Software Engineering for a High-Performance Parallel Astrophysical Visualization Tool

Thomas Kühne

Supervised by Prof. Ben Moore and Dr. Joachim Stadel

A Thesis presented for the degree of

Master of Science ETH in Computational Science
(Dipl. Rech. Wiss. ETH)

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Cosmology & Computational Astrophysics Group
Department for Theoretical Physics
University of Zürich
Switzerland

March 2005

Software Engineering for a High-Performance Parallel Astrophysical Visualization Tool

Thomas Kühne

Submitted for the degree of Master of Science ETH
March 2005

Abstract

In this thesis, I present and document the underlying design and the performance of Hubble in a Bottle!, a high-performance real-time visualization tool for astrophysical TIPSY files. The data can be rotated intuitively in real-time by a virtual trackball using quaternions, and was in contrast to TIPSY and it's successor NChialada designed with only one target in mind: Scalability and Speed.

Therefore many optimzations with respect to memory consumption, computational power as well as interconnection bandwith were investigated.

The main contribution of this thesis is the parallel reduction-tree pipeline, an elegant and easy way to ascribe partial results from all nodes to the master node, while concurrently performing operations on the collected data. This tree-structure can be extended to all other collective MPI-operations, and additionally hides the communication completely behind the computation. Furthermore a novel CORDIC-like algorithm to efficiently taking logarithms, which is used here for the scaling, is described in the appendix.

But I also want to identify the weak points of Hubble in a Bottle!, like the limited scalability, and give some references for further improvements for a potential developer.

Declaration

The work in this thesis is based on software engineering of Tiziano Mengotti and Joachim Stadel carried out at the Cosmology & Computational Astrophysics Group of the Department for Theoretical Physics at the University of Zürich, Switzerland. No part of this thesis has been submitted elsewhere for any other degree or qualification and is all my own work unless referenced to the contrary in the text.

Contents

Chapter 1

Introduction

The development of Hubble in a Bottle! was motivated by the need to quickly display and analyze the results of the astrophysical N-Body simulations, done in our group here, in real-time. Most data visualization packages are lacking either the possibility to efficiently visualize point-based data-sets, or are only available in a serial version. As the number of particles N of such simulations has increased due to algorithmic and hardware advances as $N = 10^{0.3(year-1973)}$ over the last 20 years (Lake et. al., 2004), which is slightly more than twice as fast as the raw computational power of computers has improved in the same time according to Moore's law, it is becoming evident that parallel high-performance visualization tools as Hubble in a Bottle! are getting essential to close the ever growing gap between what we can simulate and what we can actually visualize in real-time. Hubble in a Bottle! comes as a very competitive serial Version, to tackle data-sets up to 10^7 in real-time on individual workstations, and as an MPI-based parallel version for Clusters of Workstations and Supercomputers to make nearly Hubble Volume Simulations (Jenkins et. al., 2001) fly.

Beside zooming, Hubble in a Bottle! can rotate the particles in real-time in an intuitive way using a virtual trackball and quaternions, it also supports four plot modes including two for smooth images, delivers ten different colormaps and an Sobel edge detection filter.

Even though, Hubble in a Bottle! has in the meantime left the experimental stage and is now very stable, it's still a beta version. There are still some things to do, but

it's Open Source under the GNU GPL and can be downloaded for experimenting and further development from `http://hubble.sourceforge.net/` via CVS.
Therefore this thesis, and especially the next chapter, is also thought as a developer's manual in which not only the design and what was done right is described, but also what can be improved and what is missing. At this stage Hubble in a Bottle! consists of roughly 10000 lines of code and much dedication of Tiziano Mengotti and myself. Hence we, together with Dr. Joachim Stadel, would appreciate it and encourage further efforts to improve and extend the capabilities of Hubble in a Bottle!. For all of those who want to use Hubble in a Bottle! the provided readme file and the user's homepage is warmly recommended, as it is intended as a end-user manual for the program.

1.1 History: Who did what, or who is to blame

I would like to emphasise, that I also didn't started from scratch. I started from Tiziano Mengotti's Version 0.336 which can be found at our Sourceforge.net project-page: `http://sourceforge.net/projects/hubble`. Tiziano is the initiator of Hubble in a Bottle! and created the first running version during his tenure as a semester thesis student here at Ben Moore's Cosmology & Computational Astrophysics Group. His serial version was already quite stable and achieved about 5 fps with 1.3 million particles on a 400x400 pixels screen. The according parallel version was very unstable and couldn't scale beyond 1 or 2 fps. But under consideration of the short spell he spend, it was an admirable piece of work.
The trackball was a result of an introductory computer graphics course at the ETH Zürich held by Prof. Gross, which both Tiziano and myself attended. Although I did an implementation of the trackball by myself, the solution of our teaching assistant Reto Lütolf was used, with some minor modifications and optimizations by both of us.
The Tipgrid code described later is a courtesy of Dr. Joachim Stadel, my actual tutor. Except of two errors pertaining the zooming function I let his code unaltered.

Chapter 2

The Architecture of Hubble in a Bottle!

Generally speaking, Hubble in a Bottle! is a particle pipeline, in which all particles are first transformed, which comprises rotation and possible translations, and then projected onto a screen. After that each pixel is colored according to the out coming ValueMap, which is generically the quantity of interest per pixel. Finally an optional filter is performed and the color-bar is painted. All together only basic operations, but what makes it hard is that one has to perform every single step 10 million times per second to achieve 15 fps on the default 768-to-768 pixel screen, which is the best pixel-throughput achieved.

But there's more to it as that, because Hubble in a Bottle! is also available as a parallel version, which is working in a divide and conquer fashion. Therefore all particles are distributed over all concerned processors, which are all doing the necessary computation, concurrently. In the end all partial results are carried back to the master node through a tree structure and moreover pipelined, so that the necessary calculations are concealed behind the inherent communication.

2.1 The Role of OpenGL

As already mentioned Hubble in a Bottle! is designed to manipulate point-based data, for this reason the power of triangle-based graphics cards can not be capitalized

in the obvious way. At least it is challenging to exploit graphics cards for points in a parallel manner, although it would be very interesting to work on this topic. In Hubble in a Bottle! the graphics pipeline is bypassed for this reason, and all the point-based rendering is performed on the CPUs. Of course this is a disadvatage at first glance, as the pure performance of GPUs stays unused, but in this way the rendering can be much more easily parallelized using available parallel programming extensions like MPI or OpenMP.

Even so the OpenGL and the GLUT library is used for the GUI, for displaying the spheres, for lighting and for some image enhancing methods like point antialiasing, smooth-shading and dithering. These operations have to be performed for all pixels and for each frame, but a not too out-dated graphics card can perform them nearly instantaneously, so that the GPU is running most of the time idle.

Although the rendering step is performed separately on the CPUs without using OpenGL, it is consistent with OpenGL, as the rendered frames are written into the framebuffer of the graphics card, accessed through OpenGL commands. OpenGL and Hubble in a Bottle! are thus projecting on the same screen pixels and this above all fully syncronized. In principle this is rather straight-forward, but it can be very demanding to meet the real-time constraints, especially when trying to alter the underlying data-representations.

While it is written in really every OpenGL introduction, one cannot overemphasise it: OpenGL is a State Machine with all it's pros and cons. Everything is a state, and every state which is set is retained up to the bitter end if not changed, even beyond functions and files. Anyone knows it, but nevertheless, it's a major source of errors, so be warned!

2.2 The Basic Design

2.2.1 Some comments about the organization of the files

Altogether the Hubble in a Bottle! package contains 20 C-source files, each with it's own header-file for constant definitions, defines for the C-preprocessor and of course the function prototypes. For a hubble executable 13 and 14 files are used

respectively, depending on if it's the serial or the parallel version.
To give an overlook over the whole program, the essential files are shortly descibed in the following table:

viewer.*	Main program with OpenGL callbacks.
startracker.*	The particle-pipeline: The transform and projection is done here.
MPI_support.*	Functions for the parallel processing, including the reduction-tree.
tree.*	Construction of the reduction-tree and the the CPU-wiring.
trackball.*	Routines for the virtual trackball with quaternions.
colormap.*	Computation of interpolation values for the different color maps.
tipsy.*	Functions for reading TIPSY-files.
filemgr.*	File management routines.
plane.*	The Tipgrid code: Functions for smoothing the images.
filters.*	Post-processing filters, but till now only a Sobel filter is implemented.
threednow.*	Optimized 3Dnow! routines for the AMD architecture.
sse*.*	Optimized SIMD routines for the Intel architecture.
bmp.*	Functions to write bitmap screen-shots into files. (Don't work at all)
hooks.*	Empty skeleton routines for OpenGL hooks.
Makefile*.*	Diverse Makefiles for building the program.
hsmtoeps.*	Functions to integrate hsm-files into TIPSY binary files.
dentopot.*	Functions to convert eps-files into the TIPSY binary file-format.
par_read.*	Functions for reading-in files in parallel, but not used yet.

Table 2.1: Short desciptions of the source-code files

To define it's origin every functions has a unique prefix, e.g. ST_ for startracker.c, MPIS_ for MPI_support.c and so forth.

2.2.2 The Initialization and beyond

The main function is domiciled in the file viewer.c, in which first of all the command line is read, and in case MPI_SUPPORT is defined in startracker.h, the MPI environment is initialized through MPIS_Init. After that the OpenGL Utility Toolkit

(GLUT) environment is set-up followed by a call to ST_Init. In ST_Init the presence of a density-file is checked and the plot-mode is set: Max-Density in case the density information is available, and Line-of-Sight mode otherwise. Then the ValueMap and Image arrays are allocated. Both have entries for each pixel, the first array comprises the value of interest, thus the density of each pixel after transformation and projection in case of the standard max-density plot-mode, whereas the latter array contains the final color of each pixel. Proximate a colormap is selected and an optional filter is initialized. This requires the allocation of a temporary array to buffer the Image array in case the optional filtering is performed. In the end the Viewport is set through glViewport and the pixel storage mode is specified as unpacked byte-alligned.

Going back to the main function of viewer.c, the window is created, the menu is initialized and created and the various callbacks are registered obeyed by the entry into the glutMainLoop. The glutMainLoop is a never-ending loop, which is constantly conducting callbacks till the program is exit.

The underlying concept of callbacks is called event-driven programming. Callbacks are regular functions which are registered to the GLUT system, that means that they are connected to events like pressing a key of the keyboard or pressing a mouse button or even resizing the window. If for example the window is resized the function connected to reshaping the window is called, which is in this case the function callback_reshape which sets the window height and width variables, but also triggers an update of the viewport.

The actually essence of Hubble in a Bottle! is located in the display and in the idle callback. The display callback releases the current GL matrix and calls the trackball function TB_matrix, which returns the transformation matrix. The transformation matrix is used as an input for the function ST_CallList or as the case may be MPIS_CallList, whose result, the fully rendered image, is then written into the framebuffer together with the spheres through ST_DisplayBuffer. Then double buffering is performed by glutSwapBuffers and the GL matrix is fetched back from the matrix stack. It is then matter of the idle callback to mark the framebuffer for redisplay via glutPostRedisplay and to force its execution within finite time by

glFlush.

As in the parallel version the whole frontend is performed only on the master node, the glutMainLoop is emulated by a function called MPIS_Receive-MatrixAndCompute. The arising question of how to transmit user-interaction in a consistent way to all processors, especially when they are running highly asyncroniously will be in-depth covered in the forthcoming user-interaction section.

2.2.3 The Driving Core

The intrinsic engine of Hubble in a Bottle! are the functions called by ST_CallList or MPIS_CallList, if MPI support is enabled. Except of an additional call of MPIS_SendUserInteraction these two version are fully symetrically. Many other functions are also available in a ST_ and a MPIS_ version for the serial and the parallel variant of Hubble in a Bottle!. If so, both functions are implementing the same logic, but this doesn't mean that the codes are looking similar, as parallel programming is fundamental more cumbersome than serial programming.

Coming back, the function CallList wrap the function ST_Project, scales the ValueMap in event of SMOOTH is defined, is doing the color mapping and performs the filter. In general scaling is done right after the TIPSY file is imported, because this way the scaling has to be performed only once. Nothing speaks against this as long as only the line-of-sight or the max-density mode is used. The scaling is about log-scaling, which is non-linear. Making the scaling during the preprocessing and selecting the maximum value is fine, but summing up, as it is done in the smoothing modes, is not feasible. In this case the scaling has to be done on-the-fly for each frame in the function CallList.

The function ST_Project is responsible for the transformation and the projection. More precisely the transformation is the multiplication of each position vector with the 4×4 transformation - matrix coming from the trackball function TB_matrix. The position vector is a 4×1 vector with the entries x, y, z and w for the harmonic coordinate. The transformation is calculatng the new positions in the physical space, wheras the projection is resposible to figure out at which screen coordinate the point has to be shooted. For the projection the obvious parallel projection is used, which

has the benefit, that the z-coordinate is neglected, which saves valuable memory space.

Having computed the pixel screen, the according ValueMap entry is updated, and here the different plot-modes come into play. In the line-of-sight mode the points intersecting the line-of-sight are determined, the finally value written into the ValueMap is the sum of surface densities

$$\sum m_i * (n_x/L_x)^2, \tag{2.1}$$

where m_i is the mass, n_x the cell-dimension and L_x the simulation length.

The maximum density mode selects the maximum density of all the points falling onto the same screen pixel. The smoothing modes are calling either the iPlaneSlice or the iPlaneProject function, which will be descibed in the section about the Tipgrid code, but because the resulting values are summed up in the plane context, the scaling couldn't be done during the preprocesing. Speaking of the plane context, that has to be initialized through the PlaneInitialize function. In Hubble in a Bottle! the plane context is initialized and finished at every frame, as the plane context fields are reset, but above all to update the simulation langth variable, which I have choosen to be the inverse of my zooming-factor, in terms of simulating zooming even in the smoothing modes.

The only issue left is that the cell values have to be extracted from the plane context before releasing it, which is by the way done with PlaneFinish.

Additionally the calculation of the frame-rate is performed here, to take only the computation and the communication into account, but disregard the SSH latency, so that the correct frame-rate is displayed even if one is using SSH.

2.2.4 The Parallel Version of Hubble in a Bottle!

Since all topics of interest are dedicated a separate subsection, I will keep this one quite short. After all particles with their positions, densities and smoothing radii are read in and distributed, the master node went into the glutMainLoop. The glutMainLoop causes the master node the execute the display callback for each frame, which imply a call to MPIS_CallList. In contrast the other nodes are entering the

function MPIS_ReceiveMatrixAndCompute, which is a never ending loop, that reverse engineers the glutMainLoop function plus covers the user interaction.
The parallelism is realized here through divide-and-conquer, this means that the particles are distributed over all available CPUs. Each CPU is performing the transformation and projection operations locally. After each frame all intermediate results are reduced to the master node. The local ValueMaps are all of the same size as the global ValueMap, with the constraint, that only the elements which are belonging to local points after the transformation are non-zero. The two consequences out of this is that depending on the number of CPUs the local ValueMaps may be very sparse, and second that the local ValueMaps have to be merged into the global ValueMap contingent on the plot mode. In case of the max-density mode, the reduction-operation is the max-function, whereas it's otherwise the sum-opertion. To my mind the possible sparsity couldn't be exploited, as long as the CPU quantity and the sparsity aren't huge.

2.3 The Parallel Tree and it's generation

As indicated the local ValueMaps are ascribed using a tree. In Hubble in a Bottle! a binary tree is used, since my own experiments with different n-ary trees uncovered, that n=2 or at most n=3 seems to be the best choice, since higher options for n yields to much merging at every single node and leads to a decrease in performance. What many don't know is that being a binary tree is only a very week restriction. The following recurrence formula, which is equivalent with Sloane's integer sequence A001699, quote how many different valid binary trees are possible with altogether n nodes and leafs:

$$a_n = a_{n-1}^2 + a_{n-1}(1 + \sqrt{4a_{n-1} - 3}) \tag{2.2}$$

A binary tree with together 7 nodes and leaves, which is used in Hubble in a Bottle! for 4 CPUs, has for instance has 44127887745696109598901 different realizations. Even the most restricted binary tree which is the balanced binary-tree, in which each node except the tree leaves has a left and right child, and all tree leaves are at

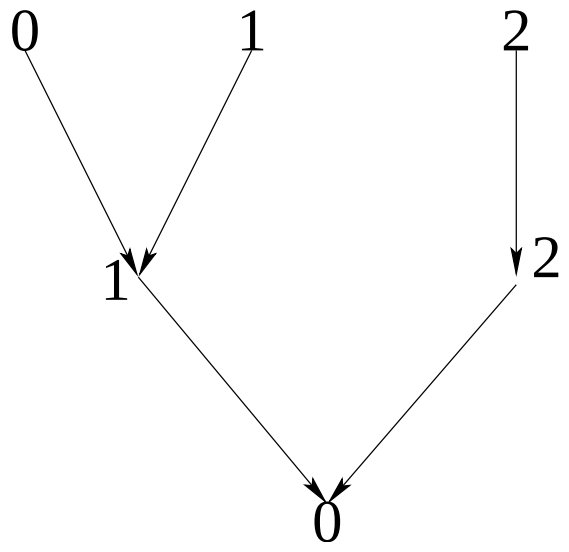

Figure 2.1: Perfect Binary Tree

the same level, has

$$C_n = \frac{1}{n-1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!n!} \tag{2.3}$$

different realizations in the event of together n nodes and leaves. By the way C_n are the Cartan numbers, or Sloane's integer sequence A000108.

The height of a binary tree can be highly variable, as for a binary tree of height h with n nodes only the follwing weak inqueality counts:

$$h \leq n \leq 2^h - 1 \tag{2.4}$$

With the benefit of hindsight I should have used the complete binary tree, in which every level, except possibly the deepest, is completely filled. This tree turned out to be the best in terms of performance, ease of implementation and elegance. The tree I used is also a balanced binary tree, more precisely the perfect binary tree, with the same height as the height-balanced binary tree, but I overlooked the third performance determining factor beside degree and height which is the offset. A perfect binary tree is a binary tree with all leaf nodes at the same level, but allowed offset, as seen in Figure 2.1.

The accumulated offset is the same, but it's not perfectly distributed, so that the load of the CPUs is not fully balanced. This is because the function TR_treeGen is assigning the available CPUs in a pairwise row-major manner from the leafs to the

root.

In detail all CPUs are first of all assigned to leaves, then it is iterated over all these leaves and they are merged together to an inner node as long as an even number of leaves is left. The resulting innernodes out of the merged leaves are numbered backwards, starting from n-1, as a binary tree has always one less inner node as leaves. In the odd case the remining node is propagated one level up, but stays a leaf. This procedure is continued till the root with CPU_ID=0 is reached.

At certain CPU numbers it could happen, that the rightmost CPU is propagated over multiple levels and gets an offset greater than 1. Now one can see the difference between the complete binary tree and the perfect binary tree: the complete binary tree allows only a maximum offset of one.

As next TR_bugGen generates the bug table out of the wiring tree. A bug is the wiring for each node, which is called bug as each node has two inlets and two outlets. To get the bug the ensemble tree is traversed, and for each node the predecessor and the succesor is recorded. These sets are scaned for manifold in- and outputs, which have to be eliminated. Now only the calculation of the offset is remaining, that is equivalent to couting the number of self-referencings at the first level. In Figure 2.1 for instance the offset of node 2 is one.

In reality the generation of the tree and the bugs is rather complicated as many exceptions has to be covered. This is an evidence that my matrix data-structure isn't the way to go, and that this ought to be done through a much more elegant and intuitive tree data-structure, since the underlying wiring is also a tree-structure. Beside that my way is horrible complicated it works fine and is reasonalby fast, therefore if one is not interested in changing e.g. the tree-structure, one should consider these functions as a working black box.

The final issue left is that each binary tree has one less inner-node as leafes, which causes that one node has only one in- and one outlet instead of two and furthermore that the master-node has only one outgoing. If not covered it's only a matter of time till Hubble in a Bottle! crashes with a buffer-overflow, triggered by underemplyed nodes, which are running too much out of sync. Therefore a self-syncronizing mechanism has to be included. As one can recognize from Figure 2.2 this is tackled

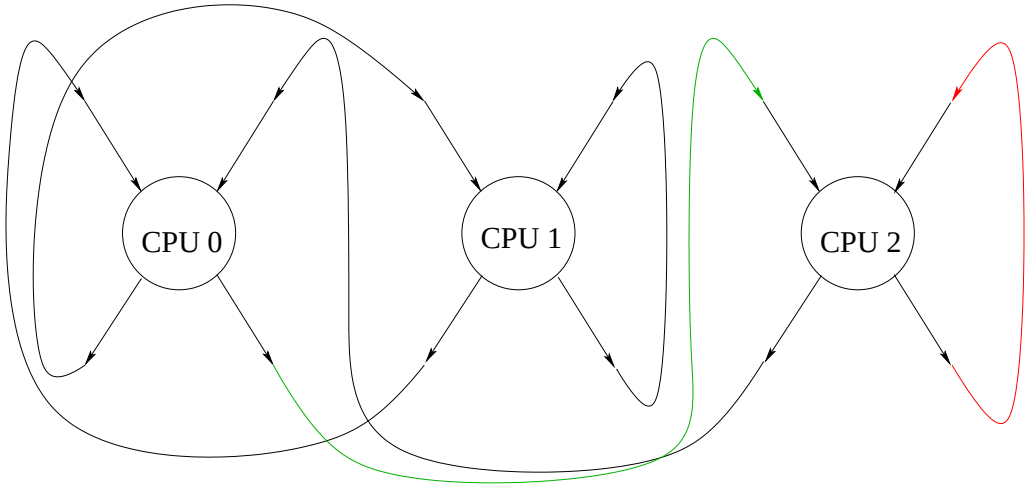

Figure 2.2: CPU-Bug wiring

here by connecting the last node to itself (red arrow) and to the master-node (green arrow). From this it follows that the last node is only sending out it's local ValueMap, the merged ValueMap is sended to itself only to syncronize the last node, and in order to give the master-node a target to send it's result to keep all bugs symmetrically. Additionally the master node is writing the final result to the global ValueMap, and is shifting the whole array to the right, by extending the loop and skiping the first iterations.

2.4 The Parallel Tree Pipeline

It turned out to my surprise, that reducing the local ValueMaps in parallel to the master node, wasn't a breakthrough in performance. The subsequent analysis exposed, that most of the time was wasted by waiting for merging and sending or receiving respectively of local ValueMaps.

The idea of leading back the local ValueMaps to the root in a tree-wise manner can only be exploited to the full extent by pipelining it. Pipelining is done through packetizing the ValueMaps into small chunks and sending them in succession instead alltogether. Moreover non-blocking sends and receives have to be used, to overlap the transmission time with calculation. In MPI non-blocking sends and receives are implemented via MPI_Irecv and MPI_Isend.

Summarized every node is getting a local and a merged ValueMap chunk at each iteration, and is also sending out it's local and it's merged ValueMap. The merged output and the according inputs are displaced by one iteration, because the inputs are merged during receiving the inputs for the next iteration, so that at no instance of time it really has to wait at all. By the way no time consuming syncronization barriers are used at all through out the whole code, since the ensemble pipelined tree is self-syncronizing as already outlined.

The adjustment of the actual buffersize is crucial for the resulting performance and denpends highly on the concrete machine. It shaped out, that 128kB seems to be the best choice for the zBox, which is in full agreement with the benchmarks performed by Dr. Joachim Stadel, but on an empty machine. As Hubble in a Bottle! is using 16 bit unsigned short int and 32 bit float respectively depending if FAST is defined or not, the 128kB are equivalent to 65536 or 32768 ValueMap array elements. Because the optimal buffersize depends on multiple factors which are in part conflicting each other, I would not like to think myself capable to make any predictions about the optimal buffersize for the forthcoming zBox2.

2.5 Parallel User Interaction

As already indicated it's far from obvious how to propagate user interactions to all processors in a consistent way, since user interaction can be carried out fairly anytime. Also in Hubble in a Bottle! the CPUs are running greatly independent and therefore asynchronous from each other due to lack of any syncronization barriers, which makes Hubble in a Bottle! fast, but also user interaction propagation rather complicated.

The solution I came up with are user interaction arrays which are broadcasted together with the transformation matrix for each frame. The variables with are send to all processors are the image size, the actual zooming factor, some status bits determining if certain function are turned on or off, the translation and a flag if these values are valid, since tranlations don't have to be performed everytime. In the end there are variables for the mask level, the projection width and the z-shift

which are all belonging to the smoothing modes, but this is all stuff that is better looked up directly in the code in the function MPIS_SendUserInteraction.
In the beginning the amount of variables that have to be tranmitted was pretty manageable, bit by bit I realized that even more variables have to be transmitted, and additionally the integration of more and more features and functions leads to anymore user interaction variables. Finally the the user inteaction arrays have become fat, so that in the meantime I arrived at the conviction, that organizing the user interaction variables in an well organized struct would be the more elegant way.

2.6 Optimizations: A better Transform & Project

The transform section is the one which consumes the most compute power, and is in the serial variant of Hubble in a Bottle! the performance determining factor. Already Tiziano Mengotti invested a good deal of work in this section, by writing dedicated SIMD procedures in SSE and 3Dnow for the Intel and the AMD architecture. As specified in the above overview the transformation consists of multiplying the $4 \times n$ particle matrix with the 4×4 transformation matrix. As only the very simple parallel projection is used here, which don't need the z-coordinate, I integrated the transformation and the projection into a single block. The benefit of this is that the z-coordiante don't have to be calculated and is hard coded to zero, and that one can save an additionally function call per iteration. Even in an ANSI C version, but with loop unrolling, this is about 10% faster than Tiziano's SIMD versions.
As frontside bus throughput is an important issue, especially in the parallel version, because both CPUs of a single zBox node were using only one frontside bus together, Joachim Stadel came up with an idea to save precious memory bandwidth. Instead of overwriting the whole particle matrix with the new values after each transformation, Joachim proposed to retain the particle matrix unchanged. The actual transformation matrix is in this case received through multiplying the old transformation matrix with the transformation matrix computed by the trackball function TB_matrix.
My implementation of this proposal is slightly different, but retains the main idea.

As the particle matrix is composed of n 4×1 position vectors, I decomposed the matrix-matrix multiplication into a vector-matrix multiplication. This way only not the whole matrix has to be stored, but only a single 4×1 vector after the transformation. In order to get it straight again the matrix-matrix multiplication is substituted by a loop of all particles with a single vector-matrix multiplication. Of course in the sum over all iterations the same amount of data is stored, but because only 4 floats are stored at once, they are buffered in registers instead of in main memory, so that the front side bus isn't used at all.

In Tiziano's version the particle matrix is many magnitudes bigger than the 4×4 transformation - matrix, so that he only took care about cache locality of the particle matrix, although he confessed on inquiry that he done it right by chance.

Cache locality is crucial in terms of performance, since not a single data is fetched from the cache, but a whole cache line of typical sizes between 8 to 64 bytes. As cache misses are inherently expensive, one should take care to get the most out of a single cache line fetched from the cache. This is done by iterating over matrices in the natural order as they are stored in memory. The irritating point is that different programming languages are mapping multidimensional arrays to linear arrays differently, which leads to that textbook solutions couldn't be inherited without taking care about the structure of the matices.

Since the particle matrix is split into position vectors, one could think that it's benetitial to rewrite the trackball functions so that the transformation matrix is built-up in a transposed way. Although this task seems to be very easy it turned out that the opposite is the case, because also the spheres should deal with the transposed matrices, without actually transposing the matrix, as the benefit of increasing the cache-locality would be away. I never got it right for the points and for the spheres at the same time, but it later turned out, that this is inconsiderably, because the cache line size of a Pentium4 CPU is 64 byte so that the whole transformation matrix fits into a single cache line no matter if the matrix is stored row- or column-major.

The present transform & project isn't anymore programmed using SIMD instructions, but in optimized ANSI C, due to lack of time. However it would be defini-

tively a very nice mini project to implement a fully optimized version of transform & project, perhaps with GPU support, because it only needs very limited knowledge of the whole code, but more to come in the outlook section.

2.7 Integration of Tipgrid for Smooth Images

The much mentioned Tipgrid code is a particle to grid converter devised from Dr. Joachim Stadel. To make a long story short, Tipgrid is projecting the particles onto a plane or a small slice respectively by weighting the particles within their smoothing radii with a kernel.

Till now tipgrid can only make orthogonal cuts, so that some workarounds were necessary, to furnish Tipgrid the whole navigation functionality of Hubble in a Bottle!, like real-time rotation, translation and zooming. Now one can also uptake the power of the, in the last section presented, vector-wise rotation of single points, since this makes adding the navigation functionality easy without changing the Tipgrid code at all.

The only difficulty left is to take care where to perform the operations. First of all in front of the loop, which iterates over all points, the Tipgrid plane is initialized through PlaneInitialize. The translation is done by adding the translation vector to the position vector of each particle before the transformation. After that the actual transformation is accomplished, this is then equivalent to a rotation of the Tipgrid plane, which isn't working correctly so far. Afterwards the ensemble plane could be shifted in z-direction if desired, by adding a constant value to the z-coordinate after the transformation. Now all preconditions are fulfilled to call the function iPlaneSlice, or as the case may be iPlaneProject. The projection step is consciously skiped, since the functions iPlaneSlice and iPlaneProject are working in physical space and performing the projection by themselves. The resulting pixel values are written into the Tipgrid context, from which they have abstracted in a separate loop over all screen pixels after the particle loop has finished. If multiple different particles are project onto the same screen pixel they are added together. This is the reason that the log-scaling couldn't be performed in the preprocessing of Hubble in a Bottle!,

if SMOOTH is defined, but I found a way to minimize the penalty of scaling the ValueMap during runtime in every iteration, but this subject of the next section.

2.8 The Theme of Scaling

The scaling is performed through the following formula:

$$ValueMap[i] = (MaxColors - 1) * \frac{6600 * \log_{10}(ValueMap[i] + 1)}{32768} \tag{2.5}$$

to receive equivalent colored frames like TIPSY.

If SMOOTH isn't defined scaling isn't a topic at all, because it's done during the preprocessing for the whole ValueMap by the master node alone and only once. Since scaling couldn't be done at preprocessing time for the well-known reason that the logarithm isn't additive, therefore the scaling has to be performed in every iteration during runtime for the ensemble ValueMap. The application of identity (2.5) results in a tremendous decrease in performance. However, as most of the pixels are black, what is equivalent to $ValueMap[i] = 0$, $\log_{10}(ValueMap[i] + 1) = 0$ and therefore also the whole term (2.5). This simple modification implicates that scaling leads to a slow down anymore.

Furthermore $\log_{10}(x)$ could be substituted by $\frac{\log_2(x)}{\log_2(10)} = \frac{\log_2(x)}{3.321928...} = 0.30103... * \log_{10}(x)$. Using the CORDIC-style algorithm, I'm describing in the appendix, to calculate the remaining logarithm dualis, taking logarithm could be potentially further accelerated. This algorithm isn't implemented yet, but is going to be integrated in the near future.

2.9 Transformations in Homogeneous Coordinates

The three transformations used by Hubble in a Bottle are:

$$\begin{aligned} \mathbf{P'} &= \mathbf{T} + \mathbf{P} \quad (Translation) \\ \mathbf{P'} &= \mathbf{S} * \mathbf{P} \quad (Zooming) \\ \mathbf{P'} &= \mathbf{R} * \mathbf{P} \quad (Rotation) \end{aligned} \tag{2.6}$$

The sum of all transformations describing an affine transformation, the translation is thereby treated as an addition. It would be desirable, if all transformations could

be described as consistent multiplications. For this reason, so called homogeneous coordinates are iintroduced. They could be considered as a extension of the vectors through an additional dimension. A 3D Point $\mathbf{P}$ is then represented through the vector $\mathbf{P} = (x, y, z, W)^T$. Every 3D point $\mathbf{P}$ has then infinitely many homegeneous coordinates depending on the choice of W. Since points which are emanating from each other, through a multiplication with a scalar t, are equivalent, every point could be understood as a straight line in a higher dimensional space of homogeneous coordinates. Through homogenization, what is the division through W, one receives $\mathbf{P} = (\frac{x}{W}, \frac{y}{W}, \frac{z}{W}, 1)^T$, which represents an affine 3D subspace, or 3D hyperplane, in the 4D space of homegeneous coordinates.

Now all transformations, and not only the three used here, can be expressed as 4×4-matrices:

$$\mathbf{T}(t_x, t_y, t_z) = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (Translation)$$

$$\mathbf{S}(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (Zooming) \tag{2.7}$$

$$\mathbf{R_x}(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (Rotation)$$

The rotations $\mathbf{R_y}(\theta)$ and $\mathbf{R_z}(\theta)$ are analog.

$$\mathbf{R_y} = \begin{bmatrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \mathbf{R_z} = \begin{bmatrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.8}$$

The generic matrix composed of multiple affine transformations looks as follows:

$$\mathbf{M} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.9}$$

Affine transformations may be arbitrarily concatenated as matrix products in homogenous coordinates, but two matrices $\mathbf{M_1}$ and $\mathbf{M_2}$ are only commutative if the following holds:

$\mathbf{M_1}$	$\mathbf{M_2}$
Translation	Translation
Zooming	Zooming
Rotation	Rotation
Zooming	Rotation

The matrix returned by the function TB_matrix is in the same line as matrix (2.8), and is therefore called transformation matrix and not only rotation matrix.
Homogeneous coordinates are used throughout the whole code and shouldn't be mixed up with the quaternions described in the following section.

2.10 Mathematics of Projection

In this section I only want to derive the projection matrix of the parallel projection, which is a special case of the perspective projection. In Figure 2.3 the perspective projection to onto a plane parallel to the XY-plane at distance $z = d$ is shown. For the perspective projection the following holds:

$$\frac{x_p}{d} = \frac{x}{z} \tag{2.10}$$

$$\frac{y_p}{d} = \frac{y}{z} \tag{2.11}$$

For $z \neq 0$, this is equivalent to:

$$x_p = \frac{d * x}{z} = \frac{x}{z/d} \tag{2.12}$$

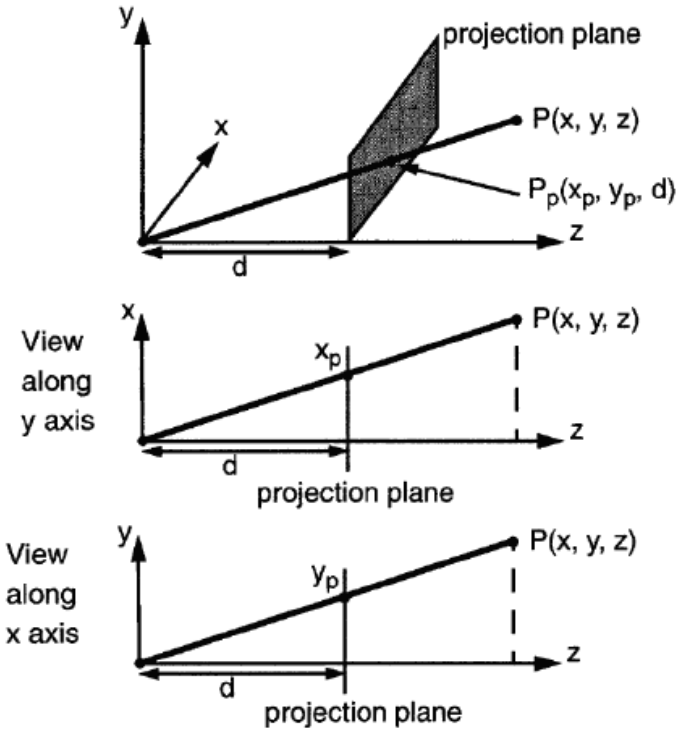

Figure 2.3: Perspective Projection

$$y_p = \frac{d * y}{z} = \frac{z}{z/d} \tag{2.13}$$

By means of homogeneous coordinates the projection could be expressed with the following 4×4-matrix:

$$\mathbf{M_{per}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{d} & 0 \end{bmatrix} \tag{2.14}$$

For an alternative perspective projectio into the $(z = 0)$-plane with the center $z = -d$, as seen in Fugure (2.4), the following holds:

$$\frac{x_p}{d} = \frac{x}{z + d} \tag{2.15}$$

$$\frac{y_p}{d} = \frac{y}{z + d} \tag{2.16}$$

or as well

$$x_p = \frac{d * x}{z + d} = \frac{x}{(z/d) + 1} \tag{2.17}$$

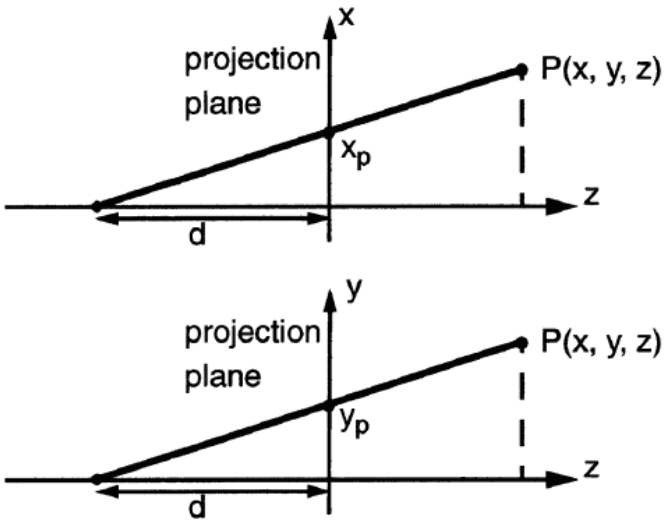

Figure 2.4: Alternative Perspective Projection into the $(z = 0)$-Plane

$$y_p = \frac{d * y}{z + d} = \frac{z}{(z/d) + 1} \tag{2.18}$$

For the according 4×4 projection-matrix one receives:

$$\mathbf{M'_{per}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{d} & 1 \end{bmatrix} \tag{2.19}$$

The parallel projection is then the limit for $d \to -\infty$. The resulting projection-matrix for the parallel projection used in Hubble in a Bottle! is finally

$$\mathbf{M'_{ort}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.20}$$

2.11 The Trackball and the Quaternion Principle

Another very elegant possibility for the definition of rotations and translations are the so called quaternions. In comparision to homogeneous coordinates they have a more compact representation, but they are also computational a little bit more

expensive.

Although homogeneous coordinates and quaternions are covering the same purpose, they are completely different concepts and shouldn't be confused. Quaternions are in Hubble in a Bottle! only used in the trackball functions, since it's much more intuitive and elegant to code movements on a virtual sphere as quaternions.

2.11.1 Defintion and Properties of Quaternions

Quaternions are defined as follows:

$$q = c + xi + yj + zk \tag{2.21}$$

whereas c, x, y, z are real and i, j, k are imaginary numbers. In general the subsequent notation is used:

$$q = c + u \tag{2.22}$$

in which c is the real part and $u = xi + yj + zk$ the absolute or pure quaternion. A quaternion may hence understood as a "hypercomplex number" and describes a reasonable extension of a complex number $c = a + bi$ into the fourth dimension. Onto quaternions a series of operations are defined:

- Addition

$$q + q' = (c + c') + (x + x')i + (y + y')j + (z + z')k \tag{2.23}$$

- Multiplication

$$qq' = (c + u)(c' + u') = (cc' - u \bullet u') + (u \times u' + < cu' > + < c'u >) \tag{2.24}$$

in which the operations inner product $\bullet$, scalar multiplication $<>$ and cross-product $\times$ are defined as follows:

$$u \bullet u' = xx' + yy' + zz' \tag{2.25}$$

$$< cu >= cxi + cyj + czk \tag{2.26}$$

$$u \times u' = (yz' - zy')i + (zx' - xz')j + (xy' - yx')k \tag{2.27}$$

The hence derived properties

- Multiplication property of the basis [1,i,j,k]

$$\begin{aligned} & i^2 = j^2 = k^2 = -1 \\ & ij = k,\ ji = -k; \quad jk = i,\ kj = -i; \quad ki = j,\ ik = -j \end{aligned} \tag{2.28}$$

- Additive and multiplicative identity

$$\begin{aligned} 0 &= 0 + 0i + 0j + 0k \\ 1 &= 1 + 0i + 0j + 0k \end{aligned} \tag{2.29}$$

- Additive and multiplicative inverse

$$\begin{aligned} -q &= -c - xi - yj - zk \\ q^{-1} &= \tfrac{1}{|q|^2}\bar{q} \end{aligned} \tag{2.30}$$

- Conjugated element $\bar{q}$ to $q = c + u$

$$\bar{q} = c - u \tag{2.31}$$

- Absolute value of quaternions

$$\begin{aligned} q\bar{q} &= (c^2 + u \bullet u) + (u \times u - < cu > + < cu >) \\ &= c^2 + x^2 + y^2 + z^2 = |q|^2 \end{aligned} \tag{2.32}$$

turning the set of quaternions to a ring $(\mathbf{Q}, +, *)$ in the mathematical sense. As a ring is an abelian group under addition, but only a semigroup under multiplication, the multiplication of quaternions is not commutative.

The, in connection with transformations, important quaternions are the unity quaternions, thus quaternions with absolute values of one. Using the unity vectors $\mathbf{N} = [N_x, N_y, N_z]^T$ in $\mathbf{R}^3$ and $\mathbf{I} = [i, j, k]^T$

$$\begin{aligned} & q = c + u \\ & u = sn,\ with\, n = \mathbf{N} * \mathbf{I} \end{aligned} \tag{2.33}$$

From this it follows, that $|q|^2 = c^2 + u \bullet u = 1$ could be rephrased to $c^2 + s^2 = 1$, so that every unity quaternion could be expressed in the form

$$q = cos(\Theta) + sin(\Theta) * n \tag{2.34}$$

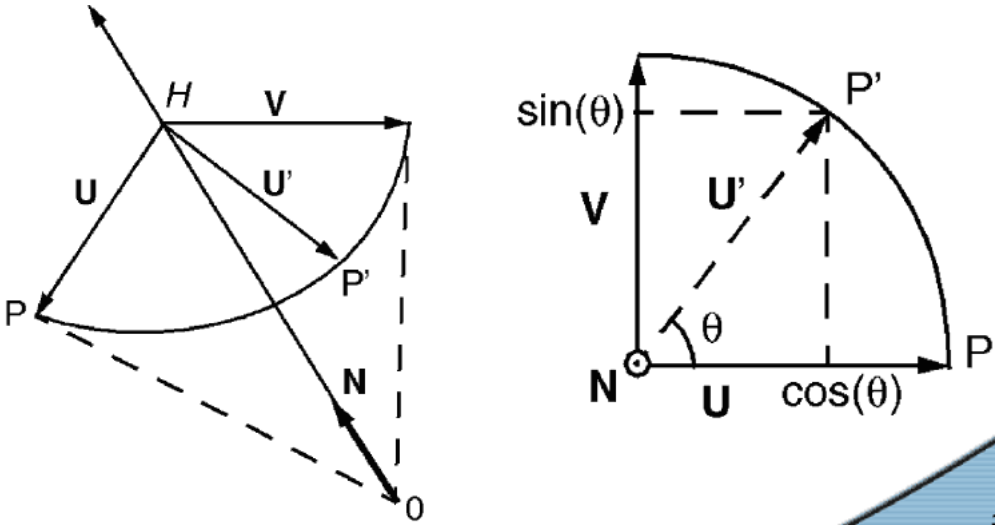

Figure 2.5: Characteristic Vectors of a 3D Rotation

2.11.2 Rotations and Translations using Unity Quaternions

The rotation of a point **P** to **P'** about an arbitrary axis **N** is shown in Figure 2.5. The conditional equation of point **P'** is

$$P' = cos(\Theta)\mathbf{P} + (1 - cos(\Theta))\mathbf{N}(\mathbf{N} * \mathbf{P}) + sin(\Theta) * (\mathbf{N} \times \mathbf{P}) \tag{2.35}$$

presumed $\mathbf{U} \perp \mathbf{V}$ and $\mathbf{V} \perp \mathbf{N}$.

The according rotation-matrix $\mathbf{R}(\Theta, \mathbf{N})$ is

$$\mathbf{R}(\Theta, \mathbf{N}) = cos(\Theta)\mathbf{I}_3 + (1 - cos(\Theta))\mathbf{N}^T\mathbf{N} + sin(\Theta)\mathbf{A}_N \tag{2.36}$$

with

$$\mathbf{N} = [N_1\ N_2\ N_3], \quad \mathbf{P} = [P_1\ P_2\ P_3], \quad \mathbf{I}_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \mathbf{A}_N = \begin{bmatrix} 0 & N_3 & -N_2 \\ -N_3 & 0 & N_1 \\ N_2 & -N_1 & 0 \end{bmatrix}$$

If one considers the analogy between the point $\mathbf{P} = [x, y, z]$ in 3D and the pure quaternion $p = 0 + \nu = xi + yj + zk$, and also a unity quaternion $q = c + u$, then the following rotation operation with quaternion could be defined:

$$R_q(p) = qp\bar{q} \tag{2.37}$$

After a few conversion, one receives

$$R_q(p) =< (c^2 - u \bullet u)\nu > + < 2(\nu \bullet u)u > + < 2c(u \times \nu) > \tag{2.38}$$

Since q is a unity quaternion, $q = cos(\Theta) + sin(\Theta) * n$ could be used, so that

$$R_q(p) =< cos(2\Theta)\nu > + < (1 - cos(2\Theta))(n \bullet \nu)n > + < sin(2\Theta)(n \times \nu) > \tag{2.39}$$

is received.

The compiled realtions are of fundamental importance, as

- R_q describes the 3D-rotation of an arbitrary point **P** at an angle of 2Θ around the axis **N**.
- Vice versa a 3D-rotation $R(\Theta)$ is fully described through a quaternion q. The rotation of a point **P** simplifies itself to two multiplications with q and $\bar{q}$.

$$\begin{aligned} p' &= qp\bar{q} \\ q &= cos(\tfrac{\Theta}{2}) + sin(\tfrac{\Theta}{2})n \end{aligned} \tag{2.40}$$

In the same way the translation and the concatenation of rotations and translations could be described with quaternions, which is especially interesting in the context of animation.

Assuming that p and t are pure quaternions, which are characterizing the point **P** in space and the translation vector **t**, so the following relation holds:

$$p' = p + t \tag{2.41}$$

Assume further that r is a unity quaternion, which specifies the rotation, thus a concatenation of translation and rotation could be carried out through an operator $M_{(t,r)}$.

$$p \rightarrow p' = M_{t,r}(p) = rp\bar{r} + t \tag{2.42}$$

Postulating that $M_{0,r}$ qualifies a pure rotation, and $M_{t,1}$ a pure translation, as one may see in Figure 2.7, the resulting transformation could be composed as

$$M_{t,r} = M_{0,r} * M_{t,1} \tag{2.43}$$

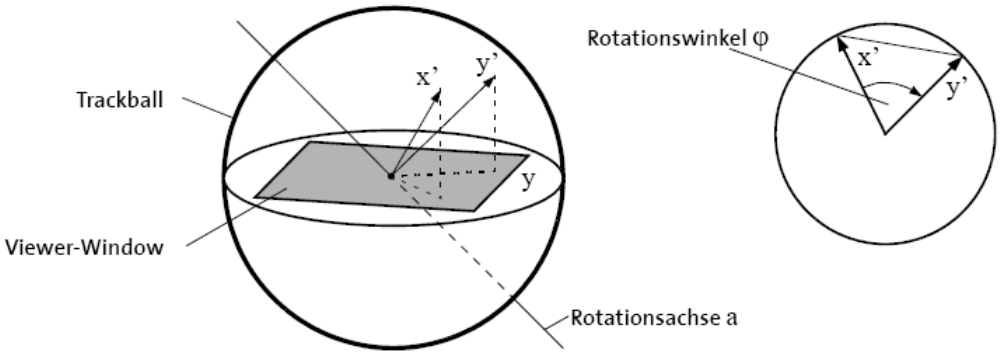

Figure 2.6: The Virtual Trackball

The implementation of transformations in this vein is now quite simple. One only have to implement routines for the described elementary operations on quaternions. The transformation is now fully represented via $\mathbf{t}$, $\mathbf{N}$ and Θ. No 4×4-transformation-matrix is needed anymore!

2.11.3 The Virtual Trackball

The virtual trackball is very well suited to rotate an object on the screen interactively and in real-time. A virtual trackball is realized, as one can see in Figure 2.6, by calculating the rotation axis a and the rotation angle φ of a virtual sphere. The rotation axis a is obtained over the cross-product of the vectors x' and y', which are themselves received out of the projection of the coordiantes in the viewer-window onto the virtual trackball. The rotation angle φ is gained through basic trigonometric relations. Out of the rotation axis a and the rotation angle φ a single quaternion is built, which completely describes the rotation. For every individual section of the animation a new quaternion is computed to encode the rotation. To simulate a longer motion, as in Figure 2.7, the respective quaternions are multiplied. To really rotate an object in the viewer-window, the calculated quaternions are in the end converted into 4×4 rotation-matrices.

In comparison to the definition of quaternions, the real part of a quternion lies, in my implementation, at the final position. This has the advantage, that all operations on the imaginary part could be directely performed on the 4×1 quaternion-vector

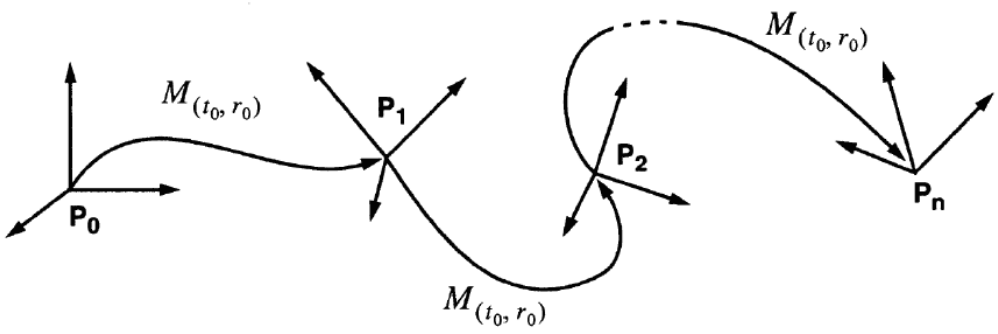

Figure 2.7: Flythrough

using OpenGL vector operations.

2.12 Read in and the corresponding tools

The read in is performed using TIPSY functions written by Joachim Stadel. In the parallel version of Hubble in a Bottle! the master node reads in the file chunk-wise, and directly sends the file pieces in a non-blocked fashion, so that the master node don't have to wait until the send is successful completed, to all slave nodes.
The original release of Hubble in a Bottle! from Tiziano Mengotti, was working with two files: a binary file, in which the particles were stored, and with an ASCII file in which all the densities were stored. Since the read in of ASCII files is noticeable slower than the read in of binary files, the unused fields of type float of the TIPSY dark_particle were used to encode the densities and later also the smoothing lengths. This was overdue, as the problem sizes have increased to tens of millions particles and first of all because the amount of read in data has doubled, since beside the particle positions and densities, the smoothing radii and the velocities are read in, although up to now the velocities are not used for the visualization.
The fast read in of data in binary form is accomplished by the two small converting tools HsmToEps, written by Joachim Stadel and the similar EpsToPot, written by myself. HsmToEps is taking a binary bin-file consisting of the particle postions and a hsm-file comprising of the smoothing radii, and outputs an eps-file via an UNIX pipe. In an eps-file the eps-fields of the TIPSY dark_particle are used to store the

smoothing radii for all particles. EpsToPot is working analog, except that an eps- and a density file are feed in and a pot-file is returned. As here a nor used field in the dark_particle is abused to encode the content, of an otherwise necessary, ASCII file. A pot-file is at the same time the only input file for the latest version of Hubble in a Bottle!, no other file is necessary anymore, and it brings down the time used for preprocessing to a acceptable level.

The next step to further speed-up the preprocessing would be to read-in the input file by all involved nodes in parallel. Because I was running out of time, the parallel read in isn't completed yet, and is only available as an imperfectly working alpha-version in the source code. It's therefore another small task to tackle, but furthermore in the outlook section.

Chapter 3

Performance

All benchmarks were performed either on my workstation (3.2 GHz Intel Xeon) for the serial version, or on the zBox for the parallel version of Hubble in a Bottle!. The zBox is an in-house designed massively parallel supercomputer with 288 AMD Athlon-MP 2200+ CPUs, connected by an 2D-torus SCI interconnection system from Dolphin Interconnect Solution.

The performance is measured in terms of frames per seconde (fps), as I observed that other kinds of performance measurement as e.g. responsiveness are fully dependent of the frame-rate, but harder to quantify. I also want to accentuate, that all measurements of the parallel version are done with full CPU and interconnection load on the zBox, except for the nodes which were used by Hubble in a Bottle!.

For the measurements, the follwing real datasets were choosen:

Virgo.pot	1314161	particles
total.00580.pot	2460000	particles
MMCluster.pot	6298040	particles
top.00038.pot	27000000	particles

(3.1)

If not stated otherwise, the maximum-density plot-mode is used. Furthermore the define FAST or SMOOTH is set, the former in case of the max-density or line-of-sight plot modes, and the latter in the case of one of the smooth image modes.

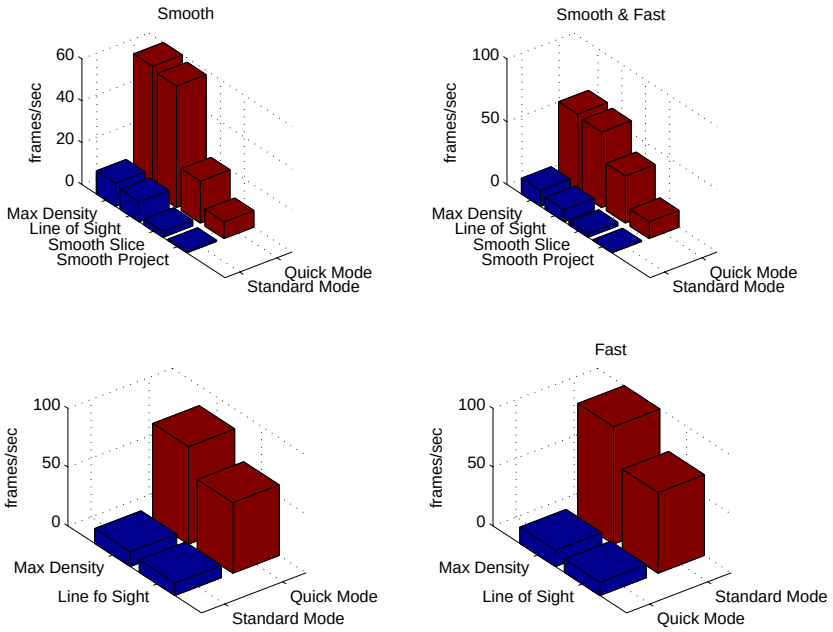

Figure 3.1: Serial Scaling of Hubble in a Bottle! for the Virgo dataset

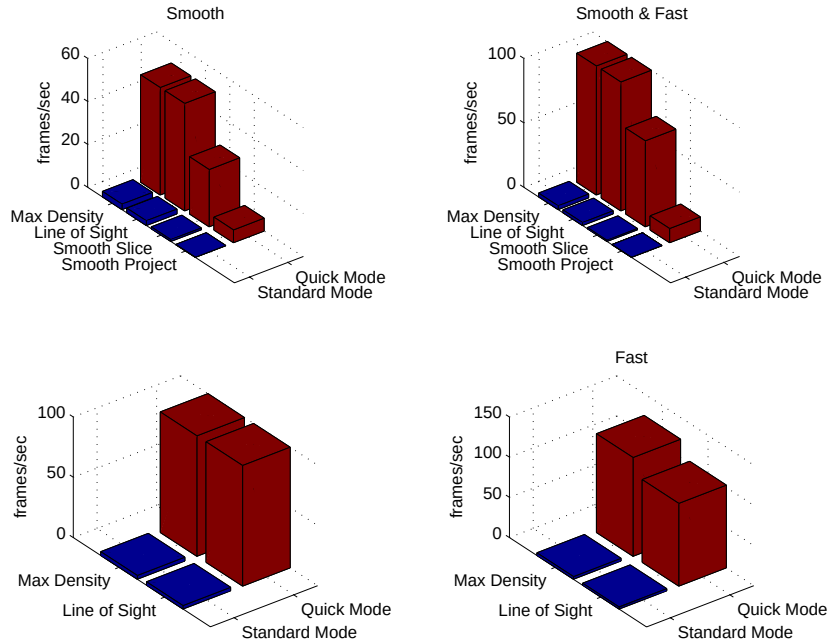

Figure 3.2: Serial Scaling of Hubble in a Bottle! for the MMCluster dataset

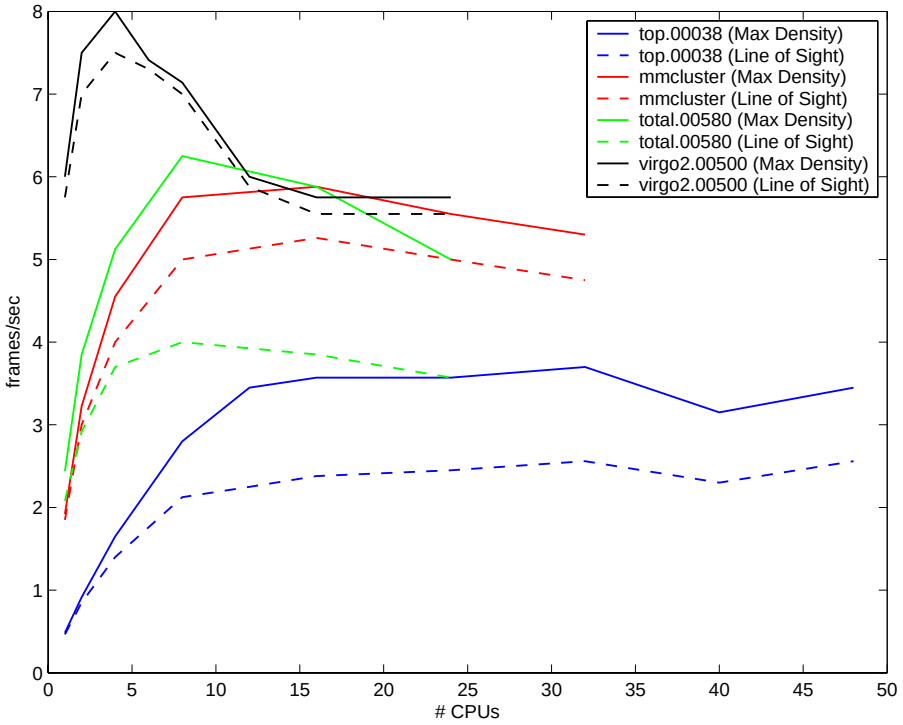

Figure 3.3: Max Density vs. Line of Sight Scaling

3.1 The Results

The performance of the serial version of Hubble in a Bottle! depends to a great extent on the pure number of particles. As one can see from Figure 3.1 and Figure 3.2, the results varying between 3.3–13.7 fps for inputs among 1.3 and 6.3 million particles, whereas the already mentioned Quick-Mode is always able to achieve values beyond 50 fps up to over 100 fps. One should keep in mind, that the default 50000 particles for the Quick-Mode are selected randomly, so that the results can vary somewhat from run to run.

From Figure 3.3 one can recognize, how the parallel version of Hubble in a Bottle! scales with the number of processors, and how the performance of the line-of-sight mode behave in comparison to the max-density mode. The conclusion is that the efficiency is starting at all sizes at about 75%, but is then constantly falling. This is of course to no surprise, since one has to increase the problem-size according to a isoefficiency function. The outcome of the isoefficiency principle is, that Hubble in a Bottle! has to switch to bigger problem sizes with increasing CPU numbers

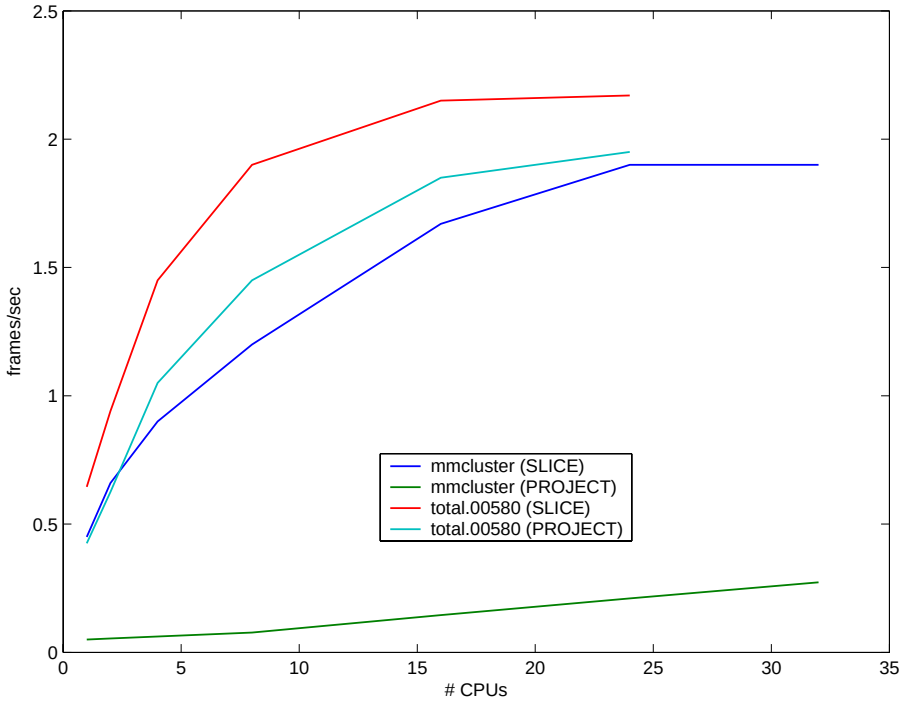

Figure 3.4: Smooth Scaling

to achieve a higher efficiency, which is in agreement with Figure 3.3. That it is not scaling beyond 16 CPUs, even in the case of 27 million particles, is a litte bit surprising, and will be discussed later.

One can also notice, that the line-of-sight mode is only by a constant addend more expensive than the max-density plot-mode. Additionally there are evidences, that the performance is somewhat smaller in case of CPU numbers unequal power of 2, an effect becoming more pronounced with increasing number of CPUs. The buckling of the blue curve of Figure 3.3 can be seen as a consequence of this speculation.

An increase of the computation to communication ratio as done here by switching to the smoothing-modes, leads to better scaling, as seen in Figure 3.4. It is also obvious, that the projection smooth-mode, has to have a higher order of N than the slice smooth-mode, and therefore a much higher dependance on the number of particles, to affirm the green line of Figure 3.4.

The Quick Scaling Figure 3.5 identifies why Hubble in a Bottle! isn't scaling well above 16 CPUs. If one consider the quick mode as an upper bound, which is reasonable, because by default 50000 particles are used in the quick mode, which can

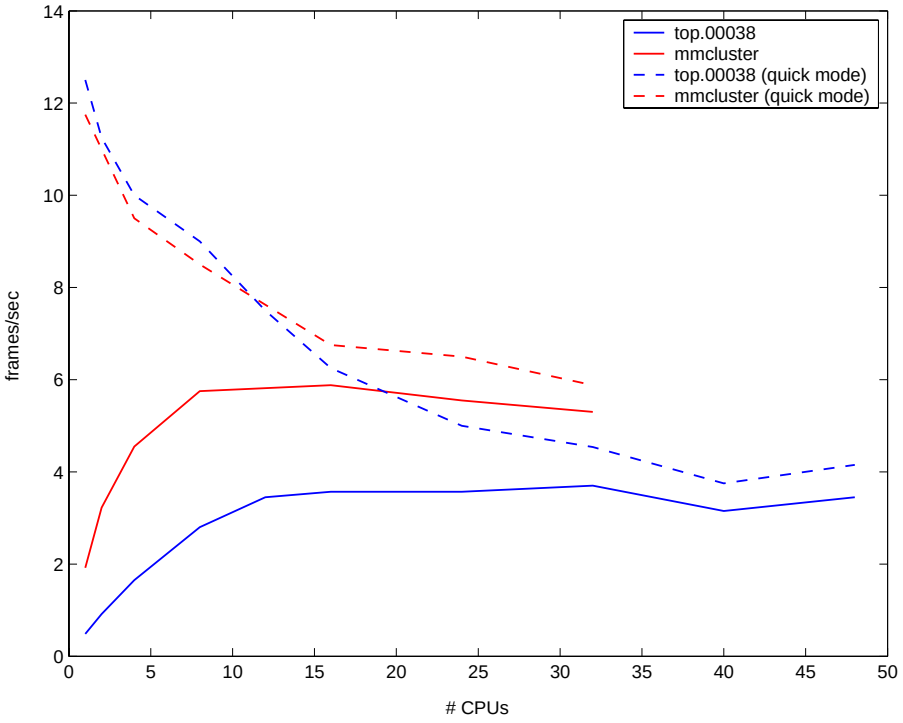

Figure 3.5: Quick Scaling

be transformed and projected even with one CPU instantaneously, the converging curves are a clear argument, that the computation is bounded by the communication.

The measured 256 MB/s point-to-point bandwith, should be good for 27 frames per second, assuming a 768-by-768 resolution and 4 bytes per pixel, since each bug has 2 input and 2 output channels. The best result for the parallel version achieved so far concerning throughput are 8.00 frames per second with turned off FAST mode (4 bytes per pixel) and 4 CPUs on the Virgo2 dataset, which is fairly precisely 3.375 times behind the theoretical peak performance. Because the bandwidth measurements were taken at zero load, and my benchmarks at maximum CPU and communication load, I conjecture that the point-to-point bandwidth is substantial lower, although I haven't tested it.

In the end one can only argue, that $2.7 * 10^7$, is by far not Hubble Volume Simulation size, but as one can recognize from Figure 3.3, Hubble in a Bottle! is scaling pretty well with increasing particle numbers, so that I'm quite confident

that Hubble in a Bottle! can visualize a few frames per second of a Hubble Volume Simulation dataset in real-time, especially on an exhausted zBox and more CPUs used.

3.2 Compiler Issues

When squezing out the maximum performance of a program, the compiler isn't just a simple translator, but rather a not to negligible repository of performance. The compiler used here was an actual version of the GNU gcc, specifically the version 3.4.0. Beside the obvious optimization-flag -O3, which consists of the compiler-flags of Table 3.1,

-fforce-mem	-foptimize-sibling-calls	-fstrength-reduce
-fcse-follow-jumps	-fcse-skip-blocks	-frerun-cse-after-loop
-frerun-loop-opt	-fgcse	-fgcse-lm
-fgcse-sm	-fgcse-las	-fdelete-null-pointer-checks
-fexpensive-optimizations	-fregmove	-fschedule-insns
-fschedule-insns2	-fsched-interblock	-fsched-spec
-fcaller-saves	-fpeephole2	-freorder-blocks
-freorder-functions	-fstrict-aliasing	-funit-at-a-time
-falign-functions	-falign-jumps	-falign-loops
-falign-labels	-fcrossjumping	-finline-functions
-fweb	-frename-registers	

Table 3.1: O3 compiler-flags

the following architecture specific and more sophisticated optimizations of Table 3.2 were used.

Intel Architeture Serial version for my Workstation	AMD Architecture Parallel Version for the zBox
-mtune=pentium4	-mtune=athlon-mp
-march=pentium4	-march=athlon-mp
-mcpu=pentium4	-mcpu=athlon-mp
-mmmx	-mmmx
-msse	-msse
-msse2	
-msse3	
	-m3dnow
-mfpmath=sse	-mfpmath=sse
-funroll-loops	-funroll-loops
-D_NO_MATH_INLINES	-D_NO_MATH_INLINES
-pg	-pg

Table 3.2: Additional sophisticated architecture specific compiler-flags

Alltogether this compiler-flags will get you an additional performance of about 60% above the gcc 3.4.0 with the standard optimization-flag -O2 and about 50% speed-improvement above the pre-installed Intel VTune compiler from 2002 with the highest optimization level, architecture optimizations, interprocedural optimization (IPO), profiling (PGO) and auto-vectorization. The proposed compiler-flags can be seen as at least nearly optimal. Even though it's not clear if compiler-flag tuning has any impact on the performance (Chan et. al. 1994 and Patterson et.al., 1995), or is even irrelevant (Pugh, 2001), I spend a whole weekend on compiler-flag tuning because to my mind speed-optimizations at large has to be approached in a more pragmatic manner.

I would find it further very interesting to give Static Single Assignment (SSA) for Trees a try (Cytron et.al., 1991 and Morgan, 1998), especially the auto-vectorization function based on SSA. SSA for trees is going to be integrated into gcc in the future, but one shouldn't overestimate the optimization capabilities of compilers, e.g. cache-locality are still the business of a system-programmer, as except of Sun's

Forte compiler (DeMone, 2002), compilers are not able to even recognize the swap of column-major versus row-major matrix representation.

For the forthcoming zBox2, I propose to recompile Hubble in a Bottle! with the Pathscale ECOPath compiler and especially with the further improved version 2.0, which is specifically designed for the 64-bit x86-64 architecture.

Chapter 4

Difficulties

Since Hubble in a Bottle! is laid out as a serial as well as a parallel version from the very beginning, the development got additional challenging. Beside that one has to create parallel code alone, one also has to deal with what is called Serial/Parallel Codesign, which implicates that everything has to be fully functional in serial and in parallel simultaneously.

4.1 What makes parallel programming that hard

What makes efficient parallel programming hard, is that an algorithm has to be resistant against race conditions, and this if possible without absolutely necessary syncronization barriers. A parallel computer is a non-deterministic computer, which means that it is executing the parallel processed commands everytime in a different order as a program is running on several CPUs concurrently. A parallel algorithm has to cover all these different execuction sequences, and since computers are performing certain code-parts millions of times, even a very unlikely case in which something can go wrong become almost certain. This requires a completely other way of thinking, to which probably only electrical engineers are used to, as operations on a digital or analog circuit are also performed fully concurrently.
To my mind one should also approach the issue of correctness after electrical engineering standards instead of computer science standards, because errors are in a parallel program much more likely and considerabely harder to detect.

4.2 Obstacles

All major obstacles were concerned with parallel programming and were emerging mostly in the parallel reduction-tree. As yet chatacterized the most difficult part was the correct timing of the parallel reduction-tree, since the buffers of the SCI-cards are rather limited with at the the same time very high data throughput. From this it follows, that the nodes aren't allowed to run too much out of syncronization, especially as no barriers are used for performance reasons. The solution I came up with, of sending zero-packages around to self-syncronize the parallel reduction-tree, isn't as easy as is it sounds at first glance. For example since the loop of the parallel reduction-tree adresses also the filling and the deflation of the pipeline, every node has to strictly take care when to feed in a packet into the pipeline and when not any longer. Additionally it's essential to assure that every two packets going to be merged in a node are of the same iteration, particularly because the delays in an interconnection system are potentially very volatile. On the other hand a reset of the local ValueMap of the last node is not necessary, because float numbers are like integers, even if in an not so obvious way, cyclic. The handling of partly filled packets had also be adressed carefully. Another profoundly difficulty appeared in the context of MPI_Irecv, the non-blocking receive. MPI_Irecv has to be called with a communication request handle, with has to be unique inside a single iteration, although this isn't explicit documented. A violation of this condition can lead to subtle mutually overtaking packets. In an earlier version of Hubble in a Bottle! this was catched by a sole remaining barrier, it took me days to conceive this and to remove the last MPI_Barrier command.

The last thing I want to mention wasn't actually an obstacle, but aroses from an syntax inconsistency of MPI. In Tiziano's version 0.336 of Hubble in a Bottle! only a forth of the actual particles were processed, which resulted in an essential performance decrease during development and should be considered while evaluating the achieved speed-up. This misadventure is rooted on the fact, that MPI accepts only it's own set of predefined types, whichever is recognizable at the MPI_ prefix. User defined struct types are not allowed at all. Because each particle consist of 4 floats, four times as much data has to be transfered, thus 4*N items of type MPI_FLOAT.

Chapter 5

Not realized features and other suggestions for improvement

This section is intended as a starting point for a potential developer. Beside a few words on the new processor of the zBox2, I want to propose three projects and try to highlight in which way they are interesting from the technical point of view and in which from the pragmatic user perspective. All projects aren't specific to Hubble in a Bottle!, but one can use it as a playground to explore these interesting concepts in general. Additional maintaining and improving the current code and documentation basis should be always on the list.
First of all one should get Hubble in a Bottle! running on the forthcoming zBox2. The details how to compile and run Hubble in a Bottle! as a serial and as a parallel version is described in detail in the attached readme file and on the user homepage.

5.1 Optimize Hubble in a Bottle! for the forthcoming zBox2

Tuning Hubble in a Bottle! for the zBox2 would be the next logical step and is also well suited to get into the code. In the first instance I'm thinking about readjusting the BUFFERSIZE variable, recompiling Hubble in a Bottle! with an optimizing compiler like the mentioned Pathscale ECOPath compiler and perhaps compiler flag

tuning.

The upcoming zBox2 is, as the original zBox, a massively parallel Cluster of Workstations, or COW in short. It is connectect in the approved 2D-Torus topology via the ATOLL high-performance interconnection system. For the CPUs 384 AMD Opterons are used, which are implementing the x86-64 instruction set, the 64bit extension of the never dying x86 instruction set. By the way it's a common misbelief, that 64bit processors are intrinsic faster than their 32bit analogons. As reported in (Stiller, 2005) turned on x86-64 support is only of very limited benefit, which isn't quite astonishing as, the pure extension of the word size hasn't any positive effect on the performance at all, as long as no doubles are used, what is the case here. Furthermore extending the word size to 64bit means, that only half of the instructions are fitting into the caches. The pretended performance benefit comes from the fact that x86-64 instructions are compiled with implicit turned on SSE2 optimization, which is also possible in the 32bit mode and already done for the binaries of the serial version of Hubble in a Bottle!, so that this argument is also not fully valid.

To prevent any misunderstandings, of course running Hubble in a Bottle! on the zBox2 will lead to a major speed improvement, but not as 64bit CPUs are used, but because the Opterons are also superb 32bit processors. Additionally they have a higher core frequency, have tremdously increased number of registers, double sized caches, an integrated dual channel memory controller and the serial point-to-point HyperTransport bus.

5.2 Dual-level parallelism

From the HyperTransport bus I'm promissing myself the most, since it makes enough bandwidth available to make dual-level parallelism possible, as it was already done in (Hutter and Curioni, 2003) for ab-initio molecular dynamics. Dual-level parallelism combines the two advantages of shared-memory and distributed-memory machines. The whole machine is seen as a distributed-memory machine down to node-level, where each node is considered as a shared-memory node. Up to present Hubble in

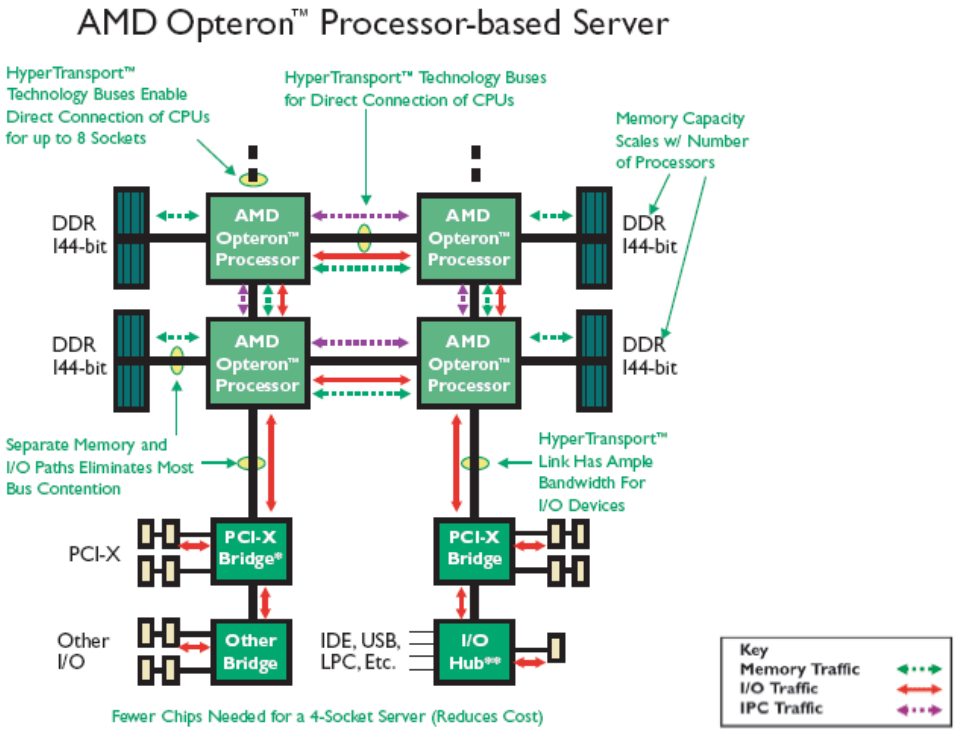

Figure 5.1: A zBox2 Node

a Bottle! is only suited for distributed-memory machines, so that extended support for shared-memory nodes have to be implemented using OpenMP. As one can see in Figure 5.1, a single zBox2 node is realized as a non-uniform memory access node, or abbreviated NUMA, which is kept coherent through at most three so-called coherent HyperTransport links per processor. It's fair to speak about shared-memory nodes, because the whole memory of a single node is combined into a single address space. For this reason NUMA machines are also called virtual shared memory machines (Boryczko et.al., 2003). To be fully accurate a single zBox2 node is a ccNUMA node, as the cache is also kept coherent through a broadcast-based SMP-like snooping protocol (MESI), in which bus snooping is simulated by explicit broadcasts of memory requests to all processors. Of course reduces the coherence traffic the connection bandwidth, but it's still far better than a SMP-node, since the bottleneck of a single shared bus is removed. This accessorily constitutes a principal predominance over the actual zBox.

Alltogether this is, the way I see it, the most effective, but also the most challanging

project I'm proposing, since it needs competences at several fronts, beyond parallel programming knowledge particularly proficiency at the operating system level and system administration skills. Additionally such a project stands and falls with the available vendor support for parallel programming APIs like MPI and OpenMP.

5.3 Speed Optimizations using SIMD Instructions

As the second project I'm proposing various speed optimizations for the critical sections of Hubble in a Bottle! using SIMD extensions, or even the GPU of the graphics card.

SIMD is an acronym for Single Instruction, Multiple Data, which was first coind by the former Stanford Professor Michael J. Flynn in his classification of parallel architectures (Flynn, 1996). It means that a single instuction is performed over multiple data and particularly concurrently. Array and vector processors are porminent members of this class.

5.3.1 x86 SIMD Extensions

In current x86 processors the SIMD principle is implemented using 64bit and 128bit registers respectively, in which multiple data are put in. These instruction set extensions turning the x86 processors into a form of vector processors. Tiziano already programmed optimized 3Dnow! and SSE routines, depending on which architecture Hubble in a Bottle! was executed, for the at that time critical matrix-matrix multiplication. In order to get rid of maintaining two different implementations, I used the common denominator of the Intel Pentium 4 and the AMD Athlon processor used in the zBox, and wrote my versions of the matrix-matrix multiplication in SSE. As already described, it later turned out that replacing the matrix-matrix multiplication through a vector-matrix multiplication is the much better idea, but the new routine is only written in optimized ANSI C.

My proposal is now to rewrite my routine in highly optimized SSE2, since SSE2 is becoming the new common denominator between the most modern processors of the Intel and AMD processor families, as the zBox2 went online. A very good step

by step instuction, how to speed up my version of a vector-matrix multiplication by about a factor of 3 or even 4, is available as a case study authored by (Stratton, 2002). This case study is very instructive and should be convinient even for novices in the area of SIMD programming.

The learned techniques and especially the memory prefetching could be also applied to other parts of Hubble in a Bottle! to further speed-up the program. The great thing about this small project is that it is quite focused to certain parts of the code, which makes this well suitable even for a very limited project, as the time it takes to familiarise oneself with the code is rather short.

For me this is a very neat assignment, the only drawback I see is that it is very uncertain, if this has an substantial effect on the performance of the parallel version, because the vector-matrix multiplication should be as far as possible overlaped by the communication if the BUFFERSIZE variable is properly choosen. The serial version in contrast should gain a major speed-up in any case.

5.3.2 GPU programming

Another place to perform SIMD operations is the GPU of the graphics card, which should be considered because of it's unchallenged price/performance ratio and the enormous evolution speed of the GPUs, that is even faster than Moore's law for CPUs. Since the introduction of DirectX9 and the associating GPUs implementing it in hardware, the development of GPUs has reached a new high-point with the adoption of single-precision 32bit floating-point capabilities and high-level shader languages such as Cg (Mark et.al., 2003), HLSL and the OpenGL Shading Language (Kessenich, 2004), as they facilitate the abstraction of current GPUs as stream processors.

As mentioned a GPU is the attempt to map as much as possible of the rendering pipeline (Figure 5.2) onto hardware, what is achieved more and more from generation to generation. For this reason a modern graphics card consist of a vertex as well as a pixel or fragment processing unit. The first one is processing triangles, whereas the latter is treating pixels or fragments, what are basically pixels with their associating colours. For historically reasons the two processing engines are

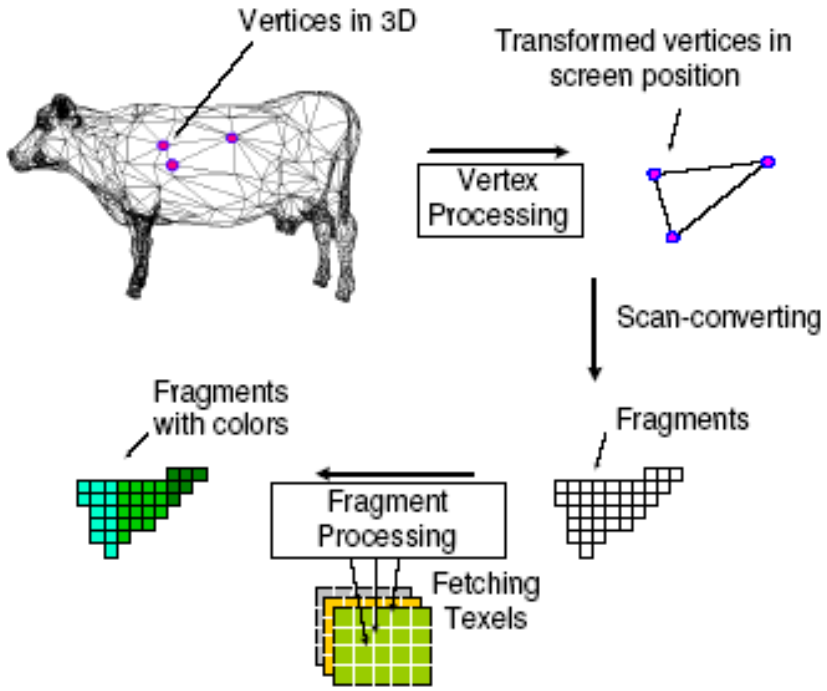

Figure 5.2: Rendering Pipeline

also called shaders.

Since DirectX9 the GPU can be programmed using conditional branches and loops, additionally the maximum number of instructions of a single fragment and vertex shader has been increased in such a way, so that it's became possible to implement general purpose computations as shader programs.

Current GPU hardware are supporting 4D-vectors for either homegeneous coordinates or RGBA color channels through a complete set of SIMD floating-point instructions on these vectors. All attemps (Larsen and McAllister, 2001) to perform non-graphical general purpose matrix calculations on the GPU were only exploiting the fragment processor, because fragment programs are able to access texels (texture elements) from random positions in the texture memory and are supporting gather operations, which is not the case for the vertex processor. Using Cg or any other high-level shader programming language, fragment programs can now be implemented in a C-like high-level language, instead of a kind of assembler.

But although such high-level shader languages are availble, GPU programming

is anything but easy, since (Hall et.al., 2003) showed that cache and bandwidth awareness are fundamental to achieve respectable performance results in matrix-multiplications. So far it is believed that a tremdous speed-up could be achieved in performing matrix-multiplications, even though nobody has really achieved them. Some authors (Moravanszky, 2003) were even struggeling to outperform general-purpose CPUs. In the meantime evidences suggests (Fatahalian et.al., 2004) that the bandwidth to main memory marking another eminent bottleneck, since GPUs are streaming processors which are only sustaining their peak performance if they are kept busy. It was reported by (Fatahalian et.al., 2004) that they were able to accomplish the same performance as a current high-end CPU, but only with a state of the art graphics card and that they were not achieving loads beyond 20%. Furthermore they proved that this couldn't be improved, as the main memory bandwidth limit is fundamental. It's therefore indistinct, if switching the matrix computation to the GPU, is a good idea at all.
The idea of a GPU Cluster for High Performance Computing as published by (Fan et.al., 2004), is only reasonable if the work/bandwidth ratio of the GPU is higher than for the matrix-computation alone. This could be perhaps achieved by letting the GPU performing even more work of Hubble in a Bottle!, as for example the projection could be done by the vertex processor, but this is also substantial harder to achieve, since only very unefficient mappings of point-based data to vertex shaders are known.

5.4 Additional functions and features

The next area, where it's worth to put extra effort into, is the never-ending field of additional functions and features.
I'm thinking in this regard about a parallel read in, as it was realized in the PKD-GRAV code. Parallel read in means, that all involved nodes are reading their own chunk of data from the input file concurrently. In addition the distribution of file pieces over the interconnection system is not necessary anymore. Code snippets and alpha version for such a parallel read in is already available.

Another nice to have feature would be the capability to write screen shots into files. Tiziano already wrote correct looking functions to write screen shots into bitmap files, but as yet it frankly didn't worked at all, and I also wasn't able to figure out what was wrong. Furthermore the support of a lossless compressing image format, like the freely available PNG format would be highly desirable, since bitmap files are getting very large. Even better would be moreover a scripting interface as it's available for TIPSY to compose whole videos made up of single images, since putting them together by individual screen shots would be far too cumbersome.

Working with interfaces is in general a good idea. For example one could think of a general programming interface, where one could connect self-programmed plug-ins. To proceed a CVS folder for such plug-ins would be an opportunity, so that already programmed add-ons could be shared.

Chapter 6

Conclusion

In this thesis I dealt largely with the parallelization and speeding-up of Hubble in a Bottle!. The original goal to make live flythroughs across huge datasets possible wasn't achieved. Only for the serial version and for datasets in the range of one to two millions the at least necessary 15fps were reached on a high-end workstation. However I showed that the frame rate of the parallel version is limited by the bandwidth and even more by the latency of the interconnection system. The scalability to bigger datasets in contrast is much more promissing and allows the visualization of tens of millions particles in real-time at a particle throughput of over one hundred million particles. In addition there are evidences that this trend could be continued to some extend to even bigger datasets.

During the development at Hubble in a Bottle! an innovative parallel reduction-tree, a consistent way to broadcast user-interactions in a parallel environment and the integration of the Tipgrid code were explored.

In the addendix a novel expansion of the CORDIC algorithm to take logarithms efficiently is described, which is going to be implemented in the scaling routine.

Acknowledgements

I would like to thank Dr. Joachim Stadel and Prof. Thomas Quinn for assistance and many fruitful discussions, and Doug Potter and Dr. Roland Bernet for helping me with several Linux and zBox issues.

A special thanks goes to Tiziano Mengotti for his spadework and his explanations.

Bibliography

[1] G. Lake, T. Quinn, D.C. Richardson and J. Stadel (2004), *The pursuit of the whole NChilada: Virtual petaflops using multi-adaptive algorithms for gravitational systems*, IBM Journal of Research and Development, **48**, pp 183–197.

[2] A. Jenkins, C.S. Frenk, S.D.M. White, et.al. (2001), *The Mass Function of Dark Matter Haloes*, Monthly Notices Roy. Astronom. Soc., **77**, 372.

[3] Y. Chan, A. Sudarsanam and A. Wolfe (1994), *The effect of compiler-flag tuning on SPEC benchmark performance*, ACM SIGARCH Computer Architecture News, **22**, Issue 4, pp 60-70.

[4] N. Mirghafori, M. Jacoby and D. Patterson (1995), *Truth in SPEC benchmarks*, ACM SIGARCH Computer Architecture News, **23**, Issue 5, pp 34-42.

[5] B. Pugh (2001), *Is Code Optimization Research Relevant?*.

[6] P. DeMone 2002, *Looking Forward to 2002*, RealWorldTech.com.

[7] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck 1991, *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph.*, ACM Transactions on Programming Languages and Systems, **13**, 4: pp 451-490.

[8] R. Morgan 1998, *Building an Optimizing Compiler*, Butterworth-Heinemann.

[9] N.J.A. Sloane 2003 *On-Line Encyclopedia of Integer Sequences*, http://www.research.att.com/ njas/sequences/Seis.html.

[10] J. Hennessy and D. Patterson 2002 *Computer Architecture A Quantitative Approach, Third Edition*, The Morgan Kaufmann Series in Computer Architecture and Design.

[11] A. Stiller 2005 *64-Bit-Werkstatt: Software-Entwicklung und SPEC-Benchmark-Ergebnisse unter Windows XP x64*, c't magazin für computer technik, **5/2005**, pp 118-123.

[12] J. Hutter and A. Curioni 2003 *Dual-level Parallelism for ab-initio Molecular Dynamics: Reaching Teraflop Performance with the CPMD Code*, IBM Technical Report RZ 3503, 2003.

[13] K. Borycko, W. Dzwinel and D.A. Yuen (2003), *Modelling Heterogeneous Mesoscopic Fluids in Irregular Geometries using Shared Memory Systems*, Submitted to Journal of Concurrency: Practice and Experience, August 2003.

[14] M.J. Flynn and K.W. Rudd (1996), *Parallel Architectures*, ACM Computing Surveys, **28**, No 1, March 1996.

[15] C. Stratton 2002, *Optimizing for SSE: A Case Study*, http://www.cortstratton.org/articles/OptimizingForSSE.php.

[16] W.R. Mark, R.S. Glanville, K. Akeley and M.J. Kilgard (2003), *Cg: A System for programming graphics hardware in a C-like language*, ACM Transactions on Graphics (SIGGRAPH), **22**, 3: pp 896-907, 2003.

[17] J. Kessenich, D. Baldwin and R. Rost 1004, *The OpenGL Shading Language*, 2004.

[18] E.S. Larsen and D. McAllister (2001), *Fast Matrix Multiplies using Graphics Hardware*, In Proceedings Supercomputing, 2001.

[19] J.D. Hall, N.A. Carr and J.C. Hart (2003), *Cache and bandwidth Aware Matrix Multiplication on the GPU*, UIUC Technicakl Report UIUCDCS-R-2003-2328, 2003.

[20] A. Moravanszky 2003, *Dense Matrix Algebra on the GPU*, 2003.

[21] K. Fatahalian, J. Sugerman and P. Hanrahan (2004), *Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication*, Presented at Graphics Hardware 2004.

[22] Z. Fan, F. Qiu, A. Kaufman and S. Yoakum-Stover (2004), *GPU Cluster for High Performance Computing*, ACM/IEEE Supercomputing Conference 2004, November 6-12, Pittsburgh, PA.

[23] J.E. Volder (1959), *The CORDIC computing technique*, IRE Transactions on Electronic Computers, vol. EC-8, No 3, pp 330-334, September 1959.

[24] C.W. Schelin (1983), *Calculator function approximation*, AMS Monthly, pp 317-325, May 1983.

[25] J.S. Walther (1971), *Aunified algorithm for elementary functions*, In Processdings of AFIPS 1971 Spring Joint Computer Conference, vol. 38, AFIPS Press, Arlington, Va., pp 379-385, 1971.

[26] T.C. Chen (1972), *Automatic computation of exponentials, logarithms, ratios and square roots*, IBM Journal of Research and Development July 1972.

[27] T. Lang and E. Antelo 2005, *High-Troughput CORDIC-Based Geometry Operations for 3D Computer Graphics*, IEEE Transactions on Computers, vol. 54, no. 3, March 2005.

[28] D.E. Knuth (1997), *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, 1997.

Appendix A

A not so new logarithm algorithm

The algorithm I'm going to describe belongs to the family of CORDIC algorithms. CORDIC is an acronym that stands for COordinate Rotation DIgital Computer and goes back to Volder (Volder, 1959). In the early 1980s, this method was used by Hewlett-Packard (Schelin, 1983) in their HP-35 calculator to evaluate trigonometric functions.

The power of CORDIC-like algorithms is due to the fact, that only shifts and adds are necessary, and no multiplications or divisions like in series expansions have to be performed. Therefore in spite of merely linear convergence, it even outperforms floating-point techniques with quadratic convergence.

The CORDIC concept has been extended and further developed by (Despain, 1974) to perform the Discrete Fourier Transform. Later Walther (Walther, 1971) unfied the CORDIC theory, which was then applied by (Chen, 1972) to compute exponentials, logarithms, ratios and square roots. Beyond this in a recent publication (Lang and Antelo, 2005) reported about how to use CORDIC-based algorithms to perform high-performance 3D geometry operations as a alternative for the quaternion representation. They also gave a brief outline to a CORDIC-based stream processor. CORDIC algorithms at large have been at all times implemented in hardware, since it's structure is inherent suitable to get implemented in hardware.

A.1 The Basic Idea

The basic idea is to decompose the input into the following product

$$\prod_{k=0}^{\infty}\left(1+\frac{1}{2^k}\right)^{d_k}, with \quad d_k=\{0,1\} \tag{A.1.1}$$

Such a decomposition is distinct, which was first proven by Richard Feynman, but never published. The logarithm of this product is identical to the sum of the logarithms of the single terms, thus $\sum_{k=0}^{\infty} d_k \log_b(1+\frac{1}{2^k})$. The logarithms are stored in a precomputed lookup table with as many entries as the desired accuracy in bits. To get an accuracy of $10^{-16} \simeq 2^{-53}$, what corresponds to double precision accuracy on a computer, only 53 iterations are required and even 24 to reach single precision, instead of tens of thousands, which are necessary for other iterative methods like e.g. series expansions.

The algorithm uses only additions, multiplications by a power of two ('shifts') and comparisons, and gains approximately one bit more precision per iteration.

A real novelty of this algorithm is the stopping criterion, which is being based on the property $0 \leq x - x_k \leq 2^{-k+1}$, which is equivalent to $2^{-k+1} > \epsilon$.

A.2 The Actual Algorithm to take Logarithms

`input:` $x \geq 1$; `precision` ϵ

`output:` $\log_b(x)$ `approximated accurately by` $x_k = t_k$

`initialize:` $t_0 := 0;\ e_0 := 1.0;\ k := 0$

`while` $2^{-k+1} > \epsilon$ `do`

$$d_k := \begin{cases} 1, if\ e_k(1+\frac{1}{2^k}) \leq x \\ 0, otherwise \end{cases}$$

$$t_{k+1} := t_k + d_k \log_b(1+\frac{1}{2^k})$$

$$e_{k+1} := e_k(1+d_k\frac{1}{2^k}) = e_k + d_k\frac{1}{2^k}e_k$$

$$k := k+1$$

`end loop`

An invariant of this algorithm is that $\log_b(e_k) = t_k = x_k\ \forall k$.

The presented algorithm converges for any $x \geq 0$, $x \neq 1$, assumed the basis b satisfies $b \geq 0$ and $b \neq 1$. Above all it solves problem 1.2.2-28 of Donald Knuth's book "The Art of Computer Programming, Volume 1" (Knuth, 1997), which is credited to Richard Feynman.

A.3 The Expansion to Compute the Power

The same algorithm can be used to compute the power, by interchanging the in- and output. This changes the above invariant $\log_b(e_k) = t_k$ into $b^{t_k} = e_k$, so that taking the logarithm of $e_k(1 + \frac{1}{2^k})$, to get an algorithm for the inverse operation, results in $t_k + \log_b(1 + \frac{1}{2^k})$ using the invariant identity. The above precomputed lookup table don't has to be changed and could be reused.

`input:` $x \in \mathbf{R}$ `; precision` ϵ

`output:` b^x `approximated accurately by` $x_k = e_k$

`initialize:` $t_0 := 0;\ e_0 := 1.0;\ k := 0$

`while` $2^{-k+1} > \epsilon$ `do`

$$d_k := \begin{cases} 1, \textit{ if } t_k + \log_b(1 + \frac{1}{2^k}) \leq x \\ 0, \textit{ otherwise} \end{cases}$$

$t_{k+1} := t_k + d_k \log_b(1 + \frac{1}{2^k})$

$e_{k+1} := e_k(1 + d_k \frac{1}{2^k}) = e_k + d_k \frac{1}{2^k} e_k$

$k := k + 1$

`end loop`

An invariant for this algorithm is $b^{t_k} = e_k = x_k\ \forall k$.

It converges for $x \in \mathbf{R}$, given $b > 1$.