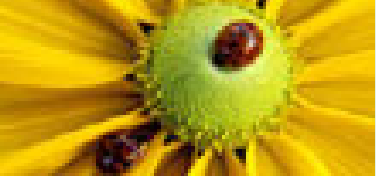
A User's Experience with Parallel Sorting and OpenMP

Talk at the EWOMP'04 conference, Stockholm

Michael Süß, Claudia Leopold

University of Kassel, Germany

http://www.se.e-technik.uni-kassel.de/se/index.php?id=msuess



Goals

Introduction
- Goals
- Sorting

Recursion

Busy Waiting

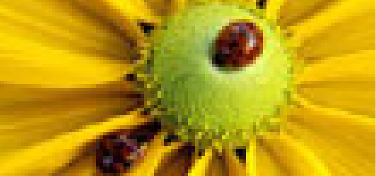
Summary

- What this paper wants to show:
  - a simple parallel sorting algorithm with OpenMP
  - some problems in the OpenMP specification:
    - Recursion
    - Busy Waiting
  - different ways to solve these problems:
    - recursion: stacks, nesting, workqueue
    - busy Waiting: condition variables, sched_yield()
  - constructs, we would have found useful in OpenMP
- What this paper does not show:
  - the fastest parallel sorting algorithm of all times
  - full proposals for inclusion into the OpenMP-specification

Sorting

```
template <typename T> void myQuickSort(std::vector<T> &myVec, int q, int r)
{
  T pivot;
  int i, j;

  /* Skipped: only small partition left -> use insertion sort! */

  /* choose pivot, initialize borders */
  pivot = myVec[r]; i = q - 1;  j = r;

  /* partition step, divides partition around pivot */
  while (true) {
    while (myVec[++i] < pivot);
    while (myVec[--j] > pivot);
    if (i >= j) break;
    std::swap(myVec[i], myVec[j]);
  }
  std::swap(myVec[i], myVec[r]);

  /* recursively call yourself with new subpartitions */
  myQuickSort(myVec, q, i - 1);
  myQuickSort(myVec, i + 1, r);
}
```

1. choose pivot element
2. iterate through elements, moving numbers smaller than pivot to the left, numbers larger than pivot to the right
3. now the partition is divided into two subpartitions: the left one contains all numbers smaller than the pivot and the right one contains all numbers larger than the pivot.
4. go to step 1 for left and right subpartitions (if there is more than one element left in that partition)



Stacks

Introduction
Recursion
- Stacks
- sort_omp_1.0
- Nesting
- Workqueue
- Performance

Busy Waiting
Summary

- every recursion can be changed to an iteration with stacks
- for OpenMP, already suggested by an Mey (EWOMP'03)
- tried this in sort_omp_1.0.cpp, making the following changes:
  - added a data structure globalTodoStack
  - a parallel region was added in main, around the first call of the myQuickSort–function
  - a call to omp_get_thread_num was added, so that only one thread initially executes the myQuickSort–function, all others wait until work for them is put on the globalTodoStack
  - all accesses to shared variables (especially the globalTodoStack) were protected by critical sections to prevent multiple threads from accessing the variables at the same time

sort_omp_1.0

```
void myQuickSort(std::vector<T> &myVec, int q, int r,...)
{ /* Skipped: Initialisation */

  while (true) {

    while (q >= r) {  /* thread needs new work */

      #pragma omp critical
      {
        /* something left on the global stack to do? */
        if (false == globalTodoStack.empty()) {
          /* Skipped: Pop a new segment off the stack */
      }
      /* if all threads are done
       * -> return from this function here */
    }
    /* Skipped: choose pivot and do partitioning step */

    #pragma omp critical
    {
      globalTodoStack.push(pair(q, i - 1));
    }
```

```
      /* iteratively sort elements right of pivot,
       * r stays the same */
      q = i + 1;
    }
}

int main(int argc, char *argv[])
{
  /* Skipped: Program Initialisation */
  #pragma omp parallel shared(myVec, \
           globalTodoStack, numThreads, numBusyThreads)
  {
    /* start sorting with one thread,
     * the others wait for the stack to fill up */
    if (0 == omp_get_thread_num()) {
      myQuickSort(myVec, 0, myVec.size() - 1,...);
    } else {
      myQuickSort(myVec, 0, 0,...);
    }
  }
  /* Skipped: Tests and Program output */
}
```

Nesting

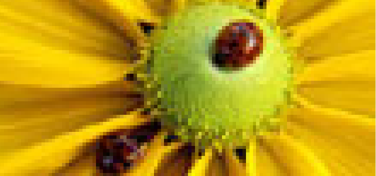
```
void myQuickSort(std::vector < T > &myVec, int q, int r, ...)
{ /* Skipped: Initialisation + Partitioning step */

  /* do not nest, if there are too many threads already */
  if (numBusyThreads >= 2 * numThreads - 1) {
    myQuickSort(myVec, q, i - 1, numBusyThreads, numThreads);
    myQuickSort(myVec, i + 1, r, numBusyThreads, numThreads);
  } else {
    #pragma omp atomic
    numBusyThreads += 2;

    #pragma omp parallel shared(myVec, q, r, ...) {
      #pragma omp sections nowait {
        #pragma omp section {
          myQuickSort(myVec, q, i - 1, numBusyThreads, numThreads);
          #pragma omp atomic
          numBusyThreads--;
        }
        #pragma omp section {
          myQuickSort(myVec, i + 1, r, numBusyThreads, numThreads);
          #pragma omp atomic
          numBusyThreads--;
        }
```

- program: sort_omp_nested_1.0.cpp
- problem 1: compilers may serialize nested parallel regions — possible solution: change the specification
- problem 2: how to limit creation of new threads, when we dive deeper into the recursion?
  - the omp_get_num_threads() – function does not help here, as it always returns two
  - we would have needed: omp_get_num_all_threads(), which would return the number of all presently running threads

Workqueue

```
void myQuickSort(std::vector<T> &myVec, int q, int r)
{ /* Skipped: Initialisation + Partitioning step */

  #pragma omp taskq {
    #pragma omp task {
      myQuickSort(myVec, q, i - 1);
    }
    #pragma omp task {
      myQuickSort(myVec, i + 1, r);
    }
  }
}

int main(int argc, char *argv[])
{ /* Skipped: Program Initialisation */

  #pragma omp parallel shared (myVec) {
    #pragma omp taskq {
      #pragma omp task {
        myQuickSort(myVec, 0, myVec.size() - 1);
      }
    }
  } /* Skipped: Tests and Program output */
}
```

- extension to the OpenMP–specification suggested by Shah et al. (EWOMP'02)
- program: sort_omp_taskq_1.0.cpp
- problem 1: not part of the specification — possible solution: change the specification
- problem 2: no way to limit the creation of new tasks during recursion — worked on by authors and was no problem during our tests
- besides that: easy and elegant solution

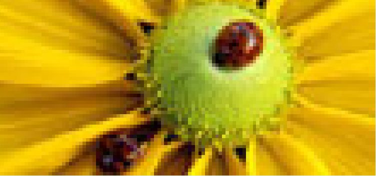


Performance

Introduction
Recursion
- Stacks
- sort_omp_1.0
- Nesting
- Workqueue
- Performance
Busy Waiting
Summary

Program	Wall–clock time (sec.)						
	AMD Opteron 2200			SUN Fire 6800			
	1Th.	2Th.	4Th	1Th.	2Th.	4Th.	8Th.
sort_seq_1.5	23.8	23.8	23.8	36.8	36.8	36.8	36.8
sort_omp_1.0	24.0	13.7	8.1	38.2	23.9	15.7	11.0
sort_omp_2.0	24.3	12.6	7.5	37.9	21.4	13.1	10.0
sort_omp_nested_1.0	23.9	21.4	12.4	43.4	25.2	25.2	25.2
sort_omp_taskq_1.0	29.8	16.3	9.1	n.A.	n.A.	n.A.	n.A.

Table 1: Wall–clock time for sorting 100 million integers in seconds



The Problem of Busy Waiting

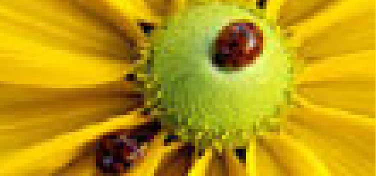
Introduction

Recursion

Busy Waiting
- Problem
- Performance

Summary

- what to do, when the stacks of the iterative sorting algorithm are empty?
- thread to wait for new work or for all other threads to signal that they are done
- while waiting, processor cycles are wasted -> busy waiting
- in OpenMP, we know no way to get rid of this behaviour
- solution 1: condition variables
  - synchronization primitive, not easy to understand
  - introduction into the OpenMP–specification has been suggested by Lu et al. (Supercomputing'98)
  - solves problem fully
- solution 2: sched_yield() — puts calling thread at the end of operating system scheduler + selects new thread to run



Performance

Introduction

Recursion

Busy Waiting

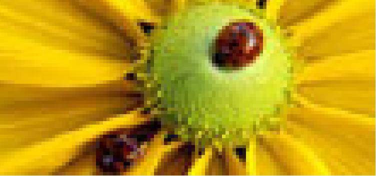
- Problem
- Performance

Summary

- system is put under heavy load by launching four times more threads than processors

Program	SUN / 96 Threads	AMD / 16 Threads
sort_omp_1.0	>600	20.5
sort_omp_2.0	>600	19.0
sort_pthreads_1.0	15.9	7.8
sort_pthreads_cv_1.0	10.6	7.7
sort_pthreads_yield_1.0	11.4	7.9

Table 2: Average wall–clock time for sorting 100 million integers on heavily loaded systems in seconds



Summary

Introduction

Recursion

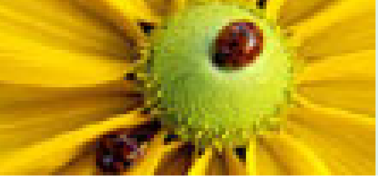
Busy Waiting

Summary

- presented several versions of a parallel quicksort implementation

- weaknesses of the OpenMP–specification:
  - difficult to deal with recursion — possible solutions: stacks, nested parallelism, workqueues, all of which have their flaws
  - difficult to deal with busy waiting — possible solutions: condition variables, sched_yield()

- future:
  - implement and test some of the ideas presented in an actual compiler
  - test more algorithms for potential problems



Thank you for your attention!