2014 IEEE Congress on Evolutionary Computation (CEC)
July 6-11, 2014, Beijing, China

A Parallel Lagrangian-ACO Heuristic for Project Scheduling

Oswyn Brent, Dhananjay Thiruvady, Antonio Gómez-Iglesias, Rodolfo García-Flores
CSIRO Computational Informatics
CSIRO
Clayton, Australia
{oswyn.brent, dhananjay.thiruvady, antonio.gomez, rodolfo.garcia-flores}@csiro.au

***Abstract*—In this paper we present a parallel implementation of an existing Lagrangian heuristic for solving a project scheduling problem. The original implementation uses Lagrangian relaxation to generate useful upper bounds and provide guidance towards generating good lower bounds or feasible solutions. These solutions are further improved using Ant Colony Optimisation via loose and tight couplings. While this approach has proven to be effective, there are often large gaps for a number of the problem instances. Thus, we aim to improve the performance of this algorithm through a parallel implementation on a multicore shared memory architecture. However, the original algorithm is inherently sequential and is not trivially parallelisable due to the dependencies between the different components involved. Hence, we propose different approaches to carry out this parallelisation. Computational experiments show that the parallel version produces consistently better results given the same time limits.**

I. Introduction

The increasing popularity of multi-core processors offers an attractive opportunity for improving the results of many existing scientific applications without increasing the time required to achieve those results. Alternatively, the possibility of attaining similar results in reduced time-frames is also appealing. In order for programs to make full use of available hardware, as well as scale into the future, they must be written in a multi-threaded manner. There are several plausible alternatives to achieve such a parallel implementation. In this study, we consider a multicore shared memory approach.

Project scheduling is a topic of great interest, and has been for decades. A typical project consists of a number of tasks and the aim is to optimise an objective related to the completion time and value of the tasks. More recently there has also been interest in maximising the net present value (NPV) of a project, which is the sum of the cash flows of all completed tasks, discounted based on the time of completion [4], [5], [6]. NPV is an important financial concept, and represents the time adjusted value of a project. The higher the NPV of a project, the more profitable it is to the company.

In constraint optimisation, metaheuristics have been widely used ([1]). However, when these techniques are combined with other methods like Lagrangian relaxation (LR), parallel approaches have not received much attention. For example, in [2], the authors use a hybrid Lagrangian - ant colony heuristic (AL-MMKP) to solve the multiple-choice multidimensional knapsack problem, though they only consider parallelising the Ant Colony Optimisation (ACO) component to improve run times. There have also been studies with parallel ACO [18] and parallel genetic algorithms [19] for similar resource constrained problems. However, to the best of our knowledge there has been no prior work on parallelising such a hybrid algorithm for project scheduling. Thus, in this study, we explore a parallel implementation of a hybrid LR - ACO implementation to efficiently solve the well-known project scheduling problem ([3]). We focus on the parallelisation of an existing implementation that has already proven to achieve excellent results for this problem.

Lagrangian relaxation (LR) is a well-known technique applied to integer programming problems ([7]). Many computationally hard problems can be tackled by considering a simpler version of the problem which relaxes one or more sets of complicating constraints. The solution to the relaxed problem can provide useful information related to the original problem, in particular, upper bounds (for maximisation problems) which provide performance guarantees. Additionally, lower bounds may also be identified by repairing the relaxed solutions. Kimms [3] showed how such a scheme could be successful in the context of project scheduling.

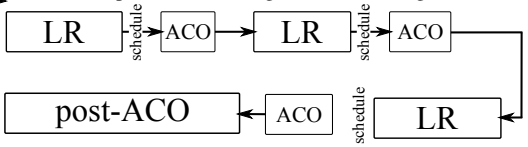
ACO is a metaheuristic based on the foraging behaviour of ants which has been successfully applied to various combinatorial optimisation problems [8]. Furthermore, it has proved to be effective on project scheduling problems [9], [4], [6]. However, ACO and more generally metaheuristics struggle in the presence of hard non-trivial constraints and unlike LR, ACO focuses on building feasible solutions to the original problem. Thus by combining the relative strengths of each of these algorithms, [10] showed that such a hybrid approach is effective for project scheduling.

In the current study, we parallelise the hybrid LR and ACO (LR-ACO) heuristic. While LR-ACO is more effective than previous methods, its data dependencies make it difficult to parallelise. Each iteration of LR depends on the previous, and each iteration of ACO depends on the LR (see Fig. 1) so OpenMP directives cannot be trivially used. Many small sections within the algorithm have no data dependencies could be easily parallelised. However, they comprise a small proportion of the overall computation, so parallelising them leads to minimal improvement in the overall performance of the algorithm. In addition to this, the large number of small components is unattractive to parallelise and a higher level approach is preferred [11].

The rest of this paper is organised as follows. The RCP problem is briefly stated in Section II. LR-ACO and its components are presented in Section III while in Section IV we detail our parallelisation of the algorithm (Par-LR-ACO). Experimental results and analysis are provided in Section

978-1-4799-1488-3/14/$31.00 ©2014 IEEE

Fig. 1. Execution pattern of the original LR-ACO algorithm



V. Finally, Section VI concludes the paper and provides a variety of perspectives.

II. Problem Formulation

The RCP scheduling problem can be formally stated as follows. There are a number of tasks, each with durations and associated cash flows. Each cash flow is discounted based on some discount rate, α and time of cash flow. The constraints to be satisfied include task precedences, resource limitations and a global deadline. We list notation and formulae used.

T	The set of tasks to be scheduled.
d_i	The duration of task i.
cf_{it}	The cash flow of task i in period t.
c_i	The total cash flow of task i. $c_i = \sum_{t=1}^{d_i} cf_{it} e^{\alpha(d_i - t)}$
s_i	The start time of task i.
P	The set of precedences. If a precedence exists between task i and j, $(i,j) \in P$
R	The set of resources. For k resources, $R = \{R_1, \ldots, R_k\}$.
r_{ik}	The amount of resource k required by task i.
δ	The deadline that all tasks must be completed by.
NPV	The summed net present value of all tasks. $NPV = \sum_{i=1}^{n} c_i e^{\alpha(s_i + d_i)}$

The objective is to maximise the NPV as defined as follows.

$$max. \quad \sum_{i=1}^{n} c_i e^{\alpha(s_i + d_i)} \tag{1}$$

$$s.t. \quad s_i + d_i \leq s_j \qquad \forall (i,j) \in P \tag{2}$$

$$\sum_{i \in S(t)} r_{ik} \leq R_k \qquad k = \{1, \ldots, m\}, \quad t = \{1, \ldots, \delta\} \tag{3}$$

$$s_i + d_i \leq \delta \qquad \forall i \in T \tag{4}$$

where $S(t)$ is the set of tasks executing at time t. Constraint (2) specifies all tasks must start after their predecessors have completed. Constraint (3) requires that all the resources are satisfied and Constraint 4 requires that all the deadlines are satisfied.

Alternatively, a solution can be represented by a permutation of tasks π which can be mapped to schedule [10]. This schedule can be made feasible by ensuring all the constraints are satisfied. The only potential issue is the deadline which may not be satisfied if the horizon is small. However, like [3], we ensure the deadlines are large enough to always generate deadline-feasible solutions.

III. The Hybrid LR-ACO Algorithm

LR-ACO uses an adaptation of the Lagrangian model used by [12] and an ACO model suggested in [13].

A. Lagrangian heuristic

For the sake of completeness, we briefly provide the integer programming (IP) model used by [12] and describe the associated Lagrangian relaxation. Let x_{jt} be binary variables such that $x_{jt} = 1$ if task j completes by time t and 0 otherwise. The model is specified as follows:

$$max. \quad \sum_{i=1}^{n} \sum_{t=2}^{\delta} c_i e^{-\alpha t} (x_{it} - x_{it-1}) \tag{5}$$

$$s.t. \quad x_{it} \geq x_{it-1} \quad \forall i \in T, t \in \{1, \ldots, \delta\} \tag{6}$$

$$x_{i\delta} = 1 \quad \forall i \in T \tag{7}$$

$$x_{jt} \leq x_{it-d_j} \quad \forall (i,j) \in P, t \in \{1, \ldots, \delta\} \tag{8}$$

$$\sum_{i=1}^{n} r_{it} (x_{it} - x_{it-d_i}) \leq R_{kt} \quad \forall k \in R, t \in \{1, \ldots, \delta\} \tag{9}$$

$$x_{it} \in \{0,1\} \quad \forall i \in T, t \in \{1, \ldots, \delta\} \tag{10}$$

Constraint (6) enforces that a task stays completed once it has finished and Constraint (7) requires that all tasks complete. The precedences are enforced via Constraint (8) and the resource constraints are specified by Constraint (9).

The Lagrangian relaxation by relaxing the resource constraints and introducing multipliers $\lambda_{kt}, k \in R, t \in \{1, \ldots, \delta\}$ [3]. The upper bound is then obtained by solving the Lagrangian dual defined by:

$$LRR(\lambda) = max. \sum_{i=1}^{n} \sum_{t=2}^{\delta} c_i e^{-\alpha t} (x_{it} - x_{it-1}) + \sum_{k \in R} \sum_{t=1}^{\delta} \lambda_{kt} \left(R_{kt} - \sum_{i=1}^{n} \sum_{\hat{t}=t}^{t+d_j-1} r_{i\hat{t}} (x_{it} - x_{it-d_i}) \right) \tag{11}$$

Terms and symbols used in the LR algorithm are as follows:

π	A permutation of tasks, as described in Section II.
π^{bs}	The best permutation found so far.
LB	Current lower bound.
UB	Current upper bound.
ST	A set of start times for tasks.
λ	The Lagrange multipliers over all time periods.
γ	A scaling factor for the Lagrange multiplier.
gap	The proportional gap between the upper and lower bounds.

Alg. 1 shows pseudocode for the Lagrangian Relaxation algorithm used.

Algorithm 1: Lagrangian Relaxation

Data: RCP instance
Result: π^{bs}
Initialise $\pi^{bs} \leftarrow NULL$; $\lambda, i \leftarrow 0$; $\gamma \leftarrow 2$;
$LB \leftarrow -\infty$; $UB, gap \leftarrow \infty$;
while $\gamma, gap > 0.01$ **do**
  $ST \leftarrow$ Solve(λ, UB);
  $\pi \leftarrow$ GenerateList(ST);
  ImproveLB(π);
  UpdateBest(π^{bs}, π, γ);
  $LB =$ NPV(π^{bs});
  UpdateMult$(\lambda, LB^*, UB, ST, \gamma)$;
  $gap \leftarrow |\frac{UB-LB}{UB}|$;
  $i \leftarrow i + 1$;
ImproveLB(π);

a) Solve(λ^i, UB)*:* Solves the relaxed problem using Eq. 11, sets UB is set to LLR(λ^i) and returns an optimal relaxed solution ST.

b) GenerateList(ST)*:* Returns a complete list of tasks, π by transforming ST.

c) ImproveLB(π)*:* Maps π to a feasible schedule, providing a lower bound, LB. This may be improved further with an alternative method. The hybrid method is obtained by using ACO here. An extended ImproveLB is called after the main loop to help further improve the lower bound.

d) UpdateBest(π^{bs}, π, γ)*:* Updates π^{bs} with π if NPV(π) > NPV(π^{bs}). Additionally if π^{bs} has not been updated in the last five iterations, $\gamma := \gamma \div 2$.

e) UpdateMult$(\lambda, LB, UB, ST, \gamma)$*:* The multipliers are updated for all time periods $t \in \{1, \ldots, \delta\}$

B. Ant Colony Optimization

ACO was first suggested by M. Dorigo [14] for combinatorial optimisation and is based on the behaviour of foraging ants. When an ant has found food it leaves a pheromone path on its way back to the nest. Other ants looking for food will choose between these trails based on the amount of pheromones on the paths, in turn depositing their own pheromones, creating a positive feedback loop. Once a food source has been exhausted, returning ants no longer deposit pheromones and the original trails evaporate, allowing the colony to find better food sources over time [15].

In the context of the RCP problem, we consider a pheromone model that is based on learning an ideal permutation of the tasks. Here, the aim is to learn permutations of the tasks which are then mapped using a scheduling scheme. The ACO used is based on the model suggested in [13] where the authors examine an ACO algorithm for the single machine problem with the total weight tardiness objective. The pheromones $\mathcal{T}$ consist of the pheromone values τ_{ij} for each task j and variable i or position in the sequence. The variant of ACO used is the ant colony system (ACS) [16].

Terms and symbols additionally used in the ACO algorithm (shown in Alg. 2) below are as follows:

$\mathcal{T}$	A pheromone matrix.
n_a	The number of ants per iteration.
S_{iter}	The set of solutions generated in an iteration.
π^{ib}	The iteration best solution.

Algorithm 2: Pseudocode of the ACS algorithm

Data: Pheromone matrix $\mathcal{T}$
Result: π^{bs}
Initialise $\pi^{bs} = NULL$;
while *termination conditions not satisfied* **do**
  $S_{iter} = \emptyset$;
  for $j \leftarrow 1$ **to** n_a **do**
    $\pi_j =$ ConstructSolution$(\mathcal{T})$;
    ScheduleTasks(π_j);
    $S_{iter} = S_{iter} \cup \{\pi_j\}$;
  $\pi^{ib} =$ argmin$\{f(\pi) | \pi \in S_{iter}\}$;
  Update(π^{ib}, π^{bs});
  PheromoneUpdate$(\mathcal{T}, \pi^{bs})$;
  $cf =$ ComputeConvergence(π^{ib});
  if $cf = true$ **then**
    initialise $\mathcal{T}$;

a) ConstructSolution$(\mathcal{T})$*:* A permutation π of tasks is constructed by selecting a task for each variable starting at π_1. A random number $q \in (0, 1]$ is generated and compared with a pre-defined parameter q_0 in order to select a task at π_i. If $q < q_0$, k is deterministically selected. Otherwise, k is probabilistically selected based on the pheromone matrix and heuristic information [10] When a task j at variable i is selected, the pheromones at τ_{ij} are updated.

b) ScheduleTasks(π)*:* Determines a schedule σ from the given permutation of tasks. The schedule is always resource feasible.

c) Update(π^{ib}, π^{bs})*:* Sets π^{bs} to π^{ib} if NPV$(\sigma(\pi^{ib}))$ >NPV$(\sigma(\pi^{bs}))$.

d) PheromoneUpdate$(\mathcal{T}, \pi^{bs})$*:* $\mathcal{T}$ is scaled down (evaporation) then is updated for each (i, j) entry in π^{bs} with a constant reward amount.

e) ComputeConvergence(π^{ib})*:* If sampling the pheromones repeatedly produces the same solution, the pheromones are considered to have converged. A list of the past θ solutions is maintained. If all θ solutions have the same objective value, the pheromones are re-initialised and normalised. The list is then cleared.

C. Hybridisation

ACO is incorporated in a straightforward manner into the LR algorithm. In the LR algorithm, *ImproveLB*(π) can be replaced with the algorithm previously described in Alg. 2. The core idea is to seed ACO with (π) to improve the lower bound. π is used as the global best solution for ACO, which biases the search towards this solution. Likewise the final *ImproveLB* is replaced with an extended call to ACO, which we call post-ACO. This post-ACO uses the converged pheromone matrix information from LR to further improve lower bounds [10].

IV. Parallelisation and Observations

As previously stated, only certain functionalities with enough workload are suitable for an efficient parallelisation. A traditional and natural approach is to parallelise the ants in the ACO algorithm, since they can work almost independently from each other and the effort required to parallelise this model is usually low. For RCP, ACO is more effective at improving solutions derived from LR than producing good solutions standalone [10]. A direct implication from this is that maximising the number of LR iterations run is central to improving the speed of convergence of LR-ACO.

A. Parallelising Ants

We considered the implementation with parallel ants first. The reason we chose this experiment first is that parallelising ants in LR-ACO has been previously done [2], requires minimal code modification, and can potentially provide good speedup. We considered two approaches. Either we would run multiple ants concurrently and reduce running time or run k times as many ants in parallel, for k threads, and try to improve the result for the same run time. We ran several tests with differing numbers of ACO solutions generated per LR iteration. We found that while a higher solution count greatly improved lower bounds in early iterations, the difference was minimal in later iterations. We saw that the increase in runtime per iteration with higher solution counts led to a slower convergence speed overall. Due to this we kept the number of ants constant, and ran them across multiple threads.

We parallelised the main loop in the ACS algorithm (Alg. 2) with an OpenMP **parallel for** directive and added critical sections for the update functions, since all the ants share the same pheromone matrix and best ant. OpenMP has **critical** constructs which provide a means to ensure multiple threads do not update shared data simultaneously [11]. This approach can be easily replicated by other technologies. We gained sub-linear speedup in the ants alone due to the proportion of work spent in the main loop and the rest of the ACO algorithm. In addition to this, we found that especially in earlier iterations, much of the run time was spent on the Lagrangian, with up to 140% thread utilisation on 12 threads, far below the optimal 1200% (a 12 cores machine was used for this test). This led us to focus on running LR and ACO concurrently.

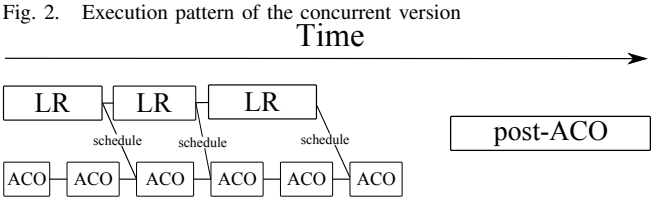
B. Concurrent LR-ACO

Our general approach was to restructure LR-ACO as follows:

1) Run a single LR iteration to produce a π to seed ACO.
2) Use OpenMP tasks to run LR on one thread and ACO on another.
3) Remove ImproveLB() from LR and instead allow the ACO thread to run on whatever π is available continually.

The resulting model can be seen in Fig. 2. The algorithm is no longer sequential as previously shown in Fig. 1, and communication between LR and ACO is no longer serial. It can be seen how in the original model after every single instance of LR, there was a call to the ACO algorithm. Now, in the parallel model, since the execution time of the Lagrangian Relaxation method is longer than ACO, several instances of the ACO algorithm can run during a single execution of each LR.

Fig. 2. Execution pattern of the concurrent version



To achieve this we used **parallel**, **single** and **task** OpenMP pragmas [11]. The **parallel** pragma defines a section to be run in parallel. When a parallel section is encountered, all available threads execute the section. The parallel pragma can take a parameter list of memory sharing patterns. We use the **shared** parameter to share the pheromone matrix between ants. The **task** pragma defines a section as a task. When a task is encountered OpenMP assigns a thread to execute it concurrently. The **single** pragma must be placed within a parallel section, and ensures that only one thread executes the following section.

The resulting code is as shown in Alg. 3.

Algorithm 3: Concurrent LR-ACO

```
1  Initialise π;
2  LR(π);
3  #pragma omp parallel shared(p)
4  {
5    #pragma omp single
6    {
7      #pragma omp task
8      {
9        while !finished do
10         LR(π);
11     }
12     #pragma omp task
13     {
14       while !finished do
15         ACO(π);
16     }
17   }
18   postACO(π);
19 }
```

This led to more LR iterations for the same run time, producing better upper bounds. These better upper bounds, subsequently led to better lower bounds, as the ACO has better targets.

We identified that this approach was not inherently scalable, as the iterative nature of LR formed a bottleneck, and we could not further increase the speed of LR through parallelisation. We saw earlier that parallelising ants was ineffective, so we decided to run multiple colonies in parallel, to attempt to improve results further.

C. Parallel Colonies

Since now LR and ACO were not sequentially run, reducing ACO runtime is less important. By parallelising at a higher level, each thread does more work and so overheads are minimised. We extended our task framework to now run multiple ant colonies parallel, in an attempt to improve the probability of at least one colony finding a better lower bound in a given iteration. This has been omitted from the code structure for clarity. The resulting code can be seen in Alg. 4.

Algorithm 4: Concurrent LR-ACO with parallel colonies

```
1  Initialise π;
2  int numThreads = omp_get_num_threads();
3  int numAco = numThreads - 1;
4  LR(π);
5  #pragma omp parallel shared(p)
6  {
7    #pragma omp single
8    {
9      #pragma omp task
10     {
11       while !finished do
12         LR(π);
13     }
14     for i = 0 to numAco do
15       #pragma omp task
16       {
17         while !finished do
18           ACO(π);
19       }
20     for i = 0 to numThreads do
21       postACO(π);
22   }
23 }
```

In addition to running ACO on all but one of the threads in the tasks, we ran a separate colony on each thread for post-ACO as well (including the thread originally running LR). A barrier pragma is used to ensure all the LR-ACO tasks have finished before postACO starts.

V. Experiments and Results

We ran two sets of experiments and compared the fully parallelised version in Section IV against the original sequential version. Experiment (a) consisted of 1080 instances from PSPLIB[1] with job counts of 60 and 120 each and experiment (b) consisted of a subset of 24 instances, chosen for specific measures and run 30 times each. This subset was used with the cash flows from [10] to ensure that results produced were in the same range.

Since parallelisation can introduce added overheads, we ran each of the subset instances 10 times on each of the parallel versions. The fully parallelised version performed better than the other two versions on average, and the gaps normalised against the parallel ants version is shown in Fig. 3.

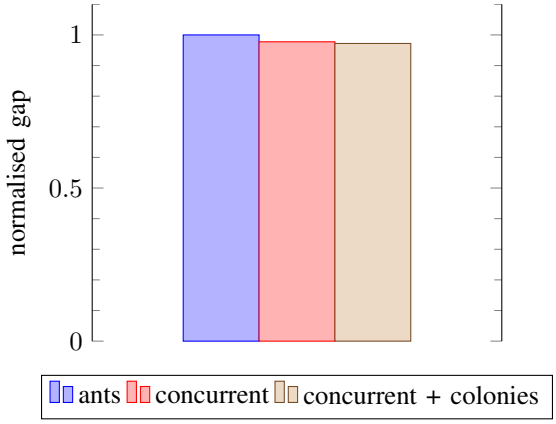


Fig. 3. Normalised gaps for the three parallel versions

For the subset 3 instances per measure and job count combination were used and the measure are found in Table I. The instances differ in network complexity (NC), resource factor (RF), resource strength (RS), and number of tasks. Network complexity indicates the proportion of precedences incorporated in the instance, with a larger value implying a larger number of precedences. Resource factor specifies how many resources are required by an activity in proportion to the total number of resources available. Resource strength measures the scarceness of resources with low values, implying that resources constraints are tight.

Instance	NC	RF	RS	Tasks
37	2.1	0.5	0.2	60
38	2.1	0.5	0.5	60
45	2.1	1.0	0.2	60
46	2.1	1.0	0.5	60
47	2.1	0.5	0.2	120
50	2.1	0.5	0.5	120
57	2.1	1.0	0.2	120
60	2.1	1.0	0.5	120

TABLE I. Parameters for subset instances

All runs were given a maximum of 5 minutes of computing time for the main LR-ACO algorithm and a flat 5 minutes for post-ACO optimisation. The subset experiments were run on a Xeon X7350 2.93Ghz. The full set experiments were run on a Xeon L7545 1.866Ghz. All experiments were run using the maximum number of compute threads available, 4 and 12 respectively. We used the same running parameters as [10] and discount rate $\alpha = 0.01$.

We provide an overview comparison of results in Table II. Listed here, are the absolute values for gap, the average lower and upper bounds over all instances, the average of the best lower and upper bounds per instance, the average number of LR iterations run over all instances and the proportional percentage improvement of Par-LR-ACO over LR-ACO. This overall result shows that the parallel method provides improved gaps across all the runs. Specifically, we see improvements in both lower and upper bounds implying that both the LR and ACO components are more effective in this setting.

[1] http://www.om-db.wi.tum.de/psplib/main.html

Result	Par-LR-ACO	LR-ACO	%Δ
Subset			
gap	0.113305	0.118497	4.582
lb-average	13912.185	13876.512	0.257
lb-best-av	14016.875	13980.864	0.258
ub-average	15703.226	15766.809	0.405
ub-best-av	15644.543	15686.511	0.268
# Iterations	41.536	31.171	33.253
Full Run			
gap	0.085759	0.088608	3.322
lb-average	15239.187	15203.957	0.232
ub-average	16653.345	16673.857	0.123
# Iterations	75.588	56.861	32.934

TABLE II. OVERVIEW COMPARISON BETWEEN PAR-LR-ACO AND LR-ACO. NOTE THAT SINCE EACH INSTANCE IN THE FULL RUN WAS ONLY RUN ONCE, THERE IS NO BEST RUN PER INSTANCE.

Figure 4 shows the average progress of serial and parallel implementations over-time for the subset. We see that even early on that the parallel implementation has a significant advantage but by about 125 seconds the two algorithms have converged. The parallel implementation again improves after about 175 seconds suggesting that with increased iterations the LR component provides better upper bounds.

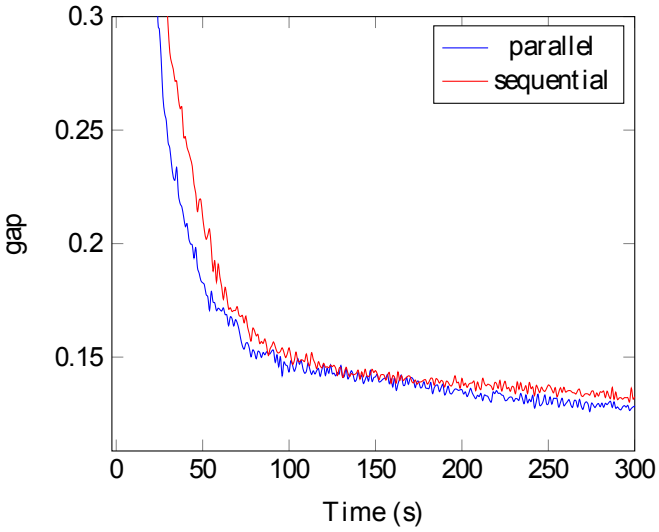


Fig. 4. Gap over time for the subset run

Now we consider the results of experiment (a), and in particular the instances with 120 jobs since they do not necessarily converge in the allowed time-frame. Figures 5, 6, 7 show the average results for instances with 120 jobs, split by resource strengths, resource factors and network complexities. We see the same pattern here as the overall result shown previously. This shows that over varying problem characteristics, the parallel implementation is always useful.

Par-LR-ACO always performs better, providing smaller average gaps than the sequential version across all problem characteristics. Looking more closely, the parallel version provides increasing improvements with decreasing resource strengths and increasing resource factor. This means that for more tightly constrained problems we see greater improvements. This is not surprising since the parallel version runs significantly more LR iterations than the sequential, and LR is designed to deal with constraints effectively. So when conditions favour LR, the difference in gap between parallel and sequential is accentuated.

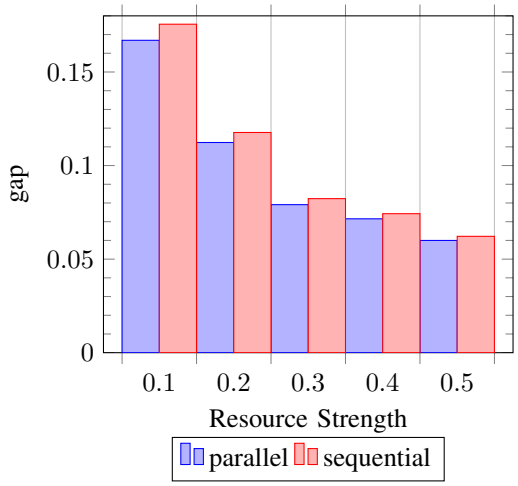


Fig. 5. Average results for all problem instances with 120 jobs, split by resource strength

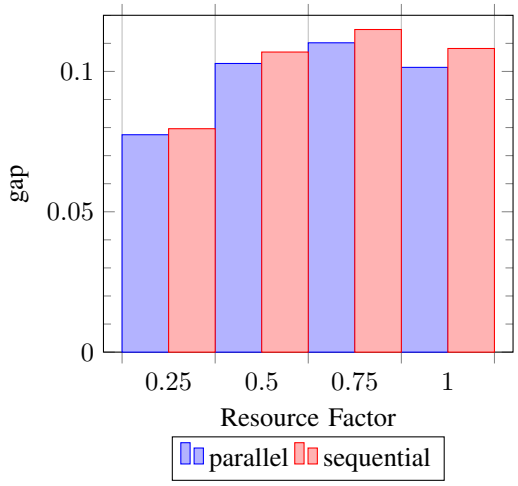


Fig. 6. Average results for all problem instances with 120 jobs, split by resource factor

VI. CONCLUSIONS

In this study, we investigate a parallel implementation of a hybrid LR-ACO heuristic for project scheduling. Our results show that such a parallelisation can be effective, outperforming the serial version in the same time-frame. Furthermore, we see increasing gains relative to the constraints being relaxed in the LR component of the algorithm.

LR-based algorithms, being iterative in nature, rely on a large number of iterations to converge to good solutions. Thus, the main reason for the improvement seen here is attributable to the increase in the number of LR iterations. Specifically, we see up to 33% increase in the same time-frame across all instances. This results in parallel LR-ACO producing a significantly smaller gap between lower and upper bounds, with 3-5% smaller gap on average. In addition, the parallel version converged more quickly.

There are several future directions related to this study. Firstly, Par-LR-ACO could be applied to other problems

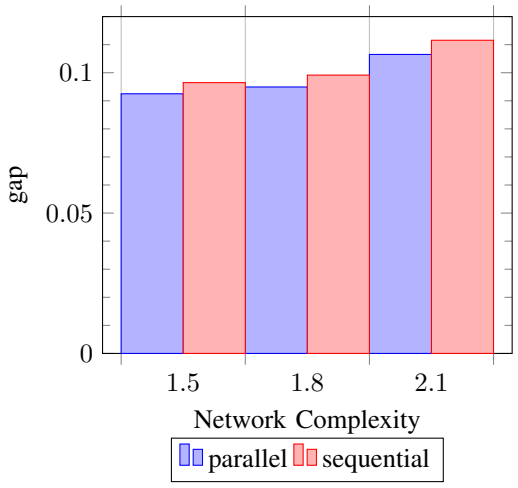


Fig. 7. Average results for all problem instances with 120 jobs, split by network complexity

which are effectively solved using hybrid Lagrangian-ACO. For example, the multiple-choice multidimensional knapsack problem [2] could benefit from such a parallel implementation. In general, where LR and ACO are effective, such a parallel implementation could be designed to provide improvements over its serial counterpart.

Here, we have considered a multicore shared memory architecture. Additionally, we see the potential for parallel LR-ACO with up to 12 cores in this study. However, different architectures with many more cores/processors could potentially provide more significant improvements. For example, with GPUs and Intel's Xeon Phi [17], a heterogenous parallelisation could produce even better performance. To fully leverage the available hardware, a heterogenous parallelisation could involve running LR on the host CPU in parallel with ACO (or other heuristic) on the co-processor could as well as managing memory sharing between the two processors. We plan to investigate this in the near future.

REFERENCES

[1] C. Blum, A. Roli, "Metaheuristics in Combinatorial Optimisation: Overview and Conceptual Comparison", *ACM Computing Surveys*, vol. 35, pp. 268–308, 2003

[2] Z. Ren, Z. Feng, and A. Zhang, "Fusing ant colony optimization with lagrangian relaxation for the multiple-choice multidimensional knapsack problem," *Inf. Sci.*, vol. 182, no. 1, pp. 15–29, Jan. 2012.

[3] A. Kimms, "Maximizing the net present value of a project under resource constraints using a lagrangian relaxation based heuristic with tight upper bounds," *Annals of Operations Research*, vol. 102, no. 1-4, pp. 221–236, 2001.

[4] W. Chen, J. Zhang, H. S. Chung, R. Huang, and O. Liu, "Optimizing discounted cash flows in project scheduling: An ant colony optimization approach," *Trans. Sys. Man Cyber Part C*, vol. 40, no. 1, pp. 64–77, Jan. 2010.

[5] M. Vanhoucke, "A scatter search heuristic for maximising the net present value of a resource-constrained project with fixed activity cash flows," *International Journal of Production Research*, vol. 48, no. 7, pp. 1983–2001, 2010.

[6] Y. Shou, "Ant colony algorithm for scheduling resource constrained projects with discounted cash flows," in *Machine Learning and Cybernetics, 2006 International Conference on*, 2006, pp. 176–180.

[7] M. Fisher, "The Lagrangian Relaxation Method for Solving Integer Programming Problems", *Management Science*, vol. 50, no. 12, pp. 1861–1871, INFORMS, 2004.

[8] M. Dorigo and T. Stützle, *Ant Colony Optimization*. Scituate, MA, USA: Bradford Company, 2004.

[9] D. Merkle, M. Middendorf, and H. Schmeck, "Ant colony optimization for resource-constrained project scheduling," in *IEEE Transactions on Evolutionary Computation*. Morgan Kaufmann, 2000, pp. 893–900.

[10] D. Thiruvady, M. Wallace, H. Gu, and A. Schutt, "A lagrangian relaxation and aco hybrid for resource constrained project scheduling with discounted cash flows," *Journal of Heuristics*, p. To be published, 2014.

[11] B. Chapman, G. Jost, and R. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[12] G. Singh and A. T. Ernst, "Resource constraint scheduling with a fractional shared resource," *Operations Research Letters*, vol. 39, no. 5, pp. 363 – 368, 2011. [Online].

[13] M. Besten, T. Sttzle, and M. Dorigo, "Ant colony optimization for the total weighted tardiness problem," in *Parallel Problem Solving from Nature PPSN VI*, ser. Lecture Notes in Computer Science, M. Schoenauer, K. Deb, G. Rudolph, X. Yao, E. Lutton, J. Merelo, and H.-P. Schwefel, Eds. Springer Berlin Heidelberg, 2000, vol. 1917, pp. 611–620.

[14] M. Dorigo, *Optimization, learning and natural algorithms*. Ph. D. Thesis, Politecnico di Milano, Italy, 1992.

[15] S. Camazine, N. R. Franks, J. Sneyd, E. Bonabeau, J.-L. Deneubourg, and G. Theraula, *Self-Organization in Biological Systems*. Princeton, NJ, USA: Princeton University Press, 2001.

[16] M. Dorigo and L. M. Gambardella, "Ant colony system: A cooperative learning approach to the traveling salesman problem," *Trans. Evol. Comp*, vol. 1, no. 1, pp. 53–66, Apr. 1997.

[17] J. Jeffers and J. Reinders, *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kauffman, 2013.

[18] D. Thiruvady, A. T. Ernst and G. Singh, "Parallel Ant Colony Optimization for Resource Constrained Job Scheduling," *Annals of Operation Research*, Springer US, pp. 1–18, 2014.

[19] U. Kohlmorgen and H. Schmeck and K. Haase, "Experiences with Fine-grained Parallel Genetic Algorithms," *Annals of Operation Research*, vol. 90, no. 0, pp. 203-219.