Autoparallelizer for a multi core x86 architecture

By David Livshin, published on December 18, 2018

Copyright © 2018 David Livshin. All rights reserved.
david.livshin@dalsoft.com

Table of Contents

Introduction

With the crop of the tools on the market that are related to parallel programming, one is conspicuously missing – the tool that will automatically generate parallel code, the auto-parallelizer. The tools available seems to be beating around the bush providing the user with the way to understand it problem, teaching how to parallelize the code, guiding and assisting with the parallelization but not really producing the parallel code or producing such a code with an extensive help from the user (like OpenMP). The automatic parallelization belongs to the future.

Why hasn't automatic parallelization become a reality?

The answer to that question is quite obvious – it is very difficult to create auto-parallelizer. And all the tools available, that are teaching, guiding, helping and assisting user in creation of the parallel code instead of creating such a code, seems to be a sign of capitulation in the efforts to create auto parallelizer.

From the tools available for parallel code development, the following two stands up.

OpenMP

OpenMP is a very powerful tool for creation of parallel code. However OpenMP requires user to be responsible for everything in the processors of the parallel code generation, including:

- specifying the code to be parallelized
  with OpenMP being very particular what code it accepts for processing
- providing proper parameters that guide the parallelization
- guaranteeing that parallelization is possible
  e.g. checking for race conditions

gcc

Supports true auto-parallelization, however lucks capability to treat any seriously complicated code. None of the examples in this article can be auto-parallelized by ***gcc***.

dco – auto-parallelizer from Dalsoft

Creation of the auto-parallelizer is very difficult but not impossible.

We conceived and implemented a solution to this problem. We developed an optimizing/parallelizing code system for the x86 family of processors. ***dco*** is a software package specifically designed to auto-parallelize x86 assembly code.

General Desription

dco is a x86 code optimizer-parallelizer. **PARALLator** is a part of ***dco*** that is responsible for auto-parallelization. **PARALLator** asumes identical x86 cores (Homogeneous Multicore) that use shared memory with Uniform Memory Access (UMA) a.k.a. Symetric Multi Processor (SMP):

each core has access to all the memory access time for any core is the same no matter the memory location.

dco shall be used to parallelize gcc-generated code. The programmer uses a compiler (C, Fortran etc.) to translate his code into x86 assembly code. This code would be used as an input to ***dco***. The output, generated by the ***dco***, will be a parallelized x86 assembly code that is logically identical to the original one; ***dco*** will extract portions of the code to be parallelized, rearranging the existing code if necessary, perform necessary verification for code being suitable for parallelization augmenting it by conditions to guarantee that and/or changing it to ensure that. To create the final object file the generated code should be assembled.

Note that ***dco*** does not require preprocessing or any other involvement from the user. It is fully automated and may be incorporated into makefiles or other product generation tools. Note also that use of ***dco*** doesn't require coding in x86 assembly language – it is possible to optimize arbitrary code that is coded in any high level language supported by the compiler (C, Fortran etc.).

dco is a compiler post-processor translation compiler generated assembly file into assembly file:

asm → asm

dco is designed to work with ***gcc*** compiler which generates an assembly output of the compiled code - the way it is achieved is by specifying ***-S*** option during compilation.

Assume that the compiler driver ***gcc*** is available on your system. To optimize the file '*test.c*' do the following:

```
gcc -S test.c
dco test.s -o otest.s
mv otest.s test.s
gcc -c test.s
rm test.s
```

1. ***gcc -S test.c*** compiles the input file '*test.c*' and generates assembly output file '*test.s*'. You may use other compiler options (to perform optimization etc.).
2. ***dco -i test.s -o otest.s*** optimizes the input file '*test.s*' generating as an output file '*otest.s*'.
3. ***mv otest.s test.s*** renames file '*otest.s*' to '*test.s*'.
4. ***gcc -c test.s*** assembles file '*test.s*' producing as an output object file '*test.o*'.
5. ***rm test.s*** deletes file '*test.s*'.

The described procedure may be easily incorporated into makefiles, batch files or other product generation tools. For example, makefiles often used to generated object files by specifying rules to translate C-source into object-file, e.g.

```
.c.o:
    $(CC) $(CFLAGS) -c $<
```

In order to incorporate ***dco*** the rule may be rewritten as:

```
.c.o:
    $(CC) $(CFLAGS) -S $<
    dco $*.s -o $*.so $(DCO_OPT)
    mv $*.so $*.s
    $(CC) $(CFLAGS) -c $*.s
    rm $*.s
```

Previous implementations

The approach described above allowed creation of optimizers for many different architectures:

- i860
- Alpha-AXP
- SHARC
- tigerSHARC
- StarCore
- x86

follow an appropriate link for the description of the particular product.

Current implementation

Current implementation (described in this article) - ***dco*** - is a x86 assembly code optimizer-parallelizer.

The parser for the input assembly code is based on the code for GNU gas and runs as a separate process

Optimizer/auto-parallelizer itself consists of 350K lines of C++ code.

Current implementation relies on OpenMP run-time library for basic operations: spawn/terminate team of threads, determine number of threads, determine number of a thread etc.

Utilized technology

In this section we describe some of the technology we developed and utilized in order to achieve auto-parallelization. Note that what follows is only a tiny fraction of the implemented algorithms and techniques.

Consider the following loop that we would like to parallelize:

```
for ( i = 0; i < cnt; )
 {
 x1 = 2.0 * x - 1.;
 if ( x1 < 1.0 )
  {
  b[i] = exp( x1 ) * cos( x1 );
  i = i + 3;
  x = x*2.;
  }
 else // if ( x1 >= 1. )
  {
  a[i] = sqrt( x1 ) * log( 1 / x1 );
  i = i + 2;
  x = x/2.;
  }
}
```

Note that this loop does not confirm to a requirement of OpenMP – loop must be “countable” - and therefore may not be processed by OpenMP.

In order to parallelize the above loop, ***dco*** needs to

- calculate loop count
- resolve memory dependency for memory writes **`b[i]`** and **`a[i]`**
- create code to calculate values needed at the entry to a thread: **`x`** and **`i`**

The solution to these problems shall:

- be efficient
  the time it take ***dco*** to generate the code for the described purpose shall be “reasonably small” - it is important to mention because ***dco*** code is often uses very complicated and expensive algorithms.
- generate slim code
  the code we generate will be executed as a part of the parallel program and in order for parallelization to be worth one’s while, the generated here code shall be “reasonable fast”.

- have no side affects allowed
  dco is capable to handle any side effects introduced by the code, except writing to non-scalar variables (e.g. arrays).

To generate desirable code we use “code slicing”. What is “code slicing” is widely known, however it doesn't seem that there is a practical algorithm to implement it. We had to bite the bullet and implemented very efficient and robust way to perform code slicing that will be demonstrated it in the following examples.

Compiling the above C code generated the following assembly fragment:

```
        jmp   .L6
.L19:
        movapd      %xmm3, %xmm0
        movsd       %xmm3, 24(%rsp)
        addl $3, %ebx
        call __exp_finite
        movsd       24(%rsp), %xmm3
        movsd       %xmm0, 16(%rsp)
        movapd      %xmm3, %xmm0
        call cos
        mulsd       16(%rsp), %xmm0
        cmpl %ebx, %r12d
        movsd       8(%rsp), %xmm2
        movapd      %xmm2, %xmm1
        movsd       %xmm0, (%r14,%rbp,8)
        jbe   .L18
.L6:
        movapd      %xmm1, %xmm2
        movsd       .LC1(%rip), %xmm4
        movl %ebx, %ebp
        addsd       %xmm1, %xmm2
        comisd      %xmm2, %xmm4
        movapd      %xmm2, %xmm3
        movsd       %xmm2, 8(%rsp)
        subsd       .LC0(%rip), %xmm3
        ja    .L19
        movsd       .LC0(%rip), %xmm0
        sqrtsd      %xmm3, %xmm6
        movsd       %xmm1, 16(%rsp)
        addl $2, %ebx
        divsd       %xmm3, %xmm0
```

```
        movsd     %xmm6, 8(%rsp)
        call __log_finite
        mulsd     8(%rsp), %xmm0
        cmpl %ebx, %r12d
        movsd     16(%rsp), %xmm1
        mulsd     .LC2(%rip), %xmm1
        movsd     %xmm0, (%r13,%rbp,8)
        ja    .L6
.L18:
```

Note that compiler generated assembly code seems to be convoluted, two exits from the loop: **`jbe .L18`** and **`ja .L6`**. ***dco*** converts this code into a "real" loop before generated the following code fragment:

Code to calculate loop count

Code to calculate number of iterations of the loop looks as following:

```
        movq $0,32(%rsp)
___dcox86_wl65761:
        addq $1,32(%rsp)
___dcox86_rlbl65755:
        movsd %xmm1,%xmm2
        movsd .LC1(%rip),%xmm4
        addsd %xmm1,%xmm2
        comisd %xmm2,%xmm4
        movsd %xmm2,40(%rsp)
___dcox86_rlbl65756:
        ja ___dcox86_rlbl65758
        movsd %xmm1,48(%rsp)
        addl $2,%ebx
        movsd 48(%rsp),%xmm1
        mulsd .LC2(%rip),%xmm1
        jmp ___dcox86_rlbl65760
___dcox86_rlbl65757:
        jmp ___dcox86_rlbl65760
___dcox86_rlbl65758:
        addl $3,%ebx
        movsd 40(%rsp),%xmm2
        movapd %xmm2,%xmm1
        jmp ___dcox86_rlbl65760
___dcox86_rlbl65759:
        jmp ___dcox86_rlbl65760
```

```
___dcox86_rlbl65760:
      cmpl %ebx,%r12d
      ja ___dcox86_wl65761
```

With `32(%rsp)` being set to the number of iterations that loop is executed (loop count). It was estimated (by the code slicer) that the cost for the generated code is 4.7% (execution time for that code is only 4.7% of the execution time of the whole loops). Note that generated code shown is not optimized – that will happen at the later stages of the parallelization.

Code to verify memory dependency

The original code has two distinct memory writes. In order to create parallel version of this code we shall verify that at executions of different threads there will be no writes to the same memory location. ***dco*** generates the following code:

```
___dcox86_el1014:
      movsd %xmm1,%xmm2
      movsd .LC1(%rip),%xmm4
      movl %ebx,%ebp
      addsd %xmm1,%xmm2
      comisd %xmm2,%xmm4
      movsd %xmm2,8(%rsp)
___dcox86_ebh1310:
      ja .L19
      movsd %xmm1,16(%rsp)
      addl $2,%ebx
      movsd 16(%rsp),%xmm1
      mulsd .LC2(%rip),%xmm1
      movsd %xmm0,(%r13,%rbp,8)
      jmp ___dcox86_eba14123
___dcox86_do899:
      jmp ___dcox86_eba14123
.L19:
      addl $3,%ebx
      movsd 8(%rsp),%xmm2
      movapd %xmm2,%xmm1
      movsd %xmm0,(%r14,%rbp,8)
      jmp ___dcox86_eba14123
___dcox86_do14166:
      jmp ___dcox86_eba14123
___dcox86_eba14123:
      cmpl %ebx,%r12d
```

Memory writes that shall be analyzed

`movsd %xmm0,(%r13,%rbp,8)` and `movsd %xmm0,(%r14,%rbp,8`

will be replaced by an appropriate code and the shown code will be significantly transformed. It was estimated (by the code slicer) that the cost for the generated code is 5% (execution time for that code is 5% of the execution time of the original loop) although the final code to verify memory dependency will be much more complex.

Code to generate values at the entry of a thread

Inside every thread the the original loop is executed for curtain number of iterations. It is assumed the every thread continues right from the point where the previous one terminated. Therefore the state of resources that loop generates by itself for it own consumption shall be provided at the entrance to the loop inside every thread. To achieve this, ***dco*** generates the following code:

```
___dcox86_el1014:
      movsd %xmm1,%xmm2
      movsd .LC1(%rip),%xmm4
      addsd %xmm1,%xmm2
      comisd %xmm2,%xmm4
      movsd %xmm2,8(%rsp)
___dcox86_ebh1310:
      ja .L19
      movsd %xmm1,16(%rsp)
      addl $2,%ebx
      movsd 16(%rsp),%xmm1
      mulsd .LC2(%rip),%xmm1
      jmp ___dcox86_eba14123
___dcox86_do899:
      jmp ___dcox86_eba14123
.L19:
      addl $3,%ebx
      movsd 8(%rsp),%xmm2
      movapd %xmm2,%xmm1
      jmp ___dcox86_eba14123
___dcox86_do14166:
      jmp ___dcox86_eba14123
___dcox86_eba14123:
```

Values needed at the entry to a thread are `%xmm1` and `%ebx`. The generated code will be executed in every thread ***T*** for the number of iterations equal to the sum of all iterations that all the bodies of the original loop were executed up to the point of getting to the thread ***T***. It was estimated (by the code slicer) that the cost for the generated code is 4.5% (execution time for that code is 4.5% of the execution time of the original loop being parallelized).

Cost for executing parallel code

In the previous sections we showed how to generate the different code fragments that shall be executed in the generated parallel code for a given loop. Now we will estimate the cost of executing of these code fragments. The comprehensive version of this analysis can be found in www.dalsoft.com/Calculating_number_of_cores_to_benefit_from_parallelization.pdf. The following is much simplified adaptation of this article.

Denote:

PET to be program execution time
PPET to be execution time of the parallelized program
NT to be number of threads (cores) utilized
CLC overhead of the code to calculate loop count
CMD overhead of the code to perform data flow analysis
CEV overhead of the all iterations of the code to calculate entry values

In the above mentioned article it was established that in order to benefit from parallelization (***PPET*** < ***PET***) the following shall hold:

$$(CLC + CMD + CEV) < PET \quad \&\& \quad NT > 1 + \frac{CLC+CMD}{PET-(CLC+CMD+CEV)}$$

which clearly shows that the decision if the given code can benefit from parallelization is far from being intuitive and require subtle and complicated analysis. It is possible that the answer will depend on number of cores available for parallelization. This analysis is performed by ***dco*** during parallelization.

It is interesting to note that although code to calculate entry values (***CEV*** overhead) is executed inside a thread, and code to perform data flow analysis (***CMD*** overhead) as well as code to calculate loop count (***CLC*** overhead) executed by a master process before executing threads, they appear about at “equal footing” in the above formulas.

results of parallelization

successfully parallelized

dco successfully parallelized great number of code fragment, benchmarks and applications. Here we list two such examples:

- code from the famous benchmark for which OpenMP solution produces unsatisfactory result.
- real code used to solve real problem written by "real" people, not necessarily professional programmers.

serial version of the NPB's EP code

EP - "Embarrassingly Parallel" suite from the NAS Parallel Benchmarks (NPB). Actually it appears to be one of the toughest cases of NPB and, definitely, there is nothing embarrassing in successfully parallelizing this benchmark.

```
do 140 i = 1, nk
      x1 = 2.d0 * x(2*i-1) - 1.d0
      x2 = 2.d0 * x(2*i) - 1.d0
      t1 = x1 ** 2 + x2 ** 2
      if (t1 .le. 1.d0) then
       t2  = sqrt(-2.d0 * log(t1) / t1)
       t3  = (x1 * t2)
        t4  = (x2 * t2)
       l  = max(abs(t3), abs(t4))
       q(l) = q(l) + 1.d0
       sx  = sx + t3
       sy  = sy + t4
      endif
 140   continue
```

The code, as it is presented in NPB serial suite, is not parallelizable. The line

```
q(l) = q(l) + 1.d0
```

introduces data dependency.

Solving this dependency by the means offered by OpenMP - making the problematic code “atomic” - produces very slow (although parallel) executable; that is due to the fact that atomic execution mode is very expensive on x86. The way it is solved by NAS in NPB OpenMP suite is by significantly altering the source of the benchmark. ***dco*** doesn't require user to change the source code and performs all necessary alterations "automatically".

Molecular Dynamics

Represents complicated real code used to solve real problem written by "real" people, not necessarily professional programmers (see the source code here). Although this code can be parallelized by OpenMP – but to do that requires expertise in parallel programming:

can, not a professional programmer, be trusted with that?

problems with parallelization

dco can successfully parallelize many code fragments that OpenMP doesn’t. One of such cases is treatment of the not structured blocks.

OpenMP treats only “structured blocks” (quote from the OpenMP spec): *The block of code must be structured in the sense that it must have a single point of entry at the top and a single point of exit at the bottom; i.e., branching in or out of a structured block is not allowed.*

dco can handle code with multiple points of exit in two ways.

treat all the iterations of the loop

Treating all the iterations of the loop in the parallelized code may lead to a "*conceptual problem*": parallel code may execute iterations of a loop that in sequential version of the code wouldn't be executed because of the exit from the loop beforehand. ***dco*** capable to detect that and deliver only the results up to the first exit from the loop. However ***dco*** is not capable to determine if exit from the loop is essential and no execution shall take place after it (e.g. due to a possible exception).

precompute loop count

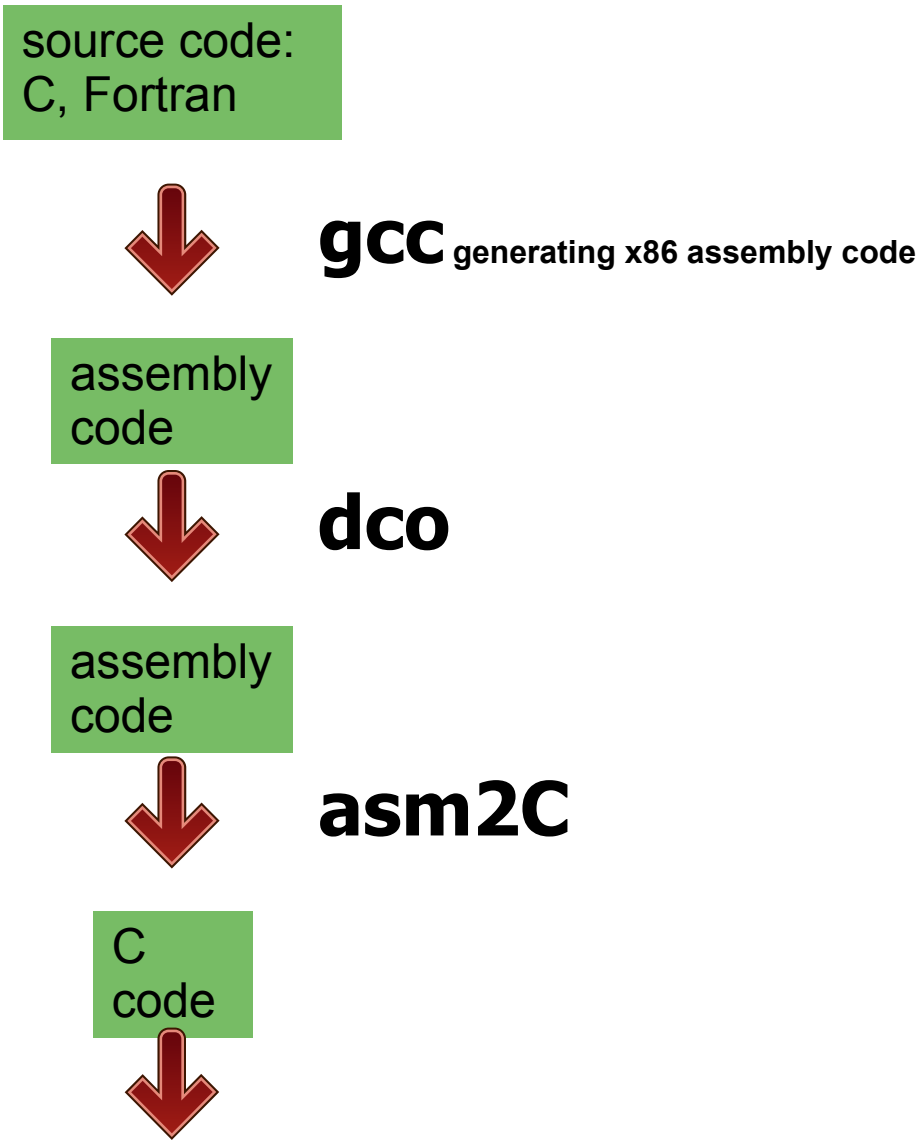
The more elegant solution is to precompute loop count, see this for example of doing that, and treat only that-many iterations. One of the *possible problems* with such a resolution is that overhead of loop count calculation may be unacceptably high. That happened when attempting to parallelize the following code-patterns (from "eqntott" benchmark)

```
int cmppth (BIT *p, BIT *q, int ninputs )
{
      int i;

      for (i = ninputs; i; i--, p++, q++)
            if (*p != *q)
                  if (*p < *q)
                        return (-1);
                  else
                        return (1);
      return (0);
}
```

auto-parallelizer for other architectures

dco performs auto-parallelization for x86 CPU. In order to apply this technology for other processors, the following scheme may be attempted:



The missing decompiler **asm2C** must be implemented - it translates x86 assembly code to C (which later will be fitted to OpenCL compiler for the target architecture). It is not clear yet how to do that although it doesn't have to be the full-blown decompiler as it can be invoked from within ***dco*** before assembly code generated. Beside that, decompilation technology is well known and understood.

The conclusions

John Dvorak once sad that “***artificial intelligence is the technology of the future and always will be***”. For the long time this could be sad about auto parallelization. Not any more: the auto parallelization became a reality, the future has arrived – welcome to the future!