Master's Degree in Informatics Engineering
Dissertation

Group Communication for Large Scale Computing Projects

Daniel Lobo
dlobo@student.dei.uc.pt

*A*dvisors:
Filipe Araújo
Patricio Domingues

September 3, 2013

FCTUC DEPARTMENT
OF INFORMATICS ENGINEERING
FACULTY OF SCIENCES AND TECHNOLOGY
UNIVERSITY OF COIMBRA

Abstract

In volunteer computing systems based on the master-workers model of communication, client machines receive jobs from the project's servers, execute them, and then return the respective results, all the while being unaware of other nodes executing simultaneously. This kind of architecture is adequate for massively parallel applications, but it raises problems for processes that would benefit from communication in parallel or replicated execution, as we would have to rely on the servers for client coordination, thus creating a server-side bottleneck.

In this research, we propose a solution that allows parallel applications to work in standard networked environments, via the implementation of a library, IGCL (Internet Group Communication Library), for inter-node communication and distributed task execution. This library provides clients with peer-to-peer capabilities inside configurable groups of nodes and allows them to communicate during execution in a server-independent way. Exchange of data is done via basic send/receive methods, n-buffering, and several common communication patterns.

To ascertain what kinds of applications are adequate for Internet-scale communication, we implement a set of example algorithms and show that it is plausible to use IGCL for such ends when applications are below certain communication requirements. We also demonstrate that the library has performance comparable to MPI when running in local groups of machines.

Keywords distributed computing, peer-to-peer, Internet, Desktop Grids, communication, peer group, communication patterns

Acknowledgments

This work has been partially supported by the project PTDC / EIA-EIA / 102212 / 2008, High-Performance Computing over the Large-Scale Internet. The project is funded by the COMPETE program from "Fundação para a Ciência e a Tecnologia" (Portuguese Government).

Index

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Field of work and Motivation

BOINC, the Berkeley Open Infrastructure for Network Computing, is an example of a platform for volunteer computing projects[1]. Users around the world provide their machines' idle power to execute scientific applications and contribute to research of various kinds, including the search for extraterrestrial life, simulation of climate conditions and the study of protein structure (respectively provided by projects SETI@home, Climateprediction.net and Predictor@home) [1].

The middleware works under a client-server architecture. Client nodes — the users' machines running the BOINC client — pull jobs from the project servers, run them and give back the respective results when finished [1]. It is common for clients to be given a time frame of a few days to complete each job. Reasons for this are varied: client machines are not always on, the environment is highly volatile (machines frequently disconnect from the network) and the system must also satisfy the users' needs before volunteering resources for BOINC [1] [2]. Because of this, BOINC projects focus on long term throughput, and the system is not expected to be suitable for low latency jobs or close to real-time applications. Despite this assumption, there exist users with highly available machines or groups of machines that, working in parallel, would be able to complete much larger tasks or fulfill a particular one with lower turnaround time.

Another feature of BOINC's architecture is that the only available path

[1] Official website: `http://boinc.berkeley.edu/`

of communication directly links clients to the server, meaning that clients cannot talk to each other. This implies that a large amount of bandwidth is required from the server, creating a bottleneck [3], and that applications that could benefit from node communication cannot run effectively. For instance, several random search methods, such as evolutionary Island Models or a parallel Particle Swarm Optimization systems, can require the exchange of solutions or other kind of information between a large number of populations to effectively explore the search space [4, 5]. This means that the expected time to complete the search (i.e. find an acceptably good solution) is also long, and that many clients (with their own populations) might be needed to improve the algorithm. The inclusion of communication can also support other application examples, like distributed models with multiple coordinators/masters, branch-and-bound applications with bound sharing between nodes, and non-embarrassingly parallel applications in general, as long as they have a relatively small communication overhead.

To mitigate these limitations, we believe that the BOINC middleware and possibly other similar systems could be extended with the concept of node communication, so that clients can exchange data during program execution. The issue can be solved by either altering the projects' servers to act as coordinators that relay messages between clients (no direct node communication exists) or with a new extension that would allow nodes to directly communicate, as made famous by peer-to-peer (P2P) systems. In this research, we implement the second option.

1.2 Goals

The goals of this research are, in a broader vision, to add peer-to-peer communication capabilities to client machines running parallel algorithms and advance towards a more distributed approach to volunteer computing. This will ideally make it possible to run parallel applications over the Internet and complete demanding tasks in less time — provided that these have controlled communication needs —, minimizing job latency and perhaps paving the way for real-time applications. It also implies the reduction of workload on volunteer computing project servers, seeing as data can be passed among peer machines instead of using the server directly, as happens in BOINC.

More concretely, this work's goals consist in implementing and testing a client-side library for typical home computers or clusters; one that automates the creation of peer groups and enables the exchange of data between the

machines during execution. The library should be able to link nodes that are located behind Network Address Translators (NATs) and firewalls, but also yield sufficiently high performance to locally execute algorithms in a single machine or cluster.

However, it is not our goal to deal with the issue of node security directly. For the time being, users can rely on available virtualization methods to mitigate attacks coming from malicious machines. We also do not intend to bridge the gap between the library and real world volunteer computing in terms of implementation, although we describe what general changes it would imply to the library and the system's servers.

1.3 Results and contributions

In this work we implemented IGCL, the Internet Group Communication Library, capable of executing algorithms over the Internet and creating independent peer groups controlled by a coordinator. We also showed how using the concept of node layouts can reduce programming overhead in general, as well as simplify the distribution and collection of data for applications that follow common patterns of communication, such as master-workers or divide-and-conquer. The effects of task buffering are also analyzed, in the scope of this work, and associated with the master-workers model and volunteer computing systems as an efficient way of handling node heterogeneity when sending jobs to remote nodes.

Our tests' results were obtained from four parallel applications examples: matrix multiplication, merge sort, ray tracing and the Traveling Salesman Problem (TSP). These show that Internet-scale communication can indeed be useful and achieve a visible speedup in the latter two — especially the parallel TSP — but is mostly detrimental to the more network-demanding examples. At a local scale, our tests revealed that IGCL runs our example applications with comparable performance to equivalent implementations in the Message Passing Interface, using Open MPI [6], and even performed rather well against a shared memory approach that uses threads in one of our examples.

The library's API and features, including registration of nodes, possible connection types, error handling and internal implementation in general, were also described, as well as its limitations from our point of view.

From a scientific point of view, our main contribution with this work is to

show that it is possible to achieve a significant speedup in parallel applications executing at Internet-scale, as long as they have reduced communication needs. We also show the counterpart of this result; i.e. that many typical parallel applications in high performance computing are simply not suitable for Internet deployment due to excessive transfers of data in frequency or size. As part of our research, we also give a few examples of applications that are expected to work well in the Internet and P2P-enabled Desktop Grid systems.

Another contribution of our work is the library itself, IGCL, which functions in both local and Internet environments and automatically establishes connections according to the nodes' locations and NAT or firewall obstacles, thus adding suitable support for P2P communication in volunteer computing systems. We also describe what worked well and what could be improved in IGCL, both of which should provide some insights for future work.

1.4 Document structure

The remaining document is structured as follows: in Chapter 2, State of the Art, we will analyze existing Desktop Grids, peer-to-peer systems, the applicability of peer-to-peer in the BOINC architecture, the Message Passing Interface standard [7], fault tolerance in volatile environments, the effects of communication on speedup, and, lastly, the application models that benefit from distributed execution and their common patterns of communication. In Chapter 3, Library Implementation, we detail the mode of usage, features and technical details of our library, IGCL. Next, in Chapter 4, Results and Discussion, we define our experimental setup, present the results achieved and discuss their relevance to the problem at hand. Finally, in Chapter 5, Conclusions, we summarize the outcomes of this work and propose a series of possible improvements and extensions to it as future work.

Chapter 2

State of the Art

2.1 Grid and Volunteer Computing

A generic computer grid is an environment of connected computers that serves the purpose of completing resource demanding tasks, generally for research in a business, scientific or academic organization. Large grids of these computers are costly, not only because of the required processing power but also due to network connectors, energy consumption and physical space, sometimes being out of reach for these organizations. Similarly, cloud computing also involves hosting costs that, although not in the same order as buying specialized hardware, are generally not desired or even possible for some projects' budgets.

On the other hand, for many people, computers are used for simple activities most of the day. Browsing the Internet, writing a report or chatting with other people are some examples. It is realistic to expect that most processing capabilities of common personal or shared machines remain unused for the majority of time [8]. Generally, a machine is only near its potential when a large amount of resources is requested by demanding applications, like computer games or video processing suites; even then, there are sometimes system bottlenecks like disk access speed that prevent other components from being useful at the same time.

The idea of volunteer computing (also referred to as "public-resource computing") draws from this aspect of the everyday usage of machines: to gather the large amount of unused resources from millions of personal computers around the world for useful computing [1]. These users' computers form

what is known as a volunteer Desktop Grid. To mitigate the aforementioned problem of lack of processing power, organizations sometimes rely on these Desktop Grids and outsource the required computation to machines from volunteer users, typically for free or for a symbolic reward like public user recognition.

Many volunteer computing projects have successfully existed over the years. The first known public-resource computing examples are the Great Internet Mersenne Prime Search (search for prime numbers of the Mersenne family) and Distributed.net (an effort to break existing challenges in cryptography), in 1996[1] and 1997[2], respectively. Some well-known projects still existing today, like Folding@home[3] (protein folding and related problems in biology) and SETI@home[4] (analysis of radio waves for signs of transmissions from extraterrestrial intelligence) were also created years later, as were other examples. In more modern days, software such as BOINC and XtremWeb (which we will detail in Section 2.1.3) have been further expanding the concept of Desktop Grids.

2.1.1 BOINC

BOINC is an open source software, responsible for the existence of many large volunteer computing projects existing today. It is developed by the same group responsible for SETI@home and is essentially a middleware consisting of two parts — server and client — that act as a bridge between the servers of a distributed application and several client machines of volunteer users [1]. With this model, the combined computational power of these machines can be used to solve large problems in small tasks, as a Desktop Grid. The average computational power of this BOINC grid at any given moment is measured to be above 7 PetaFLOPS[5] as of August 2013, which rivals the top supercomputers at that date[6]. Research projects with various objectives and requirements, and from fields as diverse as physics, chemistry, biology, astronomy, climate, mathematics and game studies, are thus able to gather the resources of contributing users instead of relying on supercomputers or clusters for that purpose [9]. Some examples of BOINC projects, besides the

[1] See `http://www.mersenne.org/various/history.php`
[2] See `http://www.distributed.net/History`
[3] Folding@home: `http://folding.stanford.edu`
[4] SETI@home: `http://setiathome.berkeley.edu`
[5] See `http://boincstats.com/en/stats/-1/project/detail/overview`
[6] See `http://www.top500.org/list/2013/06/` for a list from June 2013

previously mentioned SETI@home, are Climateprediction.net (simulation of climate scenarios), Einstein@Home (detection of types of gravitational waves) and MilkyWay@Home (defining a three dimensional model of our galaxy).

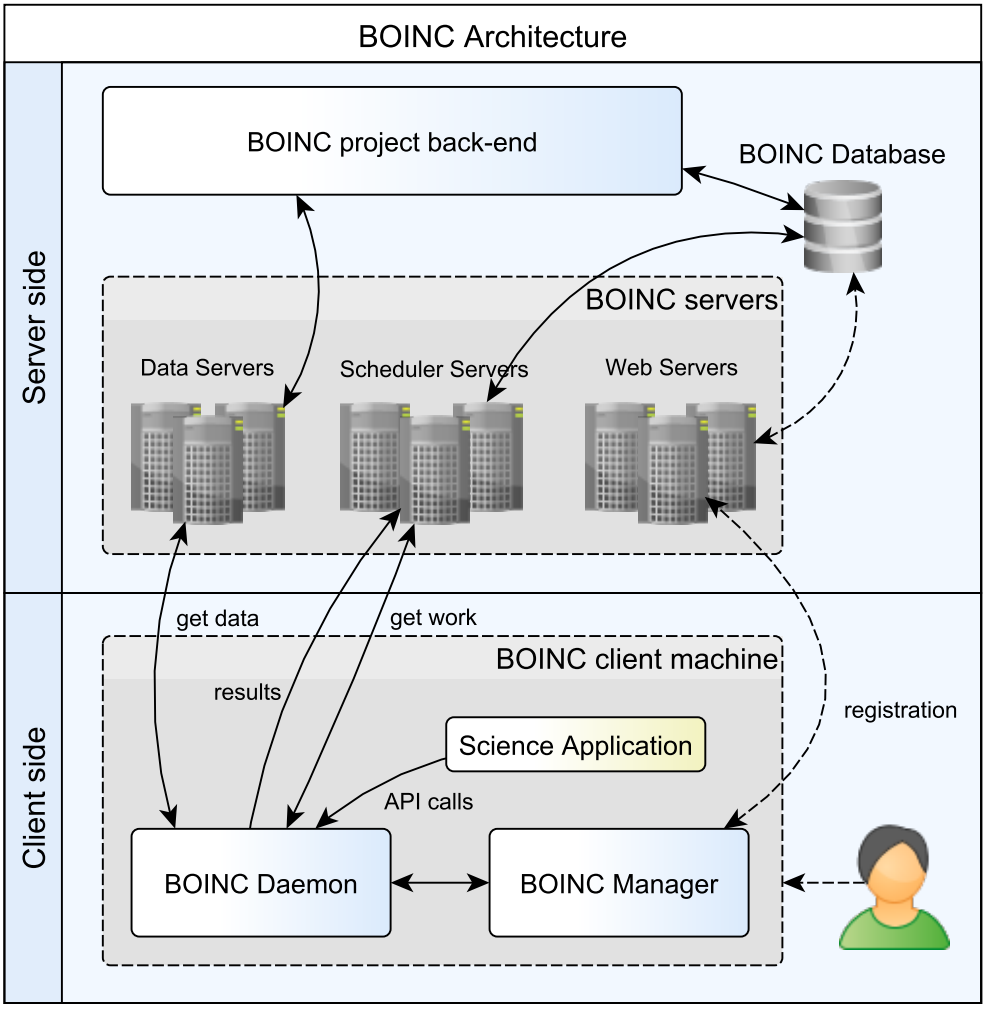


Figure 2.1: Simplified BOINC architecture with server and client side components.

In the BOINC architecture, shown in Figure 2.1, human users with potentially reduced technology knowledge start by attaching their machines to several projects via their respective web pages or an account manager. Then, through the BOINC client, these machines dynamically request, pull and process data from the project's scheduler and data servers, returning the achieved results upon completion. Besides the mandatory web, scheduler and data servers, BOINC projects are generally composed of their own databases

and servers. These are included in the project back-end shown in Figure 2.1.

The combination of server-side and clients in BOINC composes a system that is essentially a master-workers model, with a central node (the task server) responsible for the sending tasks to the worker nodes (clients) for processing. This architecture, also known as master-slaves model, permits very simple task distribution, usually embarrassingly parallel (i.e. the various parts of the algorithm are independent and can be processed in parallel and in any order), which means that the well-known MapReduce model of data processing [10] is easily applied. A simple master-workers example is seen in Figure 2.2.

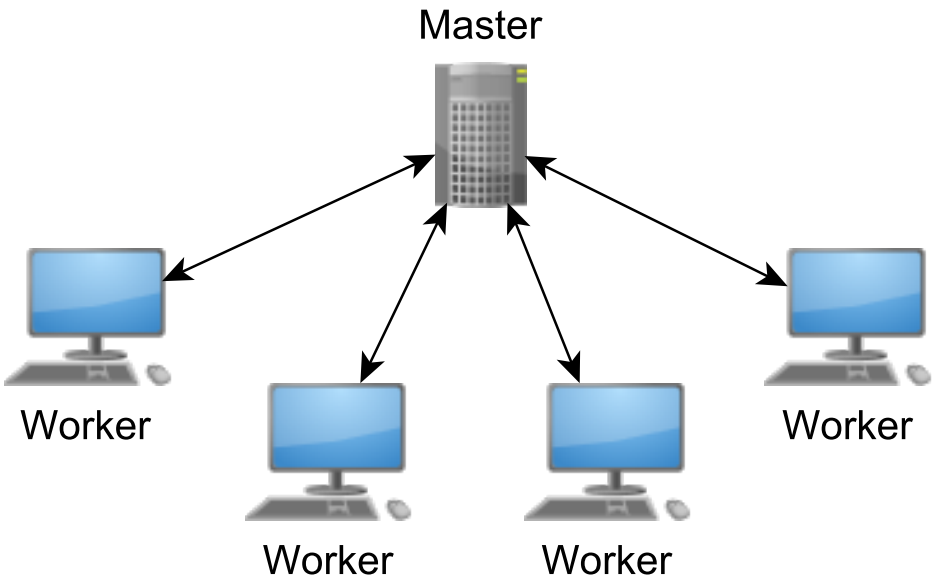


Figure 2.2: Simple master-workers model in computation.

The BOINC client is responsible for the scheduling of available processing power to the different projects, taking into account the user's settings for usage of resources in the machine [11]. Scheduling is also a task for the BOINC server, which must account for the heterogeneity of hardware and availability of clients. In fact, clients are expected to be frequently unavailable, so BOINC applications make use of a checkpointing system to save the running state of each job from time to time and be able to resume it later. Depending on the application, BOINC jobs also vary greatly in requirements of storage, computation, communication and completion time, and are issued differently

according to the known features of the machines [9]. Problems related to the validity of results coming from the users' machines, either by malicious intents or failure, are dealt with by using replicated computing. This means that jobs are sent to several hosts to check if their returned value matches, until either a consensus or a replication limit is reached [9].

One negative issue of the master-workers architecture is the fact that servers become a bottleneck of the system, limiting the maximum simultaneous data downloaded by the clients due to bandwidth restrictions. This happens because every client must connect to the server to pull jobs for processing, even if other nodes, close or far away, already have the same job. Suggestions for the use of BitTorrent to share jobs between nodes have been made by David Anderson in [1] and later researched and implemented by Fernando Costa et al. [12] [13]; research that is described in more detail in Section 2.2.3.

Another problem is latency. BOINC was not designed for low latency jobs but for maximizing throughput in long term computation, as is the case with other large scale computing systems, like HTCondor [14] or XtremWeb [15]. In fact, BOINC clients have a deadline for submitting results that is usually in the order of days (but can be shorter or longer, depending on job), which helps mitigate the previously mentioned server bottleneck problem, as clients will not be constantly communicating with the server. Based on this and the fact that clients cannot directly collaborate with each other for the completion of tasks, projects that want fast results, such as weather or seismic activity prediction, probably will not find in this unmodified BOINC an adequate environment. To try to address this problem, Yi et al. proposed, in 2011, RT-BOINC [16], or Real-Time BOINC, which is an improved version of BOINC in terms of scalability and response time in general; one that could house short-term applications with time completion requirements of about 30 seconds, assuming that hosts are highly available during the small time fractions of processing.

2.1.2 HTCondor

Other paradigms exist for creating and using grids of computers, contrasting with the volunteer computing environment provided by BOINC. HTCondor (previously named "Condor") is a software for the management of workload and scheduling of tasks in a system of distributed computational resources, with a thought for high throughput computing[7]. More specifically,

[7]See `http://research.cs.wisc.edu/htcondor/`

the case with HTCondor is that it allows organizations to build a cluster of computers with commodity hardware and effectively use it for running tasks on demand. The idea of this platform is that several "everyday" computers — dedicated or not, but pre-configured to use the software — can form pools of workers and that authorized users in the grid can then submit jobs for processing in a distributed manner [17].

HTCondor makes use of a task queue and a matchmaker between idle jobs and idle machines, both part of a workload manager, thus sending the queued jobs to certain machines according to the scheduling mechanism, user priorities, job priorities and even job dependencies [18, 17]. HTCondor checks the progress of these tasks until completion and is able to warn the job submitter at that time. It possesses features like the flocking of resources from pool to pool, checkpointing of jobs (like BOINC provides) and remote system calls that grant the existence of a shared file system and effectively allow machines to run jobs and use input/output as if they were running in the job-source computer [18, 17].

2.1.3 XtremWeb

XtremWeb is another open source software project in volunteer and grid computing. Built with Java, XtremWeb makes it easier to build Desktop Grids using unused resources, similarly to HTCondor. Computers spread over a Local Area Network (LAN) or the Internet can serve as workers, donating their spare CPU, storage and network resources to the completion of tasks [19]; therefore, XtremWeb can be used to build both institutional grids and volunteer grids. Contrasting with BOINC, where the project's servers are the only source of jobs, XtremWeb is composed of three tiers: a task coordinator, workers, and clients. Workers are allowed to submit tasks to the coordinator service — thus acting as clients — as well as process jobs [20]. This behavior raises security concerns with the intent of applications, as certification and modifications of the original application are not required to run in the system. Hence, there can exist tasks with malicious content, and users of the platform should only run trusted applications in the grid. Nevertheless, the workers implement sandboxing of Java byte code, and Java applications are run in the Java Virtual Machine, which has configurable security features [15, 19].

Despite the job submission capabilities given to workers/clients, XtremWeb still works via a pull model, much like BOINC, where workers voluntarily get jobs from the coordinator service queue when scheduled to do so. Thus,

the main differences of XtremWeb when compared to HTCondor are the pull model behavior of coordinator-workers and the possibility of running on hardware in other networks, which might be firewall-protected [21, 19]. By contrast, as we have seen, HTCondor uses the master-workers push model and relies on LAN hardware.

Besides the Desktop Grid functionality, XtremWeb is also intended to provide an environment for the exploration of the capabilities of Desktop Grids, peer-to-peer systems and global computing in general. The platform works as a framework for testing issues with the scalability of such systems, research on data- and computation-bound applications, sandboxing, safe execution of code and also benchmarking workload for scheduling algorithms [15].

2.1.4 Others

Further projects in grid and volunteer computing are also worth mentioning, in this work's scope:

SZTAKI Desktop Grid is a BOINC-related project that provides an API to build local Desktop Grids in a hierarchical manner. This means that smaller grids with spare resources are able to take and process work units from a higher level grid. The model can be extended to form a "volunteer cluster" environment for running applications that use MPI (see Section 2.3) or other communication-based computing method [22].

EDGeS and EDGI are linked concepts. The objectives of EDGeS (Enabling Desktop Grids for E-Science) were to build a bridge between cluster Service Grids (like the European Grid Infrastructure) and Desktop Grids such as BOINC and XtremWeb, as well as enable their interoperability based on authentication certificates for safe application execution [23]. Developments over EDGeS later originated EDGI, which was created with the challenge of extending EDGeS for academic clouds and institutional Desktop Grids [24].

SpeQuloS is a framework with connections to cloud computing, which aims to provide Quality of Service to Desktop Grids (which are commonly referred to as Best-Effort Distributed Computing) by dynamically making cloud resources ready for processing when its volunteer resources are

unavailable[8] [25]. This solves the problem of users dynamically leaving the system — an issue difficult to avoid in volunteer Desktop Grids — and provides a way for low latency applications to use volunteer resources with less concern for availability.

2.2 Peer-to-peer

Peer-to-peer (often abbreviated as P2P) is a model of network communication in which every participating peer/node has the same privileges and acts as both a client and a server, possibly with no central authority, thus creating a fully distributed environment. In fact, the main distinction that can be made between distributed communication like P2P and the model of master-workers mentioned in Section 2.1.1 is that, in the former, nodes communicate directly and do not require a central coordinator, while in the latter they do. Consequently, because information does not pertain only to a central unit, fully (or almost fully) distributed P2P networks avoid single-point-of-failure issues. Knowing this, P2P networks are useful to share resources like CPU or storage between nodes, based on mutual advantage, and are commonly deployed for such ends, as is the case with BitTorrent (which we will detail in Section 2.2.1). Other examples of P2P are seen in VoIP (Voice over IP) communication, video streaming and collaborative applications [26, 27, 28].

Despite directly communicating, nodes in a P2P network can still not know each other's identity, as each node can originate from many domains and from behind firewalls and Network Address Translators (NATs), especially if in the context of the Internet. Similarly to how volunteers in BOINC imply the threat of data manipulation, this "blind" node communication means that there is the possibility of data being malicious if the environment is unknown. Another clear difference with *distributed*, compared with the *master-workers* models, is that special care must be taken with node fault tolerance, as there is the possibility that no one is directly responsible for the group of connected peers. It is important to note that both described environments (BOINC and an Internet P2P network) are highly volatile and expected to work with many unreliable nodes.

[8] Official website: `http://graal.ens-lyon.fr/~sdelamar/spequlos/`

2.2.1 BitTorrent

BitTorrent is a P2P protocol created in 2001 for the direct sharing of data files between nodes in possibly different networks. It was modeled with the basic assumption that upload speed is generally slower than download speed and, therefore, the number of file servers should be greater than the number of downloading clients in order to maximize throughput [29]. BitTorrent had a large acceptance from Internet users and content distributors, and still today accounts for a large amount of total Internet traffic, only below HTTP, YouTube and Netflix traffic[9].

In the protocol, shared files are divided into pieces with an associated hash [30] and can be replicated to many different nodes, named "seeders" of that particular file. Those nodes will act as servers of the pieces in their possession, and can provide them to the currently downloading nodes. In Figure 2.3 we show a possible state of such a system in a given moment. Each peer has different pieces in its possession and hence downloads the remaining ones from other peers. In the current state of the Figure, one node already has all of the file's pieces, so it is a seeder of the complete file. As we mentioned before, the more seeders a file has, the faster the potential upload stream of data and the larger the maximum download speed achievable.

The hash in each file piece prevents tampering or corruption of data, consequently making BitTorrent resistant to malicious nodes and faulty networks. The protocol can also survive low initial availability of nodes due to its approach of fragmented files, which can start uploading as soon as acquired; nevertheless, flash crowds (sudden increases in number of downloaders) are still a problem in BitTorrent [31]. The whole system is, furthermore, dependent on the wisdom of users, who should upload enough to maintain a good flow of data.

Despite the apparent fully distributed model of BitTorrent, the knowledge of which peers have the desired files in their possession generally pertains to a server, denominated "tracker". This is the server that peers should contact in order to get other peers' locations (see Figure 2.3). To reduce stress on the tracker and, in some cases, speed up the discovery of nodes, other possibilities are available. An example is the use of a protocol based on Distributed Hash Tables (DHTs), like Chord [32]. Using these DHTs, each node initially knows only a few peers with a certain file. When downloading, the node will contact its known peers, which in turn might know the location of some more peers

[9] See `http://www.sandvine.com/news/global_broadband_trends.asp`

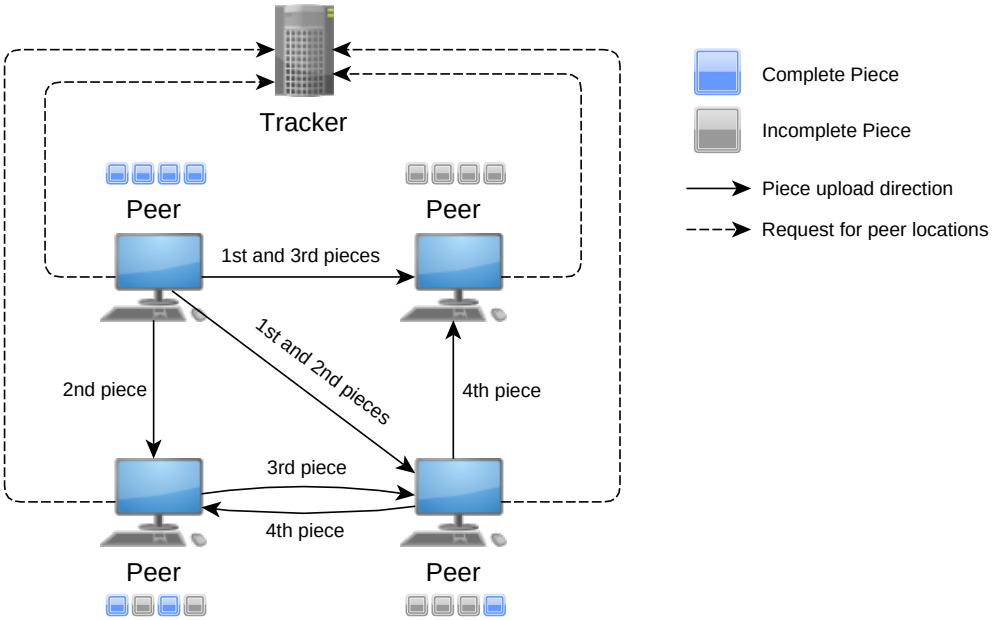


Figure 2.3: BitTorrent architecture with peers and a tracker. Peers with completed pieces can provide them to their downloading counterparts.

with the file and add them to the list. The process continues until the original node knows a sufficient amount of seeders for that file, making the process server-independent.

2.2.2 BAR Model

The Byzantine/Altruistic/Rational(BAR) model is a representation for networks with multiple administrative domains that considers that every node in the network is able to deviate from the defined protocol. Aiyer et al., who suggested the model, argue that careful actions must be taken according to this behavior, in order to support the robustness of distributed/ cooperative services [33].

The model defines that a participating node may disagree with the protocol — the desired course of action — because of three general reasons:

1. the node may be broken
2. the node may be malicious
3. the node wants to satisfy its needs before those of others

Broken and malicious nodes are called Byzantine; selfish nodes are called Rational. The third and last type, Altruistic, consists of nodes that simply follow the protocol as expected [33]. For instance, a BitTorrent network lives on the assumption that clients/nodes will give back around as much as they take in terms of data. If a node follows this "protocol", it is considered an Altruistic node in the BAR model. If another node simply downloads a file and then does not upload data back to others despite being able to, it is a Rational node. These are the ones that "think" and act in a way that benefits them the most, despite the needs of others. Finally, a node is Byzantine if it either acts with malicious reasons, tries to corrupt data on purpose to hinder the service operation, or is simply misconfigured and frequently disconnects [33].

Assuming that all nodes have a certain probability of deviating from the standard protocol, not all deviating nodes should be applied a fault when this happens (as occurs in Byzantine Fault Tolerance). Therefore, there is great interest in the existence of mechanisms that deal with this, benefiting the nodes that follow it and penalizing others in a robust way. The studies on the BAR model have tried to address this by using verifiable pseudo-randomness (it is verifiable so that faulty nodes cannot hide behind randomness of behavior), letting nodes assign "Proofs of Misbehavior" to other nodes, and using no long-term reputation (due to the mix of Byzantine and Rational nodes, which would sometimes result in the assignment of durable bad reputations in acceptable cases). This research was centered on a distributed backup service [33] and on data streaming applications built with BAR model in mind [34, 35], and showed that with the use of the model it was possible to maintain the robustness of the systems even when under a significant percentage of malicious and deviating nodes.

2.2.3 Peer-to-peer in BOINC

The research on RT-BOINC, previously seen in Section 2.1.1, might suggest that a P2P approach is not necessary if the added node communication capabilities are not desired but there is a need for low latency. RT-BOINC focuses on the inclusion of mechanisms for deadlines and improving the BOINC server in scheduling and database access, and not on the restructuring of the communication paradigms in the middleware. Nevertheless, some research has suggested that BOINC could benefit from P2P approaches.

Costa et al. argued that BOINC and Desktop Grids in general should

care about taking advantage of the client's network capabilities and not only of CPU cycles in the client machines, in order to reduce the server bottleneck and associated costs [12]. The authors suggested a hierarchical P2P approach to data distribution in BOINC. The main idea was to address the problem of security in a P2P network via super-peers, which were the only trusted nodes that could relay messages, thus reducing the probability that the network is flooded with false data from an unreliable node.

BitTorrent was also a suggested approach to data distribution in [12]. In fact, in another paper, the authors tried to apply the BitTorrent protocol to the distribution of jobs in BOINC, with mixed success [13]. The idea was to take advantage of the relatively large number of user nodes running the same or similar job and share the needed files among themselves using BitTorrent. Hence, a large amount of bandwidth could be saved on the project servers and the maximum data throughput achievable in the client's network improved. The conclusions achieved revealed savings of over 90% bandwidth in the project servers and almost negligible influence on client processing time, even when seeding intensively. Despite this, they also revealed that the protocol was not efficient for sharing small files and that the sum of BitTorrent client and tracker of peers resulted in several spikes of CPU usage and slow initial seeding of data in the server [13].

M. Cieślak proposed a total re-implementation of BOINC using JXTA. Several modules, like scheduler, data servers and even the reward system were suggested according to the paradigm of P2P. The goal was to address some BOINC limitations mentioned by the authors, such as server overload due to redundant communication, improper task distribution and the limitation of the project servers' resources [36]. Nevertheless, this was a solely theoretical work.

Another work and research project, VolpexMPI, is an example of adding node communication to volunteer computing. It will be described in more detail in Section 2.3.1.

2.2.4 NAT traversal

Common home networks are managed by a router that provides the only way for communication between the machines in the network and other machines in external networks. It is one of the router's jobs to hide the nodes that are behind it until they specifically ask to connect to the outside. Consequently, these nodes do not possess an identification/address that is valid

in the "outside world" of the IP protocol, but only a local IP that identifies them before this specific router. The router, however, does have an IP address that is valid to the outside, and can transparently translate local addresses for the outside and store those translations for future use, while blocking unsolicited connections. This is done mainly for two reasons: 1. to reduce the number of IPs in use at the Internet scale, in order to prevent their exhaustion; 2. to avoid malicious attacks on the machines directly. The Network Address Translator (NAT) is the router mechanism that handles the mentioned address translation to and from the outside [37]. The router firewall, on the other hand, blocks connections to certain addresses and ports on the local network.

Both of these are obstacles to P2P communication, because a node is not generally free to connect to other Internet nodes inside home networks. Should one try to do so, the target node's router will simply block the connection, as the machine inside the network did not request connectivity [38]. If a connection in the reverse way is also impossible due to both nodes possessing a NAT or firewall, P2P applications must solve the problem through so-called NAT traversal mechanisms. NAT traversal is made challenging by the fact that several types of NATs exist, depending on if a single local IP address is translated to only one or multiple external IPs, and the same with the connection ports (single or multiple translations). RFC 3489 [39] defines the original STUN (Simple traversal of UDP over NATs) standard, and the different types of NATs can be consulted there. Nevertheless, some other parts of the RFC were obsoleted by RFC 5389 [40], which also redefined the meaning of STUN to be "Session Traversal Utilities for NAT".

STUN (and STUNT [38]) cannot always achieve connectivity between nodes behind NATs, depending on their types. Symmetric NATs, in particular, are not possible to bypass with STUN, as every connection from an internal address to the outside will always map to a different IP and port during translation [39]. For nodes that cannot communicate using STUN, TURN helps them by specifying how to relay communication through a third node [41]. Another method for NAT traversal, which motivated the creation of TURN, is called the Interactive Connectivity Establishment (ICE) protocol [42]. Summarizing, ICE gathers specific connection candidates (pairs of IP addresses and ports) in both connecting nodes using STUN and TURN, and then tests connectivity between them until a pair of candidates is successful in connecting through the endpoints' NATs. This work and research will not delve further in the description of NAT types and their traversal methods.

2.2.5 Communication libraries and protocols

Many existing protocol implementations let programmers build P2P networks. We will go over a few specifications and libraries, old and modern, in order to define some possibilities for this research.

JXTA is an open source set of protocols for P2P networks, based on XML messages and designed to be independent of programming language, operating system, hardware and transport protocol [43]. JXTA is not an API but rather a specification of several protocols for such networks, with its main implementations (that do provide the respective APIs) existing in Java and C[10]. It specifies that peers should create an overlay network, so that communication is possible even from behind firewalls or NATs, or between different architectures. In this overlay network, nodes are allowed to move while still maintaining communication, as each node is assigned a unique ID, independent of location. Peers can have several roles, depending on their capabilities. The most evident distinction is between edge peers and special peers — also called "super-peers". The former generally have lower bandwidth; the latter have better features and are commonly tasked with the role of coordinating edge peers or relaying messages through firewalls [43].

Developments on the JXTA project are few today; though a new version of the Java implementation was released in 2011 (JXSE 2.7), the C implementation lagged behind and we currently assume that it is not being developed anymore. We also found the respective website and documentation unavailable.

Because our library should yield high performance, we have decided to implement it in the C++ programming language. We therefore opted to research C/C++ libraries for our efforts. Some current examples of free libraries for communication with an active implementation in these languages are *libjingle*[11], *ZeroMQ*[12] and *libnice*[13].

Libjingle is an open-source library whose purpose is precisely to allow programmers to build peer-to-peer applications. It is a package of functions used by Google to handle P2P sessions in its Google Talk application[14], and it closely resembles Jingle, which in turn provides support for sessions in mul-

[10] JXTA website: `http://jxta.kenai.com/`

[11] libjingle website: `https://developers.google.com/talk/libjingle/`

[12] ZeroMQ website: `http://www.zeromq.org/`

[13] libnice website: `http://nice.freedesktop.org`

[14] See `http://googletalk.blogspot.pt/2005/12/jingle-all-way.html`, by Google's Software Engineer, Sean Egan

timedia applications like VoIP. Libjingle supports connections through NATs and firewalls (it implements the ICE protocol and, therefore, also STUN), and also provides aid with handling proxies and parsing XML messages[15], which it uses for communication, similarly to JXTA.

ZeroMQ is an open-source asynchronous socket library aimed mostly at clusters and supercomputers and capable of providing concurrency capabilities for such systems. It is designed to handle a large number of connections simultaneously and support easy workload distribution and various types of communication patterns, including master-workers, pipelining or all-to-all connections[16]. Nevertheless, it is not its goal to handle the paradigm of P2P networks, being more suited to build server systems. It is, nevertheless, an example of how embedded communication patterns can simplify code.

Finally, similarly to libjingle, libnice also implements the ICE standard and automatically handles NAT traversal, becoming useful for creating P2P data streams with UDP or its pseudo-TCP implementation. Libnice is mostly suitable for multimedia applications, but the TCP-over-UDP option adds the necessary reliability for applications where packet loss is a problem, such as when running parallel algorithms.

2.3 MPI

Message Passing Interface (MPI) is a communications protocol based on message passing and independent of programming language, designed to work with most models of parallel computer systems [7]. Essentially, MPI warrants high-performing inter-process communication in a parallel program within a distributed memory system. However, given its high portability, MPI can also exist for shared memory architectures or hybrids of the two [7]. Two arguably well-known public examples of MPI implementations are MPICH[17] and Open MPI[18], though implementations exist in many languages, the most prominent of which being C, C++ and Fortran. These generally consist of a programmer API that contains many primitives for point-to-point and collective communications, both synchronous and asynchronous, making it possible to build applications that follow several distributed models [7].

[15] See `https://developers.google.com/talk/libjingle/developer_guide`
[16] ZeroMQ's patterns' specifications can be found at `http://rfc.zeromq.org/`
[17] Official website: `http://www.mpich.org/`
[18] Official website: `http://www.open-mpi.org/`

MPI is standardized and has wide use, typically by institutions that want to run demanding applications on a cluster of computers or a supercomputer for their personal use. It is common to find such distributed computing environments built with high-speed interconnects to reduce the communication overhead limitations of the model [44]. Due to the environments, distributed memory implementations are quite different from shared memory APIs like pthreads[19] and OpenMP[20]. Nevertheless, the distributed memory approach can be used even in programs running within a shared memory model system, as the two concepts can actually complement each other, by running multi-threaded applications on multiple processors in the same host. MPI implementations are free to choose which type of memory to use between processes, based on the environment.

Contrasting with the P2P environment, MPI was built mainly for contained groups of machines which are known and pose no threat. This means that security is not necessarily a relevant aspect in MPI applications [45], as its focus is on low latency, scalability and portability. For this reason, MPI is not ideal to support communication between computers in different networks, a model that faces problems with firewalls and NAT services, as mentioned beforehand. In addition, because of the way participating processes are started, MPI makes it hard to handle faulty nodes mid-processing (typically, if a node becomes invalid, the whole process cannot proceed) and the arrival of new nodes, which could still participate to some extent [46], making it somewhat inviable for direct volunteer computing use.

2.3.1 Fault-tolerant MPI

P2P-MPI[21] is a middleware running on the Java Virtual Machine that attempts to provide transparent fault handling (fault recovery is handled by the middleware and not the programmer) and automatic configuration and discovery of nodes for MPI, addressing some of its inherent problems. P2P-MPI is composed of three main modules: the Message Passing Daemon, responsible for dynamically finding participating nodes via the discovery service of JXTA, which the module uses; the File Transfer Service, for transfer of input, output and executable code between nodes; and the Fault Detection Service, which produces notifications of unavailable nodes during execution [47]. The basis of P2P-MPI is on the replication of processes, which is configurable by

[19] See `https://computing.llnl.gov/tutorials/pthreads/`
[20] Official website: `http://openmp.org/wp/`
[21] See `http://grid.u-strasbg.fr/p2pmpi/`

the user.

Also related, Volpex MPI [48] is both a project — VolPEx: Parallel Execution in Volunteer Environment — and an MPI library, with the objective of robustly executing MPI jobs in volatile environments such as public-resource computing, enabling jobs to progress even under frequent node failures. It tried to solve the aforementioned problem of mid-processing node failure and arrival by using mainly two features:

1. the efficient replication of MPI processes in the network (an approach frequently used to reduce this problem, as seen before with BOINC jobs, P2P-MPI and fault-tolerant applications in general), in which the slowest replicated nodes do not considerably hinder progress of the fastest and system progress is made by the latter.

2. the logging of messages in the sender, so that messages are kept and can be re-delivered later to nodes that fall behind. Nodes fail to keep up with the progress of the fastest machines due to either their slowness or the fact that they are recovery from a checkpoint.

A suggested improvement was the implementation of checkpoint-restart of processes. This means that nodes that arrive can start their job from the latest checkpoint of another node that is ahead, thus more efficiently replicating the already done work in case of failure. Volpex MPI uses some ideas previously suggested by implementations such as FT-MPI [49] and MPICH-V [46]. The first addresses these problems through the extension of MPI's specification to support communicator states such as "detected", "recovered", "failed", among others, instead of the simple MPI valid/invalid distinction. Faults are handled as desired by the application, at the MPI communicator level [49]. On the other hand, MPICH-V uses a memory of sent messages for posterior delivery [46], a concept similar to the message logging present in Volpex MPI. Despite these improvements towards volunteer computing, Volpex MPI is still not adequate for our target of running on nodes across different networks, as it only runs locally using MPI [48].

2.4 Speedup and Communication

In the interest of our research, we need to be able to formulate to some degree the benefits of a distributed model to volunteer jobs, so as to predict

what speedup is realistically expected from applications running in various nodes. Hence, we will briefly go over the Amdahl's and Gustafson-Barsis's laws of speedup, review the aspects that are specific to our environment and how communication overhead can be included in these known equations.

2.4.1 Amdahl's Law

Amdahl's Law, formulated after the work of Amdahl [50], states that, if we define s as the non-parallelizable fraction of time spent by an algorithm (its serial percentage) and T_{seq} the time that the same algorithm takes to complete on a single CPU, then, on a parallel system with N processing units, that algorithm will take T_{par} time, defined as:

$$T_{par} = s \times T_{seq} + \frac{(1-s)\,T_{seq}}{N} \tag{2.1}$$

That is, the parallel section of the algorithm would be equally split among all CPUs, so that its running time would effectively be divided by N. This allows us to understand what speedup S that parallel system would achieve when compared with the sequential or single-CPU system. The formula of speedup, obtained by dividing the sequential running time by the parallel running time, then becomes, according to Amdahl's Law:

$$S(N) = \frac{T_{seq}}{s \times T_{seq} + \frac{(1-s)\,T_{seq}}{N}} \tag{2.2}$$

When Amdahl's work was first published, it brought concern that parallel systems were very limited performance-wise. For example, even if the parallelizable fraction of an algorithm is only 5% of the total time, we can observe from the speedup formula that the maximum achievable speedup (that is, the speedup for an infinite number N of processing units) is only $\frac{1}{s} = \frac{1}{0.05} = 20$. This means that no matter how much parallel processing power we have to run the algorithm, we would only be able to run it 20 times faster than the sequential version, as the algorithm can never run faster than the time it takes to run its sequential fraction. Furthermore, this is for a parallelizable fraction of 95%, which can actually be much lower and subject to several kinds of overhead, meaning that in a real situation the benefits of parallelization can be even fewer.

2.4.2 Gustafson-Barsis's Law

Taking into account how algorithms are generally run in modern parallel systems, Amdahl's Law might not clearly reflect what a parallel system is capable of and how it can use its resources. When running a massive computer problem in multiple machines, the problem size is usually increased according to the number of machines, to a point where the sequential fraction becomes close to negligible and we can achieve an almost linear speedup. Furthermore, if the algorithm is simply run sequentially on each machine, we are not even trying to reduce the parallelizable fraction and total runtime of the algorithm; we are, in fact, keeping the running time constant and simply executing N tasks in parallel, so that all of them complete in that time. This means that the number of work done is increased while maintaining the sequential factor constant in the parallel part; hence, the sequential part never dominates the parallel part as N grows, challenging Amdahl's Law. These different perspectives of how speedup is attained are possible operating examples of the scaled speedup equation suggested by E. Barsis and later written by J. Gustafson [51], which created the Gustafson-Barsis's Law:

$$S(N) = \frac{s + (1 - s)\, N}{s + (1 - s)} \tag{2.3}$$

In the above equation, which can be simplified to $S(N) = s+(1-s)\, N$, the problem size is intended to scale with the number of processors N, meaning that the sequential time, $s + (1 - s)\, N$, also grows linearly with N and, thus, so does system speedup.

Yuan Shi later showed that Amdahl's and Gustafson-Barsis's laws were mathematically equivalent, despite outputting different speedup values for similar values of s [52]. The reason for this, as we mentioned, is a matter of perspective, as both laws are in fact the same, but formulated with a different concept of *serial percentage* of an algorithm.

2.4.3 Communication overhead

In a distributed environment, there are some additional issues that have to be accounted for and that do not exist in a contained multi-core machine. One very important factor in distributed computing is communication. Contrasting with a single-CPU machine with shared memory environment, where a program needs no communication, in distributed computing and even in

multiple-CPU machines, communication exists when exchanging data between multiple processes. When a processor thread is busy communicating data, it cannot do processing at the same time. Because of this, increasing the number of CPUs participating in the algorithm can actually be detrimental to the running time, as more communication can be introduced between them, depending on the application. Furthermore, the distance between connected CPUs can influence the time it takes for data to reach the target process — even at the speed of light — as they might be in different boards, machines or rooms (even countries, if global communication is considered viable). The time it takes for intermediate routers and access points to process packets of data also adds to this latency. The laws of Amdahl and Gustafson-Barsis are, therefore, not acceptable for calculating speedup in these environments, as they have no consideration for the overhead that communication introduces in such a distributed parallel system.

In [53], Li and Malek recreated Amdahl's Law for a multiprocessor environment. Generically, in their work, each processor/node exchanges a certain quantity of data and does a certain number of tasks, with each of these taking some amount of time to complete. They also considered that processing might not be uniform among nodes, implying that one or more nodes might finish their jobs sooner or later than others and affect total execution time. In their research, two different possibilities about communication dictate the speedup that can be achieved: communication can range from fully parallel (i.e. data sent between nodes is unaffected by other communications in the network and can be done immediately) to fully sequential (i.e. every transfer is made in order, one after the other) [53]. This is essentially the difference between considering only the longest data communication and summing the total time of all communications during execution.

Based on the parallel execution time of Amdahl, in Equation 2.1, a simplified formula that averages the singular communication and processing times per node was written by the authors as:

$$T_{par} = s \times T_{seq} + \frac{(1-s)\,T_{seq}}{N} + T_{comm} \tag{2.4}$$

Where:

- T_{seq}: running time of the algorithm in a single processor system
- s: percentage of the algorithm that we cannot parallelize
- N: number of participating nodes
- T_{comm}: total communication time in the parallel algorithm

This equation aggregates the individual jobs and communications of each node and assumes the processing is uniform. Nevertheless, as stated above, Li and Malek also defined the formulas for non-uniform behavior and equations with much finer granularity of communication and processing, which we are not going to further explore here. Based on Equation 2.4, speedup is then given in the following range, as the system shifts from fully sequential to fully parallel communication (left and right part of the equation, respectively):

$$\frac{T_{seq}}{s \times T_{seq} + \frac{(1-s)\,T_{seq}}{N} + T_{comm}} \leq S \leq \frac{T_{seq}}{s \times T_{seq} + \frac{(1-s)\,T_{seq}}{N} + \frac{T_{comm}}{N}} \tag{2.5}$$

This equation can be simplified to:

$$\frac{1}{s + \frac{(1-s)}{N} + \frac{T_{comm}}{T_{seq}}} \leq S \leq \frac{1}{s + \frac{(1-s)}{N} + \frac{T_{comm}}{N \times T_{seq}}} \tag{2.6}$$

2.5 Distributed Applications

Applications in most Desktop Grid systems are of the embarrassingly parallel type. This means that, running sequentially, such applications are easy to transfer to a parallel system, as they contain parts that can run independently and in a random order in each processor. This is the case with parameter sweeping applications (essentially consisting of cycles that test each possible parameter with no relation to previous ones) and is the kind that thrives on volunteer grids. However, many applications do not follow this model; in fact, some are already hard to parallelize in shared memory systems, and even harder to program in a distributed memory model. Nevertheless, there are applications that, in fact, benefit from running on a communication-enabled system, be it following a master-workers model or a fully distributed one.

2.5.1 Non embarrassingly parallel applications

In [54], the authors propose a multi-objective evolutionary computing algorithm that runs in parallel on a P2P network. The concept of Island Models is used, in which several populations of individuals in the algorithm

are created initially, each on a different node, and evolve in parallel, occasionally exchanging their best individuals (at intervals of several generations/ iterations). In their work, the exchange consists of a subset of the Pareto front[22], letting the participating nodes diversify their search space by receiving individuals from other populations. In the authors' research, the distributed model follows the master-workers approach (named "dispatcher-worker paradigm" in the paper), in which the master is tasked with handling the migration of Pareto fronts. This is an application example that could hardly run on a single machine or independently on multiple machines (like happens in BOINC), as the effectiveness of using these Island Models mostly only shows when many populations exist in parallel, exchanging elements and affecting each other's development.

Papers were also published regarding parallel and distributed algorithms of Particle Swarm Optimization (PSO), which is another evolutionary computing approach to search spaces, this time based on the innate movement of birds and other animals in a group. For instance, in [5] the authors implemented a fully distributed PSO based on the asynchronous propagation of objects along nodes. Each node possesses a small swarm of particles and sends out some of its best solutions from time to time. There is no waiting for results on the various nodes, as communication is asynchronous, which means that the algorithm achieves a communication overhead theoretically close to zero. The authors also argue that this PSO algorithm has reduced need for population size control (when compared with other distributed evolutionary algorithms) and it does not require much information to be known about the global population when creating new solutions/individuals. Figure 2.4 shows an example of communication paths in a distributed application such as this.

Another example is the suggestion of a distributed model using MapReduce, which can be seen in [55], where the authors propose a P2P MapReduce system with the participating nodes dynamically acting as slaves or masters, with the objective of preventing the premature ending of the process when a single master fails. Yet another work refers Ant Colony Optimization in a distributed environment [56], similar to evolutionary approaches previously seen. In [57], the authors implement a distributed algorithm for a numerical simulation of the propagation of electromagnetic waves — another non-embarrassingly parallel application. Still other examples of applications that do not suit the default master-workers environment of volunteer com-

[22]Pareto fronts consist of the non-dominated individuals or, put in another way, individuals that are better than all others in at least one combination of objectives.

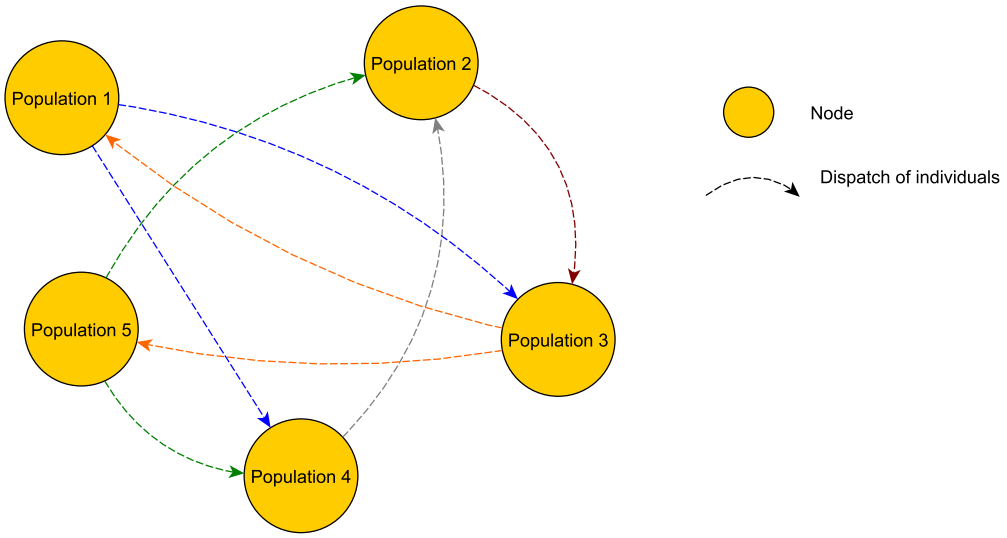


Figure 2.4: Parallel evolutionary algorithm — each peer/node has a population.

puting systems are, for instance, alpha/beta pruning of the search space in game theory (which is, essentially, a type of branch-and-bound) and parallel tempering, which consists in a simulation of physical systems with Monte-Carlo (random) methods and the exchange of neighbor information. Replica-Exchange Molecular Dynamics is an example of this [58].

2.5.2 Generalization

Based on the knowledge that communication overhead exists and is significant, applications could typically work on a large-scale distributed system if they depend on a low or, at best, moderate amount of communication data. This means that random search methods — like evolutionary algorithms or simulated annealing — are good fits, as each node can search a different section of the search space and still exchange information. The degree to which communication overhead impacts the speedup of an application is something that we will see later in Chapter 4.

As the environment of systems like BOINC is also expected to be volatile, applications in which nodes can improve global progress in a relatively short time (the time they are available) are also good candidates. With branch-and-bound methods, for example, when a node arrives it can know from other nodes what the current best value is and start searching from there, cutting-

off useless branches are potentially contributing to the search in a shorter timespan. This is a kind of checkpoint-restart, as mentioned in Section 2.3.1.

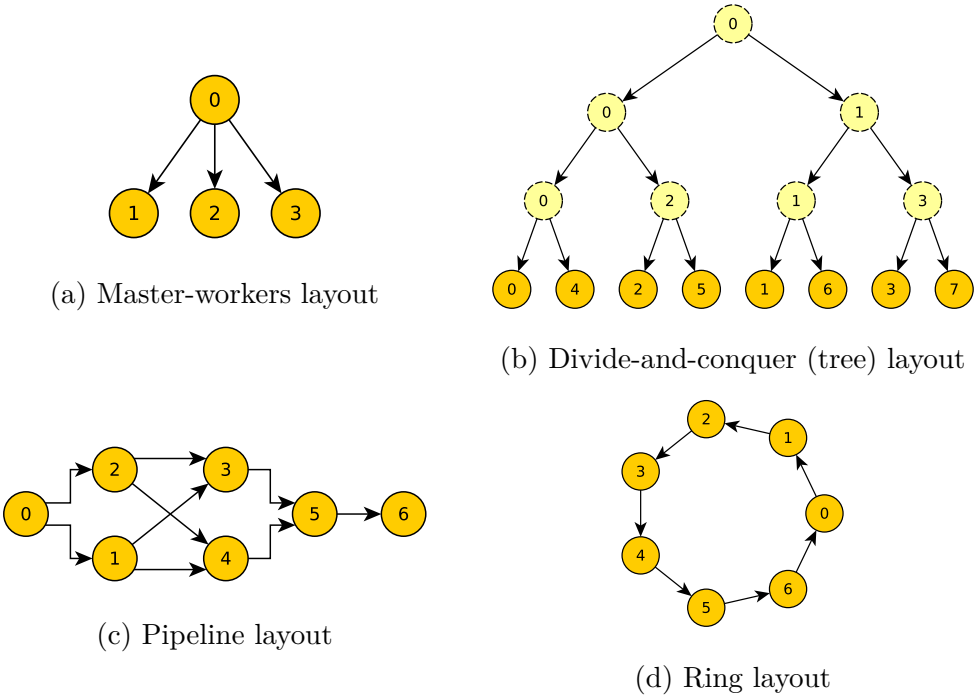
Finally, like seen before in Sections 2.2 and 2.3.1, one widely implemented solution in fault tolerance is the replication of data throughout the network. As our work centers around P2P networks, applications that can improve from the existence of nodes with similar information are also good examples, if they maintain low communication rates overall. Again, random search algorithms fit this condition, as the nodes' states could be similar but their current search areas different.

2.5.3 Communication patterns

Another topic of interest for us is the fact that parallel applications in high performance computing often follow common patterns in the interactions between participating nodes. As we have seen in volunteer computing systems such as BOINC, the main pattern used is the master-workers model, seen in Figure 2.5a. Another pattern that also shows frequently is the divide-and-conquer model, in which the communication between nodes assumes a tree-like structure of branchings, with each node dividing its data into sections, keeping part of the data and sending the remaining to the peers below for processing (Figure 2.5b). This can be seen in algorithms that divide space or objects in sections, such as a solution for the "closest pair of points" or the Barnes-Hut simulation to the n-body problem.

In a **pipeline**, each node has explicit *upstream* and *downstream* peers, from/to where the data comes/goes, respectively. The data is passed along the nodes as a stream, each node outputting the input of the following nodes (Figure 2.5c). One example of pipelining is the post processing of images, where filters are applied in succession and the result sent to the next node for further handling. Pipelining includes the more specialized ring layout (Figure 2.5d), where nodes are disposed in a circular way and the first node is also the last. Ring layouts are useful for more specific objectives, such as the election process known as Chang and Roberts algorithm [59]. Lastly, we will also mention the **all-to-all** pattern, where every node can communicate with all other nodes. Some parallel evolutionary applications that we have seen can use this pattern to freely exchange individuals between populations in different nodes.

There are also frameworks that abstract these patterns of communication and even some algorithmic structures of computation (branch and bound, dy-



(a) Master-workers layout

(b) Divide-and-conquer (tree) layout

(c) Pipeline layout

(d) Ring layout

Figure 2.5: Common communication layouts

namic programming, among others). These environments are generally called "algorithmic skeletons" and enable the programmer to implement complex algorithms with less error propensity and effort. This is achieved by writing code in terms of generic skeleton constructs, in detriment of explicit "for" cycles over ranges, "if" conditions and other instructions [60].

The research in [60] shows several examples of such skeleton frameworks, their parallelism classification (task, data or resolution level) and brief description, among other things. In the language of our implementation alone, C++, the authors name ASSIST, Muesli and SkeTo, for instance. ASSIST defines a language to create parallel applications as a graph of modules and their interactions. It then provides fault-tolerance, load balancing and other mechanisms to the nodes of this structure. Muesli supplies skeletons via elements typical of functional programming languages, such as currying and high order functions, written in C++ methods. It also supports skeletons for distributed patterns like pipelines or task farming. Finally, SkeTo is distinct from other algorithmic skeletons in the fact that it essentially implements operations over parallel data structures such as lists, matrices or trees. As we will see in later Sections, we have used an approach similar to an algo-

rithmic skeleton in implementing layouts — an essential part of our library — nevertheless not following any of them directly.

Chapter 3

Internet Group Communication Library

3.1 Overview

In this Chapter we will detail every relevant feature of the implemented library, which we named "IGCL", short for "Internet Group Communication Library".

The base idea of IGCL is to allow a group of nodes to execute an application in parallel by sending and receiving data among them, much like in MPI. The advantage of IGCL is allowing these groups to function either locally or when peers are separated by the Internet, running on common home networks. This is possible in part because IGCL automates the process of connecting peers to each other and includes NAT traversal techniques.

In this work, we define that a "peer group" is composed of several nodes, one of which is special and called the "coordinator". Other nodes in the group are simply called "peers". The job of the coordinator is to know all the necessary information about the group and manage registration, connections and termination of peers. These peers can, depending on the environment where they run, connect to each other directly with a simple socket *connect* or with the help of the ICE technique for NAT traversal, which we detailed in Section 2.2.4. If no better option is possible, there is also the possibility of peers sending messages to their target peer by relaying them through the coordinator. This is essentially similar to the third node in TURN relayed communications. IGCL tries to establish these connections automatically,

first trying the simplest solution — using direct socket connects — and only then moving on to the ICE mechanism or coordinator-relayed messages, the last of which can be disabled by the programmer if not desired.

The library allows the programmer to use basic communication primitives such as *send-to*, *send-to-all*, blocking and non-blocking versions of *receive-from* and *receive-from-any*, among others. It also allows the use of configurable or predefined group layouts, which provide a means to automatically place peers inside a well defined graph of communication and specify which nodes come "before" or "after" them in the structure. This makes it possible to automate the general distribution of data in applications that use common patterns of communication, such as task-farming or divide-and-conquer. This is a feature similar in concept to what the algorithmic skeletons mentioned in Section 2.5.3 provide, although it does not try to follow any of them in design.

We will begin with a comprehensive example of IGCL usage, then moving on to some specifics about existing methods, layouts and conventions, and finally explain the technical details of the library. We leave the full documentation of its public API to Appendix A.

3.1.1 Usage example

Before proceeding, the reader should know that a coordinator and a peer are respectively represented by the C++ classes igcl::Coordinator and igcl::Peer. Both types internally function in different ways; however, most of their API methods are the same, as they inherit the capabilities of the same base class, igcl::Node.

In Listing 3.1, we give an example of IGCL usage by building a buffering scheme for a matrix multiplication algorithm. As part of the algorithm, we will need to send one of the matrices — lets call it "matrix B" — to every node and then let each of them multiply the whole matrix B by the rows that they receive from the other matrix — "matrix A". The code here presented is executed on a coordinator process and omits the origin of *matrixA*, *matrixB* and *resultMatrix*, which all are pointers to memory spaces of MATSIZE × MATSIZE elements and are globally accessible. We will give a step by step explanation of this code to guide the reader through the usage of IGCL.

```
1 #include "igcl/igcl.hpp"
2 using namespace igcl;
```

```
3
4  Coordinator * coordinator;
5
6  void work() {
7      coordinator = new Coordinator(12345);
8      GroupLayout layout = GroupLayout::getMasterWorkersLayout(8);
9      coordinator->setLayout(layout);
10     coordinator->start();
11     coordinator->waitForNodes(layout.size());
12
13     auto buffering = new NBuffering(2, MATSIZE, 1, sendJob);
14     buffering->addPeers(coordinator->downstreamPeers());
15
16     coordinator->sendToAll(matrixB, MATSIZE * MATSIZE);
17
18     buffering->bufferToAll();
19
20     while (!buffering->allJobsCompleted()) {
21         peer_id sourceId;
22         DATATYPE * result = NULL;
23
24         coordinator->waitRecvNewFromAny(sourceId, result);
25         uint row = buffering->completeJob(sourceId);
26
27         for (uint i=0; i<MATSIZE; ++i)
28             resultMatrix[row*MATSIZE+i] = result[i];
29         free(result);
30
31         buffering->bufferTo(sourceId);
32     }
33
34     coordinator->terminate();
35 }
36
37 void sendJob(peer_id id, uint row) {
38     coordinator->sendTo(id, matrixA+row*MATSIZE, MATSIZE);
39 }
```

Listing 3.1: Coordinator code for matrix multiplication with buffering.

The first step towards using the library is including the IGCL header, *igcl.hpp*, as in line 1. Afterwards, we need to create an object of either of the previously mentioned classes. Listing 3.1 shows this construction is line 7 for

the coordinator only. In this case the constructor receives its listening port as argument, for incoming connections. Listing 3.2 shows the constructors for both classes. In addition to the listening port, a peer node would also be given the port and IP address of the group coordinator.

```
igcl::Coordinator(int port)
igcl::Peer(int port, const std::string & coordIp, int coordPort)
// examples:
auto node = new igcl::Coordinator(12345);
auto node = new igcl::Peer(50123, 10.5.1.3, 12345);
```

Listing 3.2: Creating the main IGCL objects

To prepare any of these objects for communication, we need to call their *start* method. This will make the node listen on the specified port for new connections and prepare a thread to handle received messages. In addition, the Peer's start method also goes through the process of registration with the group coordinator. The complete registration process is detailed in Section 3.2.4. Before doing that, however, we want to show the reader how to configure a peer layout, which must be done before starting the coordinator.

In lines 8 and 9 we can see the creation and setting of a *GroupLayout* object. In this case, we are using a predefined master-workers layout with 8 fixed nodes and passing it to the Coordinator object with *setLayout*. This transforms the coordinator into the master of 7 worker peers that do not exist yet. Other layouts exist in IGCL, and we will talk about them in Section 3.1.3, but we will continue the example for now.

After setting the layout, we can finally start the object. As we said before, after starting, the object is ready to receive connections from other peers and process messages. Nevertheless, the user's code should normally wait for the arrival of every peer before beginning. The *waitForNodes* method can be used for this, blocking the thread until the specified number of nodes forms the layout. This number includes the coordinator node itself, meaning that waiting for one node will immediately return, as the coordinator is already part of the layout.

When the method returns, we can now execute the algorithm. In this case, we opted to showcase the NBuffering class of IGCL, which can dynamically distribute jobs to nodes and keep them buffered with more work to do as they complete previous jobs. The construction of the buffering object, in line 13, essentially defines the level of buffering (in this case it is 2, which is equivalent to double-buffering), the number of jobs (we intend to buffer rows

of matrix A as jobs, so there are MATSIZE of them), the size of each job (1 row per job) and what function is used to send a job. For more details about buffering, refer to Section 3.2.5 and Appendix A.5.

An NBuffering object also needs to know which peers are available to work. To this end, line 14 sets a group of peers as workers. We used the method *downstreamPeers*, available in both Peers and Coordinators, which returns a vector with all peers that are receivers of information from the calling node. In the example, the master-workers layout internally defines that the downstream peers from the coordinator are every other node — and that these nodes have no downstream peers at all — thus making it easy to set the peers that should be buffered.

Before starting the buffering process, we will begin by sending matrix B to every peer. This can easily be done by calling the *sendToAll* method, which in this case takes a pointer to a memory location and the number of elements it contains. The type of the elements is not important (it can be int, float, double, among others), as it is automatically deduced from the type of pointer (see Section 3.2.1). The method *sendToAllDownstream* could also be used, as it sends the data to all downstream peers and is, therefore, equivalent in this case (all nodes are downstream from the coordinator).

Now we begin the buffering process. Lines 18–32 show how we start by buffering jobs to every node and then receive data in a loop until all jobs are completed. The *bufferToAll* method relies on the *sendJob* function that we passed to the constructor to send the jobs themselves. The buffering class will call this function with the target node ID and job index every time it wants to send/buffer something. In our example, *sendJob* calls the *sendTo* method on the Coordinator object, giving it the target ID, a pointer to a memory location, and the number of elements to send from that memory. This job consists of a row of matrix A, as we have mentioned.

Inside the loop, in line 24, we use the *waitRecvNewFromAny* method to block until the reception of a job result — which is a row of the final result matrix — from any peer. This method will fill the ID of the source, allocate memory for the received data and make the given pointer reference it. The receive methods that are named "New" (see Section 3.1.2 for more details) can also fill a third argument with the number of elements in the pointed memory, but in our case we know the size of the result to be MATSIZE.

Once we have the ID of the peer, we can query the buffering object for the index of this job, letting us know where to insert the received row in the result matrix. NBuffering keeps the indexes of sent jobs internally, precisely

to allow such usage. This, of course, implies that jobs are received in the same order that they were sent in. If this is not the case, peers should be given the index of their row/job along with it, and then use that index when returning the result to identify the job.

Still inside the loop, we free the memory pointed to the received result, which is not needed anymore, and immediately buffer another job to the peer with *bufferTo*. If there are no more jobs to buffer, the buffering class will do nothing. When the algorithm finishes, we call the *terminate* method on the Coordinator to cleanly exit the library. The call will trigger a message to all group peers that will force them to terminate, in addition to terminating the coordinator itself.

All these methods and the remaining public API are detailed in Appendix A for reference.

3.1.2 Naming conventions

IGCL follows some naming conventions of our choice. We previously mentioned the labeling of downstream peers as the ones that are receivers of information from this peer — i.e. peers that are "after" in the layout. Similarly, upstream peers are located "before" in the layout. If a node A is downstream from node B, B is upstream to A. Nodes are able to send data to peers before them as they do with peers after, seeing as they have an active connection to each of them. This generally happens when returning results to these nodes.

We have at our disposal four basic send methods: *sendTo*, *sendToAll*, *sendToAllDownstream* and *sendToAllUpstream*. These are detailed in Appendix A.1.1 of the documentation, but their functions should be understandable from their names. Methods that end in "To" or "From" have the related target/source node ID specified as the first argument, as happens with *sendTo* or the various "receive from" methods.

As can be seen from the API in Appendices A.1.2 and A.1.3, there are 8 different receive methods available, which result from all combinations of blocking/non-blocking, receive from-one/from-any, and allocates/does not allocate memory. For instance, the following method provides a way to receive a single value of any type from any peer:

```
template<typename T>
result_type waitRecvFromAny(peer_id & id, T & value)
```

In its name we can see several keywords/expressions, namely *wait*, *recv* and *from any*. *Recv* simply means that this is a receive method. *Wait* means that the method blocks until there is something to read. Finally, *from any* denotes that the method will read the value from any peer (the first whose data arrives) and not a specific one. It also sets the ID of this peer in the argument *id*.

Likewise, we now present another method, which has the opposite keywords of the previous method:

```
template<typename T>
result_type tryRecvNewFrom(peer_id id, T * & data, uint & size)
```

In this case, the method is non-blocking (it returns NOTHING when there is nothing to receive), as given by the keyword *try*. Furthermore, the *new* keyword means that new memory has to be allocated, in this case to store data of unknown size and make *data* point to it. Lastly, this method uses *from* instead of *from any*, meaning that the method will only try to receive values that come from the peer specified by the *id* argument.

As we said, all combinations of these keywords exist; the list of possible receive methods is then composed of *waitRecvFrom*, *waitRecvFromAny*, *waitRecvNewFrom*, *waitRecvNewFromAny*, *tryRecvFrom*, *tryRecvFromAny*, *tryRecvNewFrom* and *tryRecvNewFromAny*. All of these are further explained in Appendices A.1.2 and A.1.3.

3.1.3 Group layouts

As seen in Section 3.1.1, the IGCL group coordinator supports specifying a group layout via a set method, *setLayout*, which takes a GroupLayout object that defines which peers communicate with which, and uses that information when registering arriving peers. The GroupLayout objects can be created either manually, using *from* and *to* methods, or via some predefined common layouts, seen in Listings 3.3 and 3.4. We will focus on predefined layouts. For more information about manual layouts refer to A.4.3 in the documentation Appendix.

Listing 3.3 shows all fixed layouts of IGCL; i.e. those that take a fixed number of participating nodes as argument (always including the coordinator). These layouts are used in algorithms that expect a specific number of nodes; the coordinator can wait for that number of peers to arrive by using *waitForNodes*. As a specific of the tree layout, we also provide the argument *degree*, which is number of sections in which data is divided at each tree depth level (i.e. its branching factor). In all these predefined layouts, IDs are attributed in a certain order, from 0 to $nNodes - 1$. This order and general pattern layout should be clear from Figures 2.5a to 2.5d, from Section 2.5.3.

```
const GroupLayout getMasterWorkersLayout(uint nNodes)
const GroupLayout getTreeLayout(uint nNodes, uint degree)
const GroupLayout getPipelineLayout(uint... nNodesOfSection)
const GroupLayout getRingLayout(uint nNodes)
const GroupLayout getAllToAllLayout(uint nNodes)
```

Listing 3.3: Predefined fixed group layouts

Furthermore, IGCL defines two free-formed layouts, which do not need a specific number of peers. Free-forming works in layouts that have a lenient structure in which the addition of a new node might not change the way an algorithm works. Listing 3.4 reveals these two layouts to be the master-workers and all-to-all. These kinds of layouts with no specific number of nodes can be helpful in an embarrassingly parallel application, where nodes can arrive at the system during execution and still receive data to process. They can also be used in applications with replication of nodes, where an arriving node can receive from other nodes the current state of processing (for example, the current best solution found, which works as a bound). Likewise, nodes can leave at any time and let their data be processed by other nodes. This is especially useful when used in conjunction with IGCL's buffering class.

```
const GroupLayout getFreeMasterWorkersLayout()
const GroupLayout getFreeAllToAllLayout()
```

Listing 3.4: Predefined free-formed group layouts

Layouts directly affect the values returned by methods *downstreamPeers* and *upstreamPeers*, as well as the related *nDownstreamPeers* and *nUpstreamPeers* that return their sizes. They also affect the usage of higher order functions. These functions are two groups of methods seen in Appendices

A.1.4 and A.1.5 of the documentation, which are used to ease the distribution and collection of results in two common interaction patterns: master-workers and divide-and-conquer/tree. Later, in Section 4.4, we will briefly demonstrate their usage. We should note that methods for the master-workers and divide-and-conquer patterns are only compatible with the layouts returned by functions *getMasterWorkersLayout* and *getTreeLayout*, respectively.

Layouts also define if the non-coordinator peers know the total number of peers in the group or not, given by the method *getNPeers*. This is affirmative in the case of fixed layouts and negative otherwise, as the value is set upon registration with the coordinator. Nevertheless, peers know their connected nodes and their location in the layout (up or downstream), whatever the used layout.

Similarly to MPI, we can use *getId*, which returns the ID of the node, to write code like in Listing 3.5, in which different methods are called for the coordinator (ID 0) and the remaining nodes. In conjunction with *getId*, *getNPeers* can be used by peers in fixed layouts to calculate their share of work based on their ID and number of working peers. In the given example, however, we take a simpler approach to the matrix multiplication example, this time without buffering.

```
1  if (node->getId() == 0)  // master distributes data to slaves
2  {
3      node->sendToAll(matrixB, MATSIZE * MATSIZE);
4      node->distribute(matrixA, MATSIZE, MATSIZE, iniRowIndex,
           endRowIndex);
5  }
6  else
7  {
8      node->waitRecvNewFromAny(masterId, matrixB);
9      node->recvSection(matrixA, iniRowIndex, endRowIndex, masterId);
10 }
```

Listing 3.5: Different calls by checking the ID

As seen, *sendToAll* is used to send matrix B to every peer and *waitRecvNewFromAny* to receive it in each node. Dividing matrix A equally among all nodes is, in this case, done through the specialized methods for master-workers task distribution, provided by *distribute* and *recvSection*. These are part of the higher-order methods we mentioned before. Complementary methods exist for collection of final results, aptly named *sendResult* and *collect*. These four methods and a more complete example of matrix

multiplication are exemplified in Listing B.3 from the "code examples" Appendix.

3.2 Technical details

IGCL is implemented in C++ and uses some features of its most recent version, C++11. Among these, we can mention std::function pointers, std::thread, the related std::mutex and std::condition_variable, *for each* cycles, the "auto" keyword, variadic templates, and perfect forwarding of object references, which can be done through *rvalues* [61]. We took advantage of these features for commodity and, in some cases, performance, without relying on external libraries.

Since C++11 does not provide a new socket programming interface[61] and we did not want to depend on the Boost[1] or POCO[2] libraries, IGCL utilizes the default C sockets — the Berkeley (BSD) socket interface[3] — internally to send and receive messages using TCP, assuming nodes can establish a direct connection without the help of the ICE technique.

For communication between nodes behind NATs we decided to use *libnice*, which we previously described in Section 2.2.5. Libnice is the only dependency of the library, though libnice itself depends on GLib[4]. As IGCL is not pre-compiled, the programmer is required to include the *igcl.hpp* header file to access the public API. If he/she wishes to run an algorithm in a local cluster where nodes know each other's locations, libnice is not needed; consequently, we decided to include a preprocessor definition recognized by IGCL to compile without libnice functionality: DISABLE_LIBNICE. Thus, the programmer can use the library by first defining DISABLE_LIBNICE and then including the library, as shown in Listing 3.6.

```
1 #define DISABLE_LIBNICE
2 #include "igcl/igcl.hpp"
```

Listing 3.6: Including IGCL while disabling libnice.

[1] See http://www.boost.org/

[2] See http://pocoproject.org/

[3] See http://pubs.opengroup.org/onlinepubs/9699919799/functions/V2_chap02.html#tag_15_10

[4] See https://developer.gnome.org/glib/

3.2.1 Messages and data

In IGCL, a message involves three sends of data: firstly the message type, then the size of the data block, and finally the data block itself. The type of message is never seen by the programmer, as it is only used internally to recognize messages and execute the actions associated with them. Several types are used during registration, some more for send-to and send-to-all operations, relayed communication, termination, among others. The size of the data block lets the application know exactly how many bytes are coming, allocate the necessary space and progressively read data from the socket descriptor until everything is received. As the number of bytes in the message type (1 byte) and in the size header (4 bytes) are known, IGCL can deal with all messages in the same way, first reading a type, then a size and lastly the data block, whose size becomes known via the size header.

It would be possible to send these three message parts as a single block instead of using one send call for each, if all were written to a secondary buffer before sending. We did not test this approach to see how it compared to ours, performance-wise. On a related note, we are aware of the existence of Nagle's algorithm, which can end up joining data from multiple sends into one block at the cost of slightly higher send delays. Disabling the algorithm did not bring us performance advantages, as far as our tests could tell.

As for handling of data in IGCL, send and receive methods are C++ templates that allow the programmer to send any type of data through the network without runtime checks (methods with different type parameters are automatically create by the compiler). This approach releases the programmer from the necessity of having to cast memory pointers and specify sizes in bytes whenever he/she wants to send or receive data. With templates, the compiler can easily infer the total size in bytes of an array just from the number of elements it contains, as it already knows the size of the type. Send methods also work independently of connection types, as the library understands all possibilities. If a send-to-all method involves sending data to peers with different connection types, IGCL will handle this automatically.

Although any (reasonably sized) type of object can be sent through a connection, IGCL does not provide serialization — objects are, in fact, internally sent as arrays of bytes —, which means that pointer references inside objects will not be followed and only their value will be sent (becoming invalid on another process). Nevertheless, IGCL can correctly send and receive std::string objects as a special case. To avoid programming errors, the sending of a pointer type as a value is not allowed and will trigger a compile

time error. We should note that IGCL currently only supports little endian systems, due to being tested only on Intel x86 architectures.

3.2.2 Threading and blocking queues

The reception of messages is handled through a socket *select* or libnice callback functions; either way, messages are immediately checked and processed based on their type. Most types represent messages that are internal to the library, but others are sent by the programmer's application and are not always processed by his/her code the moment they arrive, as the algorithm can be doing other things. IGCL's solution to avoid blocking until the user's code handles the message is to use queues for the application's messages, where they can be stored for later extraction. Thus, public API receive methods are not directly linked to sockets, but to queues of data.

When a message from user code arrives, its source node is obtained from the socket descriptor or stream it arrived from, and the message itself is placed in a queue that ICGL associated with that source as part of the node's registration. Figure 3.1 illustrates this process. This is all done by the receiving thread and involves allocation of memory to save the received message. A performance difference exists between calling *malloc* (C function) and calling *new* (C++ operator), as the latter also calls the objects' type constructor. We decided to use the faster of the two, *malloc*, as we allocate memory to immediately write over it with the received bytes, invalidating any useful construction of objects from *new*. This implies that the user must take care of freeing the received data at some point, as written in the documentation for the relevant methods (see Section A.1.2 of the Documentation Appendix).

Although only one queue is used per source, there is another global queue, simply called the "main queue", where IGCL puts references to received messages from all sources, in the order they are received. These is IGCL's mechanism to support receive-from-any methods, when the source is unimportant. What this means internally is that, when dequeuing elements from the main queue, they are also removed from the respective queue; likewise, when dequeuing from an individual queue, the front of the main queue is checked for a reference to this queue for deletion. If it is not found because it is behind a reference to another queue (of another node's message that arrived first), a counter will be set with the current number of invalid references that exist in the main queue for this queue. These counters will be used in the next check

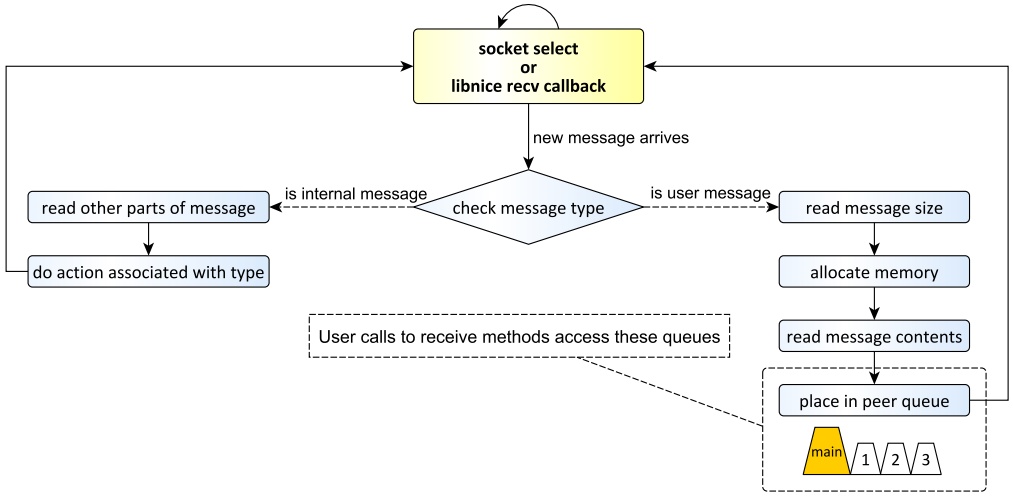


Figure 3.1: Reception of messages and queue storage in IGCL.

of the queue front. With this approach, we avoid using a secondary thread for periodic garbage collection on this special queue, at the cost of letting it temporarily grow to a potentially large number of invalid elements — until a dequeue in some individual queue removes them from the front.

All these queues are thread-safe and have a blocking dequeue method, which supports that threads wait until there are elements to dequeue (placing elements in these queues wakes threads that are waiting). This is useful to implement the blocking receive methods in the public API (seen in Appendix A.1.2). The queues also implement a try-dequeue, that lets threads test the queue for the presence of elements and return a negative value if they could not dequeue anything (in Appendix A.1.3). Although the use of these internal queues is thread-safe, they are not used or seen by the programmer directly (only indirectly, in calls to receive methods), and IGCL itself does not guarantee thread safeness if multiple threads access the public API concurrently.

3.2.3 Performance

When implementing send and receive methods, the handling of messages, and the library in general, some care was taken with the performance of code. Whenever possible, we avoided repeating our own instructions and allowed the compiler to generate code that was efficient. See Listing 3.7 for the IGCL

implementation of the public method *sendTo*. We can see that the method is templated to support multiple types and one of its arguments is actually a variadic template (which can be composed of several arguments, even of different values) passed via an rvalue reference (see [61]). As the method itself does not even touch the arguments in data, it simply forwards them along to an auxiliary method, exactly as they were received.

```
1  template <typename ...T>
2  result_type sendTo(peer_id id, T && ...data)
3  {
4      if (!knownPeers.idExists(id))
5          return FAILURE;
6
7      const descriptor_pair & desc = knownPeers.idToDescriptor(id);
8      result_type res = auxiliarySendTo(desc,
           std::forward<T>(data)...);
9      return res;
10 }
```

Listing 3.7: Implementation of the *sendTo* method.

The use of a variadic template might seem nonsensical, due to the fact that only two constructions of send methods exist: one that sends a single value and another that sends an array of values with a certain size. Nevertheless, by building *sendTo* in this way, we can implement both cases in a single method instead of two, by forwarding the arguments to functions below and let the compiler decide which methods to call. Naturally, the lowest-level send methods in the library need to have implementations for both cases. In this case, a "Communication" class possesses these basic send and receive methods, which provide some error handling for socket writes/reads when not every byte could be written/read at once (unless the error was severe, the class can retry writing/reading the remaining bytes).

The choice of internal structures is also important for performance. For example, every IGCL node possesses a structure that represents the nodes currently known to it. This structure, a peer table named "knownPeers" is frequently accessed to check the existence of peers, convert their IDs into the respective socket or stream descriptors (or the reverse), get the type of their connection or a list of all peer IDs, among other functions. If these checks and conversions were slow, it would affect almost every part of the library. Therefore, it uses C++ *maps* for logarithmic time searches. This also happens in other structures, such as the ones that map descriptors to their respective

receive queues (see Section 3.2.2).

Some relevant and mostly small methods in the library are also inlined, thus hinting the compiler to directly inject the inline method's instructions in the place of calls to it. This happens, for instance, in most calls to the NBuffering class and methods of the frequently accessed peer table we mentioned before (of which an example can be seen in Listing 3.7). It avoids having too many functions calls for a single IGCL operation in some places, which can be expensive. However, ultimately, the compiler decides if it wants to inline functions or not, and, when optimizations are turned on, it might do so even in functions that are not hinted as inlined.

Some other things we did in IGCL, such as passing complex structures by reference or returning constant references, also bring performance advantages and sometimes hint the compiler to place the result of an operation directly at the target, without copying objects around. In the case of the *downstreamPeers* and *upstreamPeers* methods, for example, the returned value is a constant reference to the internal vector that contains such peers, making it easier for the compiler to understand that it is not necessary to copy the vector if the programmer declares the recipient as also a constant vector (thus never changing its contents). A similar thing can be seen again in Listing 3.7, where the result of the call to *idToDescriptor* can be optimized by the compiler and never produce copies of the respective object. Once again, these are not guaranteed to happen — apart from passing values by reference —, as compilers are mostly free to choose how they translate instructions.

Something that hinders performance and that we could not solve is related to libnice. When IGCL wants to send data through one stream created by libnice, it may not be able to send all data at once, which is not a rare occurrence, and can also happen with normal sockets especially if the block of data we are trying to send is relatively large. However, when libnice fails to send all data at once in its reliable TCP-over-UDP mode, in forces the CPU to wait for a library callback to retry writing the remaining bytes in the stream. This wait is sufficient to introduce severe performance loss when sending a lot of information (see Section 4.8), but should not occur otherwise. For us, the option of using simple UDP connections in libnice (thus avoiding this callback at the cost of reliability) is not viable, as a failure in delivering one packet to the target or receiving packets in the wrong order is enough to completely break an algorithm.

3.2.4 Registration

IGCL provides an automated method of registration for peers in the group, which requires no further action of the programmer than specifying the coordinator listening IP and port. We refer to Figure 3.2 for the following explanation.

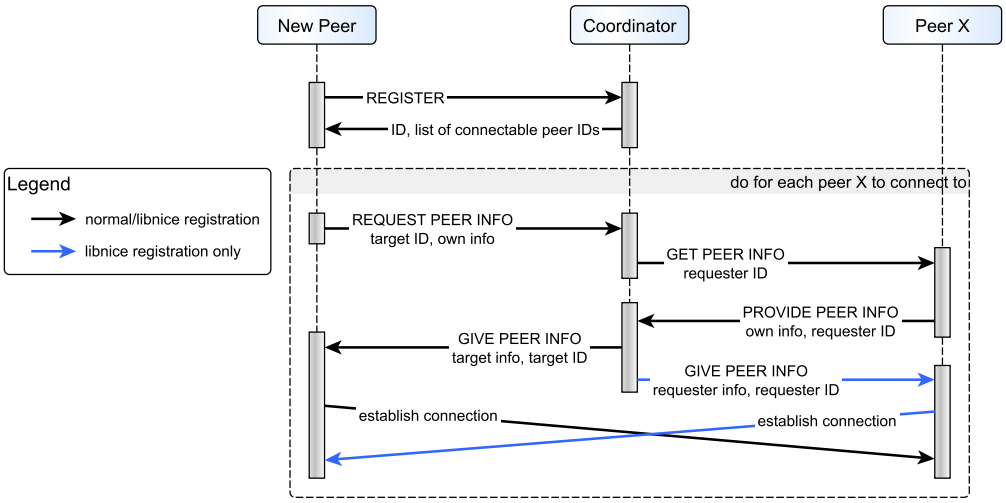


Figure 3.2: Sequence diagram of the registration process in IGCL.

Like every other node, the coordinator of the group has an address to listen for new connections. When a new peer arrives and connects to this address, it then sends a register message to the coordinator to start the registration process. The coordinator, which always has an ID equal to 0, will reply with the ID it attributed to this peer. These nodes' IDs are given starting from 1 and in increments of one, so that the last peer that registers in a fixed layout will receive an ID equal to the size of the layout minus one. Along with the ID, the peer will also receive the list of IDs of other nodes that it should directly connect to (as defined by the layout), as well as which peers are after and before in the layout, in case of fixed layouts. For free-formed layouts, the peer will automatically consider all connect-to peers to be "next" (it only happens in the free-formed all-to-all layout).

The next step is for the peer to individually ask the coordinator for information about each node it should connect to, based on their IDs. These are now the only piece of data the peer has about other nodes. As part of the request, the peer can also provide its own information — information which we will interchangeably call "peer credentials" — for the target peer,

if needed for the connection.

It was a requisite for us that the registration process was asynchronous from the viewpoint of the coordinator and target peer, despite several messages and responses being involved. This was because these nodes could already be working on a job (when in free-formed layouts) or have to respond to other peers, and thus could not block waiting for another peer to register. Therefore, every node only responds to registration/connection messages when requested, although the requesting peer will, by design, have to establish connection to a peer before requesting the credentials of further peers. For this asynchronism, the coordinator and requesting peer both maintain state about ongoing requests for connections.

Upon receiving a request for peer information, the coordinator acts as an broker and contacts the target peer. This peer then provides the requested credentials to the coordinator, at which point these can then be sent back to the requesting peer. In libnice connections, both connecting peers need each other's information to successfully connect; thus, in this case, the requesting peer is obliged to provide its own information along with its original request, which the coordinator temporarily saves. Later, when the target credentials are acquired and given to the requester, there is an additional step in which the coordinator also gives the (previously saved) credentials of the requesting peer to the target. Both peers should now have the necessary information to directly connect to each other.

One important thing to note in this process is that the coordinator does not retain peer credentials for giving them to other peers. This is intentional. For ICE-free connections, credentials are simply the address and listening port of the peer and, indeed, should always be the same; however, the credentials in ICE connections include a list of address candidates, which vary each time they are requested. For consistency, we opted to maintain the same scheme for the two connection types.

Figure 3.2 does not show that nodes can connect in three different ways, which we previously mentioned: 1. using C sockets; 2. using the ICE mechanism provided by libnice; or 3. indirectly, by relaying messages through the coordinator. This is also the priority order for testing connectivity. The Figure shows, however, the global process undertook by the requester for cases 1 and 2.

When first requesting peer info through the process in Figure 3.2, the node requests enough information to try a connection using normal sockets. After the coordinator receives that information from the target peer, it sees

if the public IPs of the requester and target node are the same. If they are, it will alert the requesting node when it gives it the target credentials. The requester can now try to connect locally, using the listening port provided by the target, which should result in an optimal, local, connection. If this fails, it will then try to connect on its public IP. If both of these fail, peers need the help of ICE to establish a connection, so the requester repeats the process by now asking for the respective "libnice credentials" as per Figure 3.2. Finally, if libnice cannot establish the connection due to very protective routers, the node can simply set the connection to the node as relayed through the coordinator. For performance reasons in some algorithms, IGCL provides the function *setAllowRelayedMessages(bool active)* in the Peer class, which can disable relaying. If relaying is inactive and a connection to some peer is impossible, peers simply send a de-register message to the coordinator.

3.2.5 NBuffering implementation

For several applications, a problem arises if the heterogeneity of the participating machines implies, for example, that one will finish much later than others. When this happens, dividing the data in equal sections, one for each peer, is not adequate, as a lot of time is wasted waiting for the results. It is for this reason that we implemented a class for N-buffering, whose usage we showed in Section 3.1.1. This class — NBuffering — is intended for applications following the master-workers model, and allows data to be buffered in smaller quantities to the worker nodes.

The summary of what the class does internally is to control what and how many jobs are sent to each node. Jobs are sent via calls to a user-defined send function until they fill the queue for each peer (which has a size equal to the buffering level). When a result arrives, in the programmer code, he/she only has to call a method on the NBuffering object to mark the job as completed and request buffering again for that peer, as we saw on Section 3.1.1. The NBuffering class is completely oblivious to what the user is sending as jobs, because these are internally represented only by their index in the total number of jobs to process — which in turn is given in the object constructor. Queues of jobs are associated with their respective peers via a *map*, much like the blocking queues in Section 3.2.2.

As we mentioned before, buffering is a good complement to the free-formed master-workers layout, as nodes can enter and leave at will and still do jobs. If a connection to a peer fails, the class supports removing a peer

from its internal structures. This will effectively re-assign the jobs that were pending for that peer (jobs that were sent but are not completed yet) to a special queue that is prioritized when buffering data to other peers in the future. Appendix A.5 should be consulted for more details.

3.2.6 Error handling

IGCL provides basic forms of error handling to the programmer, depending on the layout and algorithm. Some methods return a *result_type* value, which contains one of three possible indicators: *SUCCESS*, *NOTHING* or *FAILURE*. These values denote whether, respectively, the operation was successful, there was nothing to do (ex.: in a non-blocking read operation), or the operation failed. This enables the programmer to handle the result. When errors happen, the library can also automatically clean up after disconnecting peers or terminate execution, depending on the group layout used.

As we saw in Section 3.1.3, there are fixed-size layouts that are especially suited to controlled environments, and which would incur in error if a node suddenly disconnected. When a connection fails while using this type of layout, the group coordinator automatically ends execution in every node by sending them a termination message and then proceeding to exit itself. By contrast, in free-formed layouts, the coordinator or any connected node that notices the disconnection of the peer will automatically handle its de-registration, cleaning its reception queues, node information and other internal structures. Methods that return lists of known peers, connected peers or their sizes are immediately affected by the de-registration.

As connection errors can be found in user threads calling IGCL send methods, the internal thread that receives messages, seen in Section 3.2.2, could be blocked in the socket *select* method at that time and not be informed of the error. To solve this problem, this thread uses a timeout in *select*, which lets it check from time to time for an existing termination state and cleanly quit. The libnice API also has its own methods to exit cleanly, which IGCL uses.

Chapter 4

Results and Discussion

4.1 Experimental setup

For our tests, we utilized several environments, all of them virtualized. We will refer to these environments by their numbers throughout the next sections:

1. The first is a cluster composed of 8 virtual machines existing on a physical computer possessing an 8-core Intel® Xeon® E5-2650 CPU with 2 GHz.

2. The second environment is also a cluster, made of 6 virtual machines, each with access to a Intel® Core™i7-2600 CPU with 3.4 GHz. This CPU has 4 hyperthreaded cores, essentially giving us the ability to run eight processing threads at once.

3. The third environment is a single machine with an Intel® Core™i7-3632QM CPU with 2.2 GHz. This CPU also has 4 hyperthreaded cores. The machine was connected by cable to the network's router and using an Internet connection capable of a theoretical download rate of around 12 megabytes per second (MBps) and upload rate of 5 MBps.

4. Lastly, to allow us to execute tests with communication through the Internet, we made use of a public IP server, also virtualized. This server runs on the same physical machine as environment 1, and thus has the same features. For tests, we will never refer to this environment alone, but in conjunction with others, as its main purpose was to allow Internet scale communication as a coordinator.

Due to the necessity of testing IGCL when using computers behind NATs, the ideal environment was very specific, possibly with several collaborating users running a parallel application on their computers from various remote locations. Unfortunately, we were not able to find such an environment to execute tests in. It did not help that environments 1, 2 and 4 were located in the same network area, separated by a few milliseconds of router hops, and neither did it help that this network was institutional and contained a type of NAT that could not be traversed by libnice, invalidating direct connections from the outside, from common home networks. Therefore, we note that Internet-scale IGCL tests in Section 4.7 are less than ideal, although meant to represent general applicability of the library when deployed in the Internet.

We will start with a generalization for which applications are suitable for Internet deployment in terms of communication requirements. Afterwards, our tests consist of several comparisons of local IGCL execution times versus the equivalent threaded or MPI applications, the effects of N-buffering on speedup, the performance of several algorithms when using Internet communication and the differences in performance between using normal sockets, libnice streams and relayed connections.

In every test here presented, algorithms were compiled using the O3 optimization flag in the compiler, which is either GCC's g++ or, in the case of MPI applications, Open MPI's mpic++. Results are always an average of 30 executions and only the main processing algorithm is timed, to ignore setups or cleanups of data needed for each execution. Values are given in seconds. Furthermore, in environments 1 and 2, composed of multiple virtual machines, tests were executed with no more than a single IGCL process per virtual machine. In this Chapter we will mostly present evolution plots; the mean values that generated those plots are included in Appendix C, along with their standard deviation values.

4.2 Implemented examples

To test the library and demonstrate some of its features, we implemented a set of example algorithms that follow several communication patterns and requirements. For each algorithm, we present a figure with the evolution of processing time and communication requirements as the number of participating nodes grows.

4.2.1 Matrix multiplication

Matrix multiplication is an example of $\mathcal{O}(n^3)$ time complexity that can be solved through the master-workers pattern. Assuming we want to multiply two matrices A and B, and every node already has the whole matrix A, we can trivially decompose B in a way that each computing node gets a section of B to multiply with matrix A. Doing so is simple because the multiplication of each matrix section is completely independent of others. After doing their calculations, each node will have computed its own section of the final product matrix and can therefore send it to the master (which can also compute its own matrix section, if needed). This results in a total time complexity of $\mathcal{O}(\frac{n^3}{k})$ for this parallel version, where k is the number of participating nodes, assuming that the distributed sections are equally sized for each node.

To achieve this distributed algorithm, each worker must have either direct access to matrix A or receive it in full from the master. Assuming the latter case, which we applied in our example, the growth in number of nodes can reach a point where further dividing matrix B becomes counterproductive, due to the significant loss incurred from broadcasting matrix A to all nodes. Figure 4.1 shows the linear growth of communicated data, along with the expected decrease in processing time.

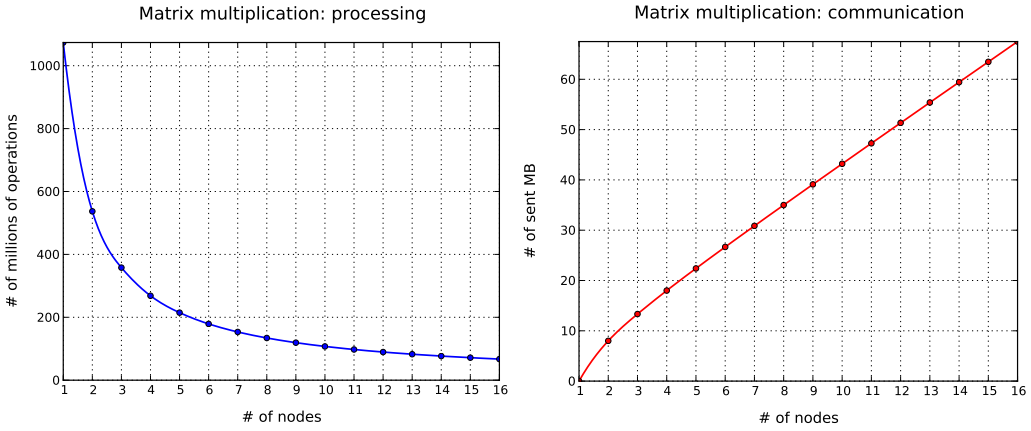


Figure 4.1: Matrix multiplication: growth of processing time and bytes exchanged with the number of nodes. 1024 x 1024 matrices.

4.2.2 Merge sort

Another pattern that also frequently shows up is the divide-and-conquer model. One example of this pattern appears in the implementation of merge sort [62], which is a recursive process with a descending and ascending part. The algorithm starts with an array of values to sort. The descending part consists in dividing this array into two equal halves in a recursive way (i.e. each half is further divided in two, like the original array; these halves also are divided in two, and so on). The division continues until the smaller arrays contain one or no elements, thus being ordered by definition. This descending part of the algorithm will have created a recursive "tree" of calls, and now the ascending part goes up this tree. Ordered halves are successively joined into a single sorted array, passed up the tree and joined again with their other equal sized half. This continues until we get the original array, now fully sorted, at the top of the call tree. The whole algorithm has a time complexity of $\mathcal{O}(n\ log\ n)$, as we process all n elements at each of the $logn$ tree levels.

In a possible parallelized version of this algorithm, each node acquires a section of the array, retaining half of it and sending the other half to another node for sorting. Each node can then split its section in half again, for nodes further down in the division tree. Afterwards, each node sorts its half using a normal merge sort and returns it to its parent, which will join both halves into a sorted array and return it up the tree, and so on. This version achieves a time complexity of $\mathcal{O}(\frac{n}{k}\ log\ \frac{n}{k} + n\ log\ k)$ when the number of nodes is a power of the branching factor — the base of log —, which is usually 2. Like in the matrix multiplication example, k is the number of nodes participating in the algorithm. The $\frac{n}{k}\ log\ \frac{n}{k}$ part comes from running a non-distributed merge sort in each of the k nodes, all of which retain $\frac{n}{k}$ elements each. All of these sorts are done independently. The $n\ log\ k$ part is the merging of array halves at each tree level in the tree of k nodes, implying that the total n elements are processed $log\ k$ times. This time complexity is not exact when the number of nodes is not a power of the branching factor, as the unbalancing of the tree causes some nodes to possess more elements than others.

We would like to note that this implementation of a parallel merge-sort is far from being the most efficient. Using this structure, a node receives a certain number of elements and then immediately passes half of them along to a child node, half of that to another child, and so on until no more children are available. It would be better if the master node did the full distribution itself,

directly, as the nodes do not process the elements before re-sending them. This merge-sort example stands as a simple model of the tree structure, albeit not the most efficient one. Figure 4.2 shows that this method has some irregularities in its evolution with the number of nodes, due to the fact that numbers of nodes that are not a power of 2 create unbalanced trees.

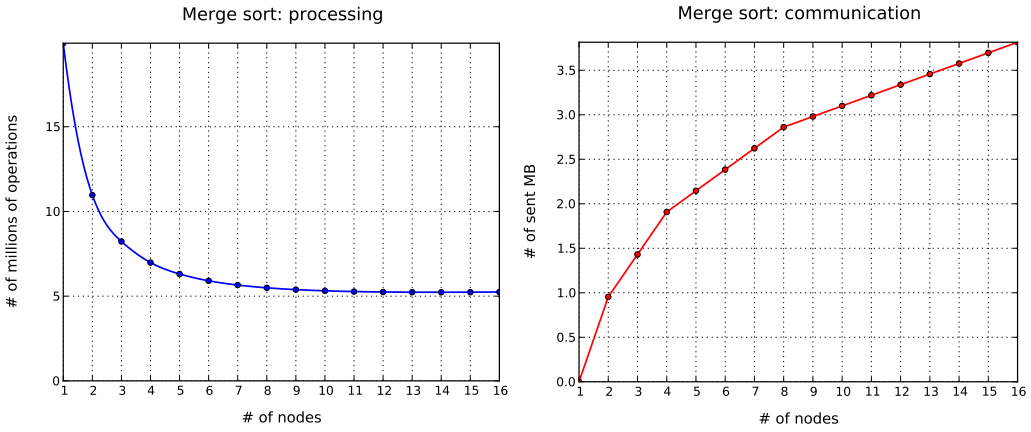


Figure 4.2: Merge sort: growth of processing time and bytes exchanged with the number of nodes. 1.000.000 elements.

4.2.3 Ray tracing

Another implemented example is a ray tracing application. Ray tracing is a computer graphics' image rendering technique that is based on the emission of invisible rays from the pixels of a virtual camera in space, following the reverse path of light across the scene. Each of these rays will recursively collide with objects in the scene, reflect, refract and perform calculations based on the objects' materials and scene's light sources, ultimately returning the color of its respective pixel. The technique is simple in concept and design, and allows almost native implementation of realistic complex effects like reflections, shadows and global illumination models, generally associated with high visual fidelity. It is nevertheless usually deemed inadequate for real-time rendering, due to its high computational requirements [63].

For this work, a ray tracing application represents, similarly to matrix multiplication, a massively-parallel example, due to the fact that every casted ray is different and independent of every other. In a parallel version of ray tracing, nodes receive the bounding indexes of sections of an image (which can fit in a very low number of bytes) and then compute the color of each

pixel in that section, thus generating part of the total image[1]. To contrast with the matrix multiplication example, this means that much more data is sent back from the workers to the master than the other way around.

Images can have very unbalanced processing power requirements for its different sections. In the rendering of a room with a lot of reflective marbles on a table, for instance, the part of the image with the marbles implies many reflections for the ray to follow to the light sources, thus taking more time to render than a hypothetical empty wall right behind the table. This essentially means that nodes should not get the same amount of pixels to process and should instead be progressively buffered new sections/jobs as they complete previous ones. We talked about buffering in Sections 3.1.1 and 3.2.5. Despite these inequalities and the heavy dependency on the complexity of the scene itself, it can be said that a ray tracing algorithm has a general time complexity of $\mathcal{O}(n)$ if n is the number of pixels to render and the scene is fixed. With buffering to balance the processing requirements, the respective parallel time complexity will be approximately $\mathcal{O}(\frac{n}{k})$. The node evolution for this application is visible in Figure 4.3. We note that, as shown, communication time should remain constant with a growth in number of nodes, as the number of jobs per image is only dependent on the image resolution and number of pixels per job.

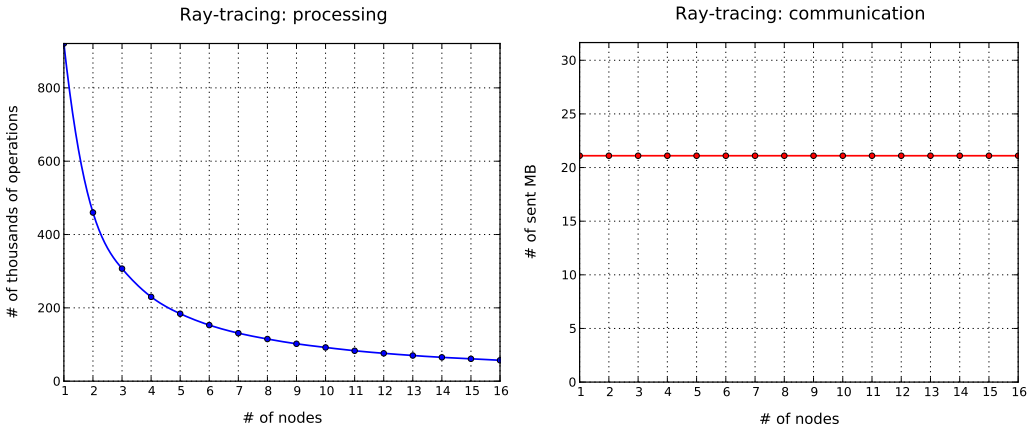


Figure 4.3: Ray tracing: growth of processing time and bytes exchanged with the number of nodes. 1280 x 720 image, 1000 pixels per job.

[1]We assume that every node has easy access to the description of the scene to render.

4.2.4 Traveling Salesman Problem

Finally, we implemented an algorithm for the Traveling Salesman Problem — henceforth "TSP" —, which has a time complexity of $\mathcal{O}(n!)$ in its brute-force approach. In the TSP, the objective is to find the shortest route that visits all nodes in a graph exactly once. Different variations of the problem exist, with varying restrictions. In our case, the graphs are complete, which means that each point has a connection to every other point. Furthermore, the route through the nodes is not a cycle; i.e. the route does not return to the starting node after visiting all others. Our implementation is a branch-and-bound algorithm that tests all possible routes and simply discards paths that have no chances of beating the current best path.

Parallelization of this approach to TSP can be achieved by defining a set of different starting points for each computing node and letting each try all path possibilities from each starting point in the set[2]. Unfortunately, and although the searches for paths are indeed independent, such a parallel approach discards most of the performance advantage of using the current best as a bound, as peers do not know if better paths have already been found by other peers and, consequently, will expectedly try more paths than they have to. To counter this tendency, we improved this parallel method by occasionally exchanging bounds between nodes to speed up their search. A secondary thread checks, from time to time, if a new (better) bound was found, and sends the value to other nodes. This kind of parallel algorithm requires little exchanging of data and is an adequate demonstration of problems where Internet-scale communication between nodes should be plausible, performance-wise.

We do not present a figure about the parallel TSP growth. Reasons for this are varied. Firstly, the number of exchanges of bounds is not constant and tends to be smaller as time progresses and findings of better bounds become harder. Furthermore, the increase in number of nodes yields a theoretical increase in communication in the factorial order (because every node sends its bounds to every other), but sending bounds implies very little bandwidth and it also decreases the time the total algorithm takes, which becomes especially hard to model in a generic way. Lastly, processing time is affected by the division of the problem instance's points and how often nodes find new bounds. For these reasons, it suffices to say that the total quantity of sent data, from our observations, is in the order of hundreds of bytes. As for

[2]This means that more than one node can search the same path, in reverse ways, but it is a detail that does not affect our tests.

the evolution of processing time, it is expected to decrease in a way that is similar to the previous examples. We will return to this matter in Section 4.7.

4.3 Communication analysis

We came up with a formula for multiprocessor environments that allows us to have an estimation of the achievable speedup for applications with communication requirements. We base our formulas in the research by Li and Malek [53], seen in Section 2.4.3. As communication depends on a number of factors, namely data size, bandwidth and distance between nodes, for a minimally accurate model we need to account for these, instead of simply providing T_{comm} to the equation.

In our definition, T_{comm}, which we henceforth label TC, is divided in two parts: 1. the duration of the initial plus final transfers of data between coordinator and workers, TC_{seq}, and 2. the total duration of intermediate communications between nodes, TC_{node}. As with Equation 2.4, we assume that times are uniform across all nodes, because it is hard to predict how much they will deviate from the average time. These parts are calculated as follows:

$$TC_{seq} = \frac{data\ size}{connection\ speed} + network\ overhead \tag{4.1}$$

$$TC_{node} = \left(\frac{data\ size}{connection\ speed} + network\ overhead\right) \times no.\ of\ comms. \tag{4.2}$$

In these equations, *data size* represents the quantity of data transferred through the network (in the case of Equation 4.1, all initial and final data; in Equation 4.2, the average transferred per node communication); *connection speed* characterizes the speed achieved by the network (cabled or wireless) in data-per-time units; *network overhead* deals with the time that information takes to travel the medium, independently of data being sent; lastly, *no. of comms* indicates the number of times each node sends *data size* units through the network, in an average run.

Based on what we have seen in Section 2.4.3 regarding speedup under

fully parallel or fully sequential communication and, particularly, Equation 2.6, we suggest a final formula for the likely speedup range of an application:

$$\frac{1}{s + \frac{(1-s)}{N} + \frac{TC_{seq}+TC_{node}}{T_{seq}}} \leq S \leq \frac{1}{s + \frac{(1-s)}{N} + \frac{TC_{seq}+TC_{node}}{N \times T_{seq}}} \tag{4.3}$$

Consider an application that takes 10 minutes to run sequentially, with a non-parallelizable part of only 2% of the total sequential time. In these conditions, Amdahl's Law predicts a speedup as seen in Figure 4.4. Note that speedup growth progressively slows down, and that, for 500 nodes, the parallel version merely surpasses 45 times the speed of the original application. This is a common consequence of using Amdahl's Law, as we saw before.

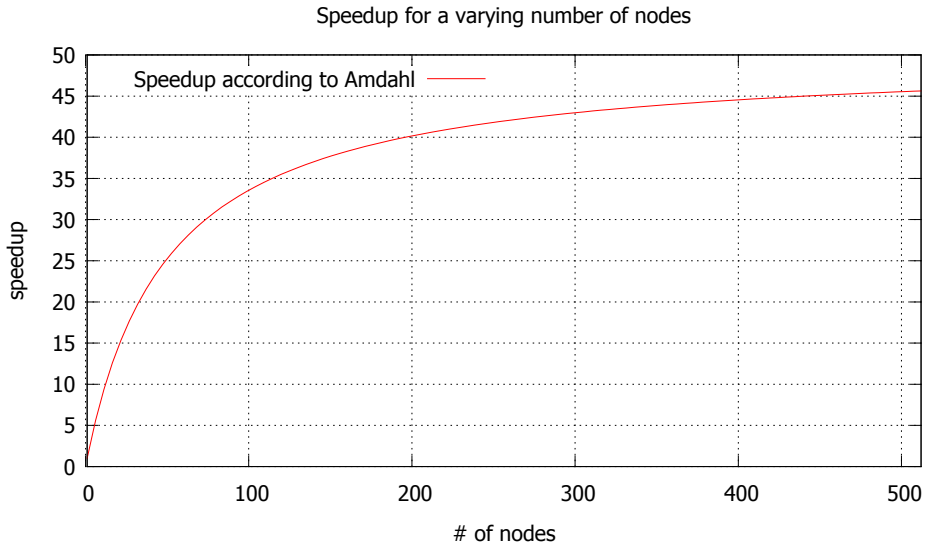


Figure 4.4: Speedup according to Amdahl's Law ($s = 0.02$, $T_{seq} = 10$ min)

Maintaining these conditions, consider that the application's initial data transmission from coordinator to workers consists of 100 MB in total, and that the total data transferred by each node mid-algorithm, including the final results, is about 20 MB each in a 100 Mbps network. It is important to note that, in this case, we assume the network overhead to be quite low due to participating nodes being located close to each other. Using Equation 4.3, Figure 4.5 presents four plots: the case where all communication can be made in parallel (therefore adding the same overhead for any N), the case where all communication is done sequentially, the average of these two, and, finally, the original speedup according to Amdahl. With fully sequential communication,

instead of speedup simply growing poorly, we actually observe its decline when even less than 20 nodes are used, due to processing time being largely surpassed by communication overhead from that point on.

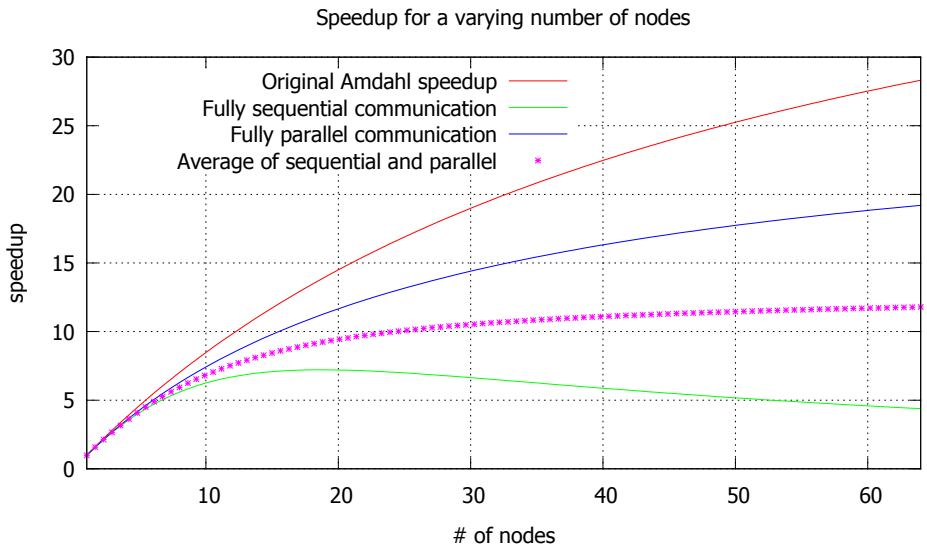


Figure 4.5: Speedup with communication ($s = 0.02$, $T_{seq} = 10$ min, 100 MB + 20 MB per node)

This is a problem that is non-existent in both Amdahl's and Gustafson's formulas, but that reveals itself in real world problems and has consequences in the design of distributed systems, network speed and even connection materials. Essentially, depending on the problem at hand, it might be infeasible to consider running applications that transfer a lot of data in a distributed environment like the one considered.

Because of these results, we recommend that volunteer computing systems only work with applications that truly benefit from such environments. As we mentioned in Section 2.5, applications that are expected to perform well in a distributed network are those that have low communication requirements and that can benefit from replication of data. It also helps that such an application is resistant to the unreliability of nodes, as we might be executing it in a highly volatile environment; nevertheless, this is not a requirement.

For comparison with the previous example, the evolutionary Island Model mentioned in Section 2.5 is, similarly to our parallel TSP example[3], estimated

[3]We modeled the evolutionary algorithm instead of the TSP due to its more predictable communication requirements.

to possess a relatively low amount of communication between nodes (only the Pareto fronts are exchanged), and, besides the application itself and eventual configurations, as little as a pseudo-random seed might be enough to start the evolutionary algorithm.

Keeping the same values of network speed, overhead, T_{seq} and s as in the example before, if an assumption is made that the initial transfer is around 10 MB and each node transfers about 5 KB of data 200 times[4] — which totals 1 MB of data — to propagate the best individuals of its island, we get speedup values much closer to the original Amdahl speedup, as can be seen in Figure 4.6.

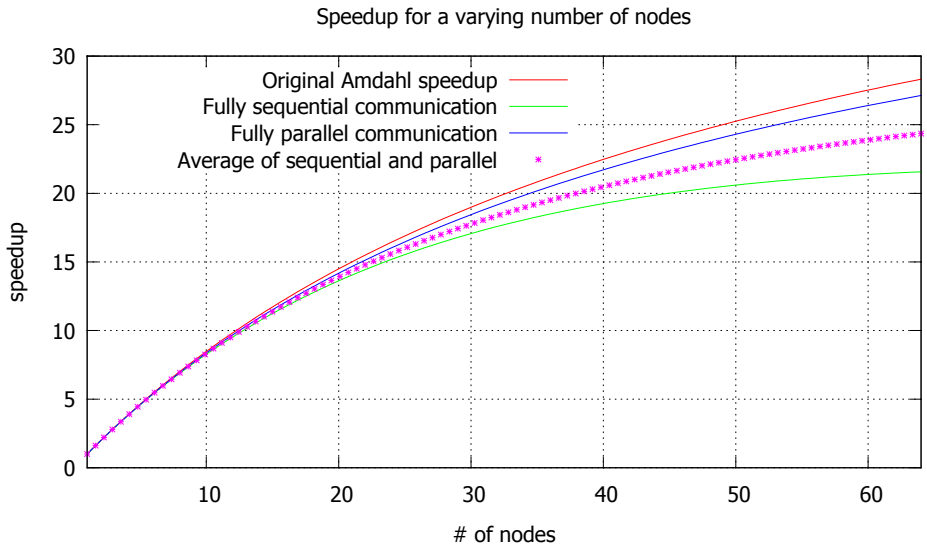


Figure 4.6: Speedup with communication in Island Model application ($s = 0.02$, $T_{seq} = 10$ min, 10 MB + 1 MB per node)

Again, we do consider in both cases that the network overhead is small. If nodes are many kilometers away from each other, as could happen with BOINC users' machines, the overhead might make it hard to run applications with many exchanges of information. Nevertheless, we should note that most — if not all — communication in the described evolutionary application can be done asynchronously, as in the TSP, so network delays are not expected to hinder the process significantly.

[4]These times are a theorization of what is expected of the Island Model example, and do not originate from a real application.

4.4 Comparison of IGCL and MPI

We will first demonstrate the advantages of IGCL versus MPI in terms of code simplicity when using the master-workers and divide-and-conquer communication patterns. We will resort to two examples for this, namely the parallel merge sort and ray tracing applications that we described in Section 4.2. As the two libraries' approaches are naturally distinct, we tried to use code that is equivalent or as similar as possible in functionality. The Listings referenced in the following paragraphs can be consulted in this document's Appendix B.

Listing B.1 demonstrates the main part of the implementation of a parallel merge sort that uses IGCL. We omit variable declarations, the *joinSort* function and all initializing or terminating code. For comparison, we present the equivalent MPI implementation in Listing B.2. As can be seen, specifying the send and receive nodes and sizes in MPI looks cumbersome and takes many more lines of code than the IGCL example. Besides making the code simpler to write, the high level patterns should as well indirectly contribute to make programs more reliable, as fewer and simpler lines of code should reduce programmer mistakes.

Likewise, we present in Listing B.3 the implementation of the matrix multiplication algorithm in IGCL, which we compare with the respective MPI code in Listing B.4. It is noticeable how calculations and IDs are mostly handled by IGCL and not the programmer.

To evaluate the local performance of our library, we now compare its execution times with Open MPI's, using the previous implementations. Our tests consisted in executing the parallel merge sort example on an array of 3×10^7 elements and the parallel matrix multiplication algorithm on two 2048-row square matrices, both on environment 1 (refer to Section 4.1). Executions with Open MPI and IGCL utilized the exact same node layouts.

In Figure 4.7 we show the average execution times of the matrix multiplication algorithm in IGCL and Open MPI using 1 to 8 nodes. It can be seen that IGCL is very close to Open MPI in execution times. Figure 4.8 presents a similar comparison, this time for the merge sort algorithm. Both libraries perform similarly, with a slight edge for IGCL when the number of nodes increases.

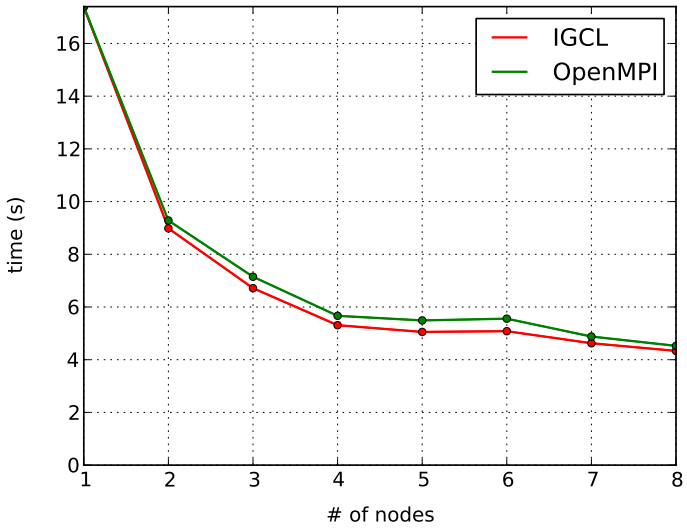


Figure 4.7: Matrix multiplication: IGCL and Open MPI performance. 2048×2048 matrices. Environment 1.

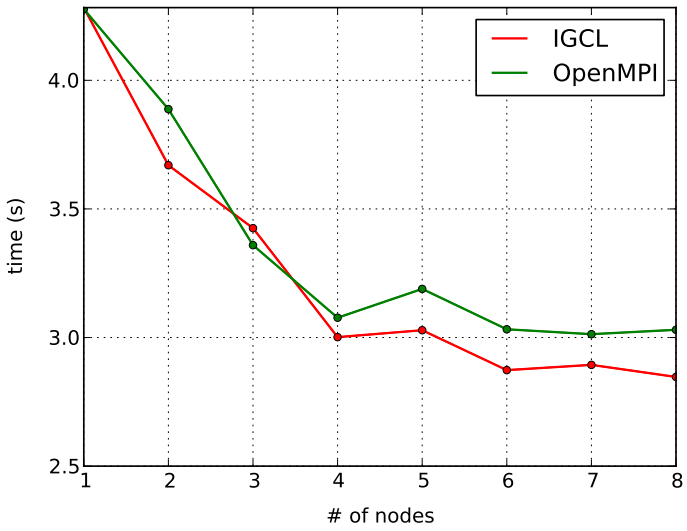


Figure 4.8: Merge sort: IGCL and Open MPI performance. 3×10^7 elements. Environment 1.

4.5 N-buffering effects on speedup

We wanted to measure the effects of buffering in our ray tracing applications, thus we generated an image of 9600 × 5400 pixels in parallel, using environment 3. The test ran in 8 nodes (the group coordinator also generated sections), used a job size of 10000 pixels and a varying level of buffering. Results are seen in Figure 4.9, which shows how processing time changes with the increase in level of buffering.

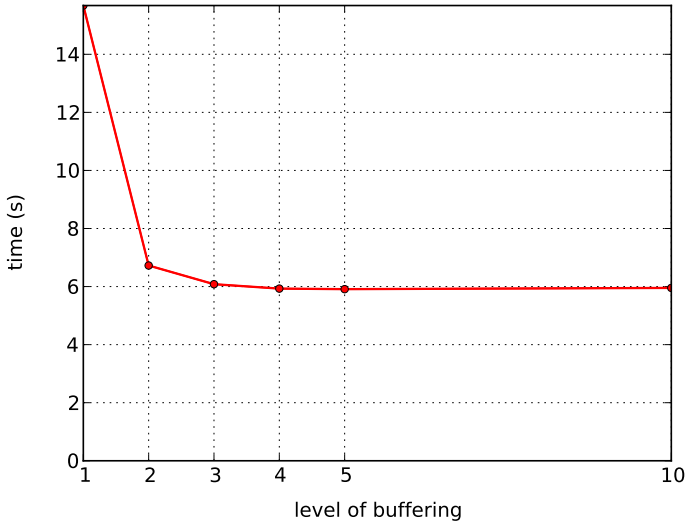


Figure 4.9: Ray tracing: effect of various levels of buffering. 9600 × 5400 image. 10000 pixels per job. Environment 3.

We can conclude from this Figure that a buffering level of 1 — which essentially means no buffering at all — leads to a very poorly performing ray tracing application. This is essentially caused by nodes processing their job of 10000 pixels rather quickly, sending the generated section to the coordinator and then having to wait for more jobs to come, resulting in a lot of time wasted in wait. We should note that all nodes are processes in the same machine, thus communication time between coordinator and peers is very reduced. If the master/coordinator was separated from the worker nodes by many kilometers and router hops, the waiting time could be significantly worse.

When using a buffering level of 2, a considerable performance improvement is seen. In this case, nodes will almost always have a job buffered to work on right after sending the generated pixels of the previous one. As a node processes the second job, the coordinator has the opportunity to send

another one to fill the now empty buffer, reducing the total waiting time. Increasing the buffering level to 3 and 4 still yields an improvement, covering cases where nodes are processing their sections faster than they are buffered to them (which can happen if the job pixels were in a part of the image that was easy to generate). As seen in Figure 4.9, this environment does not seem to benefit much from buffering levels of above 4.

The ray tracer was also run in environment 2, with a smaller image of 1280×720 pixels, jobs of 1000 pixels and using a separate coordinator that only buffered jobs and did not help generate the image. Figure 4.10 shows a behavior consistent with what we previously showed, where an increase in buffering level from one to two jobs brings the most visible improvement. We also see that increasing the number of nodes while using high levels of buffering tends to produce few improvements, as the image is smaller than in the previous example and already generated in little more than 2 seconds.

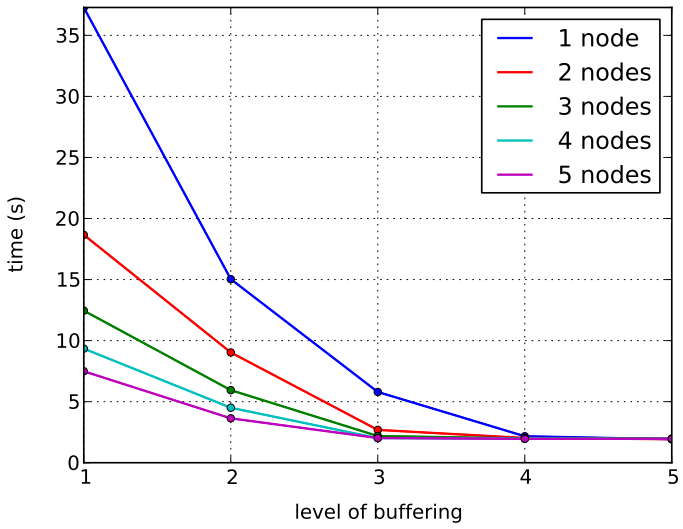


Figure 4.10: Ray tracing: effect of various levels of buffering. 1280×720 image. 1000 pixels per job. Environment 2. Quantities of nodes do not include the coordinator.

4.6 Comparison of IGCL and threading

Another comparison we wanted to do was between the IGCL ray tracing application and the respective multi-threaded version, to ascertain the overhead introduced by locally using IGCL in an algorithm. With this objective,

we ran the ray tracer to generate a 9600×5400 image using both versions, in test environment 3. Our IGCL example ran with a buffering level of 10. The threaded version uses OpenMP[5] to parallelize the main processing cycle with the following directive:

```
#pragma omp parallel for schedule(dynamic, 10000) num_threads(N)
```

We did not use the default *parallel for* directive to parallelize the code, as it uses a *static* schedule and therefore implies dividing the image in equal sized sections for each thread [64]. This was not accurate to compare to our buffering scheme because, as we have seen, images can be unbalanced, potentially resulting in bad performance. Therefore, we set the schedule to *dynamic*, so that threads can progressively get new chunks, and then set the chunk size to 10000 indexes, to mimic the jobs of that size that we used in IGCL. Figure 4.11 shows results of this comparison.

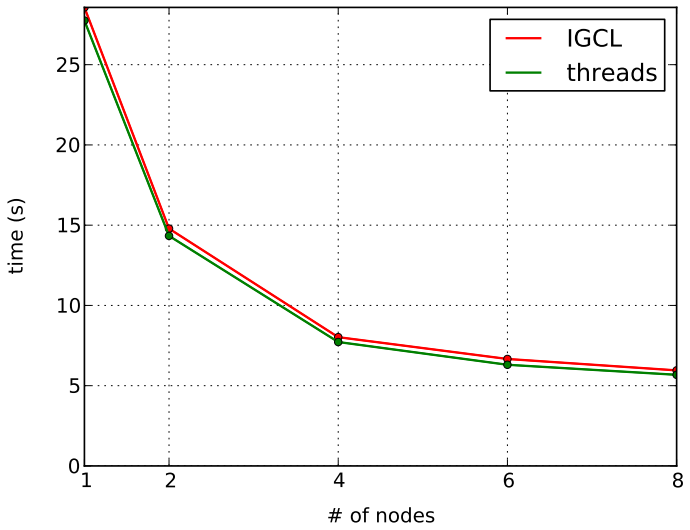


Figure 4.11: Ray tracing: performance of IGCL versus threads. 9600×5400 image. 10000 pixels per job. Environment 3.

It is clear that OpenMP threads have a slight but consistent edge when using any number of nodes, which is expected, due to the fact that there is no communication involved between processes. IGCL overhead is placed not only on the communication of data between the coordinator and worker nodes, but also on the extra memory allocations and buffering mechanism itself, the latter of which can reduce performance even for a single node.

[5]Do not confuse OpenMP with Open MPI. The former is an API for shared memory programming; the latter is a library that implements the MPI standard for parallel systems.

Execution times from IGCL are nevertheless close, increasing in the range of 2.95–5.73% for the tested quantities of nodes, as seen from Table 4.1.

# of nodes	1	2	4	6	8
OpenMP	27.748	14.334	7.721	6.303	5.678
IGCL	28.567	14.786	8.023	6.664	5.953
diff.	+2.951%	+3.156%	+3.912%	+5.728%	+4.851%

Table 4.1: Ray tracing: execution times (in seconds) using IGCL and threads, and respective difference. 9600 × 5400 image. 10000 pixels per job. Environment 3.

4.7 Internet-scale IGCL

For Internet-scale tests, we started with the TSP example with 16 graph nodes, running with the coordinator located in the public IP machine from environment 4 and all the remaining nodes on environment 3. The latter environment is, in fact, the only one that was truly separated from the server and on a completely different network, as environments 1 and 2 were in the same area as 4.

In the first test, we intended to see if it was possible to achieve speedup using TSP, as it is the example that has less data being exchanged. We executed it both with and without bound exchanges during execution, as Figure 4.12 shows. Essentially, in one case, nodes send their current best bound occasionally to every node; in the other case, they only send their final result to the coordinator, after finishing their search. It can be seen that there is a significant improvement in the first case and a substantially more reduced one in the second. Exchanging bounds seems, therefore, to be the better solution, leading to improved times and validating our parallel approach to the branch-and-bound TSP.

One detail that surprised us was the fact that using 8 nodes with no exchange of bounds resulted in a worse total running time than when using 4 or 6 nodes. We believe that adding more nodes affected the CPU negatively in environment 3, as using many nodes on the same virtual machine in this example yielded an almost full CPU occupation. Furthermore, as TSP bounds are not shared (though the search space is still divided), adding nodes does not benefit us much in terms of execution time. Consequently, a loss in performance is possible.

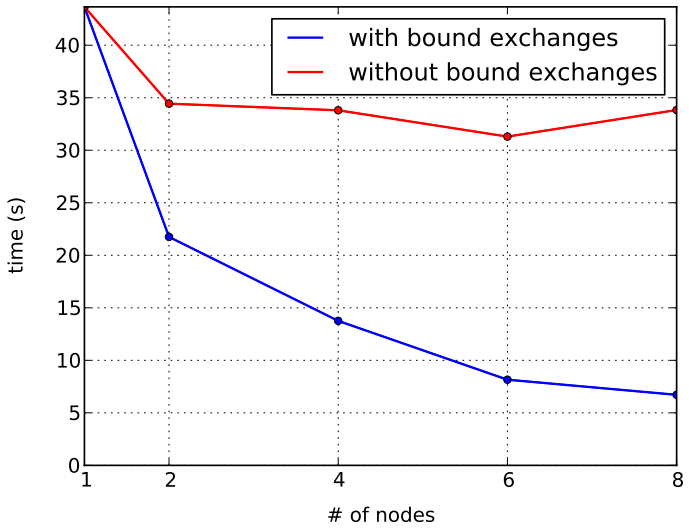


Figure 4.12: TSP: networked performance when exchanging bounds or not. 16 locations. Environment 3 with 4.

The fact that we can indeed achieve speedup in an Internet-scale environment, especially in the bound-exchange version of TSP, is important to our work, and demonstrates our conjecture that when little communication is involved, applications can indeed improve when run in remote nodes. We again note that all non-coordinator nodes were located in the same machine, thus very quickly trading bounds between them. This is expected to improve times when compared with an environment where they are all located in very distant networks. Nevertheless, we have an upper bound on speedup here, given by the example in which no exchange of bounds is done, and it still yields a visible improvement.

We also tested the matrix multiplication and merge sort applications with the same setup, but these did not perform well. Figure 4.13 shows our tests with the multiplication of two 1024×1024 matrices, again with the remote server (environment 4) acting as the coordinator and environment 3 running the remaining nodes. All nodes, including the coordinator, processed an approximately equal share of the matrix. The figure easily shows that introducing more remote nodes in the system actually degraded performance. The more nodes were used, the more data the networks of environments 3 and 4 had to support. In this example, matrix B has to be sent to all participating nodes except the coordinator. As matrices have $1024 \times 1024 \times 4$ bytes, the total data sent with 4 nodes, just for matrix B, is $(4-1) \times 1024 \times 1024 \times 4$ bytes, which totals 12 megabytes. This already requires a rather

large upload rate from a single point — the coordinator/server —, assuming that environment 3 is also capable of downloading at this rate (as we have seen from the experimental setup in Section 4.1, this value already fills the maximum theoretical speed of the network, 12 MB). By further noticing that the algorithm still needs to distribute matrix A among the worker nodes and that there is some communication latency involved, we easily justify the negative performance observable in Figure 4.13. Using 8 nodes, the algorithm was already taking more than 10 seconds to complete on average; more than 5 times the sequential time in the coordinator alone.

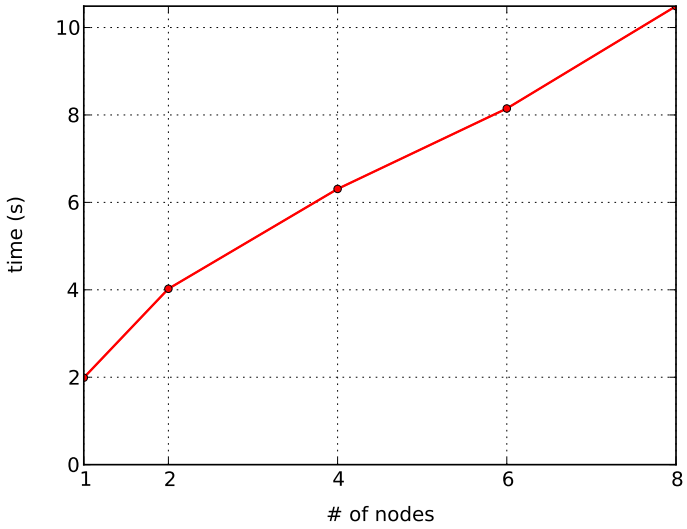


Figure 4.13: Matrix multiplication: networked execution times. 1024×1024 matrices. Environment 3 with 4.

Figure 4.14 shows a similar result for the merge sort application when sorting an array of 500000 elements. We expected this behavior from the merge sort application, as the growth in data transfers when increasing the number of nodes is high, as seen in Figure 4.2. In addition, our setup goes against the optimal placement of tree nodes, because the coordinator has to send several pieces of the array to environment 3, as all its downstream nodes are located there. Nevertheless, we could not exactly predict the degree to which execution time increased when we added even a single node to the process, compared to using the coordinator only. In the plot of Figure 4.14, we can see a very noticeable increase from a time of some milliseconds (the exact value is 59 ms) to more than one second, as the network overhead dominated processing time. Some other tests were done with larger array sizes, which take longer to process, but revealed the same type of growth, as transferred data sizes are also naturally increased.

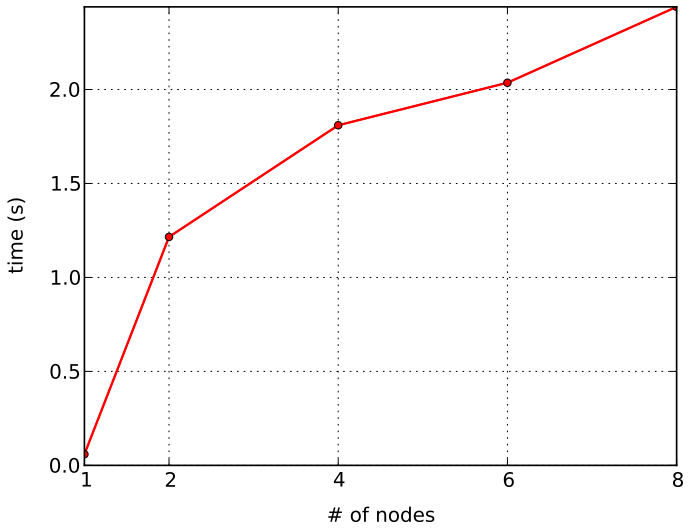


Figure 4.14: Merge sort: networked execution times. 500000 array elements. Environment 3 with 4.

We also tested the networked ray tracing application when generating an image of 2880×1620 pixels, again with the coordinator also processing jobs. We used two equivalent versions of the algorithm; one where the exchanged pixel colors were given in values of the *char* type (0–255 for each RGB color intensity), and another where they were *doubles* (0.0–1.0, also in RGB)[6]. The sizes of these values in the tested architectures were, respectively, 1 and 8 bytes, meaning that the second case transfers 8 times more data than the first when returning image sections.

We can measure the number of total jobs in this example if we divide the image size by the number of pixels per job (jobs are sections of 10000 pixels), which yields the value 467. In this application, the stress placed over the networks is theoretically better distributed in time than in the matrix multiplication or merge sort examples, as data exchanged is buffered and also approximately constant with the number of nodes (as we showed in Figure 4.3). Nevertheless, our observations for the version using *doubles*, seen in Figure 4.15, showed that the coordinator was processing between 90% and 95% of these jobs, depending on the level of buffering, meaning that the results were so slow to transfer between nodes that the coordinator

[6]This is actually the result of our ray tracer doing calculations with doubles that are only converted to chars when saving the generated images to disk. We exploited this to build two versions, where one converts the doubles to chars before sending and the other does not.

was producing most of the image on its own. The unstable number of jobs it processes throughout these examples is reflected on the standard deviation values of the execution times, seen in Table C.9 (and Table 4.16, for the second ray tracer version), which are relatively high when compared with the total processing time of the algorithm. We believe these deviations in work balance are caused by the unpredictability of network usage.

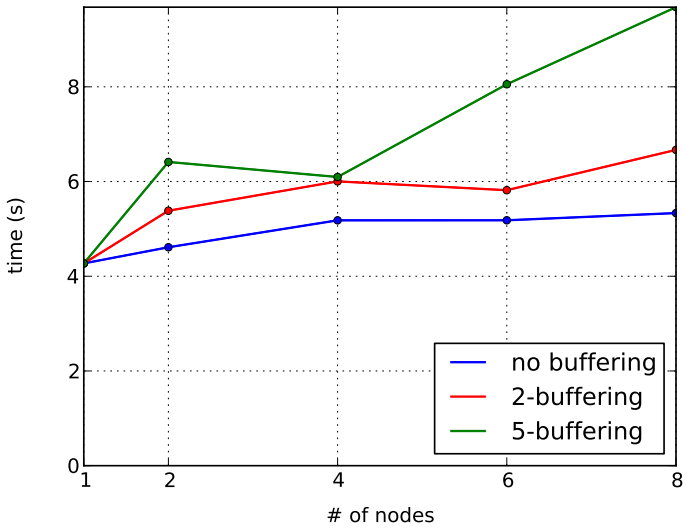


Figure 4.15: Ray tracing: networked execution times. 2880 × 1620 image (using doubles). 10000 pixels per job. Environment 3 with 4.

To explain why the process takes more time when using more than the coordinator itself, first consider that there is a certain moment in the algorithm when all jobs were sent and only the results of some are pending. After this moment, we are dependent on whichever nodes have the remaining jobs to complete the image. Now, we can see that the coordinator typically completes its jobs much faster than other nodes, because it does not have to send data to itself to complete the job. This leaves the process waiting for the results from other nodes, which traverse the network.

We note that even if the coordinator is able to process *all* jobs that were not buffered to remote nodes, and do it before the first result of arrives from these, a high-level buffering to all other nodes will nevertheless decrease the number of jobs it processes. This happens because we initially buffer a number of jobs equal to the buffering level to every peer. For example, when using a buffering level of 5, 5 jobs are sent at once to each node, and none of these will be processed by the coordinator, hence diminishing its number of jobs.

In this networked example, sending 5 initial jobs to a node can already be more than the total number of jobs the node would receive in the whole algorithm with, for example, a buffering level of 2, as the coordinator can complete its jobs faster than these nodes can send data through the network. This leads to a further increase in the final waiting time, exposed by the evolution in Figure 4.15, where we can see that increasing the buffering level (or number of nodes) tends to worsen the algorithm performance.

For comparison, we now show in Figure 4.16 the second ray tracer version, which transfers approximately 8 times less data through the network. We can observe that a small speedup is actually achieved in this case, with every level of buffering tested.

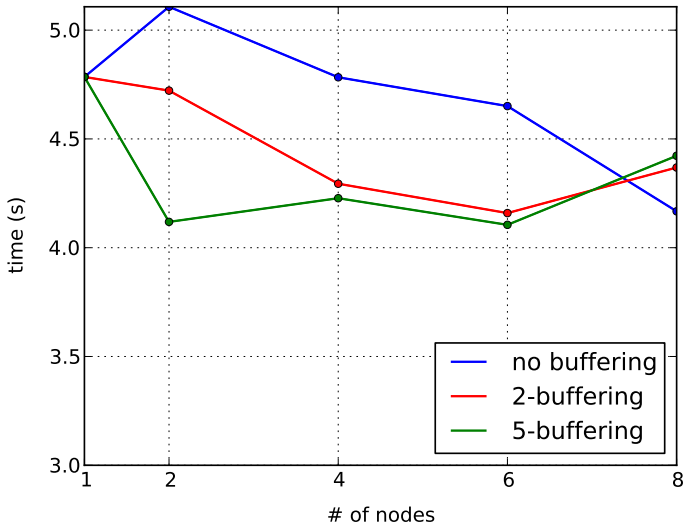


Figure 4.16: Ray tracing: networked execution times. 2880 × 1620 image (using chars). 10000 pixels per job. Environment 3 with 4.

There are some things to note here, namely the fact that using no buffering leads to a general improvement when increasing the number of nodes (which corroborates results from Figure 4.9), but not immediately when adding just one node. This is probably due to the fact that adding a single node which takes a long time to receive, process and return a result, hinders the coordinator more than if it did not exist. In fact, the average number of jobs completed by the coordinator in this case, as seen in Table 4.2, was 432.3 — approximately 92.5% of them — and this is similar to the percentages in the previous version, which should help explain this evolution.

Another thing to note is that, similarly to how increasing the number of

# of nodes	1	2	4	6	8
no buffering	467	432.3	393.5	383.4	351.0
2-buffering	467	368.6	338.3	332.8	332.8
5-buffering	467	328.3	334.1	327.0	320.9

Table 4.2: Ray tracing: average number of jobs executed by the coordinator only. 2880×1620 image (using chars). 10000 pixels per job. Environment 3 with 4.

nodes while using no buffering leads to better performance, higher levels of buffering with no change in the number of nodes seem to also increase performance *to a certain point*. This means that the coordinator benefits from sending jobs to other nodes, but this benefit is diminished when sending too many jobs to the outside. As comparisons between Table 4.2 and Figure 4.16 can show, a correlation between number of jobs processed by the coordinator and performance does not seem to exist. From our tests, it does seem that the networks of the coordinator and of environment 3, which includes all non-coordinator nodes, were being overloaded when increasing the number of jobs. This can certainly explain why the application suffers from using too many nodes in our tests.

4.8 Connection type comparison

As we said in Section 3.2, libnice streams incur in performance degradation when used to transfer large chunks of data. This is due to the waiting time for the TCP-over-UDP stream to become "writable" again. In this Section, we compare libnice streams with normal sockets, first in a local scenario and then in a networked one with another application.

Figure 4.17 shows a local example of merge sort where nodes were forced to link using libnice connections. As can be seen, there is a significant performance loss when moving from normal sockets to libnice.[7] This example is particularly bad for libnice, as our parallel merge sort includes large and redundant data transfers in both descending and ascending parts of the algorithm.

Our readings showed that libnice streams are not capable of sending more

[7]When using one or two nodes, we cannot use libnice, as peers connect to the coordinator using normal sockets. Only when more peers are introduced in the tree and connections are established between them can we test all connections types.

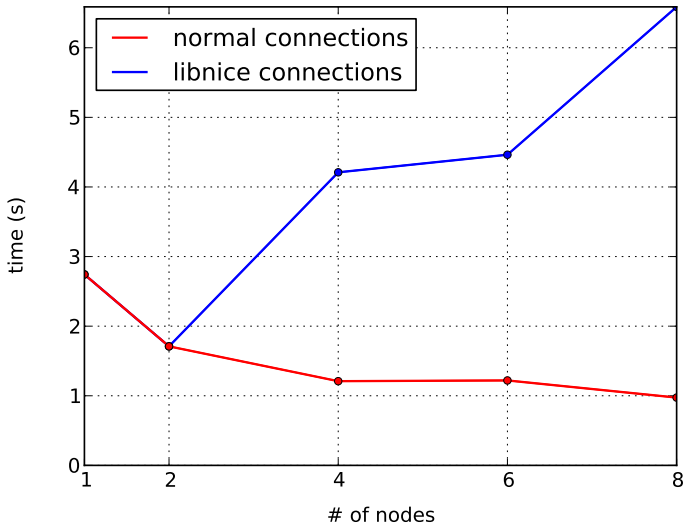


Figure 4.17: Merge sort: local analysis of normal versus libnice connections. 3×10^7 elements. Environment 3.

than 100.000 bytes at once in our test machines. Once we try to do so, IGCL generally has to wait for the stream to be writable and try again. Depending on the time elapsed between attempts, the next one can either result in a few thousand bytes being written or several tens of thousands. Notice that this is not a very large quantity of data, and the merge sort example illustrated in Figure 4.17 actually tries to send millions of bytes at once, which, in our tests, did not cause problems when C socket connections where used.

It is relevant to note that this kind of performance hit only happens in applications that send large quantities of data and cause libnice to trigger several callbacks per sending. These are already not suitable for Internet communication, as our tests have showed. As the ICE connectivity provided by libnice only makes sense in the Internet, it means that we would be using libnice in applications that already have low communication needs, therefore not causing problems to begin with. For our TSP example, running in the same conditions as the ones used to generate Figure 4.12 with networked exchanges of bounds, Figure 4.18 shows that sockets and libnice streams seem to yield performances so similar that their plots are overlapping. Note that running this test with every node on a different remote computer (thus incrementing time of data exchanges between peers) should result in equivalent observations, as a libnice stream is naturally indistinguishable from simple sockets after data is sent from the machine itself. This test is, therefore, essentially for performance at the application level.

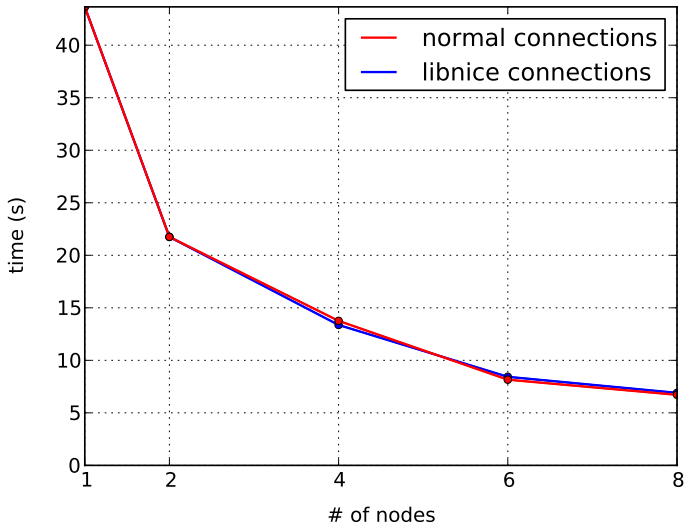


Figure 4.18: TSP: networked analysis of normal versus libnice connections. 16 locations. Environment 3 with 4. Plots are overlapping.

In fact, the TSP example is so light in communication that we can see from Table C.12 (in the Results Appendix) how all connection types, including relayed through the coordinator, result in very similar performance. Differences here are most likely attributed to fluctuations in CPU occupation and network usage.

Chapter 5

Conclusions

Over the course of this work, we have confirmed our expectations that distributed execution of applications over the Internet is hard, but possible under some restrictions. Due to the distance between nodes and the speed of connections, performance suffers significantly, sometimes even decreasing with the addition of more processing nodes, instead of improving. Our tests demonstrated that, from our four example algorithms, only two appear to be suitable for large scale Internet execution: the bound-exchanging parallel TSP and, to a lesser extent, the ray tracing application. The TSP is precisely the example that represents the class of algorithms with low communication needs; class that we believe to benefit from execution scattered throughout remote nodes. On the other hand, the ray tracer is an application with respectable communication requirements that can run effectively over the Internet if nodes have great network capabilities, the master node in particular.

For the other examples, matrix multiplication and merge sort, we recommend that Internet environments are discarded in favor of clusters of computers (either independent or connected via a common front end), or even single machines with independent jobs (for example, a full pair of matrices to multiply), if parallelization is not viable or at all desired. We must note that our test environments do not give us full confidence about the validity of remote communication for these examples; however, judging from our results, we believe that they are very likely to perform badly when using nodes in common home networks that are not sufficiently fast. Perhaps with the evolution of technology and network speeds around the world we can achieve better results years from now.

We also saw that direct connection at the Internet scale presents some other problems, such as the existence of NATs and firewalls. We showed that, when direct connections using sockets is impossible, it is viable to counter this problem by using NAT traversal mechanisms like ICE. For cases where these are insufficient, relayed connections through a third node are also possible for certain applications, as the parallel TSP proves. Our implemented library, IGCL, dynamically makes use of these three possibilities, making it useful for either local or Internet communication.

In our tests, we further showed that IGCL performs well when compared with a similar Open MPI implementation of merge sort and matrix multiplication running in a cluster of machines. It also revealed minimal overhead in the ray tracer example versus a version using multi-threading. In fact, some care was taken in developing IGCL for high performance computing, while still providing some error-control when connections fail.

Functionality similar to algorithmic skeletons is also natively implemented in IGCL, and we presented how the abstraction could make programmer's code shorter and possibly less error-prone. As part of IGCL's features, N-buffering was also demonstrated for master-workers applications where work division can be unpredictable, such as our ray tracing example. Using buffering, masters can keep their worker nodes fed, while also automatically scaling work among them by only providing more work when results of previous jobs are collected. This seems especially important in volunteer computing, where nodes have heterogeneous features and some nodes might complete work faster than others.

Indeed, for volunteer computing, we conclude that there are advantages in having nodes with direct connection capabilities, as it makes possible for several algorithms to run efficiently without excessively tasking the servers. In addition, even if direct communication is not desired, IGCL's peer groups make it possible to use a known public IP coordinator as a secondary server for volunteer computing projects; one that controls and relays data between other nodes.

5.1 Future Work

The direct communication of unknown nodes naturally implies some problems for volunteer computing. Security and reliability immediately come to mind, due to, respectively, the possibility of malicious nodes introducing in-

correct or viral data in the system, and nodes randomly disconnecting, as is common in volatile environments such as BOINC. This work did not delve in the first problem, but we believe it is a candidate for future work research. The problem of reliability could also be further analyzed and developed, as IGCL only presents basic error control mechanisms. The research we presented in Section 2.3.1, for instance, gives some hints on how reliability can be achieved in distributed systems, by checkpointing and replication.

We have mentioned that possible targets for IGCL deployment are volunteer computing systems like BOINC. It would be interesting if future related work completed the bridge between the two. Several changes must be executed in the BOINC server application to support the concept of peer groups. Firstly, the BOINC server application must be able to assign certain nodes as group coordinators. These are typically the ones with better network connections and more resources, especially if the coordinator will also work directly on the algorithm. In our work, only coordinators with a public IP can manage a group, as a direct socket connection to it is needed. In BOINC, volunteer nodes are not expected to have public IPs and, hence, we need the project's servers to help establish connections between nodes and the coordinator. This can be done with a similar method to the one IGCL uses for connections, by letting coordinator and peer exchange credentials through the BOINC servers and connect using libnice or other library that provides an implementation of ICE. The proximity of nodes could also be analyzed by these servers, prioritizing groups where nodes are closely located. It might even possible to detect that some nodes are in the same network, as IGCL itself does, and create a small group of peers that efficiently process some work locally, taking advantage of their local connections.

Directly related to IGCL, it could be useful to account for node heterogeneity and location in group layouts, letting the coordinator assign layout positions/IDs according to these (in contrast to simply assigning the next available ID). In an example similar to our parallel merge sort, for instance, we expect that better performance would be achieved if the division of work at a certain tree level — which results in two branches that never communicate with one another (see Figure 2.5b) — was done between machines whose downstream peers are spatially close to them. This should yield better results than randomly distributing machines throughout the tree, as every data transfer would take more on average on the latter case than if nodes were organized by location in the tree.

Towards other kind of future work, there are some things we would like to improve about IGCL. For example, the library could have been made

thread safe with some more work, making it easier for the programmer to have a thread receiving data while another handles sending and the algorithm itself. Right now, such a setup requires the programmer to rely on mutual exclusion constructs to avoid race conditions. The complexity and possible performance loss of handling race conditions inside IGCL made us reject that possibility in this work. With what we now know, we would have started development with threading in mind, instead of attaching mutexes and conditional variables later in development. This might have resulted in a thread safe library or, at least, a better understanding of the associated performance issues.

Another thing we would like to change was described in Section 3.1, where we mention that error alerts are returned as SUCCESS, NOTHING or FAILURE, in a *result_type* object. Though we believe this approach to be simple, it created the problem of having to frequently check for errors in calls to functions, both internally and in programmer code. This handling of errors is also mostly C-like, a language where exceptions do not exist. Because we used C++, try-catch blocks and exception throwing were a potentially more adequate mechanism for catching errors in higher level functions, with the added potential of carrying more information about the error.

We have mentioned that our layouts and patterns of communication are similar to what algorithmic skeletons provide (see Section 2.5.3). Our approach does not try to mimic any of their features, however. It is possible that some existing skeletons could be adapted to work with our idea of layouts, releasing us from the responsibility of implementing them at a lower level. They could even better prepare IGCL for new patterns or make the use of layouts more generic and useful for the programmer. At the time of implementation, however, algorithmic skeletons were unknown to us, and this feature of IGCL was considered simple enough to code from zero. In the future, we would like to try existing skeletons in IGCL.

We would also like to add better support for the GroupLayout class inside IGCL. For example, the NBuffering class should ideally not require a call to *addPeers* to add worker nodes, as the layout and the coordinator itself already know that information and could use it. The same can be said for the method *waitForNodes*, which the coordinator could use automatically if the layout is fixed. Another improvement would be related to the communication patterns. As of now, it is easy for the programmer to try and use a pattern with an incorrect layout. If IGCL had a better support for layout functionality, these methods could exist inside the layouts themselves and each layout could implement the pattern methods that are relevant to

it. These are all limitations of the library that we consider important for further developments.

5.2 Reflections and other work

As we approach the end of this document, we would like to use one last section to reflect about a few more things that we learned from these past months or that might have been done differently with the knowledge we now possess. This section also serves the purpose of sharing some details about work that ended up not being included in the remaining document.

C++ does not make it easy on the programmer to debug his/her application. As it is inherently low-level and some of its most recent constructs were added on top of already existing features, it is also a very error-prone language. The use of templates also sometimes produces cryptic compilation error messages, which programmers find frequently when using the predefined classes, where templating abounds. Furthermore, this is a work about process communication — over the Internet, even —, fact that introduces a new layer of complexity that is hardly negligible when writing and debugging code. The use of threads and frequent allocation and deallocation of memory further increase the already high complexity of developing the library. This is not to say that we should have used another programming language, as the performance of C++ and the possibility of easier extensions to volunteer computing were very important; nevertheless, with all of these summed, we do want to mention that the effort required to build and test a library like IGCL "from scratch" over one semester was underestimated and led to many hours of frustration, debugging and restructuring of code. As features accumulated, additions also became more complex to implement, especially when modifying the internal state of objects and/or implicating race conditions between threads.

We also found that most libraries we tried for NAT traversal, even others that are not mentioned in this document, were either underdeveloped, overly complex, lacked documentation, support forums or had different goals in mind than ours. Libjingle is a good example, as it was the first library that we tried with ICE support and immediately showed to be hard to compile and understand, having examples with many hundreds of lines of code that made the learning curve prohibitively steep. It was also mostly directed at VoIP and video streaming and not at a more "bare-bones" approach that could be used in distributed computing. Libnice itself presented us with

problems in understanding its examples and documentation, but ended up being the one that better worked for us. In the end, many weeks were spent trying to get libraries to compile and work — sometimes in several machines, as NAT traversal implies — and then understanding how to use them for our purposes, often with little or no success. We also talked about the issue with libnice and writing on streams, which could probably be solved by using another, perhaps more low-level library. Unfortunately, our search did not find a more adequate C/C++ one.

During library development, parts were rebuilt because some design choices were not working well. An example of this is realizing that IGCL once made extensive use of callback functions when receiving messages, and required the programmer to write his/her code inside them. In addition, these user callbacks were all running inside the main thread of the library, meaning that, while each callback executed, every new arrival of data from a peer would be ignored until completion of the callback. This was a rather bad design choice that should have been deemed so earlier in the process, and it was becoming hard to circumvent. In the end, we made large changes to the internal code and made use of the threaded mechanism described in section 3.2.2, where received data is handled by a thread and either processed immediately or put into a queue that is later accessed by the application code. The success of this approach in code organization and functionality was immediate for us, though it had the disadvantage of forcing many more allocations of memory than were previously necessary. However, some brief tests revealed that the respective performance hit was practically unnoticeable when compared to the callback version, with the added bonus that the execution of user callbacks did not block other activities anymore.

In a more technical perspective, we understood that a valuable tool would have been to write or use a logger tool that enabled us to use hierarchically configured debug messages in our code (i.e. messages that have levels of relevance and can be logged or not, as needed). Sometimes we found that we had too many debug messages being printed and often ended up deleting or commenting some of them to reduce clutter. Other times, messages were too few, or we wanted to specialize in debugging only a specific part of code. A specialized tool or C++ class would have helped in this matter, though we did not further research this topic to present good examples.

If we had the time, we could have also bridged our work with volunteer computing, including IGCL in the BOINC client, as we have mentioned in Section 5.1. Several things prevented us from reaching this point, namely the already mentioned underestimation of development time and problems with

the NAT traversal libraries; but this was also due to the fact that we decided to implement several more example algorithms than initially planned and extend the peer groups to support layouts, which ended up being a significant IGCL feature. We had originally planned to implement a single example application: an evolutionary algorithm based on Island Models, as described in Section 2.5.1. In fact, we did implement the algorithm, applied to the Traveling Salesman Problem, and added the basic exchanging of individuals to resemble the Island Model. However, the stochastic nature of the evolutionary process was, in our opinion, going to lead us to significant difficulties in obtaining results about IGCL and Internet communication. Unless we had access to many nodes and enough time to tweak and test the exchange of individuals to not harm population diversity, the algorithm could not be useful to us; consequently, we decided to implement the easier-to-test TSP algorithm we described in Section 4.2.

It is also inconvenient that we could not present tests that better represent Internet communication and NAT traversal, as we would, had we been able gather a more adequate setup of remote computers in different home networks. Unfortunately, tests with communication from home network to home network would have implied the simultaneous collaboration of multiple people for a long period of time and good network connections for the harder applications. The symmetric NAT configuration of our clustered environments did not help, either. Furthermore, even a simulated test environment was difficult to use, due to configuration requirements and a lack of real communication latency. The lack of more adequate and stable environments to test IGCL and Internet-scale execution in some of our experiments is, admittedly, a shortcoming of this work.

Bibliography

[1] D. P. Anderson, "BOINC: A System for Public-Resource Computing and Storage," *Fifth IEEEACM International Workshop on Grid Computing*, pp. 4–10, 1999.

[2] D. P. Anderson and J. McLeod, "Local Scheduling for Volunteer Computing," *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1–8, 2007.

[3] D. Anderson, E. Korpela, and R. Walton, "High-Performance Task Distribution for Volunteer Computing," *First International Conference on e-Science and Grid Computing (e-Science'05)*, pp. 196–203, 2005.

[4] W. N. Martin, J. Lienig, and J. P. Cohoon, "Island (migration) models : evolutionary algorithms based on punctuated equilibria," *Evolutionary Computation*, vol. 2, pp. 1–16, 1997.

[5] I. Scriven, A. Lewis, and D. Ireland, "Decentralised distributed multiple objective particle swarm optimisation using peer to peer networks," *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, pp. 2925–2928, June 2008.

[6] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation," *Proceedings 11th European PVMMPI Users Group Meeting*, vol. 3241, no. Springer-Verlag Berlin Heidelberg, EuroPVM/MPI 2004, LNCS 3241, 2004, pp. 97–104, 2004.

[7] Message Passing Interface Forum, "MPI : A Message-Passing Interface Standard," 2012.

[8] P. Domingues, P. Marques, and L. Silva, "Resource Usage of Windows Computer Laboratories,"

[9] D. Anderson and K. Reed, "Celebrating diversity in volunteer computing," *System Sciences, 2009. HICSS'09. ...*, 2009.

[10] J. Dean and S. Ghemawat, "MapReduce : Simplified Data Processing on Large Clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 1–13, 2008.

[11] D. Anderson, C. Christensen, and B. Allen, "Designing a Runtime System for Volunteer Computing," *ACM/IEEE SC 2006 Conference (SC'06)*, pp. 33–33, Nov. 2006.

[12] F. Costa, L. Silva, I. Kelley, and I. Taylor, "Peer-to-peer techniques for data distribution in desktop grid computing platforms," *Making Grids Work*, pp. 1–12, 2008.

[13] F. Costa, L. Silva, G. Fedak, and I. Kelley, "Optimizing the data distribution layer of BOINC with BitTorrent," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–8, IEEE, Apr. 2008.

[14] C. Chapman, P. Wilson, T. Tannenbaum, M. Farrellee, M. Livny, J. Brodholt, and W. Emmerich, "Condor Services for the Global Grid," *National Environment ...*, 2004.

[15] G. Fedak, C. Germain, V. Neri, and F. Cappello, "XtremWeb: a generic global computing system," in *Proceedings First IEEE/ACM International Symposium on Cluster Computing and the Grid*, pp. 582–587, IEEE Comput. Soc, 2001.

[16] S. Yi, E. Jeannot, D. Kondo, and D. P. Anderson, "Towards Real-Time, Volunteer Distributed Computing," *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 154–163, May 2011.

[17] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor: a distributed job scheduler," *Beowulf cluster computing ...*, 2001.

[18] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the Grid," *Grid computing: Making the ...*, 2003.

[19] O. Lodygensky, G. Fedak, F. Cappello, V. Neri, M. Livny, and D. Thain, "XtremWeb & Condor : sharing resources between Internet connected Condor pool," in *CCGrid 2003. 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid, 2003. Proceedings.*, pp. 382–389, IEEE, 2003.

[20] F. Cappello, S. Djilali, G. Fedak, T. Herault, F. Magniette, V. Néri, and O. Lodygensky, "Computing on large-scale distributed systems: XtremWeb architecture, programming models, security, tests and convergence with grid," *... Computer Systems*, 2005.

[21] "Introduction to XtremWeb. Retrieved January 2013, from http://www.xtremweb.net/introduction.html,"

[22] Z. Balaton and G. Gombás, "Sztaki desktop grid: a modular and scalable way of building large computing grids," *..., 2007. IPDPS 2007. ...*, 2007.

[23] E. Urbah, P. Kacsuk, Z. Farkas, G. Fedak, G. Kecskemeti, O. Lodygensky, A. Marosi, Z. Balaton, G. Caillat, G. Gombas, A. Kornafeld, J. Kovacs, H. He, and R. Lovas, "EDGeS: Bridging EGEE to BOINC and XtremWeb," *Journal of Grid Computing*, vol. 7, pp. 335–354, Sept. 2009.

[24] E. Urbah, "EDGeS / EDGI: Bridging Institutional Grids, Desktop Grids and Academic Clouds Applications."

[25] S. Delamare, G. Fedak, D. Kondo, and O. Lodygensky, "SpeQuloS: a QoS service for BoT applications using best effort distributed computing infrastructures," *... Distributed Computing*, no. February, 2012.

[26] X. Wan, "Analysis and design for VoIP teleconferencing system based on P2P-SIP technique," 2011.

[27] E. Setton, J. Noh, and B. Girod, "Low latency video streaming over peer-to-peer networks," *Multimedia and Expo, 2006 IEEE ...*, pp. 569–572, 2006.

[28] M. M. Driss, B. Fatima, and I. Abdessamad, "A multi-agent system for collaborative editing in mobile networks and P2P," 2009.

[29] B. Cohen, "Incentives Build Robustness in BitTorrent," 2003.

[30] "BitTorrent Protocol Specification. Retrieved January 2013, from http://www.bittorrent.org/beps/bep_0003.html,"

[31] L. D'Acunto, T. Vinkó, and H. Sips, "Bandwidth Allocation in BitTorrent-like VoD Systems under Flashcrowds," no. June, 2011.

[32] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, pp. 17–32, Feb. 2003.

[33] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth, "BAR fault tolerance for cooperative services," *ACM SIGOPS Operating Systems Review*, vol. 39, p. 45, Oct. 2005.

[34] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin, "BAR Gossip," *Systems Research*, pp. 191–204, 2006.

[35] H. Li, A. Clement, M. Marchetti, M. Kapritsos, L. Robison, L. Alvisi, and M. Dahlin, "Flightpath: Obedience vs. choice in cooperative services," *Proceedings of the 7th ...*, 2008.

[36] M. Cieślak, "BOINC on JXTA - suggestions for improvements," pp. 1–42, 2007.

[37] V. Paulsamy and S. Chatterjee, "Network convergence and the NAT/-Firewall problems," 2003.

[38] S. Guha and P. Francis, "Characterization and Measurement of TCP Traversal through NATs and Firewalls,"

[39] J. Rosenberg, R. Mahy, C. Huitema, and J. Weinberger, "STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs),"

[40] D. Wing, P. Matthews, J. Rosenberg, and R. Mahy, "Session Traversal Utilities for (NAT) (STUN),"

[41] P. Matthews, R. Mahy, and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN),"

[42] J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Methodology for Network Address Translator (NAT) Traversal for Offer/Answer Protocols,"

[43] B. Traversat, M. Abdelaziz, D. Doolin, M. Duigou, J.-c. Hugly, E. Pouyoul, S. Microsystems, and S. A. Road, "Project JXTA-C: enabling a Web of things," *.... Proceedings of the ...*, vol. 00, no. C, pp. 1–9, 2003.

[44] S. Sur, M. Koop, and D. Panda, "High-performance and scalable MPI over InfiniBand with reduced memory usage: an in-depth performance analysis," *Proceedings of the 2006 ACM/IEEE ...*, 2006.

[45] X. Ruan, Q. Yang, I. A. Mohammed, S. Yin, Z. Ding, J. Xie, J. Lewis, and X. Qin, "ES-MPICH2: A Message Passing Interface with enhanced security," *International Performance Computing and Communications Conference*, pp. 161–168, Dec. 2010.

[46] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," *Proceedings of the ...*, 2003.

[47] S. Genaud and C. Rattanapoka, "P2P-MPI: A peer-to-peer framework for robust execution of message passing parallel programs on grids," *Journal of Grid Computing*, pp. 1–25, 2007.

[48] T. Leblanc, R. Anand, E. Gabriel, and J. Subhlok, "VolpexMPI: an MPI library for execution of parallel applications on volatile nodes," *Recent Advances in Parallel ...*, 2009.

[49] G. E. Fagg and J. J. Dongarra, "FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world," *Recent Advances in Parallel Virtual Machine and ...*, pp. 1–8, 2000.

[50] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *Proceedings of the April 1820 1967 spring joint computer conference*, vol. 30, no. 3, pp. 483–485, 1967.

[51] J. Gustafson, "Reevaluating Amdahl's Law,"

[52] Y. Shi, "Reevaluating Amdahl's Law and Gustafson's Law," 1996.

[53] X. Li and M. Malek, "Analysis of speedup and communication/computation ratio in multiprocessor systems," *...-Time Systems Symposium, 1988., Proceedings ...*, pp. 282–288, 1988.

[54] N. Melab, M. Mezmaz, and E.-G. Talbi, "Parallel Hybrid Multi-Objective Island Model in Peer-to-Peer Environment," *19th IEEE International Parallel and Distributed Processing Symposium*, pp. 190b–190b.

[55] F. Marozzo, D. Talia, and P. Trunfio, "P2P-MapReduce: Parallel data processing in dynamic Cloud environments," *Journal of Computer and System Sciences*, 2011.

[56] E. Ridge, E. Curry, D. Kudenko, and D. Kazakov, "Parallel, asynchronous and decentralised ant colony system," *...for Parallel, Asynchronous and ...*, pp. 174–177, 2006.

[57] L. Baduel, D. Caromel, C. Delb, N. Gama, and S. E. Kasmi, "A parallel object-oriented application for 3d electromagnetism," *Parallel and ...*, 2004.

[58] Y. Sugita and Y. Okamoto, "Replica-exchange molecular dynamics method for protein folding," *Chemical Physics Letters*, vol. 314, no. November, pp. 141–151, 1999.

[59] E. Chang and R. Roberts, "An Improved Algorithm for Decentralized Extrema-Finding in Circular Configurations of Processes," *Communications of the ACM*, vol. 22, no. 5, pp. 281–283, 1979.

[60] H. Gonz, "A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers," *Software Practice and Experience*, vol. 40, no. 12, pp. 1135–1160, 2010.

[61] B. Stroustrup, "C++11 - the new ISO C++ standard," 2013.

[62] J. Katajainen and J. Träff, *A meticulous analysis of mergesort programs*, vol. 2. 1997.

[63] I. Wald and P. Slusallek, "State of the Art in Interactive Ray Tracing," *Eurographics Association*, 2001.

[64] OpenMP Architecture Review Board, "OpenMP Application Program Interface," 2011.

Appendix A

Documentation

This Appendix consists in documenting the full public API of IGCL, presenting an overview of the library's methods and providing reference for examples in the document. As of the writing of this document, the library's sources and this work's example algorithms can be accessed at:

`https://github.com/CanisLupus/igcl`

A.1 Common node methods

void start()

Initializes the Peer or Coordinator functionality, including binding the receiving socket and starting the message receiver thread. In a Peer, this method also handles the registration process with the group coordinator and subsequent establishment of connections to other peers, if available. This method must be called before using most other methods in the object; the exceptions are methods that set parameters which must be known before initialization.

void terminate()

Cleanly terminates the object, freeing resources and closing connections. In a Peer, the termination process alerts the group coordinator that this node terminated. In a Coordinator, the method sends a termination request to all connected peers.

void hang()

Forces the calling thread to block forever, unless an error occurs or a termination request is received from the group coordinator (only happens in Peer objects). This method allows Peers to wait until the Coordinator dismisses them, so that their exit does not trigger unnecessary connection failures and cleanups in the coordinator.

peer_id getId()

Returns the ID associated with this node. When called on a Coordinator it always returns 0. On Peer objects it returns the ID attributed by the group coordinator upon registration. The returned value is undefined until this registration completes.

uint getNPeers()

In a Coordinator, returns the number of peers currently executing the algorithm in the group. In a Peer, this function returns the number of peers executing the algorithm if the group layout is fixed, or 0 if the layout is free-formed (see Section 3.1.3).

const std::vector<peer_id> & downstreamPeers()

Returns a constant reference to the vector of nodes that are positioned after this node, according to the specified group layout. These are usually the nodes to which data is sent (and results are received from).

const std::vector<peer_id> & upstreamPeers()

Returns a constant reference to the vector of nodes that are positioned before this node, according to the specified group layout. These are usually the nodes from which data is received (and results are sent to).

uint nDownstreamPeers()

Convenience method that returns the number of downstream peers. This is equivalent to calling the *size* method of the vector reference returned by *downstreamPeers*.

uint nUpstreamPeers()

Convenience method that returns the number of upstream peers. This is equivalent to calling the *size* method of the vector reference returned by *upstreamPeers*.

std::vector<peer_id> getAllIds()

Returns a vector with the IDs of nodes that are known to this node. The coordinator will know every node in the group; peers will know the coordinator and any other nodes that are currently registered with them.

A.1.1 Send methods

```
template <typename ...T>
result_type sendTo(peer_id id, T * data, uint size)
result_type sendTo(peer_id id, T value)
```

Sends either a value of any non-pointer type or an array of such values — with length *size* — to the node with the given ID, *id*, which must be connected to this node. Nodes connected to a Peer are downstream peers, upstream peers and the group coordinator. The Coordinator is connected to every node, excluding itself. The method automatically handles sending depending on the connection to the peer: C sockets, libnice streams or relayed through the coordinator.

```
template <typename ...T>
result_type sendToAll(T * data, uint size)
result_type sendToAll(const T & value)
```

Sends either a value of any type or an array of such values — with length *size* — to every connected node. This is equivalent to calling the *sendTo* method for each connected node and, as before, automatically handles any potential connection type used.

```
template <typename ...T>
result_type sendToAllDownstream(T * data, uint size)
result_type sendToAllDownstream(const T & value)
```

These are equivalent to the *sendToAll*, but only send data to downstream peers instead of all connected peers.

```
template <typename ...T>
result_type sendToAllUpstream(T * data, uint size)
result_type sendToAllUpstream(const T & value)
```

These are equivalent to the *sendToAll*, but only send data to upstream peers instead of all connected peers.

A.1.2 Blocking receive methods

```
template<typename T>
result_type waitRecvFromAny(peer_id & id, T & value)
```

Blocking function that waits until data arrives from any peer. When it does, data is stored in *value* and the ID of the sending peer is stored in *id*.

```
template<typename T>
result_type waitRecvNewFromAny(peer_id & id, T * & data, uint & size)
result_type waitRecvNewFromAny(peer_id & id, T * & data)
```

Blocking function that waits until data arrives from any peer. When it does, data is stored in *data* and its size in number of elements is stored in *size*. The ID of the sending peer is stored in *id*. The programmer should eventually free the memory allocated for *data* using the *free* function. For cases where the size of the array is known and fixed, the function can be called without the *size* argument.

```
template<typename T>
result_type waitRecvFrom(peer_id id, T & value)
```

Blocking function that waits until data arrives from the peer with ID *id*. When it does, data is stored in *value*.

```
template<typename T>
result_type waitRecvNewFrom(peer_id id, T * & data, uint & size)
result_type waitRecvNewFrom(peer_id id, T * & data)
```

Blocking function that waits until data arrives from the peer with ID *id*. When it does, data is stored in *data* and its size in number of elements is stored in *size*. The programmer should eventually free the memory allocated for *data* using the *free* function. For cases where the size of the array-to-receive is known, the function can be called without the *size* argument.

A.1.3 Non-blocking receive methods

```
template<typename T>
result_type tryRecvFromAny(peer_id & id, T & value)
```

Non-blocking function that tests if data arrived from any peer. If it did, that data is stored in *value*, the ID of the sending peer is stored in *id* and the function returns SUCCESS. If there was no data, the function returns NOTHING and does not set any values.

```
template<typename T>
result_type tryRecvNewFromAny(peer_id & id, T * & data, uint & size)
result_type tryRecvNewFromAny(peer_id & id, T * & data)
```

Non-blocking function that tests if data arrived from any peer. If it did, that data is stored in *data*, its size in number of elements is stored in *size* and the function returns SUCCESS. If there was no data, the function returns NOTHING and does not set any values. In case the function returns SUCCESS, the programmer should eventually free the memory allocated for *data* using the *free* function. For cases where the size of the array-to-receive is known, the function can be called without the *size* argument.

```
template<typename T>
result_type tryRecvFrom(peer_id id, T & value)
```

Non-blocking function that tests if data arrived from the peer with ID *id*. If it did, that data is stored in *value* and the function returns SUCCESS. If there was no data, the function returns NOTHING and does not set any values.

```
template<typename T>
result_type tryRecvNewFrom(peer_id id, T * & data, uint & size)
result_type tryRecvNewFrom(peer_id id, T * & data)
```

Non-blocking function that tests if data arrived from the peer with ID *id*. If it did, that data is stored in *data*, its size in number of elements is stored in *size* and the function returns SUCCESS. If there was no data, the function returns NOTHING and does not set any values. In case the function returns SUCCESS, the programmer should eventually free the memory allocated for *data* using the *free* function. For cases where the size of the array-to-receive is known, the function can be called without the *size* argument.

A.1.4 Higher order functions: master-workers

These are generic functions that provide a simpler interface for sending and receiving data when using the master-workers layout or a similar customized one. See Figure 2.5a for a representation of the master-workers pattern. Internally, these methods use the downstream and upstream peers of the node, defined by the layout.

---

template<class T>
result_type distribute(T * data, uint sizeInUnits, uint unitSize, uint & startIndex, uint & endIndex)

This method is called by the master node to distribute data among all the slave (downstream) nodes and itself for processing.

data: pointer to an array of T-type values to distribute.
sizeInUnits: length of *data*, given in number of units (see *unitSize*).
unitSize: minimum unit of division for *data*. As an example, this allows sending a matrix using rows as units, instead of cells, which would likely cause incomplete rows to be sent.
startIndex: if the call is successful, it will contain the index of the start of the data section retained by the master.
endIndex: if the call is successful, it will contain the index of the end of the data section retained by the master.

---

template<class T>
result_type recvSection(T * & data, uint & startIndex, uint & endIndex, peer_id & masterId)

Method called by the receiving nodes to get their respective section and its indexes. This is the *distribute* method counterpart.

data: if the call is successful, it will contain the pointer to the received data. The memory it points should eventually be freed using *free*.
startIndex: if the call is successful, it will contain the index where the data section for this node begins.
endIndex: if the call is successful, it will contain the index where the data section for this node ends.
masterId: if the call is successful, it will contain the sender node ID.

---

template<class T>
result_type sendResult(T * data, uint sizeInUnits, uint unitSize, uint index, peer_id masterId)

After processing data, nodes call this method to return the results to the master. The type T of the elements in *data* is not necessarily the same as in *recvSection*, as the result can be completely different from the data that originated it. Likewise, the *unitSize* and *index* are related to this type and not the T of *recvSection*.

data: pointer to an array of T-type values with the results.
sizeInUnits: size of *data* in number of units.
unitSize: minimum unit of division for *data*.
index: index of the result in the final array.
masterId: ID to send result to. This is the ID acquired in *recvSection*.

---

template<class T>
result_type collect(T * data, uint sizeInUnits, uint unitSize)

Counterpart of *sendResult* that collects every result generated by slave nodes. This is called on the master node. The node should place its own section of results into the *data* array before calling this method.

data: pointer to an array of T-type values with the results.
sizeInUnits: size of *data* in number of units.
unitSize: minimum unit of division for *data*.

A.1.5 Higher order functions: divide-and-conquer

Similar to the higher order functions for master-workers pattern, these are generic communication functions for the tree layout or a similarly customized

one. See Figure 2.5b for a representation of the tree pattern for divide-and-conquer.

---

template<uint DEGREE=2, class T>
result_type branch(T * data, uint sizeInUnits, uint unitSize, uint & ownSize)

This method is called by a node to successfully branch data among its downstream peers. The branching factor defines the degree of ramification in the tree (i.e. the number of sections in which data is split at each level in a node). By default it is 2.

data: pointer to an array of T-type values to branch/divide.
sizeInUnits: length of *data*, given in number of units.
unitSize: minimum unit of division for *data*.
ownSize: if the call is successful, it will contain the size of the data section retained by this node for processing.

---

template<class T>
result_type recvBranch(T * & data, uint & sizeInUnits, peer_id & masterId)

Method called by the receiving nodes to get their respective section. This is the branch method counterpart.

data: if the call is successful, it will contain the pointer to the received data. The memory it points should eventually be freed using *free*.
sizeInUnits: if the call is successful, it will contain the size of *data*, in units.
masterId: if the call is successful, it will contain the sender node ID.

---

template<class T>
result_type returnBranch(T * data, uint sizeInUnits, uint unitSize, peer_id masterId)

After the algorithm handles the branch, nodes call this method to return it to the sender, already processed. The type T of the elements in *data* is not necessarily the same as in the *recvBranch* method, and *unitSize* is also related to this type and not the original T.

data: pointer to an array of T-type values with the results.
sizeInUnits: size of *data* in number of units.
unitSize: minimum unit of division for *data*.
masterId: ID to send result to. This is the ID acquired in *recvBranch*.

---

template<class T, class Func=std::function<void (T*,uint,T*,uint,T*)>>
result_type merge(T * data, uint sizeInUnits, uint unitSize, T * ownData,
uint ownSizeInUnits, Func merger)

Method that joins the branches of results of all nodes into a final array via a merger function. These are the results sent by calls to *sendBranch*. The function should work independently of the branching factor.

data: pointer to an array of T-type values where the results will be gathered.
sizeInUnits: size of *data* in number of units.
unitSize: minimum unit of division for *data*.
ownData: pointer to an array of T-type values with this node's results.
ownSizeInUnits: size of *ownData* in number of units.
merger: function that takes two pointers to arrays (branches) of results and their sizes, and joins them into the location of another array. This function is simply called several times if the branching factor is higher than 2.

A.2 Coordinator class methods

The Coordinator object has every previously described method, which are common to both the Coordinator and Peer classes. In addition, a few additional methods are provided.

Coordinator(int ownPort)

This is the only constructor of the Coordinator class. It receives as argument the port for the listening socket.

---

void setLayout(const GroupLayout & layout)

This method defines the layout used for the group. For a description of the available *GroupLayouts* and how to use them, refer to Section 3.1.3. This method must be called before invoking *start* on the object.

result_type waitForNodes(uint n)

For fixed-size layouts — used in algorithms that run on fixed quantities of nodes — this method lets the coordinator wait until the specified number of nodes are present. This means that $n-1$ peers should register and signal their "ready" state. Usually, the number of peers to wait for in a call to this method should simply be *layout.size()*.

A.3 Peer class methods

As is the case with the Coordinator, the Peer class also adds some additional methods to the shared ones that are described.

Peer(int ownPort, const std::string & coordinatorIp, int coordinatorPort)

This is the only constructor of the Peer class. It receives as argument the port for the listening socket, as well as the IP address and port of the group coordinator to which this peer will connect.

result_type barrier()

Synchronization method that blocks the peer until every other peer reaches a barrier. Note that this does **not** include the coordinator. When every peer arrives at the barrier, the coordinator sends an internal message that unblocks all of them.

void setAllowRelayedMessages(bool active)

Enables or disables the possibility of automatically setting connections between this and other peers as relayed through the coordinator, when no direct connection is possible.[1] If the option is disabled and the connection was unsuccessful but mandatory, the peer will automatically unregister with the coordinator. By default, messages are allowed to be relayed, but some algorithms might be inefficient when relaying messages, and thus should disable them. This method must be called before invoking *start* on the object.

[1] This does not affect methods *sendToPeerThroughCoordinator* and *sendToAllPeersThroughCoordinator*, which are explicitly called by the programmer; only the automatic definition of connections as relayed during connection establishment is affected.

```
template <typename ...T>
result_type sendToPeerThroughCoordinator(peer_id & id, T * data,
    uint size)
result_type sendToPeerThroughCoordinator(peer_id & id, T value)
```

Sends either a value of any type or an array of such values — with length *size* — to the coordinator, which will relay it to the node with the given ID, *id*. This includes nodes that have no direct connection to the sending peer.

```
template <typename ...T>
result_type sendToAllPeersThroughCoordinator(T * data, uint size)
result_type sendToAllPeersThroughCoordinator(T value)
```

Sends either a value of any type or an array of such values — with length *size* — to the coordinator, which will relay it to every peer in the group excluding the sender. This will include nodes that have no direct connection to the sending peer.

A.4 GroupLayout class methods

A.4.1 Fixed layouts

```
static const GroupLayout getMasterWorkersLayout(uint nNodes)
```

Obtains a fixed master-workers layout composed of *nNodes* nodes (see Figure 2.5a). In this layout, the coordinator has every other peer as a downstream peer. Those peers have no downstream peers and connect only to the coordinator (which is an upstream peer for them).

```
static const GroupLayout getTreeLayout(uint nNodes, uint degree)
```

Obtains a fixed tree layout composed of *nNodes* nodes (see Figure 2.5b). In this layout, the coordinator is the root node of the tree. Each node has several nodes as downstream peers; a quantity that is, at its maximum, equal to *degree* (the tree branching factor) times the number of tree levels below the node. If *nNodes* is not a power of *degree*, some nodes will have missing downstream peers at the last tree level.

static const GroupLayout getPipelineLayout(uint... nNodesOfSection)

Obtains a fixed pipeline layout composed of several nodes per section (see Figure 2.5c). This layout is called with several arguments, which are the number of nodes at each level in the pipeline. The peers from one section are downstream from the section that comes before. For example, in *getPipelineLayout(1, 2, 2)*, peers 1 and 2 are downstream from 0 and peers 3 and 4 are downstream from 2 and 3.

static const GroupLayout getRingLayout(uint nNodes)

Obtains a fixed ring layout composed of *nNodes* nodes (see Figure 2.5d). Each peer is sequentially placed in the layout with a connection to the current last peer (thus being downstream from it). In this layout, the coordinator is both the root and sink node, being simultaneously upstream of the second peer and downstream of the second to last.

static const GroupLayout getAllToAllLayout(uint nNodes)

Obtains a fixed all-to-all layout composed of *nNodes* nodes. Each peer is downstream of every other peer, including the coordinator.

A.4.2 Free-formed layouts

static const GroupLayout getFreeMasterWorkersLayout()

Obtains a free-formed master-workers layout. In this layout, the coordinator has every other peer as a downstream peer. Those peers have no downstream peers and connect only to the coordinator (which is an upstream peer for them). There is no fixed size for this layout, as the coordinator will dynamically add or remove nodes to and from it as they register or de-register, respectively.

static const GroupLayout getFreeAllToAllLayout()

Obtains a free-formed all-to-all layout. Each peer is downstream of every other peer, including the coordinator. There is no fixed size for this layout, as the coordinator will dynamically add or remove nodes to and from it as they register or de-register, respectively.

A.4.3 Manual layouts

In the manual creation mode, the programmer has to explicitly insert IDs of nodes and their connections, knowing that the group coordinator always has ID 0. In the layout example in Listing A.1, the coordinator is directly connected to the nodes with ID 1 and 2, these are both connected to nodes 3 and 4 (i.e. 1 has a connection to 3 and 4, and so does 2), and these in turn connect to 5. This layout is actually equivalent to calling *getPipelineLayout(1, 2, 2, 1)*, which uses as arguments the number of nodes at each level in the pipeline.

```
1 GroupLayout layout;
2 layout.from(0).to(1,2);
3 layout.from(1,2).to(3,4);
4 layout.from(3,4).to(5);
```

Listing A.1: Manually defined group layouts

This is possible because the *from* method returns a special object of type *Sources*, which implements the *to* method and can change the layout's contents.

A.4.4 Other layout methods

uint size() const;

Returns the current size of the layout.

void print() const;

Prints the current layout text description in terms of nodes and their connections.

A.5 NBuffering class methods

NBuffering (uint bufferingDepth, uint nJobs, uint blockSize,
std::function<void (peer_id, uint)> sendJob)

This is IGCL's N-Buffering class declaration. The programmer needs to provide the level of buffering he or she requires (double, triple, etc.), the number of jobs there will be, how much jobs are sent at once (this is different from the buffering level, as we will explain), and a function that sends a job with a certain index to a given peer. The third argument for the NBuffering constructor is merely an optimization that avoids internally creating a different object for each job. Instead, the programmer can define that jobs are sent in groups of X, thus making these blocks the units that are buffered. For example, for a buffering level of 5 and groups of 3 jobs, only a maximum of 5 calls to *sendJob* will be made, but the respective maximum number of jobs sent until the peer queue is filled is 15.

---

void addPeer(peer_id id)

Adds a peer with a given ID to the buffering class, meaning that it is now available to receive jobs.

---

void addPeers(std::vector<peer_id> peerIds)

Adds a vector of peer IDs to the buffering class, meaning that they are all now available to receive jobs.

---

void removePeer(peer_id id)

Removes a peer with a given ID from the buffering class, making it no longer receive jobs and putting the jobs that were previously sent to it in a high priority queue that will eventually be consumed by other nodes as part of their buffering (i.e. the jobs will be sent again but to new peers).

---

void bufferToAll()

Tries to buffer jobs to all available peers, accounting for the buffering level and number of already sent jobs.

---

void bufferTo(peer_id id)

Tries to buffers jobs to a specific peer, accounting for the buffering level and number of already sent jobs.

---

uint completeJob(peer_id id)

Marks the first job in the given peer queue as completed (jobs are simply assumed to be completed in order). Returns the index of the job.

---

bool allJobsSent()

Returns true when all jobs were sent; false otherwise. Note that this function can return a true value and later a false value, if a peer disconnects and its jobs are re-assigned.

---

bool allJobsCompleted()

Returns true when all jobs are marked complete; false otherwise.

Appendix B

Code Examples

This Appendix contains some code listings comparing IGCL usage to MPI's in two examples: matrix multiplication and merge sort.

```
// ...
GroupLayout layout = GroupLayout::getSortTreeLayout(nParticipants,
    2);
node->setLayout(layout); // in Coordinator
// ...
if (id > 0)
    node->recvBranch(array, originalSize, parent);

node->branch<2>(array, originalSize, 1, size);
std::sort(array, array+size);

if (node->nDownstreamPeers() > 0) {
    finalArray = (DATATYPE*) malloc(originalSize*sizeof(DATATYPE));
    node->merge(finalArray, originalSize, 1, array, size, joinSort);
    free(array);
    array = finalArray;
}

if (id > 0)
    node->returnBranch(array, originalSize, 1, parent);
// ...
```

Listing B.1: Implementation of a parallel merge sort using IGCL

```
1  //...
2  if (id > 0) {
3      MPI_Recv(&originalSize, 1, MPI_INT, MPI_ANY_SOURCE,
           MPI_ANY_TAG, MPI_COMM_WORLD, &status);
4      array = (DATATYPE*) malloc(originalSize*sizeof(DATATYPE));
5      parent = status.MPI_SOURCE;
6      MPI_Recv(array, originalSize, num_mpi_type, status.MPI_SOURCE,
           MPI_ANY_TAG, MPI_COMM_WORLD, &status);
7  }
8
9  std::stack< std::pair<int, uint> > sentSizes;
10 uint size = originalSize;
11 int mult = 1;
12 int sendId = 2*mult * id + mult;
13
14 // send according to tree layout
15 while (sendId < nNodes) {
16     uint sendSize = size / 2;
17     size = size - sendSize;
18
19     MPI_Send(&sendSize, 1, MPI_INT, sendId, 99, MPI_COMM_WORLD);
20     MPI_Send(array+size, sendSize, num_mpi_type, sendId, 99,
           MPI_COMM_WORLD);
21
22     sentSizes.push(std::pair<int, uint>(sendId, sendSize));
23     mult *= 2;
24     sendId = 2*mult * id + mult;
25 }
26
27 std:sort(array, array+size);
28
29 // upstream section that merges sections
30 if (size < originalSize) {
31     DATATYPE * other = (DATATYPE*)
           malloc(originalSize*sizeof(DATATYPE));
32
33     while (size < originalSize) {
34         const std::pair<int, uint> & elem = sentSizes.top();
35         sentSizes.pop();
36
37         MPI_Recv(array+size, elem.second, num_mpi_type, elem.first,
               MPI_ANY_TAG, MPI_COMM_WORLD, &status);
38
```

```
39          joinSort(array, size, array+size, elem.second, other);
40          size += elem.second;
41          std::swap(array, other);
42      }
43      free(other);
44  }
45
46  if (id > 0)
47      MPI_Send(array, size, num_mpi_type, parent, 99, MPI_COMM_WORLD);
48  //...
```

Listing B.2: Implementation of a parallel merge sort using MPI

```
1   // ...
2   if (id == 0) { // master distributes data to slaves
3       node->sendToAll(mat_b, MATSIZE * MATSIZE);
4       node->distribute(mat_a, MATSIZE, MATSIZE, iniRowIndex,
            endRowIndex);
5   }
6
7   if (id > 0) {
8       node->waitRecvNewFromAny(masterId, mat_b);
9       node->recvSection(mat_a, iniRowIndex, endRowIndex, masterId);
10  }
11
12  for (uint i = 0; i < endRowIndex-iniRowIndex; i++) { // multiply
13      for (int j = 0; j < MATSIZE; j++) {
14          DATATYPE sum = 0;
15          for (int k = 0; k < MATSIZE; k++) {
16              sum += mat_a[i*MATSIZE+k] * mat_b[j*MATSIZE+k];
17          }
18          mat_result[i*MATSIZE+j] = sum;
19      }
20  }
21
22  if (id > 0)
23      node->sendResult(mat_result, endRowIndex-iniRowIndex, MATSIZE,
            iniRowIndex, masterId);
24
25  if (id == 0) // master gathers results from all slaves
26      node->collect(mat_result, MATSIZE, MATSIZE);
27  // ...
```

Listing B.3: Implementation of a parallel matrix multiplication algorithm using IGCL

```
//...
MPI_Bcast(mat_b, MATSIZE * MATSIZE, num_mpi_type, 0,
    MPI_COMM_WORLD);

if (id == 0) // master distributes data to slaves
{
    int ini, end;
    int nRowsPerProcess, remainder;

    // calculate portion for each node
    nRowsPerProcess = MATSIZE / nNodes;
    remainder = MATSIZE % nNodes;

    iniRowIndex = 0;
    endRowIndex = nRowsPerProcess + (remainder-- > 0 ? 1 : 0);
    ini = endRowIndex;  // first rows stay with the master

    for (int i = 1; i < nNodes; i++) // for each slave
    {
        end = ini + nRowsPerProcess + (remainder-- > 0 ? 1 : 0);

        MPI_Send(&ini, 1, MPI_INT, i, MASTER_TO_SLAVE_TAG,
            MPI_COMM_WORLD);
        MPI_Send(&end, 1, MPI_INT, i, MASTER_TO_SLAVE_TAG,
            MPI_COMM_WORLD);
        MPI_Send(&mat_a[ini*MATSIZE+0], (end - ini) * MATSIZE,
            num_mpi_type, i, MASTER_TO_SLAVE_TAG, MPI_COMM_WORLD);

        ini = end;
    }
}

if (id > 0)
{
    MPI_Recv(&iniRowIndex, 1, MPI_INT, 0, MASTER_TO_SLAVE_TAG,
        MPI_COMM_WORLD, &status);
    MPI_Recv(&endRowIndex, 1, MPI_INT, 0, MASTER_TO_SLAVE_TAG,
        MPI_COMM_WORLD, &status);
```

```
33      mat_a = (DATATYPE *) malloc((endRowIndex - iniRowIndex) *
            MATSIZE * sizeof(DATATYPE));
34      MPI_Recv(mat_a, (endRowIndex - iniRowIndex) * MATSIZE,
            num_mpi_type, 0, MASTER_TO_SLAVE_TAG, MPI_COMM_WORLD,
            &status);
35  }
36
37  if (id > 0)
38  {
39      for (uint i = 0; i < endRowIndex-iniRowIndex; i++) { // multiply
40          for (int j = 0; j < MATSIZE; j++) {
41              DATATYPE sum = 0;
42              for (int k = 0; k < MATSIZE; k++) {
43                  sum += mat_a[i*MATSIZE+k] * mat_b[j*MATSIZE+k];
44              }
45              mat_result[i*MATSIZE+j] = sum;
46          }
47      }
48  }
49
50  if (id > 0)
51  {
52      MPI_Send(&iniRowIndex, 1, MPI_INT, 0, SLAVE_TO_MASTER_TAG,
            MPI_COMM_WORLD);
53      MPI_Send(&endRowIndex, 1, MPI_INT, 0, SLAVE_TO_MASTER_TAG,
            MPI_COMM_WORLD);
54      MPI_Send(&mat_result[iniRowIndex*MATSIZE+0], (endRowIndex -
            iniRowIndex) * MATSIZE, num_mpi_type, 0,
            SLAVE_TO_MASTER_TAG, MPI_COMM_WORLD);
55  }
56
57  if (id == 0) // master gathers results from all slaves
58  {
59      int nSlaves = nNodes-1;
60      while(nSlaves--) // receive from all slaves
61      {
62          int ini, end;
63          MPI_Recv(&ini, 1, MPI_INT, MPI_ANY_SOURCE,
                SLAVE_TO_MASTER_TAG, MPI_COMM_WORLD, &status);
64          MPI_Recv(&end, 1, MPI_INT, status.MPI_SOURCE,
                SLAVE_TO_MASTER_TAG, MPI_COMM_WORLD, &status);
65          MPI_Recv(&mat_result[ini*MATSIZE+0], (end - ini) * MATSIZE,
                num_mpi_type, status.MPI_SOURCE, SLAVE_TO_MASTER_TAG,
```

```
            MPI_COMM_WORLD, &status);
66      }
67  }
68  //...
```

Listing B.4: Implementation of a parallel matrix multiplication algorithm using MPI

Appendix C

Result Tables

In this Appendix we present the result tables that created the plots from Chapter 4. These contain the mean values of 30 executions along with their standard deviations. All values are given in seconds.

# of nodes	1	2	3	4	5	6	7	8
IGCL	17.385 ±0.039	8.984 ±0.118	6.714 ±0.120	5.310 ±0.142	5.054 ±0.107	5.082 ±0.110	4.625 ±0.088	4.334 ±0.108
Open MPI	17.398 ±0.046	9.282 ±0.103	7.148 ±0.126	5.665 ±0.127	5.490 ±0.134	5.557 ±0.172	4.879 ±0.138	4.525 ±0.136

Table C.1: Data of Figure 4.7. Matrix multiplication: IGCL and Open MPI performance.

# of nodes	1	2	3	4	5	6	7	8
IGCL	4.283 ±0.039	3.670 ±0.118	3.425 ±0.120	3.002 ±0.142	3.028 ±0.107	2.873 ±0.110	2.894 ±0.088	2.847 ±0.108
Open MPI	4.277 ±0.046	3.888 ±0.103	3.359 ±0.126	3.077 ±0.127	3.189 ±0.134	3.032 ±0.172	3.013 ±0.138	3.030 ±0.136

Table C.2: Data of Figure 4.8. Merge sort: IGCL and Open MPI performance.

# of nodes	1	2	3	4	5	10
IGCL	15.680 ±0.052	6.725 ±0.085	6.082 ±0.054	5.930 ±0.038	5.911 ±0.015	5.953 ±0.022

Table C.3: Data of Figure 4.9. Ray tracing: effect of various levels of buffering.

buffering level	1	2	3	4	5
1 node	37.282 ±0.027	15.034 ±0.018	5.795 ±0.013	2.167 ±0.009	1.925 ±0.010
2 nodes	18.661 ±1.885	9.030 ±0.281	2.688 ±0.187	2.027 ±0.114	1.922 ±0.093
3 nodes	12.454 ±1.404	5.944 ±0.707	2.186 ±0.265	1.991 ±0.085	1.980 ±0.108
4 nodes	9.345 ±0.475	4.503 ±0.185	2.013 ±0.082	1.981 ±0.077	1.955 ±0.056
5 nodes	7.489 ±0.004	3.635 ±0.009	2.023 ±0.098	1.957 ±0.058	1.967 ±0.060

Table C.4: Data of Figure 4.10. Ray tracing: effect of various levels of buffering.

# of nodes	1	2	4	6	8
IGCL	28.567 ±0.115	14.786 ±0.157	8.023 ±0.099	6.664 ±0.017	5.953 ±0.022
threads	27.748 ±0.128	14.334 ±0.061	7.721 ±0.112	6.303 ±0.012	5.678 ±0.048

Table C.5: Data of Figure 4.11. Ray tracing: performance of IGCL versus threads.

# of nodes	1	2	4	6	8
with bound exchanges	43.650 ±2.408	21.750 ±1.116	13.757 ±0.845	8.157 ±0.683	6.716 ±0.098
without bound exchanges	= =	34.435 ±2.030	33.804 ±0.948	31.300 ±0.414	33.830 ±0.153

Table C.6: Data of Figure 4.12. TSP: networked performance when exchanging bounds or not.

# of nodes	1	2	4	6	8
IGCL	1.989 ±0.354	4.023 ±0.181	6.308 ±0.587	8.148 ±0.755	10.489 ±0.846

Table C.7: Data of Figure 4.13. Matrix multiplication: networked execution times.

# of nodes	1	2	4	6	8
IGCL	0.059 ±0.021	1.216 ±0.020	1.810 ±0.125	2.036 ±0.109	2.440 ±0.200

Table C.8: Data of Figure 4.14. Merge sort: networked execution times.

# of nodes	1	2	4	6	8
no buffering	4.272 ±0.603	4.612 ±0.544	5.181 ±0.770	5.182 ±0.625	5.334 ±0.608
2-buffering	= =	5.383 ±0.701	6.003 ±0.779	5.817 ±0.644	6.668 ±0.593
5-buffering	= =	6.413 ±0.679	6.096 ±0.554	8.054 ±0.620	9.682 ±0.769

Table C.9: Data of Figure 4.15. Ray tracing: networked execution times.

# of nodes	1	2	4	6	8
no buffering	4.785 ±0.566	5.107 ±0.688	4.783 ±0.490	4.651 ±0.531	4.167 ±0.425
2-buffering	= =	4.722 ±0.658	4.294 ±0.511	4.159 ±0.470	4.369 ±0.323
5-buffering	= =	4.119 ±0.422	4.228 ±0.385	4.105 ±0.359	4.422 ±0.369

Table C.10: Data of Figure 4.16. Ray tracing: networked execution times (char version).

# of nodes	1	2	4	6	8
normal connections	2.743 ±0.029	1.710 ±0.080	1.210 ±0.130	1.220 ±0.049	0.974 ±0.107
libnice connections	- -	- -	4.210 ±0.080	4.463 ±0.090	6.587 ±0.210

Table C.11: Data of Figure 4.17. Merge sort: local analysis of normal versus libnice connections.

# of nodes	1	2	4	6	8
normal connections	43.650 ±2.408	21.750 ±1.116	13.757 ±0.845	8.157 ±0.683	6.716 ±0.098
libnice connections	- -	- -	13.367 ±0.562	8.431 ±0.767	6.904 ±0.108
relayed connections	- -	- -	13.567 ±0.962	8.214 ±0.490	7.108 ±0.303

Table C.12: Data of Figure 4.18. TSP: networked analysis of normal versus libnice connections. Includes relayed connections.