Jorge. An algorithm for optimizing a linear function over an integer efficient set. *European Journal of Operational Research*, **195**:98–103, 2009. doi:10.1016/j.ejor.2008.02.005 C70

[7] M. Ozlen, B. A. Burton and C. G. MacRae. Multi-objective integer programming: an improved recursive algorithm. *Journal of Optimization Theory and Applications*, **160**:470–482, 2014. doi:10.1007/s10957-013-0364-y C71, C74, C76, C77, C78

[8] C .H. Papadimitriou and K. Steiglitz *Combinatorial Optimization: Algorithm and Complexity*, Prentice Hall, 1982. doi:10.1109/TASSP.1984.1164450 C72

[9] K. E. Parsopoulos and M. N. Vrahatis. Particle swarm optimization method in multiobjective problems. In *SAC '02: Proceedings of the 2002 ACM Symposium on Applied Computing*. ACM, New York, 2002, pp. 603–607. doi:10.1145/508791.508907 C70

[10] A. Przybylski and X. Gandibleux. Multi-objective branch and bound. *European Journal of Operational Research*, **260**:856–872, 2017. doi:10.1016/j.ejor.2017.01.032 C71

Author addresses

1. **W. Pettersson**, School of Science, RMIT University, Victoria 3000, Australia.
mailto:william@ewpettersson.se

2. **M. Ozlen**, School of Science, RMIT University, Victoria 3000, Australia.
mailto:melih.ozlen@rmit.edu.au