Parallel Sparse FFT

Cheng Wang*, Mauricio Araya-Polo†, Sunita Chandrasekaran*, Amik St-Cyr*, Barbara Chapman* and Detlef Hohl†

*HPCTools Group, Department of Computer Science, University of Houston, Houston, TX, 77004, USA
†Shell International E&P Inc., Houston, TX, 77082, USA

IA³ Workshop on Irregular Applications: Architectures & Algorithms, in conjunction with SC’13, Denver, Colorado

November 17, 2013
Image courtesy of xkcd comics, http://xkcd.com/26/
Why is the Fourier Transform so important?

1. Discrete Fourier transform: given $x \in \mathbb{C}^n$, find

$$\hat{x}_i = \sum x_j \omega^{ij}$$  \hspace{1cm} (1)
Why is the Fourier Transform so important?

1. Discrete Fourier transform: given \( x \in \mathbb{C}^n \), find

\[
\hat{x}_i = \sum x_j \omega^{ij}
\] (1)

2. Fundamental tool
   - Signal processing
   - Compression (audio, image, video)
   - Data analysis
   - ...

Presented by Cheng Wang
November 17, 2013
cwang35@uh.edu
Why is the Fourier Transform so important?

1. Discrete Fourier transform: given $x \in \mathbb{C}^n$, find

$$\hat{x}_i = \sum x_j \omega^{ij}$$

2. Fundamental tool
   - Signal processing
   - Compression (audio, image, video)
   - Data analysis
   - ...

3. FFT: $O(n \log n)$ time
Why is FFT sometimes inefficient?

Often the Fourier transform is dominated by a small number of “peaks.” FFT performs $n$ input data only to lead to small number of large coefficients.
Why is FFT sometimes inefficient?

1. Often the Fourier transform is dominated by a small number of "peaks".
2. FFT performs $n$ input data only to lead to a small number of large coefficients.
Often the Fourier transform is dominated by a small number of “peaks”
Why is FFT sometimes inefficient?

1. Often the Fourier transform is dominated by a small number of “peaks”
2. FFT performs $n$ input data only to lead to small number of large coefficients
Why is FFT sometimes inefficient?

1. Often the Fourier transform is dominated by a small number of “peaks”
2. FFT performs $n$ input data only to lead to small number of large coefficients
Sparsity is everywhere

Figure: Seismic data
Sparsity is everywhere

Figure: Biomedical images
Sparsity is everywhere

Figure: Social graph data
• Compute the $k$-sparse Fourier transform with complexity: $O(\log n \sqrt{nk\log n})$
Compute the $k$-sparse Fourier transform with complexity:
$O(\log n \sqrt{nk \log n})$

The algorithm is faster than full FFT for $k$ up to $O(n/\log n)$
Our major contributions:

1. Reimplemented the sFFT and doubled the performance
2. Proposed a parallel sFFT (PsFFT) algorithm for multicore CPUs
3. Compared the PsFFT with other parallel full-sized FFT libraries and obtained promising performance improvements
sFFT: From theory to practice

Our major contributions:

1. Reimplemented the sFFT and doubled the performance
sFFT: From theory to practice

**Our major contributions:**

1. Reimplemented the sFFT and doubled the performance

2. Proposed a parallel sFFT (PsFFT) algorithm for multicore CPUs
sFFT: From theory to practice

Our major contributions:

1. Reimplemented the sFFT and doubled the performance

2. Proposed a parallel sFFT (PsFFT) algorithm for multicore CPUs

3. Compared the PsFFT with other parallel full-sized FFT libraries and obtained promising performance improvements
Improvements over the sequential version

1. Reimplementation from C++ to C
   - More friendly for accelerators and low-power embedded systems
Improvements over the sequential version

1. Reimplementation from C++ to C
   - More friendly for accelerators and low-power embedded systems

2. Optimizations on multiple levels
Improvements over the sequential version

1. Reimplementation from C++ to C
   - More friendly for accelerators and low-power embedded systems

2. Optimizations on multiple levels
   - **Parallel optimizations**: Eliminated loop-carried dependence and non-thread-safe functions
Improvements over the sequential version

1. Reimplementation from C++ to C
   - More friendly for accelerators and low-power embedded systems

2. Optimizations on multiple levels
   - **Parallel optimizations**: Eliminated loop-carried dependence and non-thread-safe functions
   - **Data structure optimizations**: Replaced the advanced and nested data structures with flat data types

Presented by Cheng Wang
November 17, 2013
cwang35@uh.edu
Improvements over the sequential version

1. Reimplementation from C++ to C
   - More friendly for accelerators and low-power embedded systems

2. Optimizations on multiple levels
   - **Parallel optimizations**: Eliminated loop-carried dependence and non-thread-safe functions
   - **Data structure optimizations**: Replaced the advanced and nested data structures with flat data types
   - **Locality optimizations**: Cache blocking, TLB blocking, Register blocking
Improvements over the sequential version

Intel Xeon E5-2670 (Sandy Bridge)

Runtime (sec)

Original Implementation
Optimized Implementation
FFTW

• 2x performance improvement
• Stretched the upper bound of the sparsity $k$ from 4000 to 7000 to beat FFTW
Improvements over the sequential version

Intel Xeon E5-2670 (Sandy Bridge)

- 2x performance improvement

Presented by Cheng Wang  
November 17, 2013  
cwang35@uh.edu
Improvements over the sequential version

- 2x performance improvement
- Stretched the upper bound of the sparsity $k$ from 4000 to 7000 to beat FFTW
How does sFFT work?

Keep the coordinates that occurred in at least half of the location loops

Estimate the values of the coefficients

Presented by Cheng Wang  November 17, 2013  cwang35@uh.edu
Parallel Sparse FFT (PsFFT)

Step 1: Random spectrum permutation and filtering

- Randomly permutes the signal spectrum and bins into a smaller number of buckets

n coordinates

Time

Presented by Cheng Wang  November 17, 2013  cwang35@uh.edu
Parallel Sparse FFT (PsFFT)

Step 1: Random spectrum permutation and filtering

- Randomly permutes the signal spectrum and bins into a smaller number of buckets.
Parallel Sparse FFT (PsFFT)

**Step 1: Random spectrum permutation and filtering**

- Randomly permutes the signal spectrum and bins into a smaller number of buckets
- Simply partitioning the signal spectrum for each thread does not work
Parallel Sparse FFT (PsFFT)

Step 1: Random spectrum permutation and filtering

- Randomly permutes the signal spectrum and bins into a smaller number of buckets
- Simply partitioning the signal spectrum for each thread does not work
- Multiple signal points are binned into the same bucket
- Potential race conditions
Parallel Sparse FFT (PsFFT)

Step 1: Random spectrum permutation and filtering (cont’)

- Our solution: partition the buckets

\[ B \approx \sqrt{nk} \]

\( n \) coordinates

\( B \) buckets

\( T_0, T_1, T_2 \)
Parallel Sparse FFT (PsFFT)

Step 1: Random spectrum permutation and filtering (cont’)

- Our solution: partition the buckets
- No race conditions
Parallel Sparse FFT (PsFFT)

Step 1: Random spectrum permutation and filtering (cont’)

- Our solution: partition the buckets
- No race conditions
- number of buckets $B \approx \sqrt{nk}$, still sufficient data parallelism
Parallel Sparse FFT (PsFFT)

Step 2: Subsampled FFT

- Subsampled B-dimensional FFT

B buckets (time)

B buckets (frequency)

Frequency

Use parallel FFTW to fulfill this function
Step 2: Subsampled FFT

- Subsampled B-dimensional FFT
- Use parallel FFTW to fulfill this function
Step 3: Select the k-largest Fourier coefficients

- Since signal spectrum is sparse, most of the buckets are very small.
Step 3: Select the k-largest Fourier coefficients

- Since signal spectrum is sparse, most of the buckets are very small
- Select the top $k$ largest coefficients from the $B$ sized buckets
Step 3: Select the k-largest Fourier coefficients

- Since signal spectrum is sparse, most of the buckets are very small
- Select the top $k$ largest coefficients from the $B$ sized buckets
- Use heap sort, achieved in $O(B)$ time
Parallel Sparse FFT (PsFFT)

Step 4: Reverse hash function for location recovery
Step 5: Magnitude recovery

- Find the locations of the large coefficients
- Recover the magnitudes of the coefficients found

Presented by Cheng Wang  November 17, 2013  cwang35@uh.edu
Parallel Sparse FFT (PsFFT)

Step 4: Reverse hash function for location recovery

Step 5: Magnitude recovery

- Find the locations of the large coefficients
- Recover the magnitudes of the coefficients found
- Each thread processes $k/nthreads$ elements
Performance optimizations: Challenges & solution

1. Irregular memory access pattern

- Irregular data reference pattern
- n coordinates
- B buckets
- Poor spatial locality

Solution: multi-level blocking techniques
- Keeping data in cache and registers
- Reducing memory bandwidth pressure

Presented by Cheng Wang
November 17, 2013
Performance optimizations: Challenges & solution

1. Irregular memory access pattern
   - Poor spatial locality
Performance optimizations: Challenges & solution

1. Irregular memory access pattern
   - Poor spatial locality

2. Solution: multi-level blocking techniques

Presented by Cheng Wang
November 17, 2013
cwang35@uh.edu
Performance optimizations: Challenges & solution

1. Irregular memory access pattern
   - Poor spatial locality

2. Solution: multi-level blocking techniques
   - Keeping data in cache and registers
Performance optimizations: Challenges & solution

1. Irregular memory access pattern
   - Poor spatial locality

2. Solution: multi-level blocking techniques
   - Keeping data in cache and registers
   - Reducing memory bandwidth pressure
Performance evaluation

**Figure:** Elapsed time: sequential v.s. parallel sFFT

- Implemented PsFFT using OpenMP
Performance evaluation

Figure: Elapsed time: sequential v.s. parallel sFFT

- Implemented PsFFT using OpenMP
- Pink area selects the representative ranges of sparsity $k$
Performance evaluation

Figure: Elapsed time: sequential v.s. parallel sFFT

- Implemented PsFFT using OpenMP
- Pink area selects the representative ranges of sparsity $k$
- Most of the area is shifted down to the curve of the parallel FFTW
Performance evaluation

- Implemented PsFFT using OpenMP
- Pink area selects the representative ranges of sparsity $k$
- Most of the area is shifted down to the curve of the parallel FFTW
- Over 5x faster than parallel FFTW

**Figure:** Elapsed time: sequential v.s. parallel sFFT
Scalability

• Evaluated the scalability on a 8-core Intel Sandy Bridge architecture
Scalability

- Evaluated the scalability on a 8-core Intel Sandy Bridge architecture
- Achieved over 5x for most of the cases
Develop a high-performance and portable parallel sparse FFT library
Parallel sparse FFT for GPUs and accelerators
sFFT is an interesting irregular algorithm
Eliminating the dynamic irregular data reference pattern is still an open research question
Thank You For Your Attention