Appentra
automatic software acceleration
www.appentra.com

Parallware Success Story: NAS Parallel Benchmark EP

DESCRIPTION

The NAS Parallel Benchmarks [1] (NPB) are a set of benchmarks targeting performance evaluation of highly parallel supercomputers. They are developed and maintained by the NASA Advanced Supercomputing (NAS) Division (formerly the NASA Numerical Aerodynamic Simulation Program) based at the NASA Ames Research Center. The NPB are valuable since they are rigorous, they offer a wide range of test sizes and, in contrast to other synthetic benchmarks, they simulate computation algorithms that are close to real-life applications.

The benchmark NPB EP is an embarrassingly parallel program where two-dimensional statistics are accumulated from a large number of Gaussian pseudo random numbers, which are generated using the Marsaglia polar method.

PARALLELIZATION

The extraction of parallelism in the source code of NPB EP is a great technical challenge, mainly because it performs irregular computations through subscripted subscripts. In addition, it has complex control flows arising from branches and procedure calls that may potentially lead to unpredictable race-conditions at run-time.

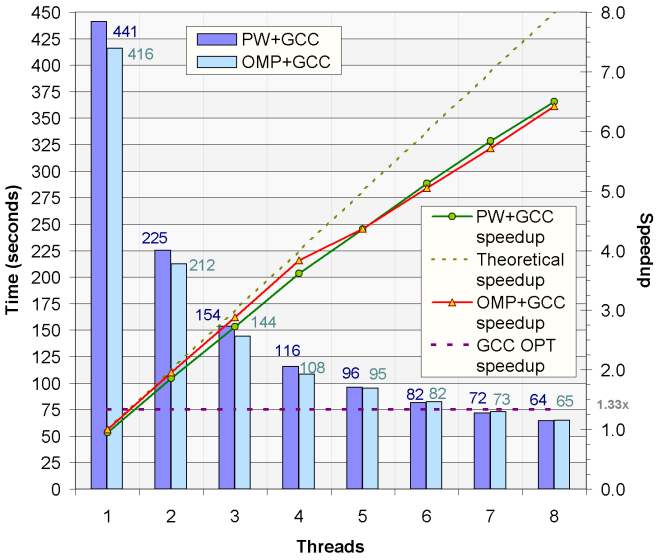
NPB provides a reference OpenMP parallel implementation [2] that handles race-conditions in NPB EP through array privatization and array reduction operations. Each thread is provided with a private copy of the array, and computes partial results on its array copy. At the end, the private copies of each thread are reduced into a single array.

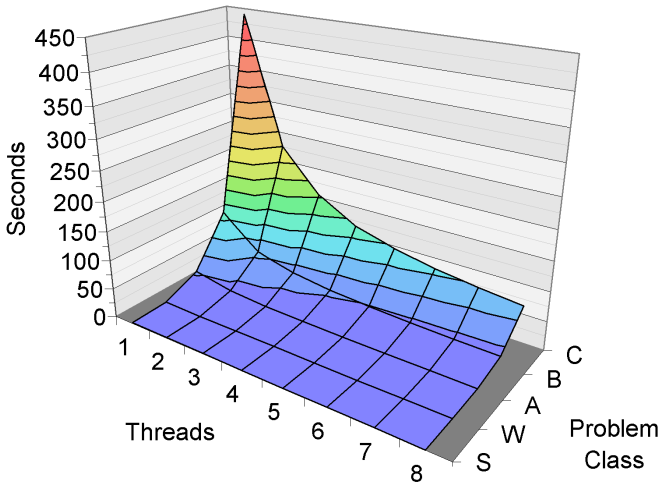
Parallware [3] supports several parallelization strategies that apply to the source code of NPB EP [4]. In particular, it supports the parallelization strategy used in the OpenMP parallel implementation of NPB EP.

PERFORMANCE

NPB EP was originally written in Fortran and later ported to the C, which is the programming language supported by Parallware. The performance of NPB 3.3 implemented in C is evaluated by comparing the following versions: the OpenMP implementation of NPB (*OMP+GCC*), the OpenMP-enabled parallel source code generated automatically by Parallware (*PW+GCC*), and the automatically vectorized/parallelized version produced by the GCC compiler (*GCC OPT*). The baseline for speedups is the sequential execution using GCC.

NPB EP offers several predefined problem classes of increasing size to fit a wide range of computers, from old uniprocessor systems to modern parallel supercomputers. The problem sizes are as follows: S is the smallest class, just for test purposes; W originally was workstation size, but nowadays is likely too small; A, B and C are the standard test problems, approximately 4x size increase going from one size to the next.





For the biggest problem class C, the performance of *PW+GCC* closely tracks the performance of the reference implementation *OMP+GCC*, which scales linearly as the number of threads raises. Similar results are obtained for the other classes S, W, A and B.

For all problem classes, the execution time of *PW+GCC* decreases as the number of threads raises. Overall, the performance of *PW+GCC* scales with the number of threads and the problem size.

CONCLUSIONS

Parallware tools raises programmer's productivity by automating the addition of OpenMP capabilities in source code of a sequential program written in the C programming language. By tuning the syntactical codification of the source code, the programmer takes advantage of the computational power of modern multicore computers.

The results demonstrate that Parallware succeeds with real scientific programs, providing performant OpenMP parallel programs.

Appentra
automatic software acceleration
www.appentra.com

Parallware Success Story: NAS Parallel Benchmark EP

TECHNOLOGY AND PRODUCTS

The parallelization of NPB EP is driven by profiling, which enabled the identification of the hot spot that consumes more than 71% of the execution time. The figure below shows a snapshot of the Parallware Viewer tool, an online interactive web application that provides easy access to the Parallware Compiler tool. The source code of the hot spot consists of a main loop `for_k` (see header at line 74) that computes the accumulations on array `q` (line 107). In each loop iteration, a set of array entries `q[l]` are updated. The position `l` is computed by evaluating a complex expression (lines 98-105) whose value is unknown at compile-time because it depends on the values stored in array `x` (lines 98-99). Additionally, the array `x` is computed interprocedurally through calls to procedure `vranlc()` (line 95). Thus, unpredictable race-conditions may arise at run-time.

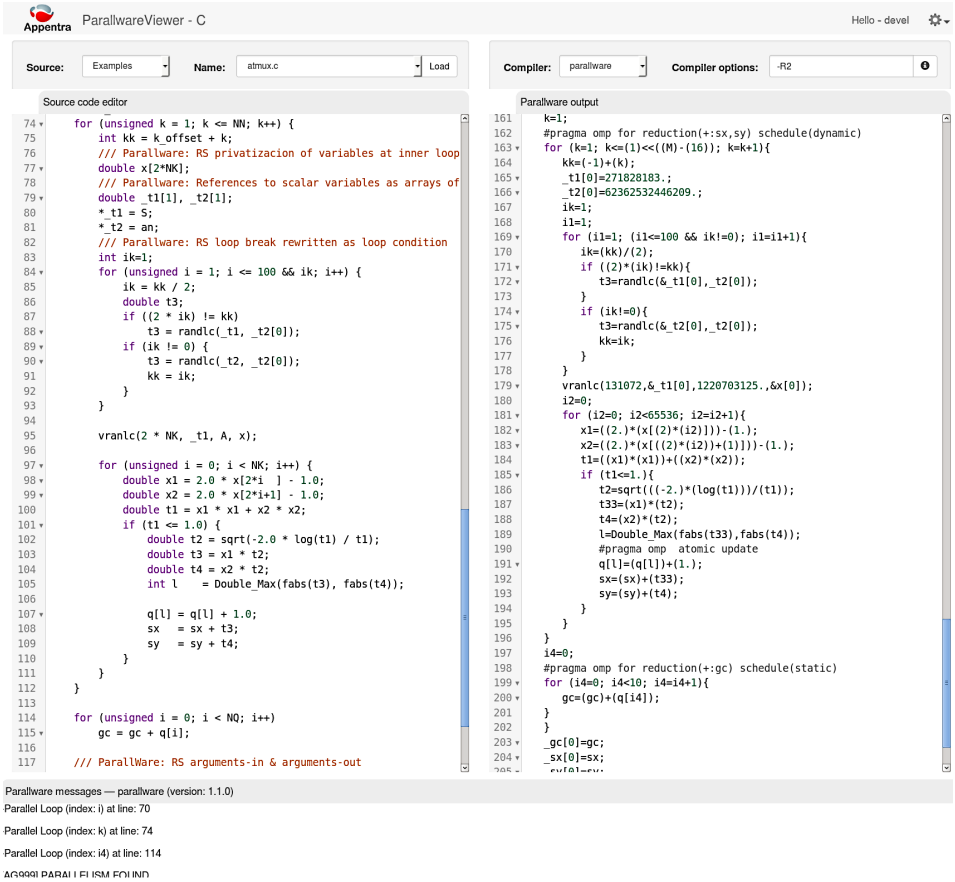
Parallware supports several parallelization strategies to handle race-conditions in irregular codes [4]. Parallware Viewer shows the default strategy that protects the memory access using the OpenMP pragma `atomic`. This parallelization strategy does not scale well as multiple threads are continuously updating the array. Through a single compiler flag, Parallware supports the same parallelization strategy used in the reference OpenMP implementation of NPB EP. Thus, each thread is provided with a private copy of the array. At the end of the process, the private copies of each thread are reduced again into a single array.

Parallware tools enable to easily select the best parallelization strategy for NPB EP, enhancing performance and productivity.

EXPERIMENTAL PLATFORM

The test platform is an Intel Core i7 4700MQ CPU running at 2.4 GHz. This processor has four physical cores, but is able to execute up to 8 concurrent threads using hyperthreading. The system has 8 GB of DDR3-1600 memory. The software setup is Lubuntu 14.04.2 x64 operating system using gcc 5.1.0 and Parallware 1.1 compilers.

The following compiler flags were used in the experiments. The baseline is GCC with the optimization flag `-O3`. The flags for *OMP+GCC* are `-O3 -fopenmp`. The compiler flags for *PW+GCC* are `-O3 -fopenmp` for GCC, and `-a -GSIZE_q_0=NQ` for Parallware to select the privatization-based parallelization strategy and to specify the name and size of the privatized array. Finally, the flags for *GCC OPT* are `-Ofast -march=native -flto -ftree-vectorize`, enabling the automatic vectorization and parallelization capabilities of GCC.



REFERENCES

[1] D.H. Bailey, E. Barszcz, J.T. Barton, D.S. Browning, R.L. Carter, L. Dagum, R.A. Fatoohi, P.O. Frederickson, T.A. Lasinski, R.S. Schreiber, H.D. Simon, V. Venkatakrishnan, and S.K. Weeratunga. The NAS Parallel Benchmarks - Summary and Preliminary Results. In Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing '91, pages 158–165. ACM, 1991.

[2] H. Jin, H. Jin, M. Frumkin, M. Frumkin, J. Yan, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical report, NASA Ames Research Center (NAS System Division), 1999.

[3] The Appentra team. Parallware: The OpenMP-enabling Source-to-Source Compiler, May 2015. [www.appentra.com]

[4] J. Lobeiras, and M. Arenaz. A Success Case using Parallware: The NAS Parallel Benchmark EP. OpenMPCon conference. Sep 2015.

ACKNOWLEDGEMENTS

We would like to thank Oak Ridge National Lab (ORNL) for providing the sequential and OpenMP parallel implementations of the NAS Parallel Benchmarks version 3.3 written in the C programming language.