Dalsoft

www.dalsoft.com

Auto-parallelizer for a multi core x86 architecture

David Livshin

www.dalsoft.com/Autoparallelizer_for_a_multi_core_x86_architecture.pdf

created: November 2018

In this presentation I will

- **Give** a brief overview of the current state of the (Auto)Parallelization technology
- **Talk** about the approach that was utilized by my auto-parallelizer
- **Discuss** some of the problems that I had to overcome during the implementation
- **Present** the results of AutoParallelization achieved and not achieved

OpenMP

very powerful but requires user to be responsible for everything

- specify the code to be parallelized
- provide proper parameters
- guarantee that parallelization is possible

gcc

no power provided by OpenMP but also no requirements imposed by OpenMP

dco

x86 code optimizer-parallelizer

PARALLator is responsible for auto-parallelization

- identical x86 cores (Homogeneous Multicore)
- 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

There is a crack, a crack in everything
That's how the light gets in.
Leonard Cohen

dco is a compiler post-processor

```
gcc -c -O2 foo.c
```

```
gcc -S -O2 foo.c
dco -i foo.s -o ofoo.s
mv ofoo.s foo.s
gcc -c foo.s
rm foo.s
```

Dalsoft

www.dalsoft.com

x86

AlphaPowered™

SHARC

STAR CORE

BRIGHTER DSP TECHNOLOGY!

implementation

- parser for assembly code
  - based on the code for GNU gas
  - runs as a separate process
- optimizer/auto-parallelizer
  - 350K lines of C++ code
- relies on OpenMP runtime library
  - basic operations:
    - spawn/terminate team of threads
    - determine number of threads
    - determine number of a thread

utilized technology

```
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.;
   }
 }
```

We need 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 shall be efficient

The generated code shall be slim

No side affects allowed

```
        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:
```

solution: code slicing

calculate loop count

```
        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
```

cost: 4.7%

memory dependency

```
___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
```

cost: 5%

values at the entry

```
___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:
```

cost: 4.5%

Dalsoft

www.dalsoft.com

the cost

based on www.dalsoft.com/Calculating_number_of_cores_to_benefit_from_parallelization.pdf

PET **P**rogram **E**xecution **T**ime ★ LC **L**oop **C**ount ★ NT **N**umber of **T**hreads

Overhead:

CLC **C**alculate **L**oop **C**ount ★ CMD **C**alculate **M**emory **D**ependency ★ CEV **C**alculate **E**ntry **V**alues

PPET (**P**arallel **P**rogram **E**xecution **T**ime) = $CLC + CMD + \frac{CEV}{LC} * \frac{LC}{NT} * (NT - 1) + \frac{PET}{NT}$

to benefit from parallelization (PPET < PET):

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

if CLC = 10%, CMD = 50%, CEV = 30% then

NT > 7

The difficult we do immediately;
the impossible takes longer.
William Tilton

results of parallelization

successfully parallelized

1. serial version of the NPB's EP code
2. Molecular Dynamics

not yet done

1. treating not structured blocks
2. support for other architectures

serial version of the NPB's EP code

EP - "Embarrassingly Parallel" suite from the NAS Parallel Benchmarks (NPB).

```
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 line: q(l) = q(l) + 1.d0 introduces data dependency that must be treated

- using OpenMP pragmas produces slow code
- 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

real code

used to solve real problem

written by "real" people, not necessarily professional programmers.

Can be parallelized by OpenMP – but to do that requires expertise in parallel programming:

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

not structured blocks

OpenMP treats "structured blocks": *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!*

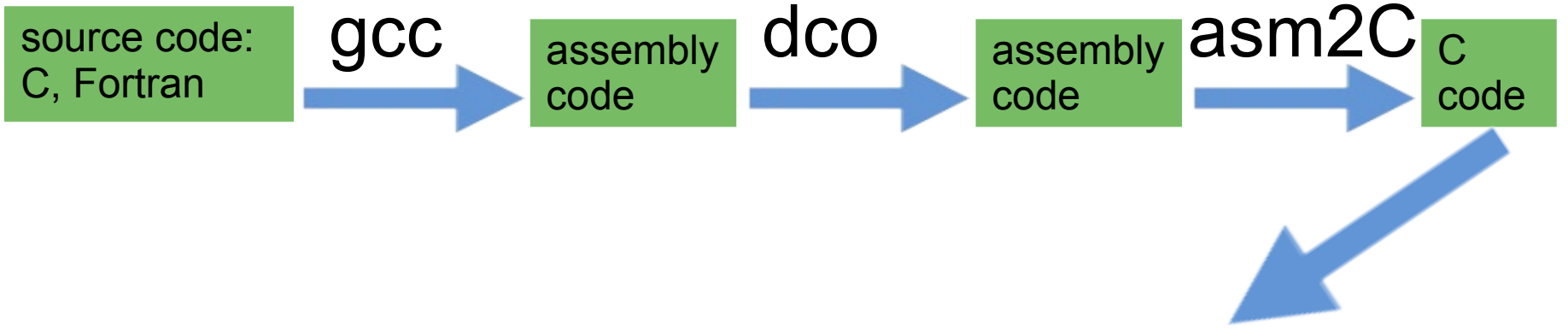
```
/*  from "eqntott" benchmark  */
int cmppth (BIT *p, BIT *q, int ninputs )
{
        int i;

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

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

treat all the iterations of the loop
conceptual problem: may lead to execution of the code that sequentially wouldn't be executed

precompute loop count and treat only that-many iterations
technical problem: the overhead may be unacceptably high

support for other architectures



use OpenCL for the target

Thank you!