Generation of Feasible Integer Solutions on a Massively Parallel Computer☆,☆☆

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

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

Abstract

We present an approach to parallelize generation of feasible solutions of mixed integer linear programs in distributed memory high performance computing environments. The approach combines a parallel framework with feasibility pump (FP) as the rounding heuristic. The proposed approach runs multiple FP instances with different starting solutions concurrently, while allowing them to share information. The starting solutions for multiple subroutines are created by rounding the most fractional k variables of an optimal solution of the continuous relaxation. Our computational results on COR@L, MIPLIB 2003, and MIPLIB 2010 test sets suggest that the improvement resulting from parallelization using our approach is statistically significant. Furthermore, running multiple short FP algorithms in parallel can significantly outperform running a single long version even if both algorithms are given the same amount of CPU time. This suggest that the benefits of parallelization are also due to information sharing.

Keywords: Mixed Integer Programming, Parallel Optimization, Feasibility Pump

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. MILP is extensively studied in the literature. We suggest interested reader to [1] for a recent review. Generating high quality feasible solutions quickly is important in practice. This is because 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. In this study, 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 environments. 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. From a practical point of view,

☆This study is supported by ONR (grant no:) and DoE (grant no:)
☆☆The main part of the study was conducted while Utku Koc was a post doctoral fellow at Northwestern University.

*Corresponding author
Email addresses: utku.koc@mef.edu.tr (Utku Koc), mehrotra@northwestern.edu (Sanjay Mehrotra)

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.* [2] 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 communication overhead, idle time for initial tasks or termination (ramp-up and ramp-down), performance effect of the redundant work (some nodes may not have been evaluated if fewer processors are used), and idle time due to latency/contention/starvation.

The FP algorithm was first proposed by Fischetti *et al.* [3]. An extension to general MILPs is proposed by Bertacco *et al.* [4]. By a modification of the objective function, Achterberg and Berthold [5] found better feasible solutions (Objective FP). Fischetti and Salvagnin proposed different rounding heuristic by using constraint propagation techniques after rounding some of the variables [6]. Baena and Castro [7] 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. In this study, we provide a parallel framework in which multiple feasibility heuristics starting from different solutions can communicate and share information. Recently, 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 [8] and Mixed Integer Convex Programs (MICPs) [9].

This paper has multiple contributions to the literature: 1) we assess the value of parallelization independent of the increase in the CPU-time, 2) we provide a parallel framework that can use multiple parameters for FP type 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 using enumeration in a distributed memory environment with many cores. Our computational experiments suggest that, running multiple algorithms for a short amount of time in parallel can significantly outperform running a single long version even if both algorithms are given the same amount of CPU clock time. Thus, the benefits of parallelization are not only due to the increase in the CPU-time (given the same amount of wall clock time) but also due to multiple algorithms running in parallel and sharing information along the course of the algorithms.

We present computational results describing our experience with the use of the FP heuristic in the parallel subroutines. The original FP algorithm starts from the rounded solution of an optimal solution of the continuous relaxation. In this study, all possible rounded points from the most fractional k variables are enumerated and 2^k subroutines are run in parallel.

The rest of the paper is organized as follows: we describe our parallel heuristic framework for the use of multiple heuristics 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 and our experience regarding the use of massively parallel systems are discussed in Section 5. Finally, we conclude in Section 6.

2. A Concurrent Framework for Finding Feasible Solutions for MILPs

In this section we describe our concurrent framework to generate feasible solutions for MILPs. In our approach, we run multiple feasibility heuristics in parallel. We refer to the algorithms running in different processors as the subroutines. Each parallel subroutine uses different random number seed with different starting solutions. Also, one may run different feasibility heuristics in parallel. 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 (due to the inherent randomness) in the course of the parallel subroutines. Whenever one of the subroutines finds a feasible solution, it broadcasts the objective function value to others. 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. All parallel instances restart their search (with the new collective information) at the time a better solution is found. In other words, all subroutines continue as if they found a better solution which is fed by the others. In this study, as a proof of concept, we use FP as the rounding procedure at the subroutines of our concurrent feasibility heuristic.

Regarding the communication during the run time, one may use so called master/slave topology. In this paradigm, 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 master includes distributing inputs to and collecting results from the slaves. When one of the slaves finds an integer solution, it sends the solution to the master, along with the objective function value. Moreover, any combination of parameter settings, rounding methods, and anti cycling rules are also valid. The main algorithm that runs at the master is presented in Algorithm 2.1.

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 $LB = \min\{c^T x : Ax \geq b, x \in \mathbb{R}^n\}$, $UB = \infty$ and $RHS = UB - \epsilon$
3: **while** termination criteria not met **do**
4: Inform slaves about new RHS
5: Collect results
6: **if** One of the slaves return an integer solution **then**
7: Update UB = minimum of the slaves
8: Update $RHS = UB - \epsilon$
9: **end if**
10: **end while**
11: Exit all the slaves and return best integer so far

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, slave subroutine receives some information from the master (if any). Then updates itself with the new information, creates a starting solution for the algorithms depending on the type of heuristic it is running. The heuristic subroutine continues until predetermined criteria is met or master provides new information. Whenever an integer solution is identified, it is shared with the rest of the concurrent subroutines by means of the master.

Algorithm 2.2 *Parallel Heuristic Subroutine Running in Slaves*

Input: a MILP, RHS
Output: an integer solution to the MILP

```
1: Listen master for the type of heuristic that will be run
2: while not killed by the master do
3:     Listen master for parameters and information (RHS)
4:     Update RHS of the objective cutoff constraint
5:     Get information form master (LP optimum (x*_lp))
6:     Update with respect to the heuristic variant
7:     Create a starting solution x
8:     Run heuristic starting from x
9:     Broadcast best integer solution
10: end while
```

The variants of the heuristic subroutines differ in Steps 6-8 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. Next, we define the variants of heuristic subroutines and the feasibility pump algorithm running in slaves in detail.

3. Variants of FP Heuristic

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

3.1. Basic and Objective FP Algorithms

FP heuristic was first proposed by Fischetti *et al.* [3] for 0-1 MILPs. 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:

$$\min \qquad \Delta(x,\tilde{x}) = \sum_{j\in I} |x_j - \tilde{x}_j| \tag{1}$$

$$Ax \geq b \tag{2}$$

$$c^T x \leq RHS \tag{3}$$

$$x \in \mathbb{R}^n, x_j \in \mathbb{Z}, \quad \forall j \in I, \tag{4}$$

where (1) is the l_1 norm distance, (2) and (4) are the constraint set defined by the original MILP and (3) is the objective cut off constraint.

Two decisions are made in this heuristic: starting solutions and rounding procedure. Moreover, one needs to define an iterative version that moves from one starting solution to the next. In other words, one meeds to define how x and $\tilde{x}$ are calculated at each iteration. Note that the above model focuses on feasibility with no consideration on the quality of the solution.

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) [5]. 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 (1) of the above MILP (1) is replaced by

$$\frac{1-\alpha}{||\Delta||}\Delta(x, \tilde{x}^{k-1}) + \frac{\alpha}{||c||}c^T x, \quad (5)$$

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 (5) 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 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 cutoff 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. Objective cutoff constraint is used to find solutions with improved objective. If objective of the problem is known to be 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 original FP algorithm continues from this solution and rounds it to another integer solution. In other words, the next iteration starts from an optimal solution of the l_1 norm minimization problem with the same rounding scheme. The algorithm termites if an optimal solution for MILP is found or time/iteration limit is reached. 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 $k = 0$, $LB = \min\{c^T x : Ax \geq b\}, UB := \infty, RHS = UB - \epsilon$
2: Set $x^k := \arg\min\{c^T x : Ax \geq b, c^T x \leq RHS\}$.
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:    $\alpha = \alpha \times \alpha_r$
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 $RHS := UB - \epsilon$ and go to step 2.
14:    **end if**
15:    **if** $\exists j \in I : [x_j^k] \neq \tilde{x}_j^k$ **then**
16:       set $\tilde{x} := [x^k]$
17:    **else**
18:       flip entries $\tilde{x}_j^k$ $(j \in I)$ randomly
19:    **end if**
20: **end while**

Note that the above algorithm may cycle. In the original implementation by Fischetti *et al.* [3], whenever a cycle is heuristically detected, a random perturbation is applied by skipping Step 15 and directly moving to Step 18. In Step 18 of the algorithm, flipping an entry means changing the rounding value of the entry. If $\tilde{x}_j^k < x_j^k$ and $\tilde{x}_j^k$ is to be flipped, we increase $\tilde{x}_j^k$ by one. Similarly, if $\tilde{x}_j^k > x_j^k$, flipping corresponds to decreasing the value of $\tilde{x}_j^k$ by one. In the cycle breaking perturbation, a random number of indexes 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 (Message Passing Interface) to ensure scalability. The communication between the slaves is done by the master. We use Mersenne twister random number generator at each slave. In order to ensure that each slave uses a different random number stream, the seed for each generator is fed by the master. The seeds are generated using a linear congruential method. Recall that whenever a feasible integer solution is found, objective cut off constraint needs to be updated at all parallel processors. In our implementation each slave updates itself then sends the objective function value to the master process, which in turn broadcasts the best of the slave objectives to the other processors. The master process checks the slaves for feasible solutions in a predetermined sequence. The broadcast of the objective function value is done in the same sequence. Due the communication lag, a slave may have already found a solution that is better than the broadcasted one. In this case, slaves do not update the objective cutoff constraint.

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. Wall clock time is imposed to ensure that all parallel subroutines complete at the same time. In a parallel setting where multiple variants are used, there may be drastic differences in the completion time of a fixed number of iterations and/or CPU-time. To get 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.

4.2. Implementation of the Rounding Heuristics

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

Stage 1 - Start point generation: The main difference within FP implementations is based on this stage, The starting solution for the original FP algorithm is an optimal solution for the continuous relaxation. One than rounds this solution. In our implementation, each parallel subroutine changes the rounding scheme in the following way: 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: FP algorithm is implemented as the rounding stage. The details are similar to the original FP implementation by Fischetti *et al.*. However, we turned off the branching phase. As the improvement phase is handled by the master algorithm, internal improvement is also disabled. All other parameters are used at the default values. Note that if FP implementation hits its internal iteration limits, the algorithm resets those as long as the limits imposed by the master is not met. 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.

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. After each rounding iteration, slaves check if there exists a new incumbent solution and update objective cut off constraint, if necessary.

4.3. Computational Environment and Test Bed

All the algorithms are coded in C++. Computations are performed on Northwestern University high performance computing (HPC) system referred to as QUEST. At the time this study is conducted QUEST clusters had 252 Intel Westmere X5650 (2.66 GHz) nodes (3052 cores), 68 Intel Sandybridge E2670 (2.6 GHz) nodes (1088 cores) and 110 Intel IvyBridge E5-2680 (2.8 GHz) nodes (2200 cores). This computations in this study is carried out in the Westmere cluster. Each node has at least 4GBs of memory per core for all the nodes. In practice, as the number of cores needed increase, it is reasonable to share the nodes with other users in HPC systems. To access the resources in a reasonable time, we allow to share the nodes with other users in all experiments. We point out that, depending on internal and external factors, controlling the process of the parallel implementations is an issue in a shared machine. The mapping of the nodes and cores may take some time depending on system settings. We used MPI 3 standards 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.

For our tests, we used 74 problems from the COR@L library [10], 28 problems from the MIPLIB 2003 library [11], and 84 feasible problems from the MIPLIB 2010 benchmark set [12]. 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 generated feasible solutions depend 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. In a set of preliminary experiments, 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. We then averaged the solution times (referred to as t). 90% of the problems (162 out of 180) have $t < 6$ seconds. We run these problems for up to $2560t$ wall clock time limit. 13 of the remaining 18 problems are run with $256t$ time limit. The limits are selected in such a way that 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 that 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. We tested both basic and objective FP algorithms in our analysis. Table 1 and 2 summarizes the results for 10 parallelization levels (1, 2, 4, . . .,512) and nine time levels ($10t$, $20t$, $40t$, . . ., $2560t$). Each cell represent the number of problems for which the algorithm finds a feasible solution at a given parallelization level and time limit. The numbers in parenthesis represent the number of problems for which an optimal solution is found. 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 optimal in this analysis. There are cases for which we find solutions that are better than the best known values reported in the library web pages, though they seem to be outdated. We start the analysis with basic FP algorithm.

Observe from Table 1 that as time increases (i.e., moving right at each row) the number of problems for which a feasible (optimal) 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 if not better. However, there are cases in which using the same amount of time with more processors result in finding less solutions. This may be due to two reasons: 1) increasing the parallelization level increases the time to map the processes to different processors, and 2) the runs for different parallelization levels are taken at different times and environments. Both reasons are due to the computational and parallel environment. We provide a detailed discussion on our experience on parallel HPC systems in Section 5.1.

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 and down-left cell. Moving in the down-left direction in the table represents the use of same resources with more processors and less time. Note 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 observe that the first row represents a single processor (a classical serial algorithm). 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 being optimal. When 32 processors are used with $80t$ time limit (32 times less), the run finds a solution for 144 problems (34 being optimal). The amount of resources used by both implementations is the same (32 processors × $80t$ = 1 processor × $2560t$). The number of problems for which feasible solutions are found increases with decreased time and increased processors. In order for a clearer understanding, we ignore the problems for which a serial algorithm can find a solution in $80t$ time limit. These problems are likely no 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 optimal solution is found through several parallelization levels, we conclude that 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 ba-

Table 1: Number of problems for which basic FP finds a feasible(optimal) 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(optimal) 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)

sic FP algorithm finds feasible solutions for more problems in almost all parallelization and time levels. This is due to the fact that objective FP algorithm searches for higher quality solutions in terms of objective function value and basic FP focuses only on feasibility. In terms of the number of problems for which each algorithm finds an optimal solution, objective FP seems to provide better results. However, the comparison cannot be generalized among parallelization and time levels. Similar to the results for 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. Consider the results anchored at single processor, $80t$. The number of problems for which a feasible(optimal) solution found is 105 (12). Increasing the time limit 32 fold increases this number to 126 (30). However, using the same amount of resources but in parallel with 32 cores, the numbers increase to 134 (46). On examining the table, we conclude that in terms of the number of problems for which a feasible(optimal) solution is found, objective FP scales linearly, for time values greater than $80t$ and parallelization level less than 512. Figures 1 and 2 show how the number of problems for which basic and objective FP finds a feasible solution changes with respect to different time and parallelization levels. For a clearer understanding, we included the time values starting from $40t$. Observe from the figures that, the slope in the parallelization level direction is more than the time increase direction. This is due to the fact that the effects of parallelization is more than that of time. Also, the effects of parallelization is more in the lower levels.

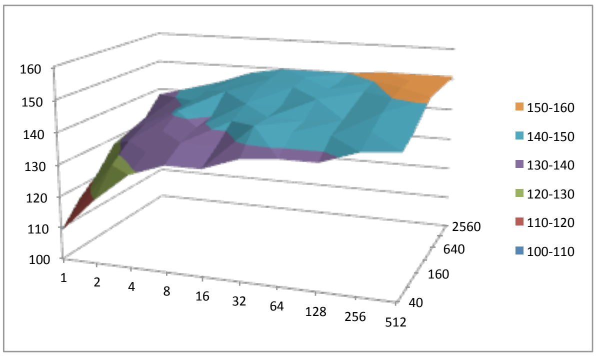

Figure 1: Number found with respect to time and parallelization level for Basic FP

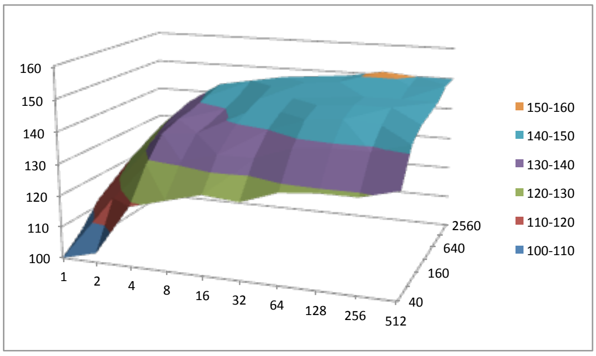

Figure 2: Number found with respect to time and parallelization level for Objective FP

Continuing with the 13 larger problems which are run for up to 256t time limit, the results in Tables 3 and 4 provide the number of problems for which a feasible (optimal) solution is found for basic and objective FP, respectively. The results show a similar trend as that for the small problems.

Considering the quality of the solutions generated at each time limit and parallelization level, we perform a pairwise comparison of all methods with equal resources using Wilcoxon signed rank test on percentage gap values. Table 5 and 6 present the significance of the difference between multiple parallelization and time levels for basic FP and objective FP algorithms, respectively. The columns of the table corresponds to time difference and rows correspond to processor difference. For example, the value at row 2-4 and column 80t-40t of Table 5 ($\alpha = -0.95$) represents that the performance difference between two processors and 80t and four processors with 40t is significant with $\alpha = 0.95$. The positive sign represents that the algorithm with more processors (and less time) is better, whereas a negative value states that the longer algorithm (and more time) provides statistically better solutions. Empty cells represent that the performance difference is not significant. Recall that both algorithms have the same resources thus, the performance difference is due to parallelization irrespective of the increase in the CPU-time. Note that empty cells and positive values are in favor of parallelization. Observe that using two cores instead of one is statistically significant at all time limits for both basic and objective FP. The significance of parallelization increases with time up to 64 cores for basic FP and up to 128 cores for objective FP. The value of parallelization of FP in the way described here decreases as the level of parallelization increases, diminishing after 128 processors. Also, the value of parallelization increase with time.

5.1. Additional Computational Experience

In this section, we share additional computational experience on running basic and objective FP in a distributed memory environment. Computers have become commodities rather than technological equipments and massive parallelization in a distributed memory setting is the current trend. However, even the simplest applications encounter serious implementation and practical issues. As the number of processors increase, it is more difficult to have a dedicated set of nodes that can only be accessed by a single user. Thus, one needs to accept sharing the computing resources with other users that may run various types of programs with different requirements. Some applications focus more on memory while others rely on CPU usage. Professional programs focus more on optimizing performance in a lower coding level and may squeeze the use of memory and CPU. Sharing a node with such applications may result in underutilization on one side resulting in unfair distribution of the resources. Depending on the underlying system and message passing structure, the time to allocate nodes, mapping of the processors varies. This also depends on the programs running in all nodes.

The iteration count of the first feasible solution would be the same across runs for the same slave if there were no interruption from other slaves (no communication). However, the course of our parallel algorithm depends on *the time* when the slaves find a solution (especially the first solution) rather than the *iteration count* of the algorithms running in slaves. The slaves, on the

Table 3: Number of problems for which basic FP finds a feasible(optimal) solution for large problems (total =13)

	1t	2t	4t	8t	16t	32t	64t	128t	256t
1	1(0)	3(0)	4(0)	6(0)	8(0)	8(0)	9(0)	9(0)	9(0)
2	2(0)	4(0)	7(0)	7(0)	9(1)	11(1)	11(1)	12(1)	12(1)
4	4(0)	4(0)	7(0)	8(0)	10(0)	11(1)	12(1)	12(1)	12(1)
8	3(0)	4(0)	7(0)	10(0)	11(0)	12(0)	12(1)	12(1)	12(1)
16	3(0)	4(0)	8(0)	12(0)	12(1)	13(1)	13(1)	13(1)	13(1)
32	4(0)	4(0)	8(0)	12(1)	13(1)	13(1)	13(1)	13(1)	13(1)
64	4(0)	4(0)	11(0)	12(0)	12(1)	13(1)	13(1)	13(1)	13(1)
128	4(0)	4(0)	11(0)	12(0)	13(1)	13(1)	13(1)	13(1)	13(2)
256	4(0)	4(0)	11(0)	12(0)	13(1)	13(1)	13(2)	13(3)	13(3)
512	4(0)	4(0)	11(0)	12(0)	13(1)	13(1)	13(2)	13(3)	13(3)

Table 4: Number of problems for which Objective FP finds a feasible(optimal) solution for large problems (total = 13)

	1t	2t	4t	8t	16t	32t	64t	128t	256t
1	0(0)	2(0)	2(0)	4(0)	6(0)	7(0)	9(0)	10(0)	10(0)
2	0(0)	2(0)	4(0)	5(0)	9(0)	9(0)	12(1)	12(1)	12(1)
4	0(0)	2(0)	4(0)	5(0)	8(2)	10(2)	12(2)	12(2)	12(2)
8	1(0)	2(0)	5(0)	6(0)	10(2)	11(2)	12(2)	12(2)	12(2)
16	1(0)	3(0)	7(0)	7(0)	11(2)	12(2)	12(3)	12(3)	12(3)
32	1(0)	3(0)	8(0)	8(0)	12(2)	13(2)	13(2)	13(3)	13(3)
64	1(0)	6(0)	8(0)	9(0)	12(2)	13(3)	13(3)	13(3)	13(3)
128	1(0)	6(0)	8(0)	9(0)	12(2)	13(2)	13(3)	13(4)	13(4)
256	1(0)	6(0)	8(1)	9(1)	12(3)	13(4)	13(4)	13(4)	13(4)
512	1(0)	6(0)	8(0)	9(1)	12(3)	13(4)	13(4)	13(4)	13(5)

Table 5: Significance of parallelization for basic FP on problems with $t < 6$

	20t-10t	40t-20t	80t-40t	160t-80t	320t-160t	640t-320t	1280t-640t	2560t-1280t
1-2	0.95	0.995	0.999	0.999	0.999	0.999	0.999	0.999
2-4	-0.995	-0.999	-0.95		0.95	0.999	0.999	0.999
4-8	0.999	-0.95	-0.95		0.95	0.999	0.999	0.999
8-16	-0.999	-0.999	-0.999	-0.999	-0.999			
16-32	-0.999	-0.999	-0.999	-0.999	-0.99			
32-64	-0.999	-0.999	-0.999	-0.999	-0.999			
64-128	-0.999	-0.999	-0.999	-0.999	-0.999	-0.999	-0.999	-0.95
128-256	-0.999	-0.999	-0.999	-0.999	-0.999	-0.999	-0.999	-0.999
256-512	-0.995	-0.9	-0.95	-0.995	-0.999	-0.999	-0.999	-0.999

Table 6: Significance of parallelization for objective FP on problems with $t < 6$

	$20t$-$10t$	$40t$-$20t$	$80t$-$40t$	$160t$-$80t$	$320t$-$160t$	$640t$-$320t$	$1280t$-$640t$	$2560t$-$1280t$
1-2	0.9	0.995	0.999	0.999	0.999	0.999	0.999	0.999
2-4	-0.9	0.95	0.99	0.999	0.999	0.999	0.999	0.999
4-8	-0.999	-0.99	-0.95			0.95	0.95	0.995
8-16	-0.999	-0.999	-0.995				0.9	0.9
16-32	-0.999	-0.999	-0.995	-0.9				
32-64	-0.999	-0.999	-0.999	-0.995				
64-128	-0.999	-0.999	-0.999	-0.999	-0.999	-0.95		
128-256	-0.999	-0.99	-0.999	-0.999	-0.999	-0.999	-0.95	-0.9
256-512	-0.999	-0.999	-0.999	-0.999	-0.999	-0.999	-0.995	-0.99

other hand, are affected by the computing resources used across different runs depending of the programs running in different nodes. Although the slaves follow the same track of iterations, the time to generate a particular solution varies across replications. Thus, a specific set of iterations may be interrupted by a solution fed by other slaves. This suggest that the results depend on the state of the computing resources, and may not replicate as this state cannot be reproduced if the resources are shared with other users. We would like to also note that due to the synchronization in parallel implementations, depending on how the processors are given priority, multiple runs may terminate with slightly different solution even in a shared memory setting.

6. Conclusion

It is already shown that FP is a useful heuristic for MILP as it usually finds feasible solutions for practical problems in a reasonable computational time [5] [4], [3]. In all studies related to the use of FP, however, no parallelization is used. In this study, we tested FP further in a highly scalable parallel framework.

We note 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 of starting from multiple rounded points in the presence of parallelization. Extensive computational test indicate that the value of increasing the level of parallelization is statistically significant for up to 128 cores.

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 [13] performs simplex like pivots to get slack variable into the basis and integer variables out of a basis. This is further extended by Balas [14]. Another heuristic for 0-1 MILP is `OCTANE`, which uses enumeration techniques on extended facets of the octahedron [15]. Fischetti and Lodi propose a local search algorithm [16] to improve an incumbent solution. A heuristic called Relaxation Induced Neighborhood Search `RINS` solves sufficiently smaller sub-MILPs to improve an incumbent solution [17]. The use of random-walks was investigated in the FP setting [8, 9]. Using these heuristics in a parallel framework are considered as future research topics.

[1] 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.

[2] T. Koch, T. Ralphs, Y. Shinano, Could we use a million cores to solve an integer program?, Mathematical Methods of Operations Research 76 (1) (2012) 67–93. doi:10.1007/s00186-012-0390-9. URL http://dx.doi.org/10.1007/s00186-012-0390-9

[3] M. Fischetti, F. Glover, A. Lodi, The feasibility pump, Mathematical Programming 104 (1) (2005) 91–104.

[4] L. Bertacco, M. Fischetti, A. Lodi, A feasibility pump heuristic for general mixed-integer problems, Discrete Optimization 4 (1) (2007) 63–76.

[5] T. Achterberg, T. Berthold, Improving the feasibility pump, Discrete Optimization 4 (1) (2007) 77–86.

[6] M. Fischetti, D. Salvagnin, Feasibility pump 2.0, Mathematical Programming Computation 1 (2009) 201–222.

[7] D. Baena, J. Castro, Using the analytic center in the feasibility pump, Operations Research Letters 39 (5) (2011) 310–317.

[8] K.-L. Huang, S. Mehrotra, An empirical evaluation of walk-and-round heuristics for mixed integer linear programs, Computational Optimization and Applications 55 (3) (2013) 545–570.

[9] K.-L. Huang, S. Mehrotra, An empirical evaluation of a walk-relax-round heuristic for mixed integer convex programs, Computational Optimization and Applications 60 (3) (2015) 559–585. doi:10.1007/s10589-014-9693-5. URL http://dx.doi.org/10.1007/s10589-014-9693-5

[10] COR@L: Computational optimization research at lehigh (http://coral.ie.lehigh.edu/mip-instances/).

[11] T. Achterberg, T. Koch, A. Martin, MIPLIB 2003, Operations Research Letters 34 (4) (2006) 361–372. doi:10.1016/j.orl.2005.07.009. URL http://www.zib.de/Publications/abstracts/ZR-05-28/

[12] 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, Mathematical Programming Computation 3 (2) (2011) 103–163. doi:10.1007/s12532-011-0025-9. URL http://mpc.zib.de/index.php/MPC/article/view/56/28

[13] E. Balas, C. H. Martin, Pivot-and-complement: A heuristic for 0-1 programming, Management Science 26 (1) (1980) 8696.

[14] E. Balas, S. Schmieta, C. Wallace., Pivot and shift - a mixed integer programming heuristic, Discrete Optimization 1 (1) (2004) 3–12.

[15] E. Balas, S. Ceria, M. Dawande, F. Margot, G. Pataki, Octane: A new heuristic for pure 0-1 programs, Operations Research 49 (2) (2001) 207–225.

[16] M. Fischetti, A. Lodi, Local branching, Mathematical Programming 98 (1) (2003) 23–47.

[17] E. Danna, E. Rothberg, C. Pape, Exploring relaxation induced neighborhoods to improve MIP solutions, Mathematical Programming 102 (1) (2005) 71–90.

APPENDIX

Table 7: Problem Profiles of COR@L Library

Problem	Lower Bound	Upper Bound	Rows	Columns	Conti	Bin	Int
neos5	15	15	63	63	10	53	0
neos-584851	-11	-11	661	445	40	405	0
neos-881765	0	$+\infty$	278	712	0	712	0
neos-905856	-8	$+\infty$	403	686	0	686	0
neos-1121679	$-\infty$	16	6	62	12	50	0
neos-1211578	-77	-77	356	260	130	130	0
neos-1228986	-123	-123	356	260	130	130	0
neos-1337489	-77	-77	356	260	130	130	0
neos-1420205	40	40	383	231	0	126	105
neos-1430701	-78	-77	668	312	156	156	0
neos-1440447	-100	-100	561	260	130	130	0
neos-1460246	2325	2606	306	285	19	266	0
neos-1480121	43	43	363	222	152	70	0
ran14x18_1	3667.46	3736	284	504	252	252	0
rlp1	-1	15	68	461	11	450	0
roy	3208.957	3208.957	162	149	99	50	0
neos14	74333.34	74333.34	552	792	656	136	0
neos15	77895.21	81525.14	552	792	632	160	0
neos-1489999	354	354	1046	534	2	532	0
neos-911880	49.85281	54.83	83	888	48	840	0
neos-504815	2296.22	2296.22	1067	674	554	120	0
bienst1	46.75	46.75	576	505	477	28	0
bienst2	54.6	54.6	576	505	470	35	0
mcf2	65.66667	65.66667	664	521	465	56	0
neos-911970	50.572	54.76	107	888	48	840	0
neos-631517	11275806	11503309	351	1090	231	859	0
neos-1439395	-182	-180.33	775	364	182	182	0
22433	21477	21477	198	429	198	231	0
neos-1620807	6	6	1340	231	0	231	0
neos-1346382	-178	-176	796	520	260	260	0
neos-1426635	-178	-176	796	520	260	260	0
23588	8090	8090	137	368	137	231	0
neos-512201	513.57	513.57	1335	838	688	150	0
neos-504674	3635.87	3635.87	1344	844	694	150	0
neos-582605	$-\infty$	1	1240	1265	865	400	0
neos-1225589	1.23E+09	1.23E+09	675	1300	650	650	0
neos-631164	10948328	11315752	406	1282	245	1037	0
neos-955215	446.5	446.5	723	1302	672	630	0
neos-1440460	-180	-179.25	989	468	234	234	0
neos-1056905	$-\infty$	30	900	463	43	420	0
neos-1595230	9	9	1750	490	0	490	0
neos-1429461	-102	-101.25	1096	520	260	260	0
neos-1467067	-103.667	-103	1084	1196	598	598	0
neos-522351	17891.08	17891.08	1705	1524	1284	240	0
neos-538867	122	122	1170	792	0	792	0
neos-1200887	-74	-74	633	234	117	117	0
neos-1616732	$-\infty$	159	1999	200	0	200	0
neos-825075	-272	-272	328	800	0	800	0

Table 7: Problem Profiles of COR@L Library (continued)

Problem	Lower Bound	Upper Bound	Rows	Columns	Conti	Bin	Int
neos-933815	759.7285	766	947	1728	888	840	0
neos-538916	134	134	1314	864	0	864	0
neos-906865	3175	3175	1634	1184	784	400	0
neos-863472	9.94541	11.69	523	588	56	532	0
binkar10_1	6742.2	6742.2	1026	2298	2128	170	0
neos11	9	9	2706	1220	320	900	0
neos-503737	50	52	500	2850	350	2500	0
neos-593853	1.17E+09	1.17E+09	1606	2400	1200	1200	0
neos-598183	18429.98	18429.98	992	1696	1260	436	0
neos-603073	16790.24	16790.24	992	1696	1260	436	0
neos-848150	0	$+\infty$	731	949	0	949	0
neos-892255	14	14	2137	1800	0	1800	0
neos-933364	760.4215	766	1006	1728	888	840	0
neos-934184	760.4215	766	1006	1728	888	840	0
neos-942323	15	17	754	732	48	684	0
neos-1311124	-182	-181	1643	1092	546	546	0
neos-1426662	-52	-44	1914	832	416	416	0
neos-1427181	-104	-102	1786	832	416	416	0
neos-1427261	-130	-127	2226	1040	520	520	0
neos-1429185	-78	-76	1346	624	312	312	0
neos-1436709	-129	-128	1417	676	338	338	0
neos-1437164	8	8	187	2256	0	2256	0
neos-1440457	-180	-179	1952	936	468	468	0
neos-1442119	-182	-181	1524	728	364	364	0
neos-1442657	-156	-154.5	1310	624	312	312	0
neos-1603512	0	5	555	730	1	729	0

Table 8: Problem Profiles of MIPLIB 2003 Library

Problem	Objective	Rows	Cols	Nonzeros	Conti	Bin	Int
10teams	924	230	2025	12150	225	1800	0
a1c1s1	11503.44	3312	3648	10178	3456	192	0
aflow30a	1158	479	842	2091	421	421	0
aflow40b	1168	1442	2728	6783	1364	1364	0
cap6000	-2451380	2176	6000	48243	0	6000	0
danoint	65.66	664	521	3232	465	56	0
disctom	-5000	399	10000	30000	0	10000	0
fixnet6	3983	478	878	1756	500	378	0
gesa2	25779900	1392	1224	5064	816	240	168
gesa2-o	25779900	1248	1224	3672	504	384	336
liu	1172	2178	1156	10626	67	1089	0
manna81	-13164	6480	3321	12960	0	18	3303
mas74	11801.2	13	151	1706	1	150	0
mas76	40005.1	12	151	1640	1	150	0
mkc	-563.846	3411	5325	17038	2	5323	0
modglob	20740500	291	422	968	324	98	0
noswot	-41	182	128	735	28	75	25
opt1217	-16	64	769	1542	1	768	0
p2756	3124	755	2756	8937	0	2756	0

Table 8: Problem Profiles of MIPLIB 2003 Library (continued)

Problem	Objective	Rows	Cols	Nonzeros	Conti	Bin	Int
pp08aCUTS	7350	246	240	839	176	64	0
pp08a	7350	136	240	480	176	64	0
rout	1077.56	291	556	2431	241	300	15
set1ch	54537.8	492	712	1412	472	240	0
seymour	423	4944	1372	33549	0	1372	0
timtab1	764772	171	397	829	226	64	107
timtab2	1096560	294	675	1482	381	113	181
tr12-30	130596	750	1080	2508	720	360	0
vpm2	13.75	234	378	917	210	168	0

Table 9: Problem Profiles of MIPLIB 2010 Library

Problem	Rows	Columns	Nonzeros	Int	Bin	Conti
30n20b8	576	18380	109706	7344	11036	
acc-tight5	3052	1339	16134		1339	
aflow40b	1442	2728	6783		1364	1364
air04	823	8904	72965		8904	
app1-2	53467	26871	199175		13300	13571
bab5	4964	21600	155520		21600	
beasleyC3	1750	2500	5000		1250	1250
biella1	1203	7328	71489		6110	1218
bienst2	576	505	2184		35	470
binkar10_1	1026	2298	4496		170	2128
bnatt350	4923	3150	19061		3150	
core2536-691	2539	15293	177739		15284	9
cov1075	637	120	14280		120	
csched010	351	1758	6376		1457	301
danoint	664	521	3232		56	465
dfn-gwin-UUM	158	938	2632	90		848
eil33-2	32	4516	44243		4516	
eilB101	100	2818	24120		2818	
enlight13	169	338	962	169	169	
ex9	40962	10404	517112		10404	
glass4	396	322	1815		302	20
gmu-35-40	424	1205	4843		1200	5
iis-100-0-cov	3831	100	22986		100	
iis-bupa-cov	4803	345	38392		345	
iis-pima-cov	7201	768	71941		768	
lectsched-4-obj	14163	7901	82428	236	7665	
m100n500k4r1	100	500	2000		500	
macrophage	3164	2260	9492		2260	
map18	328818	164547	549920		146	164401
map20	328818	164547	549920		146	164401
mcsched	2107	1747	8088	14	1731	2
mik-250-1-100-1	151	251	5351	150	100	1
mine-166-5	8429	830	19412		830	
mine-90-10	6270	900	15407		900	
msc98-ip	15850	21143	92918	53	20237	853
mzzv11	9499	10240	134603	251	9989	

Table 9: Problem Profiles of MIPLIB 2010 Library (continued)

Problem	Rows	Columns	Nonzeros	Int	Bin	Conti
n3div36	4484	22120	340740		22120	
n3seq24	6044	119856	3232340		119856	
n4-3	1236	3596	14036	174		3422
neos-1109824	28979	1520	89528		1520	
neos-1337307	5687	2840	30799		2840	
neos-1396125	1494	1161	5511		129	1032
neos13	20852	1827	253842		1815	12
neos-1601936	3131	4446	72500		3906	540
neos18	11402	3312	24614		3312	
neos-476283	10015	11915	3945693		5588	6327
neos-686190	3664	3660	18085	60	3600	
neos-849702	1041	1737	19308		1737	
neos-916792	1909	1474	134442		717	757
neos-934278	11495	23123	125577		19955	3168
net12	14021	14115	80384		1603	12512
newdano	576	505	2184		56	449
noswot	182	128	735	25	75	28
ns1208400	4289	2883	81746		2880	3
ns1688347	4191	2685	66908		2685	
ns1830653	2932	1629	100933		1458	171
opm2-z7-s2	31798	2023	79762		2023	
pg5_34	225	2600	7700		100	2500
pigeon-10	931	490	8150		400	90
pw-myciel4	8164	1059	17779	1	1058	
qiu	1192	840	3432		48	792
rail507	509	63019	468878		63009	10
ran16x16	288	512	1024		256	256
reblock67	2523	670	7495		670	
rmatr100-p10	7260	7359	21877		100	7259
rmatr100-p5	8685	8784	26152		100	8684
rmine6	7078	1096	18084		1096	
rocII-4-11	21738	9234	243106		9086	148
rococoC10-001000	1293	3117	11751	124	2993	
roll3000	2295	1166	29386	492	246	428
satellites1-25	5996	9013	59023		8509	504
sp98ic	825	10894	316317		10894	
sp98ir	1531	1680	71704	809	871	
tanglegram1 b	68342	34759	205026		34759	
tanglegram2	8980	4714	26940		4714	
timtab1	171	397	829	107	64	226
triptim1	15706	30055	515436	9597	20451	7
unitcal_7	48939	25755	127595		2856	22899
vpphard	47280	51471	372305		51471	
zib54-UUE	1809	5150	15288		81	5069
ns1758913	624166	17956	1283444		17822	134
netdiversion	119589	129180	615282		129180	
mspp16	561657	29280	27678735		29280	
bley_xl1	175620	5831	869391	5831		