Scientific Programming 13 (2005) 205–217
IOS Press

Parallel framework for cooperative processes

Mitică Craus and Laurenţiu Rudeanu
Department of Computer Engineering, Technical University "Gh.Asachi", 700050 Iaşi, Romania
E-mail: craus@cs.tuiasi.ro, lrudeanu@yahoo.com

Abstract. This paper describes the work of an object oriented framework designed to be used in the parallelization of a set of related algorithms. 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 algorithms that the framework can be used with have several things in common: they have to run in cycles and the work should be possible to be split between several "processing units". The parallel framework uses the message-passing communication paradigm and is organized as a master-slave system. Two applications are presented: an Ant Colony Optimization (ACO) parallel algorithm for the Travelling Salesman Problem (TSP) and an Image Processing (IP) parallel algorithm for the Symmetrical Neighborhood Filter (SNF). The implementations of these applications by means of the parallel framework prove to have good performances: approximatively linear speedup and low communication cost.

1. Introduction

The idea behind the paradigm 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 several ant colony algorithms, 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 and attempts of parallelization [1–4] with their shortcomings and advantages. 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. We wanted to end up with a parallel framework flexible enough to be configured for any 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". Genetic algorithms, for example, 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. Afterwards we present our design and implementation. As examples we present an Ant Colony algorithm for Travelling Salesman Problem and an Image Processing algorithm for the Symmetrical Neighborhood Filter.

2. Goals

The two main aims of our efforts 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 good speedups even for larger problem instances. In order to minimize the communication without altering the algorithm's idea we had to maintain detailed bookkeeping information and to use an updating algorithm that makes use of logical clocks, as it will be discussed later on.

ISSN 1058-9244/05/$17.00

3. General description of the parallel framework

As previously stated, the framework uses message-passing communication (the MPI library). Briefly, the parallel framework works as follows.

At first the problem instance is read and the control is passed to slaves by signaling them to start the algorithm and the master 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. In the beginning each slave learns about the input data (i.e. 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 which is to be used whenever the algorithm decides it is time to pass the control back to the framework (for exchanging data with the master and other bookkeeping operations). We will call this a *checkpoint* and we will describe it in more detail since it is a critical operation for the efficiency of communication. Basically communication between processors only takes place during these checkpoint moments.

4. 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.

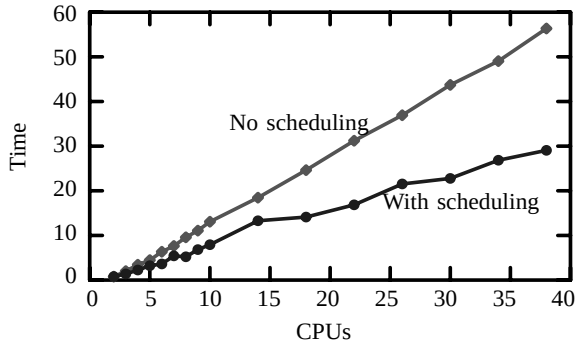

Fig. 1. The time of unscheduled versus scheduled communication on a SunFire 15 K.

4.1. How the checkpoints are made

4.1.1. The policy for sending messages

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. This slave-requested data exchanges that occur at different moments make the system asynchronous and it also makes it benefit of 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 the checkpoints are scheduled in this manner. As we have said before, the particular parallel machine's behavior in sending messages has a great deal of 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 transferring data, and serialize the data exchanges by having the master acknowledge each pending request(scheduling). Choosing between these two ways 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 Sun Fire 15 K 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 were printed in Fig. 1. It can be seen that as the number of processors increases the time for scheduled sending of messages

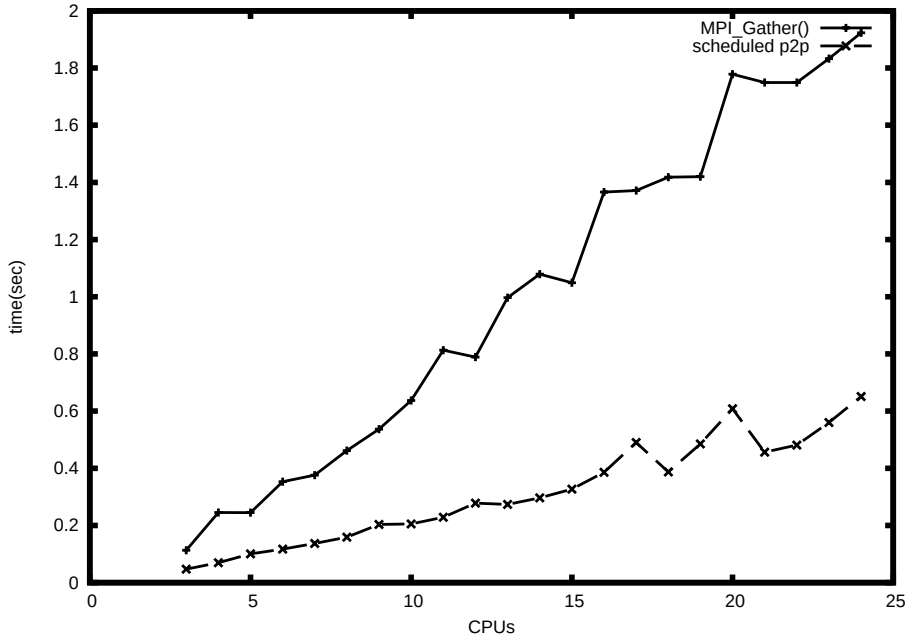

Fig. 2. The time of MPI Gather versus scheduled communication on a Sun HPC.

(the second way) drops to nearly half the time needed for unscheduled communication.

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 has to offer 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 for 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. In the tests each slave processor sent a message to a master processor, first using *MPI Gather*() and then using our scheduled point to point communication. The tests were carried out with various messages lengths and with an increasing number of processors. Communication times were measured and in each case our system behaved better. The results obtained for message lengths of 500,000 double values are depicted in Fig. 2. The tests were run on a Sun HPC system with 24 processors and the Sun MPI library was used as the MPI implementation.

In our opinion these considerations are significant and provide a strong case for choosing the latter, scheduled communication scheme over the former, unscheduled.

However it is not safe to assume that the above conclusion would hold in any context, so the choice of scheduling the data exchanges is ultimately left as a parameter to the user. A test program similar to the simple program we have made to evaluate the benefit of scheduling the transfers can help the user in making the decision. If the parameter is left unchanged the default behavior for the framework is to use scheduled communication.

4.1.2. What to do inside a checkpoint

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.

Both the framework and the sequential algorithm are aware of the generic concept of a *change*. This designates the elementary item in algorithm's data structures 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. During a checkpoint collections of changes are exchanged: the slave sends its modifications to master which in turn replies with the collection of changes that the slave is unaware of. On the slave

```
sendChangesToSlave(slave)
begin
 //get the collection of changes for slave
 changeCollection = { } //changes to be
                        //sent to slave
 for ch in all items in the data structures do
   if ch.clock > slave.clock //update
      add ch to changeCollection
 end for
 //send the changes to slave
 sendChanges(changeCollection, slave)
 //update the logical clocks
 currentClockValue++ //increment current
                     //clock value
 for ch in changeList do
   ch.clock=currentClockValue
 end for
 slave.clock=currentClockValue
end
```

Fig. 3. The procedure that the master executes in order to send updates to a slave.

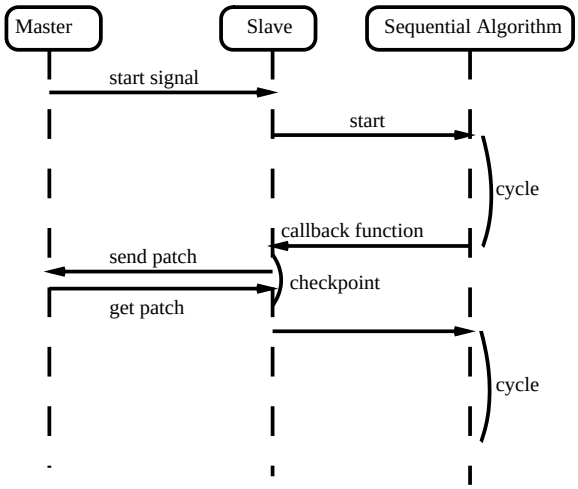

Fig. 4. How the master and the slaves work in the two-level parallel framework.

side 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). On master's side, 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 change also 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.

In order to get the list of changes and send it to a slave, the master executes the procedure in Fig. 3.

This way a slave receives only the changes whose clock values indicate that they were made after last checkpoint of that slave. Since then other slaves have certainly undertook checkpoints and have sent their changes to the master, changes which have to be transmitted to the slave that is currently undertaking a checkpoint. Another way to do this would be to allocate a queue of outgoing changes for each slave and to place each incoming change in the other slaves' queues.

In Fig. 4 below there is an outline of the runflow in the framework. The master passes the control to slaves by signaling them to start executing the sequential algorithm and then waits for checkpoint requests. The slave initializes its structures and then passes a callback function to the sequential algorithm before letting it take over. When the algorithm completes a cycle and has its partial results ready it calls this callback function, passing the control back to framework. The slave is then submitting a checkpoint request to master. When it receives the acknowledge it packs the changes to be sent, sends them and then receives and unpacks the changes from master, applying them to local structures. When the checkpoint is over the callback function returns and the sequential algorithm carries on.

As part of the checkpoint, the solution obtained by the slave in the last cycle – or a quantitative evaluation of it – is also passed to the master.

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 [1,5] 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. Framework implementation

The framework has an object oriented design and was implemented in C++, using the MPI library. Be-

side the master and slave classes there are several components that interact and are related to the specific algorithm. The framework should be able to switch an algorithm with another one with a minimum of effort on behalf of the programmer, therefore there has to be a way to change the "family" of objects to be created once the sequential algorithm has changed. One way to accomplish this is by the use of an abstract factory which user supplied concrete factories will inherit from. The abstract factory the framework is working with is called $SeqAlgFactory$. The user should create a concrete factory as a subclass of this abstract factory and pass it as an initialization parameter to the framework. Inside the concrete factory there have to be functions for creating objects of the related types described above: the sequential algorithm, the input data for the algorithm, the "bookkeeper", the "change" and the "patcher". In this section we give a short description of these classes.

The classes *CmdUnit* and *ProcessingUnit* encapsulate master and slave functionality, respectively. They work much like a template, defining the behavior of the framework. These two are the main classes and provide a frame where the sequential algorithm and all related objects fit in.

The *CmdUnit* interface is described bellow:

void init(*SeqAlgFactory* sf*): This function takes care of initializing the master processing unit. Normally the user does not need to change this function. The parameter *sf*, of type *SeqAlgFactory*, has to be provided by the user.

void run(): Once this function is called the master processor enters in running state; when the slaves become active too the framework is starting the execution of the sequential algorithm.

void abort(): Forces the framework to stop the execution of the sequential algorithm. The halting process is "graceful" and the data structures are reset so that a clean restart of the algorithm is possible.

void terminate(): This callback is being invoked by the framework after the sequential algorithm has finished. The user can define this function to run custom tasks at the end of the sequential algorithm, like finding the best solution etc.

void ckptAck(*int pid*) : Accepts a pending checkpoint request coming from the slave denoted by pid.

void getTotalWork(): The user should implement this in order to have a quantitative estimation of the total work to be done by the sequential algorithm. It is useful for determining the work share for each slave.

The *ProcessingUnit* class is composed from the functions below:

int init(*SeqAlgFactory**): Initializes the slave processing unit, in the way that *CmdUnit::init*() takes care of initialization in the master. The parameter *SeqAlgFactory* parameter has to be provided by the user.

void run(): It is similar to the function with the same name in master: it sets the slave processor in running state and synchronize with the others. When all of them are ready the framework is starting to the execution of the sequential algorithm.

void abort() and *void terminate*() These two functions are similar to ones found in *CmdUnit*.

void seqAlgReady(): A callback that is invoked by the framework at the end of each algorithm's cycle and takes care of performing a checkpoint with the master processor. Normally the user should not need to touch this unless a special behavior is needed.

int getWorkShare(): Gives an estimation of the work share attributed to this unit, based on the master's estimation of the total amount of work

void requestCkptAck(): Posts a checkpoint request to the master and waits for acknowledge.

The *SeqAlgFactory* class is responsible for providing the instance of sequential algorithm, subclass of *SeqAlg*, and the tools related to it: the *Patcher* and the *BookKeeper* objects. The following functions have to be provided when the user is using the framework with a specific algorithm: *SeqAlg* createSeqAlg*(), *Patcher* createPatcher*() and *BookKeeper* createBookKeeper*(*SeqAlg* alg*, *Patcher* p*).

SeqAlg is the sequential algorithm the framework is working with (i.e. that is to be parallelized). A subclass of this class should be provided by whoever wants to make use of the framework. The description of this class follows.

long int getTotalWork(): A user estimation of the total work to be done (in parallel).

void addCycleChange(*Change**): Adds a new change to the set of changes made so far in the current cycle.

Iterator getCycleChanges*(): Returns the changes made so far in the current cycle.

void init(): A callback which is invoked by the framework before the sequential algorithm starts in order to allow for custom initialization. This has to be defined by the user.

void readInstance(*const char**): Reads the problem instance from a file. The instance is closely related to the specific algorithm, that's why the user has to define this function.

long int getTotalCycles(): Gives the total number of cycles needed by the sequential algorithm. This has to be defined by the user.

void cycle(): The behavior of the sequential algorithm is mainly given by this function. This has to be defined by the user.

void applyChange(*Change **): Updates the local data structures with a change that was just received as a result of a checkpoint operation. This has to be defined by the user.

The *Patcher* class is used for efficiently packing and unpacking the collection of algorithm-dependant changes to be sent over the network. Here is a listing of the functions it provides:

void addChange(*Change**): Enqueues a new change to the set of changes to be sent.

void removeAllChanges(): Clears all previously added changes.

Iterator getChanges*(): Returns the current set of changes queued in this object (not yet sent).

void setChanges(*Iterator**): Sets the whole collection of changes to be sent by this object.

Iterator getPatchFromPid*(*int pid*) and *void sendPatchToPid*(*Iterator* it, int pid*) : The user has to implement these two functions in order to define the protocol to serialize/deserialize the sets of changes for transfer them from a processor to another. This is a time consuming operation and for efficiency reasons it can not have a generic implementation: since the *Change* objects are defined by the user, the functions that pack and unpack the sets of changes also have to be defined by the user.

The *BookKeeper* class handles the bookkeeping of changes and other data structures in the master. Below is a list of the functions it provides.

void initStructures() and *void freeStructures*(): These two functions are called by the framework to initialize and destroy any specific data structures the bookkeeper might have.

void handleCkptRequest(*int pid*): This takes care of a checkpoint request coming from the slave processor denoted by *pid*. Basically what it needs to do is to get the changes from the slave (by calling *receivePatch*()) and to reply with the proper list of changes for that slave (*sendChangesToSlave*()).

Iterator getChangesForPid*(*int pid*): Obtains the list of changes that the slave with id *pid* needs to know about.

sendChangesToSlave(*int pid*): Sends the set of changes obtained by *getChangesForPid*() to slave *pid* and then calls *updateClocks*() updates the logical clock values for the same slave.

void updateClocks(*int pid*): Updates the clocks after a slave with id *pid* has performed a checkpoint.

void receivePatch(*int*): Delegates to the patcher object to receive the set of changes from the slave.

*Iterator *getReceivedChanges*(): Returns the set of changes that has just been received from the slave via *receivePatch*().

void applyReceivedChanges(): This calls the user defined function *applyChange*() for each change that was received.

In the functions above we have encountered the class *Iterator* for several times; it is useful for iterating through a sequence of generic *Element* objects. For example the *Change* is also an *Element*. The semantic is straightforward and here is the content of these two classes.

void addElement(*Element *elem*)

void removeElement(*long int*)

int hasMoreElements()

Element getNextElement*()

long int getCount(): Returns the number of items in the iterator.

void rewind(): Rolls the iterator back to the first item in the collection.

void empty()

*Iterator *dupData*(): Copies the elements of this iterator into a new one.

void destroy()

An *Iterator* works with generic *Element* objects. The *Element* class is very simple, but may be subclassed by the user as needed. It exposes only one function:

long int getId(): This identification number is for indexing purposes.

One case where *Element* is subclassed is the *Change* class. In addition to the functions inherited from $Element$ this class has some functions for managing the clock value:

long int getClockValue(): Returns the current clock value associated with this change.

void setClockValue(*long int*): Sets a new clock value for this change.

void incClockValue(): Increments the value of the current clock value for this change.

6. Case study I: Ant colony algorithm for the travelling salesman problem

As we have said earlier, one of the algorithms we have chosen to parallelize using the designed framework is the Ant Colony (ACO) algorithm for Travelling Salesman Problem (TSP).

6.1. Artificial ant colonies and the basic application to 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 representing the distances between the cities. The Euclidean variant 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 generalization of the *Ant System algorithms*, which were inspired by the social behavior observed in real ant colonies. The interesting aspect of their behavior was their ability to find the shortest path from the place where food was found to the nest. The investigations showed that the ants managed to do this by communicating indirectly via pheromone trails that they left behind. 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* [6–8]. The idea of ACO is best illustrated by showing one of the first applications of the ant algorithm – the *Ant System* algorithm – which was targeted to find good solutions for the TSP. Here is a short description of how it works. The edges of the graph have pheromone values, which the ants modify. 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. The state transition rule is a heuristic based on the weight and the amount of pheromone of the edge between the two cities. Each ant move is called an *iteration* and 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. In some implementations, 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 fraction 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 presented in a more formal way in the remaining part of this section.

Let $\tau_{ij}(t)$ 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 Eq. (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.

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

- $allowed_k(t)$ is the set of cities not visited by ant k at time t.
- $\eta_{ij}(t)$ 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 Eq. (2).

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

$\Delta_{ij}^{k}(t, t+n)$ 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 Eq. (3).

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

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

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

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

The outline of the Ant System 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 ant k has completed a tour
      Probabilistic choice for the next city
      j to be visited
    end repeat
    Evaluate the solution: calculate the
    length
       Lk of the tour generated by ant k
  end for
  Save the best solution found so far
  Update the trail levels on all edges (i,j)
     used by the ants in the current cycle
  Evaporate the pheromone on all edges
  Apply the global updating rule (if defined)
end for
Print the best solution found
```

The new ACO metaheuristic further extends the concepts defined by this simple Ant System algorithm, making it possible to be used in solving any combinatorial optimization problem whose solutions can be represented as paths in a graph.

6.2. Other approaches/previous work

There are quite a few parallelizations of ACO algorithms in the literature. We describe the most important of them briefly: 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. The author compares the quality of the solution obtained by executing several independent short runs of an ant algorithm with the solution quality of the execution of one long run equaling the sum of the running times of the short runs. In some specific conditions the short runs prove to be better. Also, with no communication between them, they can be easily run in parallel with virtually linear speedup.

In [1] an MPI implementation 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 needed 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, master slave synchronous 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 proportionally with the instance size. However the communication model that was used assumes that simultaneous transmission of messages is possible and that it takes as long 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 [9] the authors take note of the fact that in a master-slave approach, with centralized data structures, a bottleneck can occur at the master. A solution for this problem is to have a hierarchy of master processes instead of a single one. At the bottom level of the hierarchy each master takes care of a number of slaves.

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 have

seen earlier – 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.

6.3. Implementation

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 be 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 contained 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).

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 for the pheromone values to be shared by all ants even if the ants are hosted by different processors. 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.

In the beginning, all workers read the problem instance and are told by the master about their work share (i.e. the number of ants). 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 therefore 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 any modified edge, even if more than one update of its pheromone value were performed (by different ants), thus minimizing communication.

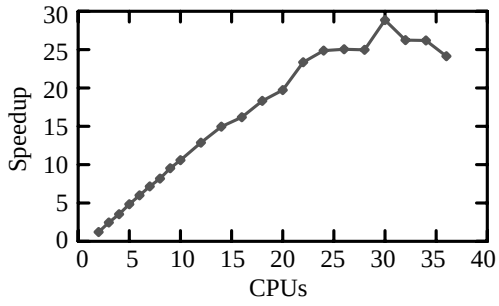

Fig. 5. The speedup achieved with the two-level parallel framework for TS.

The generic change we have mentioned earlier when describing the design of the framework is represented by an edge and a real value reflecting amount of pheromone to be added to the specified edge.

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

The generic classes described in Subsection 5 were subclassed as follows:

- *SeqAlg: AcoAlg*.
- *BookKeeper: AcoBookKeeper*.
- *Change: AcoChange*.
- *Patcher: AcoPatcher*.

As a final note, we have already showed in Subsection 4 some of the ways the framework can be customized, by modifying or even overriding different functions.

6.4. Experiments

In order to test the framework and the parallelization of ACO for TSP, a TSP instance with 229 cities (gr229.tsp) from the TSPLIB library was used. For tests and practical implementation of the parallel framework we have used a Sun Fire 15 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 below depicts the speedup that was achieved.

7. Case study II: Parallel image processing: symmetrical neighborhood filter

The framework application we are presenting in this section is from the field of image processing and it is about applying a convergent filter onto an image. We are not claiming that in this case the best choice for parallelization is to use our framework, but this rather wants to be another usage example of the framework for a new kind of application (image processing).

7.1. How SNF works

The Symmetrical Neighborhood Filter (SNF) is used in image segmentation algorithms, which cluster pixels into homogenous regions. Before starting to classify the pixels by adjacency and similar properties, the noise inherent to real images (due to physical equipment, lighting conditions, physical imperfections) has to be reduced in order to not have single pixel outliers. The SNF filter smoothes out the interior of a region to a near-homogenous level and not only it preserves the existing edges but also sharpens blurred edges (as opposed to most existing preprocessing filters, which smooth the interior of regions at the cost of degrading the edges).

The SNF works the following way: for each pixel the gray values of symmetric neighbor pairs around the center pixel are compared with the pixel's gray value. The value for a pair is chosen to be the closest value – of the two values in the pair – to the center pixel, if this value is within e of the center pixel, or the value of the center pixel itself otherwise. An average value is computed over the center pixel and the computed values of the 4 pairs, and then the center pixel's gray value is set as the mean between that average and the center pixel's current gray value. If the new value doesn't differ from the previous one the center pixel is called to be *fixed*.

The filter is convergent and is applied for a predefined number of times or until a termination condition is fulfilled (eg: until a percentage of the pixels become fixed)

7.2. Other approaches

Several previous successful parallelization attempts (for SIMD machines, systolic arrays and pipelined computers) are mentioned in [10]. The authors also have their own implementation, with an obvious data exchange pattern where a processor is taking care of a region of the matrix representing the image, of size

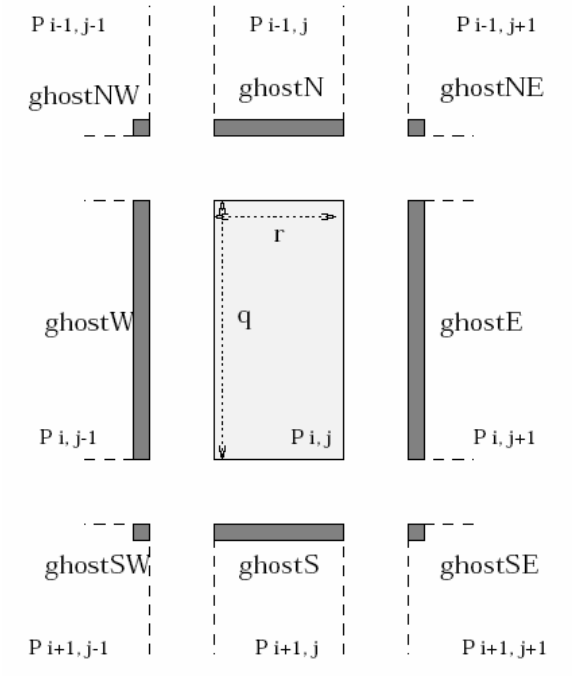

Fig. 6. The ghost regions [10].

$\frac{n}{\sqrt{p}}$ with n being the size of the (square) matrix and p is the number of processors. For computing the gray values for the pixels on the border of the region a processor needs to know those pixels' neighbors. They name those pixel regions as *ghost regions* and for a region by size $q \times r$ held by a processor they define 8 ghost regions: for North, South (r pixels each), East, West (q pixels each) and also four one-pixel regions for North-East, North-West, South-East and South-West, as shown in Fig. 6. The pixels in the ghost regions are in neighbor regions and are actually processed by a "neighbor" processor (which, in turn needs to know about pixels on the border of the current processor). Therefore when an iteration of the filter is finished each processor needs to synchronize with its neighbor processors to exchange the new values – if any – that were computed for the pixels in the ghost regions. If a pixel was or becomes fixed there is no need to synchronize it.

In the communication scheme they use, a processor exchanges "ghost regions" with its eight neighbors after each iteration of the filter.

7.3. Implementation

The strategy for data exchange is very similar to the one described in [10] as this is a natural and obvious

approach: the pixels that a processor has "touched" in the last application of the filter have to be spread to the neighbor processors in order for them to update their ghost regions. The implementation of this data exchange, however, differs in some respects.

In terms of our framework, the set of changes that a worker has made in the last cycle and needs to communicate to master in a checkpoint is the current set of non-fixed pixels on the border, i.e. those that got their gray value updated and are located on the border. The other modified pixels will be kept ready for the next iteration and only after the last cycle they will be sent to the master in order to re-assembly the whole matrix.

The bookkeeper in the master knows about what pixels each worker is interested in keeping in synch and when it receives the lists of changes from all processors it knows to assembly response-patches containing the combined lists of values which were modified by other processors and for which the worker is interested in. Moreover, as a slave can deduce how the image was split among processors, it is able to tell which pixels are not interesting to any of its neighbors and it can filter them out from the set of changes in order to lower the communication cost. This is the case with the pixels on the out-border of the big image, which are not near any of the neighbor regions.

For this application the new classes that were derived from the ones described in Section 5 are as follows:

- *SeqAlg: SnfAlg*: contains the logic for updating the gray values for the pixels in the region allocated to the current worker.
- *BookKeeper: SnfBookKeeper*: manages the update lists for each worker, based on their interest lists. The image is stored in a matrix of integers P, where each element P_{ij} contains the gray level for pixel at coordinate (i, j) in the initial image. In our approach the image is split in chunks of rows (or columns) instead of square regions. The interest lists that are built in the master's bookkeeper are based on what rows/columns each processor has.
- *Change: SnfChange*: contains a pair of coordinates for the pixel along with an integer representing the new gray value that was computed.
- *Patcher: SnfPatcher*: for handling the updates of the ghost and border regions in each processor.

At checkpoint time each worker first sends its list of cycle changes (pixels on the border that were modified in the past cycle, as discussed above) packed in a single message to the master node. In the second step it receives the list of changes that need to be applied to the local region in order to update the pixels in the ghost regions that were modified by other processors. Therefore instead of eight data transfers only two are necessary. We have not studied whether or not this brings a performance gain over the approach in [10].

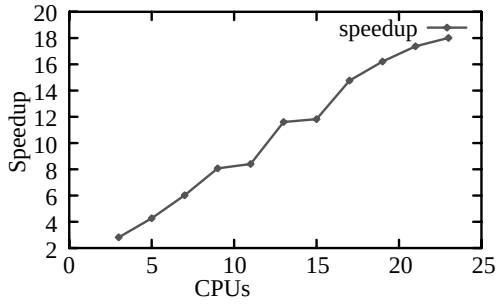

Fig. 7. The speedup achieved with the two-level parallel framework for SNF.

7.4. Experimental results

The tests of the SNF algorithm within our parallel framework were done on a Sun HPC service having a backend with 24 processors. The test runs have been carried out with an increasing number of processors, from 4 up to 24 processors. The tests consisted of applying the SNF filter on a randomly generated image of 1024 $\times$ 768 pixels in size, with 16 bit gray levels. The filter was configured to stop after 50 iterations and had the parameter e (described in section A) set to 20. The diagram in Fig. 7 below depicts the speedup that was achieved.

8. Conclusions, actual and future work

The algorithms tested by means of our parallel framework have good performance: the approximatively linear speedup (for up to 22 CPUs) and low communication cost. It is assumed that the pronounced degrading of the speedup, that occurs over 22 processors in the case of ACO, 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 reason we think it comes to such a 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 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 a good idea as it doesn't necessarily lead to better communication time.

These suppositions have driven us to develop a multi-level model, which tries to go around the discussed bottleneck issues. These is part of our actual and future work.

We briefly describe a three-level parallel framework. There are three types of processing nodes: a single *master*, several *submasters* and several *slaves*. The master communicates with the submasters and each submaster communicates with a predefined set of slaves. 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 take care of. At first, the problem instance is read and the central data structures are initialized. 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. Each slave works with a local instance of the sequential algorithm that operates over a local copy of the central data structures. It receives the input data (the problem instance) and initializes the local data structures together with the sequential algorithm. It then passes the control further to the sequential algorithm instance, providing it with a callback mechanism to be used whenever the algorithm decides it's time to pass the control back to the framework, for updating the data structures with what other slaves have computed and other bookkeeping operations. Inside a checkpoint the slave sends the data it has modified to its submaster, as a list of generic change objects. The submaster has some temporary data structures for forwarding the data between the slaves and the central master. 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. The slave then carries on executing another cycle of the sequential algorithm. When all or a tunable percent of a submaster's slaves 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. The master applies the list of changes to its structures and also packs the data that the submaster is unaware of and sends them to the submaster (that contains changes made by other slaves in other submasters). So the checkpoint between a slave and a submaster is similar to the checkpoint that takes place between a submaster and the master.

The first tests for the three-level framework have proven that the multi-level model can overcome the limitations of the basic master-slave model.

Acknowledgements

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. As well, the authors would like to acknowledge the support of the the Romanian HPC Center CoLaborator.

References

[1] D.A.L. Piriyakumar and P. Levi, *A New Approach to Exploiting Parallelism in Ant Colony Optimization*, 2001.

[2] P. Delisle, M. Krajecki, M. Gravel and C. Gagne, *Parallel Implementation of an Ant Colony Optimization Metaheuristic with OpenMP*, in Proceedings of the 3rd European Workshop on OpenMP (EWOMP?1), (Barcelone, Espagne), 2001.

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

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

[5] B. Bullnheimer, G. Kostis and C. Strauss, Parallelization Strategies for the Ant System, in: *High Performance Algorithms and Software in Non-linear Optimization*, (), R.D.L. et all., ed., Vol. 24 of Applied Optimization, Kluwer, 1998, pp. 87–100.

[6] M. Dorigo and G.D. Caro, Ant algorithms for discrete optimization, *Artificial Life* (5) (1999), 137–172.

[7] M. Dorigo and L.M. Gambardella, Ant colony system: A cooperative learning approach to the travelling salesman problem, *IEEE Transactions on Evolutionary Computation* **1**(1), 1997.

[8] A. Colorni, M. Dorigo and V. Maniezzo, *Distributed Optimization by Ant Colonies*, in Proceedings of Ecal91 – European Conference on Artificial Life, (Paris, France), 134–142, Elsevier Publishing, 1997.

[9] V.D. Cung, S.L. Martins, C.C. Ribeiro and C. Roucairol, *Essays and Surveys in Metaheuristics*, ch. Strategies for the Parallel Implementation of Metaheuristics, p. 644, Hardbound, 2001.

[10] D.A. Bader, J. Ja Ja, D. Harwood and L.S. Davis, Parallel algorithms for image enhancement and segmentation by region growing with an experimental study, *The Journal of Supercomputing* **10**(2) (1996), 141–168.