Run, Stencil, Run!
A Comparison of Modern Parallel Programming Paradigms

Helmar Burkhart, Matthias Christen, Max Rietmann, Madan Sathe, Olaf Schenk
{helmar.burkhart | m.christen | max.riethmann | madan.sathe | olaf.schenk } @unibas.ch

University of Basel
Department of Mathematics and Computer Science
Klingelbergstrasse 50
CH-4056 Basel, Switzerland

1 Introduction

While the performance of supercomputers has increased dramatically during the last 15 years, programming models and programming languages have more or less remained constant. Two de facto standards, the Message Passing Interface (MPI) for programming distributed memory architectures and OpenMP for programming shared-memory architectures still dominate the field of computational science and engineering. As current supercomputers are constellations with symmetric multiprocessor nodes that are coupled by high-speed interconnects such as Infiniband, hybrid versions OpenMP/MPI have been introduced. From the programmer's point of view, this is rather complicated and the resulting software productivity is not at all comparable with corresponding practice of commercial software elsewhere. While today's desktop computers are already parallel systems, this effect will be even more accentuated in the coming years. Extrapolation from the current TOP500 list tells us that a desktop or laptop computer will have more than 1000 processor elements in 10 years from now. Thus, in order to use computers efficiently, *all* programmers today have to address parallelism issues.

At the research and development level, new approaches can be identified:

- New general-purpose languages which introduce high-level constructs for dealing with parallelism.
- Domain-specific languages that offer problem-oriented support.
- Libraries of software patterns and application kernels.

In order to analyze the status of these innovative approaches and compare them with the standard programming models, we introduce an experiment in HPC software development. We use the class of stencil applications (also called structured-grid applications) in order to compare solutions for the programming models listed in section 2.

A stencil is a geometric structure on a structured grid consisting of a regular arrangement of nodes around a center node. In a stencil computation, the values of these nodes are used to update the value of the center nodes. Stencil computations often perform a very limited number of operations per node, and the number of operations is constant regardless of the problem size. Hence the performance of stencil computations is typically limited by the available bandwidth. Stencil calculations are part of many scientific programs such as in image processing, meteorology, geophysics, computational engineering, and life sciences.

2 Programming Models Investigated

For this experiment, six programming models were chosen, which were components in a course on High-Performance Computing (HPC CS311 Uni Basel). As Java is the programming language taught in the introductory courses, its concurrent programming libraries and

paradigms were used as an introduction to parallel programming. Chapel is a relatively new programming language with interesting features and has been taught in our course for the first time. As de facto standards for parallel programming on shared and distributed memory machines, OpenMP and MPI are covered. A brief side track on novel microarchitectures, specifically on graphics processing units (GPUs) is included in the course; thus we also include NVIDIA's CUDA. PATUS is one of our own research projects.

2.1 Java Concurrency

As of the Java Platform SE 5, the Java Concurrency framework is part of the JDK and provides access to higher abstraction constructs of parallel programming including thread scheduling mechanisms, interfaces for asynchronous execution (`Future`, `Delayed`), synchronization constructs (barriers, semaphores, locks), as well as concurrent data structures [1, 2]. The Java Concurrency package has been designed for task-level parallelism. For applications with a focus on data parallelism – such as the one chosen for this experiment – manual domain decompositions have to be done, and the subdomains have to be explicitly assigned to a task.

2.2 Chapel

Chapel is a programming language under development at Cray Inc. as part of the DARPA High Productivity Computing Systems program to improve the productivity of parallel programmers [3]. Chapel is influenced by earlier language specifications such as:

- data parallelism, index sets, distributed arrays from ZPL and High-performance Fortran
- task parallelism and synchronization from CRAY MTA C/Fortran
- object-orientation from Java
- the module concept from Modula-2
- generic programming/templates from C++

Chapel supports a global view programming model both for control and data structures, which makes it very attractive for programming stencil applications.

2.3 OpenMP

OpenMP has become the de facto standard for shared memory parallelization. Its strengths are the relative ease with which both work sharing constructs and task level parallelism can be created by inserting `#pragma` directives in the original sequential source code [4]. Thus, like Chapel, OpenMP proposes a global view programming style. In our case, a simple domain decomposition can be achieved by simply making use of the work sharing construct. As all the threads have a global view of the memory, no additional constructs are needed to handle the halo regions on domain boundaries.

2.4 MPI

MPI [5] is most prevalent programming paradigm for programming distributed memory architectures. It has become the de-facto standard for these types of architectures. MPI codes are typically written in the "Single Program Multiple Data" (SPMD) style, thus an MPI code runs as many program instances simultaneously as there are hardware resources. Parallelism is expressed at a rather coarse-grained program granularity. Communication and synchronization are done by calls to the MPI library.

As a paradigm for explicit distributed memory programming, MPI adds the difficulties of having to decompose the computation domain manually and communicate the updated artificial boundaries between subdomains explicitly. Further performance considerations such as overlapping computation and communication using asynchronous MPI constructs, significantly add to the challenge [5].

2.5 CUDA

CUDA, the *Compute Unified Device Architecture*, is NVIDIA's GPU architecture introduced with the G80 series of their GeForce GPUs in the fall of 2006. The GeForce 8800 was the first CUDA-programmable GPU. Parallel to the hardware, NVIDIA developed C for CUDA [6], the general-purpose language used to program their CUDA GPUs. CUDA C is a slight extension of C/C++. In particular, there are new specifiers identifying a *kernel*, the program portion which is executed on the GPU. Like MPI, the CUDA programming model follows the SPMD model; each logical thread executes the same kernel, and a kernel therefore is a thread-specific program in which special built-in variables have to be used to identify a thread and the portion of the data the thread is supposed to operate on.

2.6 Patus

Patus, which stands for Parallel Auto-Tuned Stencils, is a code generation and autotuning framework for stencil computations targeting shared memory machines including CPUs and GPUs [7]. It is thought of as both a productivity tool and an experimentation environment for parallelization and code optimization methods: the point-wise stencil calculation is specified using a small C-like domain-specific language (DSL), and as a second, driving input, Patus expects a "Strategy," written in another DSL, which describes the grid traversal and parallelization. It can be a bandwidth-saving algorithm such as cache blocking or some flavor of temporal blocking. The autotuning methodology is used as an approach to determine strategy-dependent parameters such that the performance (Flops per unit time) on a given hardware architecture is maximized. Currently, back-ends for CPUs (using OpenMP for parallelization) and CUDA-capable GPUs have been implemented, but the framework has been designed such that more hardware architectures can be added.

3 Problem Description

In this paper, we solve the classical wave equation with Dirichlet boundary conditions,

$$\begin{aligned} \frac{\partial^2 u}{\partial t^2} - c^2\nabla^2 u = 0 \qquad & \text{in } \Omega, \\ u \equiv 0 \qquad & \text{on } \partial\Omega, \end{aligned}$$

and initial condition

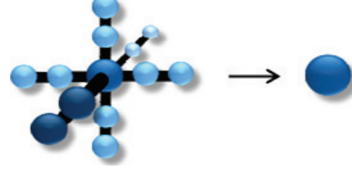
$$u(x, y, z, 0) = \sin(2\pi x)\sin(2\pi y)\sin(2\pi z)$$

on $\Omega = [-1, 1]^3$, using an explicit finite difference method both in space and time. For the discretization in time we use a second-order leap frog scheme with time step Δt. For the discretization in space, we choose a fourth-order discretization of the Laplacian on the structured uniform grid Ω_h with step size h:

$$u_{ijk}^{t+1} - 2u_{ijk}^{t} + u_{ijk}^{t-1} - \Delta t^2 c^2 \nabla_h^2 u_{ijk}^{t} = 0,$$

where

$$
\begin{aligned}
\nabla_h^2 u_{ijk}^t = & \tfrac{-15}{2h^2}\ u_{ijk}^t + \\
& \tfrac{-1}{12h^2}\ \left(u_{i-2,j,k}^t + u_{i,j-2,k}^t + u_{i,j,k-2}^t\right) + \\
& \tfrac{4}{3h^2}\ \left(u_{i-1,j,k}^t + u_{i,j-1,k}^t + u_{i,j,k-1}^t\right) + \\
& \tfrac{4}{3h^2}\ \left(u_{i+1,j,k}^t + u_{i,j+1,k}^t + u_{i,j,k+1}^t\right) + \\
& \tfrac{-1}{12h^2}\ \left(u_{i+2,j,k}^t + u_{i,j+2,k}^t + u_{i,j,k+2}^t\right)
\end{aligned}
$$



is the actual spatial stencil visualized above. The symbol ∇_h^2 denotes the discrete version of the Laplacian.

The pseudo-code for the sequential algorithm is described by Algorithm 1. It implements a Jacobi iteration that does $t_{\max}$ time steps. Rather than using one grid for each time step as suggested in the formula above, it uses 3 grids: u^- holds the values of the previous time step $t-1$; u^0 holds the values of the current time step t; and u^+ (time step $t+1$) receives the values of the computations depending on u^- and u^0 (line 3). u^+ is also the output of the computation, assuming that we are only interested in the grid values after completing the last time step.

Algorithm 1 Pseudo-code for the iterative stencil computation

Input: Arrays u^-, u^0; Stencil operator ∇_h^2
Output: Array u^+

1: **for** $t \leftarrow 1 \ldots t_{\max}$ **do** ▷ Iterate over the time domain
2: **for** $(i,j,k) \in \Omega_h$ **do** ▷ Iterate over the discrete domain, the index set Ω_h
3: $u_{ijk}^+ \leftarrow 2u_{ijk}^0 - u_{ijk}^- - \frac{\Delta t^2}{h^2} c^2 \nabla_h^2 u_{ijk}^0$ ▷ Compute the next time step at domain index (i,j,k)
4: **end for**
5: **if** $t < t_{\max}$ **then**
6: $\mathrm{tmp} \leftarrow u^-;\ u^- \leftarrow u^0;\ u^0 \leftarrow u^+;\ u^+ \leftarrow \mathrm{tmp}$ ▷ Rotate pointers for the 3 arrays u^-, u^0, u^+
7: **end if**
8: **end for**

4 Experiment Settings and Results

4.1 Test Environment

In order to compare the individual solutions, we specified a common test environment right from the beginning:

Hardware Architecture: Both a CPU and a GPU system were used:

- A four-socket quad-core Intel Xeon E7420 ("Dunnington") system. The CPUs run at 2.13 GHz, have 32 KB of L1 data cache, a 3 MB unified L2 cache, which is shared among 2 cores, and an 8 MB L3 cache, which is shared among 4 cores. The system is equipped with 128 GB of DRAM. The DRAM bandwidth was measured to be 8.3 GB/s using the STREAM Triad benchmark [8].
- A NVIDIA Tesla C2050 GPU of the "Fermi" generation. The GPU has 448 streaming processors clocked at 1.15 GHz. The graphics card is equipped with 768 KB of L2 cache and 3 GB of GDDR5 memory. The device was used with ECC turned on, in which case a bandwidth of 84.5 GB/s was measured using NVIDIA's bandwidth test.

Stencil computation: 200^3 grid points, the update calculation of each grid point performs 19 Flops. 100 time steps are computed, resulting in a 16 GFlop overall compute load.

Measurement data: Run time (in seconds) for the kernel computation only.

Speedup analysis: Absolute speedup, i.e., comparison with the original sequential code.

The programming environment settings are summarized in Table 1. Both OpenMP and MPI programs were compiled with the Intel C compiler, as was the code generated by PATUS.

Programming Language	Compiler	Optimization Flags
Java	Oracle/Sun Java 1.6.0_20	—
Chapel	Cray Chapel v1.2.0	`--fast`
C/C++ (OpenMP, MPI)	Intel C compiler icc 10.1	`-O3`
MPI	MPI Library for Linux v4.0	—
CUDA	NVIDIA CUDA 3.1	`-O3`

Table 1. Programming environments used by the students.

4.2 Questionnaire and Results

The students had to solve the stencil problem for all programming models introduced and they were asked to report both on the efficiency of the solution and their experience regarding programming ease and productivity. In order to gather data, the questionnaire shown in Fig. 1 was used.

#Threads	Time [seconds]	GFlop/s	Speedup	Efficiency
1				
2				
4				
8				
16				

Productivity:

- Time spent on parallelization: [ ] hours
- Lines of source code: sequential: [ ] parallel: [ ]
- Learning curve: ○ easy ○ medium ○ hard
- Difficulties during programming:
  - Source of errors (index calculation, parallelization, …)
  - Kind of errors (conceptional understanding, syntax, programming, ...)

Fig. 1. Questionnaire to be returned by students

The average values of the data handed in can be found in Table 2 while Figure 2 gives a graphical representation including min-max values.

The table values should be read as follows:

Working hours: The number of hours the team worked to write/parallelize the code and arrive at a working implementation.

Prog. Model	Working Hours	Parallel Overhead	Performance	Lines of Code	Learning Curve
Java	3.75	36%	4.79	231	2.0
Chapel	2.67	0%	1.49	110	2.0
OpenMP	3.25	7%	5.99	196	1.7
MPI	17.50	186%	6.19	597	2.5
CUDA	6.00	—	23.10	196	2.0
PATUS	0.85	—	7.97	22	1.3

Table 2. Data collected from questionnaires.

Lines of code: The number of lines of the *parallel* implementation.
Parallelism overhead: The percentage of lines that needed to be added for the parallelization compared to the sequential baseline code: $(\#\text{Lines}_{\text{par}} - \#\text{Lines}_{\text{seq}})/\#\text{Lines}_{\text{seq}}$.
Performance: The performance in GFlop/s when using 16 threads (or the one GPU).
Learning curve: A value between 1 and 3, 1 meaning “easy” and 3 meaning “hard.”

The parallelism overhead for the OpenMP and MPI implementations are based on a sequential C code. Parallelizing the problem in Chapel was done trivially by all the students by changing a `for` loop into a `forall` loop. Both for CUDA and for PATUS there is no sequential baseline, since a CUDA program is inherently parallel, and PATUS automatically generates parallel code.

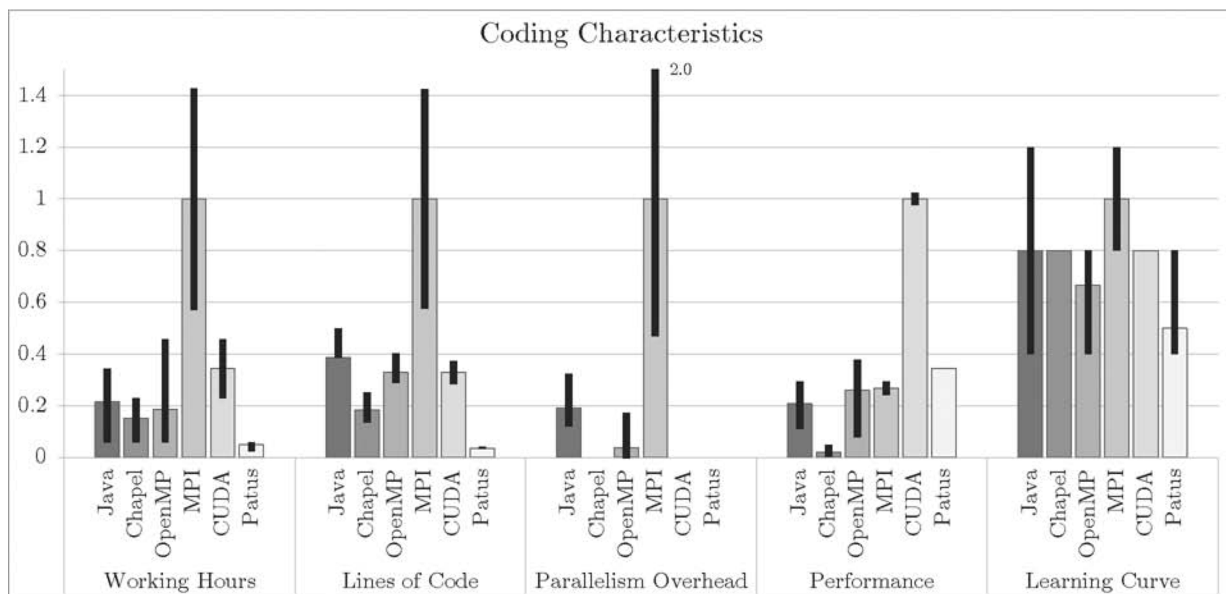


Fig. 2. Coding characteristics for the programming models evaluated. The bars shaded in gray visualize the average numbers, the overlaid black bars show the minimum and maximum values. The values are normalized such that the maximum of the average values are mapped to 1.

4.3 Performance Discussion

The students were asked to optimize their code for performance, and as an incentive for investing time into manual code optimization, the students turning in the codes delivering the best and second best performance were honored with extra credits. Nevertheless we want to point out that the performance results presented here are purely based on the efforts of the students; the figures might not show the performance numbers of optimally tuned codes.

Fig. 3 shows the scaling (from 1 to 16 threads/processes) and performance numbers on the CPU system obtained by the students. The minor horizontal axis corresponds to the number of threads/processes used; the major horizontal axis shows the programming models. The bars shaded in gray visualize the average performance numbers in GFlop/s for single precision calculations. The overlaid black bars show the minimum and maximum performance numbers which were reached.

The GPU performance result is not shown, since no scalability benchmarks were done on the GPU system. The average performance reached on the GPU system was 23.1 GFlop/s.

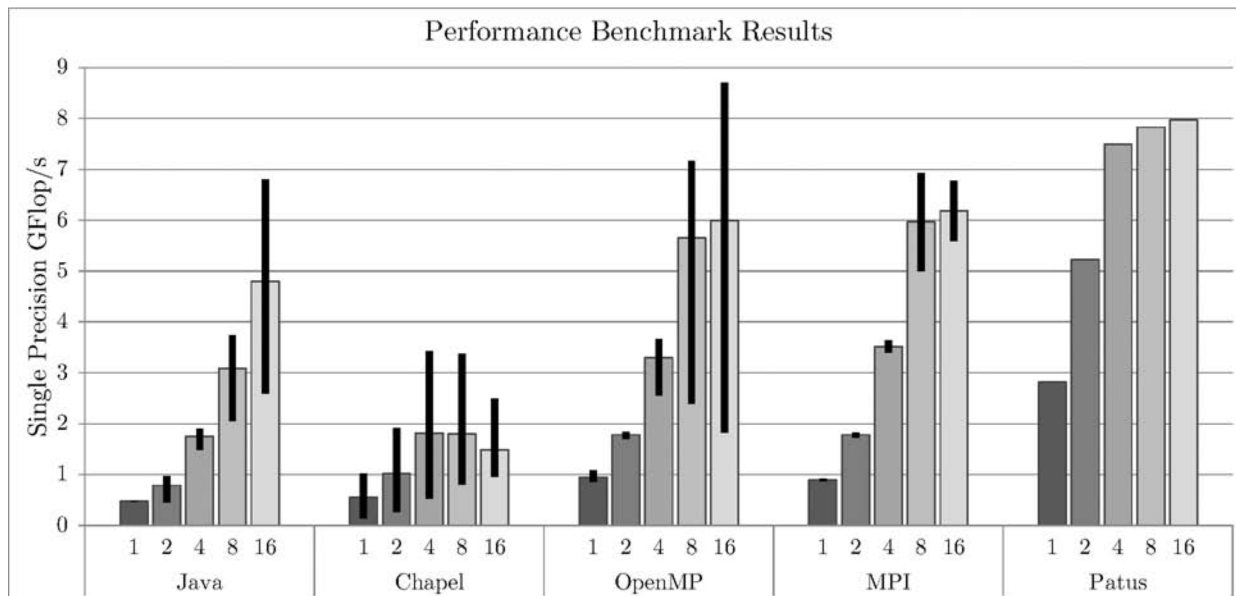


Fig. 3. Performance results obtained by the students. The bars shaded in gray visualize the average numbers, the overlaid black bars show the minimum and maximum values. Only performance numbers for the CPU system are shown.

Remarkably, the best student implementations in all the programming models which were evaluated on the CPU system achieve around 7 GFlop/s with 16 threads — except when Chapel was used. We believe that a better performance outcome can be reached with Chapel, but the figure suggests that more effort has to be spent on code optimization. Indeed, the students also commented on the fact by stating that optimization techniques used within other programming models did not result in comparable performance benefits in Chapel.

The bars suggest that performance-wise Java is around half as efficient as native compiled code (OpenMP, MPI), but the parallelization scaled nicely throughout all student implementations from 1 to 16 threads. The OpenMP, MPI, and Patus implementations fail to scale beyond around 7 GFlop/s due to the system's limited bandwidth supply.

The increasing spread in the OpenMP performance as the number of threads increases attest that OpenMP programming can be tricky. Despite the seeming simplicity of OpenMP programming, the programmer needs to have some understanding of the hardware architecture; merely knowing OpenMP constructs is not sufficient for a successful parallelization. The reasons for the slowdown when increasing the thread count are manifold; they include the potential pitfalls of OpenMP programming: wrong parallelism granularity (choosing the wrong loops for parallelization), false sharing (multiple threads write to the same cache line), needless synchronization and inefficient use of *atomic* constructs, and NUMA effects (data needs to be transferred from foreign memories).

The students also realized that their code optimizations were dependent on the hardware architecture: optimizations, which gave a performance boost on their development laptops, not necessarily showed a beneficial effect on the target machine. Yet, the winners of the performance competition were able to beat the code automatically generated by Patus.

The MPI and OpenMP codes are about on par with respect to the performance. Interestingly, the performance variance of the MPI codes is less that in the OpenMP case. This fact might indicate that MPI requires the programmer to think more carefully about the parallelization. Fig. 4 shows that considerably more time was spent to do the MPI parallelization; the figure suggests that, on average, the implementation in MPI took 5 times longer to implement than the OpenMP version.

One of the reasons for the considerably higher performance numbers on low thread counts when using Patus (around $3\times$ when using 1 thread) is the explicit vectorization of the code, i.e., Patus generates a code which explicitly uses SSE intrinsics, whereas the students did not exploit this data level parallelization manually, but relied on the compiler to do so. No minimum and maximum performance bars are shown because Patus creates the same code from valid equivalent stencil specifications.

4.4 Productivity Discussion

It has often been claimed that HPC software development needs to be more productive. While the discussion usually is qualitatively only (e.g., HPC language design issues), some researchers worked on quantitative measures [9–11]. The figures reported from our experiment do not allow fine-grain analysis (e.g. we do not know the portions of learning time and development time. In the following, we therefore define two coarse measures for HPC software productity.

$$(\textit{Coding Productivity}) = (\textit{Lines of Code})/(\textit{Working Hours}) \quad (1)$$

$$(\textit{Performance Productivity}) = (\textit{Performance Achieved})/(\textit{Working Hours}) \quad (2)$$

Fig. 4 shows our findings. Regarding *Coding Productivity* Java and OpenMP are on par. More code has been to written for Java but students are experienced in Java programming and can write code quickly while for OpenMP fewer directives had to be placed but careful thinking was necessary.

The ranking in *Performance Productivity* is more interesting. Patus is the winner which makes sense because it is specially targeted for the stencil application type. CUDA follows next because of the best overall performance achieved. At the end we see both MPI and Chapel. The first because of the many hours that need to be invested, the latter because the performance needs to be improved.

5 Conclusion and Further Work

While HPC benchmarking over the years has evolved into a sophisticated discipline, comparisons across different programming models and languages have been neglected. [12] has been in the right direction but it is outdated. Our *Run, Stencil Run!* experiment is an attempt for newer results. So far it has been a teaching experiment and we got first results which encourage us for further work in this direction. Activities on the to-do list are:

- Refinement of the questionnaire: For instance the number of work hours should be splitted into learning, development, and performance tuning.

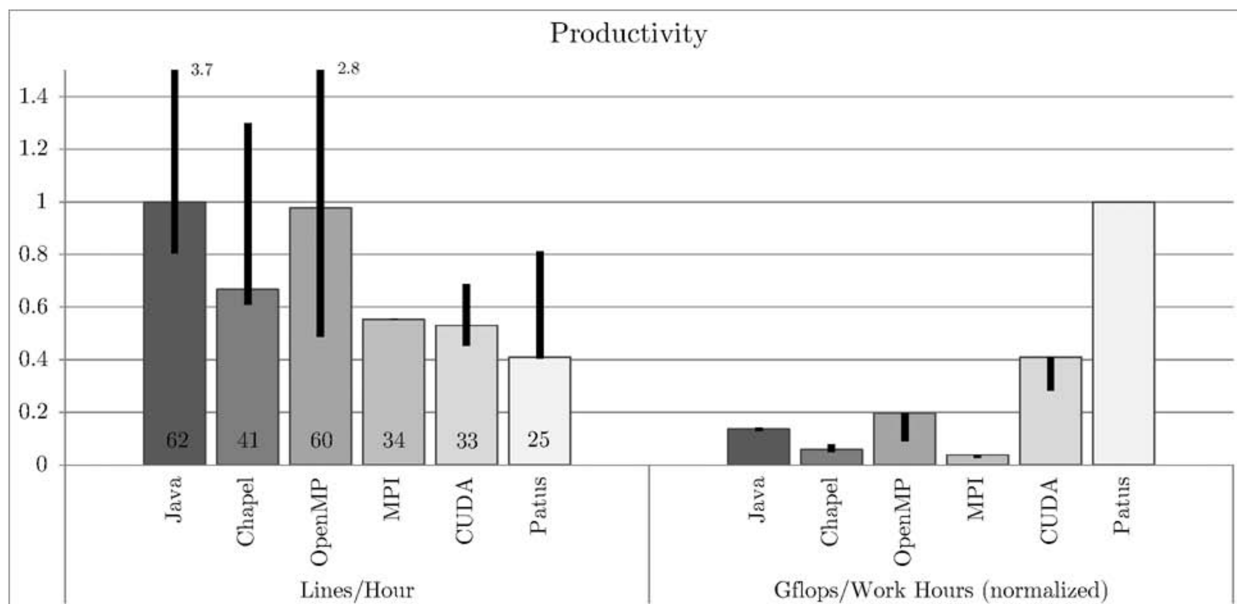


Fig. 4. Productivity metrics for the programming models evaluated. The bars shaded in gray visualize the average numbers, the overlaid black bars show the minimum and maximum values. The values are normalized such that the maximum of the average values are mapped to 1. The numbers in the bars are the absolute code lines-per-hour averages.

– High-performant comparison code: In order to compare the students' results with the optimum achievable, we have to develop optimized versions for all models.
– Experiments with other motifs: It will be interesting to see results for other motifs.

Whether the user test base can be extended is an open issue. However, repetition of the experiment in the forthcoming HPC courses at the Universities of Basel and Lugano is envisaged.

Acknowledgments

We acknowledge the students of the Master Course *High-Performance Computing (CS311)* at the University of Basel for taking part in this survey.

References

1. Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, and David Holmes. *Java Concurrency in Practice.* Addison-Wesley Professional, 2006.
2. Oracle. Java Concurrency Tutorial. `http://download.oracle.com/javase/tutorial/essential/concurrency/`. Accessed March 7, 2011.
3. Cray Inc. Chapel Language Specification 0.796. `http://chapel.cray.com`. Accessed March 7, 2011.
4. OpenMP Architecture Review Board. The Message Passing Interface (MPI) standard. `http://www.openmp.org`. Accessed March 7, 2011.
5. Message Passing Interface Forum. The Message Passing Interface (MPI) standard. `http://www.mcs.anl.gov/research/projects/mpi/`. Accessed March 7, 2011.
6. NVIDIA. NVIDIA CUDA™ — NVIDIA CUDA C Programming Guide. `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf`. Accessed July 2011.
7. M. Christen, O. Schenk, and H. Burkhart. Patus: A Code Generation and Autotuning Framework For Parallel Iterative Stencil Computations on Modern Microarchitectures. In *International Parallel & Distributed Processing Symposium (IPDPS)*, 2011.
8. J. McCalpin. STREAM: Sustainable Memory Bandwidth in High-Performance Computers. `http://www.cs.virginia.edu/stream/`. Accessed July 2011.
9. Ken Kennedy, Charles Koelbel, Robert Schreiber, Ken Kennedy, Charles Koelbel, and Robert Schreiber. Defining and measuring the productivity of programming languages. *The International Journal of High Performance Computing Applications, (18)4, Winter*, 2004:441–448, 2004.

10. Stuart Faulk, Adam Porter, John Gustafson, Walter Tichy, Philip Johnson, and Lawrence Votta. Measuring hpc productivity. *International Journal of High Performance Computing Applications*, 2004:459–473, 2004.
11. Marc Snir, David A. Bader, Marc Snir, David A. Bader, James C. Browne, Brad Chamberlain, Peter Kogge, John Mccalpin, Rami Melhem, and David Padua. A framework for measuring supercomputer productivity. *The International Journal of High Performance Computing Applications, (18)4, Winter.*
12. Duane Szafron and Jonathan Schaeffer. An experiment to measure the usability of parallel programming systems, 1996.