The Design, Implementation, and Evaluation of Scalable Reliable ACO

Amir Ghaffari

January 2015

1 Introduction

Ant Colony Optimization (ACO) is a metaheuristic that has proved to be successful a variety of combinatorial optimization problems such as scheduling problems, routing problems, and allocation problems [1, 2, 3, 4].

The Single Machine Total Weighted Tardiness Problem (SMTWTP) is a well-known scheduling problem which aims to find out the starting times for a given set of jobs on a single processor to minimize the weighted tardiness of the jobs with respect to given due dates [5]. Different heuristic methods have been developed to solve benchmark instances for the SMTWTP successfully [6]. Ant colony optimization is one of the approach that has been applied for the single machine total tardiness problem [7, 8, 9].

We improve the scalability and reliability of an existing distributed Erlang version of ACO developed by Dr Kenneth Mackenzie from Glasgow University. Section 2 describes the existing two-level version of ACO. Section 3 develops a multi-level version of distributed ACO. A comparison of the two-level and multi-level versions are given in Section 4. A reliable version of ACO is introduced in Section 5. We employ SD Erlang to improve the scalability of the reliable version in Section 6. Section 7 investigates how SD Erlang reduces the network traffic.

2 Two-Level Distributed ACO (TL-ACO)

The existing distributed ACO developed in Erlang has a two-level design and implementation. Figure 1 depicts the process and node placements of the *TL-ACO* in a cluster with N_C nodes. The master process spawns N_C colony processes on available nodes. In the next step, each colony process spawns N_A ant processes on the local node. In the figure, all captions and their relevant object have the same color. As the arrows show, communications between the master process and colonies are bidirectional. There are I_M communications between the master process and a colony process. Also, I_A

bidirectional communications are done between a colony process and an ant process.

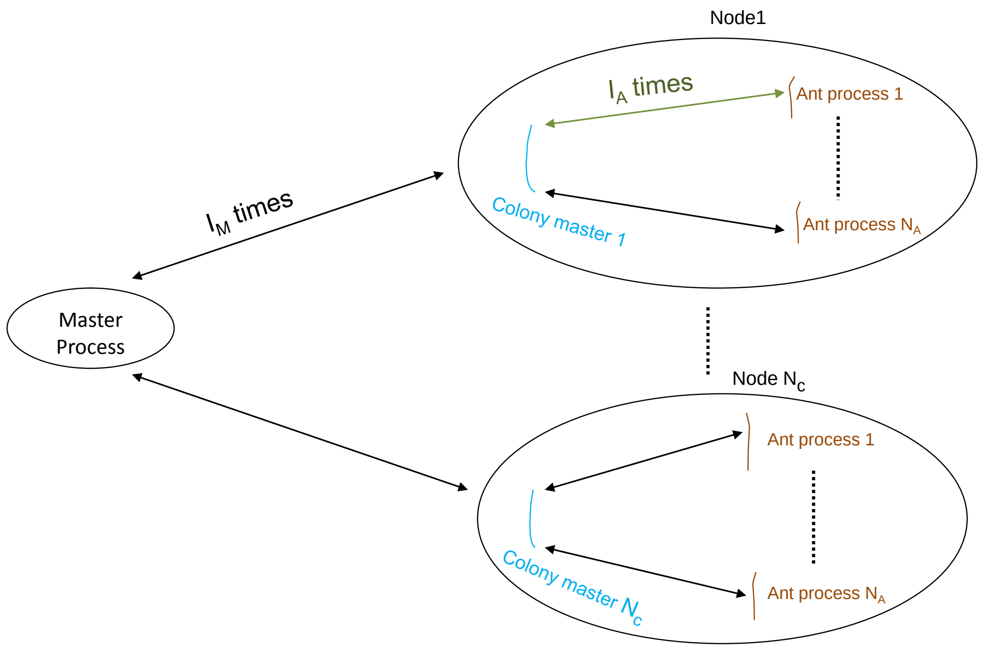


Figure 1: Two-Level Distributed ACO

3 Multi-Level Distributed ACO (ML-ACO)

In *TL-ACO* with a large number of colony nodes the master process, which is responsible of collecting and processing the result form colonies, can become overloaded and a bottleneck for scalability. As a solution to this problem, we propose a Multi-Level design for distributed ACO (*ML-ACO*), in which, in addition to the master node, there can be multiple levels of sub-master nodes to help the master through sharing the burden of collecting and processing the results from colony nodes (Figure 2). The number of sub-master nodes is adjustable based on the number colony nodes in a way that each sub-master node handles reasonable amount of loads.

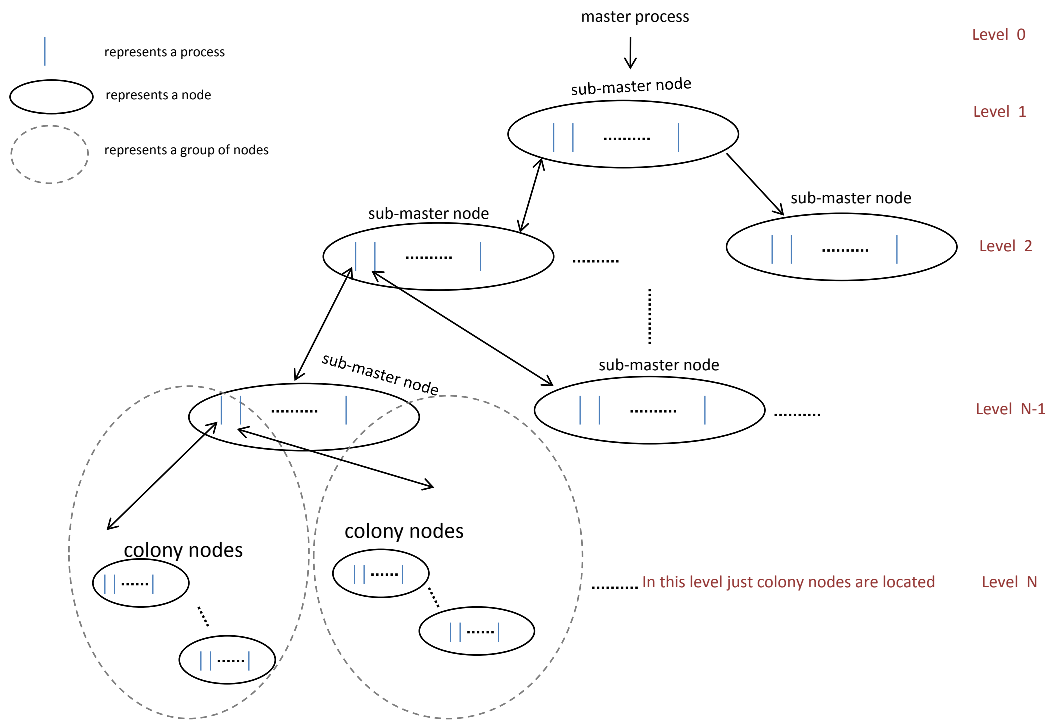


Figure 2: Node Placement in Multi Level Distributed ACO

Figure 3 depicts the process placement in ML-ACO. If there are P processes per each sub-master node, then number of processes in level N is P^N and number of node in level N is P^{N-1}. A process in level L creates and monitors P processes on a node at level $L + 1$. However, the last level is an exception because in the last level just colony nodes are located and one colony process per each colony node exists. Thus, a process in level *N-1* (one level prior the last) is responsible for P node in level N (not P processes) and consequently the number of node in level N (the last level) is P^N.

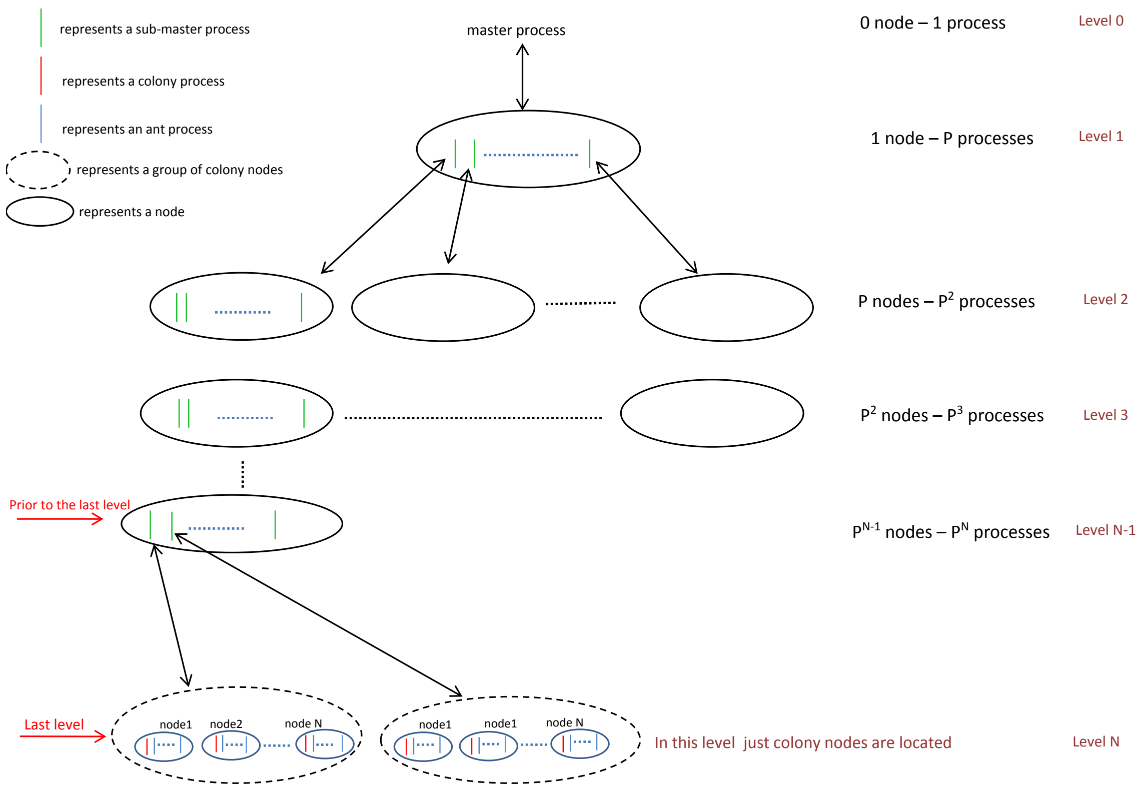


Figure 3: Process Placement in Multi Level ACO

To be able to create a multi-level tree of sub-master node, we need to formulate the relation of the number of processes, nodes and levels. If the number of processes on each node is P and the number of all available nodes is N, then to find out the number of levels (X) that the tree will have, we need to find the maximum of X in this equation: $1{+}P{+}P^2{+}P^3{+}...{+}P^{X-2}{+}P^X \leq N$. For example, if P=5 and N=150 then the tree will have 3 levels as shown here: 1+5+5^3 $\leq$150. This means we can just use 131 nodes of 150 nodes.

4 Scalability of ML-ACO versus TL-ACO

To compare the scalability of TL-ACO and ML-ACO, the following parameters are considered in our measurement:

- Input Size: our measurements with different sizes of input show that increasing the input's size just increases the loads on ants processes. For larger input size, ant processes on colony nodes need more time to find their best solution, and it does not affect the master process considerably. In other words, a larger input leads to increase in the size of messages that the master process receives and not the number of messages that it receives.

- Number of Iterations: There are two kinds of iteration in the system: local and global iteration. Local iteration represents the number of times that ant processes communicate with their colony process. Global iteration is the number of times that colonies communicate their best solution with the master process. As the master process handles iterations sequentially, increasing the number of iterations does not make the master process overloaded.

- Number of Colonies: In TL-ACO the main parameter that could lead to an overloaded master process is the number of colony nodes. For a large number of colony nodes, the master process could become overloaded and consequently a bottleneck for scalability. The maximum number of nodes that we could compare both versions is 630 nodes (10 VMS on 63 hosts at Tintin cluster [10]). Our results show that both versions, i.e. TL-ACO and ML-ACO, take roughly the same time to finish. In our measurement, there is 629 colonies on 630-node cluster, and so in TL-ACO version the master process takes 629 messages from its colony nodes in each iteration. The number of received messages (i.e. 629 messages) is not large enough to make the master process overloaded.

Simulation

To investigate the scalability of TL-ACO and ML-ACO we need a large cluster with thousands of nodes and tens of thousands of Erlang VMs. Since we have not access to such a large cluster, we develop a version in which each colony node sends more than one message to its parent instead of sending only one message. This approach simulates a large cluster of colony nodes because each colony can play the role of multiple colonies.

Figure 4 compares the scalability of TL-ACO and ML-ACO up to 112,500 simulated nodes. The figure presents the middle value of 5 executions and vertical lines show the variation. ML-ACO outperforms TL-ACO beyond 45k simulated nodes (approximately 50K nodes). The measurement is run on GPG cluster with 226 Erlang VMs, 1 VM is dedicated for master and sub-master processes and 225 VMs run colonies [11]. There are 4 ant processes per colony. I_M and I_N (global and local iterations) are 20. Input size is 40, and each colony node sends 100, 200, 300, 400, and 500 messages to its parent. Figures 5, 6 depict node placements in TL-ACO and ML-ACO respectively. In TL-ACO the master process is singly responsible for collecting results from its children (225 nodes) (Figure 5), but in ML-ACO the master process collects the results from 15 sub-master processes and each sub-master process is responsible for 15 colony nodes which are depicted in dotted circles in Figure 6.

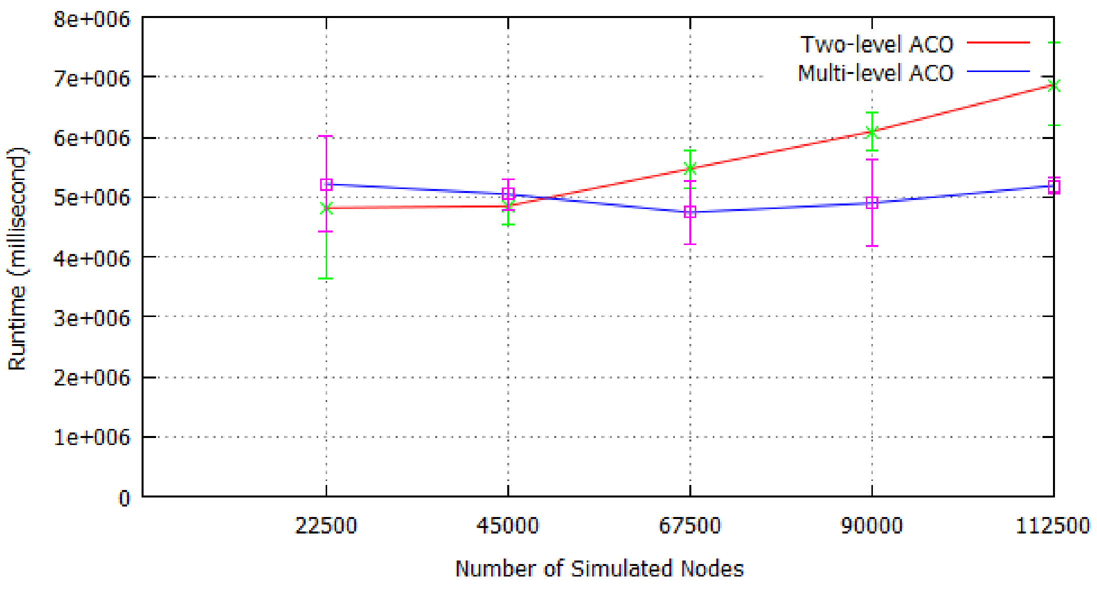


Figure 4: Scalability of ML-ACO and TL-ACO

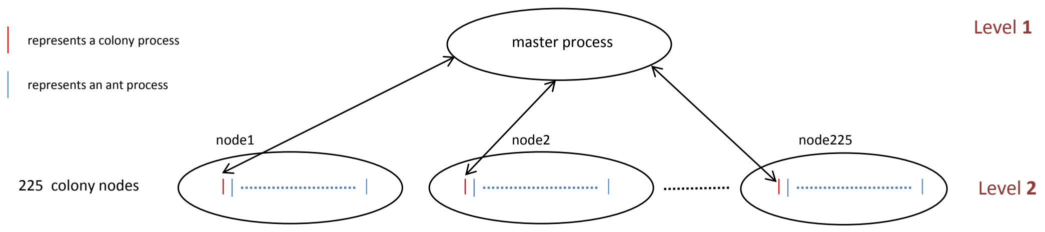


Figure 5: TL-ACO with 225 colony nodes

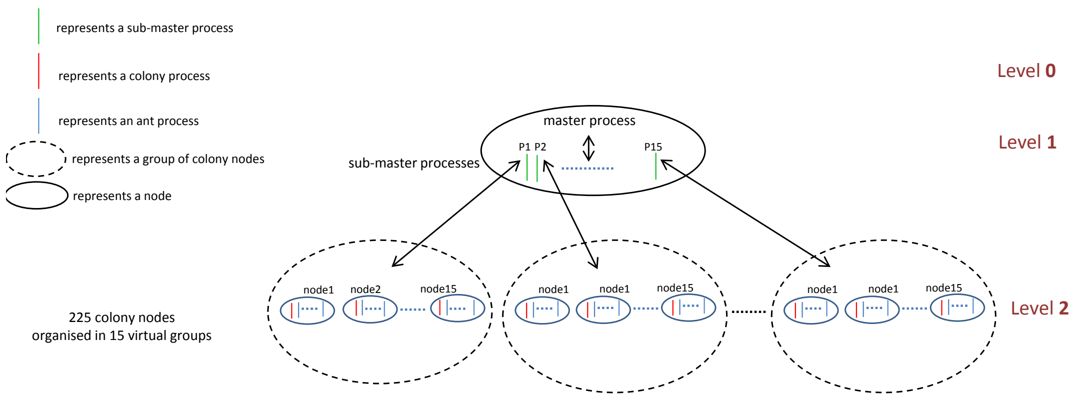

represents a sub-master process
represents a colony process
represents an ant process
represents a group of colony nodes
represents a node
Level 0
Level 1
Level 2
master process
P1 P2
P15
sub-master processes
node1
node2
node15
225 colony nodes organised in 15 virtual groups

Figure 6: ML-ACO with 225 colony nodes

5 Reliable ACO (R-ACO)

There are four kinds of processes in the ML-ACO version, i.e. Master, Sub-masters, Colonies, and Ants (Figure 3). To make the version fault-tolerant, we need to consider the possibility of failure of each kind individually.

Sub-master processes are created recursively to form a multi-level tree structure. The processes on the root node of the tree are created by the master process. Processes on the other levels are created recursively by processes on the prior levels. Thus, the master process will monitor the processes on the root node and each level processes monitor the processes on the next level. The level prior to the last level creates colonies and also collects the results from them. Thus, processes on this level can supervise all the colonies and restart them in case of failure.

Each colony creates and also collects the results from its own ant processes. Thus, a colony supervises all its own ant processes.

Ant processes are the last level, and so they do not need to supervise any other processes.

The master process can be created by the *starter process*, a process that starts the computation by creating and supervising the master process. Since the starter process is not involved in computational process over the time, it will not crash and so it can safely monitor the master process. This process can be considered as a user shell process that its duty is just running the master process, and finally printing the result.

In Erlang, if a process dies and restarts again, it will get a new process identifier (Pid). In a fault-tolerant application, other processes should be able to reach that process after its restart. Thus, we need to refer to a process by its name instead of its Pid for reliability purposes. In reliable ACO, the total number of required registered names for an L levels tree with P processes on each node is $P + P^2 + P^3 + ... + P^{L-1} + P^L = \frac{P^{L+1}-P}{P-1}$.

For example, in a tree with 3 levels and 5 sub-master processes on each node, we need 155 global name registrations: $\frac{5^{3+1}-5}{5-1} = 155$ (Figure 7).

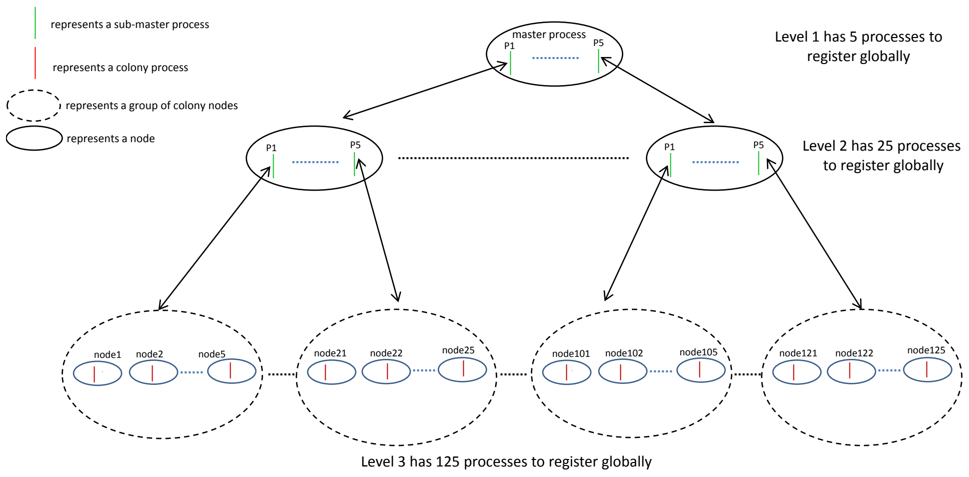

represents a sub-master process
represents a colony process
represents a group of colony nodes
represents a node
master process
P1
P5
Level 1 has 5 processes to register globally
Level 2 has 25 processes to register globally
node1
node2
node5
node21
node22
node25
node101
node102
node105
node121
node122
node125
Level 3 has 125 processes to register globally

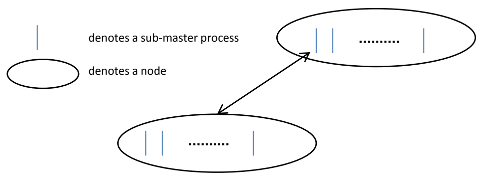
Figure 7: Required name registrations in a tree with 3 levels and degree 5

The number of name re-registration depends on the failure ratio. There will be one name re-registration per each process failure and P re-registration after a node recovery.

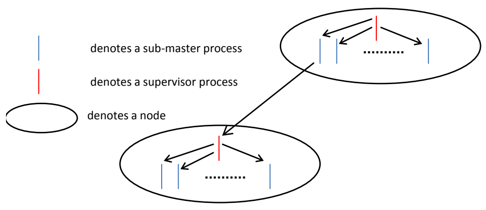
Optimizing Reliable ACO

In the given design for reliable ACO in previous section, a sub-master process supervises all its child processes (Figure 8a). For example, if a sub-master process supervises 10 sub-master processes located on a remote node, 10 global name registrations are needed for reliability purposes. To minimize the number of required global name registrations, we can have a local supervisor. As Figure 8b shows, in the local supervisor model a sub-master process only supervises a local supervisor process on a remote node, and the local supervisor process is responsible for supervising the sub-master processes located on its locale node. This approach reduces the number of required global name registration form P, where P is the number of sub-master process per node, to 1. Thus, in the optimized reliable ACO, the total number of global name registrations for an L levels tree with P processes on each node is $1 + P^1 + P^2 + ... + P^{L-2} + P^L$.

For example, in a tree with 3 levels and 5 sub-master processes on each node, we need 131 global name registrations: $1 + 5^1 + 5^3 = 131$ (Figure 9). This optimization reduces the number of required name registrations for this example from 155 to 131.



(a) Remote Supervisor



(b) Local Supervisor

Figure 8: Local Supervisor vs. Remote Supervisor

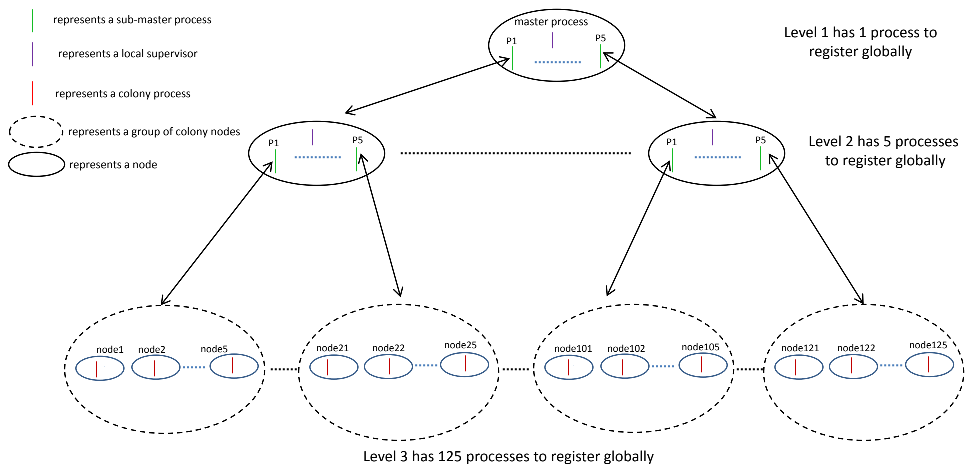

represents a sub-master process
represents a local supervisor
represents a colony process
represents a group of colony nodes
represents a node
master process
P1
P5
Level 1 has 1 process to register globally
Level 2 has 5 processes to register globally
node1
node2
node5
node21
node22
node25
node101
node102
node105
node121
node122
node125
Level 3 has 125 processes to register globally

Figure 9: Optimized name registrations in a tree with 3 levels and degree 5

Reliability Evaluation

To evaluate the reliability of ACO, we employ Chaos Monkey [12, 13]. Chaos Monkey kills available Erlang processes on a node randomly by sending an exit signal to the processes over its execution time. We ran Chaos Monkey on all computational nodes, i.e. master, sub-masters, and colonies. The results show that the reliable ACO could survive after all failures that Chaos Monkey caused. Victim processes include all kind of processes, i.e. the master, sub-masters, colonies, and ants regardless whether they are initial processes or recovered ones.

Scalability of R-ACO versus ML-ACO

Figure 10 compares the scalability of reliable (R-ACO) and unreliable ACO (ML-ACO) up to 145 Erlang nodes. The measurements are run on the GPG cluster with 20 hosts, and the number of VMs per host varies according to the cluster size. 1 VM is dedicated for the master and sub-master processes and the other VMs run colonies. There are 4 ant processes per colony. I_M and I_N (global and local iterations) are 20 and input size is 100. The scalability of R-ACO is worse than ML-ACO due to global name registration. As the cluster size grows, the runtime of R-ACO version increases with a considerably faster pace in comparison with the ML-ACO. This is because the latency of global name registration increases dramatically as cluster grows [14]. Moreover, as discussed in Section 5, more global name registrations is required as the number of colony nodes increases. As a solution to this problem, we develop an SD-Erlang version of reliable ACO in the next section.

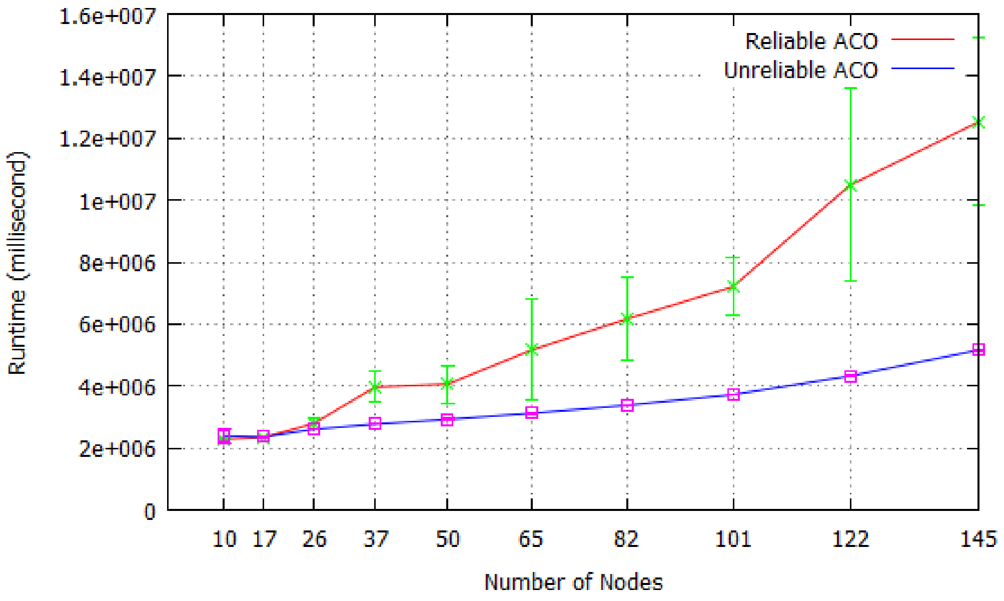


Figure 10: Scalability of Reliable vs. Unreliable ACO

6 Scalable Reliable ACO (SR-ACO)

This section investigates whether SD Erlang can improve the scalability of reliable distributed Erlang systems. We employ SD-Erlang to improve the scalability of reliable ACO by replacing global name registration with group name registration. In SR-ACO, global name registrations are replaced with group name registrations. In addition to this, SD-ACO has a non-fully connected model in which nodes are only connected to the nodes in their own group (*s_group*), and no unnecessary connection is made. In SR-ACO, nodes only need to communicate within their own group, and there is no need for a node to be connected to the nodes that belong to other groups. Figure 11 depicts s_groups organisation in SR-ACO. The master node belongs to all s_groups, however, colony nodes are grouped into a number of s_groups.

Figure 12 compares the weak scalability of scalable reliable (SR-ACO), reliable (R-ACO) and unreliable (ML-ACO) versions. The results show that SR-ACO scales much better than R-ACO because of replacing global name registrations with s_group name registrations. Surprisingly, we see from the figure that SR-ACO scales even better than the unreliable ACO. Since there is no name registration in unreliable version, this better performance could be because of the reduction in the number of connections between the nodes. This shows that reducing the number of connections between nodes, could lead to a better scalability and performance. Next section investigates the impact of SD Erlang on the network traffic.

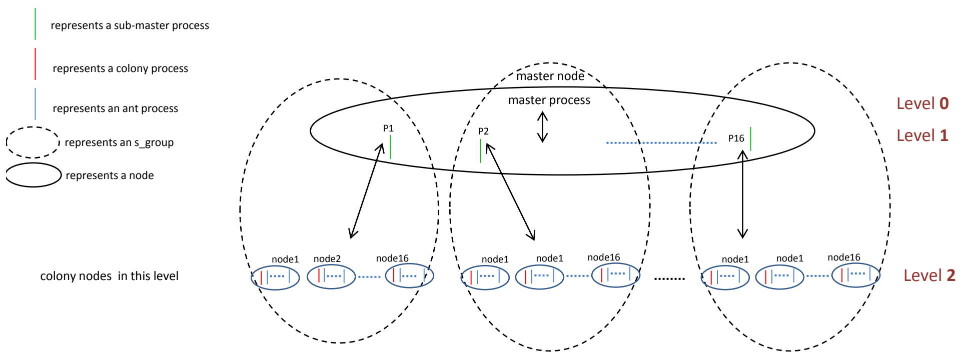

represents a sub-master process
represents a colony process
represents an ant process
represents an s_group
represents a node
colony nodes in this level
master node
master process
P1
P2
P16
Level 0
Level 1
Level 2
node1
node2
node16

Figure 11: S_Groups Organisation in Scalable Reliable ACO (SR-ACO)

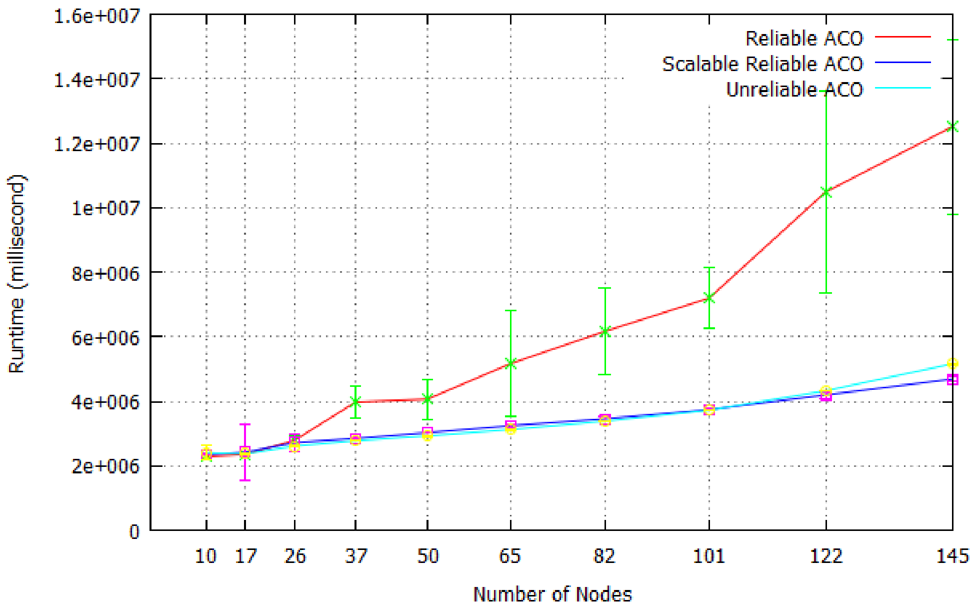


Figure 12: Weak Scalability of SR-ACO, R-ACO, and ML-ACO

7 Network Traffic

To investigate the impact of SD Erlang on the network traffic, we measure the number of sent and received packets for the three versions of ACO, i.e. ML-ACO, R-ACO, and SR-ACO. Figures 13 and 14 show the total number of sent and received packets for the measurement of Section 6 (Figure 12). The highest traffic (the red line) belongs to the reliable ACO (R-ACO) and the lowest traffic belongs to the scalable reliable ACO (SR-ACO). We observe that SD Erlang reduces the network traffic between the Erlang nodes in a cluster effectively. Even with the group name registration in SR-ACO, SD Erlang reduces the network traffic of SR-ACO more than unreliable ACO (ML-ACO) in which no global or group name registration is used.

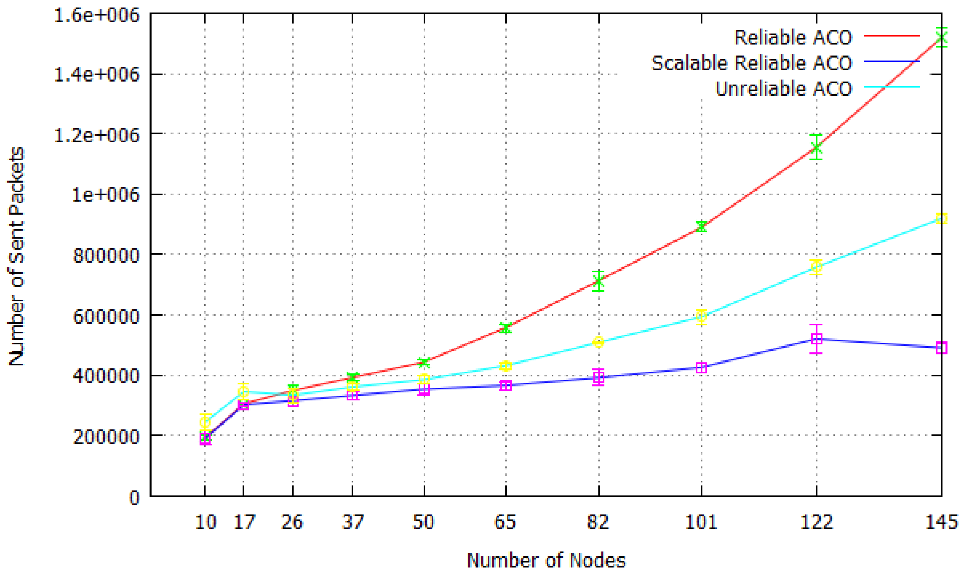


Figure 13: Number of Sent Packets in SR-ACO, Reliable ACO, and Unreliable ACO

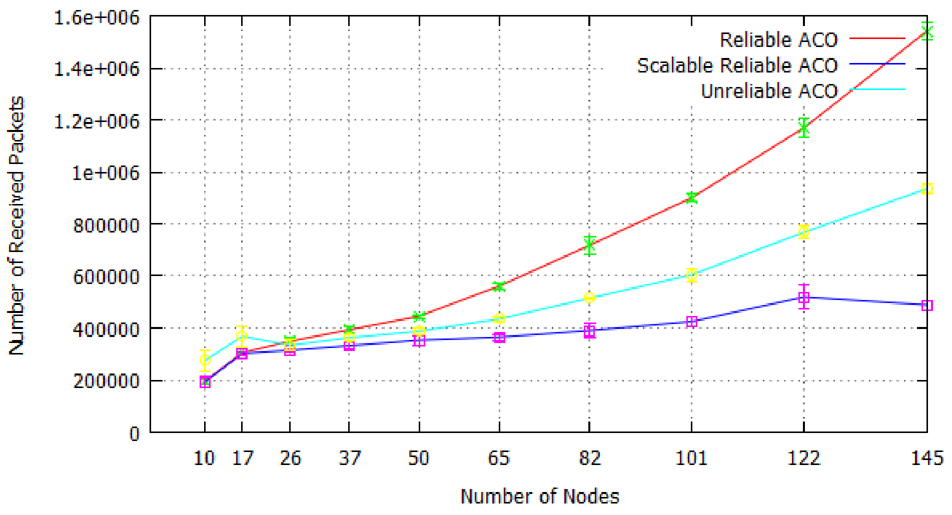


Figure 14: Number of Received Packets in SR-ACO, Reliable ACO, and Unreliable ACO

References

[1] Robert McNaughton. Scheduling with Deadlines and Loss Functions. *Management Science*, 6(1):1–12, 1959.

[2] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Company Scituate, Scituate, MA, USA, 2004.

[3] Gianpaolo Ghiani, Demetrio Lagana, Gilbert Laporte, and Francesco Mari. Ant colony optimization for the arc routing problem with intermediate facilities under capacity and length restrictions. *ournal of Heuristics*, 16(2):211–233, 2010.

[4] Wenzhu Liao, Ershun Pan, and Lifeng Xi. A heuristics method based on ant colony optimisation for redundancy allocation problems. *International Journal of Computer Applications in Technology*, 40(1):71–78, 2011.

[5] Martin Josef Geiger. On heuristic search for the single machine total weighted tardiness problem, Some theoretical insights and their empirical verification. *European Journal of Operational Research*, 207(3): 1235–1243, 2010.

[6] T.S. Abdul-Razaq, C.N. Potts, and L.N. Van Wassenhove. A survey of algorithms for the single machine total weighted tardiness scheduling problem. *Discrete Applied Mathematics*, 26(2):235–253, 1990.

[7] Andreas Bauer, Bernd Bullnheimer, Richard F. Hartl, and Christine Strauss. Minimizing Total Tardiness on a Single Machine Using Ant Colony Optimization. *Central European Journal of Operations Research*, 8:125–141, 2000.

[8] Matthijs den Besten, Thomas Stutzle, and Marco Dorigo. Ant Colony Optimization for the Total Weighted Tardiness Problem. *Parallel Problem Solving from Nature*, 1917:611–620, 2000.

[9] Daniel Merkle and Martin Middendorf. An Ant Algorithm with a New Pheromone Evaluation Rule for Total Tardiness Problems. *Real-World Applications of Evolutionary Computing*, 1803:290–299, 2000.

[10] SNIC-UPPMAX. The Kalkyl Cluster, 2012. URL http://www.uppmax.uu.se/hardware.

[11] Phil Trinder. GPG Cluster, 2014. URL http://www.dcs.gla.ac.uk/research/gpg/cluster.htm.

[12] Daniel Luna. Chaos Monkey, 2012. URL https://github.com/dLuna/chaos_monkey.

[13] Todd Hof. Netflix: Continually Test by Failing Servers with Chaos Monkey, 2010. URL http://highscalability.com/blog/2010/12/28/netflix-continually-test-by-failing-servers-with-chaos-monke.html.

[14] Amir Ghaffari. Investigating the Scalability Limits of Distributed Erlang. In *Proceedings of the Thirteenth ACM SIGPLAN Erlang Workshop*, Gothenburg, 2014.