An Experiment in Measuring the Productivity of Three Parallel Programming Languages

Kemal Ebcioğlu and Vivek Sarkar
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA*

Tarek El-Ghazawi
The George Washington University
801 22nd Street NW
Washington, DC 20052, USA

John Urbanic
Pittsburgh Supercomputing Center
4400 Fifth Avenue
Pittsburgh, PA 15213, USA

Abstract

In May 2005, a 4.5 day long productivity study was performed at the Pittsburgh Supercomputing Center as part of the IBM HPCS/PERCS project, comparing the productivity of three parallel programming languages: `C+MPI`, `UPC`, *and the IBM PERCS project's* `x10` *language. 27 subjects were divided into 3 comparable groups (one per language) and all were asked to parallelize the same serial algorithm: Smith-Waterman local sequence matching – a bio-informatics kernel inspired from the Scalable Synthetic Compact Applications (SSCA) Benchmark, number 1. Two days of tutorials were given for each language, followed by two days of intense parallel programming for the main problem, and a half day of exit interviews. The study participants were mostly Science and CS students from the University of Pittsburgh, with limited or no parallel programming experience.*

There were two typical ways of solving the sequence matching problem: a wavefront algorithm, which was not scalable because of data dependencies, and yet posed programming challenges because of the frequent synchronization requirements. However, the given problem was shown to also have a subtle domain-specific property, which allowed some ostensible data dependences to be ignored in exchange for redundant computation, and boosted scalability by a great extent. This property was also given as a written hint to all the participants, encouraging them to obtain higher degrees of parallelism.

The programming activities of the study participants were recorded, both through face-to-face observations by the IBM/PSC teams, as well as through frequent automated sampling of the programs being written, such that it was later possible to analyze the progress of each study participant in great detail, including the thought process and the difficulties encountered. The detailed logs also allowed us to infer precisely when the first correct parallel solution was arrived at (this "time to first correct parallel solution" metric was used as one measure of productivity). This paper describes the results of the experiment, with particular emphasis on our technical observations on the codes produced by the anonymous participants. Interesting insights have been obtained into the problem-solving process of novice parallel programmers, including those exposing productivity pitfalls in each language, and significant differences among individuals and groups.

1 Introduction

High-performance supercomputers are heading toward increased complexity, and thus, high productivity tools and languages are very much on the agenda of supercomputing researchers. In order to be able to evaluate such productivity tools and languages with quantitative measures, there is an increased need for performing field experiments using human programmers as subjects. Feedback from such field experiments will be crucial as a guiding tool for future innovation in

*Current author contact emails: `kemal.ebcioglu@acm.org`, `vsarkar@us.ibm.com`, `tarek@gwu.edu`, `urbanic@psc.edu`.

productivity.

In this paper, we will embark on describing the technical insights into one such field experiment, comparing the parallel languages `C+MPI`[1], `UPC`[2], and `x10`[3], where the latter is IBM HPCS/PERCS project's new parallel programming language. The detailed design rationales for the languages themselves are beyond the scope of this paper, which will focus on the productivity study.

2 The productivity experiment

From Monday, May 23 to Friday, May 27, 2005, a 4.5 day long productivity study was performed at the Pittsburgh Supercomputing Center (PSC) as part of the IBM HPCS/PERCS project, comparing the productivity of three parallel programming languages mentioned above: `C+MPI`, `UPC`, and `x10`. 27 subjects were divided into 3 comparable groups (one 9-person group per language) and all were asked to parallelize the same serial algorithm: Smith-Waterman local sequence matching – a bio-informatics kernel inspired from the Scalable Synthetic Compact Applications (SSCA) Benchmarks [4], number 1. The problem was suggested to us by David Bader. The serial algorithm was provided to the subjects as a working sequential program (serial `C` for the `C+MPI` and `UPC` groups, and serial `x10` for the `x10` group).

The study participants were mostly Science and CS students from the University of Pittsburgh, mostly with limited or no parallel programming experience. The distribution of subjects to language groups were performed by the IBM Research Social Computing Group and PSC management, with no input from the technical language teams. Throughout the experiment, the subjects remained anonymous to the technical teams.

The experiment began with two days of tutorials (Monday and Tuesday) and hands-on exercises, taught by experts in each language. The availability of the PSC lab terminal and supercomputer resources were instrumental in making this possible. Then the subjects performed intense parallel programming for two days (Wednesday and Thursday) for the main problem. During the main programming session, the subjects were allowed to ask questions, but only about language constructs and technical problems they encountered, and not about the algorithmic issues themselves. All questions and the answers given were recorded. On Friday, the subjects had half a day of exit interviews summing up their experience, conducted by the IBM Research Social Computing team.

3 The problem description

The serial algorithm that the subjects were asked to parallelize is described below:

The Smith-Waterman local sequence matching algorithm consists of computing the elements of a $N+1$ by $M+1$ matrix, from two strings of lengths $N+1$ and $M+1$, where M is usually many times larger than N. Figure 1 depicts the basic computation in serial `C` code, and Figure 2 shows the resulting matrix on two example strings: `-GGTCC` and `-GCCGCATCTT`.

```
#define Match (-1)
#define MisMatch 1
#define Gap  2
int A[N+1][M+1]; //initialized to 0
char *c1=STRING1;
char *c2=STRING2;
for(int j=1;i<=M;j++)
for(int i=1;i<=N;i++)
 A[i][j]=
    MIN(0,
        A[i-1][j]+Gap,
        A[i][j-1]+Gap,
        A[i-1][j-1]+(c1[i]==c2[j]?Match:MisMatch));
```

Figure 1. The serial version of the local sequence matching algorithm

	-	G	C	C	G	C	A	T	C	T	T
-	0	0	0	0	0	0	0	0	0	0	0
G	0	-1	0	0	-1	0	0	0	0	0	0
G	0	-1	0	0	-1	0	0	0	0	0	0
T	0	0	0	0	0	0	0	-1	0	-1	-1
C	0	0	-1	-1	0	-1	0	0	-2	0	0
C	0	0	-1	-2	0	-1	0	0	-1	-1	0

C C
C C

T C
T C

Figure 2. An example computation

Among several possibilities, we will describe two typical ways of parallelizing the local sequence matching problem:

The first one is a wavefront algorithm, as illustrated in Figure 3. Since each matrix element (i, j) depends on

its West $((i, j-1))$, North $((i-1, j))$ and NorthWest $((i-1, j-1))$ neighbors, at each step, the matrix elements on a diagonal line can be computed in parallel, based on the matrix elements computed in previous steps. This algorithm and its many variants are not scalable, because of data dependencies: the maximum parallelism at any given step is limited to $min(N, M)$, the maximum length of a diagonal line (in units of matrix cells) in Figure 3; but N is small. Yet, this wavefront algorithm poses programming challenges for the novice subjects, because of the frequent communication and synchronization requirements in a distributed matrix implementation.

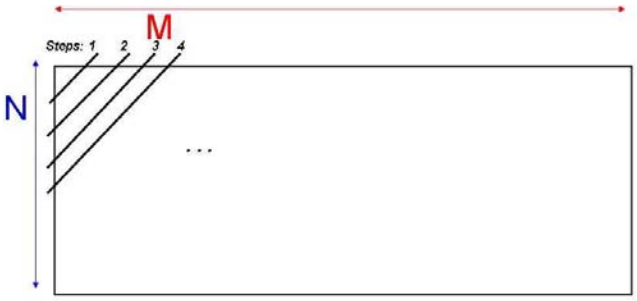

Figure 3. The wavefront computation

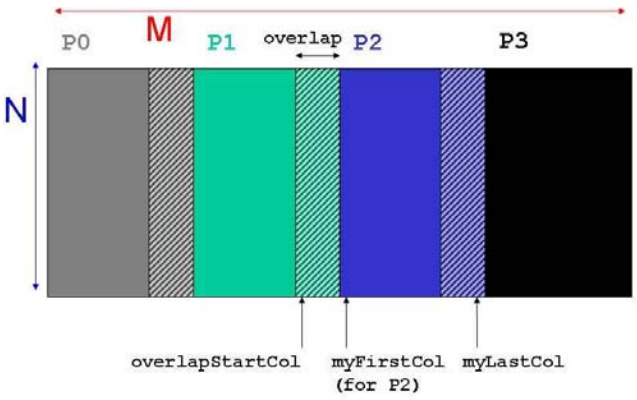

Figure 4. A scalable algorithm: Each processor computes its own columns `myFirstCol..myLastCol`**, using** `overlapStartCol..(myFirstCol-1)` **as warm-up.**

However, thanks to the bio-informatics domain experts at PSC, the given problem has been shown to also have a subtle domain-specific property, which allows some ostensible data dependences to be ignored, in exchange for redundant computation, and boosts parallel scalability by a great extent. Namely, starting the computation of figure 1 at any column of a matrix initialized to zeros (instead of column 1), and computing $N + abs((N * Match)/Gap)$ consecutive columns to the right of the start column ($1.5N$ columns in this example), ensure that the element values of the next column $(N + abs((N * Match)/Gap) + 1)$ are correct, i.e. are identical to what they would have been in the serial algorithm.

This property allows blocks of columns of the matrix to be subdivided among processors (as if with a `(*,BLOCK)` distribution), with each processor precomputing only $1.5N$ columns on the left of its block as "warm-up" (in a scratch area, without committing the results), and then, having obtained the correct element values for the leftmost column of its block, computing its entire block itself. The domain specific property mentioned above ensures that all computations are equivalent to the result of a serial computation. Figure 4 demonstrates this approach. This property was also given as a written hint to all the subjects, encouraging them to obtain higher degrees of parallelism using it.

Of course, adding wavefront computations to the scalable approach just described is also possible, for gaining additional parallelism.

4 A summary result of the productivity experiment

The productivity study team from IBM and PSC performed many automated and non-automated observations on the subjects throughout the study, by frequent sampling of the source code changes, recording of the results of compilation and execution (more than 180,000 events were automatically recorded), face-to-face observations, and interviews.

Because of the extensive automated recording of source code changes, it is possible to precisely determine when the first working parallel code was created. Figure 5 gives one summary of the study results for each subject. It shows the time between "development start time" and "development end time" as defined below, and further breaks down how that time was spent, using heuristic algorithms for identifying development phases such as authoring, debugging, and executing a program. Figure 5 also identifies which subjects never produced a working parallel program, and which subjects left the study without staying to the end.

- The *development start time* was assumed to be the first running of the serial program for the sequence

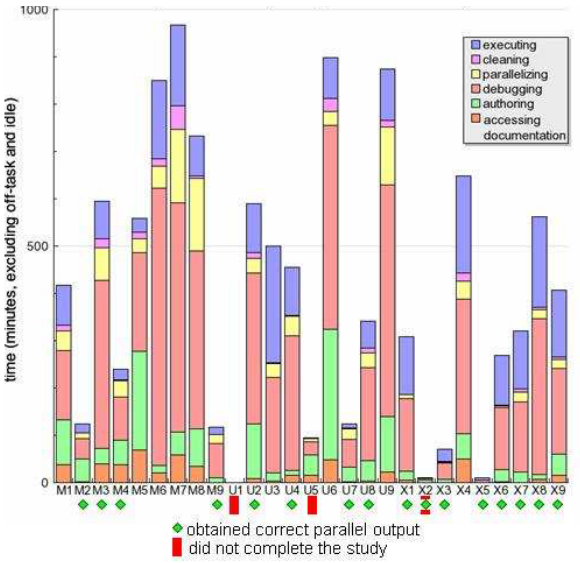

Figure 5. Development times and activities

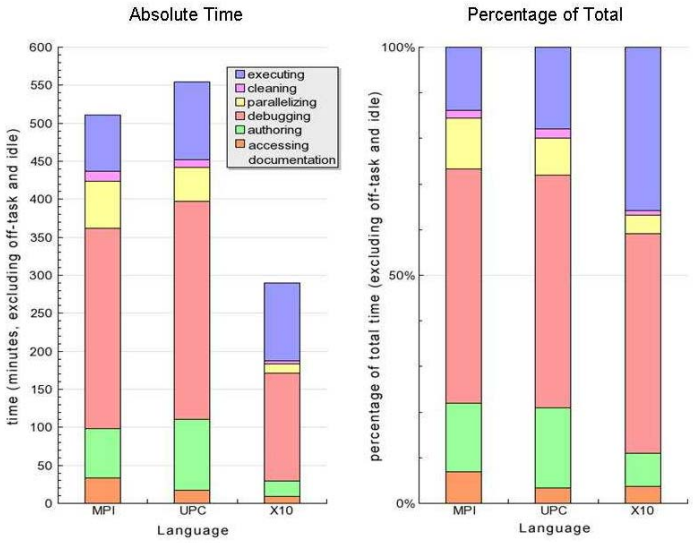

Figure 6. Development time by language

matching problem, or the appearance of the first parallel construct in the program, in case the user never ran the serial program.

- The *development end time* was taken to be the creation of a parallel program for sequence matching that gave the correct result in the PSC environment, on the reference input of $N = 10$, $M = 100$, and a fixed random pair of strings, and that indeed exhibited greater than 1X parallelism on the main computation of the matrix elements (the latter verified by a human expert). In case the subject was never able to produce such a program, the development end time was just taken to be the time the subject stopped work.

Of course, some of the subjects continued to improve and optimize their programs after creating their first working parallel program. Figure 5 does not include such optimization activities.

The additional summary result in figure 6 indicates that `x10` had an edge over `C+MPI` and `UPC` for the particular productivity metric of time-to-first-parallel-solution. Needless to say, all results are preliminary: in the conclusions section, we provide some additional insights for further study.

In the appendix of this paper, we will also focus on one additional type of analysis of the results: our technical team's own old-fashioned reading of the sequences of code produced by the subjects, which can provide some more detailed insight into the experiment. Future papers from the team will also describe other aspects of the experiment more comprehensively. The present paper is not intended to be a comprehensive presentation of the study.

5 Conclusions

Overall, we think the event that took place at PSC had the tenets of a well-designed major study. We will now summarize some of the informal observations and conclusions we have reached as the technical sub-team. We feel that these remarks may be valuable for the future experiments. While these insights are not new, it is interesting to observe their re-emergence as part of the practical productivity experiment.

Productivity insights for future language and tools design:

Abstraction mismatch: Some subjects were overwhelmed by the complexity of parallel programming (to the extent of being unable to generate any correct parallel program), or made their own work unduly complicated. The reason appears related to the inability to use the right concise abstractions, that specify what needs to be done at a high level. The parallel language and/or component library must have an arsenal of the right abstractions to express the task at hand. For example, for `x10`, it would have been better to use

an existing (*,BLOCK) distribution as an abstraction or readily available component (subject *X5* had to simulate his/her own (*,BLOCK) distribution instead). For C+MPI, it would have been better to have the abstraction to print a distributed matrix, rather than build a distributed matrix printing routine with low level synchronization and message passing. For languages using C as a base, mysterious segmentation violations (reflecting a C design trade-off that was once made in favor of performance vs. safety) are neither friendly nor can pass as a high level abstraction for a parallel programming task.

Lack of performance transparency: Tools that provide high-level, approximate feedback about the performance and parallelism being achieved by a user's program, would also be very useful to programmers at the early design stages. These tools would help recognize ostensibly parallel programs that give the correct results, but yet are not really parallel because of errors (*X4* created such a program). The tools could also provide performance awareness about language features, such as random access to arbitrary distributed array elements in UPC, or array distributions that can lead to bad locality in x10 (some x10 subjects chose a cyclic distribution for the matrix).

Lack of programming style and discipline: Programmers must not only be taught the constructs of a parallel language, but also a design style that shuns complexity and that rings mental bells of danger when a program fragment becomes too complex. Some subjects created unduly complex programs that probably reduced their potential productivity at the end. The lack of proper abstractions may have fueled this.

Lack of knowledge of parallel design idioms: Programmers must be taught the elementary parallel programming idioms such as data flow computation, divide and conquer,..., (somewhat like the design patterns of parallel programming). One UPC subject used an approach where each thread was continuously polling memory (busy-waiting), for its inputs to become available. Also, some subjects precipitated toward the comfort of having all processors do the same computation, or having all threads wait while thread 0 does all the work. Had we taught some more basic parallelism idioms up front, along with the language instruction, this could perhaps be avoided.

Nondeterminism considered harmful: Nondeterminism (available in most parallel languages, including x10) appears to be a dangerously powerful feature for novices. Nondeterminism makes it possible to write parallel programs easily, at the risk of being correct only some of the time by accident, and never noticing that in different circumstances the answer may be incorrect. The x10 team has been doing research in defining deterministic subsets of x10 to remedy this problem [5]. Also, the implementation of non-determinism must be made defensive, such that the probability of getting the right answer by accident is lowered, for example, by increasing randomness in the way threads schedulers work.

Insights for future experimental methodology and tutorials:

Tutorials are very important for getting across the basic concepts and impacting the productivity results. But sometimes it is hard for an instructor to read whether the message has gone across. The tutorials could have a non-intrusive online quiz component to them, via laptops or terminals, to provide immediate feedback to the instructors on how effective the learning has been.

Very common novice parallel programming pitfalls such as having all threads compute the same thing redundantly, or just having thread 0 do all the work, should be covered and advised against.

Correctness testing for parallel programming projects must be made rigorous, quickly testing many corner cases, and checking for deterministic stability, and should be spelled out to the subjects as well.

If at all possible, multi-day experiments must have 24-hour monitoring to accurately measure development time, not just during the day. Competitive brilliant minds will not rest, even when asked to rest during the off-hours!

Overall conclusions:

Overall, we believe that the present productivity experiment was an excellent experience. Through such a methodology, we believe that we can get one step closer to the goal of having a quantitative, measurable criterion on productivity, to guide future designs of languages and tools.

Acknowledgments

This work has been supported in part by the Defense Advanced Research Projects Agency (DARPA) under contract No. NBCH30390004. First, we are very grateful to our hard-working anonymous subjects. We thank David Bader for his suggestion for using the Smith-Waterman serial algorithm in a productivity experi-

ment. We are in particular grateful to Alexander Ropelewski and Hugh B. Nicholas, the bio-informatics experts at PSC, for their help with the domain specific property that made the scalable computation of the local sequence alignment problem possible. We are also grateful to Nick Nystrom from PSC, to Christine Halverson, Catalina Danis, and Wendy Kellogg from the IBM Research Social Computing group, as well as to Mootaz Elnozahy, Ram Rajamony, and Vijay Saraswat, for making this sizable productivity experiment possible.

Appendix: Observations on codes produced by subjects

In this Appendix, we will provide our analysis of the codes produced by each of the subjects (based on old-fashioned code reading). These are preliminary, and are subject to change in the future, as we gain a better insight into the large database of programs.

5.1 X10 subjects

- *X1:* Essentially used the following algorithm: for each element (i, i) on the diagonal, $1 \leq i \leq N$, compute the element, and then in parallel do: { sequentially compute the row elements $(i, i+1 : M)$; sequentially compute the column elements $(i+1 : N, i)$ }. This is not a very parallel or scalable solution, but does exhibit a minimum amount of parallelism (less than 2X). Also, some matrix elements were computed and written more than once, which surprisingly did not give incorrect results (the multiple writes wrote the same value).
- *X2:* This subject misunderstood the hint. *X2* subdivided the matrix into blocks of $1.5N$ columns each, and computed these blocks independently and in parallel. Surprisingly (perhaps as an idiosyncrasy of the java thread scheduler at PSC), this nondeterministic parallel program gave the correct answers at PSC in a repeatable manner.
- *X3:* Another nondeterministic program (parallel loop with inter-iteration dependences) that worked correctly by coincidence.
- *X4:* This subject also misunderstood the hint. He intended to subdivide the matrix into $1.5N$ columns and process them independently. However, he/she mixed up columns and rows and created multiple threads (`x10` activities), where only one thread was computing the entire matrix sequentially, and the remaining threads were doing nothing. Also, there was a data race (nondeterminism) which could print the matrix before it was completely computed, but the data race did not actually occur at PSC. Because the parallelism was never greater than 1X, *X4*'s parallel program was disqualified.
- *X5:* Quickly created a first parallel solution using a wavefront algorithm. Remarkable sequence of programs: one can see how he/she starts with a stock wavefront pattern, encounters problems, and solves them successfully. He/she was also the only `x10` group subject who understood the hint and later used it to create a scalable and correct parallel solution. There were many if-then-else statements to optimize for different cases, which made the code unduly complex. An HPF-style `(*,BLOCK)` distribution in `x10`, which was not available at the time, could have simplified *X5*'s programming.
- *X6:* The first parallel solution was a wavefront computation.
- *X7:* Used wavefront parallelism, but with a cyclic distribution which has poor locality.
- *X8:* The program does not work when N does not divide M evenly, but works (using wavefront parallelism) when it does.
- *X9:* Uses fine-grain wavefront parallelism.

5.2 C+MPI subjects

- *M1:* No correct parallel solution. *M1*'s code generated numerous compile-time and run-time errors which *M1* never fully resolved.
- *M2:* Remarkably concise and elegant wavefront solution based on pure send-receive synchronization. Each processor is assigned a block of rows of the matrix (a `(BLOCK,*)` distribution), and sequentially performs the following for each column of its block: receive the top element of the column from the previous processor (wait if it not available yet), then compute this column from top to bottom sequentially, and then send the bottom element of this column to to the next processor. This solution is essentially a wavefront algorithm, but it is not lock-step synchronized (it is data-flow synchronized via ordinary `MPI` send and receive),

and should be able to tolerate variable latencies very well. Unfortunately, this solution will not scale, because the maximum parallelism is limited with the wavefront approach.

- *M3:* Essentially the same solution as *M2*'s.
- *M4:* Has used the hint correctly, to create a scalable parallel solution.
- *M5:* No correct parallel solution. *M5*'s initial attempt at parallelization generated a segmentation violation, which *M5* was unable to correct despite several attempts.
- *M6:* No correct parallel solution.
- *M7:* Each processor is computing the same entire matrix sequentially. This solution was not accepted as a parallel one. *M7* tried to create more elaborate parallel solutions later, but did not succeed.
- *M8:* No correct parallel solution. Generated numerous `MPI`, `malloc`, and language errors (both compile-time and run-time)
- *M9:* Has used the hint correctly to create a scalable parallel solution.

5.3 UPC subjects

- *U1:* Did not complete the study.
- *U2:* seems to have obtained perhaps the best wavefront solution across the teams, with low communication overhead. Uses a `(*,BLOCK)` distribution which eliminates vertical communication among matrix elements, and performs wavefront computation with each wave consisting of cells of size 1 by 25 (for the 10 by 100 input). But of course, the wavefront solution is not scalable.
- *U3:* Did not obtain any correct and parallel solution.
- *U4:*
- *U5:* Did not complete the study.
- *U6:* Did not obtain any correct and parallel solution. U6's initial attempt to parallelize the task was unusually complex, relying heavily on pointer arithmetic and never generating a correct solution.
- *U7:*
- *U8:*
- *U9:* Even though this UPC program is ostensibly parallel, only thread 0 is computing the entire matrix sequentially. Threads other than 0 are not doing anything. This was not accepted as a parallel solution.

References

[1] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. "MPI: The Complete Reference", Massachusetts Institute of Technology Press, 1996.

[2] Tarek El-Ghazawi, William Carlson, Thomas Sterling, Katherine Yelick. "UPC: Distributed Shared Memory Programming", Wiley Interscience, ISBN: 0-471-22048-5, 252 p., June 2005.

[3] Philippe Charles, Christopher Donawa, Kemal Ebcioglu, Christian Grothoff, Allan Kielstra, Vijay Saraswat, Vivek Sarkar, Christoph von Praun. "X10: An Object-Oriented Approach to Non-Uniform Cluster Computing", Proc. OOPSLA 2005.

[4] David Bader. "Scalable Synthetic Compact Application Benchmarks", available at `http://www.highproductivity.org/Benchmarks`. Contact the web site owners to request access to the code.

[5] Vijay Saraswat, Radha Jaghadeesan, Armando Solar-Lezama, Christoph von Praun. Determinate Imperative Programming: A clocked interpretation of imperative syntax. http://www.saraswat.org/cf.html