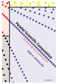
Program bspfft
(PSC §3.6)

Sequential unordered FFT: specification

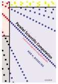
```
void ufft(double *x, int n, int sign, double *w){

/* This sequential function computes the
   unordered DFT of a complex vector x of
   length n, n=2^m, m>=0.
   If sign = 1, the forward DFT FRx is computed.
   If sign =-1, the backward UDFT conjg(F)Rx,
   where F is the n by n Fourier matrix and
         R the n by n bit-reversal matrix.
   The output overwrites x.
   w is a table of n/2 complex weights,
       exp(-2*pi*i*j/n), 0 <= j < n/2,
   which must have been initialised before
   calling this function.
*/
```



Data structure

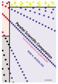
- A complex vector **x** of length *n* is stored as a real array `x` of size 2*n*.
- The real and imaginary parts alternate:

$$\begin{aligned}\texttt{x}[2*\texttt{j}] &= \mathrm{Re}(x_j)\\ \texttt{x}[2*\texttt{j}+1] &= \mathrm{Im}(x_j)\end{aligned}$$



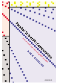
Sequential unordered FFT: body

```
for(k=2; k<=n; k *=2){
   nk= n/k;
   for(r=0; r<nk; r++){
      rk= 2*r*k;
      for(j=0; j<k; j +=2){
          wr= w[j*nk]; // real part of exp(-2*pi*i*j/k)
          if (sign==1)
             wi= w[j*nk+1];
          else
             wi= -w[j*nk+1];
          j0= rk+j; j1= j0+1;
          j2= j0+k; j3= j2+1;
          taur= wr*x[j2] - wi*x[j3];
          taui= wi*x[j2] + wr*x[j3];
          x[j2]= x[j0]-taur;  x[j3]= x[j1]-taui;
          x[j0] += taur;      x[j1] += taui;
}}}
```



Permutation to be used for bit reversal $\sigma = \rho_n$

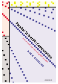
```
void permute(double *x, int n, int *sigma){
    /* This sequential function permutes a
       complex vector x by the permutation sigma
       (decomposable into disjoint swaps),
           y[j] = x[sigma[j]], 0 <= j < n.
       The output overwrites the vector x. */

    int j, j0, j1, j2, j3;
    double tmpr, tmpi;

    for(j=0; j<n; j++)
        if (j<sigma[j]){ // swap j and sigma[j]
            j0= 2*j;                  j1= j0+1;
            j2= 2*sigma[j];           j3= j2+1;
            tmpr= x[j0];              tmpi= x[j1];
            x[j0]= x[j2];             x[j1]= x[j3];
            x[j2]= tmpr;              x[j3]= tmpi;
}}
```



Initialisation of bit reversal ρ_n, $n = 2^m \geq 2$

```
void bitrev_init(int n, int *rho){
  int j;
  unsigned int n1=n, rem, val, k, lastbit, one=1;

  for(j=0; j<n; j++){
     rem= j; // j= (b(m-1),...,b1,b0) in binary
     val= 0;
     for (k=1; k<n1; k <<= 1){
        lastbit= rem & one;
        // lastbit = b(i) with i = log2(k)
        rem >>= 1; // rem = (b(m-1),...,b(i+1))
        val <<= 1;
        val |= lastbit; // val = (b0,...,b(i))
     }
     rho[j]= (int)val;
}}
```

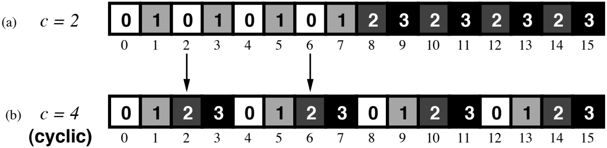
Assertions inside a loop (loop invariants) are powerful!



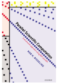
Bit operations

- Rule: use bit operations only sparingly in scientific computation.
- Reason: they obfuscate code, and good compilers often make them unnecessary.
- Here we encounter an exception: the cost of the bit reversal is of the same order $\mathcal{O}(n \log_2 n)$ as that of the FFT itself, so the bit reversal is important, and we need access to the bits anyway.

Redistribution

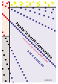


- We redistribute the vector **x** from group-cyclic distribution with cycle c_0 to cycle c_1, where $c_0 | c_1$ (and hence $c_0 \leq c_1$).
- Optimisation: vector components are sent in packets, not individually.
- BSP model: no difference in cost.
- BSPlib implementation: using packets is more efficient, and gives optimistic g-values.



Regular parallel algorithms

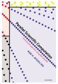
- The communication pattern of a regular parallel algorithm can be predicted exactly and each processor can determine exactly where every communicated data element goes.
- For a regular algorithm, it is always possible for the user to combine data for the same destination in a block, or packet, and communicate the block using 1 put operation.
- This requires packing at the source processor and unpacking at the destination processor.



Anything you can do, I can do better

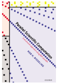
Anything you can send
I can send faster.
I can send anything
Faster than you.

- Song from the musical Annie Get Your Gun, Irving Berlin, 1946.
- The BSP system packs data, but for regular algorithms the user can do better, saving the sending of header information that identifies the data.
- This is worthwhile if the communication pattern involves sending many single data words, as happens in the FFT, or many very small data quantities.
- Not everything you can do, you should do.



How to pack

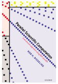
- Leave this to someone else. Good packers in theory make bad packers in practice.
- If you can leave it up to the BSP system, that's OK too.
- Main question: which data move to the same processor?
- Consider x_j and $x_{j'}$ residing on the same processor in the old distribution with cycle c_0. They are in the same block of size $\frac{nc_0}{p}$ handled by a group of c_0 processors.
- Each block of the old distribution fits entirely in a block of the new distribution, because $c_0 | c_1$.
- Thus, x_j and $x_{j'}$ will automatically be in the same new block of size $\frac{nc_1}{p}$ handled by a group of c_1 processors.



When are x_j and $x_{j'}$ on the same processor?

- Write
$$j = j_2 \frac{c_0 n}{p} + j_1 c_0 + j_0.$$
Because j_2 and j_0 depend only on the processor number, which is the same for j and j', we can write
$$j' = j_2 \frac{c_0 n}{p} + j_1' c_0 + j_0.$$
- In the new distribution:

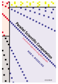
$$\begin{aligned}
& x_j \text{ and } x_{j'} \text{ are on the same processor} \\
\Longleftrightarrow \quad & j \equiv j' \pmod{c_1} \\
\Longleftrightarrow \quad & j_1 c_0 \equiv j_1' c_0 \pmod{c_1} \\
\Longleftrightarrow \quad & j_1 \equiv j_1' \pmod{\frac{c_1}{c_0}}.
\end{aligned}$$



Putting one packet

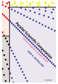
$$j = j_2 \frac{c_0 n}{p} + j_1 c_0 + j_0$$

- The local index of vector component x_j on its processor is $\mathtt{j} = j_1$.
- x_j and $x_{j'}$ on the same processor in the new distribution $\iff j_1 \equiv j_1' \pmod{\frac{c_1}{c_0}} \iff \mathtt{j} \equiv \mathtt{j}' \pmod{\frac{c_1}{c_0}}$.
- Thus, we can pack components with local indices $\mathtt{j}, \mathtt{j} + \frac{c_1}{c_0}, \mathtt{j} + \frac{2c_1}{c_0}, \ldots,$ into a temporary array and then put all of these components together into the destination processor as one packet.
- We define $\mathtt{ratio} = c_1/c_0$, the stride for packing data.



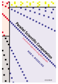
How not to unpack

- Unpacking is moving data from the location they were put into, to their final location on the same processor.
- If x_j and $x_{j'}$ are two adjacent components in a packet, with local indices at the source satisfying $\mathtt{j}' = \mathtt{j} + \mathtt{ratio}$, then the global indices satisfy $j' = j + \frac{c_1}{c_0} c_0 = j + c_1$.
- Thus, the local indices at the destination in the group-cyclic distribution with cycle c_1 satisfy $\mathtt{j}' = \mathtt{j} + 1$.
- We are lucky: if we put the first component x_j of the packet directly into its final location, and the next component of the packet into the next location, and so on, then all components of the packet immediately reach their final destination.
- Hence, we do not have to unpack!



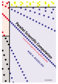
Redistribution (simplified)

```
void bspredistr(double *x, int n, int p, int s,
                int c0, int c1){ ...
   np= n/p;                 ratio= c1/c0;
   size= MAX(np/ratio,1);   npackets= np/size;
   j0= s%c0;                j2= s/c0; ...
   for(j=0; j<npackets; j++){
      jglob= j2*c0*np + j*c0 + j0;
      destproc=  (jglob/(c1*np))*c1 + jglob%c1;
      destindex= (jglob%(c1*np))/c1;
      for(r=0; r<size; r++){
         tmp[2*r]=   x[2*(j+r*ratio)];
         tmp[2*r+1]= x[2*(j+r*ratio)+1];
      }
      bsp_put(destproc,tmp,x,destindex*2*SZDBL,
              size*2*SZDBL);
   }
   bsp_sync(); }
```



Main function bspfft (forward)

```
void bspfft(double *x, int n, int p, int s,
            double *w0, double *w, double *tw,
            int *rho_np, int *rho_p){
   ...
   np= n/p; k1= k1_init(n,p); rev= TRUE;
   permute(x,np,rho_np); // local bit reversal
   for(r=0; r<np/k1; r++)
      ufft(&x[2*r*k1],k1,1,w0); // use any sequential fft
   c0= 1; ntw= 0;
   for (c=k1; c<=p; c *=np){
      bspredistr(x,n,p,s,c0,c,rev,rho_p);
      rev= FALSE;
      twiddle(x,np,1,&tw[2*ntw*np]);
      ufft(x,np,1,w); // use any sequential fft
      c0= c; ntw++;
}}
```



Summary

- We have optimised the communication in the only communication function the parallel FFT program has, the redistribution. This function is crucial for the parallel performance.
- The optimisation is done by packing data, which is always possible for regular algorithms with a predictable communication pattern.
- Where possible, we have moved computations to initialisation functions, e.g. for the table of weights.
- Because all communication is isolated in one function, the program can easily be ported to another communication library such as MPI.