

# Lab Course

## Efficient Programming of Multicore Processors and Supercomputers

### Report 2: Sequential optimization

Jonas Mayer, Paul Preißner, Konrad Pröll

Fakultät für Informatik  
Technische Universität München

May 18<sup>th</sup> 2017

# Sequential Optimization: Matrix Access

## Access patterns into the Matrix

- ▶ Before Optimization: column wise access in relax\_jacobi
- ▶ low spatial locality

```
1 for (j = 1; j < sizex - 1; j++) {  
2     for (i = 1; i < sizey - 1; i++) {  
3         utmp[i * sizex + j] = 0.25 * (  
4             u[i * sizex + (j - 1)]  
5                 + // left  
6             u[i * sizex + (j + 1)]  
7                 + // right  
8             u[(i - 1) * sizex + j]  
9                 + // top  
10            u[(i + 1) * sizex +  
11                j]); // bottom  
12     }  
13 }
```

# Sequential Optimization: Matrix Access

- Solution: change loop, row wise access

```
1 for (j = 1; j < sizex - 1; j++) {  
2     for (i = 1; i < sizey - 1; i++) {  
3         utmp[i + j * sizey] = 0.25 * (  
4             a[i + (j - 1)*sizey] + // left  
5             a[i + (j + 1)*sizey] + //  
6                 right  
7             a[(i - 1) + j*sizey] + // top  
8             a[(i + 1) + j*sizey]); //  
9                 bottom  
10        ...  
11    }  
}
```

# Sequential Optimization: No Copy

## Avoid copy operations

- ▶ Before Optimization: in `relax_jacobi`, copy back all new values back to the array

```
1 for (j = 1; j < sizex - 1; j++) {  
2     for (i = 1; i < sizey - 1; i++) {  
3         u[i * sizex + j] = utmp[i * sizex + j];  
4     }  
5 }
```

# Sequential Optimization: No Copy

- Idea: Instead of recomputing, just swap the pointers

```
1 void arraySwap(double** a, double** b, int sizex, int  
   sizey) {  
2     double *temp = *a;  
3     *a = *b;  
4     *b = temp;  
5 }
```

# Sequential Optimization: No Copy

- Important: Need to initialize uhelp with u

```
1      for (i = 0; i < np; i++) {  
2          for (j = 0; j < np; j++) {  
3              (param.uhelp)[i * np + j] =  
4                  (param.u)[i * np + j];  
5          }  
        }
```

- Note: this took us 2 days to figure out

# Sequential Optimization: Residual

## Calculation of Residuum

- Before Optimization: For residual at timestep  $t$ , recompute entire relaxation of timestep  $t+1$

```
1 for (j = 1; j < sizex - 1; j++) {  
2     for (i = 1; i < sizey - 1; i++) {  
3         unew[i + j * sizey] = 0.25 * (  
4             a[i + (j - 1)*sizey] + // left  
5             a[i + (j + 1)*sizey] + //  
6                 right  
7             a[(i - 1) + j*sizey] + // top  
8             a[(i + 1) + j*sizey]); //  
9                 bottom  
10        diff = unew - u[i * sizex + j];  
11        sum += diff * diff;  
12    }
```

# Sequential Optimization: Residual

- ▶ Idea: Use results of previous timestep to compute residual
- ▶ Issue: residual thus uses results of previous timestep, does not recompute  $t+1$
- ▶ Solution: manually compute an initial iteration, thus "shifting" the residual output to  $t+1$



# Sequential Optimization: Residual

- Followup: Compute relaxation directly with residual and save for later

```
1 for (i = 1; i < sizey - 1; i++) {  
2     for (j = 1; j < sizex - 1; j++) {  
3         atmp[i * sizex + j] = 0.25 * (  
4             a[i * sizex + (j - 1)] + //  
5                 left  
6             a[i * sizex + (j + 1)] + //  
7                 right  
8             a[(i - 1) * sizex + j] + //  
9                 top  
10            a[(i + 1) * sizex + j]); //  
11            bottom  
12  
13            diff = atmp[i * sizex + j] -  
14                a[i * sizex + j];  
15            sum += diff * diff;  
16        }  
17    }  
18    return sum;
```

# Sequential Optimization: Interleaving

## **Cache optimization by interleaving of iterations: tiling and wavefront**

- ▶ Due to the spatial dependencies between the grid points, tiling and wavefront become non-trivial
- ▶ Traditional tiling would yield wrong/old values at the borders
- ▶ Idea: Tile grid and compute one iteration on tiles
- ▶ However: Due to the already sequential access pattern, wavefront interleaving is not expected to give a significant improvement of cache behaviour in a non-parallel environment

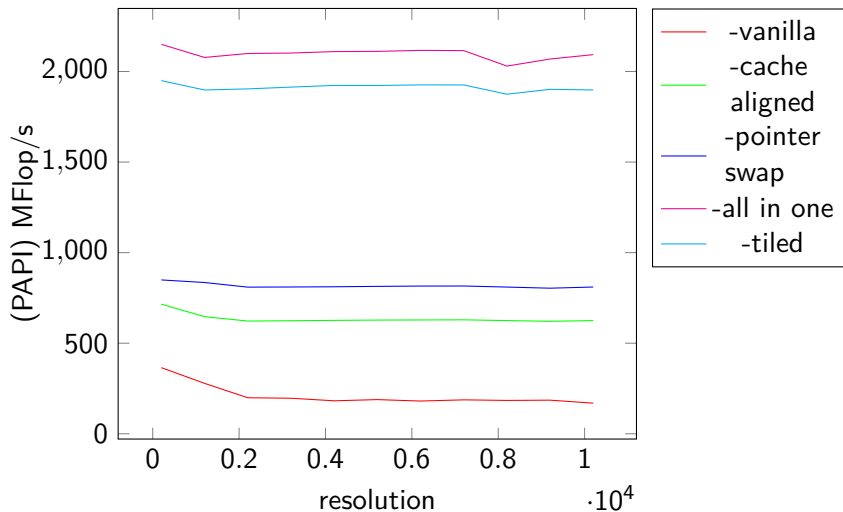
# Sequential Optimization: Interleaving

## ► Tiling Solution:

```
1 for (tiley = 1; tiley < sizey-1; tiley += tileSize){  
2     for (tilex = 1; tilex < sizex-1; tilex +=  
3         tileSize){  
4         for (i = tiley; (i < sizey - 1) && (i  
5             < tiley+tileSize); i++) {  
6             for (j = tilex; (j < sizex -  
7                 1) && (j < tilex+tileSize);  
8                 j++) {  
9                     atmp[i * sizex + j] =  
                        ...  
                        }  
                    }  
                }  
        }  
    }
```

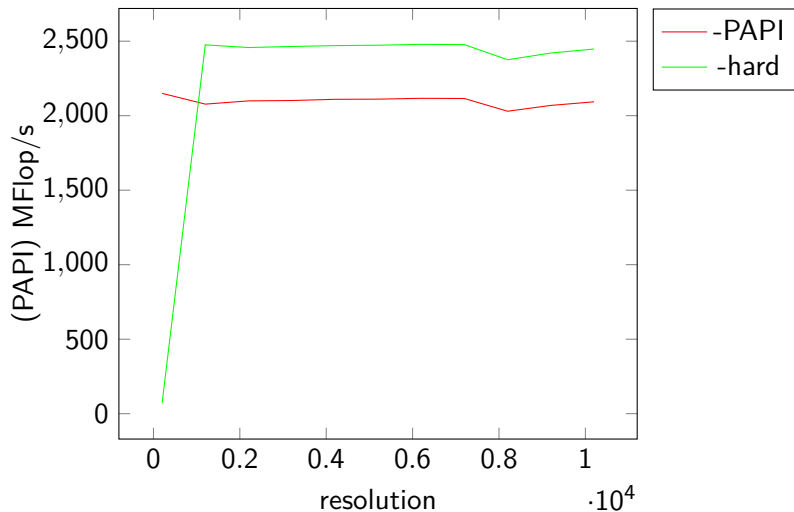
## Task 2.1 Sequential optimization - MFlop/s

Performance of Jacobi with various optimization steps (all -O2)



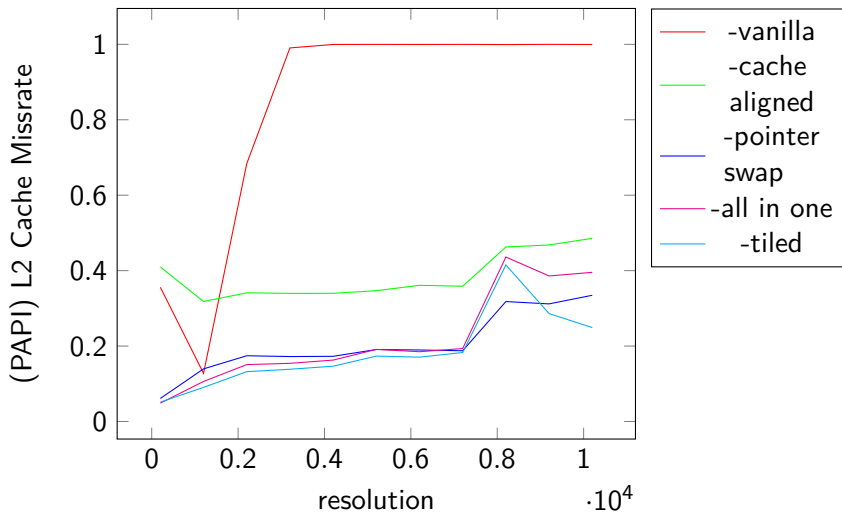
## Task 2.1 Sequential optimization - MFlop/s

PAPI vs hardcoded (iter  $\times$  11 flops  $\times$  grid<sup>2</sup>)



## Task 2.1 Sequential optimization - L2 Missrate

Performance of Jacobi with various optimization steps (all -O2)



## Task 2.2: Power of Two

**There is a performance issue around sizes of powers of 2, e.g.  $np=1020/1022/1024/1026$ .**

- ▶ Can you explain it?
- ▶ How to get rid of that issue?

When using array sizes of powers of two, the entries above and below will have similar least significant bits, this will cause them to be put into the same cache set. Hence, conflict misses will become more probable.

A way to get rid of this problem is array padding, so padding the arrays with a couple more entries, so that the least significant bits of neighboring cachelines differ, therefore reducing conflict misses.

Thank you for your attention!