Parallel FFT & Isoefficiency

1. The Fast Fourier Transform in Parallel
   - the Fastest Fourier Transform in the West (FFTW)
   - FFTW for signal processing
   - running the OpenMP version of FFTW

2. Isoefficiency
   - efficiency and scalability
   - the isoefficiency function
Parallel FFT & Isoefficiency

1. The Fast Fourier Transform in Parallel
   - The Fastest Fourier Transform in the West (FFTW)
   - FFTW for signal processing
   - Running the OpenMP version of FFTW

2. Isoefficiency
   - Efficiency and scalability
   - The isoefficiency function
the Discrete Fourier Transform (DFT)

A periodic function $f(t)$ can be written as a series of sinusoidal waveforms of various frequencies and amplitudes: the Fourier series. The Fourier transform maps $f$ from the time to the frequency domain. In discrete form:

$$F_k = \frac{1}{n} \sum_{j=0}^{n-1} f_j e^{-2\pi i (jk/n)}, \quad k = 0, 1, \ldots, n - 1, f_k = f(x_k).$$

The Discrete Fourier Transform (DFT) maps a convolution into a componentwise product.

The Fast Fourier Transform is an algorithm that reduces the cost of the DFT from $O(n^2)$ to $O(n \log(n))$, for length $n$.

Many applications, for example: signal and image processing.
the Fastest Fourier Transform in the West (FFTW)

FFTW is a library for the Discrete Fourier Transform (DFT), developed at MIT by Matteo Frigo and Steven G. Johnson available under the GNU GPL license at http://www.fftw.org.

FFTW 3.3 supports MPI and comes with multithreaded versions: with Cilk, Pthreads and OpenMP are supported.

Before make install, do

./configure --enable-mpi --enable-openmp --enable-threads

FFTW received the 1999 J. H. Wilkinson Prize for Numerical Software.
Parallel FFT & Isoefficiency

1. The Fast Fourier Transform in Parallel
   - the Fastest Fourier Transform in the West (FFTW)
   - FFTW for signal processing
   - running the OpenMP version of FFTW

2. Isoefficiency
   - efficiency and scalability
   - the isoefficiency function
Consider $f(t) = 2 \cos(4 \times 2\pi t) + 5 \sin(10 \times 2\pi t)$, for $t \in [0, 1]$. 

![Graph of the test signal](image-url)
processing a test signal

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <fftw3.h>

double my_signal ( double t );
/
 * Defines a signal composed of
 * a cosine of frequency 4 and amplitude 2,
 * a sine of frequency 10 and amplitude 5. */

void sample_signal 
 ( double f ( double t ), int m,
   double *x, double *y );
/
 * Takes m = 2^k samples of the signal f,
 * returns abscisses in x and samples in y. */
```
calling FFTW

```c
int main ( int argc, char *argv[] )
{
    const int n = 256;
    double *x, *y;
    x = (double*)calloc(n,sizeof(double));
    y = (double*)calloc(n,sizeof(double));
    sample_signal(my_signal,n,x,y);

    int m = n/2+1;
    fftw_complex *out;
    out = (fftw_complex*)fftw_malloc(sizeof(fftw_complex)*m);

    fftw_plan p;
    p = fftw_plan_dft_r2c_1d(n,y,out,FFTW_ESTIMATE);
    fftw_execute(p);
}
```
```c
printf("scanning through the output of FFTW...\n");
double tol = 1.0e-8; /* threshold on noise */
int i;
for(i=0; i<m; i++)
{
    double v = fabs(out[i][0])/(m-1)
          + fabs(out[i][1])/(m-1);
    if(v > tol)
    {
        printf("at %d : (%.5e,%.5e)\n", i,out[i][0],out[i][1]);
        printf("=> frequency %d and amplitude %.3e\n", i,v);
    }
}
return 0;
```
compiling and running

```
$ make fftw_use
gcc fftw_use.c -o /tmp/fftw_use -lfftw3 -lm

$ /tmp/fftw_use
scanning through the output of FFTW...
at 4 : (2.56000e+02,-9.63913e-14)
=> frequency 4 and amplitude 2.000e+00
at 10 : (-7.99482e-13,-6.40000e+02)
=> frequency 10 and amplitude 5.000e+00
```

We recovered the frequencies and amplitudes in the components of our test signal.
Parallel FFT & Isoefficiency

1. The Fast Fourier Transform in Parallel
   - the Fastest Fourier Transform in the West (FFTW)
   - FFTW for signal processing
   - running the OpenMP version of FFTW

2. Isoefficiency
   - efficiency and scalability
   - the isoefficiency function
**timing the FFTW – the setup**

```c
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <fftw3.h>

void random_sequence ( int n, double *re, double *im );
/*
 * Returns a random sequence of n complex numbers
 * with real parts in re and imaginary parts in im. */

int main ( int argc, char *argv[] )
{
    int n,m;
    if(argc > 1)
        n = atoi(argv[1]);
    else
        printf("please provide dimension on command line\n");
    m = (argc > 2) ? atoi(argv[2]) : 1;
```
computing \( m \) times for dimension \( n \)

double *x, *y;
x = (double*)calloc(n,sizeof(double));
y = (double*)calloc(n,sizeof(double));
random_sequence(n,x,y);

fftw_complex *in,*out;
in = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*n);
out = (fftw_complex*) fftw_malloc(sizeof(fftw_complex)*n);

fftw_plan p;
p = fftw_plan_dft_1d(n,in,out,FFTW_FORWARD,FFTW_ESTIMATE);
clock_t tstart,tstop;
tstart = clock();
int i;
for(i=0; i<m; i++) fftw_execute(p);
tstop = clock();
printf("%d iterations took %.3f seconds\n",
m,(tstop-tstart)/((double) CLOCKS_PER_SEC));
#include <omp.h>
#include <fftw3.h>

int main ( int argc, char *argv[] )
{
    int t; /* number of threads */

    t = (argc > 3) ? atoi(argv[3]) : 1;

    int okay = fftw_init_threads();
    if(okay == 0)
        printf("error in thread initialization\n");
    omp_set_num_threads(t);
    fftw_plan_with_nthreads(t);
collecting running times

$\text{time /tmp/fftw\_timing\_omp 100000 1000 } p$

Computing 1,000 DFTs for $n = 100,000$, with $p$ threads:

<table>
<thead>
<tr>
<th>$p$</th>
<th>real</th>
<th>user</th>
<th>sys</th>
<th>speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.443s</td>
<td>2.436s</td>
<td>0.004s</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.340s</td>
<td>2.655s</td>
<td>0.010s</td>
<td>1.823</td>
</tr>
<tr>
<td>4</td>
<td>0.774s</td>
<td>2.929s</td>
<td>0.008s</td>
<td>3.156</td>
</tr>
<tr>
<td>8</td>
<td>0.460s</td>
<td>3.593s</td>
<td>0.017s</td>
<td>5.311</td>
</tr>
<tr>
<td>16</td>
<td>0.290s</td>
<td>4.447s</td>
<td>0.023s</td>
<td>8.424</td>
</tr>
</tbody>
</table>

where real = wall clock time, user = cpu time, sys = system time, obtained with time on 2 8-core processors at 2.60 GHz.
collecting more running times

$ \text{time} /\text{tmp/fftw\_timing\_omp} \ 1000000 \ 1000 \ p$

Computing 1,000 DFTs for $n = 1,000,000$, with $p$ threads:

<table>
<thead>
<tr>
<th>$p$</th>
<th>real</th>
<th>user</th>
<th>sys</th>
<th>speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>44.806s</td>
<td>44.700s</td>
<td>0.014s</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>22.151s</td>
<td>44.157s</td>
<td>0.020s</td>
<td>2.023</td>
</tr>
<tr>
<td>4</td>
<td>10.633s</td>
<td>42.336s</td>
<td>0.019s</td>
<td>4.214</td>
</tr>
<tr>
<td>8</td>
<td>6.998s</td>
<td>55.630s</td>
<td>0.036s</td>
<td>6.403</td>
</tr>
<tr>
<td>16</td>
<td>3.942s</td>
<td>62.250s</td>
<td>0.134s</td>
<td>11.366</td>
</tr>
</tbody>
</table>

where real = wall clock time, user = cpu time, sys = system time, obtained with \text{time} on 2 8-core processors at 2.60 GHz.

As $n$ increases ten fold, the speedups improve.
The Fast Fourier Transform in Parallel
- the Fastest Fourier Transform in the West (FFTW)
- FFTW for signal processing
- running the OpenMP version of FFTW

Isoefficiency
- efficiency and scalability
- the isoefficiency function
efficiency and scalability

For $p$ processors:

$$\text{Speedup} = \frac{\text{serial time}}{\text{parallel time}} = S(p) \rightarrow p$$

$$\text{Efficiency} = \frac{\text{Speedup}}{p} = \frac{S(p)}{p} = E(p) \rightarrow 1$$

Let $T_s$ denote the serial time, $T_p$ the parallel time, and $T_0$ the overhead, then: $pT_p = T_s + T_0$.

$$E(p) = \frac{T_s}{pT_p} = \frac{T_s}{T_s + T_0} = \frac{1}{1 + T_0/T_s}$$

The scalability analysis of a parallel algorithm measures its capacity to effectively utilize an increasing number of processors.
relating $E$ to $W$ and $T_0$

Let $W$ be the problem size, for FFT: $W = n \log(n)$.

The overhead $T_0$ depends on $W$ and $p$: $T_0 = T_0(W, p)$.

The parallel time equals $T_p = \frac{W + T_0(W, p)}{p}$.

Speedup $S(p) = \frac{W}{T_p} = \frac{Wp}{W + T_0(W, p)}$.

Efficiency $E(p) = \frac{S(p)}{p} = \frac{W}{W + T_0(W, p)} = \frac{1}{1 + \frac{T_0(W, p)}{W}}$.

The goal is for $E(p) \to 1$ as $p \to \infty$.

The algorithm scales badly if $W$ must grow exponentially to keep efficiency from dropping. If $W$ needs to grow only moderately to keep the overhead in check, then the algorithm scales well.
Parallel FFT & Isoefficiency

1 The Fast Fourier Transform in Parallel
   - the Fastest Fourier Transform in the West (FFTW)
   - FFTW for signal processing
   - running the OpenMP version of FFTW

2 Isoefficiency
   - efficiency and scalability
   - the isoefficiency function
Isoefficiency relates work to overhead

\[ E = \frac{1}{1 + \frac{T_0(W, p)}{W}} \Rightarrow \frac{1}{E} = \frac{1 + \frac{T_0(W, p)}{W}}{1} \]
\[ \Rightarrow \frac{1}{E} - 1 = \frac{T_0(W, p)}{W} \]
\[ \Rightarrow \frac{1 - E}{E} = \frac{T_0(W, p)}{W}. \]

The isoeficiency function is

\[ W = \left( \frac{E}{1 - E} \right) T_0(W, p) \quad \text{or} \quad W = K T_0(W, p). \]

Keeping \( K \) constant, isoeficiency relates \( W \) to \( T_0 \).

- Amdahl’s Law: keep \( W \) fixed and let \( p \) grow.
- Gustafson’s Law: keep \( p \) fixed and let \( W \) grow.
The isoefficiency function: $W = K T_0(W, p)$.

For FFT: $T_s = n \log(n) t_c$, where $t_c$ is the time for complex multiplication and adding a pair.

Let $t_s$ denote the startup cost and $t_w$ denote the time to transfer a word.

Time for a parallel FFT:

$$T_p = t_c \left( \frac{n}{p} \right) \log(n) + t_s \log(p) + t_w \left( \frac{n}{p} \right) \log(p).$$
start up cost versus computation

Using the expression for $T_p$ in the efficiency $E(p)$:

$$E(p) = \frac{T_s}{pT_p} = \frac{n \log(n)t_c}{n \log(n)t_c + p \log(p)t_s + n \log(p)t_w}$$

$$= \frac{Wt_c}{Wt_c + p \log(p)t_s + n \log(p)t_w}, \quad W = n \log(n).$$

Assume $t_w = 0$ (shared memory): $E(p) = \frac{Wt_c}{Wt_c + p \log(p)t_s}$.

We want to express $K = \frac{E}{1 - E}$, using $\frac{1}{K} = \frac{1 - E}{E} = \frac{1}{E} - 1$:

$$\frac{1}{K} = \frac{Wt_c + p \log(p)t_s}{Wt_c} - \frac{Wt_c}{Wt_c} \Rightarrow W = K \left( \frac{t_s}{t_c} \right) p \log(p).$$
isoefficiency for shared memory

isoefficiency $W = p \cdot \log(p)$, for shared memory

![Graph showingisoefficiency $W = p \cdot \log(p)$ for shared memory. The graph plots work load $W$ against the number of processors $p$, showing a linear increase.](image)
transfer cost versus computation

Taking another look at the efficiency $E(p)$:

$$E(p) = \frac{Wt_c}{Wt_c + p \log(p) t_s + n \log(p) t_w}, \quad W = n \log(n).$$

Assume $t_s = 0$ (no start up): $E(p) = \frac{Wt_c}{Wt_c + n \log(p) t_w}$.

We want to express $K = \frac{E}{1 - E}$, using $\frac{1}{K} = \frac{1 - E}{E} = \frac{1}{E} - 1$:

$$\frac{1}{K} = \frac{Wt_c + n \log(p) t_w}{Wt_c} - \frac{Wt_c}{Wt_c} \Rightarrow W = K \left( \frac{t_w}{t_c} \right) n \log(p).$$
efficiency plot for distributed memory

efficiency $E(p) = 1/(1 + \log(p)/\log(n))$, for distributed memory

- number of processors $p$, for $n$ from 10,000 to 100,000
recommended reading

Summary + Exercises

In the book of Wilkinson and Allen, the parallel FFT is in §12.7.

Exercises:

1. Investigate the speed up for the parallel FFTW with threads.

2. Consider the isoefficiency formulas we derived for a parallel version of the FFT. Suppose an efficiency of 0.6 is desired. For values $t_c = 1$, $t_s = 25$ and $t_w = 4$, make plots of the speedup for increasing values of $n$, taking $p = 64$. Interpret the plots relating to the particular choices of the parameters $t_c$, $t_s$, $t_w$, and the desired efficiency.

3. Relate the experimental speed ups with the theoretical isoefficiency formulas.