

Constructing Resilient Communication Infrastructure for Runtime Environments

George BOSILCA^a, Camille COTI^b, Thomas HERAULT^b, Pierre LEMARINIER^a
and Jack DONGARRA^a

^a *University of Tennessee Knoxville*

^b *University of Tennessee Knoxville, Universite Paris Sud, INRIA*

Abstract. High performance computing platforms are becoming larger, leading to scalability and fault-tolerance issues for both applications and runtime environments (RTE) dedicated to run on such machines. After being deployed, usually following a spanning tree, a RTE needs to build its own communication infrastructure to manage and monitor the tasks of parallel applications. Previous works have demonstrated that the Binomial Graph topology (BMG) is a good candidate as a communication infrastructure for supporting scalable and fault-tolerant RTE. In this paper, we present and analyze a self-stabilizing algorithm to transform the underlying communication infrastructure provided by the launching service into a BMG, and maintain it in spite of failures. We demonstrate that this algorithm is scalable, tolerates transient failures, and adapts itself to topology changes.

Keywords. Self-stabilization, binomial graph, scalability

1. Introduction

Next generation HPC platforms are expected to feature millions of cores distributed over hundreds of thousands of nodes, leading to scalability and fault-tolerance issues for both applications and runtime environments dedicated to run on such machines. Most parallel applications are developed using a communication API such as MPI, implemented in a library that runs on top of a dedicated runtime environment. Notable efforts have been made in the past decades to improve the performance, scalability and fault-tolerance at the library level. The most recent techniques propose to deal with failures locally, to avoid stopping and restarting the whole system. As a consequence, fault-tolerance becomes a critical property of the runtime environment.

A runtime environment (RTE) is a service of a parallel system to manage and monitor applications. It is deployed on the parallel system by a launching service, usually following a spanning tree to improve the scalability of the deployment. The first task of the RTE is then to build its own communication infrastructure to synchronize the tasks of the parallel application. A fault-tolerant RTE must detect failures, and coordinate with the application to recover from them. Communication infrastructures used today (e.g. trees and rings) are usually built in a centralized way and fail at providing the necessary support for fault-tolerance because a few failures lead with a high probability to disconnected components. Previous works [2] have demonstrated that the Binomial Graph topology (BMG) is a good candidate as a communication infrastructure for supporting both scalability and fault-tolerance for RTE. Roughly speaking, in a BMG, each process is the root of a binomial tree gathering all processes.

In this paper, we present and analyze a self-stabilizing algorithm¹ to transform the underlying communication infrastructure provided by the launching service into a BMG, and maintain it in spite of failures. We demonstrate that this algorithm is scalable, tolerates transient failures, and adapts itself to topology changes.

¹Self-stabilization systems [11] are systems that eventually exhibit a given global property, regardless of the system state at initialization

2. Related Work

The two main open source MPI library implementations, MPICH [4] and Open MPI [13] focus on performance, portability and scalability. For this latter purpose, both libraries manage on-demand connections between MPI processes, via their runtime environments. MPICH runtime environment, called MPD [9], connects runtime daemons processes through a ring topology. This topology is scalable in term of number of connection per daemon, but has two major drawbacks: two node failures are enough to divide the daemons in two separate groups that cannot communicate with one another, and communication information circulation does not scale well. The Open MPI runtime environment project, ORTE [10], deploys runtime daemons connected through various topologies, usually a tree. Recently, some works have proposed the integration of a binomial graph in ORTE [2]. However, the deployment of this topology inside ORTE is done via a specific node to centralize the contact information of all the other nodes and decide of the mapping of the BMG topology over ORTE daemons. This current implementation prevents scalability, and does not reconstruct the BMG upon failures. Our work focuses on the deployment and maintenance of a BMG topology in a distributed and fault-tolerant way, exhibiting more scalability.

Self-stabilization [15,11] is a well known technique for providing fault tolerance. The main idea of self-stabilization is the following: given a property \mathcal{P} on the behavior of the system, the execution of a self-stabilizing algorithm eventually leads from *any* starting configuration, to a point in the execution in which \mathcal{P} holds forever (assuming no outside event, such as a failure). A direct and important consequence of this fault tolerance technique is that self-stabilizing algorithms are also self-tuning. No particular initialization is required to eventually obtain the targeted global property. Some self-stabilizing algorithms already exist to build and maintain topologies. Most of them address ring [5] and spanning tree topologies [12], on top of a non-complete topology. They are usually designed in a shared memory model in which each node is assumed to know and be able to communicate with all its neighbors [1]. To the best of our knowledge, our work addresses for the first time building and maintaining a complex topology such as BMG. The classical shared memory model does not fit the actual systems we target in which connections are opened based on peer's information, thus we designed our algorithm using a message passing, knowledge-based, model [14].

3. Self-Adaptive BMG Overlay Network

We present in this section a self-stabilizing algorithm to build and maintain a binomial graph topology inside a runtime environment. This BMG construction supposes that every process in the system knows the connection information of a few other processes, at most one to be considered as its parent, such that the resulting complete topology is a tree of any shape. This assumption comes from the fact that the start-up of processes will usually follow a deployment tree. The connection information can be exposed to processes along their deployment, by giving to each process its parent's connection information according to the tree deployment. Each process then contact the parent to complete the tree topology connectivity information.

The algorithm we propose is silent: in the absence of failure during an execution, the BMG topology does not change. This property is mandatory for being able to use this topology to route messages. We also focus on obtaining an optimal convergence time, in terms of number of synchronous steps, for underlying *binomial* trees, as the runtime

environment [8] we envisioned to implement this algorithm will usually deploy processes among such topology.

The construction of the BMG is done by the composition of two self-stabilizing algorithms. The first one builds an oriented ring from the underlying tree topology, while the second one builds a BMG from the resulting ring. In the next subsections we present both algorithms, the key ideas of their proof of correctness and an evaluation of the time to build a BMG from different tree shape by simulation.

3.1. Model

System model Our algorithms are written for an asynchronous system in which each process has a unique identifier. In the rest of the paper, although process identifiers and actual processes are two different notions, we will refer to a process by its identifier. We assume the existence of a unidirectional link between each pair of processes. Each link has a capacity bounded by an unknown constant, and the set of links results in a complete connected graph. As in the knowledge network model, a process can send messages to another process if and only if it knows its identifier. When a process receives a message, it is provided with the sender's identifier. The process's identifiers can be seen as a mapping of IP addresses in a real-world system, and the complete graph as the virtual logical network connecting processes in such a system.

Algorithms are described using the guarded rules formalism. Each rule consists in a guard and a corresponding action. Guards are Boolean expressions on the state of the system or (exclusively) a reception of the first message available in an incoming link. If a guard is true, its action can be triggered by the scheduler. If the guard is a reception, the first message of the channel is consumed by the action. An action can modify the process's local state and/or send messages.

The state of a process is the collection of the values of its variables. The state of a link is the set of messages it contains. A configuration is defined as the state of the system, i.e. the collection of the states of every process and every link. A transition represents the activation of a guarded rule by the scheduler. An execution is defined as an alternate sequence of configurations and transitions, each transition resulting from the activation of a rule whose guard held on the previous configuration.

We assume a centralized scheduler in the proof for the sake of simplicity. As no memory is shared between processes so that no two processes can directly interact, it is straightforward to use a distributed scheduler instead. We only consider fair schedulers, i.e. any rule whose guard remains true in an infinite number of consecutive configurations is eventually triggered.

Fault model We assume the same fault model as in the classical self-stabilization model: transient arbitrary failures. Thus, faults can result in node crash, message loss, message or memory corruption. The model of transient failures leads to consider that during an execution, there exists time intervals large enough so the execution converges to a correct state before the next sequence of failure. The consequence on the execution model is to consider no failure will happen after any initial configuration.

3.2. Algorithms

We denote \mathcal{ID} the identifiers of a process ; $List(c)$ a list of elements of type c , on which the operation $First(L)$ is defined to return the first element in the list L , and $next(e, L)$ is defined to return the element following e in the list L . Each of these functions return

\perp when the requested element cannot be found. \perp is also used to denote a non-existing identifier.

Algorithm 1: Algorithm to build an oriented ring from any tree

Constants:
Parent : \mathcal{ID}
Children : $List(\mathcal{ID})$
Id : \mathcal{ID}

Output:
Pred : \mathcal{ID}
Succ : \mathcal{ID}

- 1 - *Children* $\neq \emptyset \rightarrow$
Succ = *First*(*Children*)
Send (*F_Connect*, *Id*) **to** *Succ*
- 2 - **Recv** (*F_Connect*, *I*) **from** *p* \rightarrow
if *p* = *Parent* **then** *Pred* = *I*
- 3 - *Children* = $\emptyset \rightarrow$
Send (*Info*, *Id*) **to** *Parent*
- 4 - **Recv** (*Info*, *I*) **from** *p* \rightarrow
if *p* \in *Children* **then**
 let *q* = *next*(*p*, *Children*)
 if *q* $\neq \perp$ **then**
 Send (*Ask_Connect*, *I*) **to** *q*
 else
 if *Parent* $\neq \perp$ **then**
 Send (*Info*, *I*) **to** *Parent*
 else
 Pred = *I*
 Send (*B_Connect*, *Id*) **to** *I*
- 5 - **Recv** (*Ask_Connect*, *I*) **from** *p* \rightarrow
Pred = *I*
Send (*B_Connect*, *Id*) **to** *I*
- 6 - **Recv** (*B_Connect*, *I*) **from** *p* \rightarrow
Succ = *I*

Algorithm 2: Algorithm to build a BMG from a ring which size is known

Input:
Pred : \mathcal{ID}
Succ : \mathcal{ID}
N : integer size of the ring
Id : \mathcal{ID}

Output:
/* Clockwise links */
CW : $Array[\mathcal{ID}]$
/* Counterclockwise links */
CCW : $Array[\mathcal{ID}]$

- 1 - $\perp \rightarrow$
CW[0] = *Succ*
CCW[0] = *Pred*
Send (*UP*, *CCW*[0], 1) **to** *Succ*
Send (*DN*, *CW*[0], 1) **to** *Pred*
- 2 - **Recv** (*UP*, *ident*, *nb_hop*) **from** *p* \rightarrow
CCW[*nb_hop*] = *ident*
if ($2^{nb_hop+1} < N$) **then**
 Send (*UP*, *ident*, *nb_hop* + 1)
 to *CW*[*nb_hop*]
 Send (*DN*, *CW*[*nb_hop*], *nb_hop* + 1)
 to *ident*
- 3 - **Recv** (*DN*, *ident*, *nb_hop*) **from** *p* \rightarrow
CW[*nb_hop*] = *ident*
if ($2^{nb_hop+1} \leq N$) **then**
 Send (*DN*, *ident*, *nb_hop* + 1)
 to *CCW*[*nb_hop*]
 Send (*UP*, *CCW*[*nb_hop*], *nb_hop* + 1)
 to *ident*

3.3. Building a ring from a tree

The first step to build a binomial graph on top of a tree network consists in building a ring. This section defines a ring topology in our model and describes the proposed algorithm to build one from any tree. The last part of this section proposes a proof of correctness of this algorithm.

3.3.1. Topology description

Tree topology Let \mathcal{P} be the set of all the process identifiers of the system, $|\mathcal{P}| = N$ be the size of the system. For every process $p \in \mathcal{P}$, let $Parent_p$ be a process identifier in $\mathcal{P} \cup \{\perp\}$ that p knows as its parent. Let $Children_p$ be a list, possibly empty, of process identifiers from \mathcal{P} that p knows as its children. We define $anc_p(Q)$, the ancestry of the process p in the set of processes Q as a subset of Q such that $q \in anc_p(Q) \Leftrightarrow q \in Q \wedge (q = Parent_p \vee \exists q' \in anc_p(Q) \text{ s.t. } Parent_{q'} = q)$. A process p such that $Children_p = \emptyset$ is called a leaf. When $Children_p \neq \emptyset$, the first element of $Children_p$

is called first child of p , the last element of $Children_p$ is called the last child of p . We define the *rightmost leaf* of the set Q , noted ' rl_Q ' as the unique leaf that is a last children process such that all processes in its ancestry in Q are last children processes.

A set of processes Q builds a tree rooted in r if and only if all processes of Q verify the three following properties: 1) $\forall p, q \in Q : parent_p = q \Leftrightarrow p \in Children_q$, 2) $Parent_r = \perp$, and 3) $\forall p \neq r \in Q, r \in anc_p(Q)$.

For the rest of the paper, we consider that for all configurations of all executions of the system, the collection of variables $Parent_p, Children_p$ for all processes builds a single tree holding all processes in the system. We call *root* the process that is the root of this tree.

We define the subtree rooted in $r \in \mathcal{P}$, as the subset T_r of \mathcal{P} , such that $r \in T_r \wedge \forall p \in \mathcal{P}, r \in anc_p(\mathcal{P}) \Leftrightarrow p \in T_r$. Note that $T_{root} = \mathcal{P}$ is the largest subtree. The depth of a subtree T_r , noted $depth(T_r)$, is defined as the size of the largest ancestry in this subtree: $depth(T_r) = \max\{|anc_p(T_r)|, p \in T_r\}$.

Ring topology For every process $p \in \mathcal{P}$, let $Pred_p$ and $Succ_p$ represent its knowledge of two processes it considers as respectively its predecessor and its successor in the ring:

Definition 3.1. Consider the relation $\odot : \mathcal{P} \times \mathcal{P}$ such that $p \odot p'$ if and only if $Succ_p = p'$. We define SU_p as a subset of T_p such that $q \in SU_p \Leftrightarrow q = p \vee p \odot q \vee \exists q' \in SU_p$ s.t. $q' \odot q$.

Definition 3.2. Each process of the system is connected through a ring topology in a configuration C iff the following properties are verified: 1) $\mathcal{P} = SU_{root}$, 2) $\forall p \in \mathcal{P}, \exists q \in \mathcal{P}$ s.t. $Succ_p = q \wedge Pred_q = p$, and 3) $Pred_{Succ_{root}} = Succ_{Pred_{root}} = root$.

3.3.2. Algorithm description

We describe in this section the silent self-stabilizing algorithm 1 that builds an oriented ring from any kind of tree topology. Each process except the root of the tree knows a *Parent* process identifier. Every process also has an ordered list of *Children* process identifiers, possibly empty. The basic idea of this algorithm is to perform two independent and parallel tasks: the first one consists in coupling parents with their first child in order to build a set of chains of processes. The second one consists in coupling endpoints of every resulting chain.

The first task is performed by guarded rules 1 and 2. Rule 1 can be triggered by every process that have at least a child. When triggered, the process considers its first child as the next process in the ring by setting its *Succ* variable to its first child identifier. It then sends a message to this first child to make it set up its *Pred* variable accordingly. Rule 2 is triggered by reception of this information message and sets up the *Pred* variable using the identifier contained in the message. Note that each resulting chain

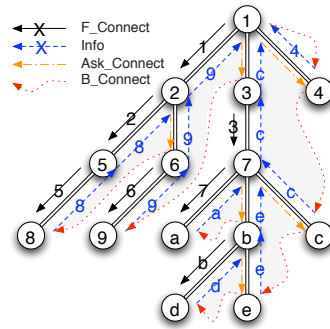


Figure 1.: Message exchanged for building a ring on top of a tree

eventually built by the first two rules has a tree leaf as one endpoint, and that every leaf of the tree is an endpoint of such a chain.

The second task consists in finding for each leaf a process among the tree, the first free sibling, to pick up as its successor in the ring. Rule 3 can only be triggered by leaf processes and sends a message *Info* to their parent to find a process. Rule 4 describes what happens upon reception of such *Info* message. When receiving *Info* from a child c and c is not the last element of its *Children* list, it looks for the process identifier c' that is the next element of c' in its *Children* list. Then it sends an *Ask_Connect* message to c' containing the identifier c so that these two processes address each other (Rules 5 and 6). If c is the last element of the *Children* list, then the process forwards *Info* to its own parent if it has one, or acts as the process looked for if it is the root of the tree.

3.3.3. Idea of the proof

Due to lack of space, we present here the main idea of the proof. The complete formal proof can be found in the appendices of this paper, and in the Technical Report [7]. As for any self-stabilizing algorithm, we first define a set of legitimate configurations, then demonstrate that any execution starting from a legitimate configuration remains in legitimate configurations (closure), and builds and maintain a ring (correctness), and that any execution starting from any configuration eventually reaches a legitimate configuration (convergence).

Legitimate configurations are defined by exhibiting a property on the state of processes (the succession of the *Succ* variables starting at the root builds a chain holding all processes, and the *Pred* variables are symmetrical to the *Succ* variables), and a property on the messages in the communication channels. We prove first that every message initially present in any initial configuration has a finite impact on the other messages in the rest of the execution and on the state of the processes, because all messages have an effect on neighbors only, except *Info* messages, which flow upstream in the tree, thus have a finite time to live in the system.

Then, we prove correction by induction on the subtrees of the system, and closure by analyzing all possible actions of the algorithm, assuming that all channels verify the properties of legitimate configurations. Finally, we prove that starting from any configuration, each channel holds a single message repeatedly, depending only on the shape of the tree, and that as a consequence the channels property of legitimate configurations is eventually verified. Using the fairness of the scheduler and following the action associated with each message identified for each channel, we demonstrate that the state-related property of legitimate configurations is also eventually verified.

3.4. building a binomial graph from a ring

The next and final step to build a binomial graph on top of a tree overlay network consists in, starting from the ring topology constructed by algorithm 1, expanding the knowledge of every process with the process identifiers of its neighbors in the BMG to be obtained.

3.4.1. Topology description

As described in [3], a binomial graph is a particular circulant graph [6], i.e. a directed graph $G = (V, E)$, such that $|V| = |\mathcal{P}|$, $\forall p \in V$, $p \in \{0, 1, \dots, |\mathcal{P}| - 1\}$. $\forall p \in V, \forall k \in \mathbb{N} \text{ s.t. } 2^k < |\mathcal{P}|, \exists (p, (p \pm 2^k) \bmod |\mathcal{P}|) \in E$. It means that every node $p \in V$ has a clockwise (*CW*) array of links to nodes $CW_p = [(p + 1) \bmod |\mathcal{P}|, (p + 2) \bmod |\mathcal{P}|, \dots, (p + 2^k) \bmod |\mathcal{P}|]$ and a counterclockwise (*CCW*) array of links to

nodes $CCW_p = [(p-1) \bmod |\mathcal{P}|, (p-2) \bmod |\mathcal{P}|, \dots, (p-2^k) \bmod |\mathcal{P}|]$. It is important to note that by definition, $\forall k > 0$ s.t. $2^k < |\mathcal{P}| : q = (p + 2^k) \bmod |\mathcal{P}| \in CW_p \Leftrightarrow q = (p + 2^{k-1} + 2^{k-1}) \bmod |\mathcal{P}| \in CW_{p+2^{k-1}}$.

3.4.2. Algorithm description

The proposed algorithm uses the property of the BMG topology. Every node regularly introduces its direct neighbors to each other with rule 1. When a process is newly informed of its neighbor at distance 2^i along the ring, it stores this new identifier to the targeted list of neighbors, depending on the virtual direction, using either rule 2 or 3. Then it sends the identity of the processes at distance 2^i in both directions to introduce the two processes that are at distance 2^{i+1} along the ring to each other, unless $2^{i+1} \geq |\mathcal{P}|$.

3.4.3. Idea of the proof

Complete proof of the self-stabilizing property can be found in the technical report [7]. Due to lack of space, we give here a simple sketch of the proof: correctness and closure are deduced straightforwardly from the algorithm. For convergence, we reason by induction: assuming that the finger table (CW and CCW variables) is correct on the first i elements, we demonstrate that any execution eventually builds the level $i + 1$. Then, stating that level 0 is the ring that has been demonstrated self-stabilizing previously, we conclude that any execution eventually builds a full BMG.

4. Evaluation of the protocols

In this section, we present some simulations of the tree to ring and ring to BMG algorithms to evaluate the convergence time and communication costs of these protocols. The simulator is an ad-hoc, event-based simulator written in Java for the purpose of this evaluation. The simulator features two kinds of scheduling: a) a synchronous scheduler, where in each simulation phase, each process executes fully its spontaneous rule if applicable, then consumes every messages in incoming channels, and executes the corresponding guarded rule (potentially depositing new messages to be consumed by the receivers in the next simulation phase), and b) an asynchronous scheduler, where for each simulation phase, each process either executes the spontaneous rule if applicable, or consumes one (and only one) message in one incoming channel, and executes the corresponding guarded rule (again, potentially depositing new messages to be consumed by receivers in another simulation phase). The asynchronous scheduler is meant to evaluate upper bound on convergence time, working under the assumption that although every process will work in parallel, the algorithms are communication-bound, and the total convergence time should be dominated by the longest dependency of message transmission. The simulator also features three kinds of trees: 1) binary trees, fully balanced and having depth as a parameter; 2) binomial trees, fully balanced and having depth as a parameter; and 3) random trees having both depth and maximal degree (each process of depth less than the requested depth having at least one child, and at most degree children) as parameter. For all simulations, every node starts with an underlying tree already defined (following the algorithms assumptions), and no other connection established ($Succ_p = Pred_p = CW_p[i] = CCW_p[i] = \perp, \forall p \in \mathcal{P}, \forall 0 \leq i \leq \log_2(N)$). Self-stabilizing algorithms cannot stop communicating, because a process could be initialized in a state where it believes that its role in the distributed system is completed. However, real implementations would rely on timers to circumvent this problem and use less resources when convergence is reached and no fault has been detected. To simulate

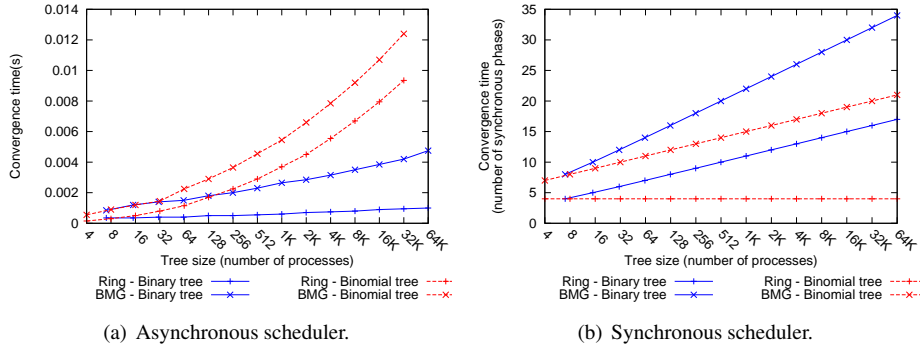


Figure 2. Convergence Time for Binary and Binomial Trees.

this behavior, each process in our simulation becomes quiet (it deactivates its local spontaneous rule, but continues to react to message receptions) as soon as its local state is correct ($Succ$, $Pred$, $CW[0]$ and $CCW[0]$ are correctly set).

Figure 2(b) presents the convergence time of the tree-to-ring and ring to BMG algorithms under a synchronous scheduler, for the case of binary and binomial trees, as function of the size of the trees. The x-axis is represented on a logarithmic scale, and one can see that in the case of an underlying binomial tree, the convergence time of the tree-to-ring algorithm is 4 synchronous phases (each $Info$ message originated at one leaf needs only to go up once to reach the parent of the tree this leaf is the rightmost leaf, then is forwarded to the parent that will step down to the next children which exist and create a $Ask_Connect$ then a $B_Connect$ message, hence 4 phases). For the case of a balanced binary tree, the longest path of $Info$ message has to go from one leaf in the “left” side of the tree up to the root, then two more messages to create the ring, hence $O(\log_2(N) + 2)$ phases. Until the ring has completely converge, exists at least one process in the system which can not start building one of its list of neighbors for the binomial graph. Thus it adds a $O(\log_2(N))$ more synchronous phases just after the ring is converged.

However, some nodes have to handle multiple communications during each phases, and communication-unbalance can happen. The consequence of this communication-unbalance is expressed in figure 2(a), that represents the same experiment under an asynchronous scheduler. With this scheduler, each simulation step consists of at most one message reception per process. Thus, if more messages have to be handled by some processes, the algorithms take significantly more time to reach convergence. To express convergence in time, we assume that each message takes 50 microseconds to be sent from one node to another (this time has been taken after measuring the communication latency of messages of 32 bytes between two computers through TCP over gigabit ethernet). As one can see on the figure, even if the projected convergence time remains very low for reasonably large trees (less than 1/50 of seconds for 64k nodes), the binomial tree presents a non-logarithmic convergence time, while in the case of binary tree, convergence time remains logarithmic. This is explained by figure 3, which presents the maximal number of messages received by a single process during the convergence period for the Binary and the Binomial tree of same size. One can see that the number of messages received by a single process on a Binomial tree is much larger than for a Binary tree. Because a process removes one and only one message at a time from its message queue

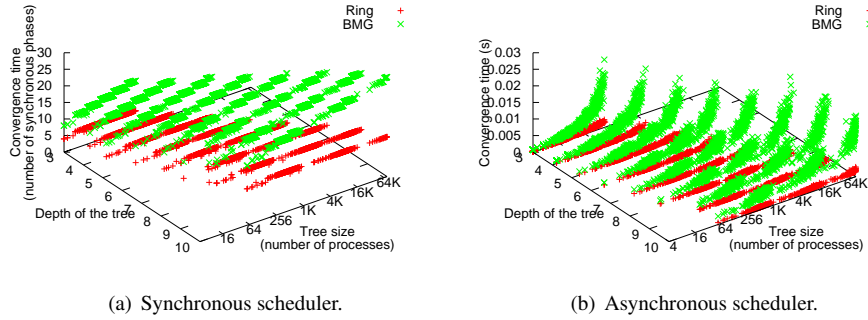


Figure 4. Convergence Time for Random Trees.

in the asynchronous scheduler, the size of the queue grows linearly with the number of direct neighbors and with time (as long as processes deposit new messages in the waiting queue). Thus, the waiting queue of the root in the binomial tree grows of $\log_2(N) - 1$ messages at each phase (until all leafs have ended generating *Info* Messages), whereas it grows of 2 messages at each phase for a binary tree. Thus, convergence time of the binomial tree in this model is impacted by a factor $\log_2(N)$, and we can see in figure 2(a) that the convergence time for the binomial tree is indeed $\log_2^2(N)$, while it is $\log_2(N)$ for the binary tree.

The last two figures 4(a) and 4(b) present the convergence times (in number of phases, or in seconds for the asynchronous scheduler) as functions of the tree size and depth, for random trees. The synchronous version presents a logarithmic progression of the convergence time for the ring construction and for the binomial graph construction. The convergence time of the ring construction algorithm is not modified by the number of nodes in the tree, only by the depth of the tree itself. It presents an increase logarithmic in the depth of the tree, which is consistent with the theoretical analysis of the algorithm. Similarly, the BMG construction algorithm highly depends on the number of nodes in the tree: each process has to exchange $2 \log_2(N)$ messages when the tree is built to build the finger table of the BMG, and this is represented in the figure. However, this progression remains logarithmic with the number of nodes. The asynchronous case is more complex to evaluate: because leafs become quiet only when their successor has received the *Ask_Connect* message causally dependent of their *Info* Message, they introduce a lot of unnecessary *Info* messages in the system. The asynchronous scheduler of the simulator takes one message after the other, following a FIFO ordering, and this introduces a significant slowdown of the *Info* message, put in waiting queues. The projected

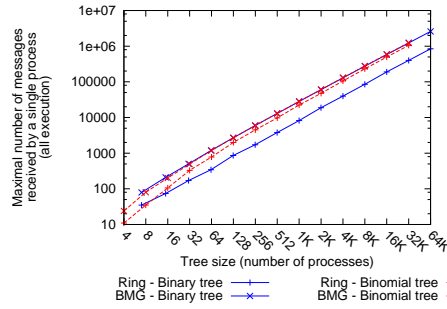


Figure 3.: Maximal number of messages received by a single process for Binary and Binomial Trees under an asynchronous scheduler

time still remains very low, with less than 1/33 second for a 100k nodes tree. However, these results must be validated on a real implementation, to evaluate if the observed trend is due to simulation effects, or will be confirmed in a real-world system.

5. Conclusion

In this work, we present algorithms to build efficient communication infrastructures on top of existing spawning trees for parallel runtime environments. The algorithms are scalable, in the sense that all process memory, number of established communication links, and size of messages are logarithmic with the number of elements in the system. The number of synchronous rounds to build the system is also logarithmic, and the number of asynchronous rounds in the worst case is square logarithmic with the number of elements in the system. Moreover, the algorithms presented are fault-tolerant and self-adaptive using self-stabilization techniques. Performance evaluation based on simulations predicts a fast convergence time (1/33s for 64K nodes), exhibiting the promising properties of such self-stabilizing approach. The algorithm will be implemented in the STCI [8] runtime environment to validate the theoretical results.

References

- [1] Y Afek and A Bremler. Self-stabilizing unidirectional network algorithms by power supply. *Chicago Journal of Theoretical Computer Science*, 4(3):1–48, 1998.
- [2] T. Angskun, G. Bosilca, and J. Dongarra. Binomial graph: A scalable and fault-tolerant logical network topology. In *Parallel and Distributed Processing and Applications, ISPA 2007*, volume 4742/2007 of *Lecture Notes in Computer Science*, pages 471–482. Springer Berlin / Heidelberg, 2007.
- [3] T. Angskun, G. Bosilca, B. Vander Zanden, and J. Dongarra. Optimal routing in binomial graph networks. pages 363–370, December 2007.
- [4] Argonne National Laboratory. MPICH2. <http://www.mcs.anl.gov/mpi/mpich2>.
- [5] A Arora and A Singhai. Fault-tolerant reconfiguration of trees and rings in networks. *High Integrity Systems*, 1:375–384, 1995.
- [6] J.-C. Bermond, F. Comellas, and D. F. Hsu. Distributed loop computer networks: a survey. *Journal of Parallel and Distributed Computing*, 24(1):2–10, January 1995.
- [7] George Bosilca, Camille Coti, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. Constructing resilient communication infrastructure for runtime environments. Technical Report ICL-UT-09-02, Innovative Computing laboratory, University of Tennessee, <http://icl.eecs.utk.edu/publications/>, 2009.
- [8] Darius Buntinas, George Bosilca, Richard L. Graham, Geoffroy Vallée, and Gregory R. Watson. A scalable tools communication infrastructure. In *Proceedings of the 6th Annual Symposium on OSCAR and HPC Cluster Systems*, June 2008.
- [9] R. Butler, W. Gropp, and E. Lusk. A scalable process-management environment for parallel programs. In Jack Dongarra, Peter Kacsuk, and Norbert Podhorszki, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number 1908 in Springer Lecture Notes in Computer Science, pages 168–175, September 2000.
- [10] R. H. Castain, T. S. Woodall, D. J. Daniel, J. M. Squyres, B. Barrett, and G. E. Fagg. The open run-time environment (OpenRTE): A transparent multi-cluster environment for high-performance computing. In *Proceedings, 12th European PVM/MPI Users' Group Meeting*, Sorrento, Italy, September 2005.
- [11] S Dolev. *Self-Stabilization*. MIT Press, 2000.
- [12] Felix C. Gärtner. A survey of self-stabilizing spanning-tree construction algorithms. Technical Report IC/2003/38, EPFL, Technical Reports in Computer and Communication Sciences, 2003.
- [13] Richard L. Graham, Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, George Bosilca, and Andrew Lumsdaine. Open MPI: A high-performance, heterogeneous MPI. In *Proceedings, Fifth International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, Barcelona, Spain, September 2006.
- [14] Thomas Herault, Pierre Lemarinier, Olivier Peres, Laurence Pilard, and Joffroy Beauquier. A model for large scale self-stabilization. In IEEE International, editor, *Parallel and Distributed Processing Symposium. IPDPS 2007*, pages 1–10, march 2007.
- [15] M Schneider. Self-stabilization. *ACM Computing Surveys*, 25(1):45–67, march 1993.