

The Bulk Synchronous Parallel Model and its Application

Rob Bisseling

Rob.Bisseling@math.uu.nl

<http://www.math.uu.nl/people/bisseling>

Mathematical Institute

Universiteit Utrecht

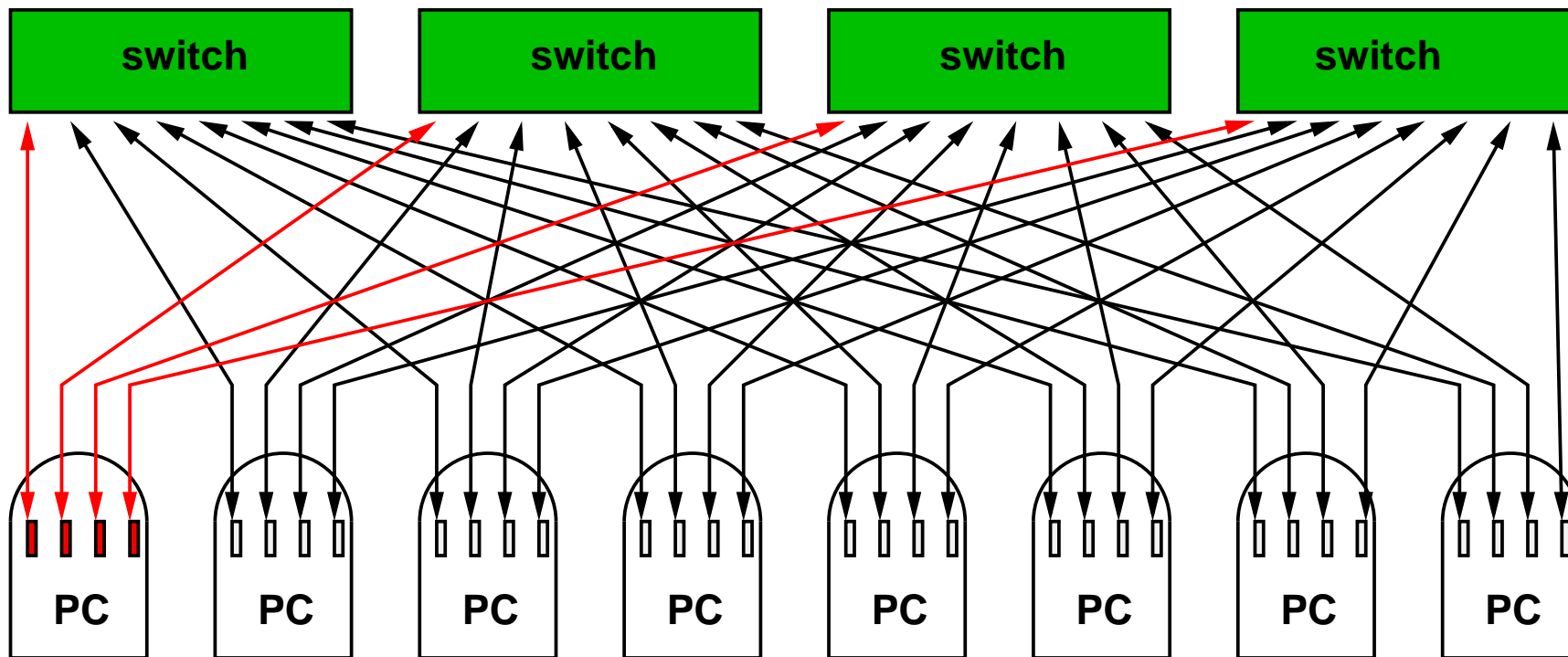


This class:

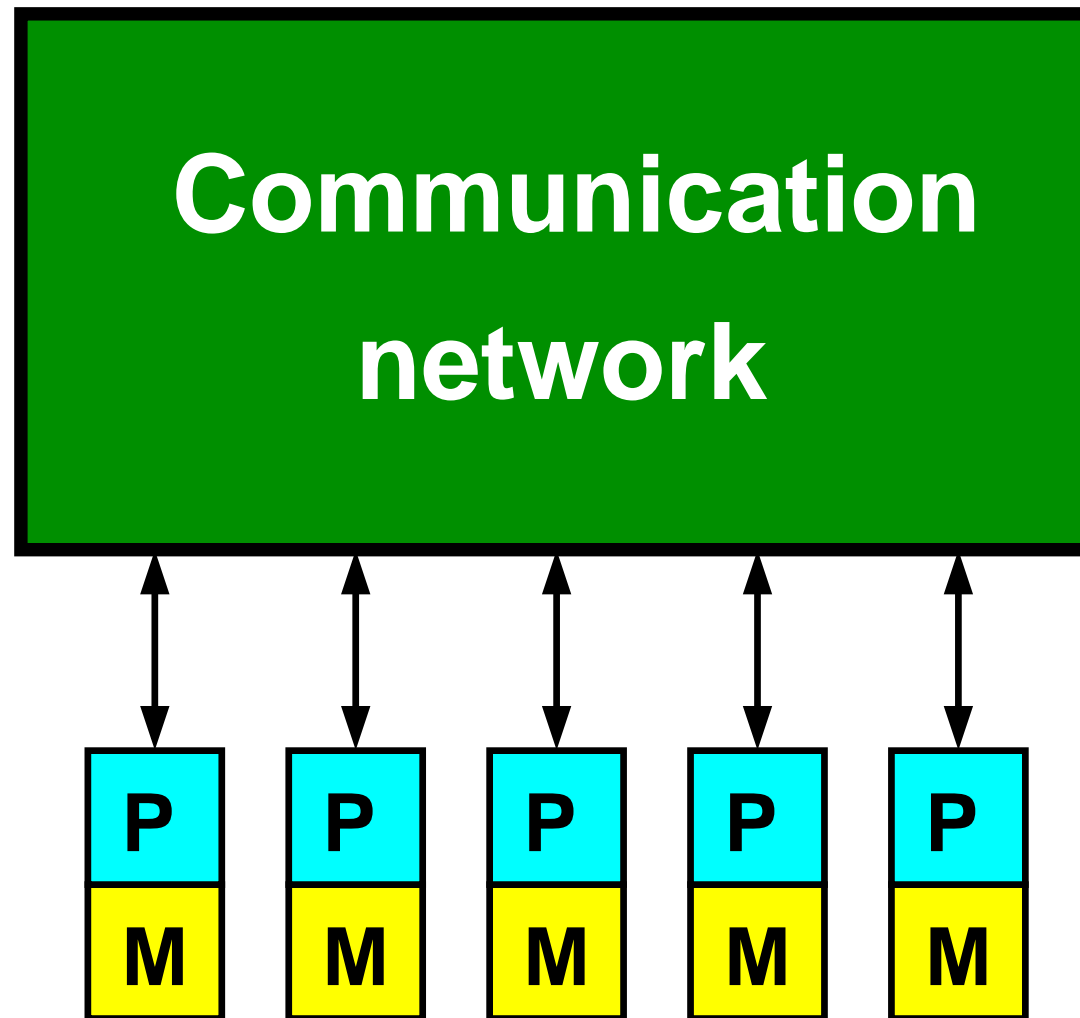
- Introduction bulk synchronous parallel computing:
 - Bulk synchronous parallel model
 - One-sided communications, MPI-2
 - Inner product computation
- Fast Fourier Transform:
 - Sequential 1D FFT
 - Data distributions: block, cyclic
 - Parallel 1D FFT
 - Quantum molecular dynamics
 - Multidimensional FFT
- Conclusions



Part 1. Introduction. Parallel computer: PC cluster



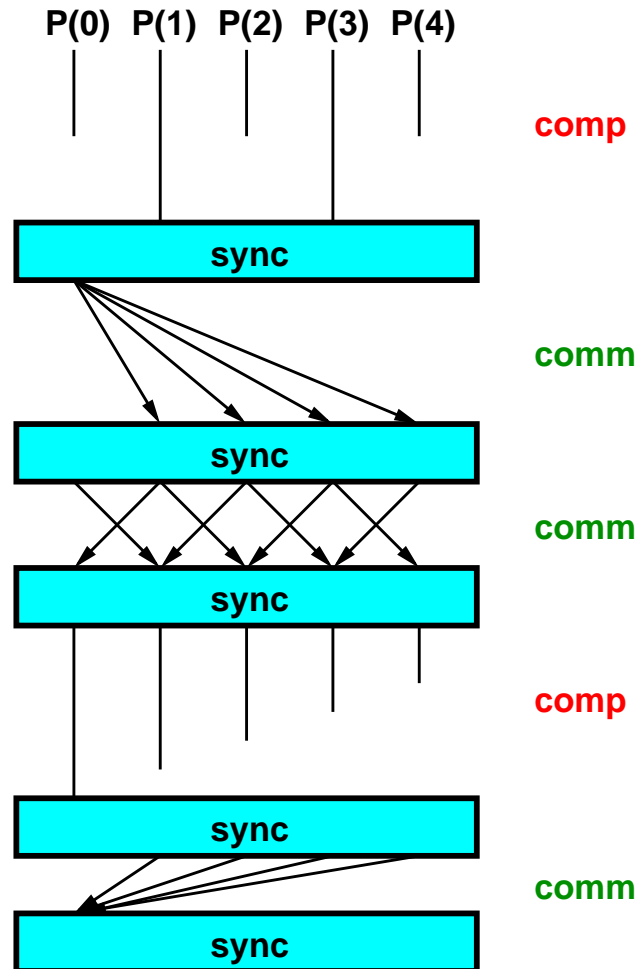
Parallel computer: abstract model



Bulk synchronous parallel (BSP) computer

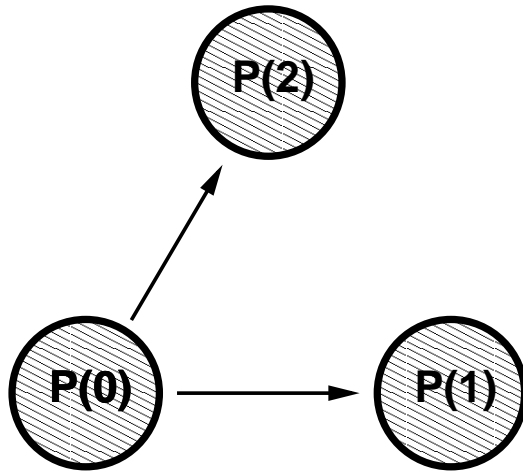


Parallel algorithm: supersteps

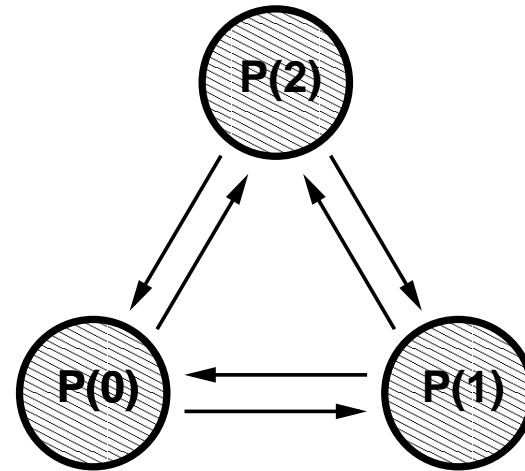


Time of communication superstep

2-relations:



(a)

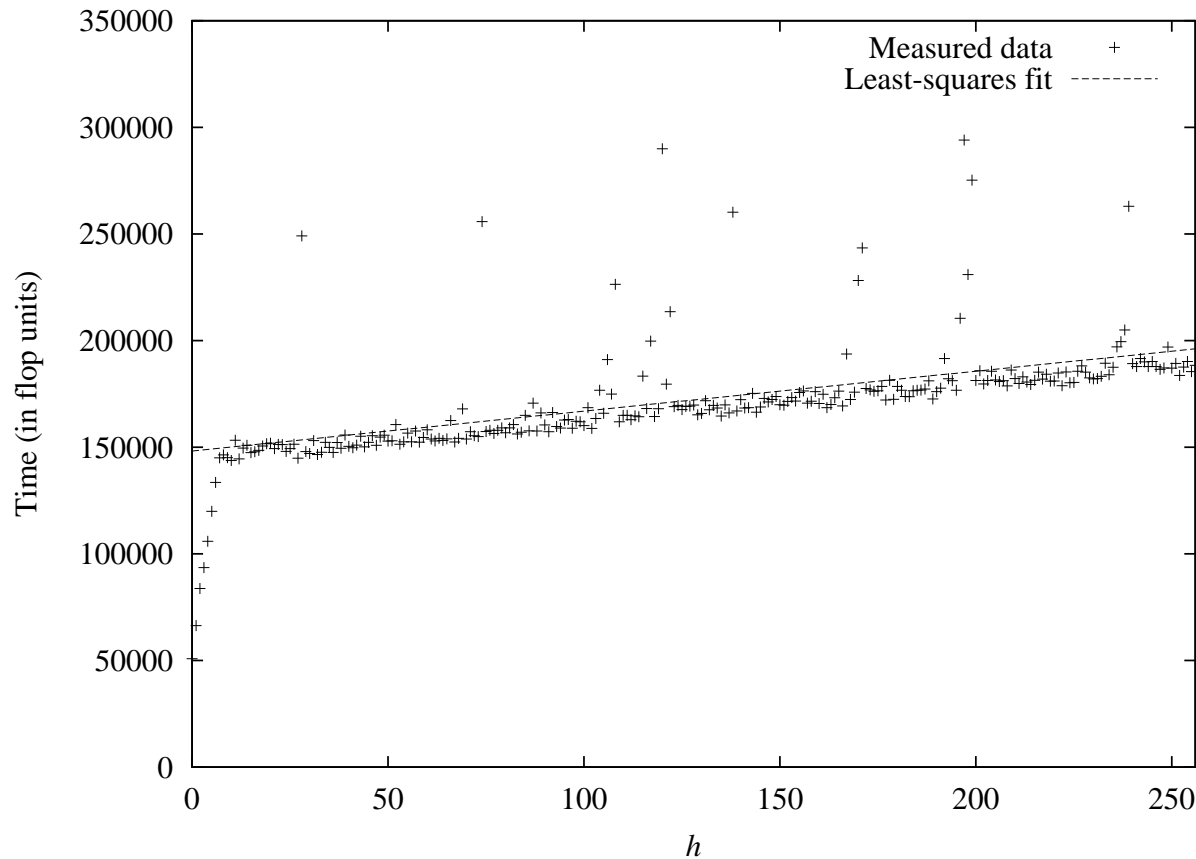


(b)

- An *h*-relation is a communication superstep in which every processor sends and receives at most *h* data words: $h = \max\{h_{\text{send}}, h_{\text{recv}}\}$
- $T(h) = hg + l$, where *g* is the time per data word and *l* the global synchronisation time



Time of an h -relation on an 8-processor IBM SP



$r = 212$ Mflop/s, $p = 8$, $g = 187$ flop ($0.88\mu\text{s}$),

$l = 148212$ flop ($698\mu\text{s}$)



BSP parameters of an SGI Origin 3800

p	g	l	$T_{\text{comm}}(0)$
1	99	55	378
2	75	5118	1414
4	99	12743	2098
8	126	32742	4947
16	122	93488	15766

$$r = 285 \text{ Mflop/s}$$

$T_{\text{comm}}(0)$ is the time of a 0-relation



MPI and BSPlib

- **MPI-1** (1994): two-sided communication = message passing

```
if (s==2) MPI_Send(x, 5, MPI_DOUBLE, 3, 0,
                  MPI_COMM_WORLD);
if (s==3) MPI_Recv(y, 5, MPI_DOUBLE, 2, 0,
                  MPI_COMM_WORLD, &status);
```

- **BSPlib** (1998): one-sided communication (buffered)

```
bsp_put(3, x, y, 0, 5*BSP_DOUBLE);
```

Later followed by a `bsp_sync`.

- **MPI-2** (1998): one-sided communication (unbuffered)

```
MPI_Put(x, 5, MPI_DOUBLE, 3, 0,
        5, MPI_DOUBLE, y_win);
```



One-sided vs two-sided communications

- One-sided communications decouple communication from synchronisation.
- They are simpler and more efficient.
- BSPlib, MPI-2 implementations now available



Latin square reordering

Supersteps allow reordering/combining of communication.
Latin squares are used to avoid congestion:

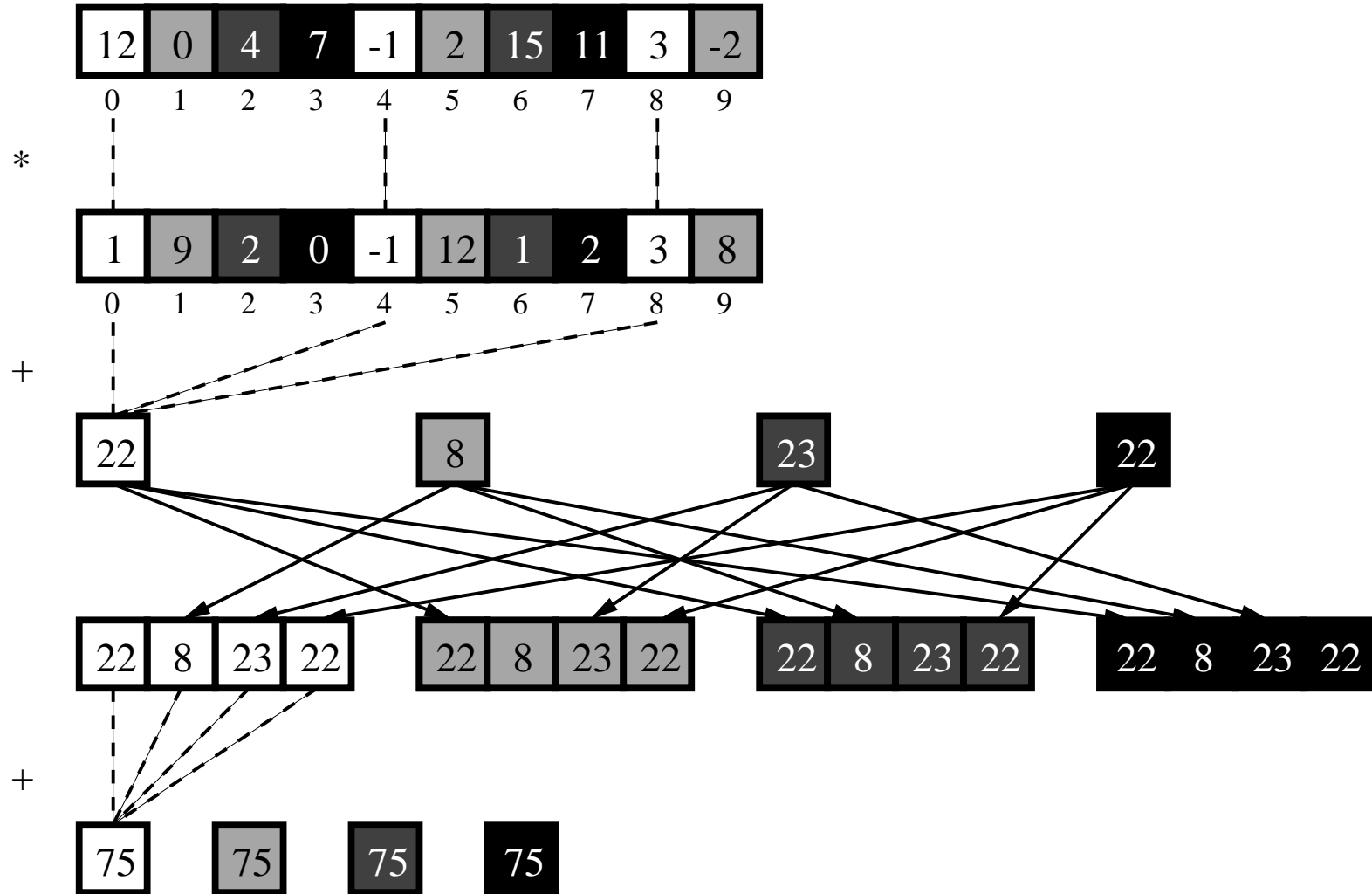
$$R = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \\ 2 & 3 & 0 & 1 \\ 3 & 0 & 1 & 2 \end{bmatrix} .$$

The communication of the superstep is done in p rounds.

In round j , processor $P(i)$ sends all data destined for $P(r_{ij})$.



Inner product computation of vectors x and y



4 processors with cyclic distribution
Universiteit Utrecht

Inner product algorithm for processor $P(s)$

input: \mathbf{x}, \mathbf{y} : vector of length n , $\text{distr}(\mathbf{x}) = \text{distr}(\mathbf{y}) = \phi$,
with $\phi(i) = i \bmod p$, for $0 \leq i < n$.

output: $\alpha = \mathbf{x}^T \mathbf{y}$.

(0) $\alpha_s := 0$;
for $i := s$ **to** $n - 1$ **step** p **do**
 $\alpha_s := \alpha_s + x_i y_i$;

(1) **for** $t := 0$ **to** $p - 1$ **do**
 put α_s in $P(t)$;

(2) $\alpha := 0$;
for $t := 0$ **to** $p - 1$ **do**
 $\alpha := \alpha + \alpha_t$;



Universiteit Utrecht $T_{\text{inprod}} = 2 \left\lceil \frac{n}{p} \right\rceil + p + (p - 1)g + 3l.$

Inner product program in BSPLib

```
double bspip(int p, int s, int n,  
            double *x, double *y) {  
    double inprod=0.0, *Inprod, alpha=0.0;  
    int i, t;  
    Inprod= vecallocd(p);  
    bsp_push_reg(Inprod,p*SZDBL);  
    bsp_sync();  
    for (i=0; i<nloc(p,s,n); i++)  
        inprod += x[i]*y[i];  
    for (t=0; t<p; t++)  
        bsp_put(t,&inprod,Inprod,s*SZDBL,SZDBL);  
    bsp_sync();  
    for (t=0; t<p; t++)  
        alpha += Inprod[t];  
    bsp_pop_reg(Inprod); vecfreed(Inprod);  
    return alpha;  
}
```



Inner product program in MPI-1

```
double mpiip(int p, int s, int n,  
             double *x, double *y) {  
    double inprod=0.0, alpha;  
    int i;  
  
    for (i=0; i<nloc(p,s,n); i++)  
        inprod += x[i]*y[i];  
  
    MPI_Allreduce(&inprod, &alpha, 1, MPI_DOUBLE,  
                 MPI_SUM, MPI_COMM_WORLD);  
  
    return alpha; }  
}
```



Complete BSPlib

Class	Primitive	Meaning
SPMD	<code>bsp_begin</code>	Start of parallel part
	<code>bsp_end</code>	End of parallel part
	<code>bsp_init</code>	Initialize parallel part
	<code>bsp_nprocs</code>	Number of processors
	<code>bsp_pid</code>	My processor number
	<code>bsp_time</code>	My elapsed time
	<code>bsp_abort</code>	One processor stops all
	<code>bsp_sync</code>	Synchronize globally



Complete BSPlib cont'd

Class	Primitive	Meaning
RDMA	<code>bsp_push_reg</code>	Register variable
	<code>bsp_pop_reg</code>	Deregister variable
	<code>bsp_put</code>	Write into remote memory
	<code>bsp_hpput</code>	Unbuffered put
	<code>bsp_get</code>	Read from remote memory
	<code>bsp_hpget</code>	Unbuffered get

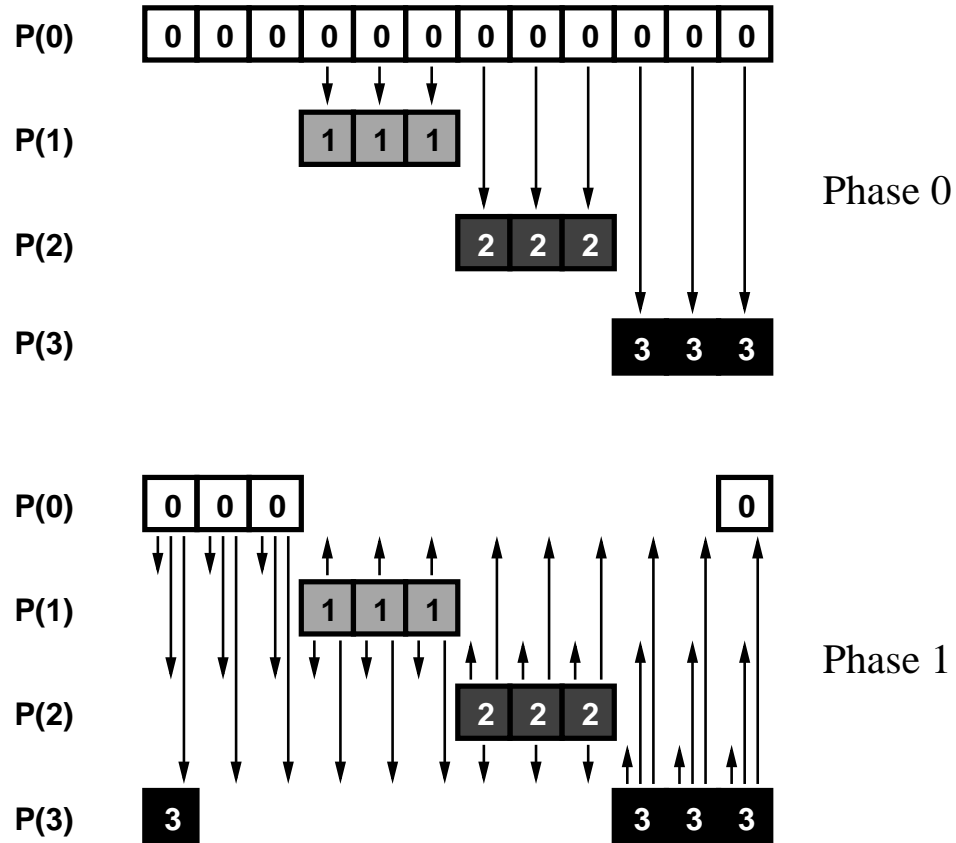


Complete BSPlib cont'd

Class	Primitive	Meaning
BSMP	<code>bsp_set_tagsize</code>	Set new tag size
	<code>bsp_send</code>	Send a message
	<code>bsp_qsize</code>	Number of received messages
	<code>bsp_get_tag</code>	Get tag of received message
	<code>bsp_move</code>	Store payload locally
	<code>bsp_hpmove</code>	Store by setting pointers



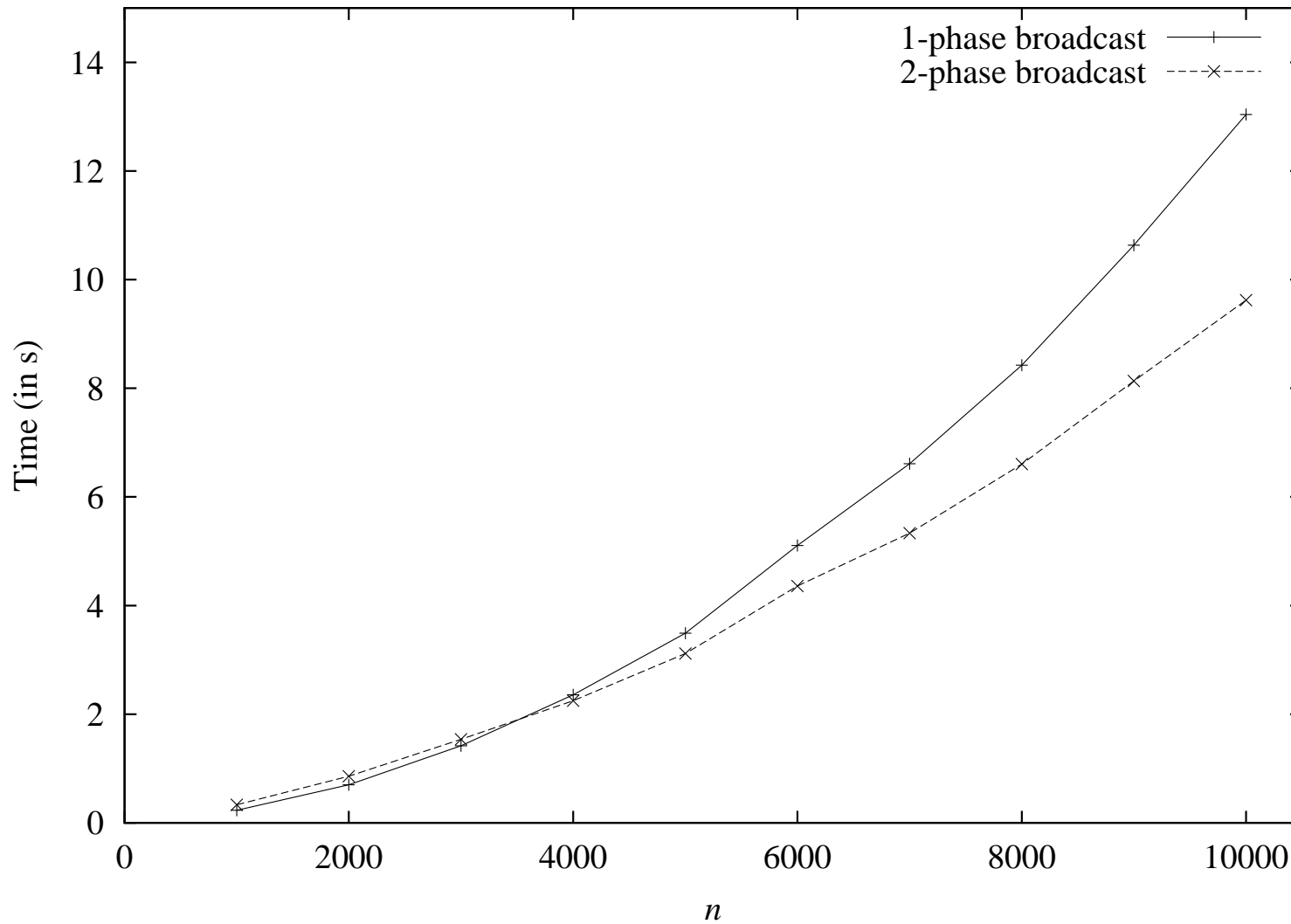
Two-phase broadcast



$$T_{\text{broadcast}} = \left(n + (p - 2) \left\lceil \frac{n}{p} \right\rceil \right) g + 2l \approx 2ng + 2l.$$



Dense LU decomposition on a 64-processor Cray T3E



Part 2. Fast Fourier Transform

The **discrete Fourier transform** (DFT) of a vector $\mathbf{x} = (x_0, \dots, x_{n-1})^T$ is the vector $\mathbf{y} = (y_0, \dots, y_{n-1})^T$ with

$$y_k = \sum_{j=0}^{n-1} x_j e^{-2\pi i j k / n} = \sum_{j=0}^{n-1} x_j \omega_n^{jk}, \text{ for } 0 \leq k < n.$$

Here, $\omega_n = e^{-2\pi i / n}$.



Basic idea of Fast Fourier Transform

$$y_k = \sum_{j=0}^{n-1} x_j \omega_n^{jk} = \sum_{j=0}^{n/2-1} x_{2j} \omega_n^{2jk} + \sum_{j=0}^{n/2-1} x_{2j+1} \omega_n^{(2j+1)k}.$$

Using $\omega_n^2 = \omega_{n/2}$ gives

$$y_k = \sum_{j=0}^{n/2-1} x_{2j} \omega_{n/2}^{jk} + \omega_n^k \sum_{j=0}^{n/2-1} x_{2j+1} \omega_{n/2}^{jk}$$

$$y_{k+n/2} = \sum_{j=0}^{n/2-1} x_{2j} \omega_{n/2}^{jk} - \omega_n^k \sum_{j=0}^{n/2-1} x_{2j+1} \omega_{n/2}^{jk}$$



Time complexity of Fast Fourier Transform

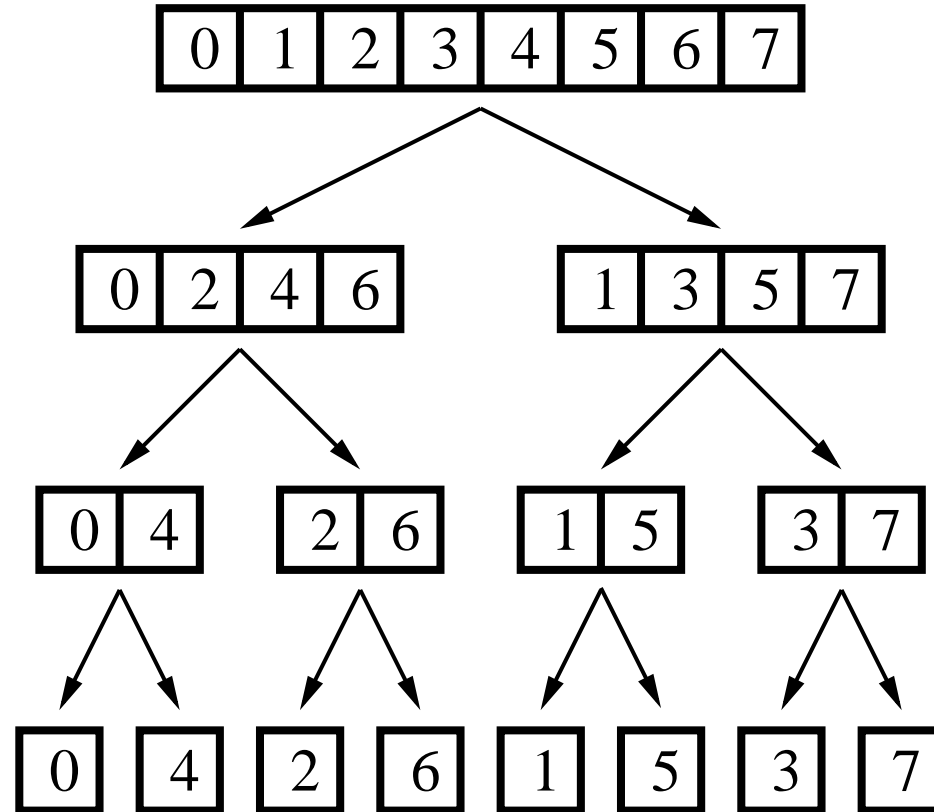
Use two $\text{FFT}(n/2)$ operations to compute $\text{FFT}(n)$:

$$T(n) = 2T\left(\frac{n}{2}\right) + 5n = 4T\left(\frac{n}{4}\right) + 5n + 5n = \dots = 5n \log_2 n$$

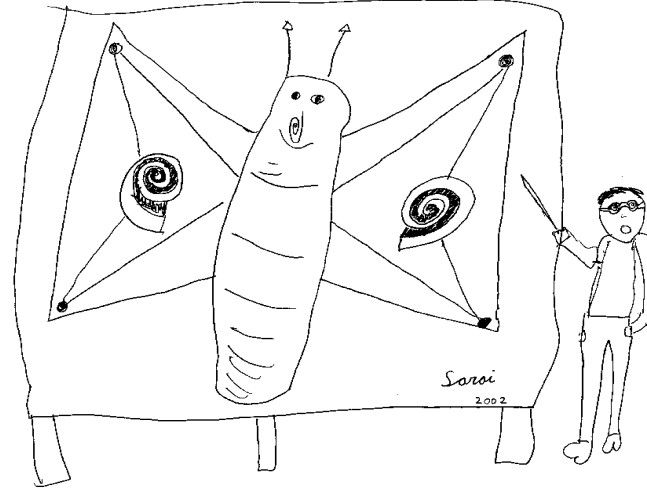
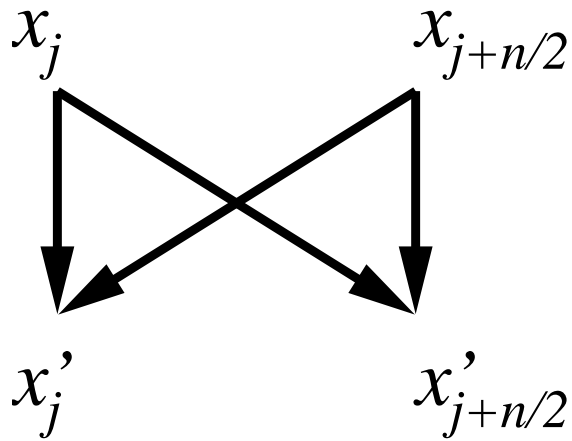
Much faster than $8n^2$ time for direct computation of DFT.



Recursive computation of FFT



Butterfly operations of FFT



$$x'_j := x_j + \omega_n^j x_{j+n/2}$$

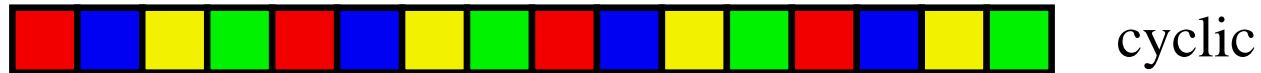
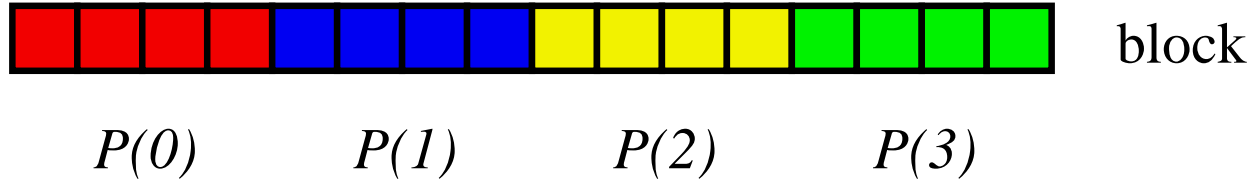
$$x'_{j+n/2} := x_j - \omega_n^j x_{j+n/2}$$

for $0 \leq j < n/2$, where $\omega_n = e^{-2\pi i/n}$.

Distance = $n/2$.



Data distributions for 1D array

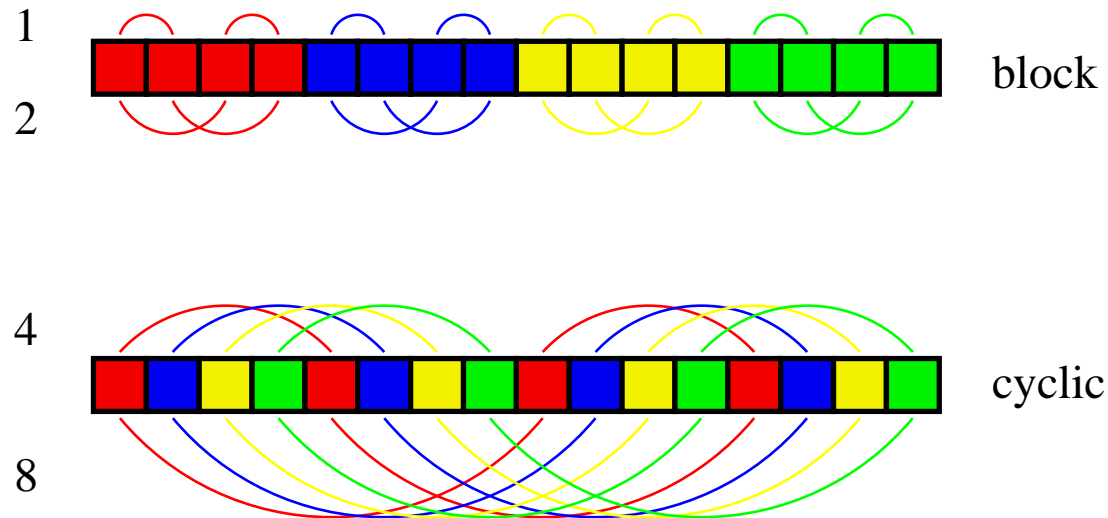


$p = \#$ processors = 4
array size = 16



Data distributions for butterflies

butterfly distance



$$p = \# \text{ processors} = 4$$
$$n = \text{array size} = 16$$

Start with block distribution: $dist < \frac{n}{p}$

Finish with cyclic distribution: $dist \geq p$

Only two distributions needed if: $p \leq \frac{n}{p}$, i.e., $p \leq \sqrt{n}$



Bit reversal of FFT

j	$(b_2b_1b_0)_2$	$(b_0b_1b_2)_2$	$\rho_8(j)$
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

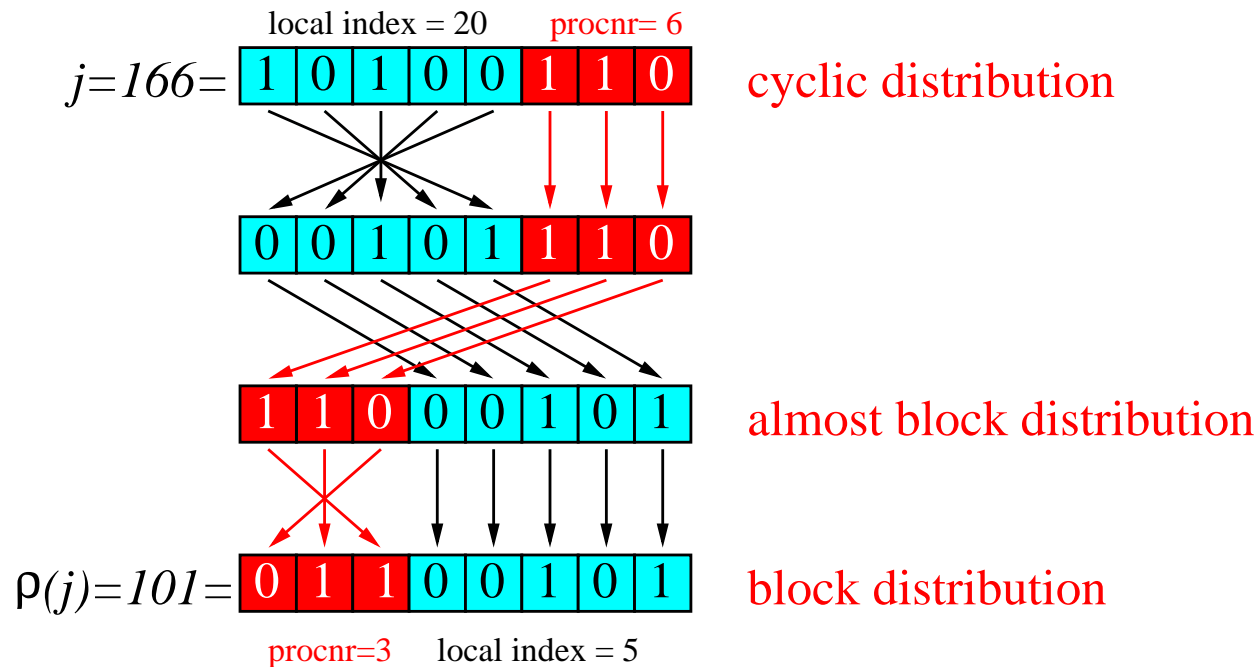
$$j = (b_{m-1} \cdots b_1 b_0)_2 = \sum_{k=0}^{m-1} b_k 2^k,$$

where $n = 2^m$ and $b_k \in \{0, 1\}$.



Parallel bit reversal

$$p = 8, n = 256$$



First: local bit reversal

Then: **do as if in block distribution**, good for small butterflies

Much later: bit reversal of processor numbering

No extra communication!



Parallel FFT algorithm for processor $P(s)$

bitrev($x(s:p:n-1)$, n/p);

$k := 2$; $c := 1$;

while $c \leq p$ **do**

(0)

$j_0 := s \bmod c$; $j_2 := s \operatorname{div} c$;

while $k \leq \frac{n}{p}c$ **do**

$nblocks := \frac{nc}{kp}$;

for $r := j_2 \cdot nblocks$ **to** $(j_2 + 1) \cdot nblocks - 1$ **do**

for $j := j_0$ **to** $\frac{k}{2} - 1$ **step** c **do**

$\tau := \omega_k^j x_{rk+j+k/2}$;

$x_{rk+j+k/2} := x_{rk+j} - \tau$;

$x_{rk+j} := x_{rk+j} + \tau$;

$k := 2k$;

$c := pc$;

(1)

if $c \leq p$ **then** redistr(\mathbf{x} , n , p);



Redistribution

function: `redistr(x, n, p);`

$s' := \rho_p(s);$

for $j := s' \frac{n}{p}$ **to** $(s' + 1) \frac{n}{p} - 1$ **do**
 put x_j **in** $P(j \bmod p);$



Total BSP cost

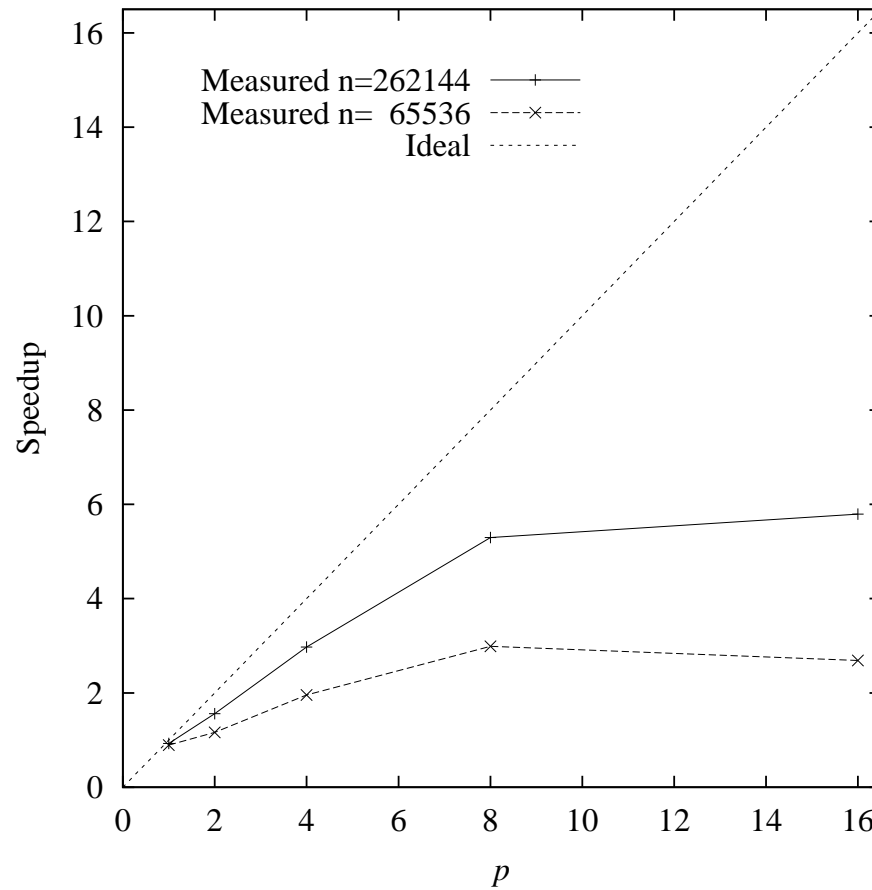
$$T_{\text{FFT}, 1 < p \leq \sqrt{n}} = \frac{5n \log_2 n}{p} + 2\frac{n}{p}g + l$$

For $\sqrt{n} < p < n/2$, a more general algorithm exists, and a freely available implementation:

[BSPedupack](#), [MPIedupack](#).



Speedup of parallel FFT on an SGI Origin 3800



Using BSPlib.



Breakdown of parallel FFT timings

p	T_{Comp}	T_{Comm}	T_{Sync}	T_{pred}	T_{meas} (BSP)	T_{meas} (MPI)
1	82.78	0.00	0.00	82.78	167.4	189
2	41.39	68.99	0.05	110.43	99.4	107
4	20.70	45.53	0.13	66.36	52.2	50
8	10.35	28.97	0.35	39.67	29.3	26
16	5.17	14.03	0.98	20.18	26.8	19

Predictions are for BSP.

MPI uses all-to-all collective communication



Application QMD

- **Quantum molecular dynamics** solves the time-dependent Schrödinger equation (Kosloff & Kosloff, 1983)
- Main task: compute Hamiltonian operation $\mathbf{H}\psi$
- Wavefunction ψ not limited to 3D.
Dimension = # degrees of freedom. Can you do 8D?
- $\mathbf{H} = \mathbf{T} + \mathbf{V}$. Potential energy operator \mathbf{V} is local, hence pointwise multiplication of data vector
Kinetic energy operator \mathbf{T} is local in Fourier domain, hence quick transform possible by FFT.
- Parallel: locality implies that **every data distribution** can be used, including cyclic (good for FFT!)

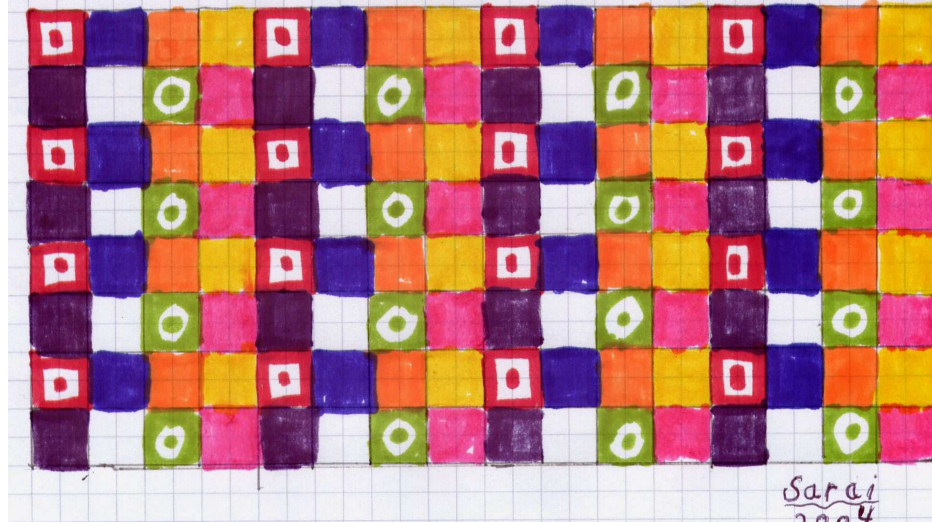


Parallel multidimensional QMD

- Classical approach: **avoid parallel 1D FFTs** by transposing data array and using sequential 1D FFTs.
- Our alternative approach: **use parallel 1D FFTs**, since data redistribution costs the same as transposition.



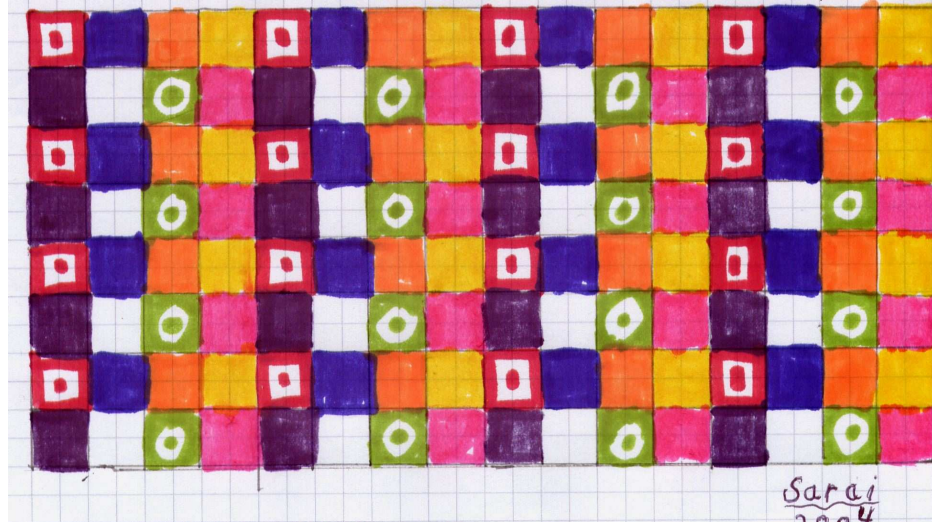
Example: Parallel 2D QMD



- $N_1 \times N_2$ grid, $N = N_1 N_2$. Use $p = p_1 \times p_2$ processors.
Here: $N_1 = 8$, $N_2 = 16$, $p_1 = 2$, $p_2 = 4$.
- Cyclic distribution: $\text{Proc}(i_1, i_2) = (i_1 \bmod p_1, i_2 \bmod p_2)$.



Rectangular grids



- Assume $N_1 \leq N_2$. Transposition works only if $p \leq N_1$.
Alternative approach works for all $p \leq N_1 N_2 / 4$.
For $p \leq N_2$, only one redistribution is needed
(using $p_1 = 1, p_2 = p$).
- Alternative approach is good for very rectangular grids.



Conclusions — 1

How to program in BSP style using MPI:

- Use collective communications
- Avoid send/receive pairs
- Use one-sided communications
- Think Bulk:
 - BSP cost model
 - separation of communication/synchronisation
 - system optimisation, not user optimisation



Conclusions — 2

- We have developed a powerful bulk synchronous parallel 1D FFT based on the cyclic data distribution that needs only one data redistribution.
- The parallel 1D FFT can be used as the workhorse in multidimensional FFTs.
- BSP is a **way of thinking**: simple, natural, and effective.
- BSP programs can be **portable and efficient**.

