Paralleles Höchstleistungsrechnen Übung 1

Markus Döring, 3153320

October 31, 2013

1. Effiziente Cache-Nutzung

a). Umordnen von Schleifen

Die äußere Schleife lässt sich nicht mit einer der inneren Schleifen vertauschen, da die Ergebnisse jeder Iteration von denen der vorherigen Abhängen. Damit bleibt nur das Vertauschen der inneren Schleifen möglich. In C++ sind die Zeilen eines Feldes zusammenhängend im Speicher, daher operiert die Schleife so wie sie ist (von links nach rechts, dann von oben nach unten) in Richtung der Cachelines. Die Cacheline links oben wird zum Beispiel nur in den ersten 3 Schritten benötigt. Ordnet man die Schleifen jetzt um, dann arbeitet der Algorithmus entgegen der Richtung der Cachelines, und nutzt sie somit weniger effizient. Ein Test mit einem normalen PC ergab eine Verlangsamung um den Faktor 2.

b). Kachelung

Das folgende Programm kachelt die Daten und versucht dabei, die Cacheline möglichst effizient zu nutzen. Dafür werden zwei 'Kacheln' jeweils ganz in den Cache gelesen. Jede Kachel benötigt höchstens die Kachel links und eine Cacheline der Kachel darüber, darum scheint es sinnvoll zwei Kacheln gleichzeitig im L1 Cache zu haben. Als Ergebnis mit auf den PC abgestimmten Werten ergab sich eine Beschleunigung um Faktor 3.

```
#define N 512
#define N_ITERATIONS 2000
#define CACHE_LINE 64
#define N_LINES (24576/64)
#define MIN(a,b) ((a)<(b)?(a):(b))

int main(int c, char** v)
{ // 0.74s
    int i,j,maxi,maxj,mini,minj;
    double field[N][N];
    //TODO implement border checks!
    for (int k = 0; k < N_ITERATIONS; k++)
    {
        for (int offset_i = 0; offset_i < N; offset_i += CACHE_LINE)
        {
            // start with element 1 if we are on left border
            mini = offset_i ? 0 : 1;

            // don't go over right border
            maxi = MIN(CACHE_LINE, N-offset_i) - mini;

            for (int offset_j = 0; offset_j < N; offset_j += N_LINES/2)
            {
                minj = offset_j ? 0 : 1;
                maxj = MIN(CACHE_LINE, N-offset_j) - minj;
                for (int curr_i = mini; curr_i < maxi; curr_i++)
                {
                    for (int curr_j = minj; curr_j < maxj; curr_j++)
                    {
                        i = offset_i + curr_i;
                        j = offset_j + curr_j;
                        field[i][j] = .25 * (field[i-1][j] +
                                             field[i+1][j] +
                                             field[i][j-1] +
                                             field[i][j+1]);
                    }
                }
            }
        }
    }
}
```

2. ILP

a).

Zwischen Zeile 21 und den Zeilen 22/23 besteht eine *Data True Dependence*. Der Hashwert muss erst berechnet werden, bevor er als Index benutzt werden kann.

Haben zwei Elemente den gleichen Hashwert, so besteht zwischen den Zeilen 21 der entsprechenden Iterationsschritte eine *Data Output Dependence*. Beide Schritte schreiben an eine Speicherstelle, die durch den berechneten Hashwert definiert wird (Zeile 33/34).

Die Variable *ptrUpdate* wird in Zeile 22 berechnet und in Zeile 34 dereferenziert, also gelesen. Damit liegt eine *Data Antidependence* vor.