# Program bspfft

## Section 3.6 of Parallel Scientific Computation, 2nd edition

Rob H. Bisseling

Utrecht University

# Sequential unordered FFT: specification

```
void ufft(double complex *x, long n, bool forward,
          double complex *w){

  /* This sequential function computes the unordered
     discrete Fourier transform of a complex vector x
     of length n, where n=2^m, m >= 0.
     The output overwrites x.

     If forward, then the forward unordered DFT is
     computed, and otherwise the backward unordered DFT.

     w is a table of complex weights of length n-1,
     which must have been suitably initialized before
     calling this function.
  */
```

# Sequential unordered FFT: body

```
void ufft(double complex *x, long n, bool forward,
          double complex *w){

    long start= 0;
    for (long k=2; k<=n; k *=2){
        butterfly_stage(x,n,k,forward,&w[start]);
        start += k/2;
    }
}
```

- All butterflies of stage $k$ use the same set of $k/2$ weights stored in array `w`.

# Butterflies of stage $k$

```
void butterfly_stage(double complex *x, long n, long k,
                     bool forward, double complex *w){

    for (long r=0; r<n/k; r++){
        for (long j=0; j<k/2; j++){
            double complex weight;
            if (forward) {
                weight= w[j];
            } else {
                weight= conj(w[j]);
            }

            double complex tau= weight * x[r*k+j+k/2];
            x[r*k+j+k/2]= x[r*k+j] - tau;
            x[r*k+j] += tau;
        }
    }
}
```

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

```c
void permute(double complex *x, long n, long *sigma){

    /* This in-place sequential function permutes
       a complex vector x of length n >= 1
       by the permutation sigma,
           y[j] = x[sigma[j]], 0 <= j < n.
       The output overwrites the vector x. */

    for (long j=0; j<n; j++){
        long sigmaj = sigma[j];
        if (j<sigmaj){
            /* swap components j and sigma[j] */
            double complex tmp= x[j];
            x[j]= x[sigmaj];
            x[sigmaj]= tmp;
        }
    }
}
```

# Initialization of bit reversal $\rho_n$, $n = 2^m \geq 2$

We compute $\rho_1, \rho_2, \rho_4, \ldots, \rho_n$ by using

$$\rho_n(j) = \begin{cases} 2\rho_{n/2}(j) & \text{for } 0 \leq j < n/2, \\ 2\rho_{n/2}(j - n/2) + 1 & \text{for } n/2 \leq j < n. \end{cases}$$

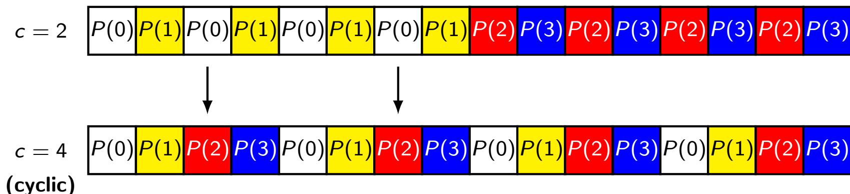
```
void bitrev_init(long n, long *rho){

    rho[0]= 0; // rho = rho_1

    for (long k=2; k<=n; k *=2){

        /* Compute rho = rho_k */
        for (long j=0; j<k/2; j++){
            rho[j] *= 2;
            rho[j+k/2]= rho[j] + 1;
        }
    }
}
```

# 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 $c_0 \leq c_1$).
- Optimization: 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.

# Packing useful stuff

# 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 travel to the same destination processor?

# Which data travel together?

- 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 will $x_j$ and $x_{j'}$ be on the same processor?

- In the old distribution, 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, $x_j$ and $x_{j'}$ are on the same processor if

$$\begin{aligned}
& j \equiv j' && \pmod{c_1} \\
\iff\quad & j_1 c_0 \equiv j_1' c_0 && \pmod{c_1} \\
\iff\quad & j_1 \equiv j_1' && \pmod{\frac{c_1}{c_0}}
\end{aligned}$$

# 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 $x_{j'}$ are 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 $\mathtt{j}, \mathtt{j} + \frac{c_1}{c_0}, \mathtt{j} + 2\frac{c_1}{c_0}, \ldots$, into a temporary array and then put all of these components together into the destination processor as one packet.
- We define $\mathtt{ratio} = \frac{c_1}{c_0}$, the stride for packing data.

# How not to unpack

- If $x_j$ and $x_{j'}$ are two adjacent components in a packet, with local indices at the source satisfying $\mathtt{j}' = \mathtt{j} + \frac{c_1}{c_0}$, 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.
- This means we do not have to unpack!

# Redistribution from $c_0$ to $c$

```
void bspredistr(double complex *x, long n, long c0,
                long c, long *rho_p){

    ...
    long j0= s%c0;
    long j2= s/c0;
    long ratio= c/c0;
    long np= n/p;

    long size= (np >= ratio ? np/ratio : 1 );
    long npackets= np/size;
    double complex *tmp= vecallocc(size);
    ...
```

# Redistribution from $c_0$ to $c$ (cont'd)

```
for (long j=0; j<npackets; j++){
    long jglob= j2*c0*np + j*c0 + j0;
    long destproc=  (jglob/(c*np))*c + jglob%c;
    long destindex= (jglob%(c*np))/c;

    for (long r=0; r<size; r++)
        tmp[r]=    x[j+r*ratio];

    bsp_put(destproc,tmp,x,
            destindex*sizeof(double complex),
            size*sizeof(double complex));
}
bsp_sync();
...
```

# Main function bspfft

```
void bspfft(double complex *x, long n, bool forward,
            double complex *w,
            long *rho_np, long *rho_p){

    long p= bsp_nprocs();
    long np= n/p;
    long c= 1;
    bool rev= true;

    /* Perform a local ordered FFT of length n/p.
       This part can be replaced easily by your
       favourite sequential FFT */
    permute(x,np,rho_np);
    ufft(x,np,forward,w);

    ...
```

# Main function bspfft (cont'd)

```
long k= 2*np;
long start= np-1; // start of current weights in w

while (c < p){
    long c0= c;
    c= ( np*c <= p ? np*c : p);
    bspredistr(x,n,c0,c,rev,rho_p);
    rev= false;

    while (k <= np*c){
        butterfly_stage(x,np,k/c,forward,&w[start]);
        start += k/(2*c);
        k *= 2;
    }
}
```

# Summary

- We have optimized the communication in the only communication function of the parallel FFT, the redistribution.
- We did this by packing data, which is always possible for regular algorithms with a predictable communication pattern.
- Where possible, we have moved computations to initialization functions, e.g. for the table of weights in a dry run of the algorithm, and also for the bit reversal permutation.
- Even higher performance can be attained by replacing the start of the algorithm by highly optimized sequential code such as FFTW or Spiral.