Redesigning Parallel Symbolic Computations Packages

Georgiana Macariu, Marc Frîncu, Alexandru Cârstea, Dana Petcu, Andrei Eckstein,

Institute e-Austria Timişoara, Romania, science@ieat.ro
Western University of Timişoara

The ability of multi-core processors to increase application performance depends on the use of multiple threads within applications. Symbolic computations, requiring both CPU power and large memory, are well-suited candidates for deriving advantages from multi-core parallel architectures. This is possible only if the specific libraries and tools are designed to allow multi-threading and multi-processes. In order to promote the changes needed to adapt these libraries and tools to the new architectures, the changes performed on two main algorithms for symbolic computations, parallel integer factorization and parallel Gröbner base computation, are described and discussed.

Several classes of symbolic computing algorithms that utilize significant amount of computational resources are multiprecision integer arithmetic, polynomial arithmetic (like symbolic differentiation, Gröbner base computation, greatest common divisor), exact solution of linear equation systems, etc. Parallelization of symbolic computations is not easy to achieve. Symbolic computations tend to have unpredictable data dependencies, irregular data access patterns and varying parameters – all these make it difficult to predict patterns of memory and processor usage. Significant initiatives for parallel and distributed symbolic computations are reviewed in [2]. The simplest approach to benefit from the multi-core architectures is to have threads running on the same core and processes that are running on different cores. A possible reduction in time response can be obtained taking into account the fact that the communication between different threads will be done through shared-memory variables faster than between different processes by standard message passing interfaces. A redesign of the underlying algorithms in necessary.

We studied two cases. The first case refers to integer factorization, one of the most studied topics due to its application in modern cryptography. Many factorization algorithms have been developed but still none of them has been proven to be the best. We designed and realized a multi-process and multi-threaded implementation of the Multiple Polynomial Quadratic Sieve (MPQS [3]) algorithm. We have implemented the MPQS algorithm in C using GNU

Multiple Precision Arithmetic Library. The message passing is ensured by MPICH 2. A first attempt of implementation employed threads only for the sieving part of the algorithm, where each thread sieved over a part of the sieving interval. This involved a strong need for synchronization between threads, which in turn introduced some delays, but still the run-time performance obtained when factorizing large numbers was better than what we obtained with only MPI communications. In the second implementation each slave creates separate threads and each thread sieves over the entire sieving interval using a different polynomial. The main thread of the slave is responsible only for receiving the number to factorize, computing the factor base and creating the threads that will do the actual sieving. Each sieving thread is capable to communicate independently with the master process using MPI primitives. This approach introduces a growth in the run-time performance proportional to the number of threads per process used. The testing architecture is a HP ProLiant PC cluster with 28 processors on 7 chips. For performance evaluation we factorized numbers of maximum 60 digits using different number of slave nodes. By analyzing the response times we conclude that using multi-threading instead of MPI processes for the code components that are running on the same chip can lead to a significant speedup.

The second case is related to Gröbner basis computation. Our starting point is the paper [1] which describes a parallel implementation. We modified it to include a combination of threads and processes on nodes with multiple processors. We also used the master-slave paradigm. We chose Java and mpiJava based on MPICH 1.2 for message passing. The tests were done on the cluster mentioned in the previous paragraph. The tests performed showed an improvement in time response when using a multi-thread multi-process variant instead of only a multi-process one if the number of polynomial pairs is of order of tens.