In: Proc. 17th IEEE International Conf. on Advanced Robotics (ICAR 2015), Istanbul, Turkey (2015).

Dynamic process migration in heterogeneous ROS-based environments

José Cano
School of Informatics
University of Edinburgh
jcanore@inf.ed.ac.uk

Eduardo Molinos
Department of Electronics
University of Alcalá
eduardo.molinos@edu.uah.es

Vijay Nagarajan
School of Informatics
University of Edinburgh
vijay.nagarajan@ed.ac.uk

Sethu Vijayakumar
School of Informatics
University of Edinburgh
sethu.vijayakumar@ed.ac.uk

***Abstract*—In distributed (mobile) robotics environments, the different computing substrates offer flexible resource allocation options to perform computations that implement an overall system goal. The AnyScale concept that we introduce and describe in this paper exploits this redundancy by dynamically allocating tasks to appropriate substrates (or scales) to optimize some level of system performance while migrating others depending on current resource and performance parameters. In this paper, we demonstrate this concept with a general ROS-based infrastructure that solves the task allocation problem by optimising the system performance while correctly reacting to unpredictable events at the same time. Assignment decisions are based on a characterisation of the static/dynamic parameters that represent the system and its interaction with the environment. We instantiate our infrastructure on a case study application, in which a mobile robot navigates along the floor of a building trying to reach a predefined goal. Experimental validation demonstrates more robust performance (around a third improvement in metrics) under the Anyscale implementation framework.**

I. Introduction and Motivation

Mobile robotics systems are typically distributed and heterogeneous, composed of multiple devices with different computation capabilities. Indeed mobile robots have on-board computers, but typically are also connected to other more powerful stand-alone computers, or even clusters. In such situations, it would be desirable to utilise each of the computational devices (or scales) available in the most optimal way possible. However, the problem of deciding what task to run at what scale is a challenging one. This is because, in mobile robotics systems, there is a need to react to events (some of which might be unexpected). Under these situations, the system should be able to adapt its behaviour without jeopardising the security of objects, people and the system itself. Therefore, the tasks that need to be performed tend to be decided at runtime, and in addition the environment is not fixed, but changing. In other words, the required task-to-scale mapping is not a static one, but can vary dynamically.

For example, let us consider a mobile robot performing object recognition. Further, let us assume the following: i) the current task for image processing runs on the robot's on-board computer; ii) however, the robot's on-board computer is not able to process the image stream at the optimal resolution for the required accuracy; iii) the robot is wirelessly connected to another computer that is able to process the stream with the optimal resolution. In such an environment, it would be interesting to migrate the image processing task to the second computer in order to achieve the required recognition accuracy. However, if the connection with the remote computer is lost, it is better to migrate back the image processing task again to the robot despite the lower recognition accuracy.

978-1-4673-7509-2/15/$31.00 2015 IEEE

In scenarios like the previous one, the AnyScale concept proposes to take advantage (when possible) of the computing resources offered by the different devices/scales available in the system in order to optimise the application behaviour based on some specific requirement(s). The key question therefore is to automatically decide *where* and *when* the computation should take place. Hence, there must be a runtime mechanism for deciding how to assign the on-going tasks to the available scales based on the resource availability and the cost-benefit trade-off related to each decision.

In this paper, we demonstrate the AnyScale concept with a general and automatic infrastructure that solves the derived task allocation problem. Our infrastructure consists of two components. First, an offline component that models the application, the scales, and the network, producing initial allowable mappings (and pruning out mappings that need not be considered). Second, an online component which dynamically produces the final mapping based on the current environment. The selected framework to develop our infrastructure is ROS (Robot Operating System) [1], as it provides hardware abstraction and transparent distributed communication – thus, an ideal candidate for realising the the proposed multi-scale concept.

Finally, as a proof of concept, we instantiate our infrastructure in a real test-bed scenario consisting of a simple ROS application, in which a mobile robot has to navigate around the floor of an office building from a predefined initial position until it reaches another predefined final position. There are two scales available: the robot's on-board computer and an external laptop connected to the robot via wireless. The specific runtime mechanism of this case study takes decisions based on the static/dynamic parameters that represent the system and its interaction with the environment, with the objective of optimising the total time required to complete the path but always guaranteeing the system correctness.

The contributions of the paper are as follows: i) We propose the AnyScale concept for heterogeneous mobile robotics systems, ii) We demonstrate the concept with our ROS-based infrastructure that solves the task allocation problem, iii) We evaluate our infrastructure with a real case study. We also validate our task allocation solution through simulation.

II. Related work

Since our approach is based on dynamic task allocation/scheduling, in this section we review previous works addressing this problem for both multi-robot and general distributed systems, highlighting the differences found.

Beginning with multi-robot task allocation (MRTA), in [2] a taxonomy that categorises types of problems based on robot type, task type, and allocation type is presented. [3] is an extension of the previous taxonomy that adds the degree of interdependence of agent-task utilities. [4] also extends the first taxonomy assessing the problems under three criteria: task model, solution model, and magnitudes used in the algorithm's cost function. The three classifications are generic enough to be applied to systems including robotics and non-robotics agents. Combining them, our system will fit: multi-task robots (MT), single-robot tasks (SR), instantaneous assignment (IA), Cross-schedule Dependencies (XD), and heterogeneous. However, even combining the three taxonomies there are aspects not covered or not clearly discussed, although addressed in our system. For example, it is unclear how the system behaviour in dynamic environments is modelled. Moreover, there are types of constraints not considered, such as connection deadlines, mobility interferences, network coverage, etc.

Lerman et al. [5] propose a dynamic task assignment mechanism based on using local observations of the environment. The proposal is well modelled mathematically but it does not account for system heterogeneity. In [6], a methodology for task allocation in heterogeneous systems is presented. The allocation algorithm is based on the turnaround time task-robot. However, it does not take into account the resource availability in the robots. Luo et al. [7] [8] address task assignment subject to restrictions such as task deadlines and robots' resource budget. In the first case, tasks are independent and have identical duration, which solves a subset of our problem. In the second case, the budget is related only to energy, which does not represent all the robot's capacities. In addition, both solutions assume perfect connections between robots. Finally, [9] proposes two distributed market-based algorithms where N tasks must be assigned to N identical robots, and where each robot can perform only one task. Again, this solution partially covers our approach.

Regarding distributed systems, Macarthur et al. [10] introduce a distributed algorithm for multi-agent task allocation problems. The dynamic environment only assumes a variable set of tasks and agents over time, but does not consider the variable resource availability given a fixed number of task/agents or that restrictions between tasks can also change over time. Page et al. [11] developed a task scheduler for dynamic heterogeneous systems based on a multi-heuristic genetic algorithm. The weak points are that tasks are independent and the scheduler does not take advantage of the capacities of the target system. [12] also proposes a heuristic-based genetic algorithm for task scheduling where there is a processor in charge of assigning the tasks. However, the solution assumes a homogeneous multiprocessor system and negligible communication costs among tasks. Finally, in [13] a hybrid heuristic-genetic scheduling algorithm for heterogeneous multi-core systems is proposed. The solution assumes a fully-connected system, which is less restrictive than our proposal. Moreover, task allocation is static.

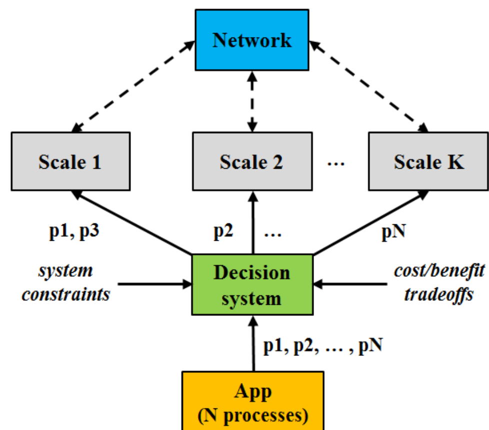

Fig. 1. AnyScale computing concept. Dynamic scenario where each scale represents a unique machine/device with different capabilities.

III. Our Approach

In this section, we first present the AnyScale computing concept, and the task allocation problem that it leads to. We then describe the static (offline) analysis required for characterising the ROS system on which the concept can be applied. Finally, we propose a dynamic (online) algorithm as a general solution to the task allocation problem.

A. *AnyScale computing concept*

The general system we want to analyse is composed of N processes and K scales, where the N processes together define the application of interest, and each scale corresponds to a unique machine/device that is able to run processes. We assume the following: i) each scale has different capabilities (e.g. CPU clock rate, memory); ii) a changing environment.

Given this system, it is quite apparent that depending on the specific mapping of processes to scales considered, the behaviour of the system may vary. Our overarching goal is to leverage the available scales in order to optimise some specific system behaviour. This will imply to dynamically discover the set of mappings that is able to meet that expected behaviour. Figure 1 shows a high-level diagram representing the AnyScale concept, where the two key points to understand are:

- Any application could be programmed in a **scale-agnostic** manner; in other words, the programmer need not take into consideration neither the number of scales available, nor their capabilities.
- During runtime, however, a decision system dynamically provides **scale-specific** assignments for every process with the objective of optimising the system behaviour in a specific way.

Therefore, the AnyScale computing concept applied to ROS basically consists of defining and implementing a decision system that is able to dynamically allocate ROS nodes to the available scales, with the objective of customising/adapting the behaviour of applications. This customisation/adaptation must satisfy the system constraints and will be based on a cost function that may be optimised for (a combination of) different goals such as performance, accuracy, etc.

B. System characterisation

The purpose of performing an offline characterisation of the ROS system we want to work with is twofold. First, to define the parameters representing the system and its interaction with the environment. Since we are considering a changing environment, we must characterise static and dynamic parameters. Second, based on these parameters, to concretely define the decision problem, specifying the cost function and the system constraints. Note that this offline analysis will allow to prune mappings that always violate some constraint(s), that is, those that are not able to meet the expected system behaviour.

1) Static parameters:

Hardware: The two hardware components in our system are the K scales and the communication network. We consider one of the scales as the **master scale**, as it is the scale in which the decision system will run. The main parameters defining each scale are the maximum CPU clock rate and the total memory. The network is defined by its technology, which determines the maximum bandwidth available.

Application: The processes (nodes) that compose a ROS application and their underlying communication flows define the computation graph (Figure 2 shows an example). While this graph gives an overview of the system, we also need to know what the requirements for both nodes and connection between nodes are, in order to make the system work as expected.

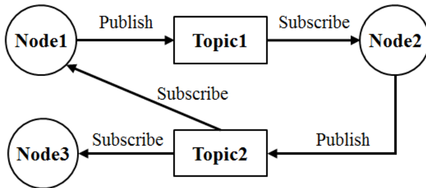

Fig. 2. Example of ROS computation graph (publisher/subscriber model).

Node requirements: Using common Linux monitoring utilities such as *top*, we can characterise the average percentage of CPU required for each node at each scale (CPU_usage_{nk}) throughout the application execution. Note that the memory required for each node will be the same for all scales.

Connection requirements: In ROS, messages are published to topics at specific frequencies. It is worth noting that these frequencies impose restrictions, because indirectly they set a deadline that connections using the associated topics must satisfy in order to have a system working as expected. If for some reason, an established frequency cannot be guaranteed, the system might behave wrongly or even become faulty.

Nevertheless, some connections are more tolerant to occasional brief periods of network outage, without significantly impacting the overall system behaviour. We model the connection tolerance Δ as the period of time during which the associated frequency could be lower than the initially established threshold value (note that during this grace period, the frequency can even degrade to 0, in which case the subscriber does not receive any data). Thus, every connection between two nodes is characterised by its associated topic (defines the message type), the threshold frequency, and its tolerance.

Finally, note that establishing a reasonable threshold frequency to every connection requires offline analysis. Basically, we initialise the frequency values for every connection to zero; then, we iteratively increment each value and check if the system behaves correctly or not. When the system starts to behave correctly, the corresponding threshold is apparent.

Critical nodes: Based on the previous definition for connections, if a node participates in at least one connection whose associated $\Delta = 0$, we define it as a critical node. Examples are nodes that provide as output the data read from sensors as input, or nodes that need to permanently be connected to robot actuators. We establish that critical nodes have to be pinned to the master scale (cannot be migrated) because is quite likely that the network connection cannot guarantee a constant threshold frequency. Therefore, the critical and migratable nodes can be easily derived from the previous static analysis for connections. Moreover, as the list of critical nodes is fixed, we may prune many mappings permanently.

2) Dynamic parameters: These represent the value of some (a subset) static parameters at a given moment in time – more specifically: the current percentage of CPU usage for each scale, the current network state and available bandwidth, and the currently observed frequency values for each of the network connections and their observed tolerances. Again, we can leverage Linux monitoring utilities to know their values.

3) Cost function: It provides the cost/benefit derived from each static mapping. As an example, we instantiate the cost function to maximise the system performance – note that it can be application specific, e.g. the total time to complete one full cycle of a sensory-motion update routine. In our general case, the performance for a static mapping is determined by the CPU usage of each node at its allocated scale and the gain factor α_k of each scale (note that a more powerful scale will imply a greater α_k, and values may differ for every system):

$$Perf_{map} = \sum_{n=1}^{N} \sum_{k=1}^{K} CPU_usage_{nk} * \alpha_k \quad (1)$$

Therefore, the overall (dynamic) performance is just the sum of the values for every mapping selected by the decision system multiplied by the time each mapping was set (the total time should be fixed), and our objective is to maximise it:

$$max \sum_{map=1}^{M} T_{map} * Perf_{map} \quad (2)$$

4) System constraints: The system is said to be correct if it satisfies the following three types of constraints: i) the frequency/tolerance restrictions imposed by the node connections; ii) the capacity restriction imposed by the current percentage of CPU usage of each scale; iii) the capacity restriction imposed by the current network bandwidth available. Every time any of these constraints is not satisfied, actions must be taken to revert to a correct system again – e.g. migrating back to the master scale nodes currently running in remote scales.

C. Dynamic allocation algorithm

Now we describe how to use the previous characterisation parameters in order to take dynamic decisions trying to

Algorithm 1: Dynamic node allocation

Input: $scale_1$, ..., $scale_K$, $node_1$, ..., $node_N$
while *App_running* **do**
  if $required_{BW} > network_{BW}$ || $freq, \Delta$ *!satisfied* **then**
    do
      select_right $(node_i, scale_j)$ from $migrated_list$
      migrate_back $node_i$
      update $cost_function$
    while $required_{BW} > network_{BW}$;
  else
    $migrated = false$
    $node_list =$ sort $migratable_nodes$ by CPU_usage
    $scale_list =$ sort $remote_scales$ by CPU_freq
    select_first $node_i$ from $node_list$
    select_first $scale_j$ from $scale_list$
    while $node_i \neq \emptyset$ && $migrated = false$ **do**
      if $node_i$ *!migrated_back in* T && $required_{BW} \leq network_{BW}$ **then**
        if $scale_j$ *can_allocate* $node_i$ **then**
          $migrated_list \leftarrow (node_i, scale_j)$
          $migrated = true$
          update $cost_function$
        else
          select_next $scale_j$ from $scales_list$
          if $scale_j = \emptyset$ **then**
            select_first $scale_j$ from $scale_list$
            select_next $node_i$ from $nodes_list$
      else
        select_next $node_i$ from $nodes_list$
  sleep (time)

maximise the system performance while always satisfying the system constraints. Note that we need to set an initial system configuration. In our case, this corresponds to running all N nodes in the master scale, which satisfies the system constraints trivially but is the worst-case scenario for performance.

Our decision system uses a simple heuristic that attempts to allocate the most CPU intensive nodes to the most powerful scales. It is worth noting that this heuristic provides optimal performance values when the number of possible mappings is small (e.g. the case study). However, for the general case, where the number of possible mappings can be huge, it provides values close to the optimal (we estimated by simulation an average deviation of 10-15%). We leave improving the general case for future work. Algorithm 1 shows the dynamic allocation process. Basically, the iterative procedure sorts the candidate nodes and the remote scales based on CPU parameters. Then, it migrates the selected node to the selected scale if the corresponding constraints are satisfied. In case of violating some system constraint(s), the corresponding node(s) running remotely must be migrated back to the master scale.

IV. TEST-BED SCENARIO

We evaluate the AnyScale concept on a test-bed running a real robotics application. The test-bed and application are characterised as follows. The test-bed is composed of two scales and eleven ROS nodes. The master scale corresponds to the KUKA YouBot [14] mobile robot, which includes an on-board computer (Intel Atom, 2 cores 1.6 GHz, 2GB RAM), and the remote scale corresponds to a laptop computer (Intel Core i7, 4 cores 2.8 GHz, 4GB RAM). Both scales communicate through a dedicated WiFi-based network (802.11g, 54 Mbps).

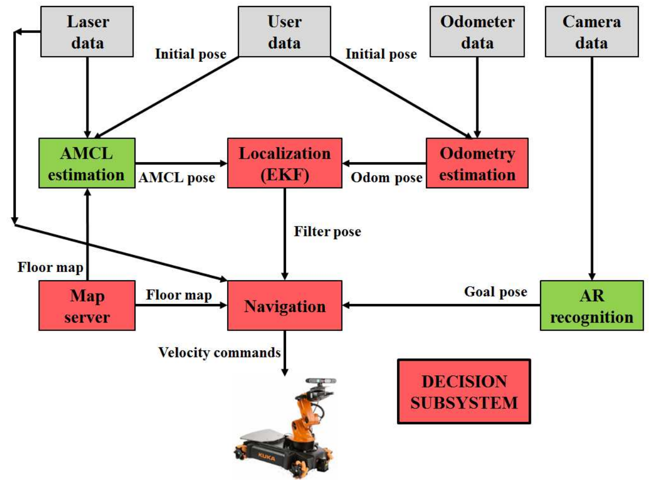

Fig. 3. System flow diagram: ROS nodes (providing data [grey], critical [red], and migratable [green]) and connections

Figure 3 shows the flow diagram of the system (the ROS computation graph is derived from it), where we can see the nodes and their connections. Data inputs come from three sensors (laser from Hokuyo URG-04LX, odometry from wheels encoders, and images from Asus Xtion Pro camera) and also from a human user (initial robot pose). For the sake of clarity, connections to/from the decision subsystem are not shown because it is connected to all the other nodes.

The ROS application consists of navigating from point A to point B in a floor of an office building. Performance is modelled as the inverse of total time required to complete that path. The robot has to recognise some intermediate goals with the camera. Moreover, it uses a floor map to get localised and safely navigate along the path. Next we describe the three key nodes in the system: localisation, navigation, and decision.

A. Localisation and Navigation

The localisation node is based on the Extended Kalman Filter (EKF) [15], which performs sensory fusion in order to generate more accurate poses. Input data to the EKF come from the odometry node (which uses dead reckoning to estimate the pose) and from the Adaptive Monte Carlo Localisation (AMCL) [16] node (which uses a laser-based particle filter to estimate the pose). Both systems perform robot tracking given a previously known position. The odometry node provides poses at a higher ratio than AMCL, but it accumulates a drift error over time. Thus, the sensory fusion consists of correcting the accumulated drift generated by the odometry node every time an AMCL pose is generated.

The navigation node receives as inputs the floor map, the current pose from the EKF node, and the goal pose from the AR recognition node. Navigation is internally divided into a global and a local planner. The global planner is based on the Dijkstra algorithm [17], which computes the shortest path to the goal. The local planner is based on the Dynamic Window Approach (DWA) [18], which generates velocity commands to the robot motors given a global plan and a costmap (a grid map where each cell has an associated cost). It also uses data from laser to avoid obstacles not present in the map.

B. Decision system

In order to take decisions, we need the information gathered in the static analysis. Table I shows the threshold frequencies and tolerances obtained. Note that both *User* and *Map_server* nodes are not considered because only provide data once. As we see, there are minimum and optimal values for frequencies. The robot is able to satisfy only the minimum values. Hence, as the laptop can satisfy the optimal frequencies, the performance derived from running nodes on it should improve. As an example, the AMCL frequency is suboptimal in the robot (2Hz), and most probably will affect the navigation behaviour. Working at its optimal value (5Hz), the robot can navigate faster, thus improving performance significantly.

TABLE I. CONNECTIONS REQUIREMENTS

Connection	Freq_min (Hz)	Freq_opt (Hz)	Δ (s)
Laser-AMCL	5	10	2
Laser-Navigation	2	10	0
Odometer-Odometry	10	40	0
Camera-AR	1	30	1
AMCL-EKF	2	5	2
Odometry-EKF	5	10	0
EKF-Navigation	2	5	0
AR-Navigation	1	30	1

Derived from Table I and Figure 3 we conclude that only the AMCL and AR nodes are candidates for migration, the rest are critical. Table II characterises the migratable nodes (note that the decision system will select AMCL first).

TABLE II. MIGRATABLE NODES CHARACTERISTICS

Node	Executed	CPU usage	Network BW	Perf. improv.
AMCL	Always	50%	200 Kb/s	Significant
AR	Sometimes	33%	6 Mb/s	Small

V. EVALUATION

In this section we present the results of applying our decision system to the test-bed scenario[1]. In order to validate the allocation algorithm, we also provide simulation results.

Prior to evaluate the system performance, we analysed the robot speed. We checked that the robot can navigate at 0.4 m/s only when the AMCL node runs in the laptop, and at 0.2 m/s in any other case. At 0.4 m/s, the AMCL node needs to receive data at the optimal value to ensure correctness (Table I), otherwise it becomes potentially dangerous. Figure 4 shows the difference in the path shape after completing the first corner when we set the robot speed to 0.4 m/s and the AMCL node runs on the robot (right), and on the laptop (left).

In order to evaluate the performance, we run four independent tests. The robot had to complete a square path of 115 meters between points A and B and also to detect 4 AR markers located in the middle of each side. We placed the laptop in the lab and we set the WiFi network from an access point located on the robot tray. Assuming this configuration, the laptop will not always receive a perfect WiFi signal while the robot navigates along the path (due to partial/total coverage loss, network interferences, objects, walls, and so on).

[1]A video related to the experiments can be found online at https://youtu.be/HDPkAjagFzo

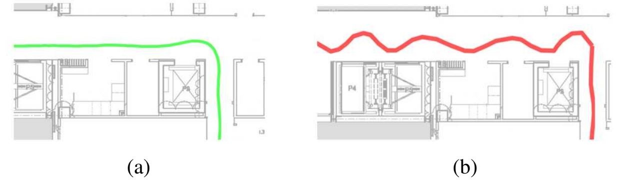

Fig. 4. Path shape. AMCL running on Laptop (a), and on Robot (b).

Figure 5 shows the floor map and the resulting robot path for Test 1. As can be seen, the path is composed of different colors, where each one represents a different configuration. Green means only AMCL node running on the laptop. Red, AMCL and AR nodes running on the robot. Blue, AMCL and AR nodes running on the laptop. And Yellow (it did not occur in Test 1), only AR node running on the laptop. Notice that navigation is counterclockwise.

Fig. 5. Test 1: Robot path configurations (AMCL on Laptop [green], AMCL/AR on Robot [red], AMCL/AR on Laptop [blue]).

Figure 6 shows (left Y-axis) the performance results for the four tests and the baseline configuration (no migration takes place). As can be seen, the total execution time to complete the path is reduced about 28% on average, which is a significant improvement. The figure also shows (right Y-axis) the average of the angular velocity for every test. This velocity is directly affected by the value of the dynamic parameters, and gives an idea of the quality of the path shape (Figure 4). We checked that 4 degrees/s is the limit from which the system behaves incorrectly. Differences across the four test are due to changes in the dynamic environment (e.g. presence of people/objects, doors open/closed, light, etc). Anyway, the average value for the four tests is just 5% higher than the baseline configuration, which demonstrates that when the AMCL runs on the laptop, the robot can navigate at 0.4 m/s safely.

Finally, we validate our allocation algorithm through simulation in MATLAB. We model a generalisation of our test-bed (1 robot, K-1 remote scales, and N nodes) considering a larger number of nodes and remote scales. The network availability is modelled as a random value between 45-65% of the total path length (derived from the results for the test-bed).

Figure 7 shows the results, where N is the total number of nodes (without considering nodes providing input data), M is the number of migratable nodes, and K is the number of scales. We consider three system types (N = {6, 12, 18}

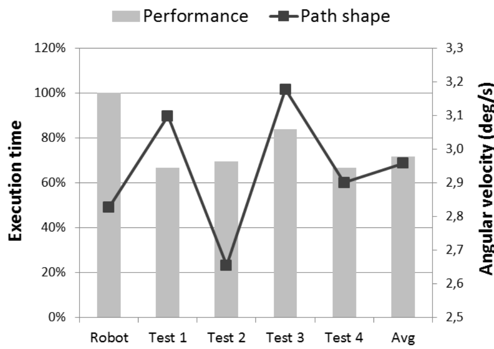

Fig. 6. Test-bed: Performance (left axis) and path shape (right axis).

nodes) with a fixed number of migratable nodes (N = {2, 6, 9} respectively). For every system type, the next configuration includes one scale more (e.g. 6Nx2Mx2K and 6Nx2Mx3K), and this new scale is always powerful than the previous one. The 6Nx2Mx2K configuration represents our test-bed. The Y-axis shows the performance improvement with respect to the corresponding baseline configuration (no migration takes place), where each value is the average of three independent runs. Moreover, for every system configuration we also represent the improvement achieved by the best case scenario (network always available with maximum bandwidth).

From Figure 7, the first observation is that the improvement for the 6Nx2Mx2K system matches with the value obtained for the real test-bed (28%). Second, we see that the trend is maintained for the three system types. Therefore, these results validate our node allocation algorithm, which improves the system performance by 39% on average (50% in the best case).

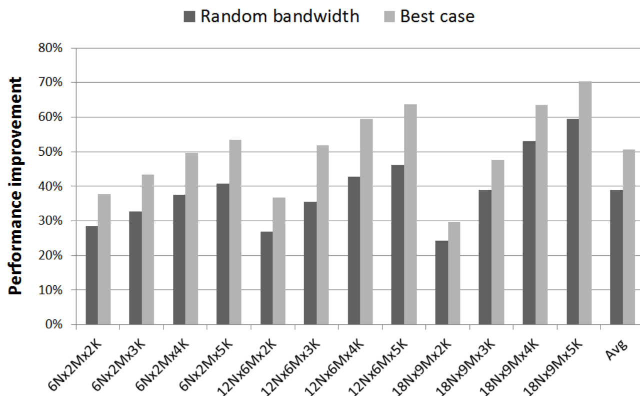

Fig. 7. Simulation results (N=total nodes, M=migratable nodes, K=scales).

VI. Conclusion and Future Work

In this paper we have addressed how to apply the AnyScale computing concept to general ROS-based systems, solving the associated node allocation problem. As a proof of concept, we have implemented a test-bed scenario and a real application. Comparing with a system where no migration is possible, the proposed test-bed improves the global performance by 28% on average. As future work, we plan: to propose improved heuristics to solve the general case; to analyse more complex systems (e.g. adding real time) and evaluate them in real test-beds and with more complete/accurate simulation models.

Acknowledgment

This work is supported by the AnyScale Applications project under the EPSRC grant EP/L000725/1, and also by the "Ayudas de Movilidad del Personal Investigador en Formación" program, funded by University of Alcalá.

References

[1] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[2] Brian P. Gerkey and Maja J. Matarić. A formal analysis and taxonomy of task allocation in multi-robot systems. *The International Journal of Robotics Research*, 23(9):939–954, 2004.

[3] G. Ayorkor Korsah, Anthony Stentz, and M. Bernardine Dias. A comprehensive taxonomy for multi-robot task allocation. *The International Journal of Robotics Research*, 32(12):1495–1512, 2013.

[4] Xiao Jia and M.Q.-H. Meng. A survey and analysis of task allocation algorithms in multi-robot systems. In *Robotics and Biomimetics (ROBIO), IEEE Int. Conf. on*, pages 2280–2285, Dec 2013.

[5] Kristina Lerman, Chris Jones, Aram Galstyan, and Maja J Matarić. Analysis of dynamic task allocation in multi-robot systems. *The International Journal of Robotics Research*, 25(3):225–241, 2006.

[6] Thareswari Nagarajan and Asokan Thondiyath. Heuristic based task allocation algorithm for multiple robots using agents. 2013. International Conference on Design and Manufacturing (IConDM).

[7] Lingzhi Luo, N. Chakraborty, and K. Sycara. Distributed algorithm design for multi-robot task assignment with deadlines for tasks. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 3007–3013, May 2013.

[8] Lingzhi Luo, N. Chakraborty, and K. Sycara. Distributed algorithm design for multi-robot generalized task assignment problem. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 4765–4771, Nov 2013.

[9] Sahar Trigui, Anis Koubaa, Omar Cheikhrouhou, Habib Youssef, Hachemi Bennaceur, Mohamed-Foued Sriti, and Yasir Javed. A distributed market-based algorithm for the multi-robot assignment problem. 32(0):1108 – 1114, 2014. The 5th Int. Conf. on Ambient Systems, Networks and Technologies (ANT), the 4th Int. Conf. on Sustainable Energy Information Technology (SEIT).

[10] Kathryn Macarthur, Ruben Stranders, Sarvapali Ramchurn, and Nick Jennings. A distributed anytime algorithm for dynamic task allocation in multi-agent systems. In *Twenty-Fifth Conference on Artificial Intelligence (AAAI)*, pages 701–706, August 2011.

[11] Andrew J. Page, Thomas M. Keane, and Thomas J. Naughton. Multi-heuristic dynamic task allocation using genetic algorithms in a heterogeneous distributed system. *J. Parallel Distrib. Comput.*, 70(7):758–766, July 2010.

[12] Probir Roy, Md. Mejbah Ul Alam, and Nishita Das. Heuristic based task scheduling in multiprocessor systems with genetic algorithm by choosing the eligible processor. *CoRR*, abs/1208.1922, 2012.

[13] Chuan Wang, Jianhua Gu, Yunlan Wang, and Tianhai Zhao. A hybrid heuristic-genetic algorithm for task scheduling in heterogeneous multi-core system. In *12th Int. Conf. on Algorithms and Architectures for Parallel Processing - Volume Part I*, ICA3PP'12, pages 153–170, 2012.

[14] R. Bischoff, U. Huggenberger, and E. Prassler. Kuka youbot - a mobile manipulator for research and education. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1–4, May 2011.

[15] S.J. Julier and J.K. Uhlmann. Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, 92(3):401–422, Mar 2004.

[16] Patrick Pfaff, Wolfram Burgard, and Dieter Fox. Robust monte-carlo localization using adaptive likelihood models. In *EUROS*, pages 181–194, 2006.

[17] Edsger. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[18] Marija Seder and Ivan Petrovic. Dynamic window based approach to mobile robot motion control in the presence of moving obstacles. In *ICRA*, pages 1986–1991. IEEE, 2007.