Exercise for Course

Parallel High-Performance Computing

Dr. S. Lang

Return: 29. Oktober 2015 at the start of the exercise or earlier

---

Task 1 Efficient Cache Usage (5 Punkte)

In the following program segment shall u be an array of dimension $n \times n$:

```
1 for (int k=0; k<1000; ++k)
2   for (int i=1; i<n-1; ++i)
3     for (int j=1; j<n-1; ++j)
4       u[i][j] = 0.25 * (u[i-1][j] + u[i+1][j] + u[i][j-1] + u[i][j+1]);
```

The array u is initialized meaningful, in the inner of the array are therefore the averaged values of the neighbors introduced. Watch out old (still not considered) and new (already considered) entries are used for the average. The computer of interest has cache lines with size $l = 32$ byte, and floating point numbers use 8 byte each. Therefore in a cache line fit 4 numbers. For simplicity the length n of a matrix line is a multiple of the size m of a cache line, and furthermore n has an amount that the matrix does not fit into the cache anymore.

1. Can we gain a better cache usage by reordering the loops?
2. Rewrite the algorithm such that blocking of data is introduced and discuss the influence onto the cache usage.

Task 2 Instruction Level Parallelism (10 Punkte)

Instruction Level Parallelism (ILP) is a meaure of how many operations of program can be executed in parallel. In this task we examine how different data and control flow dependencies have influence onto ILP. We consider the following dependencies:

- *Data true dependence, DTD*:
  Instructions depend on results of prior executed instructions,
- *Data antidependence, DA*:
  Instructions write in addresses, that are read by other instructions,
- *Data output dependence, DOD*:
  Instructions write in addresses, that are written by another instruction,
- *Control dependence, CD*:
  The execution of an instruction depends on control conditions (`if, ...`).

We investigate the dependencies for an incomplete implementation of a hash table. With hash tables questions as „*Does an element exist in the given set?*“ can be answered. Herefore the hash table store e.g. a linked, ordered list of elements. The elements can be accessed by keys, calculated by the *hash function*. To test whether an element exist in the set, its key, the *hash value*, is calculated, and only a search in the related list (the storing data structure is called *bucket*) is started. Our hash table can store 1024 buckets each with a linked list of elements. The following listing shows how the hash table is initialized:

```
1 /*************************************************************/
2 /* an incomplete implementation of a hash table              */
3 /*************************************************************/
4
5 /* struct for elements to be inserted */
6 typedef struct Element {
```

```
7    int value;
8    struct Element *next;
9  }
10
11 Element myElements[N_ELEMENTS]; /* array of items */
12 Element *bucket[1024];          /* 1024 buckets, pointers in each */
13                                 /* bucket are initialized to NULL */
14
15 for (i=0; i<N_ELEMENTS; i++)
16 {
17   Element *ptrCurr, **ptrUpdate;
18   int hashIndex;
19
20   /* find location where the new element is to be inserted */
21   hashIndex = myElements[i].value & 1023;
22   ptrUpdate = &bucket[hashIndex];
23   ptrCurr   = bucket[hashIndex];
24
25   /* find place in chain to insert the new element */
26   while ( ptrCurr && ptrCurr->value =< myElements[i].value)
27   {
28     ptrUpdate = &ptrCurr->next;
29     ptrCurr   = ptrCurr->next;
30   }
31
32   /* update pointers to insert the new element into chain */
33   myElements[i].next = *ptrUpdate;
34   *ptrUpdate = &myElements[i];
35 }
```

The elements store each an integer, the `N_ELEMENTS` elements, that need to be inserted, are stored in the array `myElements`. The has table `bucket` can store 1024 pointers onto linked element lists, all pointers point at program start into void.

Now we iterate over the existing elements and in line 21 the hash value of the element is calculated by a simple hash function: The hash value consists of the last ten bits of a value of an element, because the operator `&` connects the value of the element with the bit pattern 11 1111 1111 of the decimal number 1023 with binary `AND`. Then it is iterated over the linked list. With the pointer `ptrCurr` the list is traversed, the variable `ptrUpdate` stores the pointer, that need to be adapted, to insert an element at an individual place in the list. Therefore in line 23 `ptrCurr` is set onto the first element of the according bucket, and in the following while loop (l. 25-30) iterates until the fitting insertion entry is found. Now the new element is inserted in the lines 33, 34, by setting the `next` pointer of the element to insert onto the address of the following element, that is stored in `ptrUpdate`. In line 34 the pointer is set onto the successor element. `ptrCurr` is a double pointer, thus the operator `*` dereferences once and points then onto a pointer.

We interpret now each line as an assembly instruction and investigate the dependencies. From these we gain a dynamic dependency graph similar to that in figure 0.1 (not all dependencies are shown, and the instructions from l. 28, 29 are missing). Each node of the graph represents a machine instruction, that is executed within the cycle. Each horizontal contains also instructions (line), that can be executed in parallel. Arrows between the nodes describe dependencies. There exists e.g. between the lines 15 and 21 a DTD, since `i` is calculated in line 15 and used in line 21.

Subtask (a) (5 points)

(a1) What kind of dependency consists between the lines 21 and 22/23?

For different elements could be calculated the same hash value, that leads to dependencies between the loop iterations.

(a2) What kind of dependency consists now between the lines 21 and the same line 21 for two different elements with the same hash value?

(a3) What kind of dependency consists between the lines 34 and 22?

Subtask (b) (5 points)

We now consider a very simplified scenario, where the hash table is initially empty and the numbers $0 \ldots 1023$ have to be inserted, therefore for each bucket only one element exists. We consider the

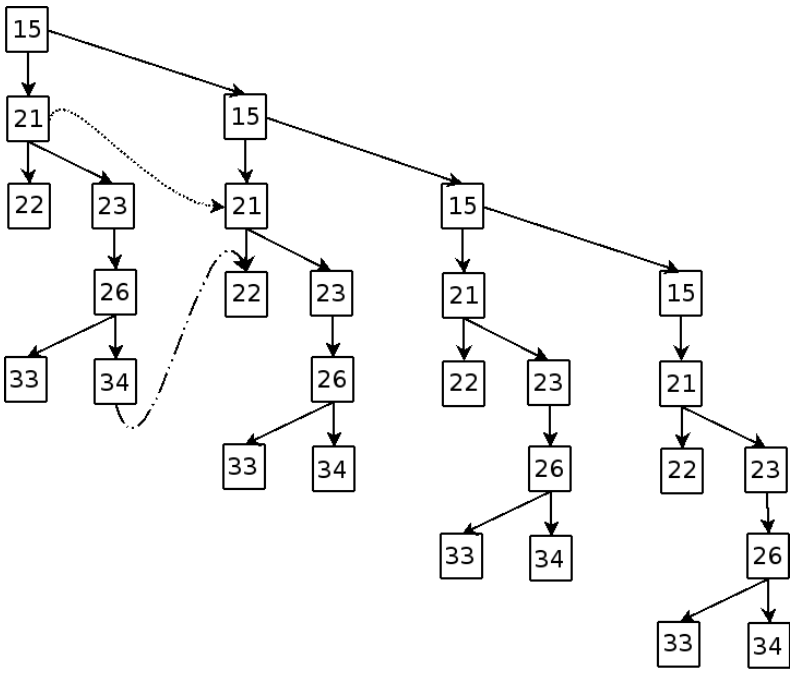

Abbildung 0.1: Dynamic dependencies of the hash table. Each node is related to a single program line, the arrows visualize different dependencies. Only the ideal case is shown (subtask b), since the `while` loop is not entered (instructions 28, 29 are therefore missing). The dashed arrows show possible dependencies in subtask (a).

dependencies when the four inital elements are inserted.

(b1) Which is the single dependency, that consists between the lines (consider the variable `i` in line 15).

(b2) Rewrite the `for` loop such that the dependency between the loop iterations is reduced (remark: implement in a loop instructions for `i` and `i+1`).

(b3) According to the graph an iteration of the outer loop consists of 7 instructions, alltogether in 1024 iterations 7168 instructions in total. These instructions are executed in 4 cycles, the loop needs thus $4 + 1024 = 1028$ cycles in total, until it is completely finished. This gives an ILP (Available Instruction Level Parallelism) of $7128/1028 = 6.97$. Which influence has your optimization from (b) onto the ILP?