Computers & Operations Research 77 (2017) 44–57

Contents lists available at ScienceDirect

Computers & Operations Research

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

A parallel local search framework for the Fixed-Charge Multicommodity Network Flow problem

Lluís-Miquel Munguía [a,*], Shabbir Ahmed [b], David A. Bader [a], George L. Nemhauser [b], Vikas Goel [c], Yufen Shao [c]

[a] College of Computing, Georgia Institute of Technology, Atlanta GA 30332, United States
[b] Industrial and Systems Engineering, Georgia Institute of Technology, Atlanta GA 30332, United States
[c] ExxonMobil Upstream Research Company, Houston, TX 77098, United States

ARTICLE INFO

Article history:
Received 8 October 2015
Received in revised form
5 July 2016
Accepted 20 July 2016
Available online 22 July 2016

Keywords:
FCMNF
Parallel computing
Primal heuristics
Discrete optimization
Multicommodity capacitated network design

ABSTRACT

We present a parallel local search approach for obtaining high quality solutions to the Fixed Charge Multicommodity Network Flow problem (FCMNF). The approach proceeds by improving a given feasible solution by solving restricted instances of the problem where flows of certain commodities are fixed to those in the solution while the other commodities are locally optimized. We derive multiple independent local search neighborhoods from an arc-based mixed integer programming (MIP) formulation of the problem which are explored in parallel. Our scalable parallel implementation takes advantage of the hybrid memory architecture in modern platforms and the effectiveness of MIP solvers in solving small problems instances. Computational experiments on FCMNF instances from the literature demonstrate the competitiveness of our approach against state of the art MIP solvers and other heuristic methods.

1. Introduction

The Fixed-Charge Multicommodity Network Flow (FCMNF) problem is a classic optimization problem arising in numerous applications. Given a directed capacitated network and a set of commodities, the objective is to route every commodity from its origin to destination through the network so as to minimize the total cost. The cost associated with an arc is the sum of a fixed cost derived from its use and a variable cost proportional to the flow going through it. The total cost is derived from the sum of all arc costs.

The FCMNF problem was proven to be NP-Hard [1]. In practice, realistic sized instances of the FCMNF problem are extremely difficult to solve to optimality. Consequently a variety of heuristic approaches and integer programming techniques have been developed and proven to be effective means to achieve high quality solutions quickly. In this paper, we introduce a local search heuristic framework for the FCMNF problem that is explicitly designed for both parallel shared-memory systems and distributed-memory systems. Our method finds competitive solutions by exploring a large number of local search neighborhoods concurrently. Given a feasible solution s, the local searches proceed by solving restricted instances of the problem where flows of certain commodities are fixed to those in the solution s while that of the other commodities are optimized. We take advantage of a state-of-the-art Mixed Integer Programming (MIP) solver to drive these local searches.

Recent works have introduced successful heuristic methods for obtaining high quality solutions. Most common heuristics consist of embedding a problem-specific mechanism for improving solutions in the context of a metaheuristic search framework. Ghamlouche et al. [2] identify cycles in the network as a heuristic strategy for finding alternative flow routes. The same methodology is used in further works in combination with machine learning techniques in order to improve and guide the local search [3]. Chouman et al. [4] use a similar approach to identify arc-balanced cycles in combination with a Tabu Search. A different heuristic approach is presented by Yaghini et al. [5], where the authors define local search neighborhoods based on simplex pivots in the context of a simulated annealing framework. Other meta-heuristic frameworks for the FCMNF problem are based on Evolutionary algorithms [6] and Scatter Search procedures [7–9]. The latter works were developed more than a decade ago and differ in the generation of the original population, and the mechanisms used for solution improvement and recombination.

Heuristic strategies can also be used in the context of an exhaustive search framework. An example is the local branching technique introduced by Fischetti et al. [10]. Their method uses linear inequalities to branch on smaller subproblems, which are

* Corresponding author.
E-mail address: lluis.munguia@gatech.edu (L.-M. Munguía).

http://dx.doi.org/10.1016/j.cor.2016.07.016
0305-0548/

solved by a black-box MIP solver. Examples of applications of local branching for the FCMNF problem are studied by Rodríguez-Martín et al. [11]. The efficacy of the previously cited work resides in the use of heuristics algorithms in combination with exact mixed-integer programming techniques.

Katayama et al. [12] develop a column generation, path-based formulation enhanced by strong inequalities in conjunction with an arc capacity scaling approach. In [13], the same scheme is improved by using local branching ideas to polish the solutions obtained through arc capacity scaling strategies.

Despite providing high quality solutions quickly, heuristic methods cannot provide optimality certificates because of their exclusive focus on primal solutions. Hewitt et al. [14] introduce an algorithm that provides lower bounds on the optimal solution in addition to primal solution improvements. Such improvements are found by solving strategically restricted MIP subproblems while tighter lower bounds are found with mathematical programming approaches. In further work, their approach combines the use of restricted MIPs in the context of a branch-and-price framework that also provides a performance guarantee upon completion [15]. The authors take advantage of parallelism to solve the pricing problems and restrictions.

Parallelizations of large neighborhood search algorithms have been successfully implemented in other applications such as the LNG inventory routing problem [16]. To our knowledge, parallel computing remains a relatively unexplored field for the FCMNF problem. Crainic et al. [17] propose an asynchronous parallel Tabu Search where every processor communicates with a centralized solution pool. They introduce and test several communication policies as well as strategies for handling the exchanged information. In [18], special emphasis is put on the control of the information diffusion between the different processors. The authors present a multilevel parallel local search algorithm that employs parallel cycle-based Tabu Searches defined by sets of fixed arcs. Their approach differs greatly from ours in many aspects. These include the solution improvement method used, the fact that our method has a solution recombination step, the arrangement and synchronization of parallel resources, the communication protocol, and the information exchanged between processors. Crainic et al. [19] provide a comprehensive literature review on the application of parallelism in meta-heuristics. Our contribution is a highly scalable parallel algorithm specifically designed to find quality primal solutions of large-scale FCMNF problems. Many algorithmic enhancements are combined in order to attain competitive levels of parallel performance: a novel parallel decomposition procedure based on the problem structure, a highly parallelizable local search scheme, and a tiered parallel procedure that is able to combine large numbers of partial solutions quickly. Solution crossover methods such as the one used in our approach have already been introduced and discussed previously [20,21]. In contrast to these works, we introduce a parallelization of the method that enables the recombination of a large number of solutions simultaneously.

We present experimental results that show the effectiveness of our parallel local search approach. For the instances in the C problem set [22], our method identifies primal solutions that are within an average optimality GAP of 0.58% with respect to the best known lower bound in an average time of 152 s per instance. We also test our parallel algorithm against the GT problem set [14], which contains substantially bigger instances. Our method takes less than 200 s on average to obtain a better solution than the best one found by CPLEX running for 5 h. We are able to identify considerably better solutions in more time. In addition, we present parallel scalability and load balancing performance results, which show that our novel implementation is able to take advantage of a large number of parallel processors to effectively reduce computation times in a load balanced execution.

The remainder of the paper is structured as follows. Section 2 presents an arc based MIP formulation of the FCMNF problem. Sections 3 and 4 provide a detailed description of our local search methodology and its parallel implementation on hybrid-memory parallel architectures, respectively. Section 5 presents computational experiments and results on standard instances from the literature. Finally, Section 6 provides some concluding remarks.

2. Problem description

Our local search approach is based on an arc-based MIP formulation of the FCMNF problem, which is described as follows. Let $G = (V, A)$ be a directed network, where V is the set of vertices and A the set of arcs. Let K be a set of commodities to be routed through G. Each commodity $k \in K$ is specified by a source vertex $s_k \in V$, a destination $t_k \in V$ and a quantity q_k of flow to be routed. Each arc $(i, j) \in A$ has an associated fixed cost f_{ij} that is imposed only when commodities are routed through it. Arcs also have a variable cost c_{ij} that is proportional to the flow traversing it and a maximum flow capacity u_{ij}. The problem consists of finding a routing for every commodity in K such that the arc capacities are respected and the costs are minimized. Let the flow variable x_{ij}^k denote the proportion of commodity $k \in K$ that is routed through the arc $(i, j) \in A$. In addition to the flow variables, we also introduce the binary variables y_{ij}, which reflect whether each arc (i, j) is used. The FCMNF problem can then be formulated as the following MIP:

$$\min_{x,y} \sum_{k \in K} \sum_{(i,j) \in A} c_{ij} q_k x_{ij}^k + \sum_{(i,j) \in A} f_{ij} y_{ij} \tag{1}$$

subject to:

$$\sum_{(i,j) \in A} x_{ij}^k - \sum_{(j,i) \in A} x_{ji}^k = d_i^k \quad \forall\, i \in V, \quad \forall\, k \in K \tag{2}$$

$$\sum_{k \in K} q_k x_{ij}^k \leq u_{ij} y_{ij} \quad \forall\, (i, j) \in A \tag{3}$$

$$y_{ij} \in \{0, 1\} \quad \forall\, (i, j) \in A \tag{4}$$

$$0 \leq x_{ij}^k \leq 1 \quad \forall\, (i, j) \in A, \quad \forall\, k \in K. \tag{5}$$

Restriction (2) ensures the conservation of flow. The flow differential for a vertex i and a commodity k is expressed by d_i^k, which is defined as:

$$d_i^k = \begin{cases} 1 & \text{if } i = s_k \\ -1 & \text{if } i = d_k \\ 0 & \text{otherwise.} \end{cases}$$

The coupling constraints (3) guarantee that the flow through each arc does not exceed the arc capacity. The capacity restrictions have a two-fold function, as they also ensure that the fixed cost is imposed when an arc is used. All commodity flow variables relative to the same arc are aggregated in the same constraint. A tighter and stronger LP relaxation can be obtained by introducing a set of $|A| \cdot |K|$ independent constraints:

$$x_{ij}^k \leq y_{ij} \quad \forall\, (i, j) \in A, \quad \forall\, k \in K. \tag{6}$$

These are redundant with respect to (3). We choose not to include them in our model due to performance issues resulting from their

large number.

We consider two problem variations that differ in whether each commodity may be split through multiple paths or not. In the formulation above, the flow variables are continuous, as specified in constraint (5). Alternatively, each variable x_{ij}^k is binary and (5) is replaced by:

$$x_{ij}^k \in \{0, 1\} \quad \forall (i, j) \in A, \quad \forall k \in K. \tag{7}$$

The techniques described in this paper are compatible with both problem variants.

3. Local search methodology

In the proposed local search scheme, an initial primal solution to an FCMNF instance is improved iteratively by sequentially applying heuristic local searches. In each heuristic local search, solutions are improved by solving a smaller, tractable MIP subproblem that is derived from the original instance. Such reduction in the problem size is obtained by fixing a chosen subset of the variables to the corresponding values of the previously obtained feasible solution. The selection of variables is such that arc sharing is encouraged to reduce costs. New improved solutions replace the primal incumbent after each iteration, and the scheme is repeated.

We parallelize this procedure by partitioning the sequence of local searches to an arbitrary number of independent sequences, each of which can be explored in parallel. To achieve this decomposition, we introduce a local search partitioning mechanism, which determines the work to be performed by each of the parallel processors. Potentially, each independent subproblem sequence can produce an improved primal solution. The final step in the algorithm combines the solution improvements found in parallel into a single feasible solution using a recombination scheme. This parallel procedure may be repeated an arbitrary number of times by using the obtained solution as an input for the next iteration. We depict this parallel local search procedure in Fig. 1. Next, we describe each component of the overall method in detail.

3.1. The local search mechanism

At the most basic level of the algorithm, improvements in solutions are found by solving restricted instances of the original problem, in which flows of certain commodities are fixed. Given a commodity c and a feasible solution s, we define the set of adjacent commodities $Adj(c)$ as the group of commodities that share flow at least in one arc with c in s, i.e., $Adj(c) = \{c' \in K | \exists (i, j) \in A \text{ s. t. } x_{(i,j)}^c > 0 \text{ and } x_{(i,j)}^{c'} > 0\}$. A local search neighborhood is then defined in the form of a new MIP subproblem, where the flow of the adjacent commodities and c are free and the remaining flow x is taken from s and fixed. Simultaneously, the arc use variables y are also free for arcs that are not used by the remaining flow. A pseudo-code description of the local search procedure is given in Algorithm 1.

Algorithm 1. Local neighborhood search.

Input: Feasible solution s and commodity c from a FCMNF instance
Output: Feasible solution to the FCMNF instance
function LocalSearch(*Solution s*, *Commodity c*)
  Compute $Adj(c)$ based on s
  for all commodities k not in $Adj(c)$ **do**
    Fix the flow variables x_{ij}^k to the values in s, $\forall (i, j) \in A$
  end for
  solveSubproblem()
  return bestSolution()
end function

Depending on the selected commodity, the resulting subset of fixed variables may vary in size. Commodities with a large number of adjacent commodities produce difficult MIP subproblems due to the small number of variable fixings. In contrast, commodities with little flow interaction may yield excessively restricted local searches. Disparities in the instance size can be reduced by establishing a threshold on the number of variables that can be fixed in addition to an optimization time limit.

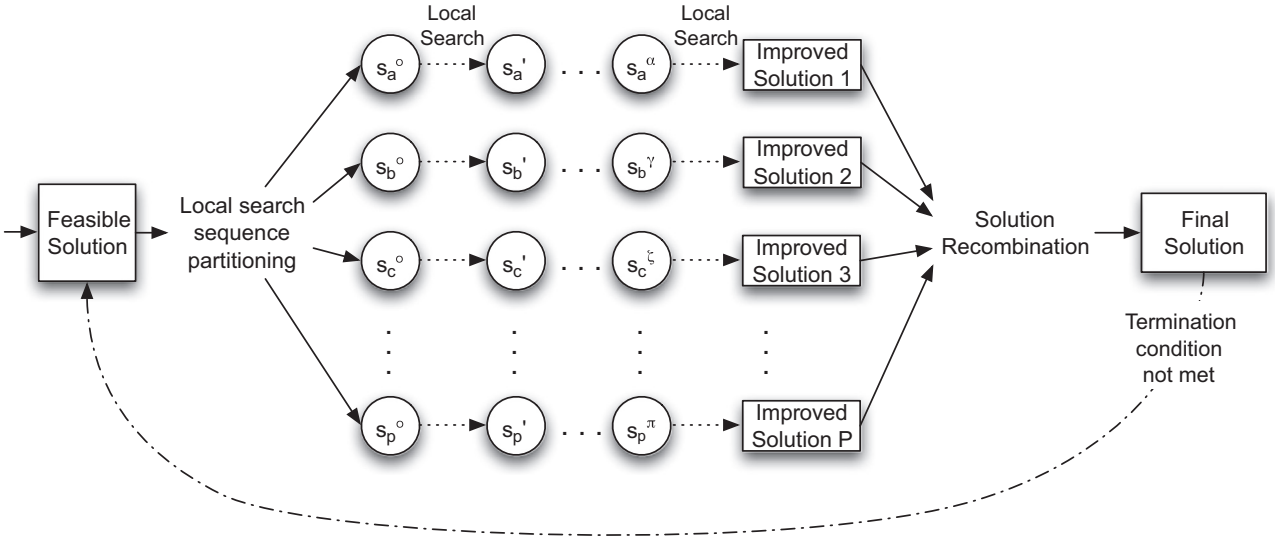


Fig. 1. Parallel decomposition of a sequential local search procedure. Local searches are distributed in a total of P sequences. As a result, P feasible solutions are obtained and recombined in an improved solution.

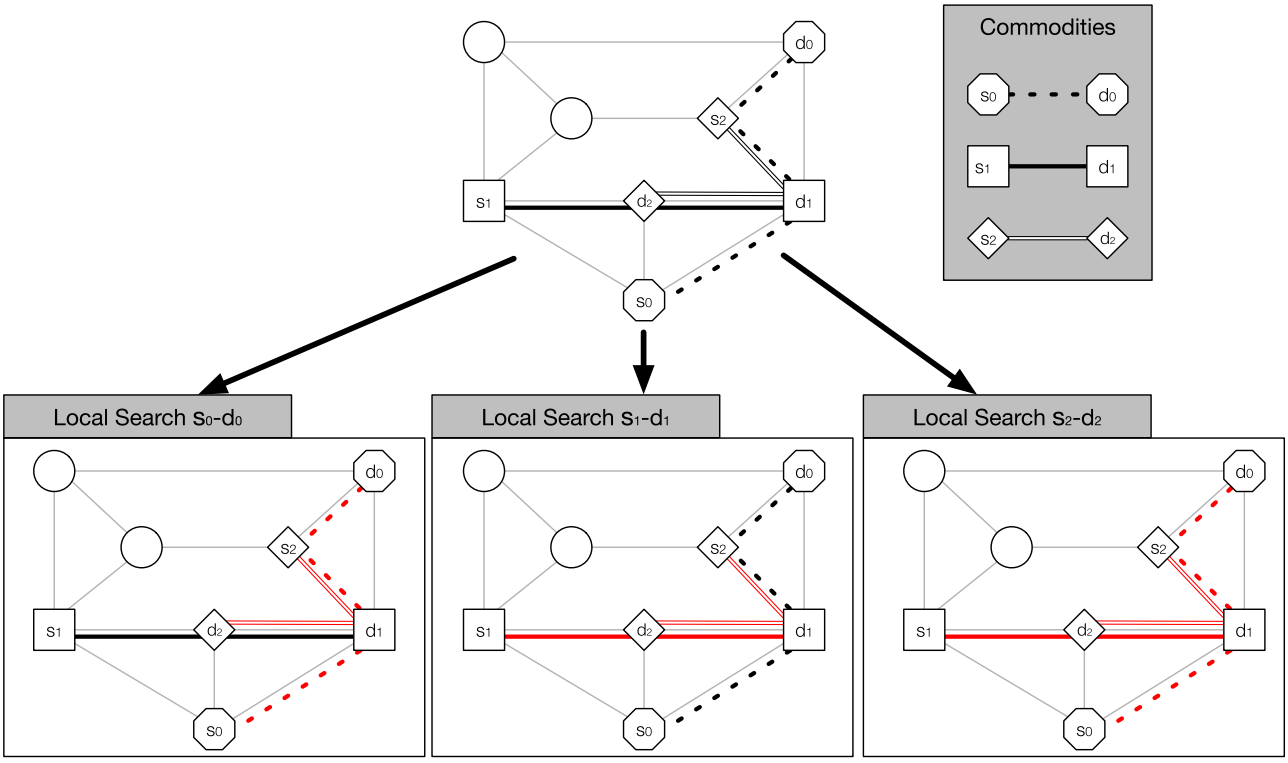


Fig. 2. Subproblem partitioning shown on a small example with three commodities. A subproblem is defined for each commodity, where the flow highlighted in red is optimized and the remaining flow is fixed. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)

3.2. Partitioning the subproblem sequence

Given the above local search neighborhood definition and a feasible solution to a FCMNF instance, we can define as many different variable fixings as commodities in the instance. A small example is shown in Fig. 2. Each derived MIP subproblem is characterized by a specific commodity and its adjacent flow and can be optimized in parallel.

Consider a set of local searches to be explored, each of which is identified by a specific commodity. As a first step towards parallelization, we require such work to be decomposed into a set of disjoint local search subsets. Work partitions that yield load-balanced optimizations are highly desirable. As an additional requirement, we incentivate that commodities with heavy flow interaction should be placed in the same subset. With this specific grouping, we would expect each local search subset to correspond to a highly interrelated subset of the variables.

Algorithm 2. Solution recombination algorithm.

Input: Feasible solution set S
Output: Feasible solution to the FCMNF instance with an accumulation of the improvements.
function SolutionRecombination(*Solution set S*)
  for $(i, j) \in Arcs$ **do**
    if (i, j) is not used in any $s \in S$ **then**
      Fix variable y_{ij} to 0
    end if
  end for
  Add all $s \in S$ as starting solutions
  solveSubproblem()
  return bestSolution()
end function

The partitioning problem can be transformed into a graph partitioning problem as follows. A new weighted graph $G^s = (V^s, A^s)$, called the connection graph, is determined from a feasible solution s, the set of arcs A and the set of commodities K. In G^s, the set K represents the vertices: $V^s = K$. $A^s = \{(u, v) | u, v \in V^s$ and $\exists\ (i, j) \in A$ s. t. $x^u_{(i,j)} > 0$ and $x^v_{(i,j)} > 0\}$, i.e., there is an arc (u, v) in $G^s(V^s, A^s)$ if and only if commodities u and v share flow in an arc in s. In addition, we specify the weight of an edge $(u, v) \in A^s$ to be the number of shared arcs. An example is depicted in Fig. 3.

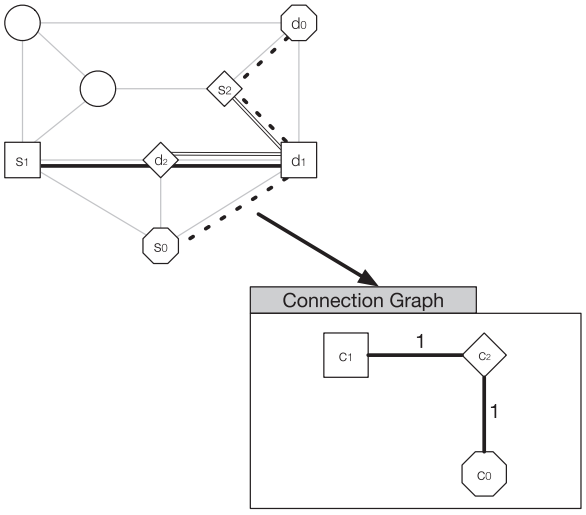


Fig. 3. Connection graph generated from a feasible solution. Commodities are represented with vertices, and there is an arc between two vertices if their corresponding commodities share an arc in the solution.

Thus, the division of commodities among P parallel processors can be translated into a graph partitioning problem of the connection graph, where the cut between the P different subsets is minimized. The connection graph partitioning can be accomplished with minimal computational efforts by specialized graph partitioning algorithms, including the Kernighan–Lin method [23], which are available in graph partitioning libraries such as Metis [24].

3.3. The solution recombination procedure

Consider a set S of improved solutions obtained from a single original feasible solution after a parallel local search. Given the nature of the local search neighborhood, it is likely that the solutions that compose S are highly similar in most parts of the arc design. Solutions can be effectively combined by focusing on the arc variations that have been produced as a product of the local search phase. To do so, a new MIP subproblem is devised, where the y variables corresponding to the arcs that are not used in any solution from the set are fixed to zero. Note that each of the solutions in S is still feasible in this new MIP subproblem and can, therefore, be used as a starting solution. In Algorithm 2 a pseudocode description of the solution recombination mechanism is given and a small example is shown in Fig. 4.

Similar to our heuristic local search scheme, the subset of fixed variables in each solution recombination may vary in size. Solutions in the earlier stages of the computation may incorporate many changes in the routing, resulting in a difficult MIP subproblem due to the small number of arc fixings. The opposite effect may be obtained if little improvement is found during the local search phase. We resolve the differences in the problem size by specifying an optimization time limit.

3.4. Obtaining a first feasible solution

The parallel method presented in this paper relies on an original feasible solution to improve upon. A starting solution is produced by solving a relaxation of the original problem, which is obtained by removing the fixed costs from the objective function computation. The fixed costs are a major complicating factor in solving the FCMNF problem. When they are omitted we have an LP if the flow can be split and an IP otherwise. We use these simplified models to obtain preliminary primal solutions that satisfy the flow restrictions, although the solutions are far from being optimal. In our experience, however, the quality of the first feasible solution has proven to be unimportant because great progress is always achieved in the initial iterations of the scheme.

4. Parallel implementation

In this section, we present more details of the algorithm implementation. For parallel scalability, our scheme is designed for hybrid memory systems that combine both distributed-memory systems as well as parallel shared-memory machines. Throughout the paper, we refer to a computing node (or processor) as a set of multiple CPU cores that share a single, unified memory subsystem as shown in Fig. 5a *(shared-memory system)*. For scalability, parallel execution may take place across several

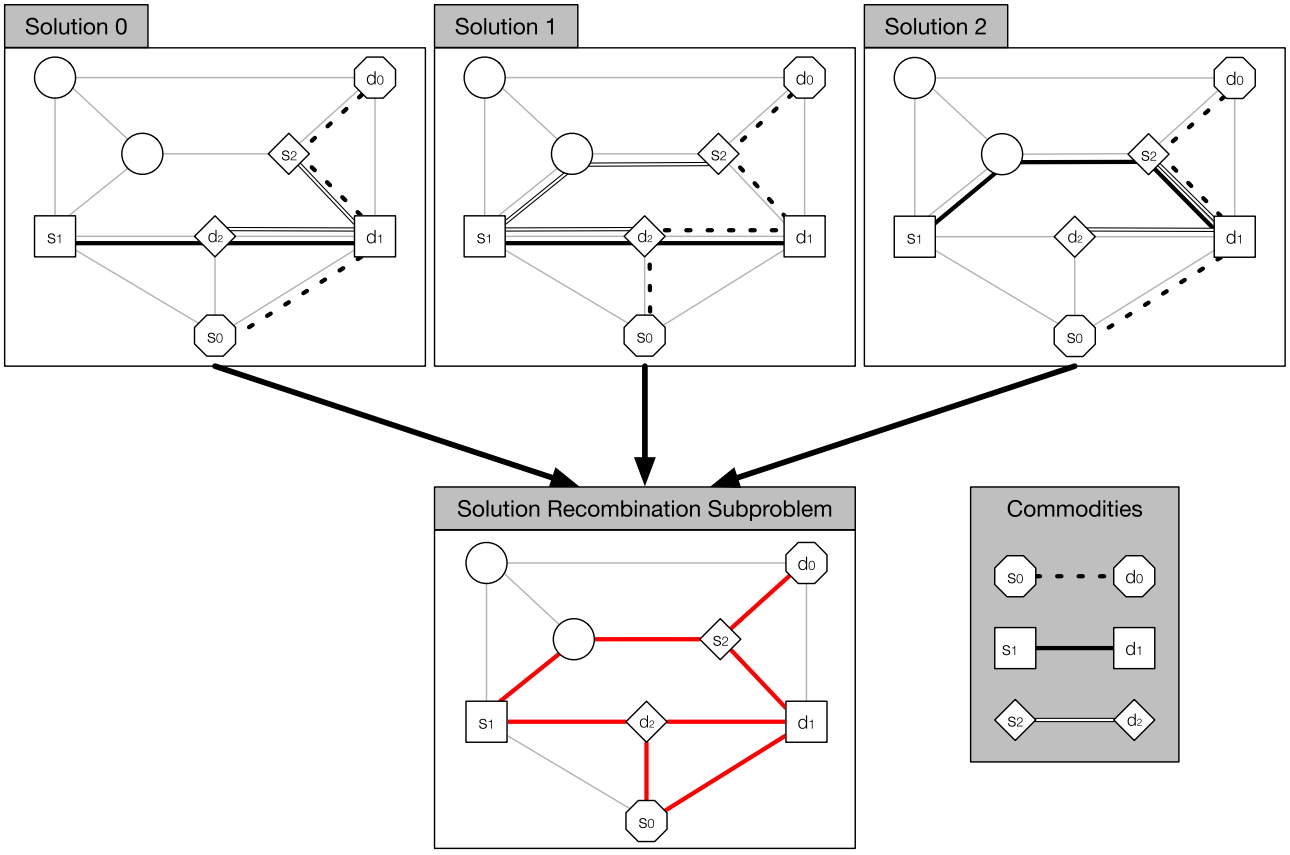
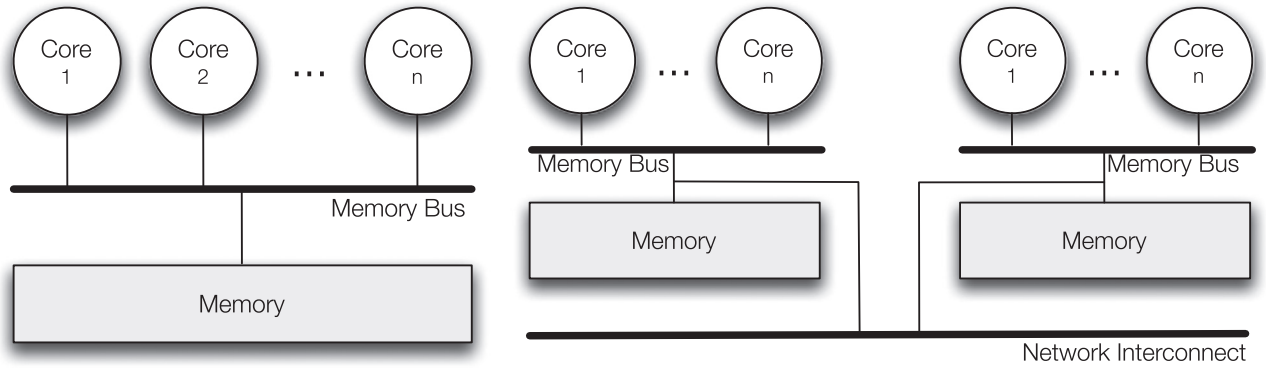


Fig. 4. The solution recombination step is shown on a small example with three commodities. A subproblem is defined for a set of input solutions, where the flow in the arcs highlighted in red is optimized while the remaining arcs fixed. (For interpretation of the references to color in this figure caption, the reader is referred to the web version of this paper.)



(a) Shared-memory system (b) Distributed-memory system

Fig. 5. Schematic depiction of parallel systems with different memory configurations. System (a) features a unified memory space, which is accessible by every parallel core simultaneously. In contrast, system (b) has a segmented memory space, which is distributed between the different parallel cores.

computing nodes concurrently. In this case, the memory space is segmented and distributed between the individual processors as depicted in Fig. 5b (*distributed-memory system*). As such, a collection of parallel processors constitute a parallel distributed-memory system. Efficient implementations require algorithm design techniques tailored for each memory environment. An important distinction in the implementation is found in the synchronization of parallel components. Parallel distributed-memory systems usually rely on synchronous message passing techniques such as MPI [25] to perform communications between processors. In contrast, communication between the parallel cores in a single computing node is much simplified because memory is shared.

We start describing our parallel scheme by its implementation in a single shared-memory parallel processor as shown in Algorithm 3. Consider a processor with C parallel processor cores and a starting solution s that is initially given to each of them. Each core proceeds to improve its local solution in parallel by resolving a different sequence of MIP subproblems and accumulating the improvements found in the optimization process. Given that the parallel cores share the same memory space, the work partitioning can be arranged dynamically. In order to do so, we employ a shared data structure which holds the subproblems that every core has access to. When a subproblem is solved, it is removed from the shared set. In this fashion, subproblems are distributed dynamically among the parallel cores such that the routing of each commodity is optimized by exactly one of them.

Each variable fixing yields a smaller integer problem as previously described, which is then solved using a black-box MIP solver. As a product of the parallel local search, improvements in the routing are accumulated in at most C different solutions. We employ our solution recombination step to combine them to a single feasible solution, which may be used as input to the next iteration. To ensure full system utilization, we may assume the number of commodities given by $|K|$ to be greater than the overall number of parallel cores P. When P is bigger than $|K|$, only $|K|$ parallel cores are used during the local search phase. The algorithm is designed such that each parallel core may improve its local solution copy by sequentially solving multiple MIP subproblems and replacing the solution when improvements are found.

We adapt our implementation to a hybrid memory parallel system by augmenting Algorithm 3 in a two-layered scheme, where each level is dedicated to a different level of parallelism. The first layer is responsible for the local search partitioning between the different distributed-memory processors by the use of MPI. The second execution tier takes place in the shared-memory setting of each parallel processor and is responsible for the actual local search exploration. It is in this inner execution level where the MIP subproblems are collectively resolved by the parallel cores in each processor.

Algorithm 3. Parallel local search iteration.

Input: Feasible solution s from a FCMNF instance, set of commodities K
Output: Improved feasible solution to the FCMNF instance
function ParallelLocalSearch(*Solution s*, *Commodity set K*)
  for every thread $t_i \in C$ in parallel **do**
    Initialize Sol_{t_i} as a copy of s
    while there exists commodity $k \in K$ **do**
      Remove k from K
      $newSol$ = LocalSearch(Sol_{t_i}, k)
      if $newSol$ represents improvement over Sol_{t_i} **then**
        $Sol_{t_i} = newSol$
      end if
    end while
  end for
  return SolutionRecombination(*Sol*)
end function

Algorithm 4. Distributed-memory parallel local search framework.

Input: Feasible solution s from a FCMNF instance, set of commodities K
Output: Improved feasible solution to the FCMNF instance
function DistributedLocalSearch(*Solution s*)
  while termination criteria is not met **do**
    Let *Part* be a partition of the commodity set K in P subsets based on s
    for all processors P_i in Parallel **do**
      $Solution_{P_i}$ = ParallelLocalSearch (s, $Part_{P_i}$)
      AllProcSolutions = All-To-AllExchange ($Solution_{P_i}$)
    end for
    s = SolutionRecombination(*AllProcSolutions*)
  end while
  return s
end function

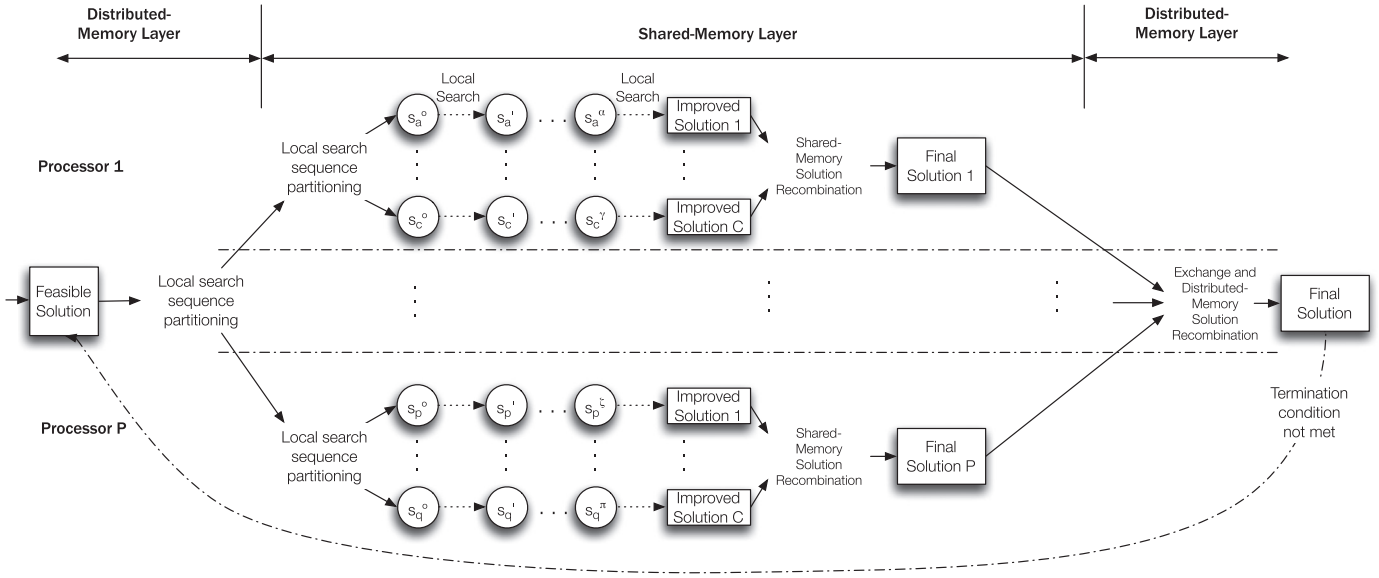


Fig. 6. Parallel hybrid memory framework. The process described in Fig. 1 is expanded to accommodate the parallel execution in distributed-memory systems.

Fig. 6 describes the interactions between each tier and the workflow of the execution. As a first step, the set of MIP subproblems is partitioned and distributed among the processors. After the partitioning, each computing node proceeds to find improvements in the primal solution by the same procedure as shown in Algorithm 3. The execution returns to the distributed-memory context, where an all-to-all solution exchange communication is performed in order to combine the improvements found by the different processors. In Algorithm 4 we present a pseudo-code description of the distributed-memory execution layer.

A distributed-memory implementation raises the issue of scalability. The effectiveness of the recombination step, for instance, is dependent on the input size. If the number of input solutions is small and all of them are highly similar, the resulting recombination MIP may be small and relatively easy to solve. However, a large number of input solutions may produce a very small number of fixings and, therefore, a problem that may be hard to optimize quickly. Thus, its scalability may be limited.

By splitting the recombination process into two consecutive phases, a large number of input solutions can be accumulated while maintaining a large number of fixings. An additional benefit lies in a better utilization of the parallelism, since each processor performs the first phase of the recombination independently.

In addition to scalability, load balancing is another component of parallel efficiency. Load balancing refers to the uniform distribution of work among parallel processors. Due to the synchronous nature of our algorithm, achieving an even load balance is essential in order to maximize the throughput of our parallel computations. Fig. 7 depicts all the potential load imbalance pitfalls of our parallel implementation. First, the parallel cores within a processor may compute each of the local search sequences

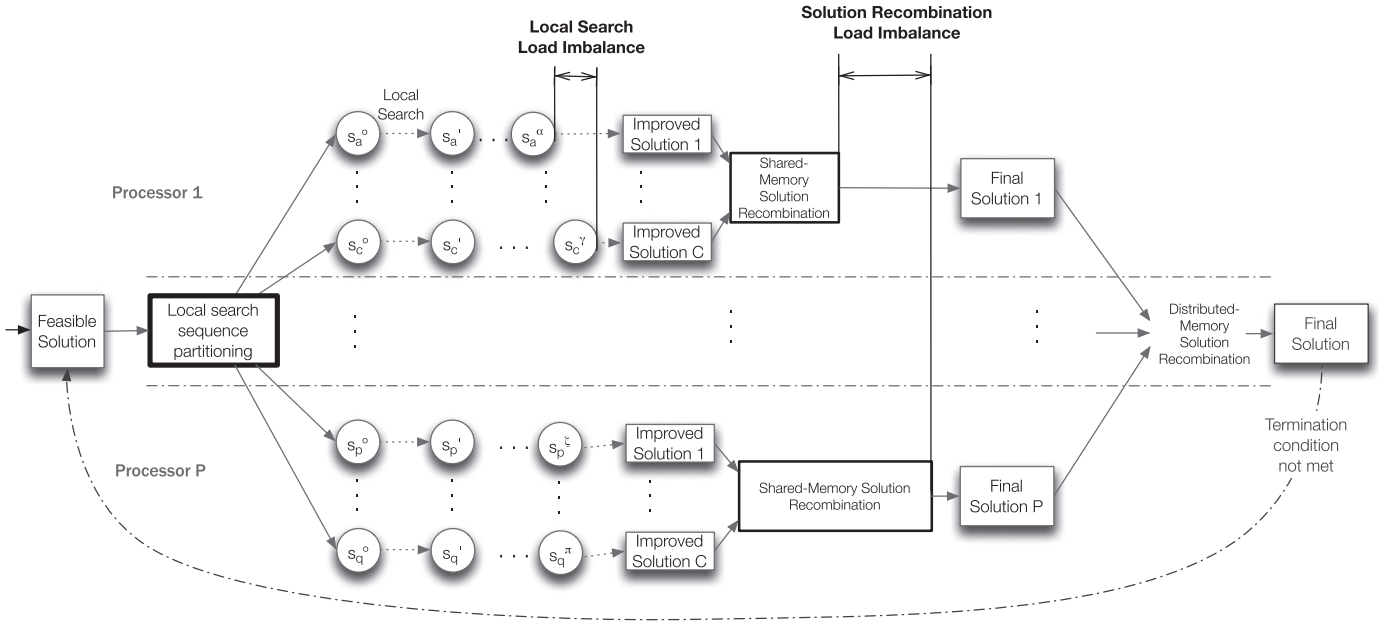


Fig. 7. Potential load imbalance causes of the implementation.

unevenly. Additionally, load imbalance could be aggravated by differences in the solving time of the first solution recombination. Both factors could affect the synchronization between processors, potentially delaying the second phase of the solution recombination. Its prevention depends on how the work is partitioned and the time limit parameters that decide the granularity of the local searches as well as the recombination times. Further experiments presented in the next section will determine the degree of load imbalance of our approach.

Another positive aspect of the synchronous implementation is an efficient handling of the communications between parallel processors. Overall, only two communication steps are required every iteration. A communication of the work partitions is performed prior to the local search, as well as an all-to-all exchange of solutions between each of the recombination phases. In both cases, collective communication primitives can be used in order to reduce overhead.

5. Experimental results

In this section, we study the performance of our parallel local search in terms of solution quality, parallel scalability and load balance. Our framework is implemented in C++ and uses CPLEX 12.4 as the MIP solver. Our tests were performed on an 8-node computing cluster, with each node having two Intel Xeon X5650 6-core processors, 24 GB of RAM memory and a Red Hat Enterprise Linux distribution. Unless otherwise noted, CPLEX was set to its default configuration whenever it was used for comparison purposes. CPLEX is configured with 12 threads, as it can take advantage of parallelism in shared-memory machines. We test the competitiveness of our parallel method by comparing its performance against previous heuristic approaches presented in the literature. For this purpose, two FCMNF problem instance sets are used, the C instances [22] and the GT instances [26] used in [14]. In order to assess the scalability of our parallel method, we test the performance under different processor configurations.

The choice of the local search parameters such as the search time limit can have a significant impact on the effectiveness of the method. An ideal combination is highly dependent on the instance size, the number of arcs and the number of commodities. To cope with this variety of choices, we test several parameter configurations and choose the best performing one in average. For each problem class, 8 representative instances were selected and sampled with a collection of parameters. In Fig. 8, we report the performance results for both the C instances and the GT instances in terms of the average optimality GAP. Each algorithm sample had a time limit of 100 s and 300 s for the C and the GT instances, respectively. For the remaining set of experiments in the C instance set, the time limits were set to 10 s for each local search and 20 s for solution recombination. When solving the GT instances, the local search time limit set to 20 s and 50 s was allowed for the recombination process.

5.1. C instance set performance results

The C instance set is composed of 37 FCMNF medium-sized instances. They have networks with 20 or 30 vertices, a number of arcs ranging from 230 to 700 and a number of commodities ranging from 10 to 400. The configuration of each problem instance is specified with the tuple {*Nvertices*, *NArcs*, *NCommodities*, *F*|*V*, *T*|*L*}, where *F* indicates that the instance's fixed costs are predominant in relation to the variable costs (*V* otherwise). *T* characterizes a tightly capacitated problem instance and *L* denotes loose arc capacities. In order to evaluate the quality of the solutions obtained with our scheme, we compare it against the results presented in prior publications and default CPLEX on the problem variant where the flow routing can be split between different paths. Specifically, results are compared with those reported in the IP Search scheme (*IPSearch*) by Hewitt et al. [14], two sets of results obtained with the capacity-scaling combined scheme (*Comb*1 and *Comb*2) by Katayama [13], and CPLEX with a time limit of 1 h. We refer to the results obtained with our Parallel Local Search as *ParLS*.

The performance results over the C instance set are shown in Table 1, where the best lower bound for each instance is reported followed by the best primal solution found by each method. The best values are denoted in bold. When a value is optimal, it is marked with an asterisk. We specify the time required to reach the best solution as reported originally by the authors. In order to eliminate the discrepancies between computer systems, we also report the normalized CPU times for *IPSearch*, *Comb*1, and *Comb*2 according to the CPU performance metrics in Passmark [27]. Normalized times are calculated as $T_{norm} = \frac{T_{orig} \cdot S_{orig}}{S_{norm}}$, where T_{orig} is the original time reported by the authors, while S_{norm}[1] and S_{orig}[2,3] are the CPU scores of the processors used in our experiments and the other authors in the comparison, respectively. The lower bounds for each problem were either obtained from the literature or found by CPLEX with a 12-h limit. The reported GAP values are relative to the optimal solution or to the best lower bound. It is computed as $GAP = \frac{P_{sol} - L_B}{P_{sol}} \cdot 100$, where P_{sol} is the solution to each instance and L_B its lower bound.

ParLS finds an optimal solution in 15 out of 37 instances. In comparison to the incumbents reported in the literature, better or equally good solutions are found in 20 cases. When we consider all 37 instances, we obtain solutions that are within an average optimality GAP of 0.58%. The convergence to such solutions is obtained very quickly, averaging 152 s per instance. Comparing our results to previous research is a difficult task due to the diversity in the experimental conditions and the differences in the hardware and software. However, our parallel method identifies quality solutions that are competitive with the ones reported in prior work and requires much less time to achieve them. The incumbents reported in *Comb*2 and *CPLEX* are results that represent an improvement over those obtained with our approach. But the solution times are larger by a factor of 20 and 15, respectively.

5.2. GT instances

The GT set is composed of 24 FCMNF instances, which range in the number of arcs from 2000 to 3000 and have 50 to 200 commodities. When an arc-based formulation is considered, the instance sizes range between 102,000 and 603,000 variables. They are additionally presented in two versions, whether the problem instances are tightly capacitated (F_T) or loosely capacitated (F_L). We compare the performance of our parallel local search (reported as *ParLS*) with the IP Search scheme from Hewitt et al. [14] (*IPSearch*), both with a time limit of one hour. We also compare the best results obtained by CPLEX with a time limit of 5 h (*CPLX*). We tested several CPLEX emphasis configurations, including optimality and feasibility, and found the default configuration to be the best performer. In addition to the default setting, we include a comparison against the solution polishing heuristic of CPLEX. In the *CPLXSP* setting, CPLEX is allowed 1 h of optimization time and 4 h of solution polishing. CPLEX is also used to determine the lower bound on every instance. Results are detailed in Table 2, where the best found primal solutions found are given, as well as the time required by the parallel local search to improve the best

[1] S_{norm} = 7605 (Intel Xeon X5650)
[2] $S_{Comb1} = S_{Comb2}$ = 8262 (Intel Core i7-2600)
[3] $S_{IPSearch}$ = 4452 (Intel Xeon E5520)

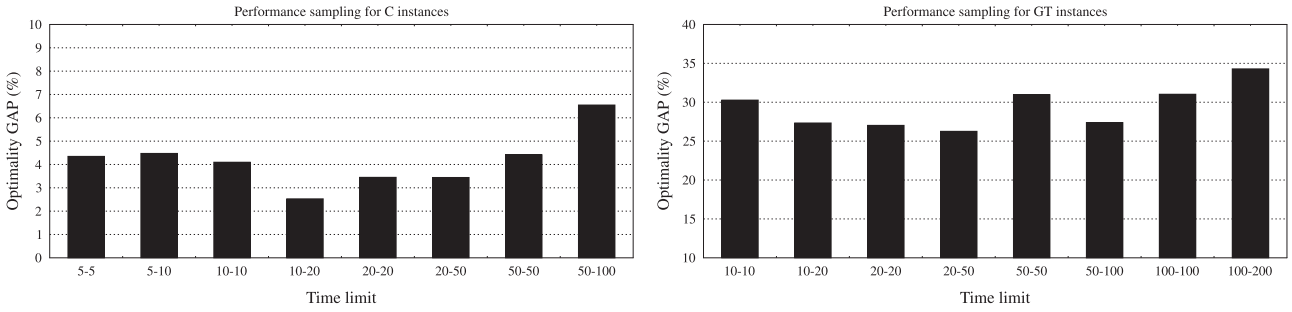


Fig. 8. Parameter sensitivity analysis for C and GT instances. Parameter configurations are specified with two numbers *X–Y*, where *X* refers to the local search time limit and *Y* to the solution recombination time limit.

Table 1
C instance set optimization results.

Problem	LB/Opt	Primal solution value					Time to best solution (s)					Normalized Time to best solution (s)				
		IPSearch	Comb1	Comb2	CPLEX	ParLS	IPSearch	Comb1	Comb2	CPLEX	ParLS	IPSearch	Comb1	Comb2	CPLEX	ParLS
100/400/010/VL	28,423*	**28,423***	28,426	**28,423***	**28,423***	28,486	35	**0**	2	1	3	20	**0**	2	1	3
100/400/010/FL	23,949*	**23,949***	24,459	**23,949***	**23,949***	24,022	9	23	106	112	**1**	5	25	115	112	**1**
100/400/010/FT	63,066	65,885	68,410	64,207	**64,143**	64,207	813	**12**	2736	3600	53	476	**13**	2974	3600	53
100/400/030/VT	384,802*	384,836	384,809	**384,802***	**384,802***	**384,802***	330	**2**	1503	680	42	193	**2**	1634	680	42
100/400/030/FL	49,018*	49,694	49,588	**49,018***	**49,018***	**49,018***	886	167	864	3600	**29**	519	182	939	3600	**29**
100/400/030/FT	132,129	141,365	142,191	138,152	138,587	**136,861**	888	**12**	11028	3600	79	520	**13**	11986	3600	79
20/230/040/VL	423,848*	424,385	**423,848***	**423,848***	**423,848***	424,075	4	**0**	1	1	2	2	**0**	1	1	2
20/230/040/VT	371,475*	371,779	371,906	**371,475***	**371,475***	371,573	41	**1**	3	**1**	**1**	24	**1**	3	**1**	**1**
20/230/040/FT	643,036*	643,187	643,649	**643,036***	**643,036***	**643,036***	45	**1**	15	5	10	26	**1**	16	5	10
20/230/200/VL	94,213*	95,097	94,218	**94,213***	94,218	**94,213***	822	**104**	3060	3600	123	481	**113**	3326	3600	123
20/230/200/FL	137,642*	141,253	137,702	**137,642***	137,854	**137,642***	691	254	4409	3600	**144**	405	276	4792	3600	**144**
20/230/200/VT	97,914*	99,410	97,968	**97,914***	**97,914***	**97,914***	821	68	2161	606	**52**	481	74	2349	606	**52**
20/230/200/FT	135,863	140,273	136,265	136,031	136,144	**135,867**	**156**	263	4538	3600	240	**91**	286	4932	3600	240
20/300/040/VL	429,398*	**429,398***	**429,398***	**429,398***	**429,398***	**429,398***	19	**0**	1	1	1	11	**0**	1	1	1
20/300/040/FL	586,077*	**586,077***	587,512	**586,077***	**586,077***	**586,077***	29	**1**	8	3	16	17	**1**	9	3	16
20/300/040/VT	464,509*	**464,509***	**464,509***	**464,509***	**464,509***	**464,509***	24	**1**	4	**1**	23	14	**1**	4	**1**	23
20/300/040/FT	604,198*	**604,198***	**604,198***	**604,198***	**604,198***	**604,198***	68	**1**	4	**1**	3	40	**1**	4	**1**	3
20/300/200/VL	74,753	75,319	74,840	**74,811**	74,929	**74,811**	802	**101**	5048	3600	361	470	**110**	5487	3600	361
20/300/200/FL	11,3862	117,543	115,801	115,748	**115,541**	115,580	686	350	7769	3600	**250**	402	380	8444	3600	**250**
20/300/200/VT	74,991*	76,198	74,995	**74,991***	**74,991***	**74,991***	388	65	2158	791	**58**	227	71	2345	791	**58**
20/300/200/FT	106,672	110,344	107,315	107,315	**107,102**	**107,102**	396	258	3798	3600	**122**	232	280	4128	3600	**122**
30/520/100/VL	53,958*	54,113	53,976	**53,958***	**53,958***	53,978	218	**7**	673	721	20	128	**8**	731	721	20
30/520/100/FL	93,570	94,388	94,201	**93,967**	**93,967**	**93,967**	226	150	3266	3600	**83**	132	163	3550	3600	**83**
30/520/100/VT	52,046*	52,174	52,248	**52,046***	**52,046***	**52,046***	455	**7**	2004	3600	32	266	**8**	2178	3600	32
30/520/100/FT	96,260	98,883	97,833	97,385	**97,107**	97,862	815	4992	4802	3600	**158**	477	5426	5219	3600	**158**
30/520/400/VL	112,735	114,042	112,787	**112,774**	**112,774**	112,787	394	**95**	5917	3600	542	231	**103**	6431	3600	542
30/520/400/FL	147,790	154,218	149,486	149,423	**149,242**	149,677	750	1184	3387	3600	**463**	439	1287	3681	3600	**463**
30/520/400/VT	114,641*	114,922	**114,641***	**114,641***	**114,641***	**114,641***	621	**54**	3726	3600	461	364	**59**	4050	3600	461
30/520/400/FT	150,685	154,606	152,630	**152,576**	153,005	154,137	466	358	5149	3600	**288**	**273**	389	5596	3600	288
30/700/100/VL	47,603*	47,612	**47,603***	**47,603***	**47,603***	**47,603***	32	**4**	64	18	179	19	**4**	70	18	179
30/700/100/FL	59,958*	60,700	60,067	**59,958***	60,066	60,058	741	228	2888	3600	**111**	434	248	3139	3600	**111**
30/700/100/VT	45,872*	46,046	46,070	**45,872***	**45,872***	45,879	371	**9**	5559	3600	258	217	**10**	6042	3600	258
30/700/100/FT	54,904*	55,609	55,164	**54,904***	**54,904***	**54,904***	387	**26**	4456	3600	173	227	**28**	4843	3600	173
30/700/400/VL	97,189	98,718	97,901	**97,875**	97,914	98,090	**222**	466	4727	3600	243	**130**	506	5138	3600	243
30/700/400/FL	131,690	152,576	134,723	**134,620**	135,892	136,257	860	2115	7312	3600	**223**	504	2299	7947	3600	**223**
30/700/400/VT	94,508	96,168	95,267	**95,250**	95,293	95,651	**365**	1570	6376	3600	374	**214**	1706	6930	3600	374
30/700/400/FT	128,243	131,629	**129,910**	**129,910**	130,140	131,104	**225**	3955	8165	3600	428	**132**	4299	8874	3600	428
Average GAP and time		1.73	0.89	0.47	0.51	0.58	408	456	3180	2317	152	239	497	3457	2317	**152**

solution found by the other three methods. The reported GAP values are calculated with respect to the best found lower bound.

The results demonstrate the considerable difficulty in solving the GT instance set, as CPLEX is only able to achieve an average optimality GAP of 24.09% after 5 h of execution. A big part of the challenge resides in the complexity of obtaining tight lower bounds due to the weakness of the arc-based formulation. The solution polishing heuristic is generally more effective than the default CPLEX configuration, even though its advantage is diminished in instances with a high commodity count. In selected instances with 50 commodities, *CPLXSP* provides solutions of similar quality as *ParLS*. However, *ParLS* achieves them in substantially less time. In comparison to all three methods, our parallel local search scheme finds better solutions for every instance. On average, it requires less than 200 s to improve the best solution found by CPLEX running for 5 h. Improvements become more noticeable in the instances with more commodities because these benefit more from parallelism and are more challenging for CPLEX.

5.3. Scaling results

One of the primary goals of our approach is to exploit a large degree of parallelism. We rely on the concurrent exploration of a large number of local searches to find competitive solutions faster.

Table 2
GT instance set optimization results.

Problem	LB	Primal solution value				Optimality GAP				Time to improve solution (s)		
		CPLX	CPLXSP	IP Search	ParLS	CPLX	CPLXSP	IP Search	ParLS	CPLX	CPLXSP	IPSearch
F_T,500,2000,50	4,326,550	5,038,580	5,100,186	4,949,780	**4,892,012**	14.13	15.17	12.59	**11.56**	114	100	178
F_T,500,2000,100	6,368,730	7,592,260	7,381,313	7,619,670	**7,273,916**	16.12	13.72	16.42	**12.44**	78	366	75
F_T,500,2000,150	7,208,800	8,640,390	9,083,303	8,807,650	**8,014,986**	16.57	20.64	18.15	**10.06**	317	155	234
F_T,500,2000,200	8,845,440	11,858,000	11,213,371	11,893,100	**10,617,796**	25.41	21.12	25.63	**16.69**	257	463	257
F_T,500,2500,50	3,927,990	4,585,510	4,448,739	4,600,200	**4,406,080**	14.34	11.71	14.61	**10.85**	72	120	72
F_T,500,2500,100	5,330,490	6,942,260	6,559,397	6,953,660	**6,365,848**	23.22	18.74	23.34	**16.26**	134	297	134
F_T,500,2500,150	5,930,530	8,094,410	7,978,909	7,571,640	**7,037,860**	26.73	25.67	21.67	**15.73**	216	302	488
F_T,500,2500,200	8,327,720	11,963,100	11,911,900	11,452,900	**10,727,261**	30.39	30.09	27.29	**22.37**	312	313	396
F_T,500,3000,50	3,529,370	4,333,310	4,069,239	4,262,350	**4,035,362**	18.55	13.27	17.20	**12.54**	99	1188	166
F_T,500,3000,100	5,442,880	7,164,410	7,046,750	7,186,810	**6,634,387**	24.03	22.76	24.27	**17.96**	229	262	214
F_T,500,3000,150	6,236,240	8,773,910	8,172,602	8,709,390	**7,517,445**	28.92	23.69	28.40	**17.04**	150	257	155
F_T,500,3000,200	7,283,080	11,236,600	11,354,647	10,390,700	**9,751,002**	35.18	35.86	29.91	**25.31**	308	259	510
F_L,500,2000,50	3,432,140	3,882,110	3,726,114	3,823,610	**3,722,839**	11.59	7.89	10.24	**7.81**	136	1022	196
F_L,500,2000,100	5,497,770	6,706,100	6,404,834	6,453,880	**6,005,177**	18.02	14.16	14.81	**8.45**	146	300	271
F_L,500,2000,150	6,750,150	8,205,000	7,886,028	8,081,600	**7,510,651**	17.73	14.40	16.48	**10.13**	198	351	211
F_L,500,2000,200	8,031,600	10,181,700	10,376,103	9,828,350	**9,338,097**	21.12	22.60	18.28	**13.99**	424	375	592
F_L,500,2500,50	3,176,040	3,818,440	3,507,652	3,612,030	**3,491,664**	16.82	9.45	12.07	**9.04**	90	1223	213
F_L,500,2500,100	5,062,110	6,893,490	6,187,629	6,400,140	**5,909,401**	26.57	18.19	20.91	**14.34**	133	1076	303
F_L,500,2500,150	6,542,600	10,022,900	9520783	9,089,920	**8,138,918**	34.72	31.28	28.02	**19.61**	155	196	319
F_L,500,2500,200	7,717,740	11,937,300	11,566,824	10,099,200	**9,788,913**	35.35	33.28	23.58	**21.16**	384	483	1976
F_L,500,3000,50	2,958,630	3,668,660	3,492,641	3,457,280	**3,369,303**	19.35	15.29	14.42	**12.19**	110	187	254
F_L,500,3000,100	4,855,420	6,692,780	6,187,593	6,015,950	**5,773,133**	27.45	21.53	19.29	**15.90**	178	377	613
F_L,500,3000,150	6,031,650	9,378,030	9,479,082	8,919,720	**7,741,294**	35.68	36.37	32.38	**22.08**	223	196	254
F_L,500,3000,200	6,722,660	1,1240,900	11,291,918	10,040,000	**9,195,115**	40.19	40.46	33.04	**26.89**	264	264	691
				Average value	24.09	21.56	20.96	15.43	196	422	365	

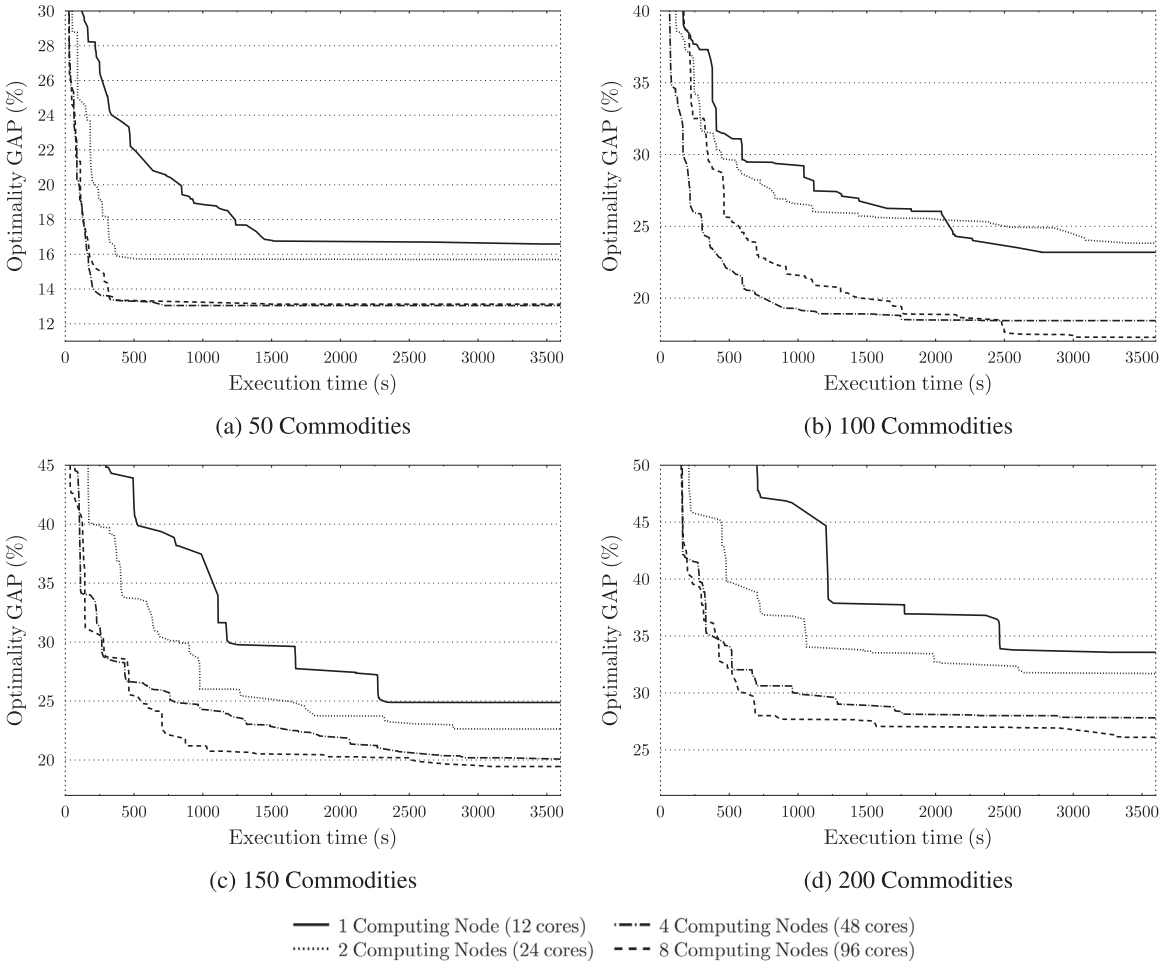


Fig. 9. Scaling results for a selected test instance with 500 vertices, 3000 arcs and varying commodities. Each plot depicts the improvement in optimality GAP with respect to the best lower bound as a function of time. The executions shown in each problem differ in the number of parallel cores used.

Table 3
Time required to reach certain gap with respect to the best found primal solution and the best lower bound.

Problem	Number of computing nodes	Time required to reach GAP with respect to best solution (s)					Time required to reach GAP with respect to best lower bound (s)				
		20%	10%	5%	1%	0%	35%	30%	25%	20%	15%
F_T,500,3000,50	1 (12 cores)	137	638	1388	–	–	18	137	315	845	–
F_T,500,3000,50	2 (24 cores)	48	184	317	–	–	10	51	146	240	–
F_T,500,3000,50	4 (48 cores)	29	85	144	227	725	9	29	62	104	182
F_T,500,3000,50	8 (96 cores)	28	78	161	314	–	3	28	48	111	282
F_T,500,3000,100	1 (12 cores)	378	2124	–	–	–	378	594	2124	–	–
F_T,500,3000,100	2 (24 cores)	285	2069	–	–	–	247	443	2529	–	–
F_T,500,3000,100	4 (48 cores)	126	300	596	–	–	81	165	303	886	–
F_T,500,3000,100	8 (96 cores)	223	517	1057	2490	3506	221	348	577	1667	–
F_T,500,3000,150	1 (12 cores)	1110	2112	–	–	–	1110	1196	2336	–	–
F_T,500,3000,150	2 (24 cores)	406	978	2325	–	–	406	825	1637	–	–
F_T,500,3000,150	4 (48 cores)	111	433	1303	2902	–	111	267	790	–	–
F_T,500,3000,150	8 (96 cores)	143	462	703	2210	3589	143	281	563	2508	–
F_T,500,3000,200	1 (12 cores)	1216	–	–	–	–	2465	–	–	–	–
F_T,500,3000,200	2 (24 cores)	478	1951	–	–	–	1060	–	–	–	–
F_T,500,3000,200	4 (48 cores)	281	518	1128	–	–	455	999	–	–	–
F_T,500,3000,200	8 (96 cores)	197	427	674	3108	3379	423	623	–	–	–

The next set of experiments is aimed at showing the effectiveness and benefits of the application of parallelism.

In Fig. 9, scaling results are shown for a representative set of instances. Since our approach parallelizes over the set of commodities, we test a variety of instances with a number of commodities ranging from 50 to 200. For each problem, we report performance results for several processor configurations, ranging from executions on one processor (12 parallel cores) to eight (96 parallel cores). The same results are specified in terms of time in Table 3, where we show the time required to reach different GAP values with respect to the best solution found for each instance as well as with respect to the lower bound.

Overall, parallelism is beneficial and substantially better solutions are achieved by using a larger number of processors. However, little improvement is observed when the number of processors exceeds or equals the commodity count. This is the case for the instances with 50 and 100 commodities, which are comparatively easier than instances with similarly sized networks and a larger number of commodities. The impact of parallelism differs from instance to instance due to their variability and the heuristic nature of our approach, including the eventuality that not every local search may yield improvements at every iteration. Instances with more commodities show better scalability, as more parallelism is exploited and there exists more opportunities for solution improvements.

5.4. Load balancing

Load balancing refers to the uniform distribution of work between parallel processors. We characterize the total execution time of a specific processor P_i as the sum of the useful computation time TC_{P_i}, the communication time TX_{P_i} and the idle time TI_{P_i}. We define the communication time as the time spent performing communication between processors, whereas the idle time TI_{P_i} accounts for the idle time spent by a processor on synchronization or waiting for other processors to finish their computations. Then, we define the utilization of a processor P_i as the ratio of useful computation time over the total execution time:

$$U(P_i) = \frac{TC_{P_i}}{TC_{P_i} + TX_{P_i} + TI_{P_i}}$$

Fig. 10 displays the average core utilization for a representative FCMNF instance under different processor configurations and different time limit parameters. Each parameter configuration is specified with two time limits, where the first number corresponds to the local search time limit and the second refers to the solution recombination time limit. We show that average processor utilizations remain very high through all the tested combinations. When a single computing node (12 cores) is used, the shared-memory dynamic work allocation mechanism proves to be

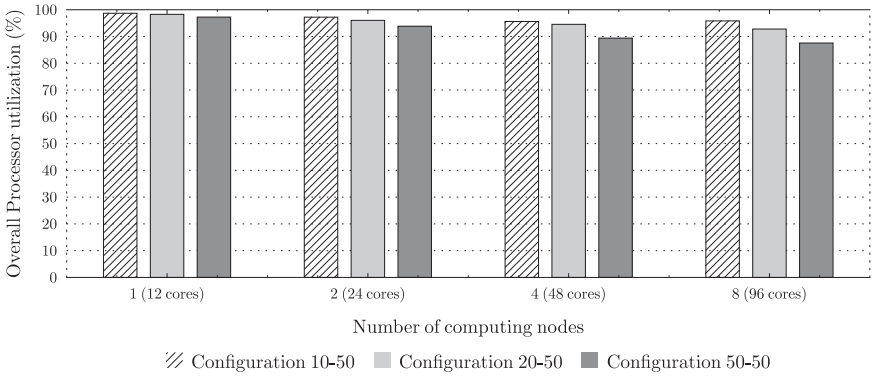


Fig. 10. Overall processor utilization results for a selected test instance with 500 vertices, 3000 arcs and 200 commodities. Each data set corresponds to a different time limit parameter configuration, where the first number refers to the local search time limit and the second refers to the solution recombination time limit.

Table 4
C instance set: performance comparison between commodity assignation schemes.

Problem	LB/Opt	Connection graph	Random assignation
100/400/010/VL	28,423*	28,486	**28,430**
100/400/010/FL	23,949*	**24,022**	**24,022**
100/400/010/FT	63,066	**64,207**	64,492
100/400/030/VT	384,802*	**384,802***	**384,802***
100/400/030/FL	49,018*	**49,018***	49,232
100/400/030/FT	132,129	**136,861**	138,550
20/230/040/VL	423,848*	**424,075**	**424,075**
20/230/040/VT	371,475*	**371,573**	**371,573**
20/230/040/FT	643,036*	**643,036***	**643,036***
20/230/200/VL	94,213*	**94,213***	94,227
20/230/200/FL	137,642*	**137,642***	138,282
20/230/200/VT	97,914*	**97,914***	98,666
20/230/200/FT	135,863	**135,867**	136,241
20/300/040/VL	429,398*	**429,398***	**429,398***
20/300/040/FL	586,077*	**586,077***	590,964
20/300/040/VT	464,509*	**464,509***	464,549
20/300/040/FT	604,198*	**604,198***	**604,198***
20/300/200/VL	74,753	**74,811**	75,220
20/300/200/FL	113,862	**115,580**	116,796
20/300/200/VT	74,991*	**74,991***	75,807
20/300/200/FT	106,672	**107,102**	108,289
30/520/100/VL	53,958*	**53,978**	54,004
30/520/100/FL	93,570	**93,967**	94,409
30/520/100/VT	52,046*	**52,046***	**52,046***
30/520/100/FT	96,260	**97,862**	98,041
30/520/400/VL	112,735	**112,787**	113,346
30/520/400/FL	147,790	**149,677**	150,616
30/520/400/VT	114,641*	**114,641***	114,729
30/520/400/FT	150,685	**154,137**	156,555
30/700/100/VL	47,603*	**47,603***	**47,603***
30/700/100/FL	59,958*	**60,058**	60,390
30/700/100/VT	45,872*	**45,879**	46,040
30/700/100/FT	54,904*	**54,904***	55,059
30/700/400/VL	97,189	**98,090**	99,369
30/700/400/FL	131,690	**136,257**	142,080
30/700/400/VT	94,508	**95,651**	96,708
30/700/400/FT	128,243	**131,104**	131,153
Average GAP		**0.51**	1.08

an effective means of load balance, as we achieve a utilization as high as 98%. The utilization also decreases with higher processor counts due to the requirement of more static partitions. Allowing more time for each local search also has a detrimental effect on processor utilization. This is due to the fact that the processors that end their work prematurely will have increased idle time penalties. Communication time is shown to be a small fraction of the overall compute time, which confirms our algorithm to be compute-bound.

5.5. Partitioning the subproblem sequence

An important component of the parallel algorithm consists in partitioning the work among the parallel processors. In Section 3.2, we introduced the use of a graph partitioning problem to determine the distribution of local searches among the processors. By this transformation, the problem of finding a set of commodity partitions is effectively equivalent to the problem of partitioning a connection graph such that the cut is minimized. Our hope is that, by enforcing tightly connected commodities to be assigned together, better solutions will be achieved.

In order to evaluate the impact of this algorithm phase in the overall solution quality, we compare its performance against a random assignation of commodities. In Table 4, a performance comparison is shown for the C instance problem set. We specify two sets of executions performed under the same conditions and parameters. The time limit was set to be the time to best solution by *ParLS* shown in Table 1.

On the C instance problem set, the random subproblem assignation shows a slower solution improvement. When the same time limit is considered, results are generally worse or equivalent except for one of the smallest instances. On average, it achieves an optimality GAP of 1.08% in comparison to the 0.51% obtained by our original connection graph partitioning. In addition, the optimal solution is only achieved in 6 instances whereas our proposed scheme reaches optimality in 15.

Performance results on the GT instance problem set with a time limit of 1 h are presented in Table 6. The original algorithm with connection graph partitioning obtains better solutions in all instances but two. These are two of the smallest instances in the set, with 50 commodities. The impact on the performance varies substantially from instance to instance, ranging from an almost identical performance (0.1% GAP difference in the F_L,500,2000,50 problem) to a jump of 2.65% in the case of the F_T,500,3000,150 instance.

5.6. The parallel solution recombination

In the last stage of the algorithm, the local search improvements are accumulated into one solution. As described in Section 3.3, a new MIP subproblem is formulated, where the arc variables that are not used in any of the input solutions are fixed to zero. For parallel scalability, the solution recombination process is split into two structured phases.

In Table 5, we compare the effectiveness of our scheme against a single-step recombination of solutions extracted from a number of representative problem instances. For each of the tested

Table 5
Performance comparison of solution recombination schemes.

Problem	Allowed time	Parallel recombination		Single recombination	
		Improvement	Fixed arcs	Improvement	Fixed arcs
F_T,500,2500,50	300	0.19	2303	0.00	1930
F_T,500,2500,50	900	0.20	2250	0.00	1754
F_T,500,2500,50	1800	0.20	2251	0.00	1754
F_T,500,2500,100	300	0.47	2283	0.00	1800
F_T,500,2500,100	900	0.95	2240	0.00	1779
F_T,500,2500,100	1800	0.22	2224	0.00	1711
F_T,500,2500,150	300	0.17	2062	0.00	1311
F_T,500,2500,150	900	0.99	2145	0.00	1566
F_T,500,2500,150	1800	0.35	2117	0.00	1569
F_T,500,2500,200	300	3.16	2181	0.17	1492
F_T,500,2500,200	900	0.33	2177	0.00	1534
F_T,500,2500,200	1800	0.01	2180	0.00	1520

Table 6
GT instance set: performance comparison between commodity assignation schemes.

Problem	LB/Opt	Connection graph	Random assignation
F_T,500,2000,50	4,326,550	**4,892,012**	4,914,193
F_T,500,2000,100	6,368,730	**7,273,916**	7,294,598
F_T,500,2000,150	7,208,800	**8,014,986**	8,155,082
F_T,500,2000,200	8,845,440	**10,617,796**	10,859,566
F_T,500,2500,50	3,927,990	4,406,080	**4,378,135**
F_T,500,2500,100	5,330,490	**6,365,848**	6,474,697
F_T,500,2500,150	5,930,530	**7,037,860**	7,199,470
F_T,500,2500,200	8,327,720	**10,727,261**	10,883,127
F_T,500,3000,50	3,529,370	**4,035,362**	4,056,824
F_T,500,3000,100	5,442,880	**6,634,387**	6,774,373
F_T,500,3000,150	6,236,240	**7,517,445**	7,765,078
F_T,500,3000,200	7,283,080	**9,751,002**	9,997,125
F_L,500,2000,50	3,432,140	**3,722,839**	3,726,822
F_L,500,2000,100	5,497,770	**6,005,177**	6,048,962
F_L,500,2000,150	6,750,150	**7,510,651**	7,587,028
F_L,500,2000,200	8,031,600	**9,338,097**	9,341,089
F_L,500,2500,50	3,176,040	**3,491,664**	3,510,419
F_L,500,2500,100	5,062,110	**5,909,401**	5,935,047
F_L,500,2500,150	6,542,600	**8,138,918**	8,350,529
F_L,500,2500,200	7,717,740	**9,788,913**	9,959,787
F_L,500,3000,50	2,958,630	3,369,303	**3,364,152**
F_L,500,3000,100	4,855,420	**5,773,133**	5,874,379
F_L,500,3000,150	6,031,650	**7,741,294**	7,947,452
F_L,500,3000,200	6,722,660	**9,195,115**	9,527,525
Average GAP		**15.43**	16.53

instances, the best 96 solutions (one per parallel core) found after certain time limit are selected and used as an input for the recombination. Each of the schemes is allowed 100 s of optimization. In the case of the two-step recombination, the time limit is equally distributed with 50 s for each phase.

We report improvements as percentages relative to the objective of the best input solution. It is computed as $I = \frac{BI_{sol} - F_{sol}}{BI_{sol}} \cdot 100$, where BI_{sol} is the best objective value among the 96 input solutions and F_{sol} is the objective value of the final recombined solution. In addition, the number of fixed arcs is also reported. In the case of the two-phase scheme, an average of the arc fixings of all the recombinations is provided.

Through all tested instances, the two-phase scheme proves to be a more effective strategy for recombining a large number of solutions. Improvements are achieved because the input is partitioned, and therefore a high level of fixings can be maintained. That is not the case for a single-phase scheme, where the number of fixings is significantly lower. As a result, improvements cannot be found in the allowed time except for one instance.

6. Conclusions

We propose a scalable parallel approach for the Fixed Charge Multicommodity Network Flow problem that is designed for both shared memory parallel systems and distributed memory systems. By the use of heuristic local searches based on solving restricted MIP subproblems obtained by variable fixings, improvements in the flow routing are found in parallel and are further combined to obtain improved solutions. We rely on the network characteristics of the instances and the given solutions to define core components of the algorithm, such as the work partitioning and the solution recombination mechanism. Computational experiments demonstrate the effectiveness and scalability of our approach, as high-quality solutions are obtained for two problems sets from the literature.

Large sized FCMNF problem instances represent a computational challenge. Commercially available solvers and previous heuristic methods struggle to provide solutions and lower bounds that are within a reasonable optimality gap. It is precisely in the size of these instances where many opportunities to exploit parallelism can be found. We demonstrate the value of parallel computing and heuristic approaches for effectively generating good primal solutions to large FCMNF problem instances. Optimality certificates in the form of lower bounds are still difficult to achieve. In future research, we will seek to solve this shortcoming by applying the notions presented in this paper in the context of an exact algorithm that combines primal and dual aspects.

Acknowledgments

We would like to thank Rodolfo Carvajal for helpful discussions. This research has been supported in part by ExxonMobil Upstream Research Company and the Air Force Office of Scientific Research.

References

[1] Magnanti TL, Wong RT. Network design and transportation planning: models and algorithms. Transp Sci 1984;18(1):1–55.
[2] Ghamlouche I, Crainic TG, Gendreau M. Path relinking, cycle-based neighbourhoods and capacitated multicommodity network design. Ann Oper Res 2004;131(1–4):109–33.
[3] Ghamlouche I, Crainic TG, Gendreau M, Sbeity I. Learning mechanisms and local search heuristics for the fixed charge capacitated multicommodity network design. Int J Comput Sci Issues 2011:5.
[4] Chouman M, Crainic T. A MIP-tabu search hybrid framework for multicommodity capacitated fixed-charge network design. CIRRELT; 2010.
[5] Yaghini M, Momeni M, Sarmadi M. A simplex-based simulated annealing algorithm for node-arc capacitated multicommodity network design. Appl Soft Comput 2012;12(9):2997–3003.
[6] Kleeman MP, Seibert BA, Lamont GB, Hopkinson KM, Graham SR. Solving multicommodity capacitated network design problems using multiobjective evolutionary algorithms. IEEE Trans Evol Comput 2012;16(4):449–71.
[7] Alvarez AM, González-Velarde JL, De-Alba K. Scatter search for network design problem. Ann Oper Res 2005;138(1):159–78.
[8] Crainic TG, Gendreau M. Metaheuristics: progress in complex systems optimization. Boston, MA: Springer; 2007. p. 25–40.
[9] Paraskevopoulos DC, Bektaş T, Crainic TG, Potts CN. A cycle-based evolutionary algorithm for the fixed-charge capacitated multi-commodity network design problem. Eur J Oper Res 2016;253(2):265–79.
[10] Fischetti M, Lodi A. Local branching. Math Program 2003;98(1–3):23–47.
[11] Rodríguez-Martín I, Salazar-González JJ. A local branching heuristic for the capacitated fixed-charge network design problem. Comput Oper Res 2010;37(3):575–81.
[12] Katayama N, Chen MZ, Kubo M. A capacity scaling heuristic for the multicommodity capacitated network design problem. J Comput Appl Math 2009;232(1):90–101.
[13] Katayama N. A combined capacity scaling and local branching approach to capacitated multi-commodity network design problem. Far East J Appl Math 2015;92(1):1.
[14] Hewitt M, Nemhauser GL, Savelsbergh MWP. Combining exact and heuristic approaches for the capacitated fixed-charge network flow problem. INFORMS J Comput 2010;22(2):314–25.
[15] Hewitt M, Nemhauser GL, Savelsbergh MWP. Branch-and-price guided search for integer programs with an application to the multicommodity fixed-charge network flow problem. INFORMS J Comput 2013;25(2):302–16.
[16] Badrinarayanan VA, Furman KC, Goel V, Shao Y, Li G. Parallel large-neighborhood search techniques for lng inventory routing, Optimization online.
[17] Crainic TG, Gendreau M. Cooperative parallel tabu search for capacitated network design. J Heuristics 2002;8(6):601–27.
[18] Crainic TG, Li Y, Toulouse M. A first multilevel cooperative algorithm for capacitated multicommodity network design. Comput Oper Res 2006;33(9):2602–22.
[19] Crainic TG, Toulouse M. Parallel meta-heuristics.Handbook of Metaheuristics, International Series in Operations Research & Management Science, vol. 146. USA: Springer; 2010. p. 497–541.
[20] Berthold T. Primal heuristics for mixed integer programs [Diploma thesis]. Technische Universitat Berlin.
[21] Rothberg E. An evolutionary algorithm for polishing mixed integer programming solutions. INFORMS J Comput 2007;19(4):534–41.
[22] Frangioni A. Multicommodity problems. URL ⟨http://www.di.unipi.it/optimize/Data/MMCF.html⟩; 2013 [Online; accessed 2016-07-05].
[23] Kernighan BW, Lin S. An efficient heuristic procedure for partitioning graphs. Bell Syst Tech J 1970;49(2):291–307.
[24] Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning

irregular graphs. SIAM J Sci Comput 1998;20(1):359–92.
[25] Gropp W, Lusk E, Skjellum A. Using MPI: portable parallel programming with the message-passing interface, vol. 1. Cambridge, Massachusetts: MIT Press; 1999.
[26] Hewitt M. Gt instances. URL ⟨https://www.researchgate.net/publication/304825234⟩; 2010 [Online; accessed 2016-07-05].
[27] P.S.P. Ltd. Passmark software. URL: ⟨http://passmark.com/⟩; 2016 [Online; accessed 2016-07-05].