Advanced programming with OpenMP

Libor Bukata a Jan Dvořák
Programme of the lab

• OpenMP Tasks – parallel merge sort, parallel evaluation of expressions
• OpenMP SIMD – parallel integration to calculate $\pi$
• User-defined reduction – parallel summation of the matrix with collapsed for loops
• The sequential code of examples is available on Course Ware.
void mergeSortRecursive(vector<double>& v, unsigned long left, unsigned long right) {
    if (left < right) {
        unsigned long mid = (left+right)/2;
        mergeSortRecursive(v, left, mid);
        mergeSortRecursive(v, mid+1, right);
        inplace_merge(v.begin()+left, v.begin()+mid+1, v.begin()+right+1);
    }
}

void mergeSort(vector<double>& v) {
    mergeSortRecursive(v, 0, v.size()-1);
}

...
Sequential merge sort

```cpp
void mergeSortRecursive(vector<double>& v, unsigned long left, unsigned long right) {
    if (left < right) {
        if (right-left >= 32) {
            unsigned long mid = (left+right)/2;
            mergeSortRecursive(v, left, mid);
            mergeSortRecursive(v, mid+1, right);
            inplace_merge(v.begin()+left, v.begin()+mid+1, v.begin()+right+1);
        } else {
            sort(v.begin()+left, v.begin()+right+1);
        }
    }
}

void mergeSort(vector<double>& v) {
    mergeSortRecursive(v, 0, v.size()-1);
}
```

Use fast $O(n^2)$ algorithm to decrease the deepness of the recursion. Typically, Insert Sort performs very well for small arrays.

The sort from the C++ standard library is used to keep the code simple.

<table>
<thead>
<tr>
<th>Sorting algorithm</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential STL sort</td>
<td>20.981 s</td>
</tr>
<tr>
<td>Sequential Merge Sort</td>
<td>29.210 s (previously 39.9 s)</td>
</tr>
</tbody>
</table>

Number of elements to sort: 160000000
2 x Intel Xeon E5-2620 v2 @ 2.10GHz (in total 12 cores + HT)
Parallel sort in 5 minutes

... void mergeSortRecursive(vector<double>& v, unsigned long left, unsigned long right)   { 
  if (left < right)   { 
    if (right-left >= 32)   { 
      unsigned long mid = (left+right)/2;
      #pragma omp taskgroup
      { 
        #pragma omp task shared(v)
        mergeSortRecursive(v, left, mid);
        mergeSortRecursive(v, mid+1, right);
      }
      inplace_merge(v.begin()+left, v.begin()+mid+1, v.begin()+right+1);
    } else { 
      sort(v.begin()+left, v.begin()+right+1);
    }
  }
}

void mergeSort(vector<double>& v)   { 
  #pragma omp parallel
  #pragma omp single
  mergeSortRecursive(v, 0, v.size()-1);
} 

... Create pool of threads and start with one of them.

<table>
<thead>
<tr>
<th>Sorting algorithm</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential STL sort</td>
<td>20.981 s</td>
</tr>
<tr>
<td>Sequential Merge Sort</td>
<td>29.210 s</td>
</tr>
<tr>
<td>Parallel Merge Sort v1</td>
<td>25.569 s (1.14 x)</td>
</tr>
</tbody>
</table>
void mergeSortRecursive(vector<double>& v, unsigned long left, unsigned long right) {
    if (left < right) {
        if (right-left >= 32) {
            unsigned long mid = (left+right)/2;
            #pragma omp taskgroup
            {
                #pragma omp task shared(v) untied if(right-left >= (1<<14))
                mergeSortRecursive(v, left, mid);
                mergeSortRecursive(v, mid+1, right);
            }
            inplace_merge(v.begin()+left, v.begin()+mid+1, v.begin()+right+1);
        } else {
            sort(v.begin()+left, v.begin()+right+1);
        }
    }
}

void mergeSort(vector<double>& v) {
    #pragma omp parallel
    #pragma omp single
    mergeSortRecursive(v, 0, v.size()-1);
}

<table>
<thead>
<tr>
<th>Sorting algorithm</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential STL sort</td>
<td>20.981 s</td>
</tr>
<tr>
<td>Sequential Merge Sort</td>
<td>29.210 s</td>
</tr>
<tr>
<td>Parallel Merge Sort v1</td>
<td>25.569 s (1.14 x)</td>
</tr>
<tr>
<td>Parallel Merge Sort v2</td>
<td>4.74 s (6.16 x)</td>
</tr>
</tbody>
</table>
void mergeSortRecursive(vector<double>& v, unsigned long left, unsigned long right) {
  if (left < right) {
    if (right-left >= 32) {
      unsigned long mid = (left+right)/2;
      #pragma omp taskgroup
      {
        #pragma omp task shared(v) untied if(right-left >= (1<<14))
        mergeSortRecursive(v, left, mid);
        #pragma omp task shared(v) untied if(right-left >= (1<<14))
        mergeSortRecursive(v, mid+1, right);
        #pragma omp taskyield
      }
      inplace_merge(v.begin()+left, v.begin()+mid+1, v.begin()+right+1);
    } else {
      sort(v.begin()+left, v.begin()+right+1);
    }
  }
}

void mergeSort(vector<double>& v) {
  #pragma omp parallel
  #pragma omp single
  mergeSortRecursive(v, 0, v.size()-1);
}

<table>
<thead>
<tr>
<th>Sorting algorithm</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential STL sort</td>
<td>20.981 s</td>
</tr>
<tr>
<td>Sequential Merge Sort</td>
<td>29.210 s</td>
</tr>
<tr>
<td>Parallel Merge Sort v1</td>
<td>25.569 s (1.14 x)</td>
</tr>
<tr>
<td>Parallel Merge Sort v2</td>
<td>4.74 s (6.16 x)</td>
</tr>
<tr>
<td>Parallel Merge Sort v3</td>
<td>4.83 s (6.05 x)</td>
</tr>
<tr>
<td>Parallel Merge Sort v3 (48 threads)</td>
<td>4.42 s (6.61 x)</td>
</tr>
</tbody>
</table>
cout<<“Evaluating expression: 2*(5*3+7*7)”<<endl;
int term1 = 0, term2 = 0, total = 0;

#pragma omp parallel
#pragma omp single
{
#pragma omp task depend(out: term1)
{
    this_thread::sleep_for(seconds(2));
    term1 = 5*3;
}

#pragma omp task depend(out: term2)
{
    this_thread::sleep_for(seconds(2));
    term2 = 7*7;
}

#pragma omp task depend(in: term1, term2) depend(out: total)
{
    this_thread::sleep_for(seconds(1));
    total = term1+term2;
}

#pragma omp task depend(in: total)
{
    this_thread::sleep_for(seconds(2));
    total *= 2;
}

#pragma omp taskwait
cout<<“Final value of the expression: “<<total<<endl;
}
Calculation of pi

\[ 4 \times \arctan(1) = \pi \quad \int \frac{1}{1 + x^2} = \arctan(x) \]

\[ 4 \int_{0}^{1} \frac{1}{1 + x^2} = 4 \left( \arctan(1) - \arctan(0) \right) = \pi \]

= 0

<table>
<thead>
<tr>
<th>Calc of π</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential version</td>
<td>5.41 s</td>
</tr>
</tbody>
</table>

10^9 of steps; 2 x Intel Xeon E5-2620 v2

double piIntStepTrapezoidalRule(int i, const double& step) {
    double x = (i+0.5)*step;
    return 1.0/(1.0+x*x);
}

... double sum = 0.0, step = 1.0/nsteps;
for (unsigned long i = 0; i < nsteps; ++i)
    sum += piIntStepTrapezoidalRule(i, step);

double pi = 4.0*step*sum;
...
Calculation of pi - vectorization

...#pragma omp declare simd
double pilnTrapezoidalRule(int i, const double& step) {
    double x = (i+0.5)*step;
    return 1.0/(1.0+x*x);
}
...

double sum = 0.0, step = 1.0/nsteps;
#pragma omp simd reduction(+: sum)
for (unsigned long i = 0; i < nsteps; ++i)
    sum += pilnTrapezoidalRule(i, step);

double pi = 4.0*step*sum;
...

The parallel processing of multiple data is hidden in hardware, the program behaves like a sequential version from the programmer point of view.

To verify that code was vectorized, you can compile with `-fopt-info-vec` argument (GCC).

$ g++ -fopenmp -fopt-info-vec -std=c++11 -march=native -O2 -o CalcOfPi CalcOfPi.cpp
CalcOfPi.cpp:36:42: note: loop vectorized
CalcOfPi.cpp:10:15: note: loop vectorized

Not all the code is vectorizable (loop dependencies)!
Calculation of pi - parallelization

... #pragma omp declare simd
double piIntStepTrapezoidalRule(int i, const double & step) {
    double x = (i+0.5)*step;
    return 1.0/(1.0+x*x);
}
...

... double sum = 0.0, step = 1.0/nsteps;
#pragma omp parallel for simd reduction(+: sum)
for (unsigned long i = 0; i < nsteps; ++i)
    sum += piIntStepTrapezoidalRule(i, step);

double pi = 4.0*step*sum;
...

<table>
<thead>
<tr>
<th>Calc of π</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential version</td>
<td>5.41 s</td>
</tr>
<tr>
<td>Vectorized version</td>
<td>2.71 s (2.0 x)</td>
</tr>
<tr>
<td>Parallel SIMD version</td>
<td>0.26 s (20.8 x)</td>
</tr>
</tbody>
</table>

10^9 of steps; 2 x Intel Xeon E5-2620 v2

- Two/three dimensional integration can be significantly accelerated. To improve the accuracy other integration rules (e.g. Simpson rule) are used in practice
- How many arithmetical operations are performed per integration step? What is the slowest operation?
- Although the speedup is amazing the algorithm achieves only 23 GFlops/s (10% of the peak), how is it possible?
OpenMP - user-defined reduction

... User defined reduction for vectors.
#pragma omp declare reduction(+ : MatrixColumn : transform(omp_in.cbegin(), omp_in.cend(),
omp_out.begin(), omp_out.begin(), plus<double>()) initializer(omp_priv(omp_orig))

// Sum all the entries in the matrix.
MatrixColumn sumOfRows(M, 0.0);
#pragma omp parallel for collapse(2) reduction(+: sumOfRows) if(M*N > 10e6)
for (int i = 0; i < M; ++i) {
    for (int j = 0; j < N; ++j)
        sumOfRows[i] += m[i][j];
}

double totalSum = 0.0;
for (int i = 0; i < M; ++i)
    totalSum += sumOfRows[i];
...

Illustration of vector reduction:

It works effectively for all the shapes of the input matrix!
OpenMP – other topics.

• **Affinity of threads**
  - Threads are fixed to cores, especially useful for NUMA systems.
  - Since threads are not migrated between cores, the number of cache invalidations is reduced.

• **Device offloading** (Intel Xeon Phi, NVIDIA)
  - Extension of pragmas to support offloading of work to coprocessors, graphics cards, etc.

• To get more information on the aforementioned topics you can check slides from Intel: [http://www.lrz.de/services/compute/courses/x_lecturenotes/MIC_GPU_Workshop/Intel-03-OpenMP-in-a-Nutshell.pdf](http://www.lrz.de/services/compute/courses/x_lecturenotes/MIC_GPU_Workshop/Intel-03-OpenMP-in-a-Nutshell.pdf)
Thanks you for your attention.