The p196_mpi Implementation of the Reverse-And-Add Algorithm for the Palindrome Quest

Romain Dolbeau [1)], Monika Kwiatkowska [2)], Lukasz Swierczewski [3)],

[1)] CAPS entreprise
[2)] Faculty of Mathematics, Physics and Computer Science, Maria Curie-Skłodowska University in Lublin
[3)] Computer Science and Automation Institute, College of Computer Science and Business Administration in Łomża

Keywords: p196, MPI, palindrome **Topic area:** Algorithms and Analysis > Performance evaluation and tuning

Introduction

To quote John Walker, the first person to brute-force the problem [1]:

Pick a number. Reverse its digits and add the resulting number to the original number. If the result isn't a palindrome, repeat the process. Do all numbers in base 10 eventually become palindromes through this process? Nobody knows.

After the three years of computing that gave its title to Walker paper, many others became interested in the issue. The problem itself is older, and was already mentioned in David Wells book on the subject of interesting numbers [2]. Many people worked on the quest over the decades. The numbers whose existence is questioned were christened Lychrel by Wade VanLandingham, who maintains a comprehensive web site dedicated to them [3].

This research poster describes the implementation of the `p196_mpi` code, a distributed code dedicated to compute iterations of the palindrome quest as fast as possible. As of the end of 2013, this is the fastest known code to work on the problem. It was the first code to break the 300 million digits mark and subsequently to reach 600 million digits.

Distributed algorithm & implementation

The distributed algorithm is essentially the splitted computation-propagation algorithm using very large sub-arrays. The entire array of digits is split in many sub-arrays, one per process. All sub-arrays are computed in parallel, each using an optimized kernel implementation.The carry are virtually "stored" by being sent to the next sub-array, then propagated; this is repeated until there is no more non-zero carry to propagate (propagating through more than one large sub-array is exceedingly rare in practice). In the actual implementation, the whole array is splitted into two sub-arrays per process. The first process owns the first and last sub-arrays, the second process owns the second and next-to-last sub-array, and so on, until the last process owns the two middle sub-arrays. The reason is data availability: computing the first digit of the first sub-array requires the last digit of the last sub-array, from the "reverse" part of the "reverse-and-add" process. Storing them in the same process saves a large amount of data exchange between processes. However, the sub-arrays are almost never perfectly aligned. First, the number of digits is unlikely to be an integer multiple of the number of processes. Second, there is a need for expansion, are each carry propagation beyond the most significant digit increases the digit count by one. So the last sub-array is smaller than the others for expansion.

The sequence of operations illustrated in figure 1 is therefore:

1. Check whether we already have a palindrome and should stop (step 1);
2. Do the additions & carry propagation inside both sub-arrays (step 2);
3. Send carries for both sub-arrays to neighboring processes, propagate. Iterate if needed (step 3);
4. Process 0 lets everyone know whether there is an extra digit (step 4);
5. Send new values that are needed for the next iteration to the neighboring processes (step 5).

The last step is required because of the imperfect matching of sub-arrays inside each process. This is a critical part: if the last sub-array is much smaller than the others, then a lot of data exchange is required. But the last sub-array grows, and if it is close in size to the others, it will quickly catch-up in size. Whenever the last sub-array becomes as large as the other, a redistribution of data is needed (step 6 in figure 1), growing all sub-arrays evenly at the expense of the last sub-array. After each such redistribution, the size discrepancy is maximal, and therefore the cost of the data propagation (step 5) is maximal. This cost goes down as the last sub-array grows, and it is minimal just before a new redistribution is required.

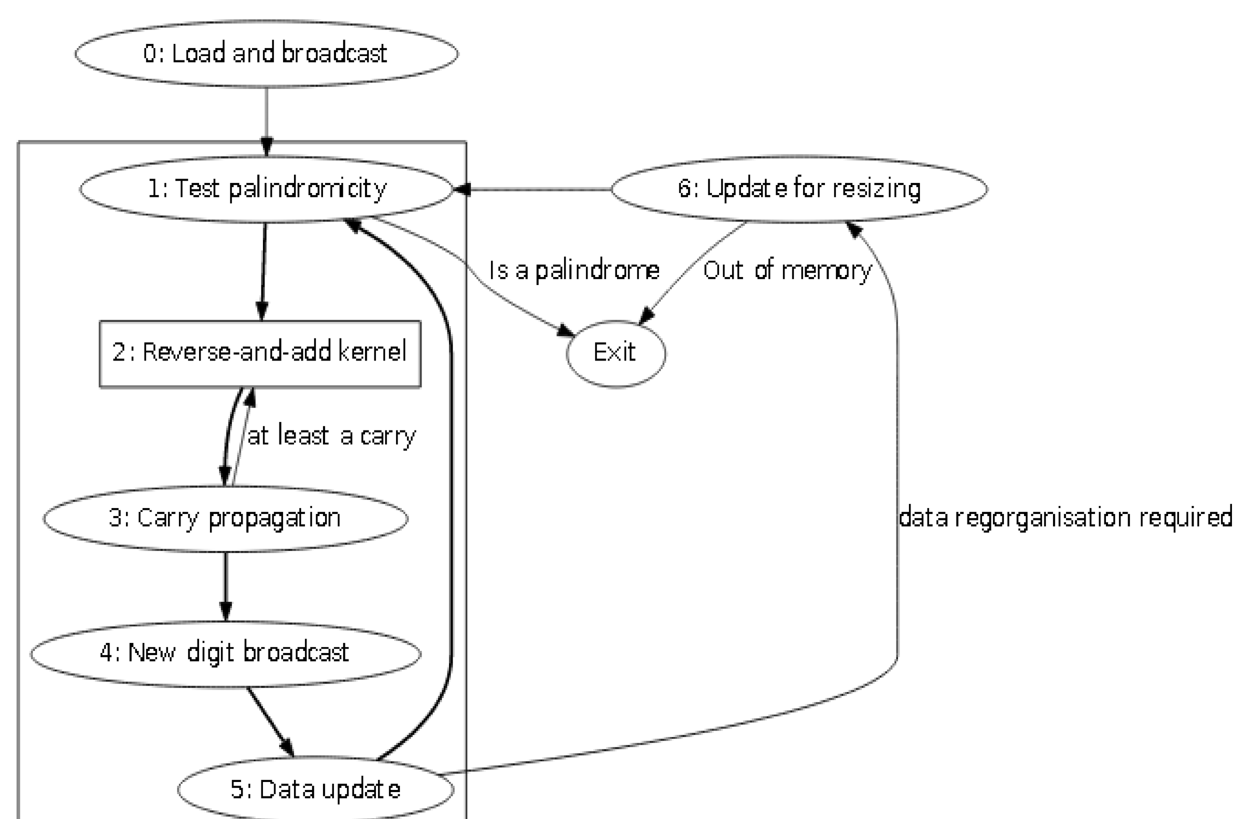

Figure 1. Overview of the distributed algorithm

Website where you can find implementation and more information:
http://www.dolbeau.name/dolbeau/p196/p196.html

Performance Results

Performance and scalability of the program were analyzed using multiple generations of Intel vector instruction sets: SSE3, SSSE3 and SSE4. SSSE3 introduces a shuffle instruction which helps with data mirroring, while SSE4 introduces data extraction and comparison instructions which help with carry propagation. SSE4ASAVX represents the SSE4 intrinsics compiled for an AVX target, and thus using the VEX-encoded 3-operands AVX-128 instructions.

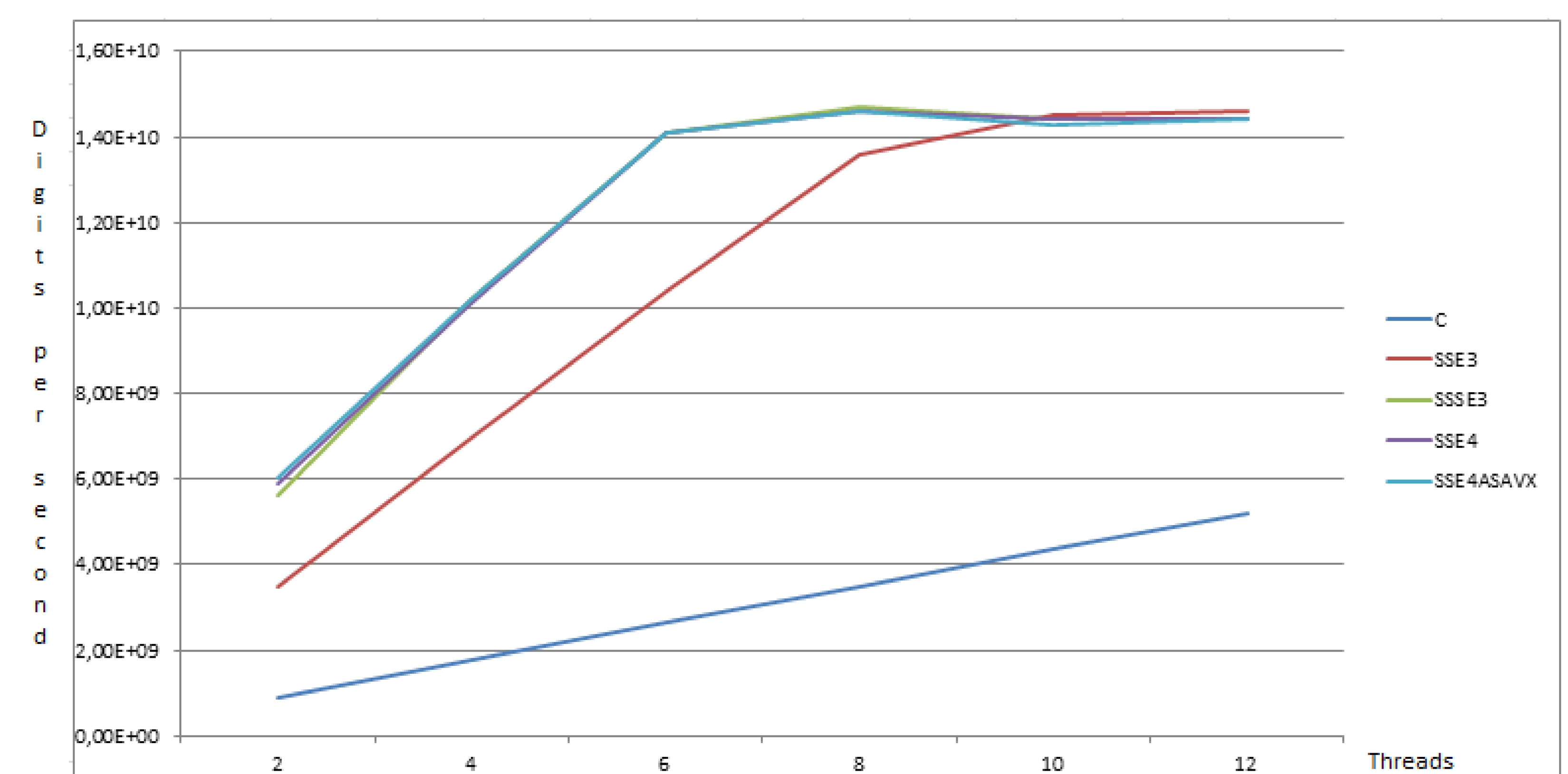

Figure 2. Graphic visualization of performance of the consecutive iterations of the program for the numbers range of 600 000 000 and a platform consisting of Intel Xeon E5-2697.

The results in Figure 2 are the average performance in digits computed per seconds over a thousand digits starting at a size of 600 millions digits, which was found to be a sufficient time to be representative. Processes were running on a single Intel Xeon E5-2697 processor of the "Ivy Bridge" family, which has 12 physical cores (Hyper-Threading was enabled but not used). Everything was compiled with the latest Intel compilers (14.0.2). Clearly, vector instructions are indispensable for good performance, as the pure C code doesn't reach the theoretical maximum of approximately one third the memory bandwidth. All others can reach that maximum, but the SSE3 version, with its less efficient mirroring, requires more cores. On a small number of cores the relative order is the one that can be expected (SSE4ASAVX being fastest, followed by SSE3, SSSE3, SSE3 and C). Once the performance is memory-bound, there is no longer a clear order as the differences are of the order of the measurement accuracy.

Conclusion & Alternatives

As of late 2013, this is the fastest known implementation of the base-10 reverse-and-add algorithm. At least two improvements could theoretically be made to the code. First, during the actual reverse-and-add computations, the data could be used to compute both sub-arrays simultaneously. This would theoretically save on memory bandwidth, a major bottleneck of the current implementation. Second, each digit is currently stored in a full byte. If each digit was stored in a nybble (half a byte), then the memory bandwidth requirement would be cut in half (along with the memory size itself). However, this would break the block-level carry propagation algorithm, as it effectively requires the ability to store up to 20 different values in the storage space of a single digit. So a different algorithm, still mostly memory-bound, would be required to effectively use this alternate data storage format. Finally, no matter how fast the reverse-and-add process, such code will never prove the existence of Lychrel numbers. While it is extremely unlikely that a palindrome will be found, not finding one does not mean there isn't one. Only a theoretical demonstration would work, as exists for some other bases.

Acknowledgment

Interdisciplinary Centre for Mathematical and Computational Modeling (ICM), Warsaw University, Poland is acknowledged for providing the computer facilities under the Grant No. G55-11.

Note

The thesis presented on International Supercomputing Conference 2014 in Leipzig (Germany).

References

1. J. Walker. (May 25th, 1990) Three years of computing. [Online]. Available: http://www.fourmilab.ch/documents/threeyears/threeyears.html

2. D. Wells, The Penguin Dictionary of Curious and Interesting Numbers. Penguin Books, 1986, no. 0-14-008029-5.

3. W. VanLandingham. 196 and other Lychrel numbers. [Online]. Available: http://www.p196.org

4. Intel Corporation, Intel R 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z, March 2013, no. 325383-046

5. Romain Dolbeau "The p196 mpi implementation of the reverse-and-add algorithm for the palindrome quest." [Online] <http://www.dolbeau.name/dolbeau/p196/p196_mpi.pdf>