computing@tanet.edu.te.ua
www.tanet.edu.te.ua/computing

ISSN 1727-6209
International Scientific
Journal of Computing

MULTI-LEVEL PARALLEL FRAMEWORK

Mitică Craus [1], Laurențiu Rudeanu [2]

Department of Computer Science and Engineering
Technical University "Gh. Asachi"
700050 Iași, Romania
[1] craus@cs.tuiasi.ro [2] lrudeanu@yahoo.com

Abstract: *This article deals with the pyramidal framework designed to be used in the parallelization of the ant-like algorithms. Such algorithms have several things in common: they run in cycles and the process can be divided among different "processing units". The parallel implementation of the Ant Colony Optimization algorithm for the Traveling Salesman Problem is an application of this system. The topology of the framework architecture is similar to a B-tree and contains three types of processing nodes: a single master (the root), several sub-masters corresponding to the internal nodes of the tree and several slaves as leaves. First the master reads the problem instance, wraps it up in a message that is sent to all the other processing nodes and initializes the central data structures. Then, the slaves take over the control by starting the algorithm while the master and the sub-masters are waiting for requests to update the data. The framework has an object-oriented design and was implemented in C++, using the MPI library.*

Keywords: *Parallel Algorithms, Framework, Message Passing Interface. Traveling Salesman Problem, Ant Colony*

1. INTRODUCTION

The idea behind the system we are describing is to have a re-usable framework for running several sequential algorithms in a parallel environment. The starting point was the need to study the behavior of some ant colony algorithms [1], i.e. to observe the relevance of certain conditions, parameter values, and to test new ideas. Because we were mainly concerned with ant colonies we had in mind the previous work/attempts of parallelization [2,3,4,5,6] with their advantages and shortcomings. Our intention was to choose a model of parallelization which would best suit the sequential ant algorithm and to overcome - to some extent - the main drawbacks of existing implementations for that model. The central problem was the communication overhead, which for big instances dramatically affects the performance, namely the speed-up.

After this we realized that the design could be easily extended in such a way that it can also be applied to other algorithms, not only to ant colonies (ant-like algorithms). The result of the analysis is a parallel framework that is flexible enough to be configured to any suitable user-provided "external" algorithm.

The algorithms with which the framework can be used have some things in common: they have to run in cycles and it should be possible to divide their work among several "processing units. For example, genetic algorithms are suitable for being used with the framework.

The paper is organized in the following way: First we state the goals of the framework with respect to running algorithms in parallel. Then, the first attempt to parallelize the Ant Colony Optimization algorithm for Traveling Salesman Problem is described. This attempt represents the use case from which the first (two-level) parallel framework has emerged. The choices made concerning this first version of the framework are motivated, and some of the issues observed in experiments are also revealed, thus showing the need of improvement. The second version of the framework, with three levels, which will be described in following sections, tries to address challenge these shortcomings.

2. GOALS

The two main aims of the article are: to create a comfortable level of abstraction and to optimize communication. The former means that the framework should allow the programmer to replace one algorithm with another with a minimum of effort. That would allow us to try out many different implementations with little effort. In order to achieve this first goal class design and application architecture (which will be detailed in the nest section) have to be dealt with: the actual algorithm

to be parallelized would inherit from a generic class for algorithms and the problem-specific tools and data structures would have to match that specific algorithm.

Achieving the second goal would result in acceptable speedups even for larger problem instances.

3. ANT ALGORITHM FOR TSP AND THE TWO-LEVEL PARALLEL FRAMEWORK

In this section the first version of the parallel framework and the problem it has to solve will be described.

3.1. ANT ALGORITHM FOR TSP

As we have mentioned earlier, the algorithm we have chosen to parallelize using the designed framework is the Ant Colony Optimization (ACO) algorithm for the Traveling Salesman Problem (TSP).

TSP is the classic problem of finding the shortest circuit through a set of n cities, visiting each city of the tour exactly once. A symmetric TSP can be represented by a complete weighted graph *G* with *n* nodes, the weights standing for the distances between the cities. The Euclidean version of the TSP defines the cities as points in a plane and weights the edges with the Euclidean distances between the corresponding cities. The resulting graph is complete. TSP is known to be a NP-hard combinatorial problem. The Ant Colony Optimization (ACO) is a new meta-heuristic that has extends the Ant System algorithm that was first applied to TSP [1]. The Ant System and ACO are inspired from the behavior of real ant colonies in nature and their ability to find the shortest path between the food source and the nest. Here is a short description of how the ant colony algorithms for TSP work.

Initially a number of ants are randomly positioned among the nodes. The ants move from one node to another following a state transition rule, until each ant has completed a hamiltonian tour. During a cycle each ants visits each city (node) exactly once. When moving from one node to another, the ants lay pheromone trails on the edges, as shown in Fig. 1. These pheromone trails act as a form of indirect communication among ants (called *stigmergy*) because they attract other ants thus generating a positive feedback called *autocatalytic effect* [1].

When every ant has completed its tour we say that a *cycle* has ended. The intensity of pheromones trails on the edges that the ants used in their tours are updated as it will be explained below. The pheromones on the edges of the best tour are strengthened once more according to a *global updating rule*. Before the next cycle begins a small percent of the pheromones on all graph edges is evaporated to encourage the ants to search for new paths rather than to exploit the ones they already know. After this operation is completed the ants can start the next cycle from the nodes where they the ended the previous cycle. After a predefined number of ant cycles (or when a stopping condition becomes valid) best result among the ants qualifies as the optimal solution.

The basic idea explained above will be explained in a more formal way in the remaining part of this section.

Let $\tau_{i,j}(t)$ be the intensity of the pheromone trail on edge *(i, j)* at time *t* and let $b_i(t)$ be the number of ants in city *i* at time *t, i=1,n;* then $m = \sum_{i=1}^{n} b_i(t)$ is the total number of ants.

The ant movement from the current node to the next is governed by the *state transition rule*: for every unvisited neighbor of the current node a probability of migration is computed. For an ant *k* which at time *t* is in node *i* the probability of the ant to migrate to node *j* at time *t+1* is defined in (1). The choice of the node to use as destination for the ant move is made using a "wheel of fortune" probabilistic mechanism which uses the probabilities that we've explained above.

$$P_{ij}^{k}(t) = \begin{cases} \dfrac{[\tau_{ij}(t)]^{\alpha}[\eta_{ij}]^{\beta}}{\sum_{l \in allowed(k)} [\tau_{il}(t)]^{\alpha}[\eta_{il}]^{\beta}}, & j \in allowed(k) \\ 0, & otherwise \end{cases} \quad (1)$$

- $allowed_k(t)$ is the set of cities not visited by ant k at time t.
- η_{ij} is a local heuristic and for TSP it's called *visibility;* it is usually defined as the distance between the nodes (the weight of the graph edge corresponding to the two nodes).
- α, β are two parameters which control the relative importance of pheromone trail versus visibility.

At time *t+n,* at the end of the cycle, all ants will have completed their tours and the intensity of the pheromone trail on edge *(i, j)* will be increased with a value corresponding to all ants which have walked on edge *(i, j)* during the cycle. The formula for this value is given by (2):

$$\Delta_{i,j}(t, t+n) = \sum_{k=1}^{m} \Delta_{i,j}^{k}(t, t+n) \quad (2)$$

$\Delta_{i,j}^{k}(t,t+n)$ is the intensity of the pheromone trail laid by ant *k* on edge *(i j)* in time interval *[t, t+n]* and is given by:

$$\Delta_{ij}{}^{k}(t,t+n)=\begin{cases}\dfrac{Q}{L_k}, if\ ant\ k\ uses\ edge(i,j)\\ 0, otherwise\end{cases} \quad (3)$$

where Q is a constant and L_k is the length of the tour found by ant *k*.

At the end of the cycle after the evaporation process is completed the intensity of pheromone value on edge *(i, j)* will be:

$$\tau_{i,j}(t+n)=(1-\rho)\cdot\tau_{i,j}(t)+\Delta_{i,j}(t,t+n) \quad (4)$$

where ρ is a coefficient representing pheromone evaporation $(0<\rho<1)$.

In Fig. 1 there is an example of an ant *k* which at time *t* is positioned in node *u.* Node *u* has four neighbors (*v1, v2, v3, v4*), none of which has been visited by ant *k* in the current cycle. Based on the state transition rule defined by (1) the ant has chosen to move to node *v3.* Thus, at the end of the cycle, the ant will cause the amount of pheromone on edge *(u, v3)* to be increased with $\delta=\Delta_{i,j}^{k}(t,t+n)$, defined by (3). If other ants also use edge *(u, v3)* in their tours the δ values are added together.

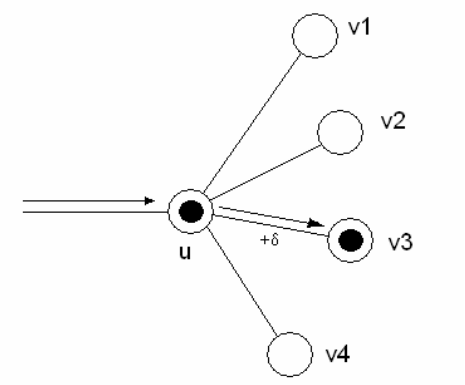

Fig.1 – When moving from node *u* to node *v3* (not visited yet) the ant lays an amount of pheromone on the corresponding edge

The outline of the Ant Algorithm is given below:

Initialize: place the m ants randomly among the cities
for t=1 to number of cycles do
  for k=1 to m do
    Repeat until k has completed a tour
      Select city j to be visited next with probability P_{ij}^{k}
    end
    Calculate the length L_k *of the tour generated by ant k*
  end
  Save the best solution found so far
  Update the trail levels $\tau_{i,j}$ *on all edges(i, j) used by the ants in the current cycle*
  Evaporate the pheromone on all edges
end
Print the best solution found

The Ant Colony Optimization Metaheuristic extends the concepts of the Ant System algorithm, in order to solve other hard combinatorial optimization problems the solutions of which can be represented as paths or circuits in graphs.

3.2. PREVIOUS PARALLELIZATION WORK

In this section a survey over the previous parallel implementation of the ACO metaheuristic will be carried out. By observing both their strong and their weak points we try to motivate the design of our own implementation.

There have been other attempts to parallelize ACO algorithms. In [3] Stutzle points out the fact that there is no rule to efficiently parallelize ACO algorithms because this process greatly depends on the underlying computing platform and on the interconnection network. He suggests the use of the MIMD architecture in the process (for example, a cluster of workstations), and then he focuses on parallel independent runs of the same sequential algorithm.

In [6] an implementation in MPI with master-slave architecture is presented, and this is similar to our approach. However for the sake of simplicity synchronous communication has been used, which affects the performance, because of the time it takes for the processors to synchronize. In order to improve the communication overhead, they have chosen to perform information exchanges between the master and the slaves once every some predefined number of iterations. This choice reduces the communication overhead but it also modifies the usual behavior of the algorithm.

A similar approach is described in [5] by Bullnheimer and Strauss, though they don't have a practical implementation. Instead they use N-MAP, a tool that can simulate the execution of message passing algorithms and analyze their performance (the ratio of computation, communication and idle times). They have achieved a speedup that increases - to some extent – proportionally with the instance size. However the communication model that was used assumes that simultaneous transmission of messages is possible and that it takes the same period of time as the delivery of a single message. This is generally not true, of course. The authors

have also felt compelled to minimize the communication overhead by performing data exchanges once every *k* iterations of the algorithm. This kind of data exchange certainly has a positive effect on efficiency and speedup but they are also aware of the fact that it distorts the ant algorithm as the ants in a processor don't interact with others at all during those *k* iterations. Furthermore, the way in which this influences the quality of the solutions is not analyzed.

In [2] a description of the implementation using the shared memory model and the OpenMP as a parallel environment is given. The authors try to show that the shared memory model is more adequate to the problem (parallelization of ant colonies) than the message passing model. Synchronization and timing issues are taken into account and also the necessary amount of effort.

An implementation using OpenMP would have at least one weak point: it hinders the programmer to have control over the slave threads by imposing the synchronization of all threads at the end of the parallel section. This results in idle times for synchronization of the threads and moreover all child threads would try to update the central data structures simultaneously. Whether or not this is the best choice greatly depends on the underlying parallel system and - as we will see in section 3.3.2 - in some cases it is preferable to do things the other way around. We have chosen to control the threads and the timing of data exchanges ourselves, with a bit of extra work.

3.3. PARALLELIZATION USING THE TWO-LEVEL FRAMEWORK

3.3.1. GENERAL DESCRIPTION

In this section we are explaining the architecture of the first framework we have designed and as a case study we are showing how it was used in parallelizing the ACO metaheuristic for TSP. More implementation details can be found in [7].

The framework has an object oriented design and was implemented in C++, using the MPI library. We have explained the choice of message-passing model and MPI over shared memory and OpenMP in the previous section. After having decided upon the most suitable model to adopt, the way in which the work will be shared among processors has still to discussed. In our case we could distribute either the vertices or the ants to processors. The first choice is not very appropriate because imbalance can occur: if there were a vertex with a high degree then the processor that contains it would have more work to do than the others. Therefore we have chosen the latter alternative (the ants are to be evenly distributed to processors).

The processors are organized in a classic master-slave structure. In Fig.2 there is an outline of the runflow in the two-level framework. Briefly, the two-level parallel framework works as follows. There are two types of processing nodes: master and slave nodes. There is a single master node and the rest of the processors are slaves. At first, the master reads the problem instance and wraps it up in a message that is broadcasted to all slaves. It then passes the control to the slaves by signaling them to start the algorithm and waits for requests coming from every slave to update the data. Each slave works with a local instance of the sequential algorithm that operates over a local copy of the central data structures. At the beginning each slave receives the input data (the problem instance), initializes the local copy of data structures together with the sequential algorithm and then waits for a start signal. When this happens the slave passes the control further to the sequential algorithm instance, providing it with a callback mechanism (*seqAlgReady()* in Fig.2) which is to be used whenever the algorithm decides it's time to pass the control back to the framework (for exchanging data with the master and other bookkeeping operations).This will call this a *checkpoint*. Basically the communication between processors only takes place during these checkpoint moments.

Both the framework and the sequential algorithm are aware of the generic concept of a *change*. This designates the elementary item in the data structures of the algorithm that can be modified. For the ACO algorithm for example a change would be a real number representing the amount of pheromone that is to be laid on an edge of the graph. In order to minimize the communication without altering the correctness of the algorithm we had to maintain detailed bookkeeping information and an updating algorithm that made use of logical clocks. They will be explained in the next section.

Since each ant acts independently of the others linear speedups can be obtained. In practice, however, the communication incurred by the management of the pheromone trails as global information is an important overhead. Since all ants use and update the pheromone trails, access to the latter is clearly the key point to efficient parallel implementations. It is necessary that the pheromone values are shared by all ants even if different processors host them. Throughout the cycle however the ants in one processor have no contact with the other ants. The "global" pheromone matrix is maintained by the master.

At the beginning the problem instance along with the work share (i.e. the number of ants) is sent to each worker. Each worker (slave) has its own local copy of the pheromone matrix, which ants modify

during the cycle. The local matrix is synchronized with the master's, as we have discussed, at the end of each ant cycle, through checkpoint operations. The synchronization is by no means accomplished by sending whole matrices over the network as for large instances this could result in serious data traffic on the interconnect and high communication overhead; instead the *patcher* object is called in to pack and send (or to receive and unpack) the collections of changes. The collections of changes for all ants in a processor are lumped together by the *patcher* object in a single transfer in such a way that there will be at most one *change* object for a modified edge, even if more than one update of its pheromone value were performed (by different ants), thus minimizing communication.

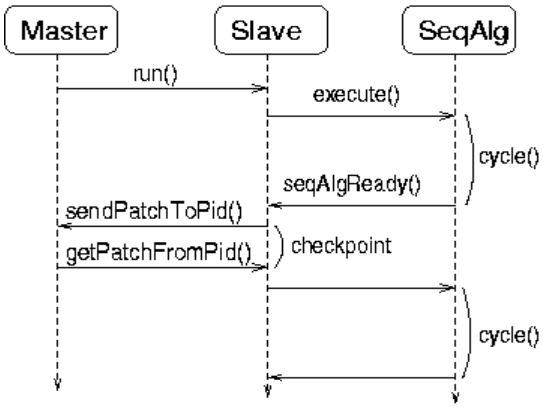

Fig.2 – How the master and the slaves work in the two-level parallel framework

Note. There is no need to take into consideration the pheromone evaporation when building the patch with changes which is to be sent over the network, as the evaporation process can be handled locally by each CPU.

3.3.2. THE CHECKPOINT

It is known that communication is the most time consuming operation in a parallel message-passing system. Since in our case all communication occurs during checkpoints this operation is critical for the communication overhead and for the efficiency. That is why it is important to implement it as carefully as possible. More specifically, we are concerned with two issues: how to schedule the checkpoints and what to do inside a checkpoint, that is, what kind of data is necessary to be sent over the interconnection network.

It is important to point out that in order to make an efficient parallel implementation; the particular parallel environment has to be considered. The underlying architecture of the parallel machine and interconnection network have major impact over the measured performance of the algorithm (mainly communication time and idle time). Since it is difficult to estimate these system traits in a theoretical formula, some tests should be run in order to have an idea about how the system behaves. We will get back to this later in this section.

The slaves request in turns data exchanges with the master; the effect of this scheduling of updates is that between two consecutive checkpoints of the same processor all other slaves have already made their changes visible in the global data structures of the master. These slave-requested data exchanges that occur at different moments make the system asynchronous and they also make it benefit from a “pipeline effect", meaning that while one processor is sending messages chances are that the others are performing computation steps.

This is not the only reason why the checkpoints are scheduled in this manner. As we have said before, the behavior of the particular parallel machine in sending messages has great influence over the performance of the parallel program. If all slaves have to asynchronously send messages to the master, one might see two ways of doing it. Either by letting them try at the same time, with no particular schedule, and let the system and the interconnect handle (presumably in an efficient way) the situation (no scheduling) or by making them take turns in performing data transfer, and serialize the data exchanges by having the master acknowledge each pending request (scheduling). Choosing between these two options is not as straightforward as it might seem. The former is expected to deliver the best performance, though the results of the tests we have run showed quite the opposite. For tests and practical implementation we have used a SunFire 15K HPC service having a backend with 48 processors. Each slave sent a message of 500,000 double values to the master with and without scheduling and the communication times were compared. The two sets of values are printed in Fig.3. It can be seen that as the number of processors increases, the time for scheduled sending of messages (the second way) is reduced to nearly half the time needed for unscheduled communication. This, we think, is a significant fact, and provides a strong argument for choosing the scheduled communication scheme over the unscheduled.

In order to collect all the changes that have occurred in the slave processors into a central master processor we cannot oversee the primitives which an MPI library offers for collective communication. Moreover one might assume that these primitives would deal with collective operations much more efficiently than the user could possibly do using only simple point to point communication primitives (send and receive operations); in our case the collective operation that would be appropriate to use is of course *MPI_Gather()*. However we found out

that - on the systems we have had access to - the "scheduled" communication we have described earlier delivered a much better performance than *MPI_Gather()* did.

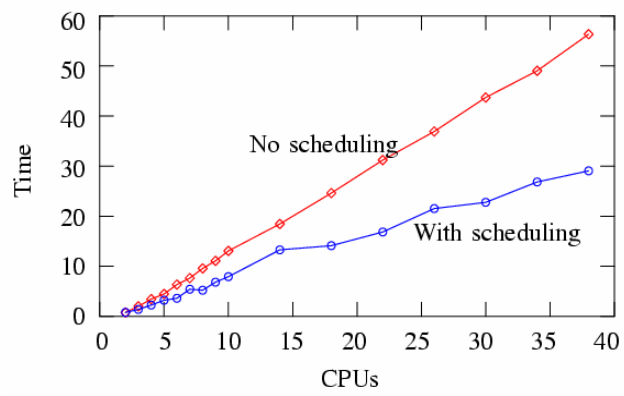

Fig.3 – The time of unscheduled versus scheduled commnication on a SunFire 15K

In the tests each "slave" processor sent a message to a "master" processor, at first using *MPI_Gather()* and then using our scheduled point to point communication. The tests were carried out with messages having lengths of 200, 1000, 10000, 100000, 200000, 300000, 400000 and 500000 double values and with a number of processors ranging from 3 to 24. Communication times were measured and in each case our system behaved better.

For example Fig.4 shows the compared communication times for the test with 10,000 doubles.

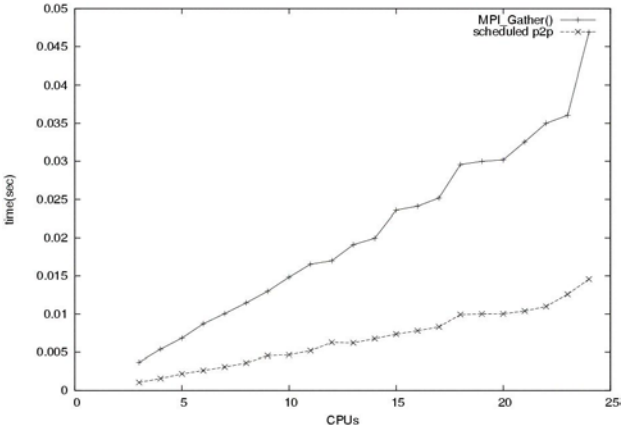

Fig.4 – Comparison of communication times: MPI_Gather() versus scheduled point to point commnication on a SMP machine with 24 UltraSPARC II processors

Now that we know how to efficiently schedule the data exchanges between the slaves and the master (the so called "checkpoint" we have mentioned), let us focus on the second issue, that is, what to send during such a checkpoint.

During a checkpoint the pair of involved processors exchanges collections of *change* objects (which we have defined in section 3.3.1): the slave sends its modifications to the master which in its turn replies with the collection of changes that the slave is unaware of. Regarding the slave, it is easy to decide what is needed to be sent in the next checkpoint: the algorithm simply adds everything that it has modified to a collection of changes, which is emptied before each cycle begins. Regarding the role of the master, however, there is a special module called the *bookkeeper* which makes use of the logical clocks to be able to determine the items in the data structure (i.e. the above mentioned changes) that are to be sent to a particular slave, should the checkpoint time come. In order to decide which changes are to be sent, an item that can be changed contains a logical clock, which can be seen as a "version number" that gets incremented. Also, each slave processor has a similar logical clock associated with it. Based on these values the master can decide which changes have to be sent to each slave.

More details about the implementation of this system can be found in [7].

3.3.3. WEAKNESSES OF THE TWO-LEVEL FRAMEWORK

For tests and practical implementation of the parallel framework we have used a Sun Fire 15K HPC service having a backend with 48 processors. The tests have been carried out with an increasing number of processors, from 2 up to 36 processors. Each value is an average over five runs and the sequential time was measured to 234.978 seconds. The diagram in Fig.5 depicts the speedup that was achieved.

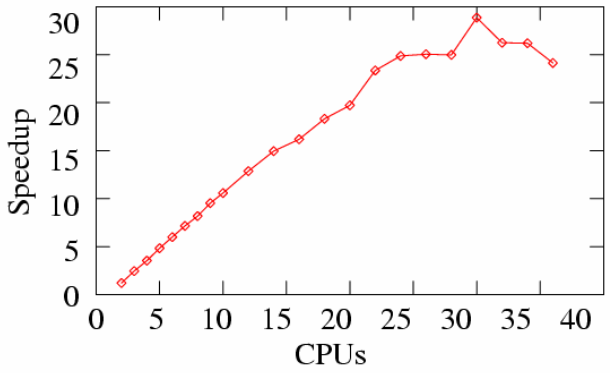

Fig5 – The speedup obtained with the implementation of the parallel ant colony algorithm for TSP using the two-level framework

It is assumed that the pronounced degrading of the speedup, which occurs over 26 processors, is happening when the sum of communication times of all slaves during a cycle reaches values close enough to the average processor computation time for one

cycle. This is the point when wait times begin to occur inside processing units when they reach checkpoints, because at that time there are still one or more processors which haven't finished their checkpoint. The cause of this bottleneck situation is that as the number of processors grows the checkpoint communication time doesn't necessarily decrease to make it possible for the increasing number of checkpoints to fit within the per-processor cycle computation time, which usually gets shorter (in the case of a parallel ACO algorithm more processors mean less ants per processor to move around, therefore less work to do). This means that a processor that is trying to perform a checkpoint while another one still has not finished its own checkpoint would have to wait until it receives the acknowledge signal from the master, signaling that the ongoing checkpoint has finished; otherwise it would have to try to overlap the checkpoints, which as we have shown is not always appropriate as it doesn't necessarily lead to better communication time.

These considerations drove us to develop the multi-level system, which tries to go around the discussed bottleneck issues of the first model.

4. THREE-LEVEL PARALLEL FRAMEWORK

In the improved three level parallel framework there are three types of processing nodes: master, submaster and slave nodes. One of the processors acts as a master, several act as submasters and the rest act as slaves. The set of slaves is partitioned so as each each slave communicates with exactly one submaster.

The system is useful only if the number of submasters is at least 2 and there is at least one submaster with more than one slave. The number of submasters (and therefore the number of slaves) is a parameter in the program and is known before runtime. Based on the rank number, each processor is able to tell whether it is a slave, a submaster or a master. Also each slave can deduce the rank of its submaster and each submaster can compute the list of the slaves it has to deal with.

The runflow in the three level framework is presented in Fig.6. First, as in the case of the two-level framework, the master detects the problem instance, wraps it up in a message that is broadcasted to all the other processors and initializes the central data structures. The control then passes to the slaves which start the algorithm while the master and the submasters are waiting for requests to update the data. The slaves' behavior is very much the same as in the two-level framework, the only difference is that now it does not communicate directly with the master but instead to its submaster.

A slave initializes its structures and then passes a callback function (*seqAlgReady()* in Fig.6) to the sequential algorithm (*SeqAlg*) before letting it take over. When the algorithm has its partial results ready (for example at the end of a cycle) it calls this callback function it was provided with, passing the control back to framework. The slave then submits a checkpoint request to its submaster. When the acknowledge is received it packs the data it has modified as a list of *change* objects, sends them and then receives and unpacks the changes from submaster, applying them to the local structures. When the checkpoint is over *seqAlgReady()* returns, and the sequential algorithm carries on.

As part of the checkpoint, the solution the slave obtained in the last cycle - or a qualitative evaluation of it - is also passed to the submaster.

A *bookkeeper* in each submaster stores the list locally, builds a complete list of changes that need to be sent to that specific slave and then sends it.

When all (or a tunable percent) of the slaves of the submaster have completed their checkpoints, the submaster initiates a checkpoint with the central master. It efficiently packs all the changes it had received from the slaves in the last cycle and sends them to the master in a single message - if possible.

After acknowledging the request the master receives the list of changes from the submaster and updates the global data structures. Then the master sends to that submaster a patch containing the changes already received in the current cycle from other submasters. The submaster then passes the control to the slave as described earlier.

So, the checkpoint between a slave and a submaster is similar to the checkpoint that takes place between a submaster and the master.

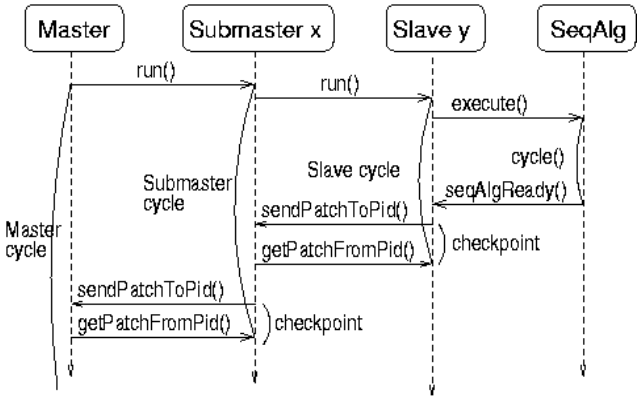

Fig.6 – How the master, submasters and the slaves work in the three-level framework

What else can be done inside a checkpoint? Basically anything that is considered important by the algorithm which is dealt with by the framework.

The procedures for sending and receiving collections of changes are supplied by the sequential algorithm and the checkpoint procedure can be overridden. In this way the protocol for data exchange can be customized to meet any specific demands. For example there are several parallel implementation of ACO meta-heuristic [5] that in order to minimize the communication overhead chose to schedule the data exchanges between the server and the master to take place once every predefined number *k* of cycles. If it's needed this can also be done in our case by making the sequential algorithm call the callback function (*seqAlgReady()*) every *k* cycles. Another example is the global updating rule in ant algorithms, which might exist or not. In our case this can be managed by changing the function that handles the checkpoint requests in the master.

5. GENERALITY OF THE FRAMEWORK

The architecture of the framework is object oriented and modular. That is why it is easy to adapt it in such a way to be used with other sequential algorithms. The changes which have to be made are local and they do not require modifications in other places in the framework. The user of the framework has to implement C++ classes for the following modules:

1. the sequential algorithm;
2. the *change*;
3. the *patcher* object, to handle the collections of changes and to apply them to the local data structures;
4. the *packager* object which is to efficiently pack the problem instance in order to be sent to all processors over the network.

6. EXPERIMENTS

In order to test the framework and the ACO for TSP parallelization, we have used a TSP instance with 229 cities (*gr229.tsp)* from the TSPLIB library.

The first test runs have been carried out on a network of 9 PC's (Pentium IV with 512 MB RAM) with an increasing number of slave processes, from 14 up to 112. The results that were achieved are depicted in Fig.7.

The execution time for sequential algorithm is 41.618 seconds in this case. As it can be observed, the time achieved by the parallel framework is better than the sequential time, but the period of parallel time increases proportionally with the number of processes. This can be explained by the fact that starting from 9 processes it is not possible to assign one process per processor, so we do not have a real parallelism.

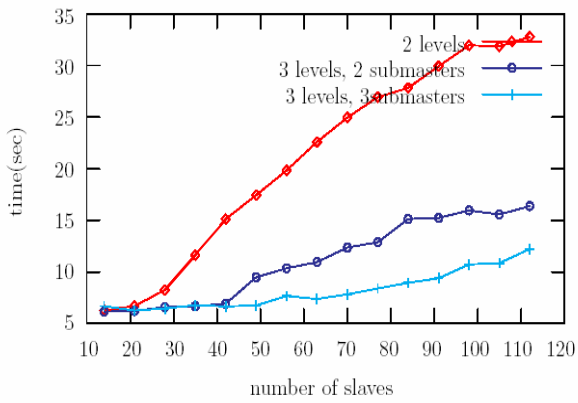

Fig.7 – The time for TSP achieved with the two-level and three-level parallel framework, using a PC's network

As Fig.7 shows, the parallel times for the three-level framework should become better than those of the two-level framework only for a number of processes somewhere between 20 and 30 processes. Therefore we believed that the advantage would only become effective on a parallel machine with at least 30 CPUs.

The first tests on a real parallel machine were done on a Sun HPC service having a backend with 24 processors. The multi-level framework showed little or no advantage over the simple master-slave system. This proved our supposition that the advantage of the multi-level framework should become effective on a parallel machine with at least 30 processors.

For tests on a parallel machine with at least 30 CPUs we have used a Sun Fire 15K HPC service having a backend with 48 processors. The test runs have been carried out with an increasing number of processors, from 10 up to 40 processors. The diagram in Fig.8 below depicts the speedups that were obtained for the two systems we are comparing: one curve is for the simple master-slave system and the other one is for the multi-level system.

It can be seen that over 30 processors, the speedup decreases in the case of the simple master-slave paradigm while the multi-level system manages to preserve an approximately linear speedup. Thus, the test results for a 3-level framework prove that the multi-level model overcomes the limitations in the basic master-slave model

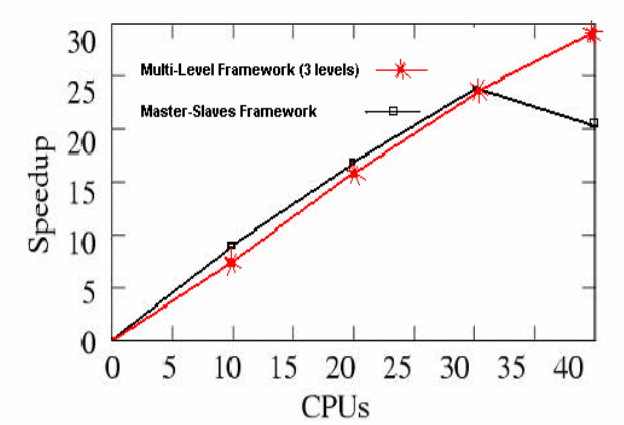

Fig.8 – The speedup for TSP achieved with the two-level parallel framework, using a Sun Fire 15K HPC

7. CONCLUSIONS AND FUTURE WORK

The ACO for TSP implemented by means of the two-level parallel framework has good results: approximately linear speedup up to 30 CPUs and low communication cost. The degradation for a larger number of CPUs is a disadvantage of the master-slave paradigm.

The test results prove that the multi-level model overcomes the limitations in the master-slave model.

Further on, we intend to implement a tree model that establishes the tree depth taking into account hardware architecture.

8. CREDITS

The authors would like to acknowledge the support of the European Commission through grant number HPRI-CT-1999-00026 (the TRACS Programme at Edinburgh Parallel Computing Centre) and the HPC-Europa consortium.

The authors would also like to acknowledge the support of the the Romanian HPC Centre, “CoLaborator”.

9. REFERENCES

[1] M. Dorigo and G. D. Caro. Ant algorithms for discrete optimization, *Artificial Life*, no. 5 (1999), pp. 137-172.

[2] P. Delisle, M. Krajecki, M. Gravel, and C. Gagne. Parallel implementation of an ant colony optimization metaheuristic with OpenMP, *Proceedings of the 3rd European Workshop on OpenMP (EWOMP’01)*, Barcelona, Spain, 2001.

[3] T. Stutzle. Parallelization strategies for ant colony optimization, *Lecture Notes in Computer Science*, vol. 1498 (1998), pp. 722-731.

[4] E. Ghazali Talbi, O. Roux, C. Fonlupt, and D. Robillard. Parallel ant colonies for combinatorial optimization problems, *Lecture Notes in Computer Science*, vol. 1586 (1999), pp. 239-247.

[5] B. Bullnheimer, G. Kostis, and C. Strauss. Parallelization strategies for the ant system, *High Performance Algorithms and Software in Non-linear Optimization* (R. D. L. et all., ed.), *Applied Optimization*, vol. 24 (1998), pp. 87-100.

[6] D. A. L. Piriyakumar and P.Levi. A new approach to exploiting parallelism in ant colony optimization", 2001.

[7] M. Craus and L.Rudeanu. Parallel Framework for Ant-like Algorithms, *Proceedings of the 3rd International Symposium on Parallel and Distributed Computing (ISPDC 2004)*, Cork, Ireland, July 5th-7th 2004, pp.36-41

Mitică Craus. *Education:*

1975 – 1979 : Student at the Faculty of Mathematics and Computer Science - "Al. I. Cuza" University of Iasi

1999 : Ph.D. in computer science.

Present job: Associated Professor at the Faculty of Automatic Control and Computer Engineering, Department of Computer Science and Engineering

Computer skills: Data structures, algorithm design (sequential, parallel and distributed), C/C++, Java, PASCAL and FORTRAN programming

Laurenţiu Rudeanu. *Education:*

1997 – 2002 : Student at the Faculty of Automatic Control and Computer Science - Computer Science Department of the "Gh.Asachi" Technical University of Iasi

2003 : MSc in Distributed Systems

Present job: Java Software Developer at a private software company.

Computer skills: Programming: Data structures and algorithms, C/C++, Java and J2EE technologies, SQL, MPI, OpenMP, Perl, shell scripting

Database administration: Oracle, MS SQLServer

Networing and system administration: Linux, FreeBSD, WindowsNT, Cisco routers