## 12.6 External Storage or Memory-Local FFTs

Sometime in your life, you might have to compute the Fourier transform of a *really large* data set, larger than the size of your computer's physical memory. In such a case, the data will be stored on some external medium, such as magnetic or optical tape or disk. Needed is an algorithm that makes some manageable number of sequential passes through the external data, processing it on the fly and outputting intermediate results to other external media, which can be read on subsequent passes.

In fact, an algorithm of just this description was developed by Singleton [1] very soon after the discovery of the FFT. The algorithm requires four sequential storage devices, each capable of holding half of the input data. The first half of the input data is initially on one device, the second half on another.

Singleton's algorithm is based on the observation that it is possible to bit-reverse $2^M$ values by the following sequence of operations: On the first pass, values are read alternately from the two input devices, and written to a single output device (until it holds half the data), and then to the other output device. On the second pass, the output devices become input devices, and vice versa. Now, we copy *two* values from the first device, then *two* values from the second, writing them (as before) first to fill one output device, then to fill a second. Subsequent passes read 4, 8, etc., input values at a time. After completion of pass $M-1$, the data are in bit-reverse order.

Singleton's next observation is that it is possible to alternate the passes of essentially this bit-reversal technique with passes that implement one stage of the Danielson-Lanczos combination formula (12.2.3). The scheme, roughly, is this: One starts as before with half the input data on one device, half on another. In the first pass, one complex value is read from each input device. Two combinations are formed, and one is written to each of two output devices. After this "computing" pass, the devices are rewound, and a "permutation" pass is performed, where groups of values are read from the first input device and alternately written to the first and second output devices; when the first input device is exhausted, the second is similarly processed. This sequence of computing and permutation passes is repeated $M-K-1$ times, where $2^K$ is the size of internal buffer available to the program. The second phase of the computation consists of a final $K$ computation passes. What distinguishes the second phase from the first is that, now, the permutations are local enough to do in place during the computation. There are thus no separate permutation passes in the second phase. In all, there are $2M-K-2$ passes through the data.

Here is an implementation of Singleton's algorithm, based on [1]:

```
#include <stdio.h>
#include <math.h>
#include "nrutil.h"
#define KBF 128

void fourfs(FILE *file[5], unsigned long nn[], int ndim, int isign)
```

One- or multi-dimensional Fourier transform of a large data set stored on external media. On input, `ndim` is the number of dimensions, and `nn[1..ndim]` contains the lengths of each dimension (number of real and imaginary value pairs), which must be powers of two. `file[1..4]` contains the stream pointers to 4 temporary files, each large enough to hold half of the data. The four streams must be opened in the system's "binary" (as opposed to "text") mode. The input data must be in C normal order, with its first half stored in file `file[1]`, its second half in `file[2]`, in native floating point form. `KBF` real numbers are processed per buffered read or write. `isign` should be set to 1 for the Fourier transform, to $-1$ for its inverse. On output, values in the array `file` may have been permuted; the first half of the result is stored in `file[3]`, the second half in `file[4]`. N.B.: For `ndim` $> 1$, the output is stored by columns, i.e., *not* in C normal order; in other words, the output is the transpose of that which would have been produced by routine `fourn`.

```
{
    void fourew(FILE *file[5], int *na, int *nb, int *nc, int *nd);
    unsigned long j,j12,jk,k,kk,n=1,mm,kc=0,kd,ks,kr,nr,ns,nv;
    int cc,na,nb,nc,nd;
```

```
float tempr,tempi,*afa,*afb,*afc;
double wr,wi,wpr,wpi,wtemp,theta;
static int mate[5] = {0,2,1,4,3};

afa=vector(1,KBF);
afb=vector(1,KBF);
afc=vector(1,KBF);
for (j=1;j<=ndim;j++) {
    n *= nn[j];
    if (nn[j] <= 1) nrerror("invalid float or wrong ndim in fourfs");
}
nv=1;
jk=nn[nv];
mm=n;
ns=n/KBF;
nr=ns >> 1;
kd=KBF >> 1;
ks=n;
fourew(file,&na,&nb,&nc,&nd);
The first phase of the transform starts here.
for (;;) {                                      Start of the computing pass.
    theta=isign*3.141592653589793/(n/mm);
    wtemp=sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0;
    wi=0.0;
    mm >>= 1;
    for (j12=1;j12<=2;j12++) {
        kr=0;
        do {
            cc=fread(&afa[1],sizeof(float),KBF,file[na]);
            if (cc != KBF) nrerror("read error in fourfs");
            cc=fread(&afb[1],sizeof(float),KBF,file[nb]);
            if (cc != KBF) nrerror("read error in fourfs");
            for (j=1;j<=KBF;j+=2) {
                tempr=((float)wr)*afb[j]-((float)wi)*afb[j+1];
                tempi=((float)wi)*afb[j]+((float)wr)*afb[j+1];
                afb[j]=afa[j]-tempr;
                afa[j] += tempr;
                afb[j+1]=afa[j+1]-tempi;
                afa[j+1] += tempi;
            }
            kc += kd;
            if (kc == mm) {
                kc=0;
                wr=(wtemp=wr)*wpr-wi*wpi+wr;
                wi=wi*wpr+wtemp*wpi+wi;
            }
            cc=fwrite(&afa[1],sizeof(float),KBF,file[nc]);
            if (cc != KBF) nrerror("write error in fourfs");
            cc=fwrite(&afb[1],sizeof(float),KBF,file[nd]);
            if (cc != KBF) nrerror("write error in fourfs");
        } while (++kr < nr);
        if (j12 == 1 && ks != n && ks == KBF) {
            na=mate[na];
            nb=na;
        }
        if (nr == 0) break;
    }
    fourew(file,&na,&nb,&nc,&nd);               Start of the permutation pass.
    jk >>= 1;
    while (jk == 1) {
        mm=n;
```

```
            jk=nn[++nv];
        }
        ks >>= 1;
        if (ks > KBF) {
            for (j12=1;j12<=2;j12++) {
                for (kr=1;kr<=ns;kr+=ks/KBF) {
                    for (k=1;k<=ks;k+=KBF) {
                        cc=fread(&afa[1],sizeof(float),KBF,file[na]);
                        if (cc != KBF) nrerror("read error in fourfs");
                        cc=fwrite(&afa[1],sizeof(float),KBF,file[nc]);
                        if (cc != KBF) nrerror("write error in fourfs");
                    }
                    nc=mate[nc];
                }
                na=mate[na];
            }
            fourew(file,&na,&nb,&nc,&nd);
        } else if (ks == KBF) nb=na;
        else break;
    }
    j=1;
```

The second phase of the transform starts here. Now, the remaining permutations are sufficiently local to be done in place.

```
    for (;;) {
        theta=isign*3.141592653589793/(n/mm);
        wtemp=sin(0.5*theta);
        wpr = -2.0*wtemp*wtemp;
        wpi=sin(theta);
        wr=1.0;
        wi=0.0;
        mm >>= 1;
        ks=kd;
        kd >>= 1;
        for (j12=1;j12<=2;j12++) {
            for (kr=1;kr<=ns;kr++) {
                cc=fread(&afc[1],sizeof(float),KBF,file[na]);
                if (cc != KBF) nrerror("read error in fourfs");
                kk=1;
                k=ks+1;
                for (;;) {
                    tempr=((float)wr)*afc[kk+ks]-((float)wi)*afc[kk+ks+1];
                    tempi=((float)wi)*afc[kk+ks]+((float)wr)*afc[kk+ks+1];
                    afa[j]=afc[kk]+tempr;
                    afb[j]=afc[kk]-tempr;
                    afa[++j]=afc[++kk]+tempi;
                    afb[j++]=afc[kk++]-tempi;
                    if (kk < k) continue;
                    kc += kd;
                    if (kc == mm) {
                        kc=0;
                        wr=(wtemp=wr)*wpr-wi*wpi+wr;
                        wi=wi*wpr+wtemp*wpi+wi;
                    }
                    kk += ks;
                    if (kk > KBF) break;
                    else k=kk+ks;
                }
                if (j > KBF) {
                    cc=fwrite(&afa[1],sizeof(float),KBF,file[nc]);
                    if (cc != KBF) nrerror("write error in fourfs");
                    cc=fwrite(&afb[1],sizeof(float),KBF,file[nd]);
                    if (cc != KBF) nrerror("write error in fourfs");
                    j=1;
                }
```

```
            }
            na=mate[na];
        }
        fourew(file,&na,&nb,&nc,&nd);
        jk >>= 1;
        if (jk > 1) continue;
        mm=n;
        do {
            if (nv < ndim) jk=nn[++nv];
            else {
                free_vector(afc,1,KBF);
                free_vector(afb,1,KBF);
                free_vector(afa,1,KBF);
                return;
            }
        } while (jk == 1);
    }
}
```

```
#include <stdio.h>
#define SWAP(a,b) ftemp=(a);(a)=(b);(b)=ftemp

void fourew(FILE *file[5], int *na, int *nb, int *nc, int *nd)
Utility used by fourfs. Rewinds and renumbers the four files.
{
    int i;
    FILE *ftemp;

    for (i=1;i<=4;i++) rewind(file[i]);
    SWAP(file[2],file[4]);
    SWAP(file[1],file[3]);
    *na=3;
    *nb=4;
    *nc=1;
    *nd=2;
}
```

For one-dimensional data, Singleton's algorithm produces output in exactly the same order as a standard FFT (e.g., `four1`). For multidimensional data, the output is the *transpose* of the conventional arrangement (e.g., the output of `fourn`). This peculiarity, which is intrinsic to the method, is generally only a minor inconvenience. For convolutions, one simply computes the component-by-component product of two transforms in their nonstandard arrangement, and then does an inverse transform on the result. Note that, if the lengths of the different dimensions are not all the same, then you must reverse the order of the values in `nn[1..ndim]` (thus giving the transpose dimensions) before performing the inverse transform. Note also that, just like `fourn`, performing a transform and then an inverse results in multiplying the original data by the product of the lengths of all dimensions.

We leave it as an exercise for the reader to figure out how to reorder `fourfs`'s output into normal order, taking additional passes through the externally stored data. We doubt that such reordering is ever really needed.

You will likely want to modify `fourfs` to fit your particular application. For example, as written, KBF $\equiv 2^K$ plays the dual role of being the size of the internal buffers, and the record size of the unformatted reads and writes. The latter role limits its size to that allowed by your machine's I/O facility. It is a simple matter to perform multiple reads for a much larger KBF, thus reducing the number of passes by a few.

Another modification of `fourfs` would be for the case where your virtual memory machine has sufficient address space, but not sufficient physical memory, to do an efficient FFT by the conventional algorithm (whose memory references are extremely nonlocal). In that case, you will need to replace the reads, writes, and rewinds by mappings of the arrays `afa`, `afb`, and `afc` into your address space. In other words, these arrays are replaced by references to a single data array, with offsets that get modified wherever `fourfs` performs an I/O operation. The resulting algorithm will have its memory references local within blocks of size `KBF`. Execution speed is thereby sometimes increased enormously, albeit at the cost of requiring twice as much virtual memory as an in-place FFT.

CITED REFERENCES AND FURTHER READING:

Singleton, R.C. 1967, *IEEE Transactions on Audio and Electroacoustics*, vol. AU-15, pp. 91–97. [1]

Oppenheim, A.V., and Schafer, R.W. 1989, *Discrete-Time Signal Processing* (Englewood Cliffs, NJ: Prentice-Hall), Chapter 9.