Multi-Agent Distributed Adaptive Resource Allocation (MADARA)

Abstract

The component placement problem involves mapping a component to a particular location and maximizing component utility in grid and cloud systems. It is also an NP hard resource allocation and deployment problem, so many common grid and cloud computing libraries, such as MPICH and Hadoop, do not address this problem, even though large performance gains can occur by optimizing communications between nodes. This paper provides four contributions to research on the component placement problem for grid and cloud computing environments. First, we present the Multi-Agent Distributed Adaptive Resource Allocation (MADARA) toolkit, which is designed to address grid and cloud allocation and deployment needs. Second, we present a heuristic called the Comparison-based Iteration by Degree (CID) Heuristic, which we use to approximate optimal deployments in MADARA. Third, we analyze the performance of applying the CID Heuristic to approximate common grid and cloud operations, such as broadcast, gather, and reduce. Fourth, we evaluate the results of applying genetic programming mutation to improve our CID Heuristic.

Keywords

Grid, cloud computing, component placement, heuristics for NP hard problems, deployment planning, MPI, reactive agents

1. Introduction

Hardware is becoming increasingly parallel and many businesses are establishing their own clusters of off-the-shelf components and/or leasing time in cloud computing environments. Users can thus access parallel computational power that was not cost effective a decade ago. With this high availability of resources comes the need to utilize those resources effectively, and users can easily become overwhelmed with the many parallel frameworks and libraries available.

Getting parallel frameworks and libraries to work, however, does not necessary achieve the goal of harnessing available processing power. Users must also deploy their components and programs in configurations that are not only semantically correct, but which use the underlying hardware efficiently and scalably. We call this process the *deployment problem*, and call the process of solving a such problems *deployment planning*.

Much deployment planning research (see Section 5) uses a fine-grained constraint satisfaction problem approach. In this approach, users define every characteristic of a target system, including expected CPU usage, memory, bandwidth, and other profiling information, and apply an offline algorithm to "solve" or approximate a deployment of tasks, components, or actors onto a set of hardware in a set configuration. This type of deployment planning scenario is often called the *component placement problem*.

Component placement is a planning problem (Kichkaylo 2004) whose goal is to map a component to a particular location and maximize global component utility. How that utility is calculated is arbitrary, but an optimal component placement should result in the following:

For all i, $\sum g(C_f) >= \sum g(C_j)$

where C_f is the final component deployment and g is a utility function for evaluating a component deployment. In this paper, g is based on the latency between nodes, but it could be extended to form utility based on bandwidth between nodes or a combination of the two. This paper specifically addresses deployments involving group communication and targeted point-to-point communication using the *Multi-Agent Distributed Adaptive Resource Allocation* (MADARA) application programming interface. MADARA provides application developers with an interface into MADARA deployment services and can either be used as an integral part of a user's software infrastructure, or a separate set of services that may be queried when deployment assistance is needed.

The remainder of this paper is organized as follows: Section 2 motivates how the component placement problem can be applied to network communication operations, such as barrier, broadcast, and gather; Section 3 shows how MADARA addresses the component placement problem by measuring the latency between nodes in the system and using this information to estimate an appropriate deployment; Section 4 details empirical testing of MADARA; Section 5 compares MADARA with related work; and Section 6 presents concluding remarks and lessons learned.

2. Motivating Example

To provide a concrete context for our work on MADARA, this section presents a motivating example in the context of MPI (Argonne 2009), which is a popular grid message-oriented middleware specification. In MPI, users write distributed applications that use a common library (e.g., MPICH1 or MPICH2) to communicate to other individual MPI ranks (which can be viewed as individual components in the component placement problem) or to a group of ranks for broadcast, gather, scatter, and other types of common group communication operations. Consider the following broadcast scenario shown in Figure 1, which shows a common root rank broadcasting to 4 other ranks in the group). In MPI, a rank is an identifier for a processor (i.e. a participant in the communication).

In MPI middleware, broadcast operations like the one depicted in Figure 1 *may* be optimized, but there is no requirement that MPI implementations *must* provide these optimizations. Consequently, most MPI implementations use a random deployment scheme or simply resolve MPI ranks in the order they are provided to the underlying rank deployment system (e.g. the message passing daemon in the MPI standard).

In most circumstances, MPI is used in a traditional cluster environment (i.e., multiple nodes connected via Ethernet in a local area network). Consequently, MPI performance is fine in small clusters (even when doing many group communication operations in small job assignments) because nodes are often behind the same switch, so data requires just a few hops to reach any destination from any source host. When experiments require nodes across multiple routers and switches, however, communication overhead can be reduced and application performance improved significantly by generating targeted deployments for the types of group communication or point-to-point communication required by user applications.

Figure 2 shows a broadcast involving 5 nodes behind a set of 4 routers. Assuming each message transmitted costs 1 ms, with this setup a broadcast would take at least 5 ms, which is the cost from Root to either Node 3 or Node 4. A better root would exist, however, if we assume that users do not care which hardware node serves as root, as long as a deployment of the appropriate application logic is accomplished (ie the user’s program which does a broadcast from MPI rank 0 to the other 4 ranks). If the Root were designated instead as Node 2, the maximum latency would be 4, meaning that barriers and broadcasts would be ~20% faster, since MPI executes group communications as blocking calls and root would have to wait for the underlying system to return.

The scenario shown in Figure 2 highlights a core deployment planning problem for MPI applications. Calculating an optimal group broadcast according to pure constraint satisfaction, however, would require intractable calculation time. Moreover, the MPI specification allows runtime modifications of deployments and logical internal ranks and descriptors to optimize existing deployments (see MPI_Cartesian operations (Argonne 2009)). We therefore need a solution that scales readily, finds rank deployments and redeployments quickly (for operations like MPI_Cart_shift (Argonne 2009)), and minimizes system overhead.

This common scenario can be decomposed into the following set of challenges:

- **Challenge 1: Minimize communication and processing cost of generating a new solution**. In this challenge the MADARA API must minimize the overhead of information gathering and profiling to avoid degrading user application performance significantly. An MPI implementation should not add more overhead in computation or network communication than users can gain in performance increases during execution of application logic. Section 3.1 shows how MADARA minimizes costs when establishing latency information and the standard networking patterns we utilized, and Section 3.2 outlines the $O(n^2)$ heuristic we developed to quickly generate deployment solutions.
- **Challenge 2: Effectively query the user for deployment information**. In this challenge the MADARA API must provide a user interface (e.g., a library, console interface, etc.) and format that allows users to accurately depict their intended deployments. If users cannot determine how to use the deployment solver, they will not utilize the underlying hardware effectively. Moreover, the deployment specification format should allow users to specify at least their system information flow, e.g., whether point-to-point or group communication. Section 3.1 shows an example user deployment constraint request and describes how MADARA interact with the user to address this problem.
- **Challenge 3: Generate deployments that approximate optimal information flow (as defined by users).** In this challenge the MADARA API must produce deployments that meet the communication requirements set forth by the user and seek to minimize latency between the nodes, ranks, components, or whatever entity types the user is optimizing for. As noted in Section 1, research tends to define all system characteristics and potentially solve for variables that have a negligible effect on performance. In our experience with modern clusters, however, the number of hops and overall throughput between communicating nodes will likely group communication bottlenecks, rather than differences in CPU and RAM. Sections 3.2 and 3.3 show the heuristics and algorithms MADARA used to address this problem.

3. Solution Approach: MADARA

Our approach to solving the deployment problem in Section 2 involves placing worker agents (agents can be thought of as operating system daemons or services) on each node to establish latency information between nodes (Figure 3) and then using a specialized broker agent that gathers latencies and processes them (Figure 4) using heuristics and genetic algorithms outlined in this section. These brokers, workers, heuristics, and algorithms form the core of MADARA.

3.1. Structure and Functionality of MADARA

MADARA utilizes multiple workers placed at each node available for deployment. The MADARA workers are complemented by decentralized brokers that query the workers for latency information, process the information to create deployment candidates, and provide users an interface to add or update a deployment plan or retrieve a deployment candidate that match the user-provided deployment plan.

The MADARA brokers serve as redundant information repositories with which users may interact. Though only one broker is necessary for the purposes of the experiments in Section 4, multiple brokers can be run on a single node for redundancy. Application developers can specify the number of desired brokers by changing MADARA parameters. In the case of MPI, only one broker for the entire network is necessary; this broker also serves as an interface to users. Figure 3 shows the interaction between users and MADARA brokers and workers (an example of a deployment description is shown in Listing 1 below). A deployment candidate (which becomes a deployment if a user accepts the deployment candidate) is simply a pairing of component ID (or MPI rank in the case of our motivating example) to the host:port of the MADARA worker that meets the constraints according to the deployment description. For MPI, this deployment candidate would be a vector of hosts with ports (since MPI ranks are 0 to n, which maps well to a vector).

MADARA workers communicate to each other via sockets and acceptors with TCP/IP forming the messaging backbone. This communication provides the MADARA control message interface and is independent of how users may interact with their own application logic. MADARA workers discover each other by using a common port range and a host file list that indicates on which hosts MADARA workers may be discovered. Users may also specify specific host names and ports via the host file for more targeted, automatic discovery of MADARA workers. Forming latency tables is done through the communication paradigm shown in Figure 4.

Figure 4 also shows how latency resolution and latency discovery results in forming a latency discovery round for 2 nodes within 3 total messages, which eliminates the need for synchronized clocks since latency calculation is only done with the local clock on the initiating swarm worker, using a round trip latency scheme. The last 5 latencies per host are kept by the workers in a circular queue. By minimizing communication cost, MADARA helps address Challenge 1 (minimize communication cost of the solution) outlined in Section 2.

After latency discovery completes, the MADARA broker communicates with workers to request averages of the latencies per host that have been discovered by each worker. Although this approach may seem to incur a large communication overhead for informing a broker, the total message size cost per worker cannot exceed sizeof (unsigned int) * n, where n is the number of fully connected, discoverable nodes. Even for 5,000 nodes, therefore this approach would constitute only 20 KB of data transfer.

To address Challenge 2 (effectively query users for deployment information), MADARA presents users with interfaces and results that are relevant to their needs (e.g., group communication from a common root to many other components or detailed flow information from each participating MPI rank). Users can specify how information in the deployment needs to flow (e.g., rank 0 communicates frequently to rank 1), and not the resource constraints and minute details attached to each node that are not relevant to user needs.

Listing 1 shows the format for a deployment plan, which is the textual representation of information flow between components, ranks (MPI), etc. After the MADARA broker receives a deployment plan, it starts using the CID Heuristic to quickly generate a deployment candidate that fits user-defined constraints. In MPI terms, the deployment in Listing 1 details the MPI ranks and how they interact with each other (an arrow indicating a source and a destination from arrow tail to head).

3.2. Approximating an Optimal Solution with the CID Heuristic

To address Challenge 3 (generate deployments that approximate optimal information flow), users need a solution that is timely and relevant. The heuristic MADARA uses to handle this deployment generation is called the *Comparison-based Iteration by Degree* (CID) Heuristic. To understand this heuristic, consider a coalition formation for a video game where one player (a server player) hosts the game and other players (client players) join the game. In such games, one player often tends to have the best equipment and connection, so players choose this person to host the game, which is the equivalent of serving as root in a broadcast, gather, or scatter. In terms of our heuristic, this person is the comparatively best candidate by degree, where degree is the connectivity required of the candidate. The top rank candidate will have the best average latency between connected nodes of that degree.

To find the top rank candidate, the MADARA broker collects all average latencies from the participating workers and sorts these average latencies between the workers, forming a rank candidate list. If p is the number of processors involved and the graph is fully connected, this results in an overhead of $p^2 + O(p \log p)$. For the purposes of computing the top rank, the workers compute their best latencies for the degree of the deployment plan (e.g., the maximum optimized links requested for a single node in the plan). If n is the degree of the deployment plan, then this results in overhead of O(n p) + O (p log p). If n != p, then the n – p ranks would be filled with the remaining worker node candidates.

Figure 5 shows a deployment where rank 0 needs to communicate quickly to rank 1, 2, and 3, but there is no need to optimize for rank 0 to 4 or 5. Rank 4, however, needs to communicate quickly to rank 5, but the user does not care about optimization between rank 4 and any other rank in the deployment. This graph has a degree of 3 (the number of connections defined from rank 0). Consequently, after the MADARA broker has finished harvesting latency information from all MADARA workers, this particular broker would average only the top 3 fastest connections from each worker node. We ideally do this to find Rank 0, which according to our heuristic should be best suited for the spot.

Computing the degree average latency per candidate and sorting that list yields an ascending list of average latencies, forming a rank candidate list. While there are still ranks to fill in the deployment, the highest degreed rank candidate is found and drawn from the lowest average latency rank candidate that has not yet been assigned. The heuristic then picks the n lowest worker candidates (where n is the connected degree of the current rank you are trying to fill) from the current top rank.

The process above is repeated until no more unfilled ranks remain in the deployment plan. A remaining ranks and remaining rank candidates list is used to minimize the processing overhead. This selection process may be performed no longer than p times (since a candidate is selected each turn), and this diminishes by 1 candidate and rank per turn, starting from p. Consequently, this process should result in O (p (p – 1) / 2), which is additive with the average latency sort, meaning that total overhead of this heuristic is $O(p^2)$ + O (p log p) + O (p (p – 1) / 2). $O(p^2)$ is very fast for the types of deployment problems commonly seen in MPI programs (usually less than 100 processing elements). These heuristics and optimizations ensure an $O(p^2)$ to $O(n^2)$ worst case performance that helps meet challenge 1 (an efficient component deployment generator).

3.3. Improving on the CID Heuristic with a Genetic Algorithm

The CID Heuristic was designed to approximate optimal broadcasts from a common root, but the approximation will not always be best for all deployment problems possible. To address this issue, we turn to a simple genetic algorithm that uses random mutations to improve CID Heuristic generated deployment candidates. The genetic algorithm employed in MADARA is a simple O(n) mutating algorithm that takes a deployment and swaps the ranks up to n times for a total of 3n moves (a swap requires 3) per time step. Mutations are simply changes to an existing setup with the ultimate intent of forming a better setup after those changes are in place (more can be read about genetic algorithms and mutating algorithms in (Mitchell 1997)).

The genetic algorithm evaluates the deployment candidate after all swaps are made against the original, and if an overall improvement in average latency is seen, the new candidate is accepted; otherwise, the original is kept. The genetic algorithm implemented here has the following key advantages: the algorithm can be ran as long as the user has available processing time, and the algorithm may be used to improve deployment results with or without using the CID Heuristic. Section 4.2 shows results of using this genetic algorithm both after a random deployment and after a CID Heuristic deployment.

The primary downside to the genetic algorithm described in this section is that it does not use a heuristic to target the mutations. Instead, random mutations are made to the deployment. This algorithm could be further augmented with heuristics that rank swap candidates to result in even better performance, but such heuristics are beyond the scope of this paper.

4. Analysis of Empirical Results

This section presents empirical results obtained by testing MADARA on a range of deployment problem sets. The results show how the CID Heuristic from Section 3.2 approximates optimal deployments quickly. Likewise, the results show how the genetic algorithm from Section 3.3 helps bring deployment candidates closer to the optimal.

To test the CID Heuristic, we dynamically created random deployment plans for thousands of nodes, under conditions typical to our initial problem statement (i.e., broadcast communication described in Section 2) and measured how long the algorithm requires to generate a deployment candidate. To test the genetic algorithm, we developed a utility function that calculates a simple deployment utility based on the summation of latencies in deployment candidate connections. We then graph and analyze the output of these tests to show correlations between generation time and deployment size in Figure 6, and utility improvement and iterations of the genetic algorithm in Figures 7 and 8.

4.1 Heuristic Timing Experiment

Experiment design. To show that the CID Heuristic runs in O(p^2) time, we performed experiments on large, randomly generated deployment plans with randomly generated latencies on connected graphs with thousands of ranks. We hypothesized that the correlated times would show a data trend that rose within polynomial time.

The experiment was constructed using standard C++ STL constructs, such as vectors, strings, and maps. The platform used was Visual Studio 2005 compiled for release and a MADARA broker running the CID Heuristic was tested on a dual core 2.16 GHZ Intel Centrino with 4 GBs of RAM. The results in Figure 6 showed that the CID Heuristic achieved our polynomial time estimates.

Our second set of tests investigated the performance of both the CID Heuristic (Section 3.2) and the genetic algorithm (Section 3.3) in comparison to a random deployment, which is the default behavior for the major MPI implementations. The setup for this test involved taking a randomly generated component placement problem, using the CID Heuristic to generate a deployment, and showing overall utility improvement from the initial CID Heuristic result. We do not attempt to solve the component placement problem to compare to an actual optimal solution because solving the problem is NP hard.

Analysis of results. For a 4,000 node deployment, featuring a worst-case runtime performance scenario for the CID Heuristic (i.e., optimizing a broadcast from a common node), the heuristic took ~400 milliseconds. The CID Heuristic used in MADARA was so fast that generating the actual problem set of a random deployment plan and random latencies accounted for over 80% of testing time.

The CID Heuristic heuristic is thus an efficient way to compute a deployment candidate, with sizes of over 4,000 ranks easily computed within a second. This test, however, does not answer the question of how much CID Heuristic improves the deployment candidate over a random method (as is currently used in systems like MPICH). The test results do highlight the remarkably fast performance of the heuristic, which motivates its integration into online, continuous deployment decisions. The breakdown of the CID Heuristic in Section 3 informally proved O(n^2) behavior and Figure 6 provides empirical validation that the time requirements of this heuristic are predictably O(n^2).

4.2 MADARA Fitness Experiment

Experiment design. Section 4.1 analyzed the timing performance of the CID Heuristic in generating a deployment candidate. Information is still needed, however, about the overall deployment improvement with CID Heuristic and the subsequent simple genetic algorithm. We therefore created two experiments: (1) a broadcast communication between one rank and 1,000 ranks (inclusive) and (2) two broadcasts between one rank and 500 ranks (neither broadcasts are overlapping).

The cluster network was generated randomly to involve links with order of magnitude differences in latency (e.g., 10us, 100us, and 1ms). This generated network represented a varied host architecture where some ranks shared a common host (and may have used shared memory), some were on the same router, and some communicated across multiple routers between ranks. The broker again ran on a dual core 2.16 GHZ Intel Centrino with 4 GBs of RAM, though these tests do not have any relevant broker timings associated with them, and the results produced would have been the same regardless of where they were ran on (given the same worker latency inputs). For this experiment, the deployment strategy with the lower total latency for the deployment performs better (total latency is on the vertical axis). The x axis is labeled mutations, but in truth the random deployment strategy never changes (the random deployment line is included in the graph as a point of reference).

Analysis of results. Figure 7 shows the results of a broadcast from one rank to 1,000 ranks optimized with either (1) the genetic algorithm (Mutation), (2) CID Heuristic with Mutations (CID Heuristic), and (3) a random deployment (Random), e.g., as MPICH does by default.

Figure 8 shows the results of a broadcast from one rank to 500 ranks being optimized along with another broadcast between one rank and 500 completely different ranks. This configuration is evaluated with with (1) the genetic algorithm (Mutation), (2) CID Heuristic with Mutations (CID Heuristic), and a random deployment (Random), e.g., as MPICH does by default.

Figures 7 and 8 show the performance of CID Heuristic and the genetic algorithm in a common group communication context (a broadcast). CID Heuristic shows improvements of 10% to ~50% over a random deployment in the 1-to-1,000 broadcast and 1-to-500 broadcast. This improvement is due largely from the specialization of the CID Heuristic, which was developed to closely approximate optimal group communication from one rank to any number of other ranks. There are exceptions to this rule, however, as discussed in Section 6.

The genetic algorithm also showed significant improvements over the random deployment method, though less of an improvement in the two broadcast experiment. This result is due largely from the genetic algorithm having no intelligent selection routine for mutations. The mutations are simply random, and when mutations were tried between the two broadcasts, so the results were not as dramatic as in the 1-to-1,000 case (Figure 7).

5. Related Work

Grid and cloud computing have been researched for decades, and the component placement problem has received significant attention. The majority of research in this area centers on minimizing system utilization of key resources (e.g., CPU, memory and network bandwidth) or accomplishing load balancing or fault tolerance (e.g., through redundancy). This section compares related work with MADARA.

Component placement solver frameworks. Other researchers like Kichkaylo and Karamcheti (Kichkaylo and Karamcheti 2004), and Kee, Yokum, and Chin (Kee, Yokum and Chin 2006) handle detailed deployments resource constraint, such as CPU, bandwidth availability, etc. This approach features a powerful, useful set of algorithms for detailed planning, but the algorithms require users to estimate all usage (e.g., down to individual application timing using) only monotonic functions, bandwidth specified usage per link (which can be highly variable as the program runs), and CPU and memory usage. Kichkaylo and Karamcheti describe STRIPS-like planning routines to solve the planning problem. Kee, Yokum, and Chin introduce a novel selection and binding technique based on hierarchical data using SQL queries.

MADARA was constructed to not rely on user specified resource settings or estimates. Users simply specify the deployment configuration they would like to optimize according to latency (or some other combination of constraints like bandwidth and latency) and the system does all the work of figuring out what is currently available. Our approach allows users to specify how information should flow instead of finely detailing all resource constraints and allows a great degree of flexibility, as the particular constraint is comparatively determined and not statically done. For frameworks like MPI, Hadoop, and other architectures formed from distributed clouds, this type of interface and constraint breakdown seems more useful, especially as these frameworks are used in bridged cloud settings where a super cloud formed from two or more clouds in disparate geographic locations, e.g., one in New York, Tokyo, Silicon Valley, and Seattle, all communicating over the Internet.

MADARA differs from the other work presented above since it does not constrain components or distributed applications into a situation where they are overly resource bound. It is thus intended for deployments where a distributed application can benefit from optimizing communication links on speed or throughput, but not necessarily within strict limits. This approach results in users specifying deployments that are more likely to remain in place long after hardware is upgraded, whereas specifying a deployment as other work mentioned above would require completely changing the deployment radically when networks, CPU, or RAM are upgraded.

Offline performance profilers to be used in component placement problem solvers. A separate vector of component placement concerns appears in the work of Stewart and Shen (Stewart and Shen 2004), who create offline performance profiles for application performance and use these to predict performance in proposed configurations. Their approach differs from our work since MADARA does not perform any offline evaluation or prediction of how the system will behave. It does keep a running system profile of each node's latency information, but this profile image is very small (kbytes in data size), which allows MADARA to analyze deployments involving thousands of components quickly.

Artificial intelligence (AI), genetic algorithms, and machine learning. Many AI and Machine Learning textbooks ((Bonabeua, Dorigo and Theraulaz 1999), (Kennedy and Eberhart 2001), (Mitchell 1997)) handle deployment planning and deployment optimization, but these are often addressed abstractly. Moreover, these textbooks tend to focus on timing-based constraint problems where actions must be completed in a set order. In contrast, MADARA addresess the problem of distributed applications optimally connecting certain nodes, rather than requiring detailed timing-based planning.

6. Concluding Remarks

MADARA is an API that allows users to quickly formulate deployment plans based on observed latencies in a grid or cloud environment. MADARA exploits powerful reactive learning mechanisms using lightweight workers that establish latency information via optimized communication protocols and brokers that can be redundant and require little interaction with the workers to perform their calculation tasks.

CID Heuristic provides an accurate, quickly computed heuristic for establishing initial deployment plans. After this initial plan is generated, a mutating genetic algorithm can be used to better approximate the ideal deployment for as long as needed to allow this mutating algorithm to compute better approximations. This approach is helpful to online, continuous applications that need a good approximation and have time constraints. For systems that can afford full offline analysis, however, a traditional constraint satisfaction solver (see Section 5) may produce more optimal results.

The following are lessons learned thus far developing and evaluating MADARA:

1. MADARA does not solve all component placement problems in P. Many NP problems have certain configurations that are tractable – we would argue that a single broadcast from a common root node is tractable (see Section 3.2), and that the CID Heuristic finds such a solution quickly. However, Finding an optimal solution to all deployment problems is still NP hard, and if MADARA must provide online generation of deployments, we will always be approximating and not solving. The generation of a deployment candidate via the CID Heuristic is good enough for many deployment needs (including broadcast and barrier). There is room for improvement, however, because the result is not necessarily optimal, especially when deployments concern many interconnected rank communications (e.g., 0 -> 1, 1 -> 2, and 2 -> 0). This result is compounded by deployments with high degrees among multiple nodes with interdependencies.

For example, in Figure 9, Rank 0 is found first since it has the highest degree. From the nodes connected to Rank 0, the top 3 fastest latencies are selected for Rank 1, Rank 2, and Rank 3. These ranks are removed from the remaining rank candidates and the next best candidate may be omitted from this pivotal selection.

There are several solutions for the scenario presented in Figure 8: (1) use evolutionary learning with genetic mutations of candidates to improve on initial heuristics and (2) make a better heuristic guess by recalculating degree for each connected part of the deployment graph.

We could discern if there is any improvement by creating utility values for each connection based on a simple formula (e.g. L_{max} / L_{act}) where L_{max} is the maximum latency available in the latency tables and L_{act} is the actual latency current assigned to this deployment connection). The example utility equation above showcases how straightforward it is to compute and integrate the prep work performed earlier in CID Heuristic, where we established average largest-degree latency. With the utility formula, we can use the genetic algorithm presented earlier or a variation that uses targeted heuristics to better our deployment candidate (if possible), and generate a potentially better final deployment for users.

2. Testing for optimality is problematic in an NP hard problem. Testing for a closed solution to the optimal one is complicated since find an optimal solution to the component placement problem is NP hard. This problem came up during testing and results analysis when we were trying to compare our solutions for 4000 node deployments against a known optimal solution. Since n=4000 in a O(n!) situation or higher is intractable, we had to scale back our testing ambitions.

For our purposes, we found that simply showing improvement over a random solution was enough to highlight the benefits of our heuristic and the subsequent genetic mutation algorithm.

3. Group communication is not just about latency. Latency is not the only constraint required in deployment planning and other group communication constraints (like bandwidth between ranks) are just as important. Our proposed solution to this is to try to include perceived or tested bandwidth between ranks. This is future work however and not covered in the scope of this paper.

4. MPI and other libraries have other hard problems to solve. A more complete toolkit would allow for program execution within the framework, which we will add to MADARA in future work. The latter is not only required for MADARA to form the framework of a tool like MPI's message passing daemon (which MPICH uses to launch parallel applications), but it also has interesting ramifications on fault tolerant, self optimizing brokers or important components. MPI, for the most part, ignores fault tolerance completely, but building in fault tolerance through APIs like MADARA could make MPI implementations much more robust. More information on this topic is planned for future work.

MADARA is programmed using ACE (Schmidt and Huston 2003) and C++ STL and utilize reactors, thread pools, and other patterns to leverage fast processing and data delivery with portability and code maintainability. It is available in open-source form from madara.googlecode.com.

References

[1] Schmidt Douglas C. and Huston Stephen D. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley 2003. Boston, MA. Pg. 40-49, 70-85, 88, 99-100, 104-108

[2] Bonabeua Eric, Dorigo Marco, Theraulaz Guy. *Swarm Intelligence from Natural to Artificial Systems*. Oxford University Press 1999. Oxford, NY. Pg 69-70, 221, 226, 249-250

[3] Hadoop Project Site. hadoop.apache.org/core/

[4] Kennedy James and Eberhart Russel C. *Swarm Intelligence*. Academic Press 2001. San Diego, CA. 40-42, 49-51, 94-98, 150-151, 158-159, 169, 171, 299-301, 307-309

[5] Kee Yang-Suk, Yokum Ken, Chin Andrew A., Casanova Henri. "Improving Grid Resource Allocation via Integrated Selection and Binding" *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. November 2006. Article 99.

[6] Kichkaylo Tatiana. "Timeless Planning and the Component Placement Problem." *ICAPS 2004 Workshop on Planning and Scheduling for Web and Grid Services.* June 2004. Pg. 37-44.

[7] Kichkaylo Tatiana and Karamcheti Vijay. "Optimal Resource-Aware Deployment Planning for Component-based Distributed Applications." *High performance Distributed Computing, 2004.* June 2004. Pg. 150-159.

[8] Stewart Christopher and Shen Kai. "Performance Modeling and System Management for Multi-component Online Services." *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation.* May 2005. Pg. 71-84.

[9] Mitchell Tom M. *Machine Learning*. McGraw-Hill International 1997. Singapore. Pg. 249-270,

[10] Argonne National Laboratories. MPICH Project Site. www.mcs.anl.gov/research/projects/mpi/mpich1. June 2009.

Figures

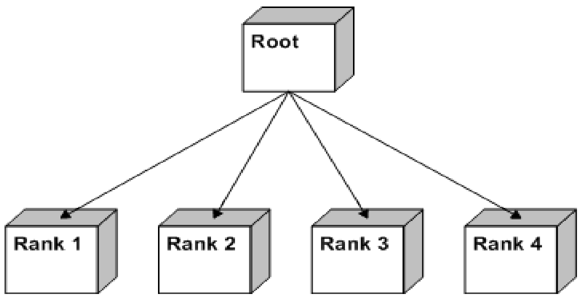

Figure 1. Broadcast from a Common Root Node

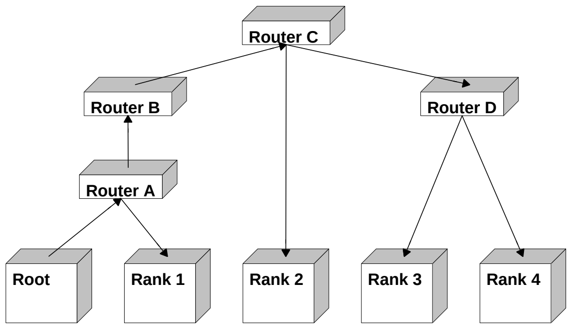

Figure 2. Broadcast from a Root Behind 3 Routers

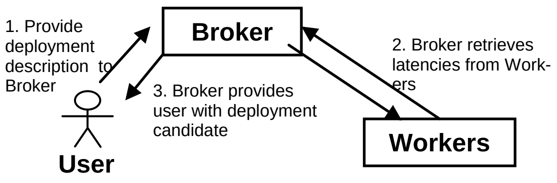
1. Provide deployment description to Broker
Broker
2. Broker retrieves latencies from Workers
3. Broker provides user with deployment candidate
Workers
User

Figure 3. Interaction Between MADARA Entities

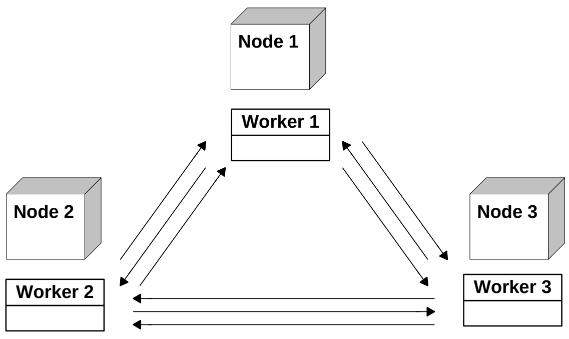

Figure 4. Workers Pinging Each Other to Establish Latency

```
0 -> 1  // optimize for 0 to 1
0 -> 2  // optimize for 0 to 2
0 -> 3  // optimize for 0 to 3
```

Listing 1. Example User Deployment Constraint Request

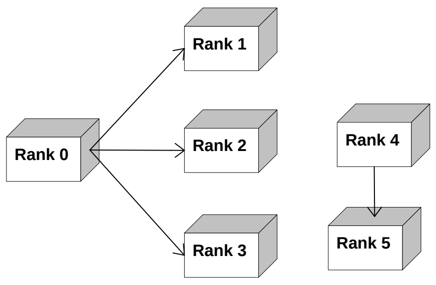

Figure 5. Example Deployment Where Degree is Less Than a Full Broadcast

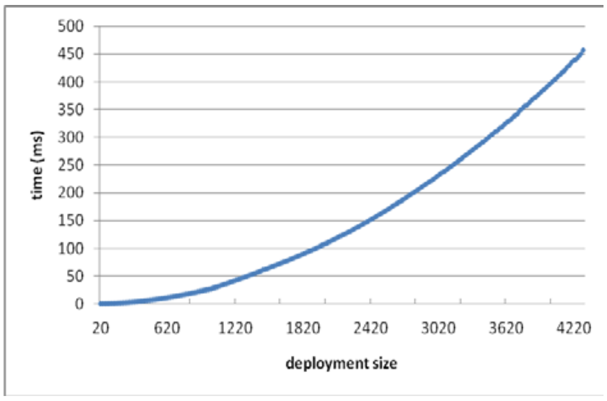

Figure 6. Heuristic Performance with Broadcast Deployments from a Common Root

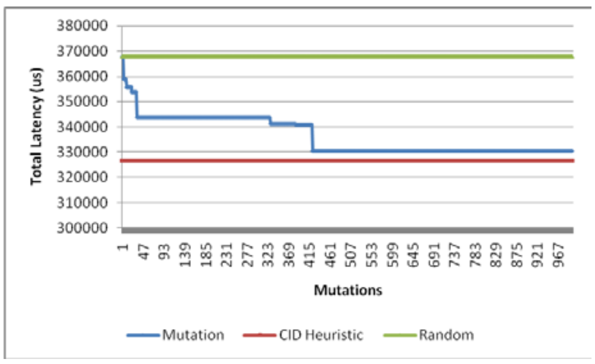

Figure 7. 1-to-1000 Broadcast Performance

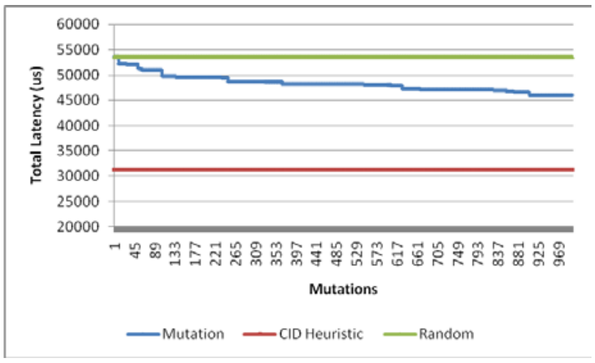

Figure 8. Two 1-to-500 Broadcasts Performance

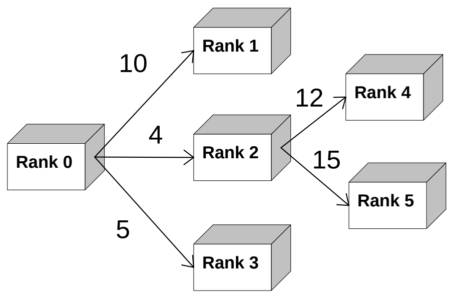

Figure 9. Example Interdependent Deployment