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 18th 2017

Sequential Optimization: Matrix Access

Access patterns into the Matrix

- Before Optimization: column wise access in relax_jacobi
- low spatial locality

```
for (j = 1; j < sizex - 1; j++) {
        for (i = 1; i < sizey - 1; i++) {
                utmp[i * sizex + j] = 0.25 * (
                                u[i * sizex + (j - 1)] + // left
                                u[i * sizex + (j + 1)] + // right
                                u[(i - 1) * sizex + j] + // top
                                u[(i + 1) * sizex + j]); // bottom

                                ...
        }
}
```

Sequential Optimization: Matrix Access

- Solution: change loop, row wise access

```
for (j = 1; j < sizex - 1; j++) {
        for (i = 1; i < sizey - 1; i++) {
                utmp[i + j * sizey] = 0.25 * (
                        a[i + (j - 1)*sizey] +  // left
                        a[i + (j + 1)*sizey] +  // right
                        a[(i - 1) + j*sizey] +  // top
                        a[(i + 1) + j*sizey]); // bottom

                        ...
        }
}
```

Sequential Optimization: No Copy

Avoid copy operations

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

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

Sequential Optimization: No Copy

- Idea: Instead of recomputing, just swap the pointers

```c
void arraySwap(double** a, double** b, int sizex, int sizey) {
        double *temp = *a;
        *a = *b;
        *b = temp;
}
```

Sequential Optimization: No Copy

- Important: Need to initialize uhelp with u

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

- Note: this took us 2 days to figure out

Sequential Optimization: Residual

Calculation of Residuum

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

```
for (j = 1; j < sizex - 1; j++) {
        for (i = 1; i < sizey - 1; i++) {
                unew[i + j * sizey] = 0.25 * (
                        a[i + (j - 1)*sizey] +  // left
                        a[i + (j + 1)*sizey] +  // right
                        a[(i - 1) + j*sizey] +  // top
                        a[(i + 1) + j*sizey]); // bottom

                        diff = unew - u[i * sizex + j];
                        sum += diff * diff;
        }
}
```

Sequential Optimization: Residual

- Idea: Save results of previous timestep to compute residual

```
for (j = 1; j < sizex - 1; j++) {
        for (i = 1; i < sizey - 1; i++) {
                diff = utmp[i+j*sizey] - u[i+j*sizey];
                sum += diff * diff;
        }
}
```

Sequential Optimization: Residual

- Second Idea: Compute relaxation directly with residual and save for later

```
for (i = 1; i < sizey - 1; i++) {
        for (j = 1; j < sizex - 1; j++) {
                atmp[i * sizex + j] = 0.25 * (
                        a[i * sizex + (j - 1)] +  //
                            left
                        a[i * sizex + (j + 1)] +  //
                            right
                        a[(i - 1) * sizex + j] +  //
                            top
                        a[(i + 1) * sizex + j]); //
                            bottom

                        diff = atmp[i * sizex + j] -
                            a[i * sizex + j];
                        sum += diff * diff;
        }
}
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 improval of cache behaviour in a non-parallel environment

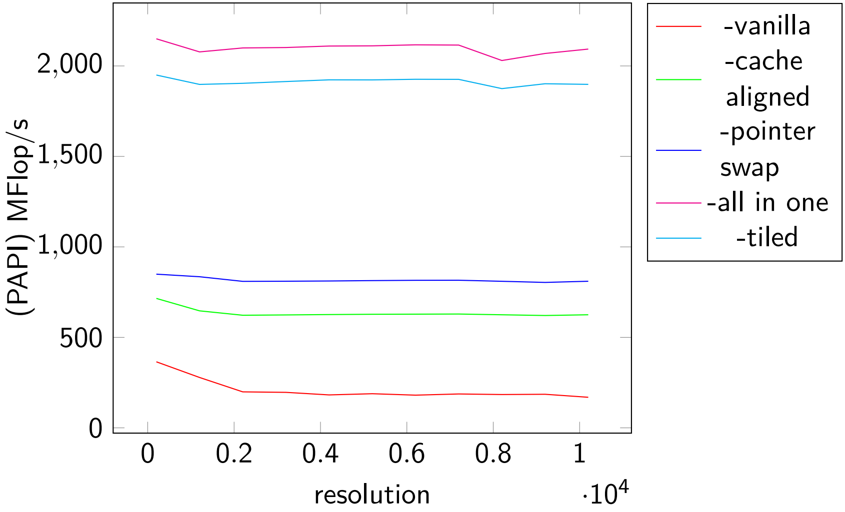
Sequential Optimization: Interleaving

- Tiling Solution:

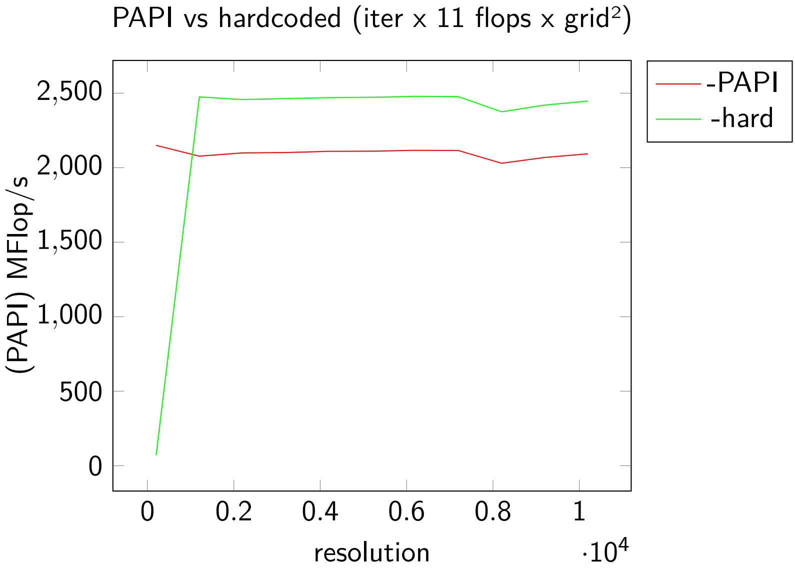
```
for (tiley = 1; tiley < sizey-1; tiley += tilesize){
        for (tilex = 1; tilex < sizex-1; tilex +=
            tilesize){
                for (i = tiley; (i < sizey - 1) && (i
                    < tiley+tilesize); i++) {
                        for (j = tilex; (j < sizex -
                            1) && (j < tilex+tilesize);
                            j++) {
                                atmp[i * sizex + j] =
                                    ...
                        }
                }
        }
}
```

Task 2.1 Sequential optimization - MFlop/s

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

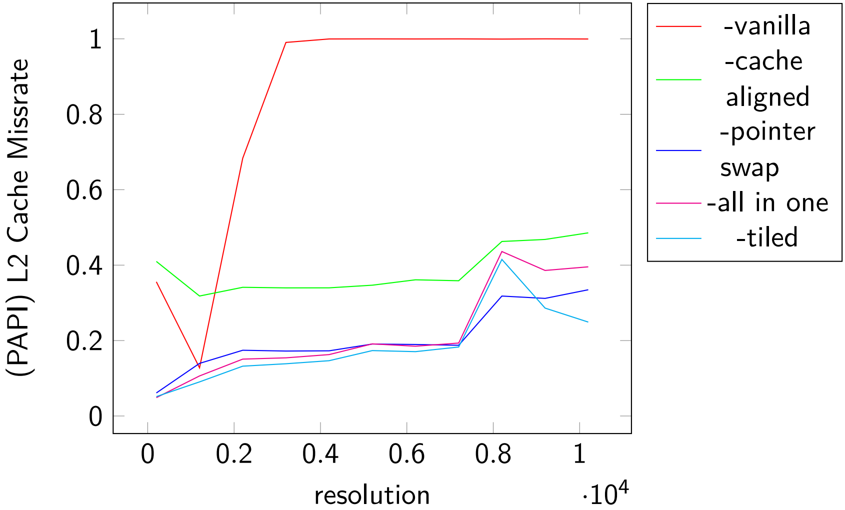


Task 2.1 Sequential optimization - MFlop/s



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!