GPU, a framework for distributed computing over Gnutella

Tiziano Mengotti
Master Thesis in Computer Science
ETH Zürich, Switzerland

March 29, 2004

Advisors: Prof. Wesley P. Petersen
Dr. Fabrice Marchal
Prof. Kai Nagel

Abstract

This thesis proposes a framework for distributed computing based on the Peer-to-Peer network Gnutella, suitable for Monte Carlo jobs, for brute-force searches, and for randomized algorithms in general, where almost no communication is necessary. Gnutella does not recognize privileges between users: everyone can get CPU resources, if needed. The framework is being developed using common Open Source techniques and is intended as a proof of concept. Finally, we discuss the coupon collection problem and attempt a fractal argument for the small world problem, both relevant for the framework.

Contents

1 Introduction

Distributed Computing is a way to cluster computers, so that they perform a common computation. Clients are specially assigned low priority processes which use only computing power that would be wasted anyway, which can be well in excess of 90%. In fact, modern operating systems are most of the time idle and just wait for user input. (Wikipedia definition [4]). Although most frameworks are centralized, the framework discussed here is distributed on a Peer-To-Peer network [20] called Gnutella [5], instead.

Gnutella is a protocol sitting on top of the TCP/IP layer that allows computers to connect each other in a random fashion to form a network. The first Gnutella client was developed by Justin Frankel and Tom Pepper of Nullsoft, a division of AOL, in early 2000. After connections between computers are established, users can share their files and download them from each other. Similarly, the proposed framework (called GPU[1] [2,3,20]) allows users to share CPU-cycles on the network.

The GPU application runs in background and is able to start several threads with low priority to handle incoming requests. Thus, the user volunteering his/her computer is not bothered too much.

These threads access functionality stored as plugins. Plugins are the backend side of the GPU framework: they extend its capability to handle new requests. Plugins are "signed" with PGP [7], a freeware framework for asymmetric cryptography, to ensure they were not modified to run malicious code.

Frontends permit users to submit jobs and to visualize their results in an easy way. Users can keep up-to-date frontends, backends and the GPU itself by accessing the "autoupdate" features of the application.

Chapters 2, 3, 6, 7 and 8 are intended for everyone interested in distributed computing. Chapter 4 focuses on two theoretical problems that relate to Gnutella and to the framework. Chapter 5 and 6 are for developers that would like to extend the framework with plugins or frontends.

2 Architecture Overview

2.1 Application Overview

The application is organized as follows: GPU discovers new computers on the network, establishes and keeps a predefined number of connections with

[1] Global Processing Unit or Gnutella Processing Unit

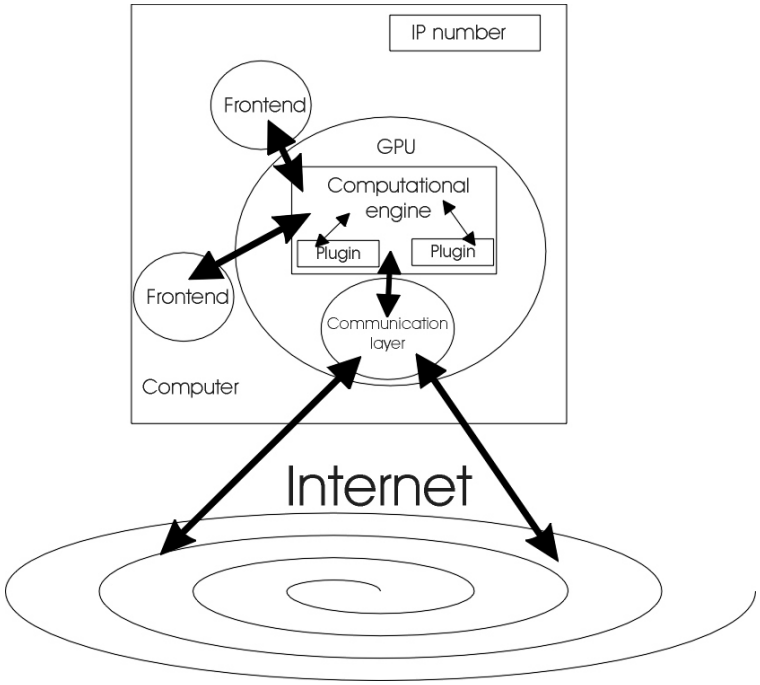

Figure 1: Application Overview

other computers. When GPU detects incoming requests, it will start threads or queue them for later handling, if the maximum number of threads is reached. Once the computing thread is done, GPU sends the result back to the requester and frees that thread from memory.

Plugins encapsulate algorithms that answer the incoming request: the brute force attack to the discrete logarithm is one example [3], another is the computation of a partial differential equation using the random walkers approach and the Feynman-Kac formula [1]. Games like chess, where chessboards become leafs of a tree to be evaluated by a fitness function, could be suitable for the framework as well, if implemented correctly.

Plugins are a library of functions, loaded at runtime by GPU. In Windows, this dynamic link mechanism is provided by DLLs (files with extension `.dll`). In Linux, the same mechanism is called *shared objects* and the corresponding extension is `.so`. We give here a brief overview on how to write plugins with graphical output.

Frontends are a complement to plugins; they simplify the submission of jobs and the visualization of results: imagine playing chess by typing the 64 numbers representing the chessboard each time or visualizing the result of the partial differential equation by reading the list of results for each

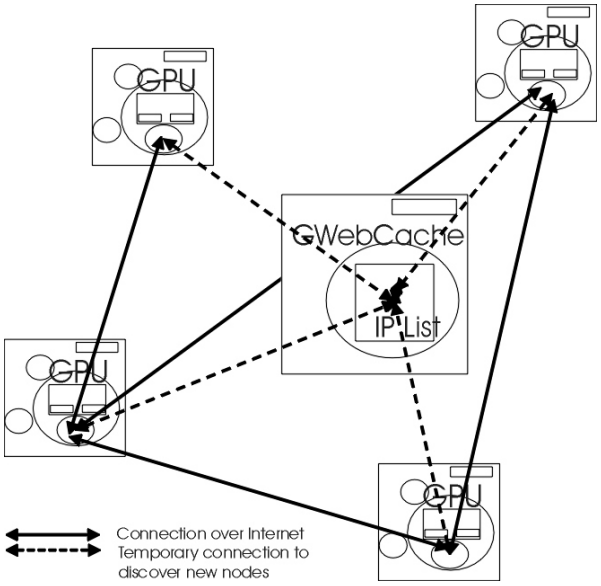

Figure 2: Network Overview

coordinate. Frontends communicate with GPU using Windows messages (a Linux implementation would use pipe and signals to achieve the same goal). Through this privileged channel, the frontend is able to submit jobs and to receive results.

In this document, we give a detailed overview on how to implement frontends. For more insights on the framework, one should read the previous work on GPU as well [3].

2.2 Network Overview

At startup, GPU does not know of any other computer to connect to. Gnutella provides a node discovery technique to solve this problem (the `GWebCache`[9]): web servers run by volunteers host special scripts (implemented in `php`, `Perl` or as compiled `cgi`) where a new GPU client can register its IP number. GPUs successively download a list of IPs from previous registrations. These IPs are then tried one by one to find other computers that still run the GPU.

Depending on the connection speed, GPU tries to keep the number of established connections between a minimum and a maximum. Typical values for a 28 kbit modem are a minimum of 3 and a maximum of 6 connections, for an ADSL connection, these are 8 and 20, respectively. If GPU is below the minimum number and short of IPs, it connects again to the above mentioned web servers (sometimes referred to as *host catchers*), registers its IP number

again and tries to connect to the new IPs it downloads to establish *outgoing connections*. At the same time, new nodes entering the network see the IP number registered by the GPU on the web servers and attempt to connect to them, so that the GPU gets *incoming connections*.

Incoming packets representing computational jobs or file searches are routed using *flooding*. In *flooding*, once a packet reaches a node, it is immediately sent out over all its other connections [3]. A Time-To-Live stamp on packets prevents them from walking indefinitely through the network.

Additionally, GPUs keep a list of packets they routed recently: an incoming packet that already belongs to the list is thrown away. In this manner, the list cuts circles in the network. If we think of the network as a graph where nodes represent computers and links connections between them, the list ensures that our graph is a *Direct Acyclic Graph* (*DAG*).

In this list, the tuple `(packet, incoming connection)` is stored. Using tuple information, GPUs can then route back the answer directly to the job submitter, without using the *flooding* approach.

Briefly, Gnutella uses *flooding* to submit requests and dedicate routing to receive answers. Finally, because Gnutella does not scale well in this configuration, *Ultrapeers* are introduced so that the network tends to shape itself into a tree. *Ultrapeers* are fast nodes that maintain many connections, while slow nodes only keep one connection to one *Ultrapeer*.

A short introduction to Peer-to-Peer is [20], Gnutella and GPU are discussed in [3].

3 Programming for the GPU framework

3.1 Programming language

The framework is implemented in Delphi/Kylix [18], a Pascal dialect. Delphi compiles Pascal source code for the Windows architecture and Kylix does the same for for Linux. However, the produced object files are targeted to the Intel architecture only. Another disadvantage is that only Borland produces Delphi/Kylix.

Delphi/Kylix offers advantages as well; it is a standard, common RAD tool (*Rapid Application Development*) used in industry, where robustness, simplicity and speed, both in development and use, are a vital requirement for companies. Delphi and its Linux counterpart use a component system that allow a deeper encapsulation of data and methods than the normal object oriented model does (for another programming language that uses components, see *JavaBeans*[22]). Once written, components look like visual

icons with properties and events. For example, a click on an icon's event automatically generates the method call in the correct object. The task left to the developer is to add the lines of code for the method itself.

Pascal is well-known for its friendly compiler messages, too. Semantic type-checks avoid passing wrong arguments to subroutines. In particular, integers and pointers cannot be exchanged arbitrarily. Range-checking on arrays at runtime ensure that the program never violates its address space. The debugger, helped by the exception handling mechanism, is able to track down the wrong line of source code in most of the cases.

A broad range of Open Source components is available and interfaces to common libraries like OpenGL is provided. Good programmed components with good debugging capabilities allow developers to get complex and robust applications running quickly.

In short, the availability of Java to run on multiple architectures is sacrificed in exchange for a RAD-tool with comparable object oriented features, but able to generate compiled code like C++ does.

3.2 Differences with centralized frameworks

Many centralized frameworks exist today. In essence, a server distributes tasks to clients and collects back results when the clients finish.

`Seti@home` [10], the first successful distributed computing framework works as follows: an old supercomputer distributes data from a radio-telescope to normal computers run by three million volunteers. A small program installed on these computers analyzes the data in the background using little CPU-power while the user is working but full CPU-power if the screensaver is active.

The analysis of data is done with a Fast Fourier Transform, to search for Gaussian and peaks that might be of extraterrestrial nature. Results are then sent back to the old supercomputer. Possibly interesting results are then post-processed by scientists.

On the same track of `Seti@home`, many others have followed: for example `Folding@home` [11], `Climateprediction.net` [12], `distributed.net` [13] and `Chessbrain` [14]. An attempt to unify many projects under the same infrastructure is currently done by `BOINC` (*Berkeley Open Infrastructure for Distributed Computing* [15]). As of today, (February 2004) `BOINC` is in Beta-Test and we can soon expect to see it running on millions of machines.

Being of distributed nature, the GPU project cannot position itself with these big centralized projects. With Gnutella, each user reaches only around

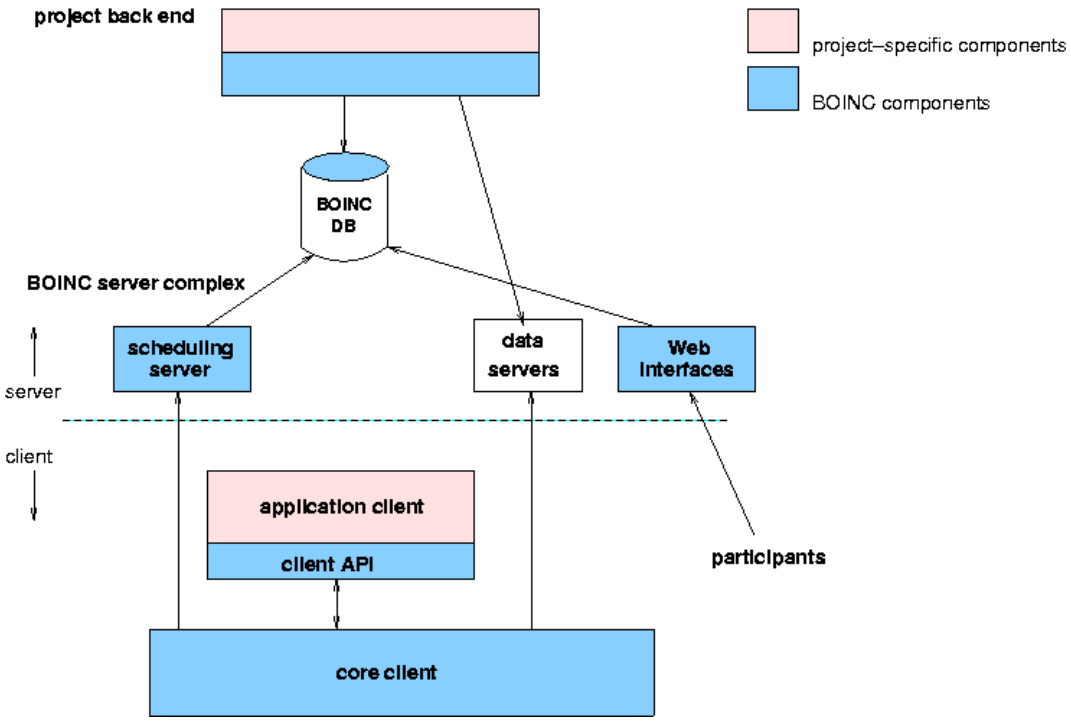

Figure 3: Architecture of the `BOINC`[15] project, an extensible centralized model.

2000 computers[2] (with a Time-To-Live stamp of 7), about the size of a normal Beowulf Cluster. More computers could be built into a cluster, but each machine will see only about 2000 because Gnutella packets have a count-down counter decreased each time a computer is reached; Once decremental count gets zero, the packet is destroyed and no more nodes could be seen [3].

Additionally, clients have to stay online and operational; GPUs cannot simply disconnect and crunch data offline because they have to keep the network operational by forwarding jobs and answers. This disadvantage might slowly fade away, thanks to the new ADSL connections that provide 24-hour access for a reasonable monthly fee.

The main advantage of the GPU framework is that everyone can use the framework for his/her own purposes. Users running `Chessbrain` on their home computer follow one match only (in February 2002, against Grand Master Peter Nielsen). Users running GPU can occasionally play against the entire GPU framework (although please notice that the provided Chess plugin, `chessbackend.dll`, is an example of a frontend and still does not

[2] see 4.3 for an estimation of this number

implement parallelism).

We can think of scientists with small budgets developing plugins for the framework or of developers implementing distributed databases in a similar way. Through the "autoupdate" routines, the plugin will eventually spread to all GPUs. Note that the choice to update is left to the user.

Some other weaker differences and considerations are:

- A centralized model is normally backed by an institution that constantly produces data for the computing network.
- In a centralized model, volunteers can group in teams and compete both individually or in a team. This is an important motivation factor.
- Motivation for GPU should arise from the fact that everyone is part of a bigger entity, represented by the virtual supercomputer. Volunteers compute for others, but can ask for computational time as well.

3.3 GPU Protocol

The GPU protocol is very simple. A more advanced proposal is given in `BOINC` for the centralized model, and in [16] for a Peer-to-Peer approach. Due to Delphi, and viewed that the computing engine is encapsulated in a component, getting other protocols on the framework is easy, namely throw another component on the Delphi form. Currently in GPU, the component that does connections to Gnutella was written and is maintained by a German programmer, Kamil Pogorzelski [17].

As we saw previously, in the file-sharing network there are two types of packets (there are more [5], but these are the ones that concern us): the request for files and the answer. The GPU framework does it similarly; it creates a request string or an answer and spreads it throughout the Gnutella disguised as search string for files. This is inelegant, but fits our need of rapid prototyping and of compatibility with other Gnutella clients[3]. This might change in future, as we hope to provide a cleaner implementation with Gnutella packets dedicated to the GPU framework. A priority queue might be useful, too. Users with long uptime and many computed results should be privileged with high-priority requests.

A request for computational time is built as follows; fields are separated by GPU with ':'.

[3]although most Gnutella clients drop GPU strings because they are too long

Request Header	identifies search for files as computational job
Internal ID	each job gets a different identifier, so that the list in Gnutella does not block the packet, if the same job is submitted twice.
GUID	Global Unique Identifier, identifies requester on network
JobID	ID for the GPU Job
Command	String to be interpreted by the Virtual Machine

Table 1: GPU Request

Result header	identifies search for files as result of a computation
Internal ID	each result gets a different identifier, so that Gnutella does not block the packet, if the result is submitted twice.
GUID of requester	Global Unique Identifier of the requester
JobID	ID for the GPU Job
Computing Time	how long it took to compute the result
Result	String in the Virtual Machine Format

Table 2: GPU Result

4 Theoretical considerations

4.1 Randomized algorithms

To implement plugins and frontends in the proposed framework, it is first important to recognize that the entire network does not provide any guarantees of availability. Computing nodes might go down in the middle of a job for several reasons like power failure, network congestion or just because the computer is shut down by the user. This is a crucial difference, compared to the `MPI` [1,8] or the `OpenMP`[1] library, where the developer decides *a priori* how many nodes will compute.

A second issue, at least as serious as the first, involves the nature of Gnutella: all packets sent to the connected computers will be exactly the same. Each computing node will get the same arguments to start the computation, although they may compute different results. In a similar way, all file sharing programs may get the same request for a particular song's author, but are able to answer with different files they host, all from the same author. Notice that this issue is intrinsic in the Gnutella network: each node knows only its neighborhood[4]. We do not have topology information

[4]although this could be changed with the Network Mapper, see section 6.6

and therefore we cannot split our computation in pieces of different length. Moreover, the entire network topology changes rapidly. So, there is no way to pass particular arguments to particular nodes.

To solve the issue, we propose several solutions that all involve some use of randomness.

Special randomized algorithms can alleviate our concerns. For some categories of problems, Monte Carlo methods produce reasonable results. In this category, we can place the plugins that solve Pi and the partial differential equations with the Feynman-Kac formula [1] (`pi.dll, pde.dll, pde3d.dll`). Most Monte Carlo methods are not suitable to get an exact solution though they provide good approximations.

Genetic algorithms, and genetic algorithms with neural networks might be suitable for the framework: however, frontends will be charged with additional management tasks.

Some brute-force tasks can be randomized using mathematical properties of the problem. As an example the discrete logarithm problem can be solved using the *Pollard-Rho* algorithm[5].

As further brute-force tasks, we mention here the Search for Golomb Rulers and attacks on encrypted codes.

All brute-force tasks, where a big range of possible solutions is searched, can be randomized; We split our search space into m sub domains. How many times n do we need to solve an incoming job with random parameters that specify a sub domain i ($1 \leq i \leq m$) of our task, to receive at least an answer from each sub domain?

4.2 Throwing N stones into M boxes

We reformulate the problem in the previous section differently: assuming we have M boxes, how many stones N do we have to throw randomly to get at least one stone inside each box with probability p_0 provided that $N \geq M$?

Notice that in the reformulation, beside the constraint were we ensure that the number of stones is at least equal to the number of boxes, we introduced a probability p_0. This because it is impossible to ensure that at least one stone will be in each box with 100% probability. Although with very little probability, all stones might fall in the same box even if we threw thousands of them. Therefore, we can think only in expectation values. In particular,

[5]The plugin that solves the discrete logarithm problem in GPU (`crypto.dll` as presented in [3]) uses a modified version of the *baby step-giant step* algorithm that underwent modifications described above. It was developed before the author was aware of the *Pollard-Rho* algorithm.

we are interested if the number of stones to be thrown N grows with the sub domain size M while keeping a reasonable and constant probability p_0.

The problem is often referred in literature as the "coupon collector problem" or "classical occupancy problem"[28]. A nice Java applet to visualize it is [27].

4.2.1 Focus on one particular box

Firstly, we focus on one box only:
For each thrown stone, the probability that a stones falls in this particular box is $\frac{1}{M}$. The probability that it falls on another box is $1 - \frac{1}{M} = \frac{M-1}{M}$.

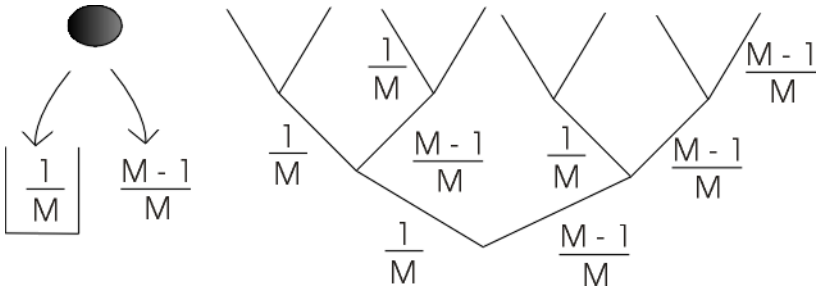

Figure 4: Probability tree leads to a binomial distribution.

If we throw N stones, we can build a binary tree of height N. In each tree's node, we choose the left path if the stone falls into the particular box we focus. The right path is chosen if it falls in another one. The probability that no stone falls in our box after N throws is $p_{none} = \frac{M-1}{M}^N$ and follows the rightmost arm of the tree with height N.

In general, the probability to have k stones into one box follows the binomial distribution. With $p = \frac{1}{M}$, we state

$$p(k) = \binom{N}{k} p^k (1-p)^{N-k}.$$

In N binomial k, we select all paths in the tree with k stones inside the box and we assign to each sub path the correct probability depending if it was a success or a failure. In short, we have a $B(n,p)$ probability distribution for each box of our problem where $n = N$ and $p = \frac{1}{M}$. $E[B(n,p)]$ is $\frac{N}{M}$.

To compute the standard deviation, one can choose a Poisson distribution as approximation for $B(n,p)$. This is because n is big (N) and p is small $(\frac{1}{M})$. For the Poisson distribution, λ is then $\lambda = nP = \frac{N}{M}$. The variance of the

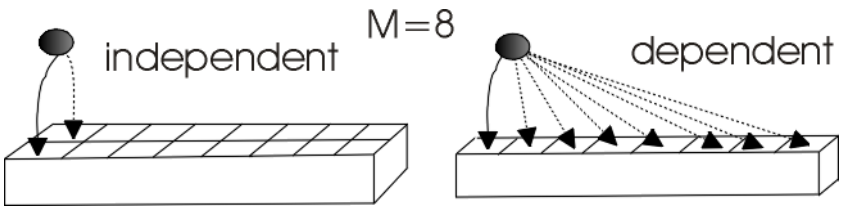

Figure 5: Dependence versus independence

Poisson distribution is $\lambda = \frac{N}{M}$, The standard deviation for Poisson is therefore $\sqrt{(\frac{N}{M})}$. Alternatively, one can compute σ for $B(n,p)$ using $Var(k) = E(k^2) - E(k)^2$ and a differentiation trick. For large Ms, standard deviation and variance are the same for both distributions, $N(1-\frac{1}{M})\frac{1}{M}$ for Binomial and $\frac{N}{M}$ for Poisson.

Here, it is important to notice the following: while $p_{none} = (\frac{M-1}{M})^N$ is the probability that one particular box is empty, it is not possible to generalize with $(1-p_{none})^M$ to the problem where all boxes have at least one stone; the boxes are not independent from each other: a stone that does not land into a box does not fall apart but lands in another box, as in Figure 5.

4.2.2 Problem solved with a recurrence

Let $A(N,M)$ be the number of arrangements leaving *none of the M bins empty* [28]. We imagine adding an additional bin. This bin contains k stones ($1 \leq k \leq N$) but not 0, so that we can express the number of arrangements in the other bins with $A(N-k,M)$. Therefore, the number of arrangements leaving *none of the* $M+1$ *bins empty* satisfies a recurrence,

$$A(N,M+1) = \sum_{k=1}^{N} \binom{N}{k} A(N-k,M).$$

The solution to the recurrence is given by:

$$A(N,M) = \sum_{\nu=0}^{M} (-1)^{\nu} \binom{M}{\nu} (M-\nu)^N.$$

To prove it by induction, one can plug the solution into the recurrence formula. It is necessary to change the order of summation and use the binomial formula to express $A(N,M+1)$ as the difference of two simple sums [28].

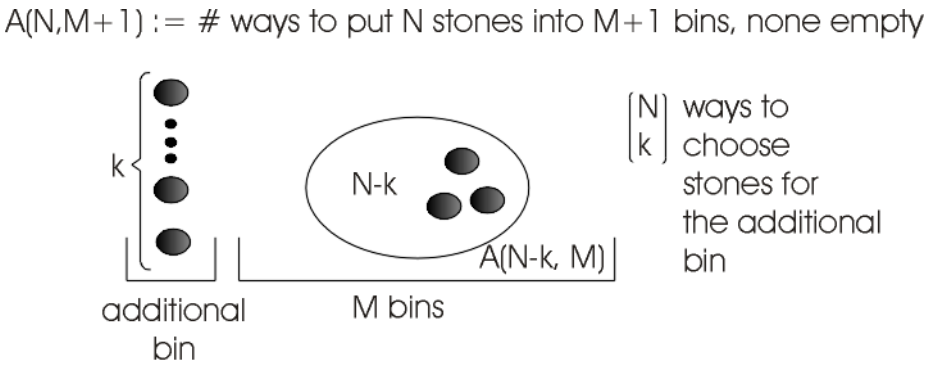

Figure 6: Composing the recurrence formula

4.2.3 Problem from a different perspective

We follow [28] here. We first change the way we look at the problem; we imagine now to have the N stones in a line, and to assign to each of them a number between 1 and M - this number is the box the stone falls in. There are M^N possible distributions, each with probability $\frac{1}{M^N} = M^{-N}$. We seek the probability $p_0(N, M)$ of finding all cells occupied.

Let A_k be the event that cell number k is empty ($k = 1, 2, \cdots, n$). In this situation all N stones are placed in the remaining $(M-1)$ cells, and this can be done in $(M-1)^N$ different ways. Similarly, there are $(M-2)^N$ arrangements, leaving two preassigned cells empty, etc. Accordingly and with $\frac{(M-1)^N}{M^N} = (1 - \frac{1}{M})^N$:

$$p_i = (1 - \frac{1}{M})^N, p_{ij} = (1 - \frac{2}{M})^N, p_{ijk} = (1 - \frac{3}{M})^N, \cdots$$

and hence for every $1 \leq \nu \leq M$

$$S_\nu = \binom{M}{\nu}(1 - \frac{\nu}{M})^N.$$

S_ν is the sum of probabilities, where each arrangement with ν cells empty appears once and only once.
We use now an important theorem proved in [28, pg 99]:

Theorem *The probability P_1 of the realization of at least one among the events $A_1, A_2, \cdots, A_M$ is given by an alternating sum*

$$P_1 = S_1 - S_2 + S_3 - S4 + - \cdots \pm S_M.$$

Hence, the *the probability that all cells are occupied* is

$$p_0(N,M) = 1 - S_1 + S_2 - + \cdots = \sum_{\nu=0}^{M} (-1)^{\nu} \binom{M}{\nu} (1 - \frac{\nu}{M})^N.$$

This probability is the solution to the recurrence formula that gave the number of arrangements with no cells empty divided by the total number of arrangements M^N.

4.2.4 Approximation with a Poisson distribution

It is clear that a direct numerical evaluation of p_0 is limited to the case of relatively small M and N. According to [28], it is possible to estimate p_m, *the probability to have exactly m cells empty*, with a Poisson distribution.

$$\lambda = Me^{-\frac{N}{M}}$$
$$p_m(N,M) = e^{-\lambda}\frac{\lambda^m}{m!}$$

In our case, we use $m = 0$ to get all cells occupied:

$$p_0(N,M) = e^{-\lambda}$$

4.2.5 Simulation with a plugin

Using the plugin `stattest.dll`, we estimate our problem with a simulation.

We computed for each point in the graph the N stones to M boxes problem 100000 times. Each time, we looked if we were able to cover the domain M with at least one stone and we averaged to get the $p_{!0}(N,M)$ probability.

Syntax for the plugin is:

```
[M],[N],[number of attempts],throwNstonestoMboxes
```

The plugin gives two probabilities back:

```
[probability that all cells are occupied],
[1-(average coverage probability)]
```

The first probability is $p_0(N,M)$. The numbers fit our formula and the approximation with $e^{-\lambda}$.

In the second probability, we looked at what percentage of the domain was not covered with at least one stone through all 100000 realizations of the experiment. The graphs shows that the uncovered domain decreases independently from the size M of our problem, in respect to $\frac{N}{M}$.

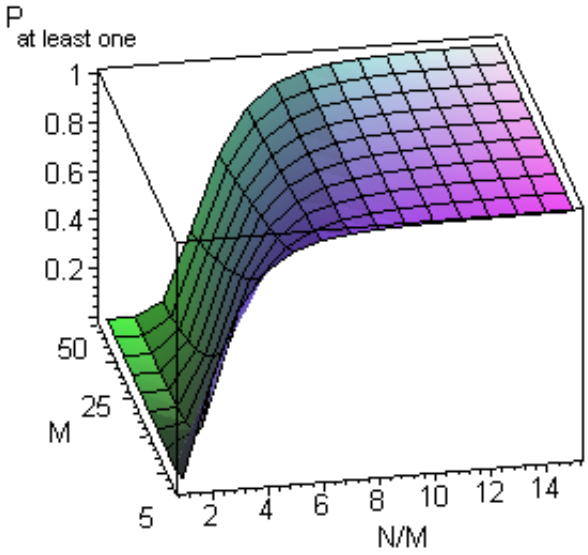

Figure 7: Probability $p_0(N, M)$ to cover entire domain with at least one stone

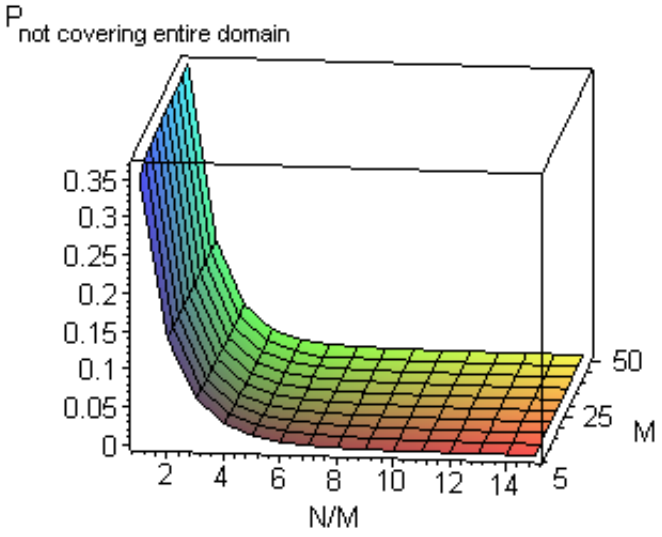

Figure 8: Average failure probability in respect to coverage

4.2.6 Conclusion to the occupancy problem

While GPU is not suitable for tasks where the domain M has to be covered entirely, it could work for tasks where the analysis of big parts of it suffices.

Having multiple answers to the same request is not that bad: available computing power is amazing. In November 2003, `Seti@home` crunched all available data and is now reprocessing data again; the telescope does not record enough data, although a minute of sky observation turns into 10 hours of Fast Fourier transform analysis.

Additionally, people might compute wrong results intentionally, but their efforts will disappear in random noise due to the nature of Monte Carlo computations, because results will be computed multiple times by different users.

4.3 Small world problem estimated with a fractal argument

`Limewire`, a popular Gnutella client, estimates the network size each day with a Gnutella Crawler[23]. In the early days of Gnutella, there were 500000 nodes. Now, Gnutella feels competition of other popular file sharing networks like `eMule` and `Kazaa`; network size dropped down to about 100000 (March 2004).

An interesting problem arises: how many users are reachable by one user in the Gnutella network? How should one set the count-down timer in each packet to reach all nodes? The problem is equivalent to the following one: how many people are between any two people or between you and the President of the United Nations, as an example? This is sometimes referred as the small world problem, a common experience of many of us: we meet a new friend and we discover he/she knows someone we know as well. Milgram's experiment showed empirically that any two people in the United States are distant six edges from each other.

To solve the problem, we attempt a volumetric argument. Some fractal arguments are given in [26], although they go by far more in depth. However, we start our argument with an analogic machine to compute the solution. Results are consistent with Mandelbrot's fractal theory [24,25].

Stating the problem Given a random graph G with N nodes, $G(N, E)$. The average number of connections per node[6] is M. In other words, each node has M connections to other nodes. Take randomly two nodes out of the graph: what is the average path length between these two nodes? To compute the average path length, we always choose the shortest distance[7].

4.3.1 Analogic machine to compute solution

Imagine a table with a random graph on it. Unlike normal graph theory, we force all edges to the same length. The reason for this will soon be clear; we take the node that represents ourself and elevate it over the table, the rest of the graph falls down in disorderly fashion. Equipped with scissors, we cut edges that go back to upper levels of the graph, until our graph turns into a tree; as soon as all double edges are cut, the entire tree falls down like a line thanks to the force of gravity. Levels in the tree are all of the same length, thanks to our initial constraint.

[6] the degree of a node

[7] In Gnutella, duplicates are thrown away: packets travel through the shortest path

We take the tree and put it level by level into a bin. We count how many levels we put into the bin until we reach the presidential node; this is then the path length between us and the President. Unfortunately, depending on how we cut the graph, the path length might change a lot.

Notice that because we force the same edge length on all edges, a random graph might not fit on the table, but partially elevate into air. And even with 3 dimensions, there will be random graphs that we cannot construct, if they have lot of edges and if these edges cannot stretch. Nonetheless, these graphs exist in high-dimensional spaces we cannot easily imagine, with $d > 3$.

This is strong evidence for a dimensional path to the solution.

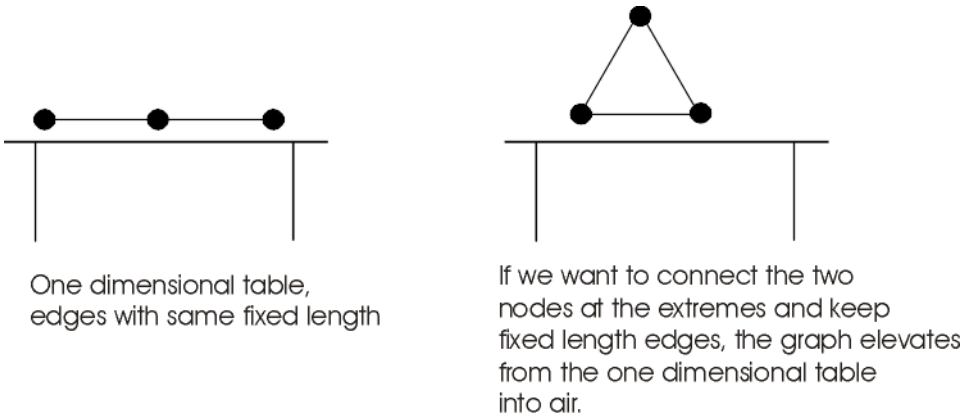

Figure 9: Forcing fixed length edges on a simplified one dimensional table.

4.3.2 Reducing the problem to a 2D-volumetric argument

Given a random subgraph $G_{sub}(n \subseteq N, e \subseteq E)$ of $G(N, E)$. How many edges connect G_{sub} to $G \setminus G_{sub}$? We imagine G_{sub} and G really big graphs embedded in the plane; imagine coloring red nodes belonging to G_{sub} and yellow for nodes in $G \setminus G_{sub}$. Now we look at them from far away: nodes and edges look like a tiny lattice and eventually disappear into two uniform areas, one red and one yellow. Assume we set $n = A \propto r^2$ where A is a graph's area. The perimeter of the graph's area is then proportional to r, and the number of outgoing connections proportional to $r \propto \sqrt{n}$, too.

It is important to note here that the form of the subgraph changes area up to a constant ($A_1 = \pi r^2$ for a circle and $A_2 = 4r^2$ for a square) but does not change the radius dimensionality.

Assume we could solve the problem stated in this paragraph: we could then sum up e_i until we reach N. In the analogic mechanism, e_i is the number

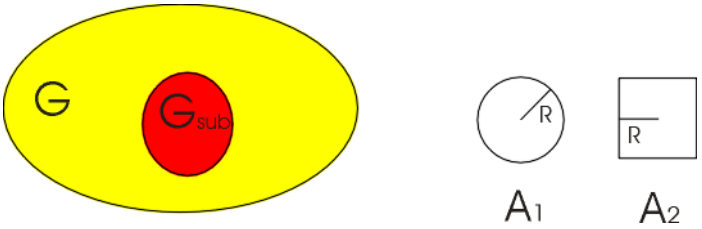

Figure 10: G_{sub} and G seen from far away

of nodes added to each level of the tree to the bin.

$$\sum_{1}^{l} e_i = N$$

with e_i: number of outgoing edges for graph with i nodes and l maximum path length.

4.3.3 Volumetric argument in more dimensions

If we decide that our lattice is not constrained to planes, then we propose the following nodes with constant degree M as building blocks.

Building block					
M	0	1	2	4	6
d	0	0<i<1	1	2	3

Figure 11: Building blocks, lattice where we embed G and G_{sub}.

We chose building blocks so that they generate homogeneous grids and with fixed length edges. Both constraints are necessary for the volumetric argument we want to use. Homogeneity is similar to the requirement for planar graphs in normal graph theory that edges do not cross. Edge-crossing would generate inhomogeneities with fixed length edges on 2D grids.

If M is even, we naturally choose edges perpendicular to each other. Two counter-directed edges of the building block form an independent axis in the

d dimensional axis. Therefore, we set

$$d = \frac{M}{2}$$

If M is odd, we might looked puzzled at the fractional dimension, a sort of in between dimension.

Furthermore, we know that the boundary of a subgraph has a dimension less (for a sphere $V = \frac{4\pi}{3}r^3 >$, but $A_{surface} = 4\pi r^2$). In general, if the subgraph is d-dimensional, the volume is proportional to r^d and the boundary to r^{d-1}. Size of the random subgraph G_{sub} with n nodes is then:

$$V_{sub} = n \propto r^d.$$

The number of outgoing edges is proportional to the size of the boundary, r^{d-1} namely.

$$e_n \propto r^{d-1}$$

We now start in the middle of our subgraph, and we subsequently add nodes until we reach the boundary. In the analogous mechanism of adding nodes means putting them into the bin, i, representing the height of the tree and the boundary is reached until all nodes are into the bin.

$$\sum_1^l i^{d-1} \propto N$$

We imagine a tiny lattice of nodes and we therefore turn summation into an integral:

$$\sum_1^l i^{d-1} = \int_0^l i^{d-1} di = \frac{i^d}{d} \,|_0^l \propto N$$

Thus:

$$\frac{l^d}{d} \propto N$$

and:

$$N \propto \frac{l^{\frac{M}{2}}}{\frac{M}{2}}$$

Solving for l gives:

$$l \propto (N\frac{M}{2})^{\frac{2}{M}}.$$

Solving for M requires a numerical approximation.

Resuming, we consider G_{sub} embedded in G, both in a homogeneous d-dimensional grid composed by myriads of the above mentioned building blocks. We let G_{sub} grow until it reaches the same size of G through summation and we infer l.

Results are very similar to the Mandelbrot's formula

$$N \propto l^d$$

that relates mass (N) and linear extension l with the exponent d [25].

4.3.4 Applying the formula

A homogeneous Gnutella is a fractal with dimension $d = 7.79$ according to the following table that lists some related problems estimated with our formula, in bold text we highlight the computed result, the other numbers are assumptions:

Description	N	M	d	l
Gnutella	500000	**15.58**	**7.79**	7 (TTL)
Milgram's experiment	$250 \cdot 10^6$	**24.38**	**12.19**	6
World	$6 \cdot 10^9$	24.38	12.19	**7.78**
Switzerland	$7 \cdot 10^6$	24.38	12.19	**4.47**
Poschiavo	3500	24.38	12.19	**2.39**
CPU	$3 \cdot 10^6$	3	1.5	**27257**
Brain	$100 \cdot 10^9$	10	5	**218.67**

For Gnutella, we know l as the standard TTL[8] and the number of nodes computed by the Gnutella crawler[23]. Using the formula, we compute the dimension d and the average number of outgoing connections M.

Milgram's experiment showed that in the United States, a country with about 250 million (N) inhabitants, there are 6 (l) degrees of separation. The formula estimates that each person knows about 25 people enough well to perform the experiment.

Using Milgram's M, we compute l for the entire world, for Switzerland and for a little village in the mountains.

We try the formula on the brain, a complex network with 100 billion neurons. Each neuron has about 15000 connections but is connected to a neighborhood of about 10 other neurons only, about 1500 connections for each neuron. The path length would be then 218. We idealized the CPU

[8] see also the Benchmark section for the line topology

as it would be composed by 3 million NAND ports, with 2 inputs and one output.

However, Gnutella and the above mentioned problems are far from homogeneous in the degree of the nodes, so that our formula gives only a rough estimation.

4.3.5 Diffusion effect

In [3], we compared Gnutella and the alternative routing algorithm with random walkers to a diffusion experiment. Following Mandelbrot, we could imagine the diffusion of Gnutella packets in the $d = 7.79$ dimensional spatial lattice. In a physics analogy, we could compare diffusion speed of one node per unit of time to the speed of light in Euclidean space. Finally, if we would relax the fixed length constraint on our edges (edge length could be set at the temporal distance between two Gnutella nodes), we would then embed Gnutella in a curved hyperspace!

4.4 The centralized model as subset of GPU

How do we turn GPU back to the centralized model? Viewed that GPUs all receive the same string as a parameter, we cannot transmit different information on which data to process to each GPU.

A natural workaround would be to implement a web server with a `.php` script that sends different data each time. To avoid all GPUs accessing the web server at the same time, plugins should distribute connection attempts with exponential waiting. The entire Gnutella mechanism becomes a trigger to start the centralized computation.

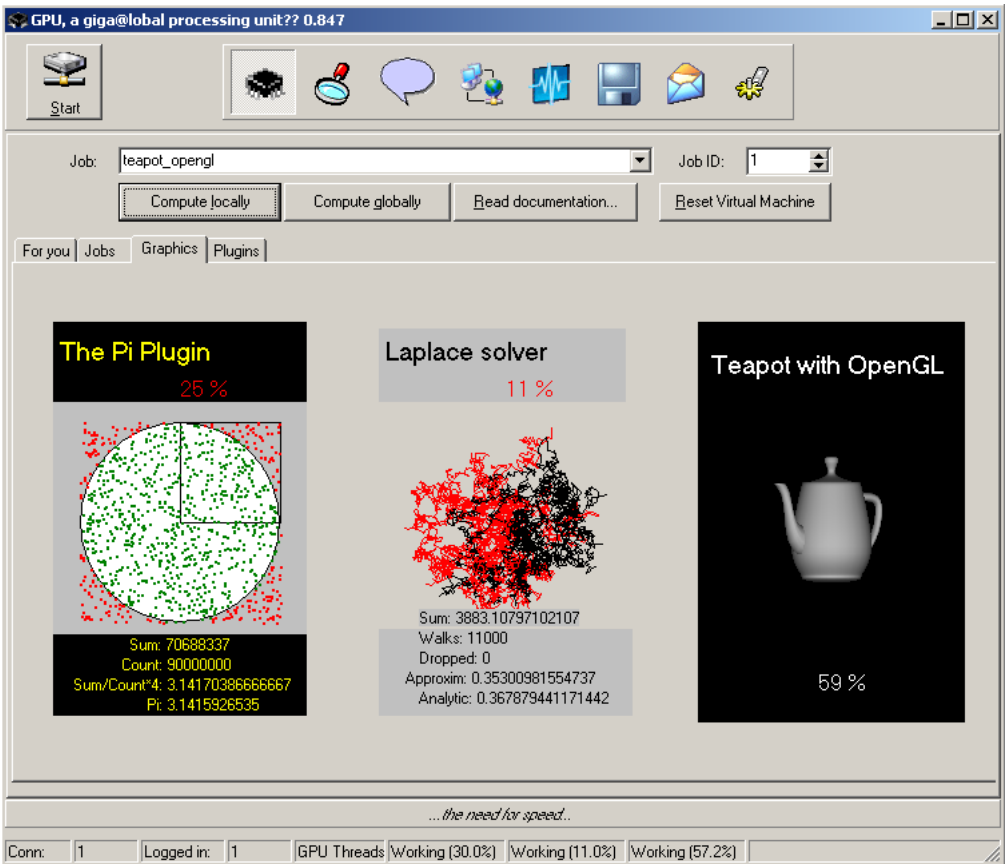

Figure 12: Three threads are computing with graphics output.

5 How to implement a plugin with graphics output

We gave a detailed overview on how to implement plugins in [3], with some additional explanations to help C-developers to see some Delphi-C differences. We briefly repeat some basics here.

5.1 Introduction

Dynamic link libraries (in Windows) or *shared objects* (in Linux) are collections of methods loaded and linked at runtime. "Dynamic Linking" means that the memory address of the function is known only at runtime, and not previously at compilation time (as in *static libraries*).

GPU loads at runtime all *dynamic link libraries* in the subdirectory

`plugins` of the application. They do not necessarily have to be implemented in Delphi and could be implemented in C as well.

In order to be called by the virtual machine, a method has to have the following signature:

```
function add(var stk : TStack) : Boolean; stdcall;
```

`stk` is a variable passed by reference. In C-terminology, it is a pointer to a `struct` that contains an array of `double` (extended-precision floating points) and an index value that points to one of the array's elements. The function can modify this array and the index pointer as well, although typical functions like `add` simply add the two last parameters and push the result onto the stack. Finally, `add` returns `true` if the computation was successful; if there are not enough parameters, `add` returns `false`.

5.2 Graphics output

All centralized frameworks offer a way to graphically display partial results of the ongoing computation. Also to debug, it is important to have some graphics. However, a much higher computational speed could be achieved, probably at least one order of magnitude, if graphics were disabled. For this reason, `Seti@Home` [10] provides one graphical, and one non-graphical version of the client. In this section, we will see how GPU tries to minimize the problem.

We explored several possibilities of adding graphics to the ongoing computation. Some constraints of Delphi/Kylix limited our choice: in particular, graphical output cannot be displayed at any time, but must be synchronized with *VCL*, the *Visual Component Library* assigned to manage graphics and control behavior in Delphi.

A simple example, the `pi` plugin extended with graphics output, can be found in `/dllbuilding/pi`.

To provide graphics output, a plugin implements an additional method in the dynamic link library:

```
function update(var stk : TStack):Boolean; stdcall;
```

For each plugin, there can be only one update method. Update receives the same `stk` parameter described before. However, the `struct stk` contains now additional fields that were not present at the time [3] was written.

`TStack`, the `struct` definition is defined by:

```
type TStack = record
  Stack           : Array [1..MAXSTACK] of Extended;
  StIdx           : LongInt; { Index on Stack where
                               Operations take place. StIdx
                               cannot be more than MAXSTACK
                             }
  {new fields}
  Progress        : Real;      { indicates plugin progress
                                 from 0 to 100
                               }
  My              : Pointer;   { used to store data passed
                                 between update and
                                 function itself
                               }
  {$IFDEF MSWINDOWS}
  hw              : HWND;      { handle to the window where
                                 graphics are displayed
                               }
  {$ENDIF}

  Update          : Boolean;   { function desires an update
                               }

  {$IFDEF STRINGS}             { future extension }
  StrStack        : Array [1..MAXSTRINGS] of Strings;
  StrIdx          : LongInt;
  {$ENDIF}
end;
```

Besides the earlier described `Stack` and `StIdx` [3] we have new fields.

If a plugin wants graphical output, it needs a sort of *cooperative multithreading*. A plugin should release control by exiting the function from time to time while slowly increasing the field `Progress`. In between, the update function is called. Once `Progress` reaches `100`, the virtual machine [3] considers the plugin finished and is ready to continue the stack evaluation. For backward compatibility reasons, if `Progress` remains `0` after the first call, the plugin is considered finished. Additionally, a graphics plugin will set `stk.Update` to `true`, to signal that the function wants an update.

We take a closer look at the internals, to understand better what happens; if GPU is idle and gets a new incoming job, it creates a `ComputationThread` (in `ComputationThread.pas`) that is executed via the following main loop

(simplified):

```
Stk.Update   := False;
Stk.Progress := 0;
repeat
  Found := PlugMan.ExecFuncInPlugins(FunctionName,stk,...);
  if (UpdateNeeded and Stk.Update) then Synchronize(Update);
until ((Stk.Progress=0) or (Stk.Progress=100) or Terminated);
```

`PlugMan` (in `PluginManager.pas`) deals with plugins. Method `ExecFuncInPlugins` searches through all available plugins for a method called `FunctionName`. Once found, it calls `FunctionName` and passes `stk` as parameter.

`UpdateNeeded` is true only if the user is in the window, where plugins display graphics output. If GPU is minimized, or the volunteer is using other functionality of GPU (like the chat system), `UpdateNeeded` stays false and there is no need to synchronize with the VCL; in particular, we do not waste computer cycles to display graphics on a hidden component. In this way, we do not need to provide two separate plugins, one graphical and one non-graphical.

`Terminated` is set if the thread is forced to exit, if the user closes GPU, or if he wants to reset the virtual machine.

5.3 Drawing on the window

Field `hw` is enclosed in a Delphi compiler directive. This means for Linux, graphics output is not available yet. Field `hw` is a handle to the little window where graphics are displayed, as in Figure 12. This handle has to be initialized and freed. Using `hw`, a `Canvas` object has to be instantiated and freed. Object `Canvas` provides then lot of painting methods, like `LineTo` and `Ellipse`. Please refer to source code of the `pi.dll` plugin for more details. An even more advanced plugin with 3D graphics implemented in OpenGL [21] is `teapot.dll`.

5.4 Passing data between update and plugin function

The last topic we discuss is how to pass computational data from the plugin function to the update function. Two methods are quite inelegant: it is possible to write a file, or to pass data on the stack.

The best solution, however, is to use the `my` pointer in `TStack` to create a struct containing data of interest for the update function.

Assume, we want to pass a list of points. First, one should define a data structure to be passed:

```
type TMyData = record
      xP                : Array[1..MAX_POINTS] of Real;
      yP                : Array[1..MAX_POINTS] of Real;
end;
```

In the plugin method, at the beginning of the computation, we reserve memory for this structure, then we store the pointer in `my` and initialize it.

```
  if stk.Progress = 0 then
           begin
           GetMem(Stk.My, SizeOf(TMyData));
           with TMyData(Stk.My^) do
               begin
               for i:=1 to MAX_POINTS do
                      begin
                      xP[i]:=0;
                      yP[i]:=0;
                      end;
               end; {with}
            end;
```

In the update function, we free the reserved memory at the end of the computation with

```
if Stk.Progress = 100 then FreeMem(Stk.My, SizeOf(TMyData));
```

To access the first element of the `xP` array in both function and update method we typecast the `my` pointer to the `TMyData` structure before accessing the field `xP`:

```
 TMyData(Stk.My^).xP[1];
```

5.5 Future extensions

The last two fields, `StrStack` and `StrIdx` enclosed in the disabled `$Strings` directive, will allow plugins to get strings as parameters as well. This will be crucial if a plugin needs an URL to download data, for example. At time of writing (February 2004), this feature is not implemented in the framework.

5.6 Example: Feynman Kac plugin

5.6.1 Seed management

An important fundament of science is to guarantee reproducibility of results. However, the Gnutella system does not allow providing nodes with different initial parameters; all packets addressed to the nodes contain the same command string. Therefore, it is not possible to pass a different, initial seed for the pseudo-random generator [19] to the nodes. Assuming it would be possible to pass different initial seeds to the nodes, reproducibility of results would be ensured by passing the same seeds to the same nodes each time.

GPU solves the issue as follows: each time a new GPU is installed and executed for the first time, the standard Delphi generator is seeded with the clock (with millisecond resolution) and date. Using this random generator, a seed file with 256 words is generated and stored in a file, once. Each new GPU gets in this way another seed file with high probability, unless they are installed on the same millisecond.

ISAAC[19] is a powerful pseudo-random generator implemented in `isaacrnd.dll` by Sebastian Sauvage. ISAAC loads the 256 words previously stored by the standard Delphi-generator to initialize itself. From then on, each node will provide a different sequence of pseudorandom numbers. However, if we restart all GPUs, each machine will regenerate the same sequence.

Reproducibility of results on one machine, not connected to other machines, is therefore guaranteed.

Ensuring reproducibility of results in GPU while running on a network requires more work: we have to ensure all GPUs are started afresh; we have to know the seed file for each node and we have to connect GPUs with each other in the same manner we did before. Consequently, reproducibility of results is theoretically possible, although different operating system load might introduce an additional uncertainty in how jobs spread in the network.

5.6.2 Feynman-Kac plugin description

This plugin solves Exercise 2.2 of the "Introduction to Parallel Computing" book described at page 82 [1].

We solve a partial differential equation inside an elliptical region by Monte Carlo simulations of the Feynman-Kac formula. The following partial differential equation is defined in a three-dimensional ellipsoid E:

$$\frac{1}{2}\triangle u(x,y,z) - v(x,y,z)u(x,y,z) = 0,$$

where the ellipsoidal domain is

$$D = \left\{ (x,y,z) \in \mathbf{R}^3; \frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} < 1 \right\}.$$

The potential is

$$v(x,y,z) = 2(\frac{x^2}{a^4} + \frac{y^2}{b^4} + \frac{z^2}{c^4} + \frac{1}{a^2} + \frac{1}{b^2} + \frac{1}{c^2}).$$

Our Dirichlet boundary condition is $u = g(\mathbf{x}) = 1$ when $\mathbf{x} \in \partial D$.

The goal is to solve this boundary value problem at $\mathbf{x} = (x,y,z)$ by a Monte Carlo simulation. At the heart of the simulation lies the Feynman-Kac formula, which in our case ($g = 1$) is

$$\begin{aligned} u(x,y,z) &= \mathbf{E}\, g(\mathbf{X}(\tau))\, exp\left(-\int_0^\tau v(\mathbf{X}(s))ds \right) \\ u(x,y,z) &= \mathbf{E}\, exp\left(-\int_0^\tau v(\mathbf{X}(s))ds \right) \\ u(x,y,z) &= \mathbf{E}\, Y(\tau). \end{aligned}$$

which describes the solution u in terms of an expectation value of a stochastic process Y whose initial value $Y(0) = 1$. Here $\mathbf{X}(s)$ is a Brownian motion starting from $\mathbf{X}(0) = \mathbf{x}$ and τ is its exit time from D.

It is important to point out that the $\mathbf{E}$ operator in simulation means $\mathbf{E}f = \frac{1}{N}\sum_{i=1}^{N} f_i$ for samples of size N. The plugin `pde3d.dll` can be called in two different ways:

```
0,[a],[b],[c],feynmankac3d
1,[a],[b],[c],[initialx],[initialy],[initialz],feynmankac3d
```

The first call passes the ellipses axes as parameter. The function `feynmankac3d` stored in `pde3d.dll` generates first randomly a point $\mathbf{x}$ inside the ellipse. In the second call, we can specify the initial

$$\mathbf{x} = ([initialx],[initialy],[initialz]).$$

`0` or `1` inform the plugin on how many parameters are loaded on the stack.

From this initial interior point $\mathbf{X}(0) = \mathbf{x} \in D$ we integrate the following system of stochastic differential equations ($\mathbf{W}$ is a Brownian motion). We use N realizations of $\mathbf{X}(t)$ to integrate.

$$\begin{aligned} d\mathbf{X} &= d\mathbf{W} \\ dY &= -v(\mathbf{X}(t))Y dt \end{aligned}$$

For each realization, we integrate this set of equations until $\mathbf{X}(t)$ exits at time $t = \tau$ using the trapezoidal rule [1]. Finally, we compute $u(\mathbf{x}) = \mathbf{E}Y(\mathbf{X}(\tau))$ to get the solution.

To plot the error, we compare it to the exact analytical solution of the partial differential equation, computed by two differentiations:

$$u(x, y, z) = exp(\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} - 1).$$

The error in the plugin's graphical output is plotted as an oscillating green line around the y axis. After oscillating for a while, the error tends to converge to the y axis. Only one out of many random walks ($\mathbf{X}(t)$) is plotted with red or white colors.

Roughly speaking, we free a horde of random walkers from an initial point. These walkers diffuse while Y is integrated with rate $-v(\mathbf{X})$ on the potential v until they reach the boundary of the ellipse. All walks are averaged in a similar way as it is done with the *pi* plugin to get the function value at the chosen initial point. The diffusion process of the walkers, distributed as a trivariate normal density, weights points near the initial point more than points far away.

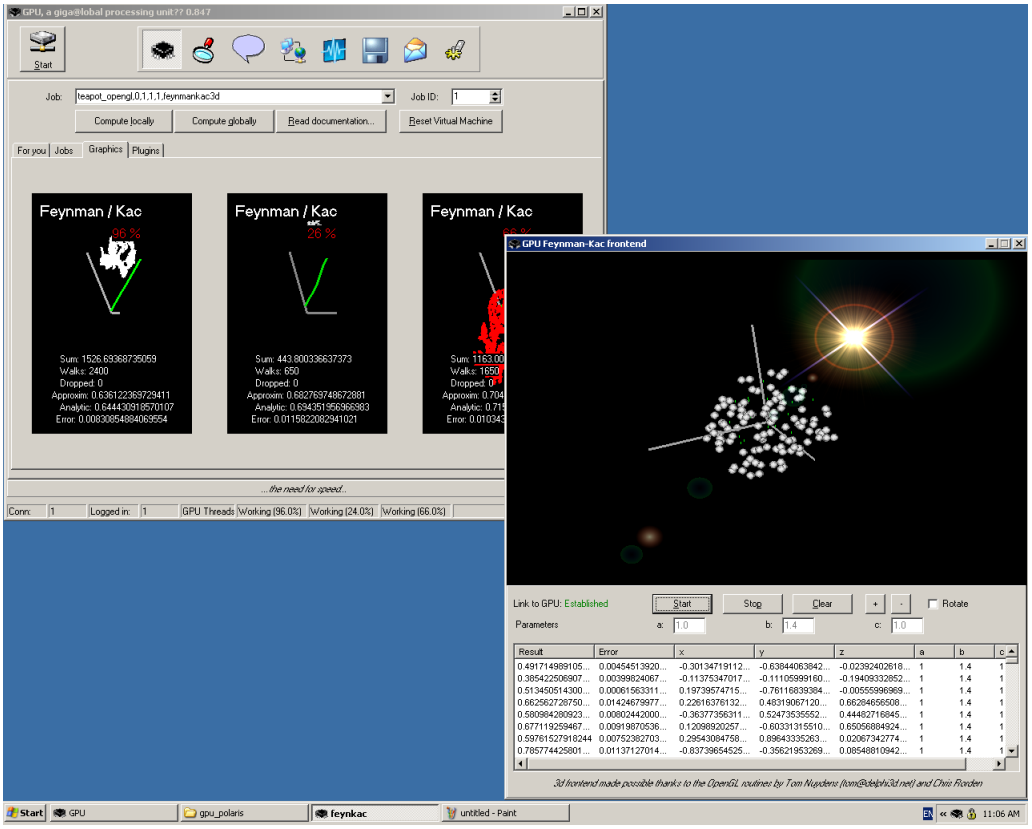

Figure 13: GPU computing the Feynman-Kac problem and corresponding frontend.

6 How to implement a frontend

6.1 Introduction

In this section, we explain in depth how to implement a new frontend for the framework. Frontends are normal applications that communicate to GPU through messages. They are stored in the `frontend` subdirectory of the GPU package.

Frontends send requests to the GPU application, and GPU spreads their requests through the Gnutella network. Once results came back, GPU notifies the frontend[9].

[9]The autoupdate routines are implemented as a frontend that sends a little shutdown message to GPU. Doing so, it is possible to overwrite the GPU executable without getting errors.

The following paragraphs come from "Getting the Message - Simple Techniques for Communicating Between Applications" by Robert Vivrette [6]. You can refer to the source code `simple_frontend.pas` in the `/frontend/simple` subdirectory of the GPU application as well.

Interprocess Communications is a broad description of any kind of communication between separate applications or processes. In particular, we will speak here about *Windows messaging* and *Memory mapped files*.

6.2 Windows Message

A *Windows message* is a record structure that Windows uses to communicate information around between controls, forms, windows, etc. The message structure first has a field called `HWnd` which is a handle to the window or control that the message is directed to. Next comes the `message` field which is the actual message type being sent. For a mouse click, the message type would be `WM_MOUSEDOWN`. Next come two fields for passing message-specific data along with the message. These are named `WParam` and `Lparam`, two longint values [6].

```
type
 TMsg = packed record
  hwnd: HWND;       // the handle of the Window for which the
                    // message is intended
  message: UINT;    // the message constant identifier
  wParam : WPARAM;  // 32 bits of message-specific information
  lParam : LPARAM;  // 32 bits of message-specific information
  time   : DWORD;   // the time that the message was created
  pt     : TPoint;  // Mouse cursor position when the message
end;                // was created
```

6.3 Message loop

All applications, if idle, are in a loop that checks if there are new *Windows messages*. By incoming messages, the loop calls the corresponding function. If we want to create our own inter-process message `WMsgGPUResult`, we will need to override the `WndProc` procedure called by the loop and listen in for the new message type we defined [6].

```
TForm1 = class(TForm)
    ...
  public
```

```
    procedure WndProc(var TheMsg: TMessage); override;
  end;

procedure TForm1.WndProc(Var TheMsg: TMessage);
begin
  if TheMsg.Msg = WMsgGPUResult then
    begin
      {It is our custom message - deal with it!}
    end;
  Inherited WndProc(TheMsg);
end;
```

6.4 How to define a new message type

To define a new message type, we need a constant that does not collide with something previously defined by the operating system or by other applications. In GPU, we call a Windows API function at startup that hashes a string to a constant.

```
procedure TForm1.FormCreate(Sender : TObject)
begin
 WMsgGPURequest := RegisterWindowMessage('GPURequest');
 WMsgGPUResult  := RegisterWindwoMessage('GPUResult');
end;
```

To send a message we can then call

```
SendMessage(GPUHWnd,WMsgGPURequest,0,0);
```

`GPUHWnd` is the Handle that identifies the GPU application. It can be retrieved using

```
GPUHWnd := FindWindow('TMainForm', nil);
```

6.5 Memory Mapped Files

We know now how to receive messages through the message loop and how to send them using `SendMessage(...)`. However, we can only send the little *Windows message record* defined in section 6.2. This is not enough, we would like to send our command string for the virtual machine [3].

Memory Mapped Files is a way of sending an arbitrary block of data between two cooperating applications. This is done by obtaining a handle

to a common piece of memory space from the operating system. When both applications have a handle to this space, they may write information back and forth to each other. The concept of *Memory Mapped Files* is actually very similar to creating a disk file and writing out the data while letting the other application open it up and read it.

We then define an `ObtainMappingHandle(...)` routine, that asks the operating system for memory. If the request for memory, done through `CreateFileMapping(..)` fails with `ERROR_ALREADY_EXISTS`, someone else reserved memory before. In this case, a call to `OpenFileMapping(...)` returns a handle to the common memory block. Since both applications will use the same string (`GPU_Comm_Space`), they will ultimately obtain a handle to the same memory mapped file [6].

```
function TForm1.ObtainMappingHandle: THandle;
begin
  Result := CreateFileMapping(\$FFFFFFFF,nil,PAGE_READWRITE,
                              0,2000, GPU_Comm_Space);
  if Result <> 0 then      // Did it return a valid handle?
                           // Did it already exist?
   if GetLastError \= ERROR_ALREADY_EXISTS then
      begin
        CloseHandle(Result); // Close this one - we will
                             // open existing one instead

        Result := OpenFileMapping(FILE_MAP_WRITE,False,
                                  GPU_Comm_Space);
      end;
end;
```

`GPU_Comm_Space` is a string constant. The `FileMappings` routines hash the value to get a memory block. All frontends and GPU hash the same `GPU_Comm_Space` string constant to the same integer value, so that they all share the same piece of memory.

6.5.1 Send a request to GPU

Now we have everything and can start putting pieces together. The packet string we send to GPU will have the following format: we specify first the name of our application (here `TForm1`) so that GPU can knows where to send the result back, then the Job ID and finally the GPU command (as in [3]).

We separate the three fields using ":".

```
Packet := 'TForm1'+':'+JobID+':'+ Command;
```

The first code line below ensures that we have a valid handle to the other application and also have a valid handle to our file mapping. Then we open up a view to the file with `MapViewOfFile`.

Think of creating the file mapping like using the `AssignFile` method on a disk file and the `MapViewOfFile` function like using textttReset or `Rewrite` on that file. The return result from `MapViewOfFile` though is an actual pointer to the memory space used by the file mapping. All we need to do now is put the string in it and let the other application know it is there. We first copy the string to the mapped area using `StrPLCopy` (which just copies the string up to a maximum number of characters).

Next we send a message to the other application telling it that a string is waiting for it in the Memory Mapped File. You will note that we do not actually pass any data in the `lParam` field of our "string" message. The message here is simply used to notify the other application that a string is available. After the message is sent, we close our view to the memory mapped file with `UnmapViewOfFile` [6].

```
procedure TForm1.SendJob(JobID, Command: String);
var
 ThePtr   : PChar;
 Packet   : String;
begin
 if (GPUHWnd = 0) or (GPUCommSpace = 0) then Exit;
 Packet := 'TForm1'+':'+JobID+':'+ Command;

 {send message}
 ThePtr := MapViewOfFile(GPUCommSpace, FILE_MAP_WRITE,0,0,0);
 StrPLCopy(ThePtr, Command, MaxMapLen);
 SendMessage(GPUHWnd, WMsgGPURequest, 0, 0);
 UnmapViewOfFile(ThePtr);
end;
```

6.5.2 Receive an answer from GPU

We go back to the `WndProc` method. Here the `WMsgGPUResult` case has been triggered by the incoming GPU message result. That tells us that GPU has placed a string in the Memory Mapped File and that it is waiting for us to

retrieve it. All we do is simply open up a view to the file with `MapViewOfFile` which returns a pointer to the start of the string. We split the string in two parts with `ExtractParamString(...)` in `utils.pas`. `JobID` and `ResultStr` contain the result of one computation. The only thing left to do is to close our view of the file with `UnmapViewOfFile` [6].

```
procedure TForm1.WndProc(Var TheMsg: TMessage);
var
 thePtr          : PChar;
 IncomingString,
 JobId,ResultStr : String;
begin
  {handling a user defined windows message that
   is a result}
  if TheMsg.Msg = WMsgGPUResult then
    begin
      ThePtr:=MapViewOfFile(GPUCommSpace,FILE_MAP_WRITE,0,0,0);
      IncomingString := ThePtr;
      UnmapViewOfFile(ThePtr);

      JobID := ExtractParam(IncomingString,':');
      ResultStr := IncomingString;
    end;
  Inherited WndProc(TheMsg);
end;
```

6.6 Example: Monitoring the GPU Network

A good framework for distributed computing should offer a way to monitor the status of the ongoing computations. GPU does it only in a primitive way, by applying the plugin-frontend idea once more.

The frontend sends a `gpustatus` command to GPU, which then spreads the command through the Gnutella network. All connected GPUs, even if they are busy[10] answer with status information; in particular, they send back the percentage of ongoing computations, the number of jobs waiting in queue and the other GPUs they are connected with. Further information like Gnutella up and download traffic, IP number, node name and country complete their report.

[10] the `gpustatus` request has high priority and is always answered even if a GPU is computing with all three available threads.

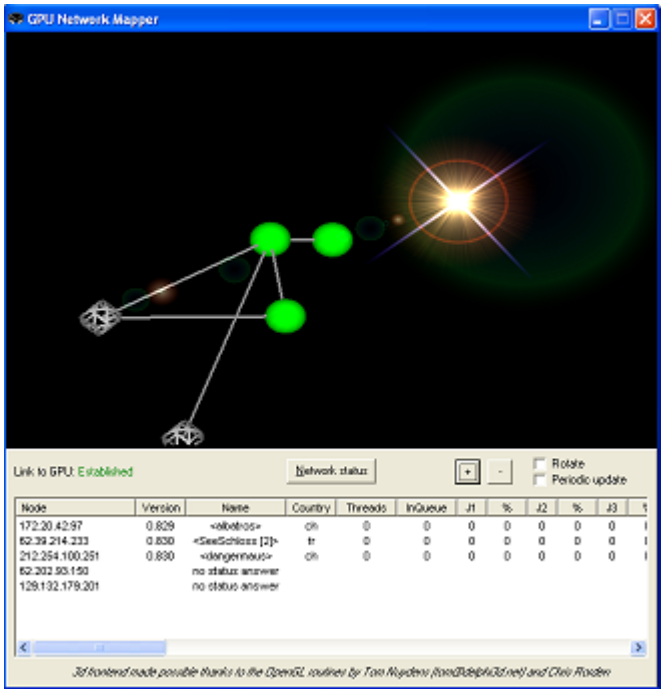

Figure 14: The frontend "Network Mapper"

All answers are stored by the frontend "Network Mapper" in a list. After a while (2-3 seconds) a simple algorithm is started on the data; from the IP address information, it builds a 3D graph[11] on how the computers are connected. Nodes are green solid spheres if they are idle or blue, violet and red depending on how many computations they are executing. Connections between nodes are plotted as lines.

For the moment, nodes are placed randomly in spheres that increase their radius according to their incoming ticket. Later, we might want to place nodes so that the length of connections reflect the distance between nodes[12] though it requires much more complex algorithms.

Gray wire spheres are plotted by the algorithm if other nodes reported the existence of a computer, although that computer did not explicitely answer our status request. That answer might got lost in a network congestion, because the node recently went down or because it is an older version that does not implement the status request[13] feature.

[11] a 2D graph would suffice for our purposes, but we wanted something nice to look at

[12] to be measured by a ping command, for example

[13] status request functionality is implemented since version 0.827

6.6.1 Known problems with NAT and dynamic IP

IP v6 is the protocol that might replace the current version 4 created to supply the increasing number of devices connected to the Internet. However, this technology might never be established because of a cheaper technology, called *NAT* (for *Network Address Translation*), that gives with one *IP v4* number Internet access to an entire subnetwork, like the ones that can be found in a student house or a small company. Using *NAT*, the old *IP v4* routers can work as before, while *IP v6* claims for a complete renewal of the Internet infrastructure.

NAT uses the infrequently used source ports numbered above 20000[14] to store additional information on the local computer that sends a packet. In TCP/IP, to answer a packet, the couple `([destination IP:Port][source IP:Port])` is simply inverted to build an answer. Therefore, the router will map particular ports ranges to computers on the local network.

A little disadvantage for GPU is that all users behind a *NAT* router surf with the same IP number and some information is lost on how these computers are interconnected.

Another disadvantage is caused by the dynamic IP that most ADSL providers offer: After a period of 20 hours, the IP number changes. This is a commercial trick to avoid having people provide web services or other server features with these cheap connections[15]. Every three hours, GPUs touch special websites to update the IP numbers they send in status request. Meanwhile, they might answer status request wrongly. In GPU, we call these nonexistent nodes that arise because of the dynamic IP problem "ghosts"; they are plotted as a wire sphere as well.

7 Benchmarks

We sent the `pi` job (`360'000'000,pi`) to clusters of different sizes, composed by Intel Pentium III 930 MHz with 512 MB of RAM and running Windows XP SP1. The job was sent through the "simple frontend" in `/frontend/simple`, so that the master computer was participating in the computation, too.

Jobs were computed in $\mu = 45.59$ seconds with a $\sigma = 2.87$. Variations were caused by random operating system load and GPU load. Tests with

[14] a port is a number between 0 and 65535, processes listen for incoming packets on particularly defined ports

[15] a remedy can be found at http://www.dyndns.org, that equips a server with dynamic DNS update

nodes outside the local network in the same city or abroad showed the same mean, but a higher sigma: the cluster was less homogeneous and there were faster and slower nodes.

To compute speedup, we simplify a little and plot the number of results received versus number of computers in a cluster.

7.1 Different topologies on local LAN network

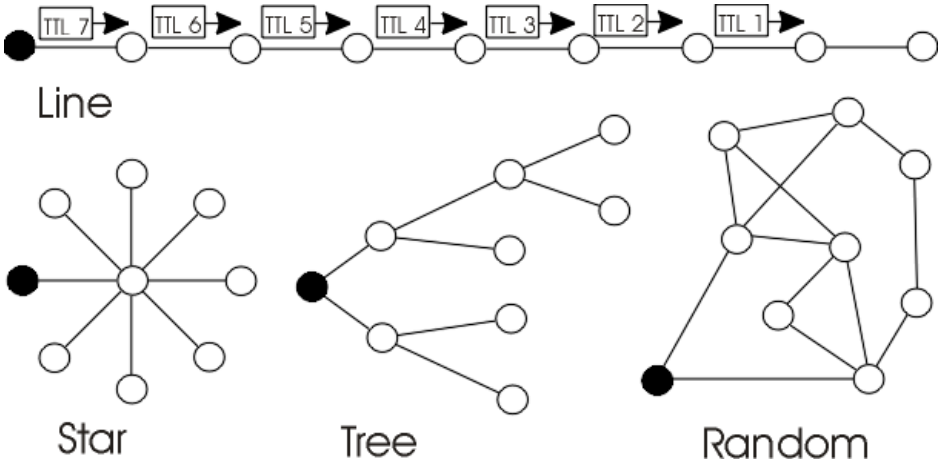

Figure 15: Topologies on Local Area Network: master node is black.

In the star topology, all computers mimic the centralized model. However, we used a branch computer as master. All computers received their job, but the central computer had trouble sending back results. Linear speedup was not achieved because the central computer was overloaded and lost some packets.

In the line topology, we have some packet losses for $n > 5$. For $n > 8$ speedup will not increase linearly anymore, but remains at the $n = 8$ level. In fact, the ninth computer did not receive the job; the packet carrying the job was discarded by the eighth GPU because its Time-To-Live counter reached zero. Notice that TTL is set to 7 in the GPU program.

For the tree topology, we get the best results: only one packet got lost once while running on 9 computers.

Results for the random graph topology show that GPU is still not ready to scale: in fact we found duplicates containing the same answer. This is the reason for the impossible super linear speedup. Therefore, we should fix the answer mechanism for version 0.847.

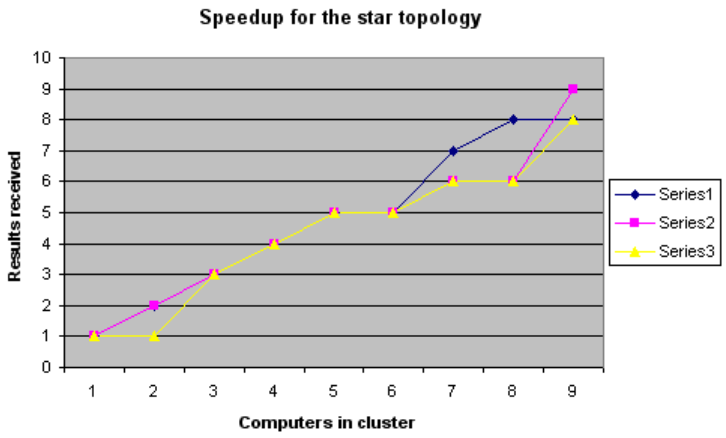

Figure 16: Speedup for the star topology

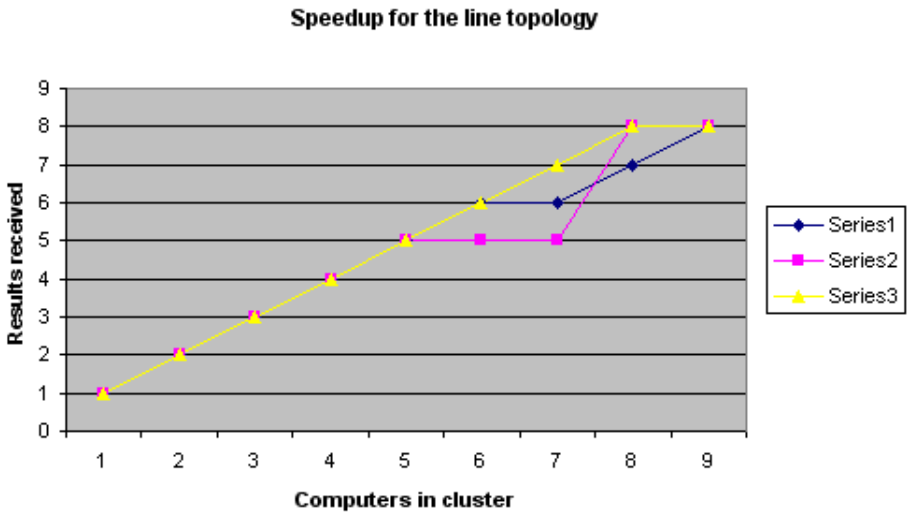

Figure 17: Speedup for the line topology

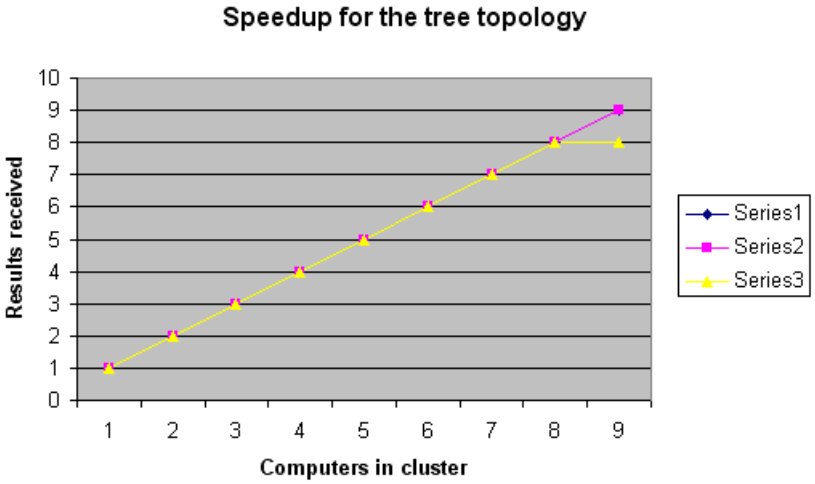

Figure 18: Speedup for the tree topology

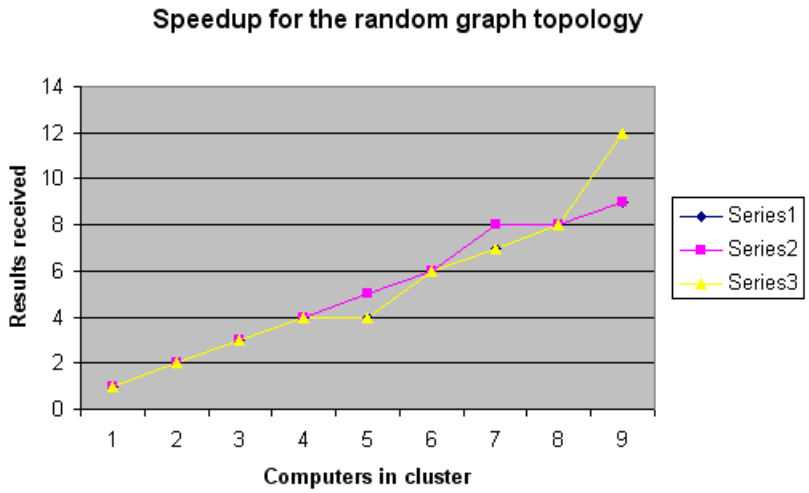

Figure 19: Speedup for the random graph topology

7.2 Big clusters

First test was run using GPU 0.80x at the end of October in the room HG E27. Results clearly show that GPUs in this version do not scale linearly: the number of received results is growing exponentially in one of the three series; in that series, we found one crashed GPU. In the other series, we occasionally had a number of results greater that the number of computers involved, this was also incorrect.

A second test at the beginning of January failed because of a router crash in IFW C31. As support people at ETH said, GPU was not involved in the crash, fortunately.

We run a last test with version 0.847. To reduce a little the duplicates problem, we chose a square grid where each node is connected to 4 neighbors. We subsequently added nodes, 21 were on a local network at ETH, Zürich, one in a student's house in the same city and one was in Russia from a user who occasionally connected to the network.

In this particular 2D-grid configuration, GPU scaled quite well; sometimes we had more results, sometimes less though there was no exponential growth of packets. GPU should scale up to a 8x8 grid with 64 computers. Larger grids are not possible because of the TTL constraint we saw in the line topology.

7.3 Permanent host

We ran a permanent host during the month of February 2004 in Poschiavo, Switzerland. GPU was running on an Internet Access Point hidden from the user's view. All GPUs around the world automatically connected to this node at startup; using the IP recorded in the logfile of the permanent host we were able to tell from which country the connection came from.

We first thought that TCP/IP timeouts might shape Gnutella in geographical clusters. However, TCP/IP proved itself very reliable and we recorded connections from everywhere, including countries like Brasil, Malaysia, Iran, China and Japan. Gnutella is probably very close to a random graph with no geographical clustering features.

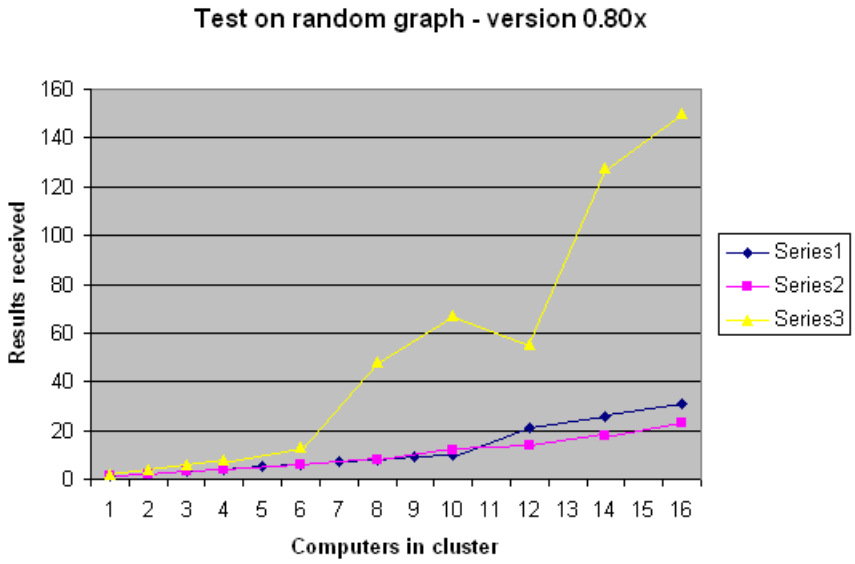

Figure 20: First big cluster on a random graph (January, version 0.80x).

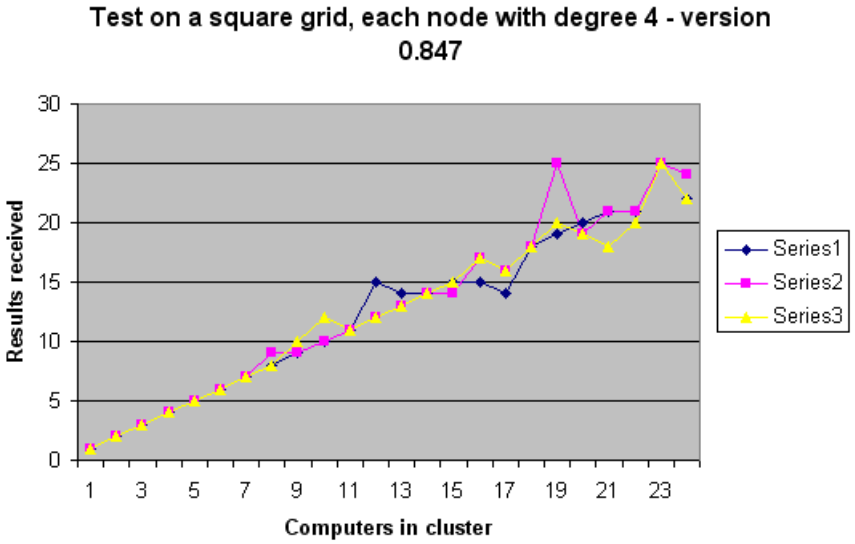

Figure 21: Second big cluster test on a square grid (March, version 0.847)

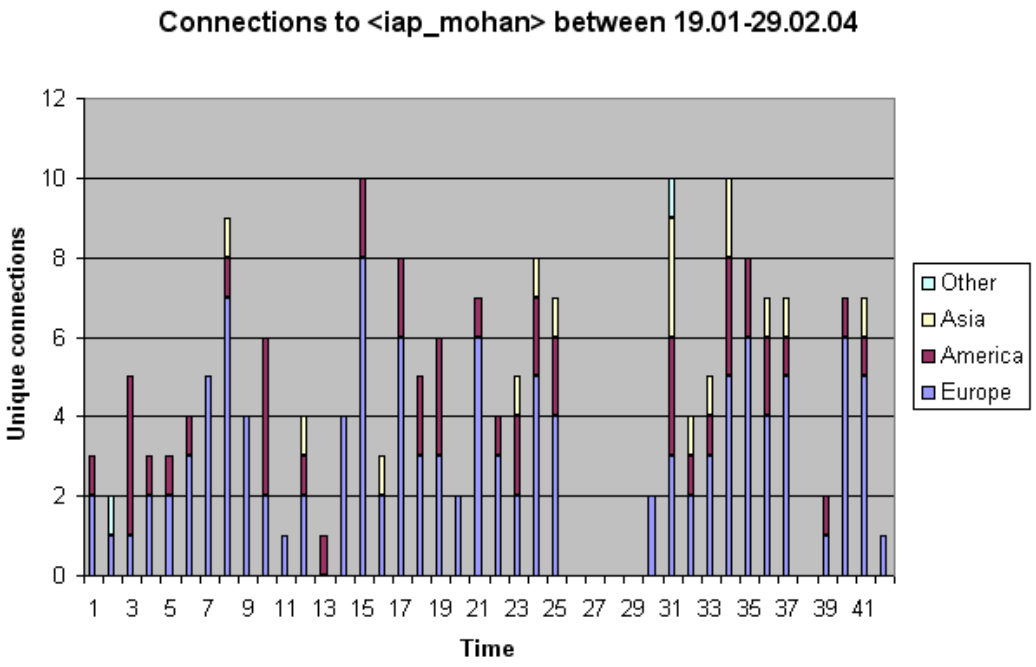

Figure 22: Connections from around the world to a host in Switzerland

8 Open Source development

We opened the source code of GPU at early stages of the project. In the beginning, we occasionally got criticism for the code's quality. In general, it was very difficult to involve other developers though many people helped with documentation, support and testing. Developers did not participate directly, but gave support for the many Open Source components employed in GPU.

`Sourceforge.net`, a platform that hosts around 70000 Open Source projects, gave to the project CVS repository, webspace, download mirrors, task manager and bug tracker for free. The statistics compiled each day gave a significant motivational boost: GPU was competing with professional research projects and established Open Source solutions for clustering.

`Sourceforge.net` permitted advertising the project through XML feeds and project's news; we reached a community of interested people, from the computer scientist to the computer's beginner, spread everywhere around the world.

We thank warmly `Sourceforge.net` staff for their commitment to the Open Source community.

8.1 Current project status and future work

We performed a lot of work to get a stable application and still have to invest time to eliminate bugs like the Gnutella duplicates problem. Since GPU was released while writing [3], we have made some progress.

We plan to invest more work in the following areas: we could give to GPU a cleaner protocol with priorities and dedicated Gnutella packets; we could invest more time in the Linux port of GPU; the core loop of `TComputationThread` could be optimized for speed; we will enhance GPU so that plugins get strings as parameters.

Users proposed several ideas for plugins and frontends: the existing Chess plugin could be rewritten for parallel execution; a DVD ripping farm; a Core-Wars execution environment; distributed databases and distributed crawlers could be interesting areas for further development.

Improvements like color maps for the Feynman-Kac frontend and a Netmapper with real distances and node's names complete the picture.

8.2 Conclusion

GPU shows with a prototype that distributed computing is feasible on a Peer-To-Peer network if we employ randomized algorithms. The plugin-frontend

paradigm with GPU as a middle layer arises because we want all users not only to compute but also to send computational requests. A *cooperative multithreading* is used to display graphics only if needed: if the plugin window is not visible, we save CPU-time.

As an example, we implemented a plugin-frontend couple that solves a partial differential equation on an elliptic domain with a random walkers approach [1].

According to benchmarks, GPU scales up to about 64 computers in a particular 2D-grid configuration where each node has degree 4. In a random topology, the duplicates problem is still present and should be fixed in order to achieve the normal Gnutella scaling.

Some theoretical considerations are attempted: in particular, we discuss the coupon collector problem [27,28] and we give an estimation for the small world problem using fractal theory [24,25,26].

As final note, we remark that in most cases, it is not the lack of resources that stop us from using the CPU-power of so many idle computers, but the difficulty to implement parallel algorithms that produce meaningful results. Computations have a strong serial nature, indeed.

8.3 Acknowledgments

I thank my family for the support in difficult times and Wesley Petersen who was my advisor through my studies. I thank all my friends from Poschiavo, Coira, Zürich and Sevelen for their help if there was need and for the happy hours in front of a steaming cup of coffee!

A list of contributors follows:

Developers Kamil Pogorzelski (TGnutella), Tom Nuyden (OpenGL port, flare), Chris Rorden (Arcball), Michael in der Wiesche (PGP support), Jan Verhoeven (multilingual component), Cap'n Bry (TGnutellaTrans),
Bob Jenkins (Isaac random generator), Sebastian Sauvage (Isaac port), Tempest Software (TTrayIcon), Jay (Linux plugins port), Abe Timmerman (huge numbers support), Tom Kerrigan (simple chess engine), Daniel Terhell (Chess GUI), Simon Wichtermann (Delphi fixes), Sayan Chakraboti (pi), Robert Vivrette (IPC)

Theory Wesley P. Petersen, Sid Nascimento da Silva, Stefan Kleiber Steifenauer, Fabrice Marchal, Kai Nagel, Livio Mengotti, Maurice Cochand, Ricardo Coreia

Website Nim van der Schee, Ayhan Koese, David A. Lucas, Mark Grady (logo)

Translators Cassandra Greer (Slideshow into English), Ranch Camal (Tamil), Jian Huang (Chinese), Muhammad Panji (Indonesian), Alexis Pigeon (French), Nim van der Schee (Slideshow into French)

Support Stefano Godenzi, Marino Crameri, Nurhan Cetin, Marcel Bauer, Flavio Crameri, Christian Wolfgang Hujer, Zillion Zhang, Barry W. Black, Phyrum Tea, Paul Peet, Atoma Gudissa, James Dixon, Fabio Zanetti, Zeki Ahmed Reshid, Vladislav Kharichev, Stabsoft ETH Support Team

Revisors Wesley P. Petersen, Aaron B. Hockley

Thank you for your precious help! The list reflects current contributors, new contributors will be listed in further documentation or on the website. If you found omissions or mistakes in this document, please mail them to `dangermaus@users.sourceforge.net`. This document is version 1.000.

8.4 Legal notice

9 References

[1] W. P. Petersen and P. Arbenz, *Introduction to Parallel Computing*, Oxford University Press, 2003.

[2] GPU Development Team, *The GPU project*, Sourceforge, 2002-2004, source code available from `http://gpu.sourceforge.net`.

[3] T. Mengotti, W. P. Petersen and P. Arbenz, *Distributed computing over Internet using a peer to peer network*, September 2002, available from `http://gpu.sourceforge.net`.

[4] *Wikipedia, The Free Encyclopedia*, Sourceforge, 2001-2004, `http://www.wikipedia.org`.

[5] M. A. Jovanovic, F. S. Annexstein, and K. A. Berman, *Scalability issues in large peer-to-peer networks- a case study of gnutella*, Technical Report, Univ. of Cincinnati, 2001, `http://www.ececs.uc.edu/~mjovanov/Research/paper.html`.

[6] R. Vivrette, *Getting the Message, Simple Techniques for Communicating Between Applications*, Delphi Informant, November 1997, `http://www.undu.com/Articles/991221b.html`.

[7] Godmar Back, *The RSA Algorithm and PGP*, 1996, `http://citeseer.nj.nec.com/back96rsa.html`.

[8] P. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann Publishers, San Francisco, 1997.

[9] Gnucleus Team, *Gnutella Web Caching System*, 2002, `http://www.gnucleus.com/gwebcache`.

[10] Seti@Home, *The Search for Extraterrestrial Intelligence at Home*, University of Berkeley, 1999-2004, `http://setiathome.berkeley.edu`.

[11] S. M. Larson, C. D. Snow, M. Shirts and V. S. Pande, *Folding@Home and Genome@Home*, Stanford University, 2003, `http://citeseer.nj.nec.com/589744.html`.

[12] M. Allen et al. *ClimatePrediction.net project*, 2003, `http://climateprediction.net`.

[13] distributed.net Team, *Distributed.net project*, `http://distributed.net`.

[14] chessbrain.net Team, *Chessbrain.net - the world's largest chess computer*, 2003, `http://chessbrain.net`

[15] BOINC Team, *Berkeley Open Infrastructure for Network Computing*, University of California - Berkeley, `http://boinc.berkeley.edu`.

[16] Tiago Andrade e Silva, *Gnutella High-Throughput Computing*, 2004, `http://www.geocities.com/tiagonmas`.

[17] K. Pogorzelski, *TGnutella Delphi component*, 2002-2003, `http://delphi.pogorzelski.de`.

[18] delphi.about.com, *Introducing Borland Delphi*, 2001, `http://delphi.about.com/library/weekly/aa031202a.htm`.

[19] B. Jenkins, *ISAAC: a fast cryptographic random number generator*, 1996, `http://burtleburtle.net/bob/rand/isaacafa.html`.

[20] J. McKeeth, *A guide to Peer-2-Peer*, presented at Borland Conference in California, 2003, `http://www.bsdg.org/jim/Peer2Peer/Paper/3214.html`.

[21] T. Nuyden et al., *Delphi 3D, rapid OpenGL development*, 2001, `http://www.delphi3d.net`.

[22] sun.developers.com, *JavaBeans: The Source for Java Developers*, `http://java.sun.com/products/javabeans/`.

[23] C. Wilhelm, *Semesterarbeit - Mapping the Gnutella Network*, Department Informatik ETH Zuerich, Sommersemester 2003, `http://dcg.ethz.ch/theses.html`.

[24] B. Mandelbrot, *Gli oggetti frattali - Forma, caso e dimensione*, Einaudi Paperbacks, 1987.

[25] K. Nagel, *Simulation of Complex Systems*, Lecture Notes, ETH Zürich, 2003, `http://www.sim.inf.ethz.ch/teach/cosy`.

[26] S. Osokine, *The Flow Control Algorithm for the Distributed 'Broadcast-Route' Networks with Reliable Transport Links*, January 2001, `http://www.grouter.net/gnutella/flowcntl.htm`.

[27] S. P. Holmes ,*a Java Applet for the Coupon Collector Problem*, `http://www-stat.stanford.edu/~susan/surprise/Collector.html`.

[28] W. Feller, *Introduction to Probability Theory and its Applications*, John Wiley and Sons, New York, 1968 (third edition).

List of Figures