Operations Research Letters 45 (2017) 652–658

Contents lists available at ScienceDirect

Operations Research Letters

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

Generation of feasible integer solutions on a massively parallel computer using the feasibility pump

Utku Koc [a,b,*], Sanjay Mehrotra [a]

[a] Northwestern University, Evanston, IL, USA
[b] MEF University, Istanbul, Turkey

ARTICLE INFO

Article history:
Received 19 July 2016
Received in revised form 3 October 2017
Accepted 3 October 2017
Available online 19 October 2017

Keywords:
Mixed integer programming
Parallel optimization
Feasibility pump

ABSTRACT

We present an approach to parallelize generation of feasible mixed integer solutions of mixed integer linear programs in distributed memory high performance computing environments. This approach combines a parallel framework with feasibility pump (FP) as the rounding heuristic. It runs multiple FP instances with different starting solutions concurrently, while allowing them to share information. Our computational results suggest that the improvement resulting from parallelization using our approach is statistically significant.

1. Introduction

In this study we consider the problem of generating high quality feasible solutions for unstructured Mixed Integer Linear Programs (MILPs) in a parallel computational environment. We refer the interested reader to Lodi [18] for a recent review of MILP literature. Generating high quality feasible solutions quickly is important in practice. Additionally, availability of feasible solutions with close to optimal objective value, may help reduce the number of nodes in the branch and bound (B&B) tree in a branch and cut algorithm. We propose a scheme that can use multiple heuristics with various parameter settings in parallel. Specifically, we empirically investigate the use of Feasibility Pump (FP) to find feasible solutions for unstructured MILPs in a parallel framework.

The motivation of this study is the emerging computing environment. The clock speed of the high-tech processors is more or less stable for the past few years. Computer technology is now mainly focused on increasing the number of processors and memory. With this in mind, we move to a new era of developing parallel algorithms for a variety of problems for desktop and high performance computing (HPC). From a practical point of view, it is important to solve a problem or identify a good solution within a reasonable amount of wall-clock time, de-emphasizing the CPU-time used.

For MILPs, a way to use the power of parallel computing is to search the branch and bound tree in parallel. Koch et al. [17] discuss that the speed up of a B&B algorithm is around 20,000 compared to a sequential run, even if a million cores are used to search the B&B tree. They discuss that the dis-proportionality in the performance is mainly due to the communication overhead, performance effect of the redundant work, and idle time due to latency/contention/starvation.

The FP algorithm is a constructive heuristic for MILPs that starts from a feasible solution x of the continuous relaxation, searches for another solution $\hat{x}$ that is as close as possible to a rounded solution of x (which is infeasible but integral) by solving an ℓ_1 norm minimization problem. The algorithm continues until a feasible integral solution is found. The FP heuristic was first proposed by Fischetti et al. [10] for 0–1 MILPs. An extension to general MILPs is proposed by Bertacco et al. [7]. By a modification of the objective function, Achterberg and Berthold [1] found better feasible solutions (Objective FP). Fischetti and Salvagnin proposed a different rounding heuristic by using constraint propagation techniques after rounding some of the variables [13]. Baena and Castro [3] extended the FP, so that the integer point is obtained by rounding a point on the (feasible) line segment between the computed feasible point and the analytic center for the relaxed LP. Huang and Mehrotra studied a combination of different types of random walks and FP in which the FP algorithm is used as the rounding procedure for interior random points. They generate feasible solutions for MILPs [14] and Mixed Integer Convex Programs (MICPs) [15]. Fischetti et al. studied to increase the stability of the root node LP solutions through parallel independent runs with diverse initial conditions [12]. Munguia et al. [19] combine the use of parallelization and simple large neighborhood search schemes to generate feasible solutions for MILPs.

* Corresponding author at: MEF University, Istanbul, Turkey.
E-mail addresses: utku.koc@mef.edu.tr (U. Koc), mehrotra@northwestern.edu (S. Mehrotra).

https://doi.org/10.1016/j.orl.2017.10.003
0167-6377/

There are several other heuristics for finding feasible solutions for MILP problems that can be used as a part of a parallel implementation. Among them, Pivot-and-Complement [5,6] performs simplex like pivots to get slack variable into the basis and integer variables out of a basis. Another heuristic for 0–1 MILP is OCTANE, which uses enumeration techniques on extended facets of the octahedron [4]. Fischetti and Lodi propose a local search algorithm [11] to improve an incumbent solution. Relaxation Induced Neighborhood Search (RINS) heuristic solves sufficiently smaller sub-MILPs to improve an incumbent solution [9], and the use of random-walks was investigated in [14,15].

In this study, we provide a parallel framework in which multiple feasibility heuristics starting from different solutions can communicate and share information. We assess the value of parallelization independent of the increase in the CPU-time and provide a parallel framework that can use multiple parameters for feasibility heuristics. Each parallel subroutine uses a different rounding scheme so that the most fractional variables are rounded in an enumerative fashion independently. This study is the first of its kind in terms of using many cores to generate feasible integer solutions in parallel in a distributed memory environment with many cores.

The rest of the paper is organized as follows: we describe our parallel heuristic framework in Section 2. Details of the rounding procedure are given in Section 3. Section 4 gives the implementation details of the proposed algorithms. The computational results are given in Section 5. We show that the proposed approach is effective up to 128 cores in today's HPC environment. Finally, we conclude in Section 6.

2. A concurrent framework for finding feasible solutions for MILPs

In our approach, we run multiple feasibility heuristics in parallel. We refer to the algorithms running in different processors as subroutines. Each parallel subroutine uses a different random number seed with different starting solutions. One may also run different feasibility heuristics in parallel. Moreover, any combination of parameter settings, rounding methods, and anti-cycling rules are also valid. Note that, even if all the subroutines start from the same solution and run independently, final integer solutions may still be different. This is because multiple instances can take different paths in the course of the parallel subroutines. Whenever one of the subroutines finds a feasible solution, it broadcasts the objective function value to others via the master. Then, all subroutines continue their search with a new and better objective cut-off constraint. Thus, the information gained in one of the subroutines is shared with the rest to enhance their search. This is an important feature of our concurrent optimization approach. All subroutines update themselves as soon as the first feasible solution is found. In this study, as a proof of concept, we use FP as the rounding procedure of the subroutines of our concurrent feasibility heuristic.

Regarding the communication during the run time, one may use the so called master/slave topology. In this paradigm, the master controls the overall course of the algorithm. Slave programs, on the other hand, follow the commands from the master, run the instances of the heuristic, and return integer solution(s) to the master, if any. The role of the master is distributing inputs to and collecting results from the slaves. The main algorithm that runs at the master is presented in Algorithm 2.1.

We illustrate the algorithm running at the slaves in Algorithm 2.2. Each slave uses a different random number seed and may run a different variant of a heuristic. At each iteration of Algorithm 2.2 , the slave subroutine receives relevant information from the master (if any), updates itself with the new information, creates a starting solution for the algorithms depending on the type of heuristic it is running and sends the integer solutions to the master, if any.

Algorithm 2.1 *Parallel Feasibility-Pump Running in Master*

Input: a MILP $\min\{c^T x : Ax \geq b, x \in \mathbb{R}^n, x_j \text{ integer } \forall j \in I\}$, number of slaves each heuristic will run
Output: an integer solution to the above MILP
1: Spawn Slaves
2: Set $LB = \min\{c^T x : Ax \geq b, x \in \mathbb{R}^n\}$ and $UB = \infty$
3: Inform slaves about heuristic to run and LB
4: **while** termination criteria not met **do**
5: Collect results
6: **if** One of the slaves returns an integer solution **then**
7: Update UB = minimum of the slaves
8: Inform slaves about the new UB
9: **end if**
10: **end while**
11: Exit all the slaves and return best integer so far

Algorithm 2.2 *Parallel Heuristic Subroutine Running in Slaves*

Input: a MILP, UB
Output: an integer solution to the MILP
1: Receive the type of the heuristic to run and LB form the Master
2: **while** not exited by the master **do**
3: Listen to the master for information (UB)
4: Update RHS of the objective cut-off constraint
5: Update with respect to the heuristic variant
6: Create a starting solution x
7: Run heuristic starting from x
8: Broadcast best integer solution
9: **end while**

The heuristic subroutine continues until predetermined criteria are met, or the master provides new information.

The variants of the heuristic subroutines differ in Steps 5–7 of Algorithm 2.2. The update procedure, generations of starting solutions, and running conditions of the heuristics depend on the heuristic itself and information provided by the master.

3. Variants of FP heuristic

In this section we describe the details of the rounding subroutine, as well as the generation of the starting solutions for rounding. We start with the details of the basic FP algorithm as the rounding procedure.

3.1. Basic and objective FP algorithms

The FP algorithm starts from a solution x, searches for another solution $\hat{x}$ that is as close as possible to a rounded solution of x ($\tilde{x}$) by solving an ℓ_1 norm minimization problem of the form:

$$\min \left\{ \Delta(x, \tilde{x}) = \sum_{j \in I} |x_j - \tilde{x}_j| : Ax \geq b, c^T x \leq RHS, x \in \mathbb{R}^n \right\}$$

where $\Delta(x, \tilde{x})$ is the ℓ_1 norm distance, $Ax \geq b$ is the constraint set defined by the original MILP and $c^T x \leq RHS$ is the objective cut-off constraint. For problems with general integer variables, Δ is defined by adding artificial variables. At each attempt to solve the problem, ℓ_1 norm distance function Δ is updated with respect to the rounded solution $\tilde{x}$. In this heuristic, one needs to decide on how the starting solutions (x^k) and rounding procedure ($\tilde{x}$) are defined at each iteration k.

Using a normalized convex combination of the original objective function and the above ℓ_1 norm objective, one can generate better quality solutions (Objective-FP) [1]. The idea is to focus more

on the objective value quality in the beginning of the algorithm, and feasibility at the later stages by controlling the parameter $\alpha \in (0, 1)$. For this purpose, the objective function of the above MILP is replaced by

$$\frac{1-\alpha}{\|\Delta(x, \tilde{x}^{k-1})\|}\Delta(x, \tilde{x}^{k-1}) + \frac{\alpha}{\|c\|}c^T x, \tag{1}$$

where Δ is the ℓ_1 norm distance, c is the original objective vector and $\|\cdot\|$ is the euclidean norm. The parameter α reduces gradually at each iteration of the Objective FP algorithm provided in Algorithm 3.1.

We refer to the process of solving the problem of minimizing the convex combination defined in (1) as an FP iteration. The original FP algorithm starts from an optimal solution of the relaxation problem and rounds it to the nearest integer. We refer to a solution $\hat{x}$ to be an integer solution if $\hat{x}_j$ is *integer* for all $j \in I$. If the FP iteration terminates with an integer solution, we have a feasible solution for the original MILP. The objective function value of this solution is fed back to the model as an artificial objective cut-off constraint $c^T x \leq UB - \epsilon$, where UB is the objective function value of the best incumbent solution so far and ϵ is the improvement coefficient. The objective cut-off constraint is used to find solutions with improved objective. If the objective value of a problem is known to be an integer, then $\epsilon = 1$, else one needs to set ϵ to a small tolerance (we use $\epsilon = 0.1$). If the solution of the FP iteration is not integer, the next iteration starts from an optimal solution of the ℓ_1 norm minimization problem with the same rounding scheme. The algorithm terminates if time/iteration limit is reached or an optimal solution for MILP is found. Depending on the choice of the starting solutions and the rounding scheme, multiple FP variants can be defined.

Algorithm 3.1 *Objective Feasibility Pump for MILP*

Input: a MILP $\min\{c^T x : Ax \geq b, x_j \text{ integer } \forall j \in I\}$
Output: an integer solution to the above MILP

1: Initialize $k = 0$, $LB = \min\{c^T x : Ax \geq b\}$, $UB := \infty$
2: Set $x^k := \arg\min\{c^T x : Ax \geq b, c^T x \leq UB - \epsilon\}$.
3: **if** x^k is integer **then**
4: return x^k
5: **else**
6: let $\tilde{x}^k := [x^k]$ (= rounding of x^k)
7: **end if**
8: **while** termination criteria not met **do**
9: k := k + 1
10: $\alpha = \alpha \cdot \alpha_r$
11: compute $x^k := \arg\min\{\frac{1-\alpha}{||\Delta||}\Delta(x, \tilde{x}^{k-1}) + \frac{\alpha}{||c||}c^T x : Ax \geq b, c^T x \leq RHS\}$
12: **if** x^k is integer **then**
13: set $UB := c^T x^k$ and go to step 2.
14: **end if**
15: **if** $\exists j \in I : [x_j^k] \neq \tilde{x}_j^{k-1}$ **then**
16: set $\tilde{x}^k := [x^k]$
17: **else**
18: set $\tilde{x}^k = \tilde{x}^{k-1}$, flip entries $\tilde{x}_j^k$ $(j \in I)$ randomly
19: **end if**
20: **end while**

In the original implementation by Fischetti et al. [10], whenever a cycle is heuristically detected, a random perturbation is applied by skipping Step 15 and directly moving to Step 18. In the cycle breaking perturbation, a random number of indices among the most fractional entries are flipped. Note that, depending on the flipping (rounding) scheme, the output of the algorithm can significantly change. Additionally, using different random number streams may have a significant impact on the performance of the solutions generated by the FP algorithm.

4. Implementation details

We now describe the implementation details for our concurrent optimization framework and FP. The details of the computational environment are also given.

4.1. Parallel FP implementation

For the master/slave paradigm, we use MPI 3 (Message Passing Interface) standards to ensure scalability. We use a Mersenne twister random number generator at each slave. Each slave uses a different random number stream, fed by the master. The seeds are generated using a linear congruential method. If a feasible integer solution is found, the objective cut-off constraint is updated by each slave for itself, it then sends the objective function value to the master process, which in turn broadcasts the best of the slave objectives to other processors.

The implementation allows any combination of mixing multiple FP variants in a parallel setting. Multiple termination criteria are implemented, however, we share the results with a wall clock time limit. In a parallel setting where multiple variants are used, there may be significant differences in the completion time of a fixed number of iterations and/or CPU-time. To get a maximum computational advantage, we allow all parallel subroutines to complete at the given wall clock time. Also, the algorithm terminates if an optimal solution is found by one of the slaves. Slaves compare the solution objective function value with the lower bound to prove optimality.

4.2. Implementation of the rounding heuristics

We speculate that the challenge in solving difficult mixed binary problems comes from the effectiveness in picking a correct branching variables at an early stage. We think that the same difficulty continues with FP when initializing with a certain rounding of the fractional solutions of the LP relaxation solution. It is surmised that the enumeration of the most important fractional variables in parallel in FP provides the same gains as one expects in a parallel branch and bound that is based on enumerating difficult fractional variables. Our implementation of the rounding heuristic is based on this idea.

The algorithm is split into three basic stages, (1) Start point generation, (2) Rounding, and (3) Communication.

Stage 1 — Start point generation: The starting solution for the original FP algorithm is an optimal solution for the continuous relaxation. In our implementation, the most fractional k variables are set to its floor or ceiling by different subroutines, k depending on the total number of CPUs. If the parallelization level is 2^k, the most fractional k variables are enumerated by considering both floor and ceiling of the values.

Stage 2 — Rounding: the FP algorithm is implemented at the rounding stage. The details are similar to the original FP implementation by Fischetti et al., except the internal improvement – where the objective cut-off constraints are implemented, and the branching phases – in which solutions are enumerated if no solution is found. All other parameters are used at the default values. Moreover, when a feasible solution is found, the solution is polished by fixing all the integer variables and re-optimizing on the continuous variables, if any. The aim of polishing is to find better solutions in terms of objective function value, by keeping all the integral variables fixed at their current levels (maintaining feasibility) and optimizing only on continuous variables.

Stage 3 — Communication: As soon as a feasible solution is found by one of the parallel subroutines, it is shared by the master. The master then updates all slaves with the new incumbent's objective function value. After each rounding iteration, the slaves check if there exists a new incumbent solution and update the objective cut-off constraint, if necessary.

Table 1
Number of problems for which basic FP finds a feasible (best-known value attained) solution (total = 162).

	10t	20t	40t	80t	160t	320t	640t	1280t	2560t
1	69 (6)	92 (9)	109 (11)	116 (14)	121 (17)	128 (20)	130 (28)	132 (33)	138 (35)
2	90 (10)	114 (14)	120 (17)	128 (25)	133 (33)	136 (38)	136 (45)	140 (48)	141 (50)
4	99 (10)	116 (13)	129 (21)	133 (29)	137 (37)	141 (42)	142 (52)	143 (55)	144 (56)
8	106 (16)	127 (19)	133 (23)	137 (33)	139 (41)	140 (48)	146 (60)	146 (61)	148 (64)
16	92 (10)	123 (16)	133 (23)	137 (31)	142 (44)	145 (54)	148 (61)	149 (63)	150 (65)
32	117 (15)	130 (17)	137 (27)	144 (34)	144 (45)	145 (51)	149 (61)	150 (65)	150 (68)
64	112 (16)	132 (20)	138 (28)	143 (38)	144 (44)	148 (58)	149 (67)	149 (68)	150 (73)
128	117 (14)	133 (18)	138 (23)	143 (30)	145 (34)	147 (46)	149 (58)	150 (66)	151 (73)
256	124 (15)	134 (18)	142 (22)	145 (22)	148 (30)	150 (39)	151 (48)	151 (58)	151 (70)
512	126 (13)	135 (15)	143 (19)	145 (24)	148 (32)	151 (41)	151 (44)	151 (51)	151 (58)

Table 2
Number of problems for which Objective FP finds a feasible (best-known value attained) solution (total = 162).

	10t	20t	40t	80t	160t	320t	640t	1280t	2560t
1	37 (2)	66 (6)	89 (10)	105 (12)	112 (15)	116 (18)	120 (23)	123 (26)	126 (30)
2	52 (7)	83 (11)	103 (17)	117 (24)	124 (29)	128 (36)	131 (40)	134 (48)	136 (51)
4	72 (12)	98 (20)	119 (26)	129 (32)	136 (45)	138 (50)	139 (53)	141 (56)	143 (59)
8	74 (17)	103 (26)	122 (33)	131 (40)	137 (45)	138 (53)	140 (55)	142 (61)	145 (62)
16	77 (18)	109 (25)	125 (35)	134 (44)	139 (50)	142 (57)	144 (60)	146 (65)	147 (69)
32	78 (19)	108 (27)	124 (34)	134 (46)	140 (54)	143 (60)	145 (63)	148 (67)	148 (69)
64	77 (19)	110 (31)	128 (41)	138 (49)	144 (60)	146 (65)	146 (66)	149 (70)	149 (70)
128	81 (18)	111 (29)	129 (41)	139 (45)	145 (56)	147 (67)	147 (70)	149 (73)	151 (76)
256	84 (19)	112 (31)	129 (38)	139 (47)	144 (52)	147 (63)	147 (68)	149 (73)	150 (73)
512	85 (17)	112 (29)	132 (43)	142 (49)	145 (53)	146 (56)	147 (68)	150 (76)	150 (79)

4.3. Computational environment and test bed

All the algorithms were coded in C++. Computations ware performed on the Northwestern University HPC system referred to as QUEST. The computations in this study were carried out in the Westmere cluster of QUEST (252 Intel Westmere X5650 (2.66 GHz) nodes (3052 cores) with 4GBs of memory per core). To access the resources in a reasonable time, we allow node sharing in all experiments. We used openMPI 1.8.1 to employ the master/slave paradigm. Cplex 12.5 with a coin-OR interface is used for solving the linear programming relaxations at the slaves.

We used 74 problems from the COR@L library [8], 28 problems from the MIPLIB 2003 library [2], and 84 feasible problems from the MIPLIB 2010 benchmark set [16] for our test set. As some of the problems are duplicate in the test sets, the total number of problems in our test bed is 180. The details on the test problems are provided in the Appendix.

5. Computational results

In order to assess the value of parallelization irrespective of the increase in the CPU-time, we run the algorithm in an increasingly parallel environment using 1, 2, 4, 8, 16, 32, 64, 128, 256, and 512 parallel subroutines. The amount of time one wants to spend on heuristics to generate feasible solutions depends on the user/solver settings. In our runs the time limit for each problem is calculated depending on the solution time of the first continuous relaxation. We calculated the time to solve the first relaxation at different times of the day and different days of the week. This is done to get an understanding of how the load of the HPC system effects the results even for a single LP relaxation. Even when the underlying hardware is homogeneous, sharing the resources with multiple users makes the system heterogeneous. For example, the coefficients of variation of the solution times for problems varied between 0.008 and 0.340. We averaged the wall clock time to use a representative LP solution time (referred to as t) to establish a base value. The time limit of all the algorithms are multiples of this value. A significant portion of the problems (162 out of 180) have $t < 6$ seconds. We run these problems for up to 2560t wall clock time limit. From the remaining 18 problems, 13 are run with 256t time limit. The limits are selected in such a way that the maximum time to run each problem is limited to four hours. The remaining five problems are not included in the analysis, as 256t is more than four hours. This was needed to ensure that we get the resources on QUEST in a timely manner. We recorded the results at 10t, 20t, 40t, ..., 2560t for $t < 6$, and 1t, 2t, 4t, ..., 256t for $t \geq 6$. The idea is to understand the trade off between the wall clock time limit and the number of processors.

5.1. Analysis with respect to number of instances for which a solution is found

We tested both basic and objective FP algorithms in our analysis. Tables 1 and 2 summarize the results for 10 parallelization levels (1, 2, 4, ...,512) and nine time levels (10t, 20t, 40t, ..., 2560t). Each cell represents the number of problems for which the algorithm finds a feasible solution at the given parallelization level and time limit. The numbers in parenthesis represent the number of problems for which a best-known value is attained. Note that if an algorithm finds a solution that is better than, or equal to, the best known solution (reported at library web pages), it is considered as best-known value attained in this analysis. There are cases for which we find solutions that are better than the best known values reported.

Observe from Table 1 that as the time limit increases (i.e., moving right at each row) the number of problems for which a feasible (best-known value attained) solution is found increases. We observe that in most of the cases, increasing the parallelization level (i.e., moving down at each column) provides at least the same results, or better. However, there are cases in which using the same amount of time with more processors result in finding lower quality solutions. This may be due to (1) increasing the parallelization level increases the time to map the processes to different processors, (2) the runs for different parallelization levels are taken at different times and environments, and (3) inherent randomness in parallel implementations. All reasons are due to the computational and parallel environment.

The total amount of resources used by an algorithm can be calculated by multiplying the time spent and number of processors. Each cell uses the same amount of resources with its up-right

Table 3
Geometric average gap for basic FP on problems with $t < 6$ anchored at $80t$, one processor.

	$10t$	$20t$	$40t$	$80t$	$160t$	$320t$	$640t$	$1280t$	$2560t$
1	74%	69%	66%	63%	52%	43%	38%	32%	27%
2	43%	44%	36%	29%	21%	20%	16%	14%	12%
4	43%	41%	28%	21%	18%	15%	13%	12%	11%
8	43%	34%	25%	19%	14%	12%	11%	11%	10%
16	37%	34%	28%	19%	14%	11%	10%	9%	9%
32	45%	32%	26%	19%	14%	10%	9%	9%	8%
64	35%	30%	24%	19%	14%	10%	8%	8%	8%
128	37%	33%	28%	24%	20%	14%	10%	8%	7%
256	43%	39%	35%	31%	25%	22%	17%	13%	8%
512	42%	38%	34%	31%	27%	23%	20%	15%	11%

Table 4
Geometric average gap for objective FP on problems with $t < 6$ anchored at $80t$, one processor.

	$10t$	$20t$	$40t$	$80t$	$160t$	$320t$	$640t$	$1280t$	$2560t$
1	64%	43%	45%	48%	39%	35%	32%	30%	27%
2	24%	29%	36%	24%	20%	17%	15%	14%	12%
4	16%	21%	19%	15%	13%	12%	12%	11%	9%
8	15%	17%	16%	13%	12%	11%	10%	8%	7%
16	17%	21%	14%	9%	9%	8%	7%	6%	6%
32	16%	19%	14%	9%	8%	8%	6%	6%	6%
64	14%	15%	12%	9%	8%	7%	6%	6%	5%
128	17%	18%	14%	11%	9%	8%	6%	5%	5%
256	20%	20%	14%	12%	11%	10%	8%	8%	5%
512	18%	18%	14%	13%	11%	9%	8%	7%	6%

and down-left cell. Moving in the down-left direction in the table represents the use of the same resources with more processors and less time. We observe that the numbers in each cell in columns with more that $80t$ is comparable with its up-right and down-left cell. For a general conclusion, we observe that the first row represents a single processor (a classical serial algorithm). We also note that even if it is given a long time (i.e., $2560t$) a serial basic FP algorithm can find a solution for 138 problems, 35 of which attain the best-known value. When 32 processors are used with $80t$ time limit (32 times less), the run finds a solution for 144 problems (34 being best-known value attained). The amount of resources used by both implementations is the same (32 processors $\times$ $80t$ = 1 processor $\times$ $2560t$). The number of problems for which feasible solutions are found increases with decreased time and increased processors. In order to reach a better understanding, we ignore the problems for which a serial algorithm can find a solution in $80t$ time limit. These problems are likely not to benefit from parallelization. We refer to this situation as anchoring at a single processor, $80t$. Considering both the number of problems for which a feasible (and best-known value attained) solution is found through several parallelization levels, we conclude that the parallel version of the basic FP algorithm linearly scales, for time values greater than $80t$ and parallelization level less than 512.

The results for objective FP are provided in Table 2. Comparing the results with Table 1 indicates that the basic FP algorithm finds feasible solutions for more problems in almost all parallelization and time levels. This is due to the fact that the objective FP algorithm searches for higher quality solutions in terms of objective function value and the basic FP focuses only on feasibility. In terms of the number of problems for which each algorithm finds a best-known value attained solution, the objective FP seems to provide better results. However, the comparison cannot be generalized among parallelization and time levels. Similar to the results for the basic FP, moving in the down-left direction in Table 2 provides better or equivalent results for time values greater than $80t$ and parallelization level up to 512. On examining the table, we conclude that in terms of the number of problems for which a feasible (best-known value attained) solution is found, objective FP scales linearly, for time values greater than $80t$ and parallelization level less than 512.

Figures 3 and 4 in the Appendix show that the ratio of the slope in the *parallelization level* to *time increase* ratio is greater than one till 64 processors, suggesting that the value of parallelization is better than that of increasing time till these many processors.

On observing Tables 1 and 2 one can see that the number of instances for which a feasible solution is found does not linearly scale with the increase in the number of processors. The biggest jump is from one to two processors, and the benefits diminish as the number of processors is increased. We offer the following explanation. Given a time limit, let the probability of finding a solution using one of the subroutines be p. If n subroutines are run in parallel, the probability that at least one of them finds a solution is $1-(1-p)^n$. The change in the probability by doubling the number of processors to $2n$ is $(1-(1-p)^{2n})-(1-(1-p)^n) = (1-p)^n-(1-p)^{2n}$, which decreases with n (for $0 < p < 1$). There is little value in parallelizing with p closer to 1. Also, this expression suggests that the benefits of parallelization are larger for a smaller value of n ($n = 1$ giving the largest benefit), and when p is closer to 0, we need a large n. Unfortunately, for large n, communication overhead increases.

The results for the 13 larger problems, which are run for up to a $256t$ time limit, show a similar trend as that for the small problems.

5.2. Analysis with respect to quality of the solutions

In this section we share the results of an analysis starting with the average performance of both basic and objective FP for all time and parallelization levels. For a proper comparison, we provide results for easy problems anchored at $80t$ and single processor for basic FP (105 problems are provided in cell $80t$ and one processor of Table 1), however, we excluded four problems for which the anchored method finds a solution which is very poor (gap $>$ 10 000%), resulting in 101 problems.

Tables 3 and 4 provide average gap values of both basic and objective FP, for all parallelization and time $\geq$ $80t$ levels, for the aforementioned 101 problems. The value in each cell is calculated by the following shifted geometric mean formula: $\sqrt[101]{(g_1+1)\cdot(g_2+1)\cdots(g_{101}+1)}-1$, where g_i represents the optimality gap for problem i.

We observe from Tables 3 and 4 that increasing time always improves performance of both algorithms for all parallelization levels (moving right at each row). We also observe that, the performance of both algorithms first increases and then decreases with increasing level of parallelization (moving down at each column), especially for time $\geq$ $80t$. For each column, we underline the cells for which increasing the parallelization level decreases the performance of the algorithms for the first time. As time limit and parallelization level increases, the benefits of parallelization increases up to 64 processors for basic FP and up to 128 processors for objective FP, diminishing thereafter.

Figures 1 and 2 in the Appendix give information on the statistical significance of knowing if using up to 64 (128) processors in the basic (objective) FP require less wall clock time while producing the same or better quality solution, when compared with the FP counterpart that is run for a longer wall clock time using less processors while giving an equal amount of total cpu time.

5.3. Value of information sharing

In order to investigate the value of information sharing independent of other factors, we turned off the exchange of primal bounds and let the subroutines run independently. The master returns the best of the solutions provided by all the subroutines. Therefore, the performance difference of this version

Table 5
Communication effect: average decrease in optimality gap for basic FP with information sharing.

	10t	20t	40t	80t	160t	320t	640t	1280t	2560t
2	14%	21%	22%	16%	16%	15%	12%	6%	3%
4	14%	23%	24%	25%	22%	21%	18%	11%	6%
8	18%	25%	26%	27%	27%	26%	22%	16%	11%
16	14%	20%	27%	27%	27%	27%	23%	18%	12%
32	8%	17%	26%	27%	27%	26%	22%	18%	11%
64	10%	20%	26%	30%	29%	28%	24%	20%	14%
128	9%	17%	18%	18%	17%	18%	23%	20%	15%
256	5%	9%	9%	7%	7%	6%	14%	13%	12%

Table 6
Geometric average gap for serial and sequential versions of Basic FP.

#proc	Serial	Sequential	Difference
2	36%	36%	0%
4	28%	27%	1%
8	37%	33%	4%
16	33%	31%	2%
32	29%	28%	1%
64	27%	25%	2%
128	21%	21%	0%
256	19%	20%	−1%

with the proposed algorithm is only due to the information sharing/communication. Table 5 shows the average improvement of the basic FP algorithm in the presence of communication. For the results in this table, we only included problems where both algorithms find a feasible solution. Each cell in Table 5 shows the average decrease in the optimality gap.

Table 5 indicates that the value of information sharing first increases and subsequently decreases with increasing parallelization and time levels. We observe that the effect of information sharing is statistically significant at all levels of parallelization and time with p-value $\leq$ 0.1. We conjecture that it is because in parallel FP, the marginal value of increasing time and parallelization levels reduces as the shared information becomes less and less useful, with increasing parallelization levels. The results for the objective FP algorithm show similar trends.

5.4. Value of starting from multiple solutions

One may think that any parallel algorithm can be sequentially solved by running subroutines one after another and using the accumulated information in the later subroutines to quantify the value of starting from multiple solutions and running in parallel. This may be possible for some algorithms with synchronous communication where at predetermined time points, all subroutines synchronize with others. In our algorithm, all subroutines update themselves in an asynchronous fashion whenever the first solution is found by any of the subroutines. Since it is not known which subroutine will result in an integer feasible solution in advance, one cannot implement a sequential version of our algorithm to evaluate the effects of a better exploitation of starting points. Although our algorithm cannot be imitated by a sequential algorithm, we ran multiple short basic FPs starting from different rounded solutions, one after another and compared the results with the serial version. Both algorithms are run for 2560t time units saving the best solution found at 10t, 20t, 40t, ..., 2560t. The sequential algorithm is run for 10t for each of the 256 starting solutions. Table 6 shows shifted geometric means of the optimality gap values of the solutions found by both methods and the corresponding difference. The value of better exploitation of starting points increases up to 64 processors and decreases thereafter. We conclude there is value of starting from multiple solutions even in a sequential algorithm.

5.5. Additional experience

As part of the extensive computational experiments we also tried many other possibilities, including a wide range of combinations of FP instances starting from LP optimum with different random number streams, near analytic center, random vertices close to LP optimum, and the line combining the analytic center and LP optimum. Unfortunately, however, none of these paradigms resulted in a superior performance. For this larger set of experiments, approximately 40 years of computational time was spent only on QUEST resources on top of the tests on shared memory devices when drawing the conclusions.

6. Conclusion

It is known that FP is a useful heuristic for MILP, as it typically finds feasible solutions for practical problems in a reasonable computational time [1,7,10]. In all studies related to the use of FP, however, no parallelization is used.

We found that starting FP from multiple rounded points in parallel outperforms using the same starting solution (that is an optimum solution for the continuous relaxation) and running with different random number streams. There is a significant value in starting from multiple rounded points in the presence of parallelization. Extensive computational tests indicate that the value of increasing the level of parallelization is statistically significant for up to 128 cores. While in this paper we have presented our experience with using the feasibility pump heuristic to generate good quality solutions in parallel, exploration of other algorithms within the proposed framework remains a topic of future research.

Acknowledgment

This study is supported by ONR (Grant No.: N000141210051) and DoE (Grant No.: DE-SC0005102). The main part of the study was conducted while Utku Koc was a post doctoral fellow at Northwestern University.

Appendix A. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.orl.2017.10.003.

References

[1] T. Achterberg, T. Berthold, Improving the feasibility pump, Discrete Optim. 4 (1) (2007) 77–86.
[2] T. Achterberg, T. Koch, A. Martin, MIPLIB 2003, Oper. Res. Lett. 34 (4) (2006) 361–372. http://dx.doi.org/10.1016/j.orl.2005.07.009.
[3] D. Baena, J. Castro, Using the analytic center in the feasibility pump, Oper. Res. Lett. 39 (5) (2011) 310–317.
[4] E. Balas, S. Ceria, M. Dawande, F. Margot, G. Pataki, Octane: A new heuristic for pure 0-1 programs, Oper. Res. 49 (2) (2001) 207–225.
[5] E. Balas, C.H. Martin, Pivot-and-complement: A Heuristic for 0-1 Programming, Manage. Sci. 26 (1) (1980) 8696.
[6] E. Balas, S. Schmieta, C. Wallace, Pivot and shift - a mixed integer programming heuristic, Discrete Optim. 1 (1) (2004) 3–12.

[7] L. Bertacco, M. Fischetti, A. Lodi, A feasibility pump heuristic for general mixed-integer problems, Discrete Optim. 4 (1) (2007) 63–76.
[8] COR@L: Computational Optimization Research at Lehigh (http://coral.ie.lehigh.edu/mip-instances/), Industrial and Systems Engineering, Lehigh University.
[9] E. Danna, E. Rothberg, C. Pape, Exploring relaxation induced neighborhoods to improve MIP solutions, Math. Program. 102 (1) (2005) 71–90.
[10] M. Fischetti, F. Glover, A. Lodi, The feasibility pump, Math. Program. 104 (1) (2005) 91–104.
[11] M. Fischetti, A. Lodi, Local branching, Math. Program. 98 (1) (2003) 23–47.
[12] M. Fischetti, A. Lodi, M. Monaci, D. Salvagnin, A. Tramontani, Improving branch-and-cut performance by random sampling, Math. Program. Comput. (2016) 113–132. http://dx.doi.org/10.1007/s12532-015-0096-0.
[13] M. Fischetti, D. Salvagnin, Feasibility pump 2.0, Math. Program. Comput. 1 (2009) 201–222.
[14] K.-L. Huang, S. Mehrotra, An empirical evaluation of walk-and-round heuristics for mixed integer linear programs, Comput. Optim. Appl. 55 (3) (2013) 545–570.
[15] K.-L. Huang, S. Mehrotra, An empirical evaluation of a walk-relax-round heuristic for mixed integer convex programs, Comput. Optim. Appl. 60 (3) (2015) 559–585. http://dx.doi.org/10.1007/s10589-014-9693-5.
[16] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R.E. Bixby, E. Danna, G. Gamrath, A.M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D.E. Steffy, K. Wolter, MIPLIB 2010, Math. Program. Comput. 3 (2) (2011) 103–163. http://dx.doi.org/10.1007/s12532-011-0025-9.
[17] T. Koch, T. Ralphs, Y. Shinano, Could we use a million cores to solve an integer program?, Math. Methods Oper. Res. 76 (1) (2012) 67–93. http://dx.doi.org/10.1007/s00186-012-0390-9.
[18] A. Lodi, Mixed integer programming computation, in: M. Jnger, T.M. Liebling, D. Naddef, G.L. Nemhauser, W.R. Pulleyblank, G. Reinelt, G. Rinaldi, L.A. Wolsey (Eds.), 50 Years of Integer Programming 1958–2008, Springer Berlin Heidelberg, 2010, pp. 619–645.
[19] L.-M. Munguia, S. Ahmed, D.A. Bader, G.L. Nemhauser, Y. Shao, Alternating criteria Search : A parallel large neighborhood search algorithm for mixed integer programs, Opt. Online (2016) 1–20.