Automatic Parallelization of Programming Languages: Past, Present and Future

[Extended Abstract]

Wolfram Schulte, Nikolai Tillmann
Microsoft Research
Redmond, WA 98052, USA
schulte@microsoft.com, nikolait@microsoft.com

Automatic parallelization of modern object-oriented languages, like Java, C#, Python or JavaScript, is considered to be a grand challenge. But what is the challenge exactly? Let us simplify the discussion by focusing on loop parallelization only. As usual loop parallelization requires answering two questions: (1) is it worthwhile to parallelize a loop? (2) is it safe to parallelize a loop?

In the past, automatic parallelization has mainly focused on parallelizing loops of imperative programs at compile time. Answering (2) needs accurate information about which locations are read and written in each iteration. The good news is that for numeric algorithms, ie. algorithms which only operate on non-heap allocated collections of numeric values, we have a range of reasonable precise dependency analyses and tests. For present object-oriented languages these analyses, however, are not useful, since they do not capture the possible aliasing relationships between heap allocated objects. For understanding those relationships pointer analyses have been developed, but their results are usually too coarse grained to allow for safe auto-parallelization at compile time, or, if their result is fine grained enough, they do not scale.

Answering (1) at compile time is an even more difficult proposition: for instance we have to know how often a loop iterates, how the workload of the machine at runtime looks like, or whether the different iterations take the same amount of time. In the past, ie. when vector machines or SIMD machines were king, we auto-parallelized programs that had fixed inputs, statically bound methods, and fixed workloads. As a consequence, auto-parallelization was doable and partly successful. Today we would like to parallelize arbitrary object-oriented programs with unknown input sizes on standard multicore-machines with varying workloads.

Since auto-parallelization efforts seemed hopeless, the community moved on to make parallel programming explicit: many modern object-oriented languages introduce data and/or task parallelism, threads and/or actors, locks and/or message passing. But while this hardly addresses the writing of bug-free new concurrent programs, billions of lines of existing programs still await to be rewritten to exploit the new hardware. Furthermore there are millions of new programs in dynamic languages, like JavaScript, which are written today that cannot even express concurrency.

But what would change if we would auto-parallelize at runtime instead of at compile-time, ie. if we would use a tracing just-in-time (JIT) compiler?

A tracing JIT compiler has almost perfect information to answer (1). After collecting program traces at runtime, the JIT compiler knows which loop is hot, the worst case execution time of a single iteration, and dependencies of this iteration on global resources like IO; in addition it can ask the OS at runtime for the number of available processors and the current workload of the machine. A tracing JIT can thus start executing a loop sequentially. Provided a loop is hot and safe to parallelize, it can generate code for the parallel loop in a background thread. As soon as the parallel version is available, execution can switch from the sequential loop to its parallelized loop.

The opportunity to re-compile different versions at runtime also helps with addressing (2). By generating additional guards, a tracing JIT can safely parallelize a loop. Let us assume that we execute the loop in two phases. Phase one checks sequentially whether there are any dependencies between objects, expressed as potential write/write or read/write conflicts, which are extracted from previously gathered sequential traces. The iteration stops with index n as soon as a conflict is detected. After phase one, we know that the first $n-1$ iterations have no conflict, so we can safely execute them in parallel. After having finished these iterations, we run the same two phases for the second stride, and so on until the loop bound is reached. In the extreme case where there is a sequential dependency between the first and second iteration, n is 1, resulting in sequential execution. Thus this parallelization is always safe. Of course this scheme can be optimized: statically gathered dependency information can be exploited to minimize the conflict checking at run-time; automated theorem provers such as SMT solvers can be used at recompile time to further reduce the number potential conflicts by eliminating implied facts; the checking for possible conflicts, i.e. phase one, can not only be done sequentially but also in parallel; conflicts can also be scheduled as dependencies between tasks making the two phase protocol much more dynamic; the computed access paths can be stored instead of being recomputed, etc.

Are auto-parallelization JIT compilers the panacea for the future? We will report on our experience with auto-parallelized C# and JavaScript programs at the talk.

IWMSE'10, May 1, 2010, Cape Town, South Africa.
ACM 978-1-60558-964-0/10/05.