Description of a Parallel Application

CS267 Assignment 0

Jay Mistry
Brief Bio

• I am an EECS MEng. Student with specialization in Integrated Circuits.
• My project topic is design exploration of Network-on-Chip.
• I expect to learn the general methodology for parallel programming that I expect to play a key role in the future of multiple core computing domain.
Problem Definition

• Parallelize the computation of one dimensional Discrete Fast Fourier Transform (DFFT) on Intel Xeon processor.

• Original code is a single-threaded kernel, which is part of Intel’s Math Kernel Library (MKL) that was parallelized to produce Enormous Fast Fourier Transform (EFFT).
Concept

• The concept is to take original single-threaded kernel and modify it to achieve thread level parallelism to run it on multi-core processor using Intel’s Cilk Plus framework.

• The parallelization is achieved by implementing techniques like tiling, strip-mining, and vectorized data.
Original Algorithm

- Original algorithm calculates the transform based on the following equation:

\[ F_k = \sum_{n=0}^{N-1} x_n \cdot \exp \left( -i \frac{2\pi kn}{N} \right) \]

- \( F_k \) = k\textsuperscript{th} coefficient in the transform
- \( x_n \) = input data points
- \( N \) = length of transformed sequence

```c
// (1) creating the descriptor for single-precision DFFFT on real data
const int N = 1<<28;
float* data = (float*)
__mm_malloc(N*sizeof(float), 64);
MKL_LONG fftsize = N;
DFTI_DESCRIPTOR_HANDLE* fftHandle =
new DFTI_DESCRIPTOR_HANDLE;
DftiCreateDescriptor(fftHandle, DFTI_SINGLE,
DFTI_REAL, 1, fftsize);

// (2) configuring the descriptor:
// single-threaded transform,
// packed permuted data format
DftiSetValue(*fftHandle,
DFTI_NUMBER_OF_USER_THREADS, 1);
DftiSetValue(*fftHandle, DFTI_PACKED_FORMAT,
DFTI_PERM_FORMAT);
DftiCommitDescriptor (*fftHandle);

// (3) carrying out in-place DFT
DftiComputeForward(*fftHandle, data)
```
Modified Algorithm

• In modified algorithm the even and the odd part of the transformed are separated as follows:

\[
F_k = \sum_{n=0}^{N/2-1} x_{2n} \cdot \exp \left[ -i \frac{2\pi kn}{N} \right] + \sum_{n=0}^{N/2-1} x_{2n+1} \cdot \exp \left[ -i \frac{2\pi k(n+1)}{N} \right] = \\
= \sum_{n=0}^{N/2-1} x_{2n} \cdot \exp \left[ -i \frac{2\pi kn}{N/2} \right] + \\
e^{-i2\pi k/N} \sum_{n=0}^{N/2-1} x_{2n+1} \cdot \exp \left[ -i \frac{2\pi kn}{N/2} \right] \equiv E_k + e^{-i2\pi k/N} O_k.
\]

• Value of \( E_k \) and \( O_k \) need to be found only for \( 0 < k < N/2 \), because from periodicity the complex exponential functions follows the symmetry:

\[
F_k = E_k + e^{-i2\pi k/N} O_k, \\
F_{k+N/2} = E_k - e^{-i2\pi k/N} O_k.
\]
Modified Algorithm

- The modified algorithm is divided into three components:
  - “Scattering” of \( x_n \) into multiple continuous arrays (bins), which are then transformed.
  - “Processing” the transform and
  - “Reassembly” of the bins to form \( F_k \)
Modified Algorithm - Scattering

- Scattering using conventional \texttt{for}-loop.

```cpp
for (int i = 0; i < N; i+=2) {
    temparray[i] = data[2*i];
    temparray[i+N/2] = data[2*i+1];
}
```

- The scattering is done using \texttt{cilk_for} that parallelize the scattering process among multiple threads.

```cpp
int scatter_index[numofbins];
find_scatter_index(scatter_index, numofsplits);
cilk_for(int j = 0; j < numofbins; j++)
    for (long i = 0; i < binsize; i++)
        tempArray[scatter_index[j]*binsize+i] =
            dataArray[i*numofbins+j];
```

- In this fully optimized version the scattering is distributed among multiple threads and the loop is strip-mined to take advantage of spatial locality.

```cpp
int scatter_index[numofbins];
find_scatter_index(scatter_index, numofsplits);
const long iTILE=16;
long ii;
cilk_for (ii = 0; ii < binsize; ii += iTILE)
    for(long j = 0; j < numofbins; j++)
        for (long i = ii; i < ii+iTILE; i++)
            tempArray[scatter_index[j]*binsize+i] =
                dataArray[i*numofbins+j];
```
The code on the right represents the Processing part of the algorithm. Each segment of \( x_n \) that were produced in Scattering are transformed as an independent individual tasks that in turn will be executed in parallel through the \texttt{cilk\_spawn} flag. The \texttt{cilk\_sync} flag will synchronize the spawned tasks and resume the execution after the completion of each transform.

```c
void ProcessAndReassemble(
    DFTI_DESCRIPTOR_HANDLE* fftHandle,
    float* array, float* temp,
    const long n, const long binsize) {
    const long size = n/2;

    if (n > binsize) {
        cilk_spawn // Create parallel task
            ProcessAndReassemble(fftHandle,
                array, temp, size, binsize);
            ProcessAndReassemble(fftHandle,
                &array[size], &temp[size],
                size, binsize);
        cilk_sync; // Wait for spawned task

        // CT algorithm to combine evens and odds
        // ReassemblePair() is multithreaded
        ReassemblePair(temp, size);
    } else {
        // Process the small enough array
        // with serial MKL implementation of DFT
        int wid = __cilkrts_get_worker_number();
        DftiComputeForward(fftHandle[wid], temp);
    }
}
```
The reassembly code organizes the array with strides such that the compiler can vectorize the strided data.

```c
// Start gather second pair (mirror)
oddmirrek[:] = odds[size-kk :k TILE:-2];
oddmirimk[:] = odds[size-kk+1:k TILE:-2];

// Finish reassembly of first pair
odds[size-kk :k TILE:-2] = evenrek[:] -
    coslist[:]*oddrek[:] +sinlist[:]*oddimk[:];
odds[size-kk+1:k TILE:-2] = -evenimk[:] +
    sinlist[:]*oddrek[:] +coslist[:]*oddimk[:];

// Finish gathering second pair (mirror)
evenmirrek[:] = evens[size-kk :k TILE:-2];
evenmirimk[:] = evens[size-kk+1:k TILE:-2];

// Reassemble second pair of elements
// and scatter results into stride-2 array
evens[size-kk :k TILE:-2]= evenmirrek[:] -
    sinlist[:]*oddmirrek[:] +
    coslist[:]*oddmirimk[:];
evens[size-kk+1:k TILE:-2]= evenmirimk[:] -
    coslist[:]*oddmirrek[:] -
    sinlist[:]*oddmirimk[:];
odd [kk :k TILE: 2]= evenmirrek[:] +
    sinlist[:]*oddmirrek[:] -
    coslist[:]*oddmirimk[:];
odd [kk+1 :k TILE: 2]= evenmirimk[:] -
    coslist[:]*oddmirrek[:] -
    sinlist[:]*oddmirimk[:];
```

// Vectorized twiddle factor precomputation

```c
const float theta=(float)(kk/2)*trigconst;
#pragma simd
#pragma vec aligned
for (int i = 0; i < k TILE; i++) {
    sinlist[i] = sinf(theta+i*trigconst);
    coslist[i] = cosf(theta+i*trigconst);
}
```

// Gather stride-2 data for first pair

```c
evenrek[:] = evens[kk :k TILE:2];
evenimk[:] = evens[kk+1:k TILE:2];
oddrek [:] = odds [kk :k TILE:2];
oddimk [:] = odds [kk+1:k TILE:2];
```

// Reassemble first pair

```c
// and scatter results into stride-2 array
evens[kk :k TILE:2] = evenrek[:]+
    coslist[:]*oddrek[:]-sinlist[:]*oddimk[:];
evens[kk+1:k TILE:2] = evenimk[:]+
    sinlist[:]*oddrek[:]+coslist[:]*oddimk[:];
```
Performance Result

• Performance measurement in terms of GFLOPS/s between EFFT, Intel’s MKL library, and Fast Fourier Transform in the West (FFTW) library as a function of transform size N.
• Performance was measured on Colfax ProEdge™ SXP8600 powered by Intel Xeon E5-2697 v2 processor (12 cores per socket, 24 cores total).
• The performance of MKL is superior for transform size $N \leq 10^{22}$ after which EFFT dominates the performance.
Accuracy Measurement

- Accuracy measurement of EFFT, FFTW, and MKL libraries in terms of L2 norm of the deviation from the exact solution as a function of transform size $N$.
- All libraries possess accuracy with L2 norm in range of $10^{-7}$ up to $N = 10^{22}$ after which the accuracy of MKL degrades while the accuracy of EFFT and FFTW persists.
Scalability

• The scalability of EFFT amongst multiple core was measured by distributing each thread across individual cores for transform $N = 10^{30}$.

• Performance measurements in terms of GFLOPS/s versus no. of threads is compared with STREAM – a benchmark program that measures sustainable memory bandwidth and the corresponding rate for simple vector kernels.

• Up until threads $T \leq 12$, EFFT performs better than STREAM beyond which its performance degrades because for $T > 12$, part of computation is sent to second socket in which case there’s overhead of data transfer between the two sockets.
Conclusion

• Overall EFFT represents an efficient library that exploits thread level parallelism and can be run on many core machines.
• Parallel techniques like tiling, strip-mining, and vectorized data are used and task level parallelism was obtained through Intel’s Cilk Plus framework.
• EFFT efficiently scales to multiple cores by distributing the threads across individual cores. There’s quite an overhead when the distribution is across multiple CPUs, however this can be compensated when the number of transform is large.
• Based on performance results there’s still a room for improvement for the library. While exact reason is not diagnosed for its under-performance for $N < 10^{22}$, the library can be fine tuned to operate differently for $N < 10^{22}$ that might improve the results.
Source


• Web. *Fastest Fourier Transform in the West (FFTW).* en.wikipedia.org/wiki/FFTW. Date 01/25/2015.