

Explicit Sequential Programming for Implicit Parallel Performance on Many Cores

Devesh Tiwari (devesh.dtiwari@ncsu.edu) and Yan Solihin (solihin@ncsu.edu)

Electrical and Computer Engineering, NC State University Raleigh NC USA

Multi-cores are here and many cores are not far either. However, it remains unclear as to how many cores can be exploited to speed up sequential programs or already existing legacy codes which cannot be parallelized for many obvious reasons. Several parallel programming models try to exploit multi cores by making programmers write applications using parallel constructs and executing parallel sections on different cores, but such parallel programming models are not suited for all applications (e.g. inherently sequential programs, legacy sequential codes) and sequential programmers find it very hard to code their applications in such programming models that often put a lot of synchronization and communication burden on programmers. This paper proposes a novel concept of *Many Core Compiler (Many-CC)* framework that has the potential of high performance for sequential programs and legacy codes in many core architecture.

In the proposed framework, programmers will write programs as if they are writing sequential programs without having to know that their program would run on many-core architecture. Many-CC takes the program and divides it into many *phases*, each phase is to be executed on different core, we call each such phase a *phase-thread* (these phase threads need not be parallel they are just significant portion of the whole program decided based on number of cores available and over all control flow of program). Now, Many-CC analyzes each phase and decides two important sets for each phase-thread, 1) *input set*: set of all the program variables this phase will need to read to execute the phase correctly, 2) *Output set*: set of all the program variables this phase-thread will modify. Many-CC changes all program variables used in a phase to *phase-local variables*, that way for a particular phase all other phases are black boxes which operate on their own input set using their own phase-local variables (not conflicting with program variables or other phases' phase-local variables) and finally write to output set (set of program variables). There is one *master thread* that dictates each phase-thread when to write its output set variables. As soon as the program starts, all phase-threads start executing on different cores using values as read from input set (these values might be incorrect, but still phase threads continue working on phase-local variables), however it will not write its output set variables until the master-thread *signals* the phase-thread that its turn (as determined by program control flow) has come and it should now read input set variables again, execute and write output set variables, this execution would be called *final-run*. It should be noted that any phase thread could execute the same phase whenever it sees the change in values of input set variables. Such redundant executions have performance implications that will be discussed later. By the time the phase thread receives the *signal* from master thread, in the most fortunate case, it might have completed all the computation using the (speculative but) correct values of input set variables. In other case, it might be in the middle of phase computation with (speculative but) correct values of input set variables. Intuitively, the worst case would be when a phase-thread has to start over if it happened to read any incorrect input set variable. However, even this case might be a lot better than executing sequentially for many possible reasons. Executing the same thread many times will help in training the cache contents the best for a particular phase thread on its assigned core. Please note that multiple execution of same phase would be very similar as input set variables will change slowly and multiple executions will open opportunities for learning many phase specific details which are input set variable independent, thus many online and profiling optimizations can be applied to the final-run of phase thread because multiple similar executions should help in applying very effective run-time phase specific optimizations which would have not been possible or would have been ineffective otherwise. However, multiple executions of the same phase might also result in biased/incorrect learning of phase specific behavior (e.g. branch prediction), but simple criticality based techniques (e.g. identifying variables in the input set critical to all branches in that phase) can help in alleviating the problem. While one phase thread is waiting on system call or I/O operations other phases can learn about phase specific optimizations and store few expensive operation results for final run. This way, even worst-case final run might be much faster than normal sequential run. Working on phase-local variable concepts can potentially avoid much of complex coherence and consistency issues. This approach also hides synchronization and parallelization burden from programmers and even Many-CC has to perform minimal synchronization.

Finally, Many-CC framework can be seen as breaking the sequential program into multiple black boxes which execute on different cores multiple times using input set variables and writing the result to output set variable by doing the scratch pad computation on phase-local variables while ensuring the correctness by correct execution order of black boxes. This paper also makes the case that previously proposed related techniques like multiscalar, dynamic multithreading and run-ahead execution should be revamped aggressively in many core era.

It should be further noted that Many-CC framework is very intuitive and simple so it can be exposed to programmers. Programmers can make Many-CC job easy by writing programs close to Many-CC expected program structures (phase based black boxes), it is easy to program similar to that structure because Many-CC expects program structure to match with sequential programming notions and intuitions. More importantly, programmers do not have to think about data races, synchronization and communication unlike traditional parallel programming. Such a programming model (or compiler framework) provides parallel performance with deterministic execution and no race conditions. Therefore Many-CC framework can also be extended to a Many Core Black Box Sequential Programming Model that can fully exploit multi cores for sequential applications.