Teaching parallel programming to undergrads with hands-on experience

Rainer Keller
Hochschule für Technik Stuttgart, University of Applied Science
Stuttgart, Germany
Email: rainer.keller@hft-stuttgart.de

***Abstract*—This paper describes the didactic concept, the content and the lessons learned of a lecture on parallel programming for undergraduate students held during summer term 2013. The course's focus was on providing hands-on experience, hence students were programming on real life codes using an actual HPC cluster. The lecture's aim was to provide an in-depth understanding of each parallel programming paradigm used in industry, starting with OpenMP, MPI and OpenCL as an example of a general approach for accelerators. Additionally, MapReduce based on Apache's Hadoop was introduced as a new and different way of parallelization. The course's strong points are highlighted, such as useful practicals on real life codes.**

***Keywords*-OpenMP, MPI, OpenCL, MapReduce**

I. Introduction

It has been mentioned in various publications, e. g. [1], that today's Supercomputing and HPC in general is the basis for future advance in technology in industry. Multicore processors nowadays are omnipresent, in our student's mobile phones and laptops – even processors of engine-management systems have multiple cores to process data from various sensors in the car. This field of engineering is particularly strong in the Stuttgart area [1].

Multicore chips therefore are getting commonly used at future employers of our University's graduates. Hence, parallel and distributed computing (PDC) has become a required skill-set for our Computer Science majors. This summer term, a course on parallel programming and HPC was taught at our university, with a strong focus on gaining hands-on experience, as is typical for our curriculum.

According to the categorization in [2], this course covered a broad set of three of the four topics: architectures, applicability (in Bloom's level of coverage) of programming topics and the latter topics within the algorithm level. This paper describes the course's content, actual exercises, experience gained using these examples, and some methodologies which had positive results. Section II describes the prerequisites and didactic concept, while Section III provides an overview of the infrastructure used to allow students hands-on experience, Section IV presents the topics and actual examples used to study the various parallel programming concepts. Section V concludes the paper, providing lessons learned and techniques to be used in subsequent lectures.

[1] Among the companies: Mercedes, Porsche, Bosch and Vector Informatik

II. Didactic concept

Our CS students start with Java as first language, and are trained to be good, high-level software developers. This course was taught to 3^{rd} and 4^{th} semester students, i. e. their 2^{nd} year; C/C++ are taught the very same semester. Hence, the learning curve (low-level, hardware-oriented C, accessing HPC-machines using modules and shell-scripts, queueing jobs, etc.) was quite steep. The 13 lectures were split in such a way that emphasis was given to the "hard" topics, such as MPI and OpenCL. Overall, the topics were Parallel Programming introduction (two lectures), OpenMP (two), MPI (three), OpenCL (three), MapReduce, Tools for Parallel Programming and finally Recapture and wrap-up.

As the focus was on getting programming experience, each lecture of four consecutive 45 minute slots was split roughly into half: two slots of slide presentation, then two slots of practicals. Unfinished work was left as home-work.

Each lecture started with a questionnaire of five to ten questions on the content of last week's lecture, which were answered on paper and collected, but not graded. The questions were then reviewed in the auditorium and (mostly) orally answered by the students themselves, however animated slides were prepared and shown in class.

Presentations were mainly done as PowerPoint slides, however for a shift in media an overhead projector was used. Recurring easy and medium technical questions (which were subsequently presented in red color but deleted in the distributed PDFs) were injected into the slides to engage students. Activation worked really well whenever students were supposed to explain a technical issue to their neighbor [2].

All slides, including answered questionnaires, the templates and software packages for exercises were distributed using the University's Moodle platform [3]. Of Moodle's abundant features for student-teacher interaction (e. g. Chat, Forum and Survey) only Quickmail and Glossary were used. Moodle's Glossary feature at first was used to describe common HPC terms (CPU vs. core, compute node, etc.) however was quickly abandoned due to lack of time and interest on behalf of the audience. In other courses we have gained mixed experience with Moodle's features Quiz, Wiki and Survey.

[2] For example: "What would application programmers need to do, if MPI would not guarantee message order between two distinct processes."

To better understand the actual HPC architectures, a half-day field trip was organized to HLRS center and experience HPC machines in person. This enabled students to not only get a feeling for such a machine, but also get the impact of the science and usage of Supercomputing. (see Sec. IV-B).

III. HPC Infrastructure

The course made use of the BW-Grid cluster. The BW-Grid is infrastructure installed at nine Universities of the state of Baden-Württemberg, consisting of a total of 1750 nodes installed in 2008. In 2013, the HfT Stuttgart with additional funding from the Ministry for Science, Research and Art acquired 24 HPC compute nodes and installed these into the existing HPC cluster at the University of Applied Science in Esslingen. This HPC cluster was installed in 2010. Taking advantage of existing infrastructure, knowledge and most importantly by replacing nodes being integrated into the existing InfiniPath IB network, all users of BW-Grid may take advantage of the new hardware.

The clusters within BW-Grid feature the same software setup, using the same OS and the same Linux kernel [3]. Using Oscar modules [4], the BW-Grid offers more up-to-date versions of common software and offers a point of differentiation between the sites to accommodate the need of the site's user community.

The newly installed nodes comprise dual-socket eight-core Intel Xeon E2650 CPUs with 20 MB L3-cache with three channels into the socket's memory. Per node, 64 GB of ECC-DDR3 memory and a 120 GB SSD for local storage and fault tolerance research are installed.

Every student enrolled was allowed access to the cluster and to the new nodes using their own accounts, copying course-relevant data from a common Lustre workspace. The new nodes were reserved for the lecture with a special queue. Execution of large processes on the head-node is discouraged, but not prohibited. Offenders were nevertheless identified, notified and encouraged to use the proper PBS job scheduling, as documented in the course material.

Having access to this new hardware, the students for the first time were able to see the effect of processor caches on application performance: the Stream benchmark [5] was extended to use the processor's high-resolution time measurement functionality [4], added functionality to clean caches between runs and warm the caches.

Thereby we compared old Intel Nehalem-based nodes to new Intel Sandy Bridge ones. The results may be seen in Fig. 1 with additional visualization of the theoretical maximum memory bandwidth per cache-level. The benchmark was run with one thread, only, and best times are reported. The bandwidth to the L1, L2 and L3 cache match (to some extent) the reported theoretical limit.

[3] Scientific Linux 5.5 and kernel v2.6.18-348.

[4] Intel and AMD processors offer a time-stamp counter (TSC), incremented with constant frequency, independent of the processor's ACPI state.

IV. Parallel Programming Content

The first lessons started bottom-up with an introduction into the architecture of modern processors, Moore's law, examples of compute nodes as building blocks for HPC, the current Top500 systems [6] and giving an introduction into the architecture of the (in comparison tiny) BW-Grid system we start off with.

An early example for parallelization was the old, well-known Commodore C64 and it's diskette drive, which featured the same MOS 6502 processor as the main board itself. This disc's processor was used by a Fractal program to offload computation. To students, this provided an early example of parallelization using commodity hardware, which they could relate to.

The theoretical part included the observation of Moore's law, an introduction and comparison of Amdahl's law [7] with Gustafson's law [8]. The introduction into modern CPUs laid the basis to explain the concept of pipelining, multiple units for superscalarity, "vector" instructions and of course the necessity as well as the boon and bane of CPU caches.

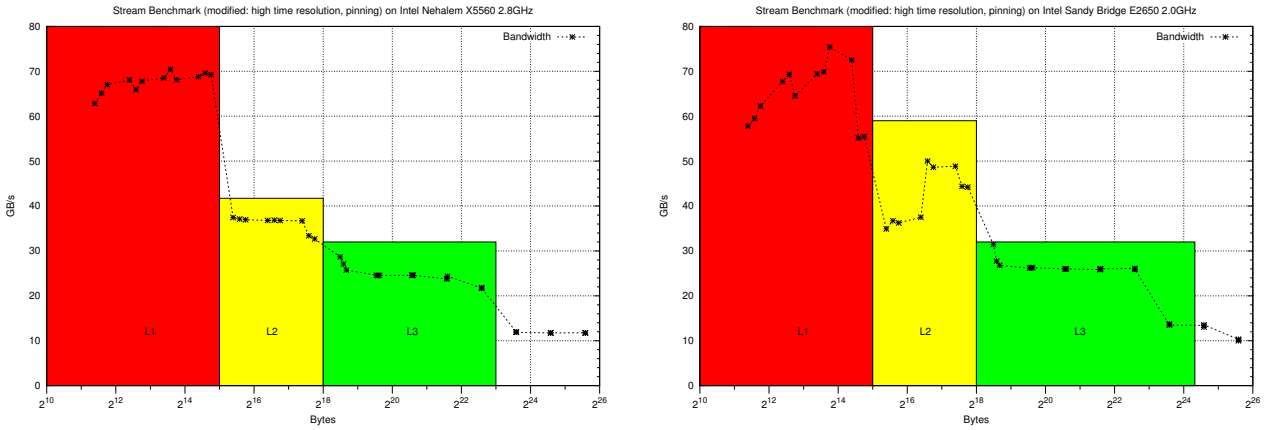
In order to start off with an easy example, a Monte-Carlo method (MC) for computing Pi was introduced. Here, the influence of compiler optimizations could be seen, as well as the parallelization using PThreads [9]. This provided a good example of *non*-thread-safe functions: the `rand` function used to select a new candidate in MC had to be replaced with it's thread-safe equivalent `rand_r`.

A. OpenMP

In comparison to PThreads, OpenMP is a rather easy programming paradigm. To relate to the previous exercises, the theoretical part included formal definitions of race-conditions, historic problems, such as `errno` [5], as well as deadlocks and circular dependencies using the famous Dining Philosophers problem. To demonstrate this effect, it was acted out by a group of students to substantiate the time-critical aspect of the circular dependencies – an activation which was both fun and efficient. Various synchronization methods were introduced – Java's `synchronized` directive was shown to not help with circular dependencies. The OpenMP introduction started with a typical linear path across the standard's parallel fork-join model, all workshare directives, ways and best practices to define data visibility, synchronization directives and finally OpenMP's API of function calls.

Since OpenMP is targeting scientific applications (which mainly are loop-based), the before-mentioned Pi example provided a good starting case to show the simplicity of OpenMP. To extend the exercise, the GNU Image Manipulation Program (GIMP) [10] was selected as a real life

[5] Traditionally, Unix declares the error variable `errno` as globally visible integer – which is not thread-safe. Linux' solution is to define a macro, selected when compiling with `_REENTRANT`, to use a per-thread variable.



(a) Intel Nehalem nodes

(b) Intel Sandy Bridge nodes

Figure 1. Stream Triad benchmark to visualize cache effects on the old and new compute nodes.

code example. GIMP's source code is large, modular, well structured and the various filters and plugins are encapsulated and compiled into stand-alone programs executed from the main GUI[6].

GIMP itself is not parallelized. There is an effort to use OpenCL in the underlying GEGL library – however, this will parallelize basic, often used graphics operations, not the actual plugins and effects, such as oilify, ripple or even blur. The latter being simple operations, we concentrated on the oilify effect, which is more complex (code- and computational-wise).

The main issue to solve in this exercise was to first identify the main computational loop in the plugin (in fct. `oilify`) and to then understand the loop structure and data access. To efficiently handle images, GIMP plugins are expected to work on the source and destination images in so-called tiled regions. The outermost loop as may be seen in Fig. 2 iterates over these regions – and is not parallelizeable, since tiles are iteratively retrieved out of the tile cache and handled in a non-thread-safe way.

The inner two loops over X and Y within a tile build a histogram over a so-called mask (of user-selected size). This histogram is then used to write into the destination tile. Close inspection of the memory access patterns shows:

- the histogram buffer has to be thread-local,
- the loop structure has to be normalized to abide to OpenMP's canonical loop form,
- therefore, one may not use combined `parallel for` workshare directives, but rather open a parallel region, initialize variables, and then distribute the work,
- some helper variables (such as `drow`), which are incremented at the end of each loop depend solely on the loop-variables and can be scrapped.

[6]Gimp-2.8.4 has 902 thsd. lines-of-code alone in C in 1600 files, without the various dependent libraries such as gtk, glib and GEGL.

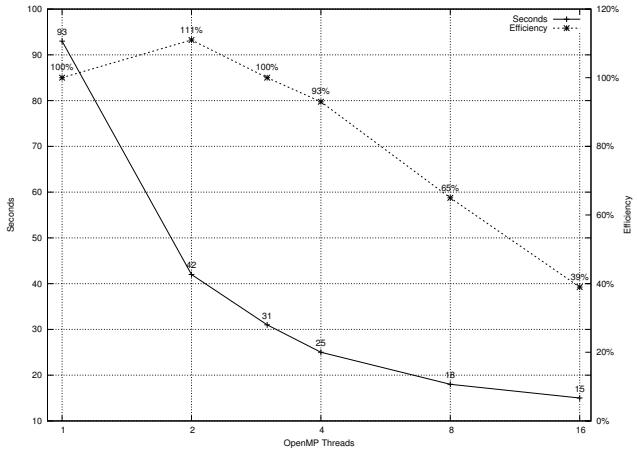


Figure 3. Execution time of the plugin using OpenMP (size: 3888x2592, mask size: 30 pixel). Efficiency (on right) relative to one OpenMP thread.

The transformation of the original loop in Fig. 2a to the parallelized version is shown in Fig. 2b. The diff-based patch is only 124 lines long – including the slider in the GUI for selection of the number of threads. The most intricate part with regard to parallelization was to first define OpenMP's `default(none)`, identify all of the variables to be declared private, and finally replace the loop-carried dependent helper variables. Overall the addition of a slider element to the GUI was the most time-consuming change.

This parallelized version was used on a example image, originating from a typical 10 MPixel camera. This image was used in all comparisons and execution benchmarks. To compile for parallel execution, the standard optimization flag `-O2` were only amended by adding gcc's compiler flag for OpenMP `-fopenmp`. The results of running the plugin on a new compute node of BW-Grid using the example image may be seen in Fig. 3. The runtime of the plugin was

```
for(pr=gimp_pixel_rgns_register2(n_regions, regions);
    pr!=NULL;
    pr=gimp_pixel_rgns_process(pr)) {
  guchar *drow, *src_msmap_row, *src_emap_row;

  for(gint y=drgn.y,
      drow=drgn.data,
      src_msmap_row=mask_size_map_rgn.data,
      src_emap_row=exponent_map_rgn.data;
      y < (drgn.y + drgn.h);
      y++,
      drow+=drgn.rowstride,
      src_msmap_row+=mask_size_map_rgn.rowstride,
      src_emap_row+=exponent_map_rgn.rowstride) {
    guchar *dest, *src_msmap, *src_emap;
    /* Same for-loop within region in X-dir. */
    { ...
      for(mask_y=mask_y1,
          srow=sbuf + soffset * bpp,
          sinten_row=sinten_buf + soffset;
          mask_y < mask_y2;
          mask_y++,
          srow+=width * bpp,
          sinten_row+=width) {
        guchar *src, *sinten=NULL;
        gint mask_x, dy_sqr=sqr_lut[ABS(mask_y-y)];
        /* Same for-loop within mask in X */
        { ...
          gint dx_sqr=sqr_lut[ABS(mask_x-x)];
          ...
          /*  Stay inside a circular mask area  */
          if((dx_sqr + dy_sqr) > radius_sqr)
            continue;
          ...
          for(gint b=0; b<bpp; b++)
            ++Hist_rgb[b][src[b]];
        }  /* for mask_x */
      }  /* for mask_y */
      for(gint b=0; b<bpp; b++)
        dest[b]=weighted_avg_val(Hist_rgb[b], exp);
    }  /* for x */

  }  /* for y */
  gimp_progress_update(progress/max_progress);

}  /* for pr */
```

(a) Original source code (shortened)

```
whole_start=omp_get_wtime();
for(pr=gimp_pixel_rgns_register2(n_regions, regions);
    pr!=NULL;
    pr=gimp_pixel_rgns_process(pr)) {
  guchar *drow, *src_msmap_row, *src_emap_row;
#pragma omp parallel num_threads(ovals.num_threads) \
            private(y,src_msmap_row,src_emap_row)
  {
  src_msmap_row = mask_size_map_rgn.data;
  src_emap_row = exponent_map_rgn.data;
#pragma omp for
  for(gint y=drgn.y;
      y < (drgn.y + drgn.h);
      y++) {
    guchar *dest, *src_msmap, *src_emap;
    /* Same for-loop within region in X-dir. */
    { ...
      for(mask_y=mask_y1,
          srow=sbuf + soffset * bpp,
          sinten_row=sinten_buf + soffset;
          mask_y < mask_y2;
          mask_y++,
          srow+=width * bpp,
          sinten_row+=width) {
        guchar *src, *sinten=NULL;
        gint mask_x, dy_sqr=sqr_lut[ABS(mask_y-y)];
        /* Same for-loop within mask in X */
        { ...
          gint dx_sqr=sqr_lut[ABS(mask_x-x)];
          ...
          /*  Stay inside a circular mask area  */
          if((dx_sqr + dy_sqr) > radius_sqr)
            continue;
          ...
          for(gint b=0; b<bpp; b++)
            ++Hist_rgb[b][src[b]];
        }  /* for mask_x */
      }  /* for mask_y */
      for(gint b=0; b<bpp; b++)
        dest[b]=weighted_avg_val(Hist_rgb[b], exp);
    }  /* for x */
    src_msmap_row += mask_size_map_rgn.rowstride;
    src_emap_row += exponent_map_rgn.rowstride;
  }  /* for y */
  gimp_progress_update(progress/max_progress);
  }  /* pragma omp */
}  /* for pr */
whole_stop=omp_get_wtime();
```

(b) Restructured with OpenMP

Figure 2. Source code of Gimp's `oilify()` internal loop structure

measured with OpenMP's functions `omp_get_wtime()`. The execution was done on various x86-64 based machines, however here only the runtime of the plugin with a default mask size of 30 pixel on the new compute nodes with one to 16 threads is shown. The left scale of Fig. 3 shows the time, while the right scale shows the efficiency in comparison to the sequential run. Both the parallel run with one thread and the sequential run took 93 seconds (the baseline for 100%). With eight threads, the execution took only 18 seconds (65% efficiency), while using all available cores with a runtime of 15 seconds only achieved 39% efficiency. It is noteworthy, that having only one thread on each socket, one may take full benefit of the processor's caches and memory bandwidth, due to the dual-socket nodes. Therefore one may achieve superlinear speedup [11], i. e. an efficiency of 115%.

B. MPI

The author has been giving workshops for MPI and OpenMP. The technique to teach MPI does not deviate much from the expected, and in this paper is therefore kept short.

Before diving into the many library calls of MPI, the students were introduced to different historic approaches to message passing, and the machines underlying network technologies. The multitude of platforms at HLRS allowed a practical introduction to different network topologies based on existing hardware:

- Cray XE6 Hermit system uses a 3D Torus topology based on the Cray Gemini interconnect,
- the NEC Cluster uses IB DDR as fat-tree topology,
- the NEC SX-9 vector machine uses a crossbar switch,
- the BW-Grid cluster uses a 3D Hypercube topology using Infiniband QDR HCAs.

During the excursion week the author organized a trip to the High-performance Computing Center Stuttgart (HLRS) where the students were a talk introduced the various machines and the research done on these. Most importantly, the students were able to experience these machines in person, connecting the theory of network topologies to the actual hardware [7], getting a grip on the dimensions, and a feeling for return-on-investment for tax-payer's money.

The Message Passing Interface is a library-based approach, offering a variety of communication, I/O and synchronization methods. Due to this variety, MPI-3 in total defines 437 function calls, many of which are rarely used. Although not advisable, it is not uncommon to see complex simulation software with only six MPI functions (`MPI_Init`, `MPI_Comm_rank`, `MPI_Comm_size`, `MPI_Send`, `MPI_Recv`, `MPI_Finalize`). Source code of such type was demonstrated to students – and reasons discussed, why and where the further, high-level communication routines do make sense.

Furthermore, there are multiple implementations available, whose differences and advantages are not obvious to beginners. In this course, we used Open MPI [12] on the cluster and aided students through installation on their private laptops on Linux, Windows and Mac OSX, allowing the students to work locally.

The first of three lectures handled the compilation, runtime and point-to-point communication calls, including concepts such as communicators, buffering of messages and the need for asynchronous communication. The second and third lecture introduced MPI datatypes as well as all the main collective communication patterns (`MPI_Barrier`, `MPI_Bcast`, `MPI_Gather`, `MPI_Scatter`, `MPI_Alltoall`, `MPI_Reduce`, `MPI_Reduce_scatter` and `MPI_Scan`) as well their `All-` and `v`-variants. Writing own operations for reductions trained in an exercise to compute the 1-norm of a buffer. In order to relate to the previous thread-centric lectures, MPI and thread-safety was very roughly introduced theoretically (using `MPI_Init_thread()`, different ways to isolate communication, using separate duplicated communicators), as well as in the exercises (see Sec. IV-E on tools).

However, many of the MPI-3 features, e. g. non-blocking collectives, even important older concepts like parallel I/O were not introduced. These topics would be considered for an advanced course for Master students at our University.

The introduction and splitting into three lectures paved the way for the exercises: programming for basic communication patterns based on the Pi example, then switching to collective communication. To understand the data communication aspect, students were programming benchmark tests with processes bound first to cores on a socket, then within a node and finally across nodes. Some students even noticed the differences and intricacies when multiple hops across IB-switches deter communication performance (in latency) and how other concurrent data transmission on a switch (another program started in parallel) may in this case inhibit the reproducability of the benchmark.

C. OpenCL

Using Graphics processing units (GPUs) to accelerate general computing is the latest and lasting trend in HPC, then called GPGPU. In order to really make the learning effort lasting and "general purpose", despite the dominance of NVIDIA's CUDA programming environment, the more portable OpenCL was chosen.

The theoretical part of the lecture introduced the current Top500, in detail the then no. 1 system, ORNL's Cray Titan machine [6] and its XK6 nodes. We then compared the performance and Flops-per-Watt ratio of state-of-the-art microprocessors and current GPUs. In order not to complicate the theoretical part too much, the underlying concepts of GPUs with the many execution units running in lock-step fashion (in bundles of 32 per warp for current NVIDIA GPUs and 64 per wave-front in ATI GPUs) were postponed into the second and third lecture.

Rather, the basic concept of programming for independent work-items (data-parallelism) was used to show the simplicity of the method and of the mapping to highly-parallel hardware. The programmer then benefits from OpenCL's ability to allow compilation at runtime for any hardware (for which a driver is installed), such as the SIMD units of the processors using AVX or the student's laptop's GPU.

Again, a framework for the exercises was provided, implementing the aspects of initializing the device, creating a context and enqueueing work. Part of the exercise consisted of printing of all OpenCL variables queried using `clGetPlatformInfo` and `clGetDeviceInfo`: the concept of computational units and their respective capabilities for both the CPU and GPU nicely match the architectural

[7] The cables of the former NEC SX-8 remain in the raised floor: for 72 nodes eight cables each to the crossbar switch are an impressive sight.

description. This provided a wealth of information, with which the different features of CPU and GPU may be classified, e. g. for activation the students were asked to explain to their neighbors the different values of the CPU's and GPU's OpenCL local memory.

The students were asked to program a simple kernel for vector-vector addition, a really simple task – and then were to examine the execution. This led to getting to know supported work-item dimension and work-group sizes and making the connection with the other variables queried earlier. Having successfully mastered the execution on the GPU, all the involved steps and functions to start the execution were revisited and the OpenCL calls examined in detail. This allowed the second exercise to use the texture-centric capabilities of accessing 2D texture memory structures: brighten an image, which requires knowledge on how to define and access textures and how to transfer data to/from the GPU.

The GPU being a separate device, possibly "far away" from the main CPU, requires a high-level programming interface to get profiling data and other general timing information. OpenCL offers an API with so-called events, that track each particular operation and can be queried afterwards to get the GPU-local timing information. This provides an albeit limited way to gather execution times of queued items. Together with the functionality for out-of-order execution, the last exercise were memory and kernel benchmarks to time different ways to transfer data to/from the GPU (plain, pinned memory, and copying data on the accelerator).

The main exercise however was a heat conduction example, which has been used in previous tutorials and lectures for OpenMP and MPI at HLRS, now in the C language. The basic Poisson PDEs of the time-dependent heat conduction equations were explained on the overhead projector, while the discretization using a 2^{nd}-order Newton approximation were augmented by animated slides (see Fig. 4a as one of the figures used in the slides for introduction into the 1D problem). While there exist better numerical methods to solve, the example allowed a short and concise introduction on how a phenomenon is solved by describing as mathematical equations and solved in a numerical scheme.

In the practicals, the iterative solution of finite difference of a 2D problem on a unit square was to be computed, first using a sequential program and then parallelized using OpenCL. The easy (and yet important) task was to identify the inner 5-point stencil as kernel operation. The harder part was to implement the double-buffering and to copy values back-and-forth into and out of the GPU for further processing. Again, some students were experimenting with different solutions, in this case a kernel working on multiple items using SIMD datatypes.

Here, the author wanted to combine visualization and computation of the values taking advantage of OpenGL – however, even after several time-consuming attempts, this has not yet worked! This is probably due to wrong specification of read-only and write-only textures (to be used as input and output for the OpenCL part, respectively), However, the textual output and visualization via the CPU worked and demonstrated the concept.

D. MapReduce

After Google's Jeffrey Dean and Sanjay Ghemawat famous paper [13] describing MapReduce, its applicability for the PageRank algorithm (and subsequently for many other in-house applications), the popularity of this approach increased alot. However, since this method of parallelization is currently uncommon in the industry our students aim for, MapReduce was not as prominent as in other CS courses on parallelization [14].

The MapReduce paradigm allows for distributed processing of large amounts of data (now coined "Big Data") through two main operations: Map to generate key-value pairs, sorting and filtering, and an ensuing reduction operation on the generated pairs. The model tries to hide details from the application programmer such as marshaling the operations, data-locality and even fault-tolerance. It was demonstrated and students experienced for themselves that installation and deployment of Apache Hadoop [15] is rather easy (with one minor problem, described below).

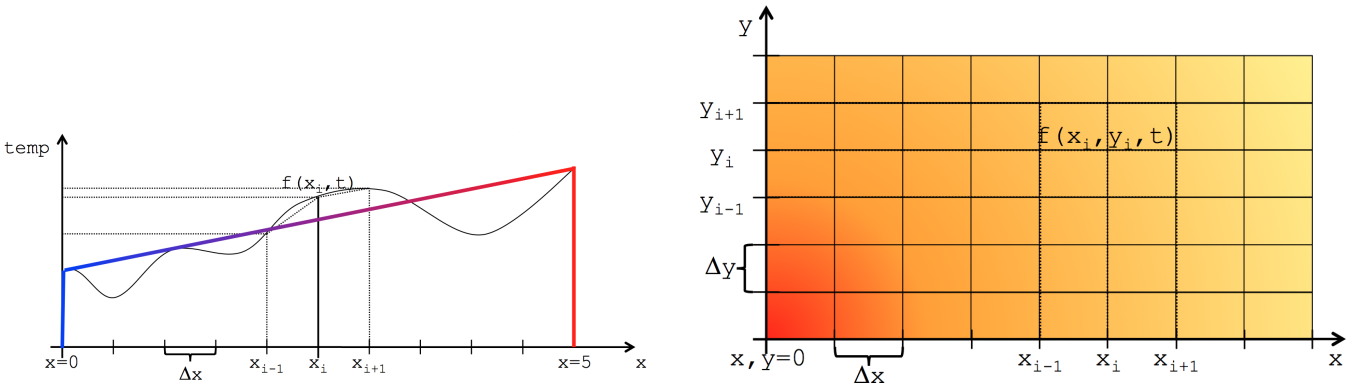
Up to this point, all exercises were to be implemented in C, this was the first exercise done in Java. So this was more natural to the students given their previous experience.

The first exercise consisted of the usual word-count example described in the Apache Hadoop tutorial. The further exercises included understanding of the distributed file system (HDFS) using Hadoop's tools to access files. At first the, deployment of the HDFS on multiple nodes was a problem.

E. Parallel programming tools

The author's background is in parallel programming tools, so he felt that any such course without a proper exposure to parallel programming tools is incomplete. Without knowledge of parallel programming, the tools for it cannot be introduced – students therefore referred to the usual method of debugging (mostly `printf` and once a traditional debugger). In effect, students were surprised to see the efficiency of the graphical debugger used, and particularly saw the necessity for performance analysis tools to handle the amount of data.

The lecture covered a broad set of tools from parallel programming extensions in IDEs (PTP for Eclipse), information system (supportive tools within Open MPI, e. g. hwloc), programming frameworks for memory error (Valgrind memcheck) and race condition detection (Valgrind helgrind), to graphical parallel debuggers (Allinea DDT and Etnus Totalview) and finally performance analysis tools for sequential



(a) Example for 1D discretization

(b) 2D discretization of a point source

Figure 4. Discretization and approximation of a gradient explained by animated PowerPoint slides.

execution (Intels Performance-Monitor, Valgrind's callgrind with kcachegrind, Intel Pin tools and Google-Perftools) and parallel performance analysis tools (Paraver, Vampir and Scalasca).

Of course, such a variety of tools cannot be exercised in depth in practicals, especially, since students may not have access to commercial software after the course. Therefore the focus was on firstly open tools and secondly an available debugger. In this case, exercises were centered around the Valgrind framework [16]: students were to take a faulty application (with multiple errors) and debug with Valgrind memcheck – errors to be detected included:

- Usage of uninitialized memory,
- Buffer overrun in an allocated array on the heap,
- Overlapping buffer arguments in libc-function calls,
- Unfreed memory,
- Buffer overrun on array on the stack causing corruption of local variables (not detected by Valgrind).

Then Valgrind was to be applied to MPI-parallel programs with techniques developed in [17]. Additionally students were instructed to use Allinea DDT to debug their MPI execution. One interested student has been accepted to do his practical term at Allinea in the UK.

V. Conclusion and Outlook

The course introduced undergrad students to a wide variety of methods for parallel programming. After the first few lectures, this course was considered as "being hard". Some of the students had prior experience with parallel programming, some joined the course because of their specific interest in GPU programming. The evaluation sheets collected right after the exam, were very positive. Specifically the real life codes used for the exercises were highly valued, as were the questionnaires.

To explain the various methods, current hardware architectures were described theoretically and practically – by having access to the newly acquired cluster nodes (based on new Intel SandyBridge and old Intel Nehalem), the students were able to compare the differences in high-end CPUs and even more importantly to their laptop's processor. With regard to other architectures, we did not provide access to accelerators (such as Intel MIC, GPUs and such), however the students who worked on their own machine were aided to use OpenCL e. g. on their local GPU. Having the possibility to show the actual HPC machines gave the course's content special credibility, significance and a coolness factor. However the "raison d'être" of the course was exemplified by the student's smartphone's multi-core processors.

As outcome for future lectures, the author will again use the method of questionnaires in subsequent lectures. This technique gives students also a feeling on what kind of questions to expect in final exams – and may be used to increase student's motivation.

The general structure of this course for undergrads is considered good. Nevertheless there are many points that would need to be improved. The OpenCL exercise would really impress students with working interactive visualization of the physical phenomenon. Based on the experience gained, the frameworks for the exercises needs to be worked on to be more intuitive and require less documentation. The MapReduce part must be extended by real life problems, e. g. data mining on publicly available data such as evaluating and aggregating actors and movies from www.imdb.org – again a computational problem students can relate to and are interested in.

From an organizational point of view in the future, access to the cluster would need to be provided via BW-Grid's certificate-based authentication scheme. For the time being, access was given via standard ssh-based logins, which inhibited login from disallowed IP addresses, i. e. from home.

Apart from the various standards, the books suggested to students are [18], [19], [20]. Some of the material developed here may be of interest to other groups as well. The content provided in the OpenMP part may be made available as a

Module to csinparallel.org. Finally the students agreed, that the parallelization of commonly used open source software such as GIMP in the frame of a student project would help the community and benefit the students themselves. This may be the topic of the next summer-term's "student project".

ACKNOWLEDGMENT

The author would like to thank the Hochschule Esslingen, and specifically Adrian Reber and Prof. Väterlein, as well as the High Performance Computing Center Stuttgart (HLRS). This paper is made possible by a grant from the Ministry for Science, Research and Art of the state of Baden-Württemberg in the frame of the bwHPC-C5 project.

REFERENCES

[1] Council on Competitiveness, "Advance. Benchmarking industrial use of High Performance Computing for innovation," IDC, Framingham, MA, USA, Tech. Rep., 2008.

[2] "IEEE computer society Technical Committee on Parallel Processing (TCPP)," Internet, 2013, http://www.computer.org/portal/web/TCPP.

[3] "Moodle project website," Internet, 2013, http://www.moodle.org.

[4] J. L. Furlani, "Modules: Providing a flexible user environment," in *Proc. of the* 5^{th} *Large Installation Systems Administration Conf. (LISA V)*, San Diego, CA, Sep. 1991, pp. 141–152.

[5] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE TCCA*, Dec. 1995, http://www.cs.virginia.edu/stream.

[6] H. W. Meuer, E. Strohmeier, J. J. Dongarra, and H. Simon, "The Top500 list," Tech. Rep., Nov 2012.

[7] G. M. Amdahl, "The validity of the single processor approach to achieving large scale computing capabilities," in *Proc. of the Spring Joint Computing Conference (AFIPS)*, vol. 30, 1967, pp. 483–485.

[8] J. L. Gustafson, "Reevaluating Amdahl's Law," *Comm. ACM*, vol. 31, no. 5, pp. 532–533, May 1988.

[9] D. Butenhof, *Programming with POSIX Threads*. Addison-Wesley, 1997.

[10] "GNU Image Manipulation Program," Internet, 2013, http://www.gimp.org.

[11] J. L. Gustafson, "Fixed time, tiered memory, and superlinear speedup," in *Proc. of the* 5^{th} *Distr. Memory Comp. Conf. (DMCC5*, 1990.

[12] E. Gabriel *et al.*, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proc. of the* 11^{th} *European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science (LNCS), D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., vol. 3241. Budapest, Hungary: Springer, Sep. 2004, pp. 97–104.

[13] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI'04:* 6^{th} *Symp. on Operating System Design and Implementation*, San Francisco, CA, Dec. 2004.

[14] R. Brown and E. Shoop, "CSinParallel and synergy for rapid incremental addition of PDC into CS curricula," *2012 IEEE 26th Int. Par. and Distr. Proc. Symp. (IPDPS) Workshops and PhD Forum*, vol. 0, pp. 1329–1334, 2012.

[15] T. White, *Hadoop: The Definitive Guide*, 3rd ed. O'Reilly, 2012.

[16] J. Seward and N. Nethercote, "Using Valgrind to detect undefined value errors with bit-precision," in *Proceedings of the USENIX'05 Annual Technical Conference*, Anaheim, CA, USA, Apr. 2005.

[17] S. Fan, "MPI-semantic memory checking tools for parallel applications," Ph.D. dissertation, University of Stuttgart, High-Performance Computing Center (HLRS), Jul. 2012.

[18] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*. Chapman & Hall / CRC, 2010.

[19] M. Scarpino, *OpenCL in Action: How to accelerate Graphics and Computations*. Manning Publications, 2011.

[20] B. R. Gaster, L. Howes, D. R. Kaeli, P. Mistry, and D. Schaa, *Heterogeneous Computing with OpenCL*. Morgan Kaufmann, 2011.