COMPARATIVE ANALYSIS

OF CETUS

USING BENCHMARKS

by

Jan Sher Khan

A thesis submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Master of Science in Electrical & Computer Engineering

Summer 2020

COMPARATIVE ANALYSIS

OF CETUS

USING BENCHMARKS

by

Jan Sher Khan

Approved: _____

Rudolf Eigenmann, Ph.D.
Professor in charge of thesis on behalf of the Advisory Committee

Approved: _____

Mark Mirotznik, Ph.D.
Interim Chair of the Department of Electrical and Computer Engineering

Approved: _____

Levi T. Thompson, Ph.D.
Dean College of Engineering

Approved: _____

Douglas J. Doren, Ph.D.
Interim Vice Provost for Graduate & Professional Education and Dean of the Graduate College

ACKNOWLEDGMENTS

I would like to express my deepest gratitude to many people who have helped me make this work possible, especially my advisor Dr. Rudi Eigenmann who has brought me into this exciting area of automatic parallel compilers. He has always helped me to narrow down my research ideas and be focused. Dr. Chase Cotton has always been there for me since semester one. Without his tremendous help, I would not have reached my goal. My labmates, Akshay Bhosale and Parinaz Barakhshan who would participate and give feedbacks in every research group meeting.

I would especially like to thank my mom, dad, siblings and Sara for the unconditional love and help. They have been supportive of me throughout the process. My friends, Abdul Rehman, Abdul Haseeb, Imran Khan Panezai, Majid Aziz, Yuksel Karahan and Mostafa Rasulzada who have always been there to help.

I would also like to thank a few graduate fellows, Ismail Hakki Uluer, Zahra Vahdat, Emal Masood and Abdullah Goker whose friendships has immensely helped me become a better engineer. Last by not least, I thank all my friends whom name is not mentioned, I thank you all!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

With the increased induction of multicore architecture in the computing devices, the automatic parallelization has emerged as an important tool to make the source code optimized with the multicore architectures, which in return make the computing more efficient. Cetus, a source-to-source C compiler, helps laying the ground for the research on optimizing the automatic parallelization. It facilitates the source-to-source transformations of the C programs with important parallelization passes and their enabling techniques. It also aids researchers in analyzing the original and the transformed code. [1]

Having focus on for-loops parallelization, we analyze Cetus' performance by running it over various known benchmarks and some custom algorithms and later compare its output with one of the highest-quality source-to-source parallelizer named as Rose. The goal of our analysis is to identify algorithms/programs where Cetus fails to optimize compared to Rose or voice-versa. Some bugs and their solution in Cetus will also be identified. In the end, a future work to improve Cetus and its usage, a Cetus wish-list will be proposed which has all the identified bugs and weaknesses that can be fixed/improved in future.

Tools to be utilized are Cetus compiler, Rose compiler and GCC by GNU. Ubuntu 16.04.1 LTS operating system environment will be used on the MacBook Air 13-inch, Early 2015, with a hardware specification of 2.2 GHz, Intel core i7 processor with 8GB of RAM.

Chapter

INTRODUCTION

We will use two automatic source-to-source compilers Cetus and Rose. For Benchmark the NAS Parallel Benchmarks will be used. The Cetus and Rose's general introduction is further below.

1.1 Cetus

Cetus is a source-to-source C translating compiler. In addition to parallelizing the independently executing parts of the program, it also transforms the dependent parts to independent and later parallelizes them.

Cetus performs analysis of the input source code such as; Data dependence analysis, Reduction recognition, Array privatization analysis, symbolic range analysis, pointer alias analysis, symbolic expression manipulators. After the analysis it performs transformations of the source code such as; Induction variable substitution pass, Loop parallelize, program normalizers, loop outliner etc.

Cetus also has the GUI version which is very simple to us and thus helps improving the user interaction.

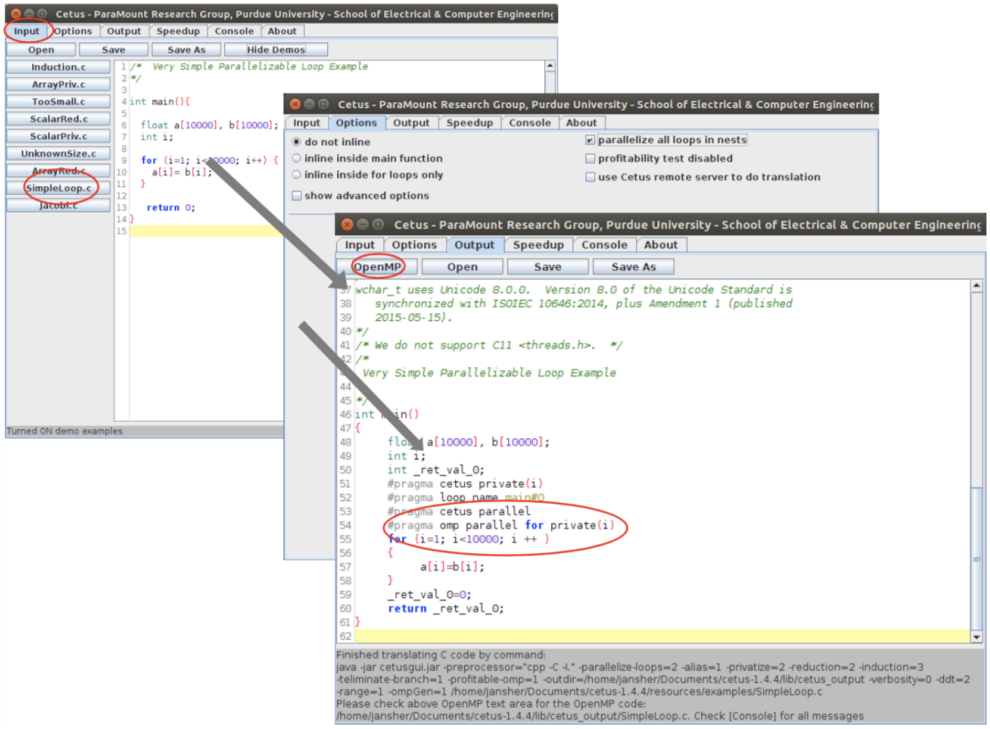

Figure 1.1: Cetus GUI, left to right source code transformation steps

1.2 Rose

Rose is also an automatic source-to-source translating open source compiler, suitable for people building tools which performs the analysis and transformation to improve any desired aspect of a software. Rose is a suitable tool to optimize large scale software with million or more lines. Rose can also analyze and optimize the source code of a complete project [3]. Rose built-in tools are:

- Loop analyzer
- OpenMP translator
- Source-to source instrumenter
- Symbolic complexity analyzer

- Array class abstraction optimizer

The Rose version to be used is: 10.9.10.54

1.3 Benchmarks

The benchmark to be used is the modified version of NAS Parallel Benchmarks which is implemented in C from the SNU NPB suite [4] It contains a set of C serial version of NPB and OpenMP version of NPB.

1.3.1 NAS Parallel Benchmarks

The Nasa Advanced Supercomputing (NAS) Parallel Benchmarks (NPB) are made to measure the performance of the parallel computers. The benchmarks are Computational Fluid Dynamics (CFD) application derivative. It's made of five kernels and three pseudo applications [5]:

1.3.1.1 Kernels

IS - Integer sort, Random memory access.

EP - Embarrassingly parallel.

CG - Conjugate Gradient, irregular memory access and communication.

MG - Multi-Grid on a sequence of meshes, long- and short-distance communication, memory intensive.

FT - Discrete 3D fast Fourier Transform, all-to-all communication.

1.3.1.2 Pseudo Applications

BT -Block Tri-diagonal solver.

SP - Scalar Penta-diagonal solver.

LU - Lower-Upper Gauss-Seidel solver.

1.4 Rose Repository:

The Rose test repository contains 99 C codes used to test various capabilities of an automatic compiler. The repository can be found in their GitHub repository [7]. It contains the serial c files and also contains their rose parallelized version of the serial files.

Chapter 2

IMPLEMENTATION

We will use NAS parallel benchmark and rose repository to test the Cetus compiler by running Cetus compiler on the benchmarks and Rose repository, later comparing them to the Rose compiler

2.1 Analysis and Findings

In all the cases, the Cetus compiler is run over the serial code found in the NAS parallel benchmark and the Rose repository to later analyze their outputs.

2.2 Analysis

NAS parallel benchmark and Rose repository from GitHub will be used to analyze the performance of Cetus by comparing its output with Rose's output. The output can be found in GitHub [8]

2.2.1 IS Benchmark

The IS benchmark contains a serial version and a manually parallelized code [4] by NPB. To begin comparison, we run Cetus over the serial version of IS benchmark. Our comparison mainly involves comparing loops to see if Cetus is successful in parallelizing the loops where Rose is successful or voice-versa.

The IS benchmark has overall 17 for-loops, one interesting for loop *(at is.c, line 673)* was found to be not parallelized by the Cetus but it was parallelized by the

Rose and a statement where Cetus performs forward propagation and Rose cannot. Examples discussed below:

2.2.1.1 Example 1

Here in the loop example given below, the for loop contains a switch statement inside its body and Rose is able to parallelize it, the code is given below:

2.2.1.1.1 Rose Output

```
#pragma omp parallel for private (i)
 for (i = 0; i <= 4; i += 1) {
        switch('A'){
        case 'S':
        test_index_array[i] = S_test_index_array[i];
        test_rank_array[i] = S_test_rank_array[i];
        break;
        case 'A':
        test_index_array[i] = A_test_index_array[i];
        test_rank_array[i] = A_test_rank_array[i];
        break;
        case 'W':
        test_index_array[i] = W_test_index_array[i];
        test_rank_array[i] = W_test_rank_array[i];
        break;
        case 'B':
        test_index_array[i] = B_test_index_array[i];
        test_rank_array[i] = B_test_rank_array[i];
        break;
        case 'C':
        test_index_array[i] = C_test_index_array[i];
        test_rank_array[i] = C_test_rank_array[i];
        break;
        case 'D':
        test_index_array[i] = D_test_index_array[i];
        test_rank_array[i] = D_test_rank_array[i];
        break;
        }}
```

Cetus cannot parallelize the above loop which contains a switch statement in it. The Cetus output is given below

2.2.1.1.2 Cetus Output

```
#pragma cetus private(i)
#pragma loop name main#0
for (i=0; i<5; i ++ ){
        switch ('A'){
                case 'S':
                test_index_array[i]=S_test_index_array[i];
                test_rank_array[i]=S_test_rank_array[i];
                break;
                case 'A':
                test_index_array[i]=A_test_index_array[i];
                test_rank_array[i]=A_test_rank_array[i];
                break;
                case 'W':
                test_index_array[i]=W_test_index_array[i];
                test_rank_array[i]=W_test_rank_array[i];
                break;
                case 'B':
                test_index_array[i]=B_test_index_array[i];
                test_rank_array[i]=B_test_rank_array[i];
                break;
                case 'C':
                test_index_array[i]=C_test_index_array[i];
                test_rank_array[i]=C_test_rank_array[i];
                break;
                case 'D':
                test_index_array[i]=D_test_index_array[i];
                test_rank_array[i]=D_test_rank_array[i];
                break;
        }}
```

After looking for the cause of Cetus not parallelizing the loop, it was found that the "*break;*" statement inside the switch statement is making the Cetus not parallelize the loop. To prove the "*break;*" statement is causing the problem, the "*break;*" statement is removed or commented out and then again ran Cetus over it.

We found out that our assumption was right, and Cetus indeed parallelized the for loop. It can be seen in the output that Cetus has disabled the parallelization because of its low profitability, once the number of iterations is increased to a significant amount then it will enable parallelization. This profitability feature helps avoiding the overhead where there is a loop with a smaller number of iterations. Its output is given below:

2.2.1.1.3 Cetus Output

```
#pragma cetus private(i)
#pragma loop name main#0
#pragma cetus parallel
  /*
  Disabled due to low profitability: #pragma omp parallel for private(i)
  */
  for (i=0; i<5; i ++ ){
        switch ('A'){
              case 'S':
              test_index_array[i]=S_test_index_array[i];
              test_rank_array[i]=S_test_rank_array[i];
              /* break; */
              case 'A':
              test_index_array[i]=A_test_index_array[i];
              test_rank_array[i]=A_test_rank_array[i];
              /* break; */
              case 'W':
              test_index_array[i]=W_test_index_array[i];
              test_rank_array[i]=W_test_rank_array[i];
              /*  break; */
              case 'B':
              test_index_array[i]=B_test_index_array[i];
              test_rank_array[i]=B_test_rank_array[i];
              /* break; */
              case 'C':
              test_index_array[i]=C_test_index_array[i];
              test_rank_array[i]=C_test_rank_array[i];
              /*  break; */
              case 'D':
```

```
            test_index_array[i]=D_test_index_array[i];
            test_rank_array[i]=D_test_rank_array[i];
            /* break; */
    }}
```

2.2.1.2 Example 2

Cetus removed an if-statement *"if ('A' != 'S')"* from inside a for-loop but Rose did not remove the same if-statement *(at is.c, line 734)* Their outputs are given below:

2.2.1.2.1 Rose Output

```
for (iteration = 1; iteration <= 10; iteration += 1) {
        if ('A' != 'S')
        printf("        %d\n",iteration);
        rank(iteration);
 }
```

2.2.1.2.2 Cetus Output

```
for (iteration=1; iteration<=10; iteration ++ ){
         printf("        %d\n", iteration);
         rank(iteration);
   }
```

The 'A' represents as the input class which compiler then substitutes, it seems both rose, and Cetus do forward substitution, but Cetus is able to go one step further and actually removes the if statement as evidently it can evaluate that this is always true.

2.2.2 EP Benchmark

The NAS's EP benchmark stands for Embarrassingly Parallel, which refers to the programs in which the workload can be easily divided into separate tasks for parallel execution. These separable tasks usually have no dependencies and

communication required between them while running in parallel. The EP benchmark contains a serial and a manually parallelized code by NPB.

The EP benchmark has overall 7 nested for-loops, we discovered *(at ep.c, line 197-210)* one interesting for-loop which was parallelized by Cetus but was not parallelized by Rose. Discussed below:

2.2.2.1 Example 1

2.2.2.1.1 Rose Output:

As it can be seen below, the rose output file "*rose_ep.c*", for-loop on line 197 is not parallelized by Rose.

```
for (i = 0; i <= 65535; i += 1) {
        x1 = 2.0 * x[2 * i] - 1.0;
        x2 = 2.0 * x[2 * i + 1] - 1.0;
        t1 = x1 * x1 + x2 * x2;
        if (t1 <= 1.0) {
        t2 = sqrt(- 2.0 * log(t1) / t1);
        t3 = x1 * t2;
        t4 = x2 * t2;
        l = ((fabs(t3) > fabs(t4)?fabs(t3) : fabs(t4)));
        q[l] = q[l] + 1.0;
        sx = sx + t3;
        sy = sy + t4;
        }
        }
```

2.2.2.1.2 Cetus Output:

As it can be seen below, the Cetus output file "ep.c", for-loop on line 252 is parallelized by Cetus. Cetus did a better job by first making a parallel region and later within, performed reduction over the for-loop.

```
#pragma cetus parallel
#pragma cetus private(i, l, t1, t2, t3, t4, x1, x2)
```

```
#pragma omp parallel private(i, l, t1, t2, t3, t4, x1, x2)
{
        double * reduce = (double * )malloc(10*sizeof (double));
        int reduce_span_0;
        for (reduce_span_0=0; reduce_span_0<10; reduce_span_0 ++ ){
                reduce[reduce_span_0]=0;
        }
        #pragma loop name main#3#1
        #pragma cetus reduction(+: sx, sy)
        #pragma cetus for
        #pragma omp for reduction(+: sx, sy)
        for (i=0; i<(1<<16); i ++ ){
                x1=((2.0*x[2*i])-1.0);
                x2=((2.0*x[(2*i)+1])-1.0);
                t1=((x1*x1)+(x2*x2));
                if (t1<=1.0){
                        t2=sqrt((( - 2.0)*log(t1))/t1);
                        t3=(x1*t2);
                        t4=(x2*t2);
                        l=((fabs(t3)>fabs(t4)) ? fabs(t3) : fabs(t4));
                        reduce[l]=(reduce[l]+1.0);
                        sx=(sx+t3);
                        sy=(sy+t4);
                }
        }
        #pragma cetus critical
        #pragma omp critical
        {
        for (reduce_span_0=0; reduce_span_0<10; reduce_span_0 ++ ){
                        q[reduce_span_0]+=reduce[reduce_span_0];
                }
        }
}
```

Table 2.1: CPU time for Cetus, Rose, and Manual parallelized Benchmark with Class "C"

Serial	Cetus Output	Rose Output	NAS Manual Parallel
22.30 sec	16.14	21.91	6.60

2.2.2.2 Example 2

We removed the OpenMP directives from the manually parallelized code by NAS NPB of the same loop in example 1 and ran Cetus over it and found a Cetus bug. Here we were interested in comparing the Cetus output with the manually parallelized version and so we did not run Rose. Given Below:

While running Cetus, options; *-profitable-omp=1 -parallelize-loops=4 -alias=3*, yields an output where an innermost for-loop of a nested for-loop is parallelized with a reduction directive which is unsupported by the OpenMP standards; ***reduction (+: qq[l], sx, sy)***. Since the option ***-parallelize-loops=4*** is used which makes Cetus parallelize every for-loop. The Cetus's reduction directive over an array variable qq is incorrect to even the Cetus's supported OpenMP version. The example is given below:

2.2.2.2.1 NAS Parallelized Code

```
#pragma omp threadprivate(x,qq)
#pragma omp parallel default(shared) private(k,kk,t1,t2,t3,t4,i,ik,x1,x2,l)
 {
  for (i = 0; i < NQ; i++) {
    qq[i] = 0.0;
  }
   #pragma omp for reduction(+:sx,sy) nowait
  for (k = 1; k <= np; k++) {
    kk = k_offset + k;
    t1 = S;
    t2 = an;
    for (i = 1; i <= 100; i++) {
      ik = kk / 2;
      if ((2 * ik) != kk) t3 = randlc(&t1, t2);
      if (ik == 0) break;
      t3 = randlc(&t2, t2);
      kk = ik;
    }
    if (timers_enabled) timer_start(2);
    vranlc(2 * NK, &t1, A, x);
```

```
    if (timers_enabled) timer_stop(2);
    if (timers_enabled) timer_start(1);
    for (i = 0; i < NK; i++) {
      x1 = 2.0 * x[2*i] - 1.0;
      x2 = 2.0 * x[2*i+1] - 1.0;
      t1 = x1 * x1 + x2 * x2;
      if (t1 <= 1.0) {
        t2   = sqrt(-2.0 * log(t1) / t1);
        t3   = (x1 * t2);
        t4   = (x2 * t2);
        l    = MAX(fabs(t3), fabs(t4));
        qq[l] = qq[l] + 1.0;
        sx   = sx + t3;
        sy   = sy + t4;
      }}
    if (timers_enabled) timer_stop(1);
  }
  for (i = 0; i < NQ; i++) {
    #pragma omp atomic
    q[i] += qq[i];
  }}
```

2.2.2.2.2 Cetus Output

```
        #pragma cetus private(i)
        #pragma loop name main#3
        #pragma cetus parallel
        /* Disabled due to low profitability: #pragma omp parallel for private(i) */
        for (i=0; i<10; i ++ ){
                qq[i]=0.0;
        }
        #pragma cetus private(i, ik, k, kk, l, t3, t4, x1, x2)
        #pragma loop name main#4
        #pragma cetus reduction(+: qq[l], sx, sy)
for (k=1; k<=np; k ++ )  {
                kk=(k_offset+k);
                t1=2.71828183E8;
                t2=an;
                #pragma loop name main#4#0
                for (i=1; i<=100; i ++ ) {
                        ik=(kk/2);
                        if ((2*ik)!=kk) {
                                t3=randlc( & t1, t2);
```

```
                }
                if (ik==0) {
                        break;
                }
                t3=randlc( & t2, t2);
                kk=ik;
        }
        if (timers_enabled) {
                timer_start(2);
        }
        vranlc(2*(1<<16),  & t1, 1.220703125E9, x);
        if (timers_enabled){
                timer_stop(2);
        }
        if (timers_enabled){
                timer_start(1);
        }
        #pragma cetus private(i, l, t1, t2, t3, t4, x1, x2)
        #pragma loop name main#4#1
        #pragma cetus reduction(+: qq[l], sx, sy)
        #pragma cetus parallel
        #pragma omp parallel for private(i, l, t1, t2, t3, t4, x1, x2) reduction(+:
qq[l], sx, sy)
        for (i=0; i<(1<<16); i ++ ) {
                x1=((2.0*x[2*i])-1.0);
                x2=((2.0*x[(2*i)+1])-1.0);
                t1=((x1*x1)+(x2*x2));
                if (t1<=1.0) {
                        t2=sqrt((( - 2.0)*log(t1))/t1);
                        t3=(x1*t2);
                        t4=(x2*t2);
                        l=((fabs(t3)>fabs(t4)) ? fabs(t3) : fabs(t4));
                        qq[l]=(qq[l]+1.0);
                        sx=(sx+t3);
                        sy=(sy+t4);
                }}
        if (timers_enabled){
                timer_stop(1);
        }}
```

Error Message shown in terminal (GCC version 4.9):

```
ep.c:256:81: error: expected ‘)’ before ‘[’ token
  #pragma omp parallel for private(i, l, t1, t2, t3, t4, x1, x2) reduction(+: qq[l], sx, sy)
```

```
                                                                    ^
ep.c:256:75: error: user defined reduction not found for ‘qq’
  #pragma omp parallel for private(i, l, t1, t2, t3, t4, x1, x2) reduction(+: qq[l], sx, sy)
```

This type of reduction is not supported by the OpenMP 4.0, but the support gets started from OpenMP 4.5.

2.2.3 CG Benchmark

The CG benchmark contains a serial version and a manually parallelized code by NPB [4]. To begin comparison, we run Cetus over the serial version of CG benchmark. Our comparison mainly involves comparing for-loops to see if Cetus is successful in parallelizing the loops where Rose is successful or voice-versa.

The CG benchmark has overall 43 for-loops, one interesting for-loop *(at cg.c, line 887)* was found to be not parallelized by the Cetus but parallelized by the Rose. Discussed below:

2.2.3.1 Example 1

2.2.3.1.1 Rose Output

```
#pragma omp parallel for private (k)
 for (k = 0; k <=  *nzv - 1; k += 1) {
        if (iv[k] == i) {
        v[k] = val;
        set = true;
        }}
```

It should be noted that when rose choses to parallelizes the above loop it violates the data dependence on variable *‘set’*. Cetus as shown in its output below is not parallelizing the same loop:

2.2.3.1.2 Cetus Output

```
#pragma cetus private(k)
#pragma loop name vecset#0
```

```
for (k=0; k<( * nzv); k ++ ){
        if (iv[k]==i){
                v[k]=val;
                set=true;
        }}
```

2.2.4 MG Benchmark

The MG benchmark contains a serial version and a manually parallelized code by NPB [4]. The MG benchmark has overall 77 for-loops, we discovered two interesting for-loops that were well parallelized by the Cetus but were not parallelized by Rose. They are discussed below:

2.2.4.1 Example 1

The nested for-loop inside the rprj3() function, on line 632 and 640 is parallelized by Cetus but is not parallelized by the Rose.

2.2.4.1.1 Cetus Output

```
#pragma cetus private(i1, i2, i3, j1, j2, j3, x2, y2)
#pragma loop name rprj3#0
for (j3=1; j3<(m3j-1); j3 ++ ){
        i3=((2*j3)-d3);
        #pragma cetus private(i1, i2, j1, j2, x2, y2)
        #pragma loop name rprj3#0#0
        for (j2=1; j2<(m2j-1); j2 ++ ){
                i2=((2*j2)-d2);
                #pragma cetus private(i1, j1)
                #pragma loop name rprj3#0#0#0
                #pragma cetus parallel
                #pragma omp parallel for if((10000<(-4L+(5L*m1j)))) private(i1, j1)
                for (j1=1; j1<m1j; j1 ++ ){
                        i1=((2*j1)-d1);
x1[i1]=(((r[i3+1][i2][i1]+r[i3+1][i2+2][i1])+r[i3][i2+1][i1])+r[i3+2][i2+1][i1])
;
y1[i1]=(((r[i3][i2][i1]+r[i3+2][i2][i1])+r[i3][i2+2][i1])+r[i3+2][i2+2][i1]);
                }
                #pragma cetus private(i1, j1, x2, y2)
```

```
                #pragma loop name rprj3#0#0#1
                #pragma cetus parallel
  #pragma omp parallel for if((10000<(-11L+(6L*m1j)))) private(i1, j1, x2, y2)
                for (j1=1; j1<(m1j-1); j1 ++ ){
                        i1=((2*j1)-d1);
y2=(((r[i3][i2][i1+1]+r[i3+2][i2][i1+1])+r[i3][i2+2][i1+1])+r[i3+2][i2+2][i1+1
]);
x2=(((r[i3+1][i2][i1+1]+r[i3+1][i2+2][i1+1])+r[i3][i2+1][i1+1])+r[i3+2][i2+1]
[i1+1]);
s[j3][j2][j1]=((((0.5*r[i3+1][i2+1][i1+1])+(0.25*((r[i3+1][i2+1][i1]+r[i3+1][i2+
1][i1+2])+x2)))+(0.125*((x1[i1]+x1[i1+2])+y2)))+(0.0625*(y1[i1]+y1[i1+2])));
}}}
```

2.2.4.1.2 Rose Output

```
for (j3 = 1; j3 <= m3j - 1 - 1; j3 += 1) {
        i3 = 2 * j3 - d3;
        for (j2 = 1; j2 <= m2j - 1 - 1; j2 += 1) {
        i2 = 2 * j2 - d2;
        for (j1 = 1; j1 <= m1j - 1; j1 += 1) {
        i1 = 2 * j1 - d1;
        x1[i1] = r[i3 + 1][i2][i1] + r[i3 + 1][i2 + 2][i1] + r[i3][i2 + 1][i1] + r[i3 +
2][i2 + 1][i1];
        y1[i1] = r[i3][i2][i1] + r[i3 + 2][i2][i1] + r[i3][i2 + 2][i1] + r[i3 + 2][i2 +
2][i1];
        }
        for (j1 = 1; j1 <= m1j - 1 - 1; j1 += 1) {
        i1 = 2 * j1 - d1;
        y2 = r[i3][i2][i1 + 1] + r[i3 + 2][i2][i1 + 1] + r[i3][i2 + 2][i1 + 1] + r[i3 +
2][i2 + 2][i1 + 1];
        x2 = r[i3 + 1][i2][i1 + 1] + r[i3 + 1][i2 + 2][i1 + 1] + r[i3][i2 + 1][i1 + 1]
+ r[i3 + 2][i2 + 1][i1 + 1];
        s[j3][j2][j1] = 0.5 * r[i3 + 1][i2 + 1][i1 + 1] + 0.25 * (r[i3 + 1][i2 + 1][i1]
+ r[i3 + 1][i2 + 1][i1 + 2] + x2) + 0.125 * (x1[i1] + x1[i1 + 2] + y2) + 0.0625 *
(y1[i1] + y1[i1 + 2]);
        }}}
```

It can be seen that Rose is unable to parallelize the loops, while on other hand Cetus does a better job as those loops are clearly parallelizable.

2.2.4.2 Example 2

The for-loops inside the *norm2u3() function, line 855-856,* When passed through Cetus, it is unable to parallelize them. These loops consist of some intrinsic functions and a conditional statement. To make it easily understandable, we made a similar but smaller pseudo code/program with common pattern/functionality. The Cetus output for the new program was unsurprisingly similar; It could not parallelize the loops at all. The output for the Rose compiler was not parallelized too.

To find out why Cetus failed, in the given code below, the code contains intrinsic math functions *pow()*, *fabs()* and a conditional statement which assigns greatest value of variable *a* or *my_rnmu* to the variable *my_rnmu*. In the analysis it was found that Cetus cannot detect a reduction operation when an intrinsic math function is in the same line on the right hand of a reduction variable.

```
for (i =0; i<limit; i++ ){
    s = s + pow(i, 2.0);
    }
```

While if the syntax is changed a little then it is detected by Cetus as a reduction operation, as given below:

```
#pragma omp parallel for reduction(+: s)
    for (i =0; i<limit; i++ ){
     s +=  pow(i, 2.0);
    }
```

It can also parallelize an intrinsic math function as there occurs no dependency inside the loop, given below:

```
#pragma omp parallel for private(i, s)
    for (i =0; i<limit; i++ ){
  s =  pow(i, 2.0);
    }
```

From the above examples, it can be extracted that Cetus cannot parallelize or apply reduction directive if there exists one specific kind of pattern where the potential reduction variable has an intrinsic math function beside it.

After stumbling upon the manually parallelized code by SNU NPB, Cetus also cannot detect and apply the OpenMP Max or Min reduction directive to a for-loop which contains a Min or Max conditional statement given below:

```
for (i =0; i<limit; i++ ){
my_rnmu = (a > my_rnmu) ? a : my_rnmu;
}
```

The above conditional statement simply means the variable my_rnmu will be assigned the maximum value among a or my_rnmu variables. If the above loop is parallelized manually, the OpenMP directive applied are given below as:

```
#pragma omp parallel for reduction(max:my_rnmu)
    for (i =0; i<limit; i++ ){
    my_rnmu = (a > my_rnmu) ? a : my_rnmu;
    }
```

The Cetus and Rose outputs are given as under:

2.2.4.2.1 Cetus Output

```
#pragma cetus private(a, i1, i2, i3)
#pragma loop name norm2u3#0
for (i3=1; i3<(n3-1); i3 ++ ){
        #pragma cetus private(a, i1, i2)
        #pragma loop name norm2u3#0#0
        for (i2=1; i2<(n2-1); i2 ++ ){
                #pragma cetus private(a, i1)
                #pragma loop name norm2u3#0#0#0
                for (i1=1; i1<(n1-1); i1 ++ ){
                        s=(s+pow(r[i3][i2][i1], 2.0));
                        a=fabs(r[i3][i2][i1]);
                        if (a>( * rnmu)){
                                ( * rnmu)=a;
                        }}}}
```

2.2.4.2.2 Rose Output

```
for (i3 = 1; i3 <= n3 - 1 - 1; i3 += 1) {
        for (i2 = 1; i2 <= n2 - 1 - 1; i2 += 1) {
        for (i1 = 1; i1 <= n1 - 1 - 1; i1 += 1) {
        s = s + pow(r[i3][i2][i1],2.0);
        a = fabs(r[i3][i2][i1]);
        if (a >  *rnmu)
        *rnmu = a;
        }}}
```

2.2.5 FT Benchmark

The FT benchmark contains a serial version and a manually parallelized code by NPB. To begin comparison, we run Cetus over the serial version of FT benchmark. Our comparison mainly involves comparing for-loops to see if Cetus is successful in parallelizing the loops where Rose is successful or voice-versa.

The FT benchmark has overall 39 for-loops, some interesting for-loops were found to be not parallelized by the Cetus but parallelized by the Rose.

Another interesting finding was that Cetus was not able to run the FT benchmark but later after some changes to the benchmark made it run. Which will also be discussed, and all the examples are discussed below:

2.2.5.1 Example 1

A for-loop in the file auxfnct.c, line 122 is parallelized by the Rose but is not parallelized by the Cetus. There appears to be a race condition on the variable *csum_temp*, it's really a reduction and this parallelization appears to be incorrect. Rose and Cetus outputs are given below:

2.2.5.1.1 Rose Output

```
#pragma omp parallel for private (i1,ii,ji,ki,i) firstprivate (d1,d2,d3)
 for (i = 1; i <= 1024; i += 1) {
```

```
        i1 = i;
        ii = i1 % d1;
        ji = 3 * i1 % d2;
        ki = 5 * i1 % d3;
        csum_temp = ((dcomplex ){csum_temp . real + u[ki][ji][ii] . real, csum_temp .
imag + u[ki][ji][ii] . imag});
         }
```

2.2.5.1.2 Cetus Output

```
#pragma cetus private(i, i1, ii, ji, ki)
#pragma loop name CalculateChecksum#0
  for (i=1; i<=1024; i ++ ){
        i1=i;
        ii=(i1%d1);
        ji=((3*i1)%d2);
        ki=((5*i1)%d3);
  csum_temp=(dcomplex){csum_temp.real+u[ki][ji][ii].real,
csum_temp.imag+u[ki][ji][ii].imag};
          }
```

2.2.5.2 Example 2

In the file "*fft3d.c*", *line 161* there are two nested loops which are parallelized by the Rose but are not by the Cetus. Both Cetus and Rose outputs are given below:

2.2.5.2.1 Rose Output

The values of *n1 = 256*, *n2 = 256*, *n3 = 128* and *blkp = 33*.

```
#pragma omp parallel for private (i,j)
        for (j = bls; j <= ble; j += 1) {
#pragma omp parallel for private (i)
        for (i = 0; i <= n1 - 1; i += 1) {
        plane[j - bls + blkp * i] = x[k][j][i];
        }}
```

2.2.5.2.2 Cetus Output

```
#pragma cetus private(i, j)
#pragma loop name fftXYZ#0#0#0
```

```
for (j=bls; j<=ble; j ++ ) {
        #pragma cetus private(i)
        #pragma loop name fftXYZ#0#0#0#0
        for (i=0; i<n1; i ++ ){
                plane[(j-bls)+(blkp*i)]=x[k][j][i];
                }}
```

Later, in the "*fft3d.c*", there is another kind of nested for-loop where the outer loop is parallelized by the Rose but is not by the Cetus. Their outputs are given below:

2.2.5.2.3 Rose Output

```
#pragma omp parallel for private (i,j) firstprivate (ble)
        for (i = 0; i <= n3 - 1; i += 1) {
#pragma omp parallel for private (j)
        for (j = bls; j <= ble; j += 1) {
        plane[j - bls + blkp * i] = x[i][k][j];
        }}
```

2.2.5.2.4 Cetus Output

```
#pragma cetus private(i, j)
 #pragma loop name fftXYZ#2#0#0
 for (i=0; i<n3; i ++ ){
         #pragma cetus private(j)
        #pragma loop name fftXYZ#2#0#0#0
         #pragma cetus parallel
        #pragma omp parallel for if((10000<((4L+(3L*ble))+(-3L*bls)))) private(j)
        for (j=bls; j<=ble; j ++ ) {
                plane[(j-bls)+(blkp*i)]=x[i][k][j];
                }}
```

Cetus used to break when ran on the FT benchmark and will show an error in the terminal. Below:

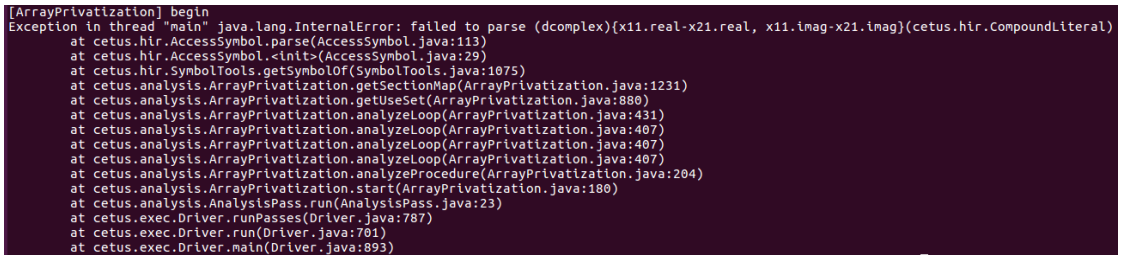

Figure 2.1: Cetus output in the terminal

After bringing a small change inside a loop which is in file "*fft3d.c*" solved the issue. The code is given below:

```
for (k = 0; k <= lk - 1; k++) {
        for (j = 0; j < vlen; j++) {
        x11 = scr[i11+k][j];
        x21 = scr[i12+k][j];
        x[i21+k][j] = dcmplx_add(x11, x21);
        x[i22+k][j] = dcmplx_mul(u1,dcmplx_sub(x11, x21));
        }}
```

According to the observation the Cetus was unable to interpret the line which contains the following statement:

```
x[i22+k][j] = dcmplx_mul(u1,dcmplx_sub(x11, x21));
```

After splitting the above statement into two statements as given below, the Cetus ran successfully.

```
dcomplex j444 = dcmplx_sub(x11, x21);
 x[i22+k][j] = dcmplx_mul(u1, j444);
```

Cetus was not able to process the complex hash functions and structs. So, the above simplification helped make it run.

2.2.6 LU Benchmark

The LU benchmark contains a serial version and a manually parallelized code by NPB [4]. To begin comparison, we run Cetus over the serial version of LU

benchmark. Our comparison mainly involves comparing for-loops to see if Cetus is successful in parallelizing the loops where Rose is successful or voice-versa.

The LU benchmark has overall 172 for-loops, one interesting for-loop was found to be parallelized by the Cetus and Rose, but Rose was unable to perform reduction over an array. Discussed below:

2.2.6.1 Example 1

Here it can be seen in the file *"l2norm.c", line 57* that Rose is parallelizing the for-loop but is not performing reduction over an array "sum[]". Cetus on the other hand parallelizes the outer loops and performs reduction over the array "sum[]" which is in the inner loop. Rose and Cetus outputs given below:

2.2.6.1.1 Rose Output

```
for (k = 1; k <= nz0 - 1 - 1; k += 1) {
        for (j = jst; j <= jend - 1; j += 1) {
        for (i = ist; i <= iend - 1; i += 1) {

#pragma omp parallel for private (m)
        for (m = 0; m <= 4; m += 1) {
        sum[m] = sum[m] + v[k][j][i][m] * v[k][j][i][m];
        }}}}
```

2.2.6.1.2 Cetus Output

```
#pragma cetus private(i, j, k, m)
#pragma loop name l2norm#1
#pragma cetus reduction(+: sum[m])
#pragma cetus parallel
#pragma omp parallel for if((10000<((((((((((((-5L+(-
6L*jend))+(6L*jst))+(3L*nz0))+((-
36L*iend)*jend))+((36L*iend)*jst))+((36L*ist)*jend))+((-
36L*ist)*jst))+((3L*jend)*nz0))+((-3L*jst)*nz0))+(((18L*iend)*jend)*nz0))+(((-
18L*iend)*jst)*nz0))+(((-18L*ist)*jend)*nz0))+(((18L*ist)*jst)*nz0)))) private(i, j, k,
m) reduction(+: sum[m])
```

```
  for (k=1; k<(nz0-1); k ++ ){
        #pragma cetus private(i, j, m)
        #pragma loop name l2norm#1#0
        #pragma cetus reduction(+: sum[m])
        #pragma cetus parallel
        #pragma omp parallel for if((10000<((((((1L+(3L*jend))+(-
3L*jst))+((18L*iend)*jend))+((-18L*iend)*jst))+((-18L*ist)*jend))+((18L*ist)*jst))))
private(i, j, m) reduction(+: sum[m])
        for (j=jst; j<jend; j ++ ){
              #pragma cetus private(i, m)
              #pragma loop name l2norm#1#0#0
              #pragma cetus reduction(+: sum[m])
              #pragma cetus parallel
              #pragma omp parallel for if((10000<((1L+(18L*iend))+(-18L*ist))))
private(i, m) reduction(+: sum[m])
              for (i=ist; i<iend; i ++ ){
                    #pragma cetus private(m)
                    #pragma loop name l2norm#1#0#0#0
                    #pragma cetus parallel
                    /*
        Disabled due to low profitability: #pragma omp parallel for private(m)
                    */
                    for (m=0; m<5; m ++ ){
                          sum[m]=(sum[m]+(v[k][j][i][m]*v[k][j][i][m]));
                    }
}}}
```

2.2.6.2 Example 2

In the file *"pintgr.c", line 114* Cetus has performed dead code elimination by omitting all the if-statements while on the other hand Rose has not omitted them. Dead code elimination usually helps making code compact thus using less processor and making it efficient than before.

2.2.6.2.1 Rose Output

```
  if (jbeg == ji1) {
#pragma omp parallel for private (i,k)
        for (k = ki1; k <= ki2 - 1; k += 1) {
#pragma omp parallel for private (i)
```

```
        for (i = ibeg; i <= ifin - 1; i += 1) {
        phi1[k][i] = 0.40e+00 * (u[k][jbeg][i][4] - 0.50 * (u[k][jbeg][i][1] *
u[k][jbeg][i][1] + u[k][jbeg][i][2] * u[k][jbeg][i][2] + u[k][jbeg][i][3] *
u[k][jbeg][i][3]) / u[k][jbeg][i][0]);
        }}}
```

2.2.6.2.2 Cetus Output

```
#pragma cetus private(i, k)
  #pragma loop name pintgr#4
  #pragma cetus parallel
  #pragma omp parallel for if((10000<(((((((1L+(-
3L*ki1))+(3L*ki2))+((3L*ii1)*ki1))+((-3L*ii1)*ki2))+((-
3L*ii2)*ki1))+((3L*ii2)*ki2)))) private(i, k)
  for (k=ki1; k<ki2; k ++ ){
        #pragma cetus private(i)
        #pragma loop name pintgr#4#0
        #pragma cetus parallel
        #pragma omp parallel for if((10000<((1L+(-3L*ii1))+(3L*ii2)))) private(i)
        for (i=ibeg; i<ifin; i ++ ){
                phi1[k][i]=(0.4*(u[k][jbeg][i][4]-
((0.5*(((u[k][jbeg][i][1]*u[k][jbeg][i][1])+(u[k][jbeg][i][2]*u[k][jbeg][i][2]))+(u[
k][jbeg][i][3]*u[k][jbeg][i][3])))/u[k][jbeg][i][0])));  }}
```

2.2.7 DC Benchmark

The DC benchmark contains a serial version and a manually parallelized code by NPB [4]. To begin comparison, we run Cetus over the serial version of DC benchmark. Our comparison mainly involves comparing for-loops to see if Cetus is successful in parallelizing the loops where Rose is successful or voice-versa.

The DC benchmark has overall 87 for-loops, there are two interesting for-loops found to not be parallelized by the Cetus but parallelized by the Rose. Discussed below:

2.2.7.1 Example 1

The first for-loop in the file *"adc.c", line 138* which is parallelized by the Rose but is not parallelized by the Cetus is given below:

2.2.7.1.1 Rose Output

```
#pragma omp parallel for private (i) firstprivate (LARGE_NUM,maxprmfctr)
 for (i = 0; i <= maxprmfctr; i += 1) {
        while(expons[i] > 0){
        LCM *= i;
        if (LCM > LARGE_NUM / maxprmfctr)
        return LCM;
        expons[i]--;
        }}
```

2.2.7.1.2 Cetus Output

```
#pragma cetus private(i)
#pragma loop name GetLCM#2
   for (i=0; i<=maxprmfctr; i ++ ){
         while (expons[i]>0) {
                LCM*=i;
                if (LCM>(LARGE_NUM/maxprmfctr)) {
                        return LCM;
                }
                expons[i] -- ;
        } }
```

2.2.7.2 Example 2

The second for-loop in the file *"rbt.c", line 9* which is parallelized by the Rose but is not parallelized by the Cetus is given below:

2.2.7.2.1 Rose Output

```
#pragma omp parallel for private (i)
 for (i = 0; i <= n - 1; i += 1) {
        if (a[i] < b[i])
        return - 1;
        else if (a[i] > b[i])
```

```
        return 1;
}
```

2.2.7.2.2 Cetus Output

```
#pragma loop name KeyComp#0
  for (i=0; i<n; i ++ ){
        if (a[i]<b[i]){
                return ( - 1);
        }
        else{
                if (a[i]>b[i]){
                return 1;
}}}
```

In both *Example 1* and *Example 2* it is observed that the for-loops have return statements in them. According to OpenMP, the return statements are not supported inside parallel loops [6]. Hence it can be observed the Cetus strictly follows the OpenMP rules. What's more, the Rose parallelized form seems incorrect.

2.2.8 Rose Repository

Some general Cetus related bugs will be discussed which are experienced after Cetus is run over some mini programs. These programs are taken from Rose repository in GitHub [7].

2.2.8.1 Loops with multiple variables in the header *(comma operator)*

There are some loops which contain more than one variable in the statements section. Their upper and lower bonds can be common or different sometimes. In C, in order to make two statements considered single, a comma operator is the one used, which basically combines two statements in a loop. It is used in the for loops in order to allow either multiple initializations or provide multiple decrement or increment. This pattern was seen in the Rose repository. Given below are two example codes:

```
int a[100], b[100];
int main() {
int i,j;
int i_ub =100;
for (i = 0, j=0 ; i < i_ub ; i++, j++ ) {
b[j]=a[i];
37
}
return 0; }
```

Cetus and Rose were unable to parallelize them to applying OpenMP pragmas. As it can be seen that these programs can easily be parallelized by changing the codes a little bit or a functionality can be added to those compilers to recognize them as parallelizable loops. A fix for the above loops to make a general normalization pass for comma expression

2.2.8.2 For-loops having multiple reduction operations over a single reduction variable:

When there exists a reduction variable inside a for-loop and the same variable goes through multiple reduction operators; Multiplication and addition, under the OpenMP rules the reduction directives can only be applied to one type of reduction operator, in the above case, either multiplication or addition. This pattern was seen in the Rose repository.

```
int main(int argc, char** argv) {
 int pi = 1;
int i;
for (i = 0; i < num_steps; i++) {
pi*=3;
pi+=(i*1/((i)+1)); }
printf("pi done - %d \n", pi);
 return (EXIT_SUCCESS);
}
```

2.2.8.2.1 Cetus Output

The Cetus output for the example 3 fails to parallelize the for-loop with an incorrect OpenMP reduction directive; *reduction(*: pi+: pi)*. As there are two types of reduction operations on a same reduction variable, the Cetus output seems to apply reduction directive on both reduction operations. Given below output:

```
int main(int argc, char * * argv) {
int pi = 1;
int i;
int _ret_val_0;
#pragma cetus private(i)
#pragma loop name main#0
#pragma cetus reduction(*: pi) reduction(+: pi) #pragma cetus parallel
#pragma omp parallel for private(i) reduction(*: pi+: pi)
for (i=0; i<2000000000; i ++ ) {
pi*=3;
pi+=((i*1)/(i+1)); }
 printf("pi done - %d \n", pi);
 _ret_val_0=0;
return _ret_val_0;
 }
```

2.2.8.2.2 Fix

A syntactically correct fix for the Cetus's OpenMP directive should be *#pragma omp parallel for private(i) reduction (*: pi) reduction (+: pi)*. As tested, this fix yields improved execution efficiency and a correct output. Given below:

```
 int main(int argc, char * * argv) {
int pi = 1;
int i;
int _ret_val_0;
#pragma cetus private(i)
#pragma loop name main#0
#pragma cetus reduction(*: pi) reduction(+: pi) #pragma cetus parallel
//#pragma omp parallel for private(i) reduction(*:pi) reduction(+:pi)
for (i=0; i<2000000000; i ++ )
{
```

```
pi*=3;
pi+=((i*1)/(i+1)); }
printf("pi done - %d \n", pi);
_ret_val_0=0;
return _ret_val_0;
}
```

Chapter 3

FUTURE WORK PROPOSITION

After indicating some of the weakness and strength of Cetus, there are weaknesses in Cetus which can be improved and for that a Cetus wish-list is compiled which contains the weaknesses/bugs which can be fixed by amending the Cetus compiler.

3.1 Cetus wish-list

3.1.1 Re-organizing the OMP's time commands *(Bug)*:

The statements in the code below, which contains the *"omp_get_wtime()"* are used to measure the time taken by the for loop to run.

```
double ctime1 = omp_get_wtime();
for (i=0; i<199999999; i ++ ){
        sum=a[i]+sum;
   }
double ctime2 = omp_get_wtime();
```

After passed through Cetus, the output code re-organizes the lower time stamp and moves it by the upper time stamp. Given below is its output:

```
double ctime1 = omp_get_wtime();
double ctime2 = omp_get_wtime();
#pragma omp parallel for reduction(+:sum)
for (i=0; i<199999999; i ++ )
   {
        sum=a[i];
   }
```

Cetus should not change the arrangement as the code shows wrong output.

3.1.2 Flipping of declaration variable to the top *(Bug)*:

Cetus changes the order of statements inside a for-loop; if there is a variable initialized inside the loop and it's not the first statement order-wise, then the Cetus changes the order of that variable by bringing it to the top of statements inside that loop. Below is an original code:

```
for (i =0; i<100; i++){
        X = y;
Int     z = x;
}
```

After passed through Cetus, the output code has its order changed:

```
#pragma omp ...
for (i =0; i<100; i++){
Int     z = x;
        X = y;
}
```

The correct way for Cetus is to not flip those variables. As the output is completely changed.

Another bug of Cetus is that it takes out the declaration variable out of the loop header and should be fixed. Given below:

```
for (int i = 0; i < i_ub ; i++){
  b[i]=a[i];
}
```

3.1.2.1 Cetus Output

```
int i = 0;
#pragma loop name main#0
for (; i<i_ub; i ++ ){
  b[i]=a[i];
}
```

3.1.3 Failure to detect Min and Max reductions *(Missing feature)*

Cetus has a missing feature of detecting min and max reductions.

3.1.3.1 Cetus Output

```
int foo (){
  double max_val=-1e99;
  double min_val=1e99;
  int i;
  for( i=0;i<10; i++){
        if(a[i] > max_val){
        max_val = a[i];
        }
        if(a[i] < min_val)
        min_val = a[i];
  }}
```

3.1.3.2 Rose Output

```
int foo(){
  double max_val = - 1e99;
  double min_val = 1e99;
  int i;
#pragma omp parallel for private (i) reduction (max:max_val) reduction
(min:min_val)
  for (i = 0; i <= 9; i += 1) {
        if (a[i] > max_val) {
        max_val = a[i];
        }
        if (a[i] < min_val)
        min_val = a[i];
  }}
```

3.1.4 Loop parallelizable via minor change *(Missing feature)*

In the for-loop below, the value of variable i and j is same throughout the loop

```
int i,j;
  for (i = 0, j=0 ; i < limit ; i++, j++ ){
        b[j]=a[i];
  }
```

The Cetus output where it cannot parallelize the loop for the above code is given below:

```
#pragma cetus private(i, j)
#pragma loop name main#0
```

```
for (((i=0), (j=0)); i<limit; ((i ++ ), (j ++ ))){
        b[j]=a[i];
}
```

Future work for Cetus is to apply loop normalization to make it eligible for parallelization.

3.1.5 Cetus removes <omp.h> header *(Bug)*

Cetus, in its output, always omits the header line "#include<omp.h>", which is a bug and that can be an easy future fix.

3.1.6 Unable to parallelize switch-statement in for-loops *(Missing feature)*

As discussed in *2.2.1 IS Benchmark* in *2.2.1.1Example 1* Cetus is unable to parallelize the for-loops which has the switch-statements inside them. The break statements inside the switch statement which clearly does not break out of the loop iterations are eligible for parallelization and Cetus should parallelize those kinds of loops.

3.1.7 Loops with print statement inside the if-condition *(Missing feature)*

It can be seen in the loops where there is a print statement inside the if-condition, even though the if-statement is false in all the iterations of the loop, the Cetus nor Rose can parallelize them.

If these loops are manually parallelized, they successfully run and show speedup. If the Cetus's forward propagation technique is improved, then parallelizing those loops are doable. The below pattern is contrived and not taken from any benchmark.

```
int i,j;
j = 20000;
int limit =531000000;
for (i = 0; i < limit ; i++ ){
        b[i]= i + 1;
```

```
if(b[i] == 0){
j = j+ i;
printf("Hello its if statement true");
}}
  return 0;
```

3.1.8 Loops taken from FT Benchmark *(Missing feature)*

Inside the FT Benchmark, file "*fft3d.c*", *line 161* there are two nested loops which are not parallelized by the Cetus:

The values of n1 = 256, n2 = 256, n3 = 128 and blkp = 33.

```
#pragma cetus private(i, j)
#pragma loop name fftXYZ#0#0#0
for (j=bls; j<=ble; j ++ ) {
        #pragma cetus private(i)
        #pragma loop name fftXYZ#0#0#0#0
        for (i=0; i<n1; i ++ ){
                plane[(j-bls)+(blkp*i)]=x[k][j][i];
                 }}
```

REFERENCES

[1] C. Dave, H. Bae, S. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," *Computer,* vol. 42, no. 12, pp. 36-42, 2009, doi: 10.1109/MC.2009.385.

[3] D. Quinlan *et al.*, *ROSE Tutorial: A Tool for Building Source-to-Source Translators*. 2019.

[4] Centre for Manycore Programming. " NAS Parallel Benchmarks (NPB) " http://aces.snu.ac.kr/software/snu-npb/ (accessed.

[5] NASA. "NASA Advanced Supercomputing Division." https://www.nas.nasa.gov/publications/npb.html#url (accessed.

[6] OPTCA. "OCA (Optimizing Code Advisor)." http://www.optca.org/omp-transformations.html#_Toc461635026 (accessed.

[7] Rose repository in GitHub. https://github.com/rose-compiler/rose/tree/master/projects/autoParallelization/tests.

[8] Cetus output for NPB and Rose repository in GitHub. https://github.com/jansher4/Cetus-Analysis