MulticoreBSP for C

A high-performance library for shared-memory parallel programming

Albert-Jan Yzelman,
Rob H. Bisseling, D. Roose, and K. Meerbergen.

2nd of July 2013
at the 'International Symposium on High-level Parallel Programming and Applications', Paris 1-2 July 2013.

Introduction

A BSP computer $C = (p, r, g, l)$.

Primary assumption:

- the bottleneck of communication are the exit points and the entry points of communication.

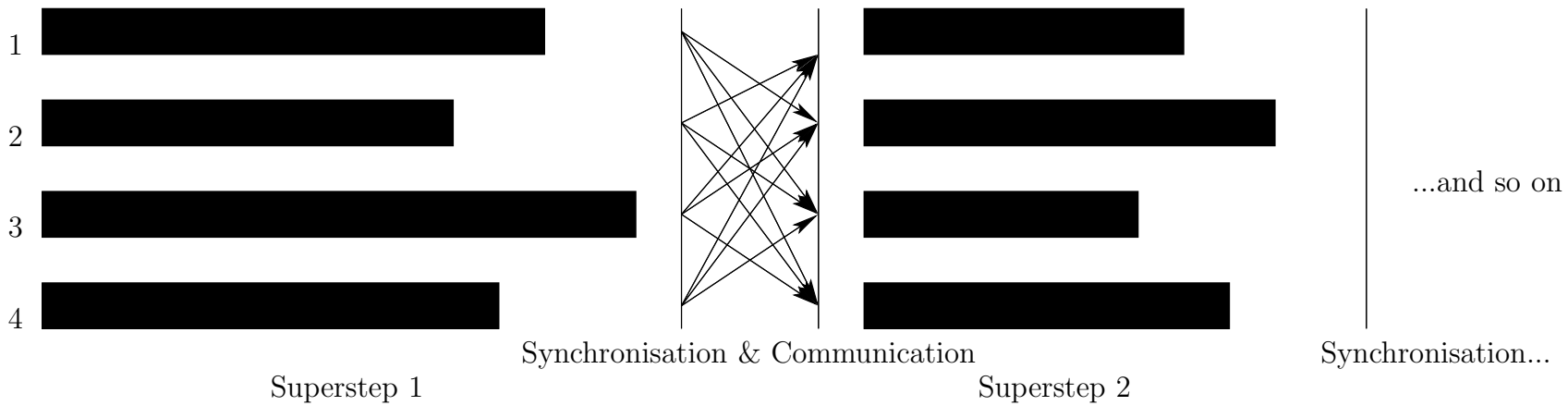
Parameters:

- A BSP computer has p processors,
- each processor runs at speed r.
- sending and receiving data during an all-to-all communication costs g,
- preparing the network for all-to-all communication costs l.

Introduction

For Bulk Synchronous Parallel algorithms:

- computations are grouped into **phases**,
- **no communication** during computation, but
- communication is allowed **in-between** computation phases.



Introduction

The time spent in computation during the ith superstep is

$$T_{\text{comp},i} = \max_s w_i^{(s)}/r.$$

The total cost of communication is

$$T_{\text{comm}} = \sum_{i=0}^{N-1} h_i g.$$

Adding up the computation and communication costs, and accounting for l gives us the **full BSP cost**:

$$T = \sum_{i=0}^{N-1} \max w_i^{(s)}/r + h_i g + l.$$

Goals

Why another BSP library?

We aim to show that:

- existing BSP software runs equally well on shared-memory systems as it does on distributed-memory,
- BSP can attain high performance on non-trivial applications, comparable the state-of-the-art.

Goals

Why another BSP library?

We aim to show that:

- existing BSP software runs equally well on shared-memory systems as it does on distributed-memory,
- BSP can attain high performance on non-trivial applications, comparable the state-of-the-art.

Goals

Why another BSP library?

We aim to show that:

- existing BSP software runs equally well on shared-memory systems as it does on distributed-memory,
- BSP can attain high performance on non-trivial applications, comparable the state-of-the-art.

Goals

Thus, MulticoreBSP for C:

- Is fully backwards-compatible with BSPlib (optionally),
- is based on BSPlib but with an updated interface,
- defines two new high-performance primitives.

Technologies employed:

- MulticoreBSP for C is written in ANSI C99, and
- depends on two standard extensions:
  1. POSIX Threads for shared-memory threading.
  2. POSIX realtime for high-resolution timings.

Goals

Thus, MulticoreBSP for C:

- Is fully backwards-compatible with BSPlib (optionally),
- is based on BSPlib but with an updated interface,
- defines two new high-performance primitives.

Technologies employed:

- MulticoreBSP for C is written in ANSI C99, and
- depends on two standard extensions:
  1. POSIX Threads for shared-memory threading.
  2. POSIX realtime for high-resolution timings.

Changes from BSPlib

Programming interface updates:

- `size_t` instead of `int` when appropriate;
- `unsigned` types whenever appropriate;

Standard updates:

- asymptotic running times of all BSP primitives;
- support for hierarchical execution (Multi-BSP);
- adds `bsp_direct_get` and `bsp_hpsend`.

Library additionally features thread affinity/pinning.

All 22 BSP primitives

SPMD:

- `bsp_init`: $\Theta(1)$
- `bsp_begin`: $\mathcal{O}(p)$
- `bsp_nprocs`: $\Theta(1)$
- `bsp_end`: $\mathcal{O}(l)$
- `bsp_pid`: $\Theta(1)$
- `bsp_sync`: $\Theta(l + g \cdot h_i)$
- `bsp_abort`: $\Theta(1)$
- `bsp_time`: $\Theta(1)$

High-performance:

- `bsp_hpput`: $\Theta(1)$
- `bsp_hpget`: $\Theta(1)$
- `bsp_hpsend`: $\Theta(1)$
- `bsp_hpmove`: $\Theta(1)$
- `bsp_direct_get`: $\Theta(size)$

BSMP:

- `bsp_send`: $\Theta(size)$
- `bsp_set_tagsize`: $\Theta(1)$
- `bsp_qsize`: $\mathcal{O}(messages)$
- `bsp_get_tag`: $\Theta(1)$
- `bsp_move`: $\Theta(size)$

DRMA:

- `bsp_push_reg`: $\Theta(1)$
- `bsp_pop_reg`: $\Theta(1)$
- `bsp_put`: $\Theta(size)$
- `bsp_get`: $\Theta(1)$

BSP 'DIRECT GET'

The 'direct get' is a **blocking** one-sided get instruction.

- bypasses the BSP model, but is consistent with `bsp_hpget`.

Its intended case is within supersteps that

- contain only BSP 'get' primitives,
- guarantee source data remains unchanged.

Replacing those primitives with calls to `bsp_direct_get` allows **merging** this superstep with its following one, thus

saving a synchronisation step.

A. N. Yzelman & Rob H. Bisseling, **An Object-Oriented Bulk Synchronous Parallel Library for Multicore Programming**, Concurrency and Computation: Practice and Experience 24(5), pp. 533-553 (2012).

BSP 'direct get'

The 'direct get' is a **blocking** one-sided get instruction.

- bypasses the BSP model, but is consistent with `bsp_hpget`.

Its intended case is within supersteps that

- contain only BSP 'get' primitives,
- guarantee source data remains unchanged.

Replacing those primitives with calls to `bsp_direct_get` allows **merging** this superstep with its following one, thus

saving a synchronisation step.

A. N. Yzelman & Rob H. Bisseling, **An Object-Oriented Bulk Synchronous Parallel Library for Multicore Programming**, Concurrency and Computation: Practice and Experience 24(5), pp. 533-553 (2012).

BSP 'DIRECT GET'

The 'direct get' is a **blocking** one-sided get instruction.

- bypasses the BSP model, but is consistent with `bsp_hpget`.

Its intended case is within supersteps that

- contain only BSP 'get' primitives,
- guarantee source data remains unchanged.

Replacing those primitives with calls to `bsp_direct_get` allows **merging** this superstep with its following one, thus

saving a synchronisation step.

A. N. Yzelman & Rob H. Bisseling, **An Object-Oriented Bulk Synchronous Parallel Library for Multicore Programming**, Concurrency and Computation: Practice and Experience 24(5), pp. 533-553 (2012).

BSP 'HP SEND'

A BSMP message consists of two parts:

- an arbitrarily-sized **payload**, and
- a fixed-size identifier **tag**.

BSPlib is "buffered on source, buffered on receive":

- When sending a BSMP message, source data is **copied** in the **outgoing communications queue**.
- When receiving a BSMP message, the message is put in an **incoming queue** (during the communication phase).

(Dual-buffering also occurs for the `bsp_put` and `bsp_get`.)

BSP 'HP SEND'

A BSMP message consists of two parts:

- an arbitrarily-sized **payload**, and
- a fixed-size identifier **tag**.

BSPlib is "buffered on source, buffered on receive":

- When sending a BSMP message, source data is **copied** in the **outgoing communications queue**.
- When receiving a BSMP message, the message is put in an **incoming queue** (during the communication phase).

(Dual-buffering also occurs for the `bsp_put` and `bsp_get`.)

BSP 'HP SEND'

A BSMP message consists of two parts:

- an arbitrarily-sized **payload**, and
- a fixed-size identifier **tag**.

BSPlib is "buffered on source, buffered on receive":

- When sending a BSMP message, source data is **copied** in the **outgoing communications queue**.
- When receiving a BSMP message, the message is put in an **incoming queue** (during the communication phase).

(Dual-buffering also occurs for the `bsp_put` and `bsp_get`.)

BSP 'HP SEND'

BSP programming is transparent and safe because of

1. buffering on destination,
2. buffering on source.

This costs memory. Alternative: high-performance (`hp`) variants.

- `bsp_move`; **copies** a message from its incoming communications queue into local memory.
- `bsp_hpmove`; evades this by returning the user a pointer into the queue.
- `bsp_hpsend`; delays reading source data until the message is sent. Local source data should remain unchanged!

(`bsp_hpput` and `bsp_hpget` also exist.)

BSP 'HP SEND'

BSP programming is transparent and safe because of

1. buffering on destination,
2. buffering on source.

This costs memory. Alternative: high-performance (`hp`) variants.

- `bsp_move`; **copies** a message from its incoming communications queue into local memory.
- `bsp_hpmove`; evades this by returning the user a pointer into the queue.
- `bsp_hpsend`; delays reading source data until the message is sent. Local source data should remain unchanged!

(`bsp_hpput` and `bsp_hpget` also exist.)

BSP 'HP SEND'

BSP programming is transparent and safe because of

1. buffering on destination,
2. buffering on source.

This costs memory. Alternative: high-performance (`hp`) variants.

- `bsp_move`; **copies** a message from its incoming communications queue into local memory.
- `bsp_hpmove`; evades this by returning the user a pointer into the queue.
- `bsp_hpsend`; delays reading source data until the message is sent. Local source data should remain unchanged!

(`bsp_hpput` and `bsp_hpget` also exist.)

BSP 'HP SEND'

BSP programming is transparent and safe because of

1. buffering on destination,
2. buffering on source.

This costs memory. Alternative: high-performance (`hp`) variants.

- `bsp_move`; **copies** a message from its incoming communications queue into local memory.
- `bsp_hpmove`; evades this by returning the user a pointer into the queue.
- `bsp_hpsend`; delays reading source data until the message is sent. Local source data should remain unchanged!

(`bsp_hpput` and `bsp_hpget` also exist.)

BSP 'HP SEND'

BSP programming is transparent and safe because of

1. buffering on destination,
2. buffering on source.

This costs memory. Alternative: high-performance (`hp`) variants.

- `bsp_move`; **copies** a message from its incoming communications queue into local memory.
- `bsp_hpmove`; evades this by returning the user a pointer into the queue.
- `bsp_hpsend`; delays reading source data until the message is sent. Local source data should remain unchanged!

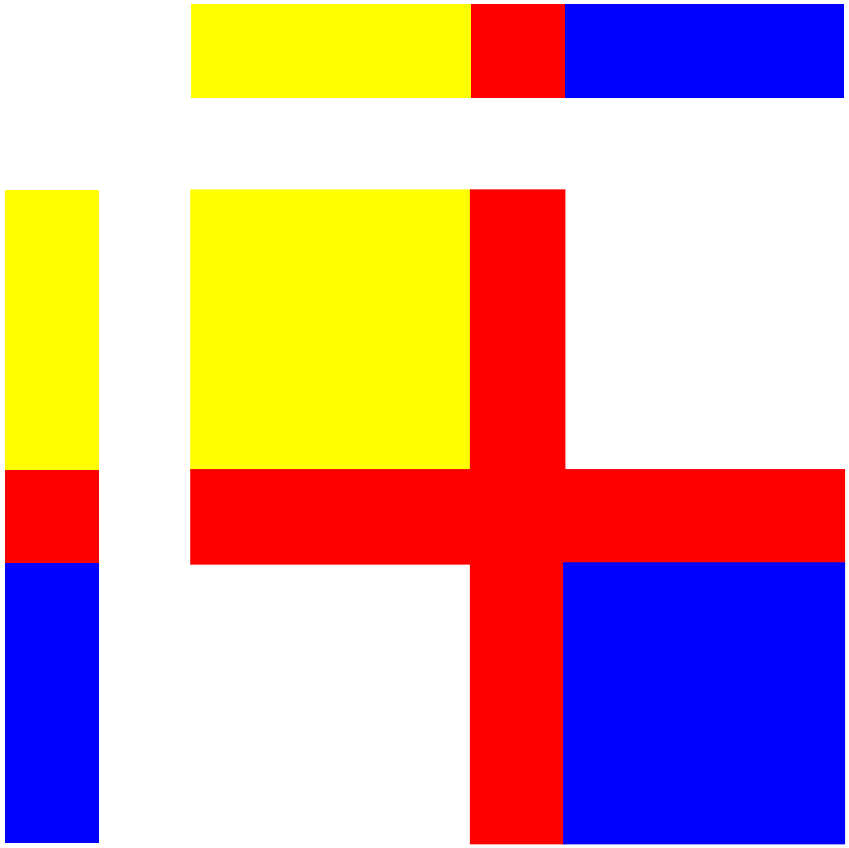
(`bsp_hpput` and `bsp_hpget` also exist.)

Two applications

1. The SpMV multiplication:
   - can we attain state-of-the-art performance?
2. The Fast Fourier Transform (FFT):
   - can we indeed run older (distributed-memory) BSP algorithms, on shared memory, without penalty?

BSP 2D SpMV

Two-dimensional sparse matrix–vector (SpMV) multiply $Ax = y$, using two processors ($p = 2$):



Three steps: (1) fan-out, (2) local SpMV multiply, (3) fan-in.

BSP 2D SpMV

Step 1: fan-out. Request contiguous ranges of x.

```
typedef std::vector< fanQuadlet >::const_iterator IT;
for( IT it = fanIn.begin(); it != fanIn.end(); ++it ) {
    const unsigned long int src_P    = it->remoteP;
    const unsigned long int src_ind  = it->remoteStart;
    const unsigned long int dest_ind = it->localStart;
    const unsigned long int length   = it->length;
    bsp_direct_get( src_P,
                    x,
                    src_ind * sizeof( double ),
                    x + dest_ind,
                    length  * sizeof( double )
                  );
}
```

BSP 2D SpMV

Step 2: local SpMV multiplication:

```
if( A != NULL )      //purely local block
    A->zax( x, y ); //('zax' stands for z=Ax)


if( S != NULL )      //separator blocks
    S->zax( x, y );
```

We use Compressed BICRS storage with the nonzeroes in row-major order.

Yzelman & Roose, 'High-level strategies for parallel shared-memory sparse matrix–vector multiplication', IEEE TPDS, 2013 (in press); paper: http://dx.doi.org/10.1109/TPDS.2013.31, software: http://albert-jan.yzelman.net/software/#SL

BSP 2D SpMV

Step 3: fan-in (I). Send individual row contributions.

```
//the tagsize is initialised to 2*sizeof( ULI )
//fanOut[ i ] has the following layout:
//{ ULI remoteP, localStart, remoteStart, length; }
typedef unsigned long int ULI;

for( ULI i = 0; i < fanOut.size(); ++i ) {
    const ULI dest_P    = fanOut[ i ].remoteP;
    const ULI src_ind   = fanOut[ i ].localStart;
    const ULI length    = fanOut[ i ].length;
    bsp_hpsend( dest_P,
                &( fanOut[ i ].remoteStart ),
                y + src_ind, length * sizeof( double ) );
}
bsp_sync();
```

BSP 2D SpMV

Step 3: fan-in (II). Handle incoming contributions.

```
unsigned long int *msg_tag;
double *msg_payload;
while( bsp_hpmove( (void**)&msg_tag,
                   (void**)&msg_payload
                 ) != SIZE_MAX ) {
    const unsigned long int y_dest = msg_tag[ 0 ];
    const unsigned long int length = msg_tag[ 1 ];
    for( unsigned long int i = 0; i < length; ++i )
        y[ y_dest + i ] += msg_payload[ i ];
}
```

This finishes our implementation of the 2D SpMV multiply.

Fast Fourier Transform

The presented algorithm is a simplified version from that in Chapter 3 of

- Rob H. Bisseling, "**Parallel Scientific Computation** – a structured approach using BSP and MPI", Oxford Press (2003).

The BSP FFT algorithm is

- designed for use on classical distributed-memory systems, and
- modified to use optimised sequential FFT kernels.

(The experiment code uses the full algorithm described in the book.)

Fast Fourier Transform

The discrete Fourier transform takes an input vector $x \in \mathbb{C}^n$ and calculates $y \in \mathbb{C}^n$:

$$y = DFT(x) = F_n x, \quad \text{s.t. } y_i = \sum_{k=0}^{n-1} x_k e^{-2\pi i i k/n}.$$

The FFT computes this in $\Theta(5n \log_2 n)$ flops:

$$F_n = \prod_{i=0}^{m-1} \left(I_{2^i} \otimes B_{n/2^i}\right) \prod_{i=1}^{m} \left(I_{n/2^i} \otimes S_{2^i}\right).$$

With $m = \log_2 n$, B *butterfly* and S *even-odd sorting* matrices.

Fast Fourier Transform

The right-hand series of products amounts to a **bit-reversal**:

$$\prod_{i=1}^{m} \left(I_{n/2^i} \otimes S_{2^i} \right) = R_n.$$

The left-hand series is an **unordered** FFT (UFFT):

$$UFFT(v) = U_n v = \prod_{i=0}^{m-1} \left(I_{2^i} \otimes B_{n/2^i} \right) v.$$

Setting $q = \log_2 p$ and splitting the UFFT yields:

$$\prod_{i=0}^{m-q-1} \left(I_{2^i} \otimes B_{n/2^i} \right) \prod_{i=m-q}^{m-1} \left(I_{2^i} \otimes B_{n/2^i} \right) = G_n \left(I_{n/p} \otimes U_p \right).$$

Fast Fourier Transform

For cyclically distributed x, the local operation $G_n x$ is actually a **generalised** FFT with shift s/p. We can

- use a single unordered GFFT with shift s/p of length n/p to finish our computation,
- use a single multiplication with a diagonal matrix followed by a regular UFFT ($y = G_n x = U_{n/p} R_{n/p} D^{s/p}_{n/p} R_{n/p} x$, where D^{α}_{n} is a diagonal matrix with $d_{jj} = \{e^{-2\pi \imath \alpha j/n}\}$), or
- use a single multiplication with a diagonal matrix, followed by a multiplication with R_n^{-1}, followed by a regular FFT.

...which optimised sequential kernels are available?

Fast Fourier Transform

For cyclically distributed x, the local operation $G_n x$ is actually a **generalised** FFT with shift s/p. We can

- use a single unordered GFFT with shift s/p of length n/p to finish our computation,
- use a single multiplication with a diagonal matrix followed by a regular UFFT ($y = G_n x = U_{n/p} R_{n/p} D^{s/p}_{n/p} R_{n/p} x$, where D^{α}_{n} is a diagonal matrix with $d_{jj} = \{e^{-2\pi\imath\alpha j/n}\}$), or
- use a single multiplication with a diagonal matrix, followed by a multiplication with R_n^{-1}, followed by a regular FFT.

...which optimised sequential kernels are available?

Fast Fourier Transform

For cyclically distributed x, the local operation $G_n x$ is actually a **generalised** FFT with shift s/p. We can

- use a single unordered GFFT with shift s/p of length n/p to finish our computation,
- use a single multiplication with a diagonal matrix followed by a regular UFFT ($y = G_n x = U_{n/p} R_{n/p} D^{s/p}_{n/p} R_{n/p} x$, where D^{α}_{n} is a diagonal matrix with $d_{jj} = \{e^{-2\pi\imath\alpha j/n}\}$), or
- use a single multiplication with a diagonal matrix, followed by a multiplication with R_n^{-1}, followed by a regular FFT.

...which optimised sequential kernels are available?

Fast Fourier Transform

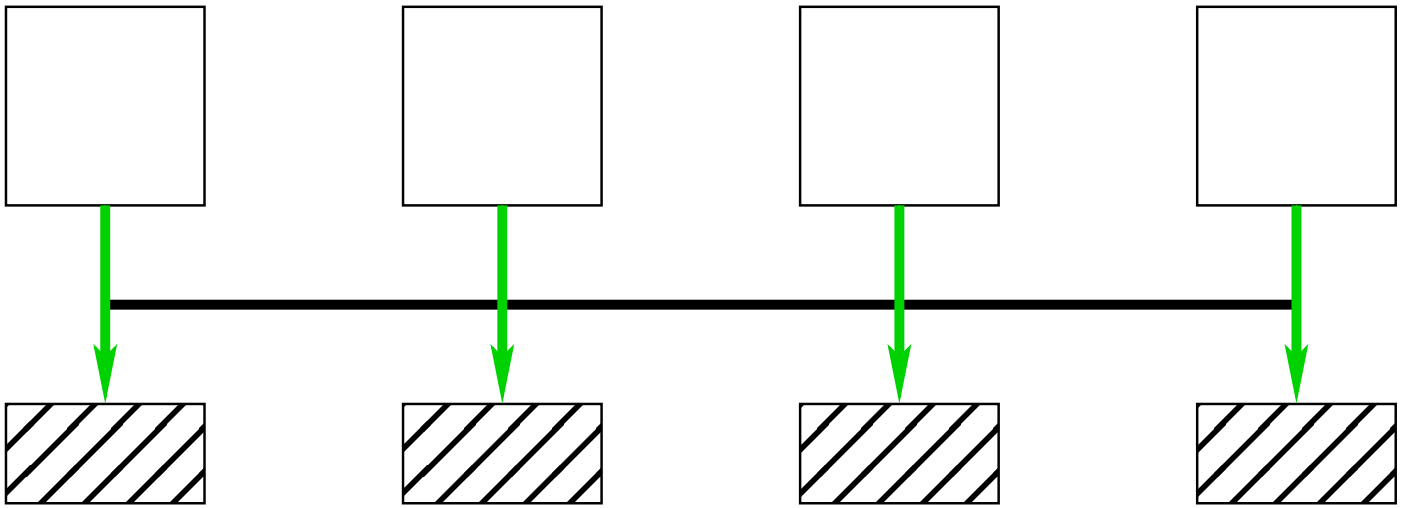
Using sequential FFTW locally, the final algorithm:

1. Initialise x cyclically.
2. Do local bit-reversion.
3. Undo n/p^2 bit-reversions of length p.
4. Do n/p^2 local optimised FFTs of length p.
5. Redistribute y to a cyclic distribution.
6. Twiddle y with $T^{s/p}_{n/p}$.
7. Undo the local bit-reversion of length n/p.
8. Do one local optimised FFT of length n/p.

Thread affinity

Different affinity options:

- **scattered**, maximises bandwidth.
- **compact**, maximises data locality.



MULTIBSP

MulticoreBSP supports nested BSP runs.

E.g., instead of reverting to optimised sequential (U/G)FFTs, we can also use optimised **parallel** FFT kernels.

Consider an 8 socket machine with eight quadcore processors:

- once can start 8 BSP FFT processes, that each revert to
- a parallel FFT using 4 cores.

While this introduces **more** data redistribution stages, the BSP g and l are lower in each of these step; each redistribution step is more **cheap**.

MultiBSP

MulticoreBSP supports nested BSP runs.

E.g., instead of reverting to optimised sequential (U/G)FFTs, we can also use optimised **parallel** FFT kernels.

Consider an 8 socket machine with eight quadcore processors:

- once can start 8 BSP FFT processes, that each revert to
- a parallel FFT using 4 cores.

While this introduces **more** data redistribution stages, the BSP g and l are lower in each of these step; each redistribution step is more **cheap**.

MultiBSP

MulticoreBSP supports nested BSP runs.

E.g., instead of reverting to optimised sequential (U/G)FFTs, we can also use optimised **parallel** FFT kernels.

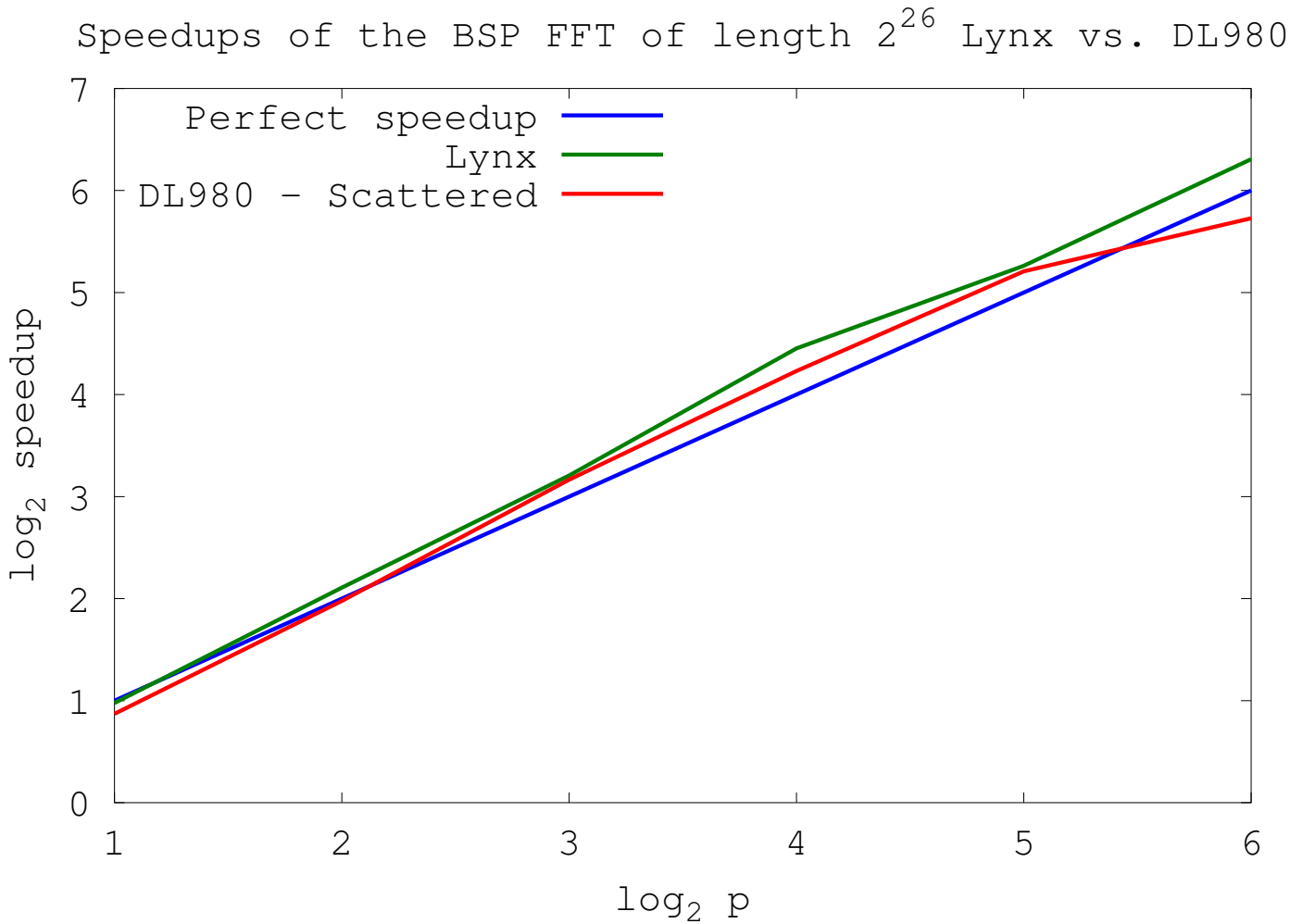
Consider an 8 socket machine with eight quadcore processors:

- once can start 8 BSP FFT processes, that each revert to
- a parallel FFT using 4 cores.

While this introduces **more** data redistribution stages, the BSP g and l are lower in each of these step; each redistribution step is more **cheap**.

UFFT – SHMEM COMPARISON

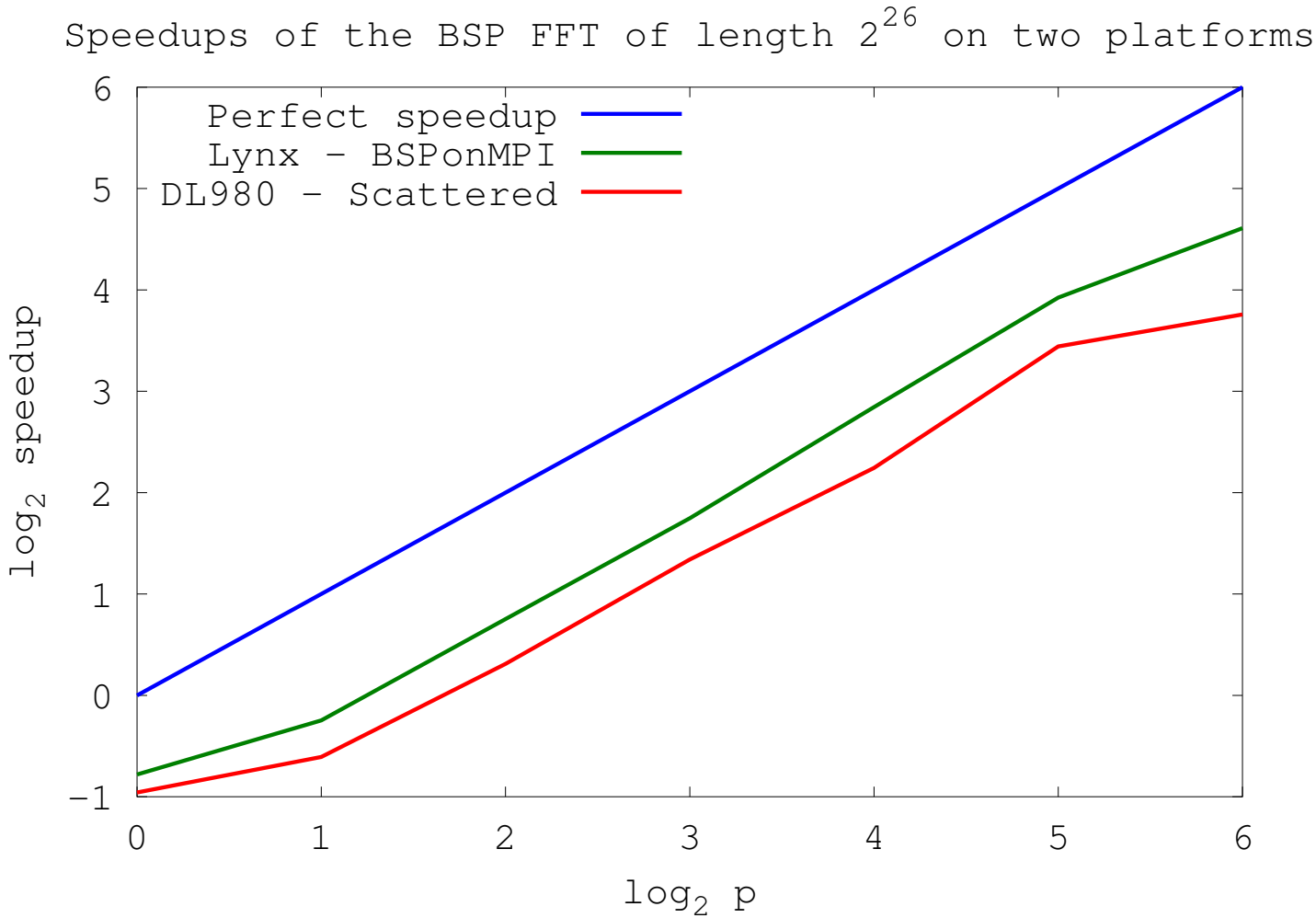
Behaviour on shared-memory is similar to that on distributed memory, but the latter has larger combined caches and bandwidth.



FFTW – SHMEM COMPARISON

Using FFTW is about 1.7x faster on average; 2.6x at most.

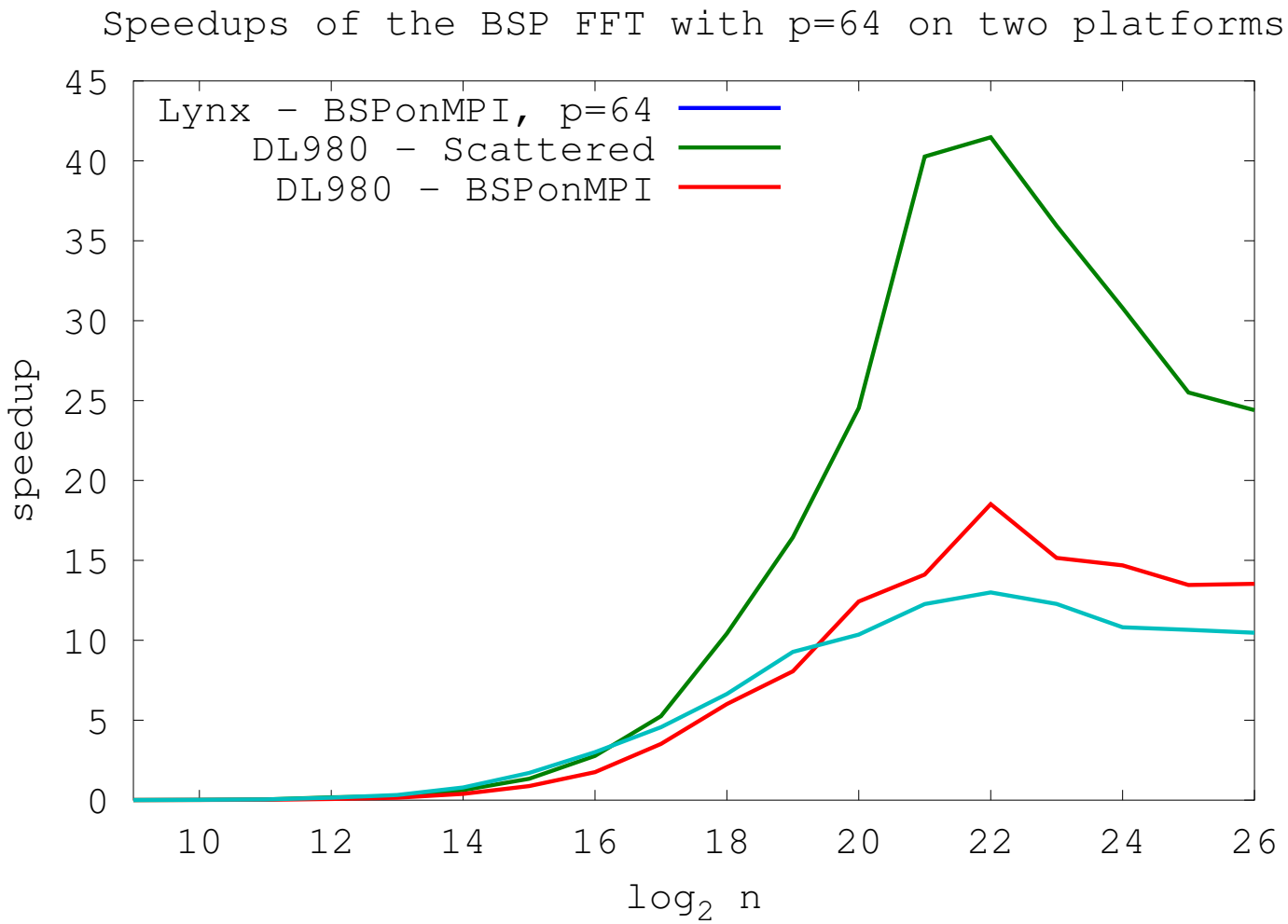
All extra operations cost a factor two in scalability, however.



FFTW – WEAK SCALABILITY

Behaviour in weak scalability is also similar.

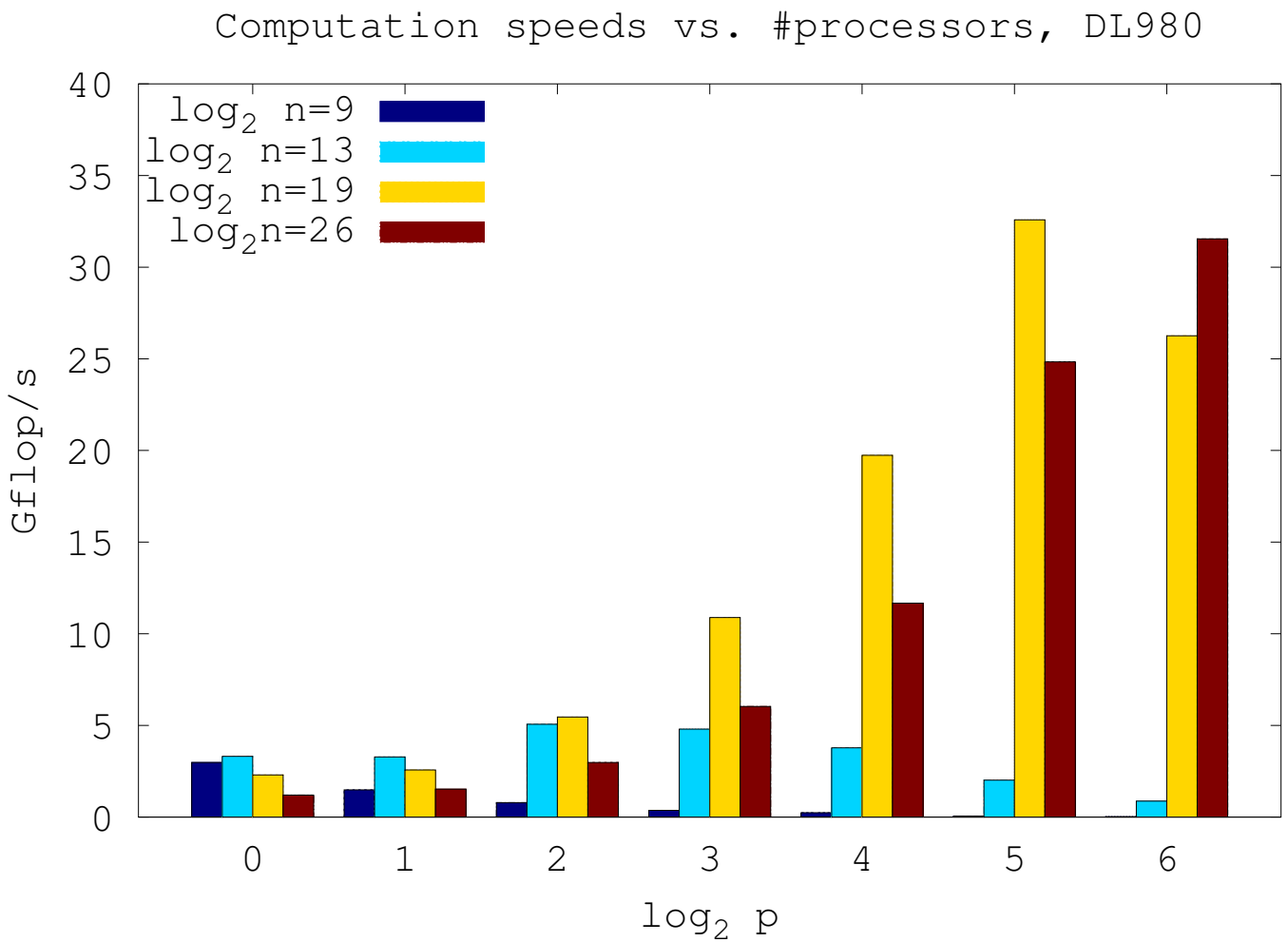
Note that BSPonMPI performs better for small n.



FFTW – RAW SPEEDS

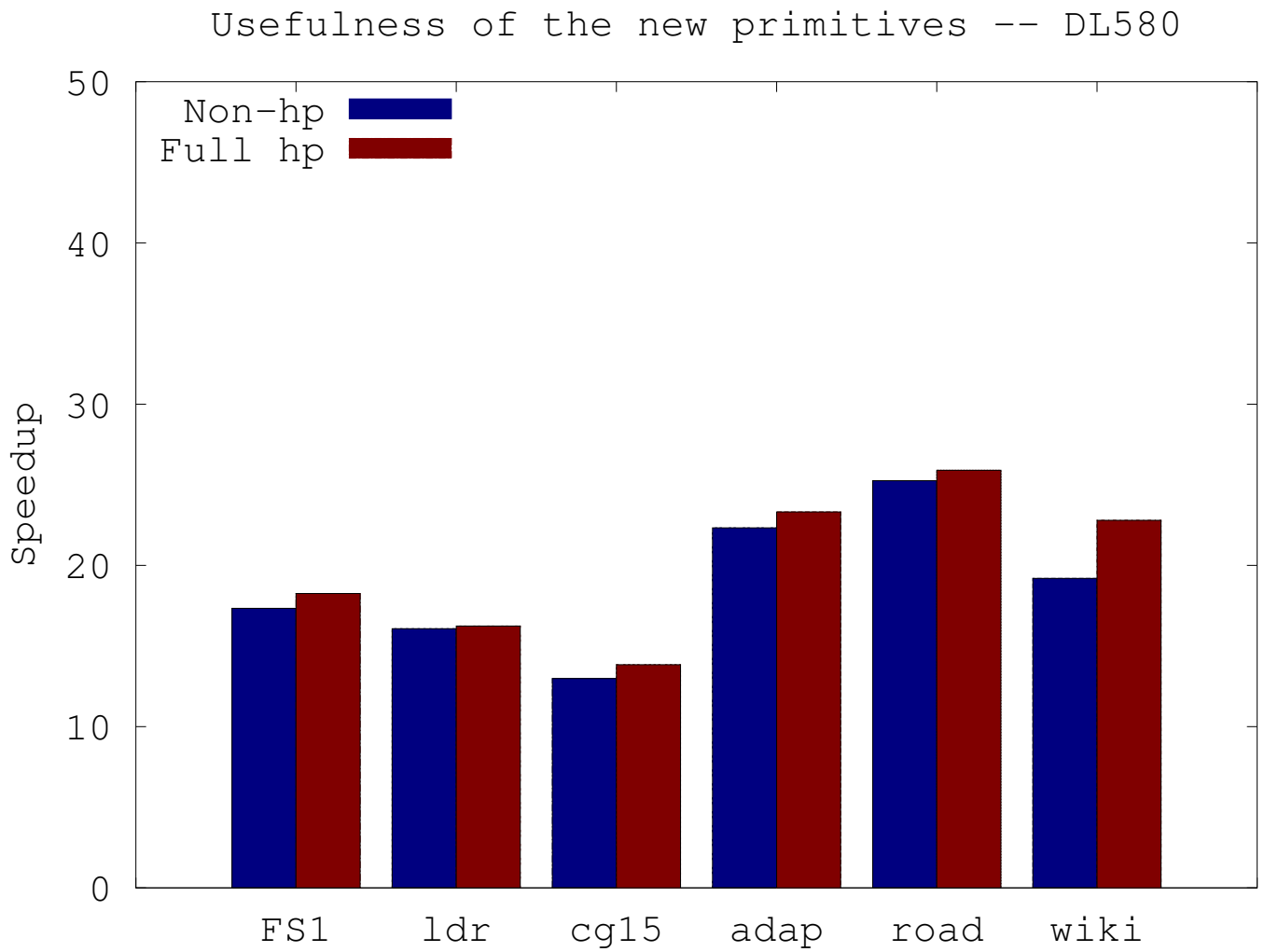
Peak performance ≈ 273 Gflop/s.

Peak bandwidth ≈ 85 GByte/s (27 Gflop/s with $5n \log_2 n$ flops/double).



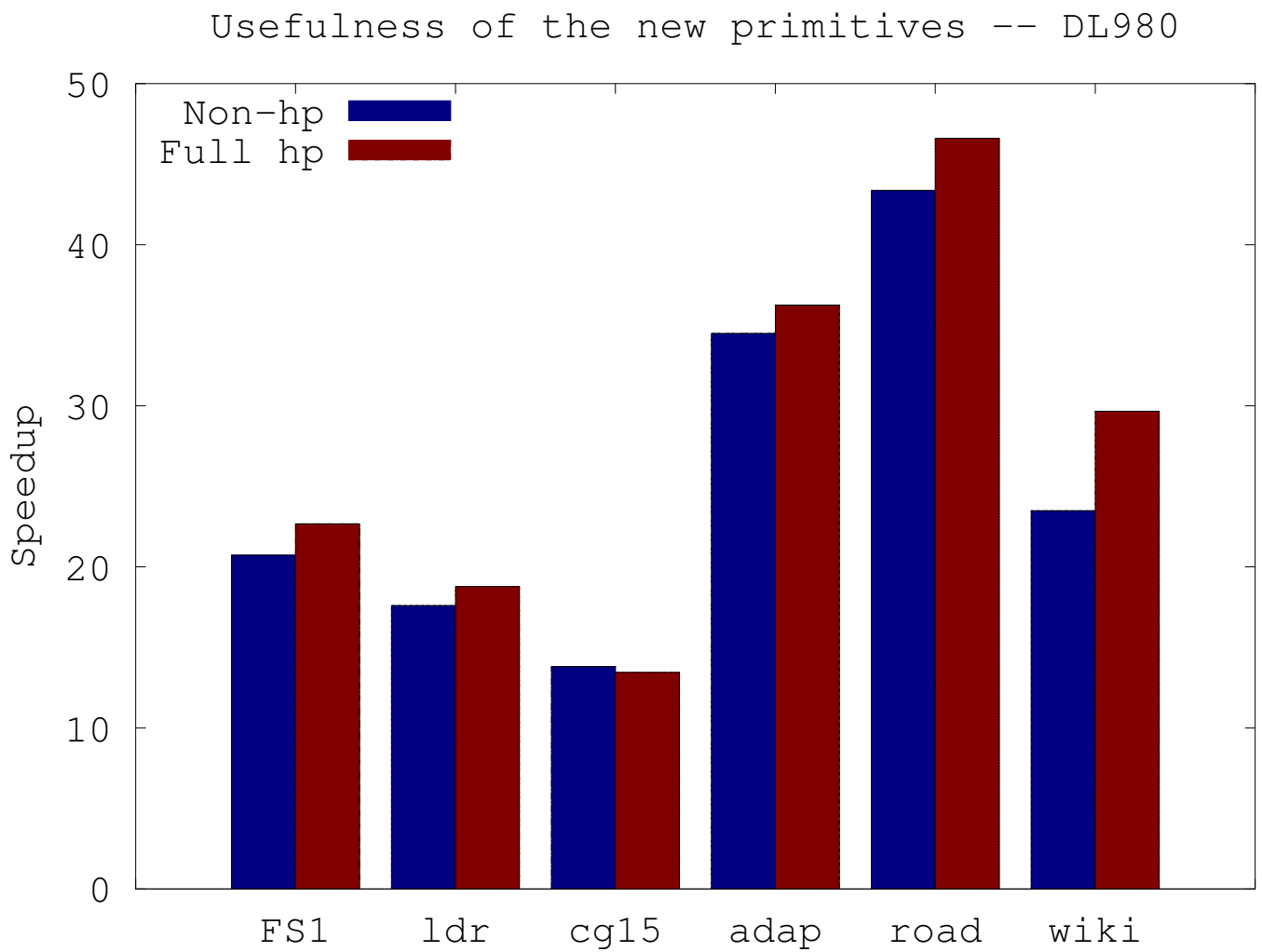
SpMV – new primitives

We test the new primitives using the BSP 2D SpMV multiply:



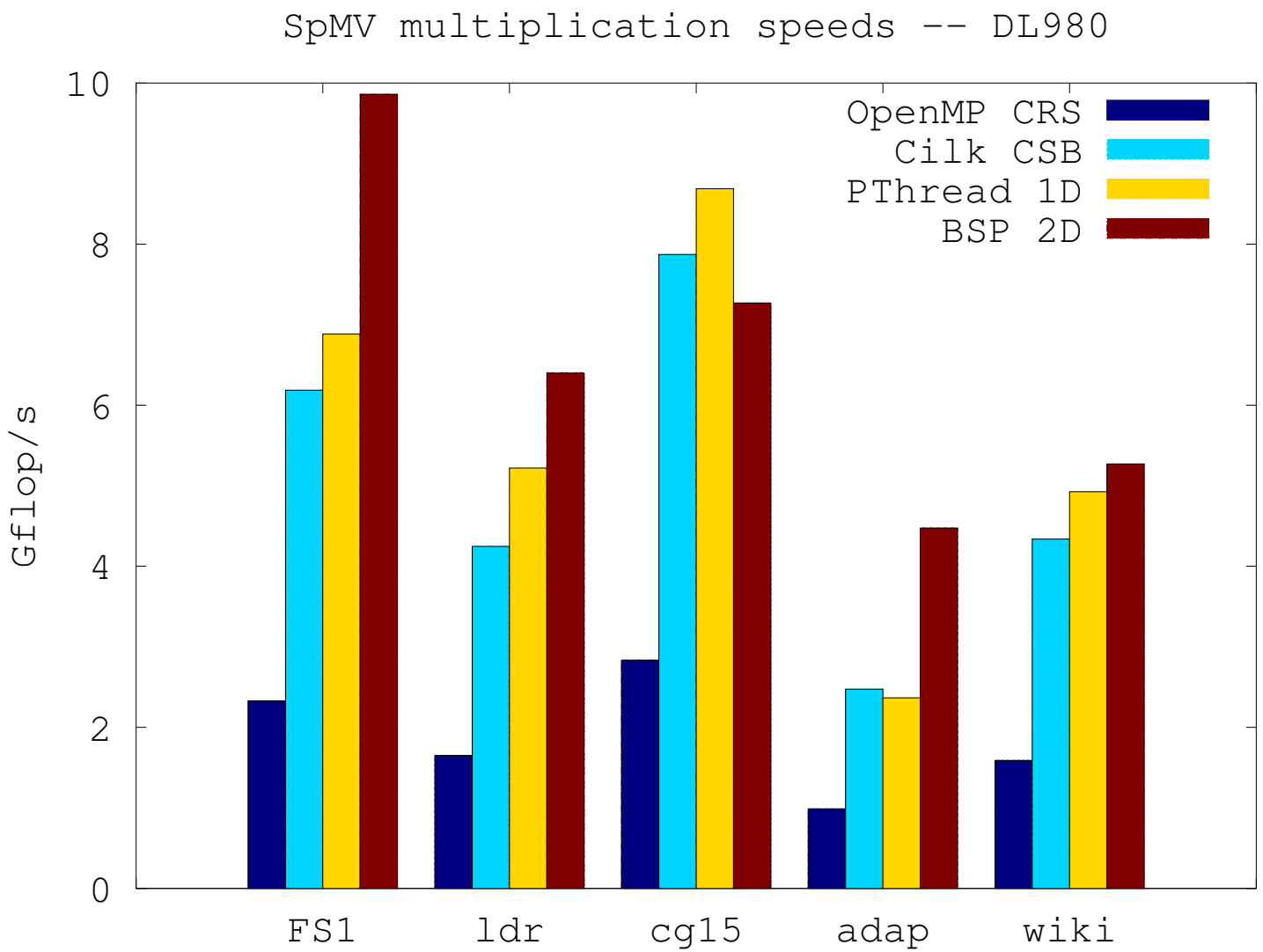
SpMV – new primitives

Same test, different architecture:



SpMV – comparison

The BSP 2D SpMV is often faster than the previous state-of-the-art!



Future work

- Implement and demonstrate the practical gain of a hierarchical execution for the BSP FFT algorithm.
- Compare BSP FFT to the state-of-the-art in multicore FFTs.
- Find the limits of high-performance BSP computing.
- Incorporate distritubed-memory capabilities.
- Avoid global synchronisation barriers.
- Enable fault tolerance.

Conclusions

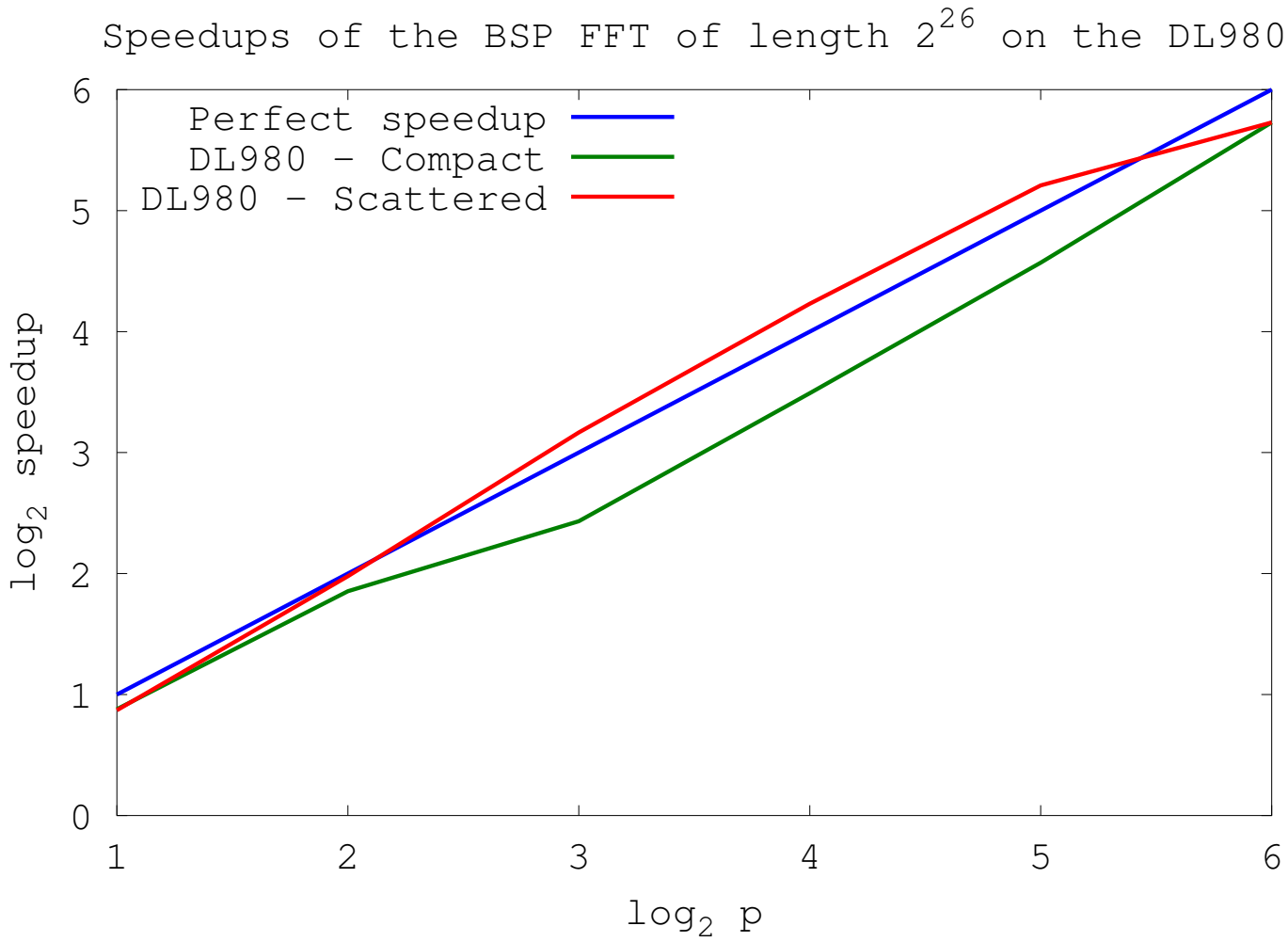
We have

- introduced MulticoreBSP for C and its novel concepts,
- shown running existing BSP algorithms attains similar performance, and
- shown BSP algorithms compete with the state-of-the-art in high-performance computing.

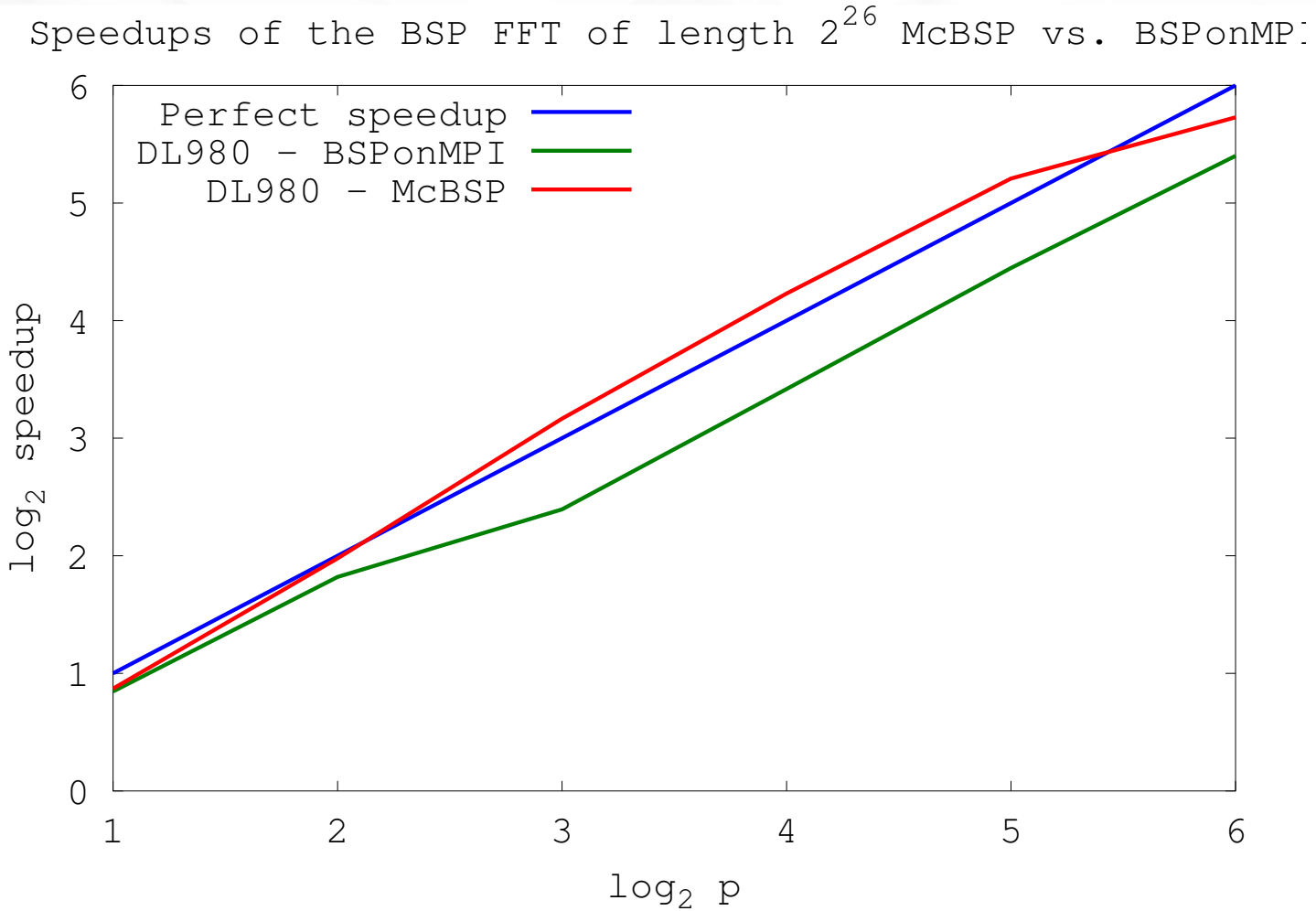
Thank you for your attention!

FFT – THREAD AFFINITY

We experiment on an 8-socket, 64-core machine.

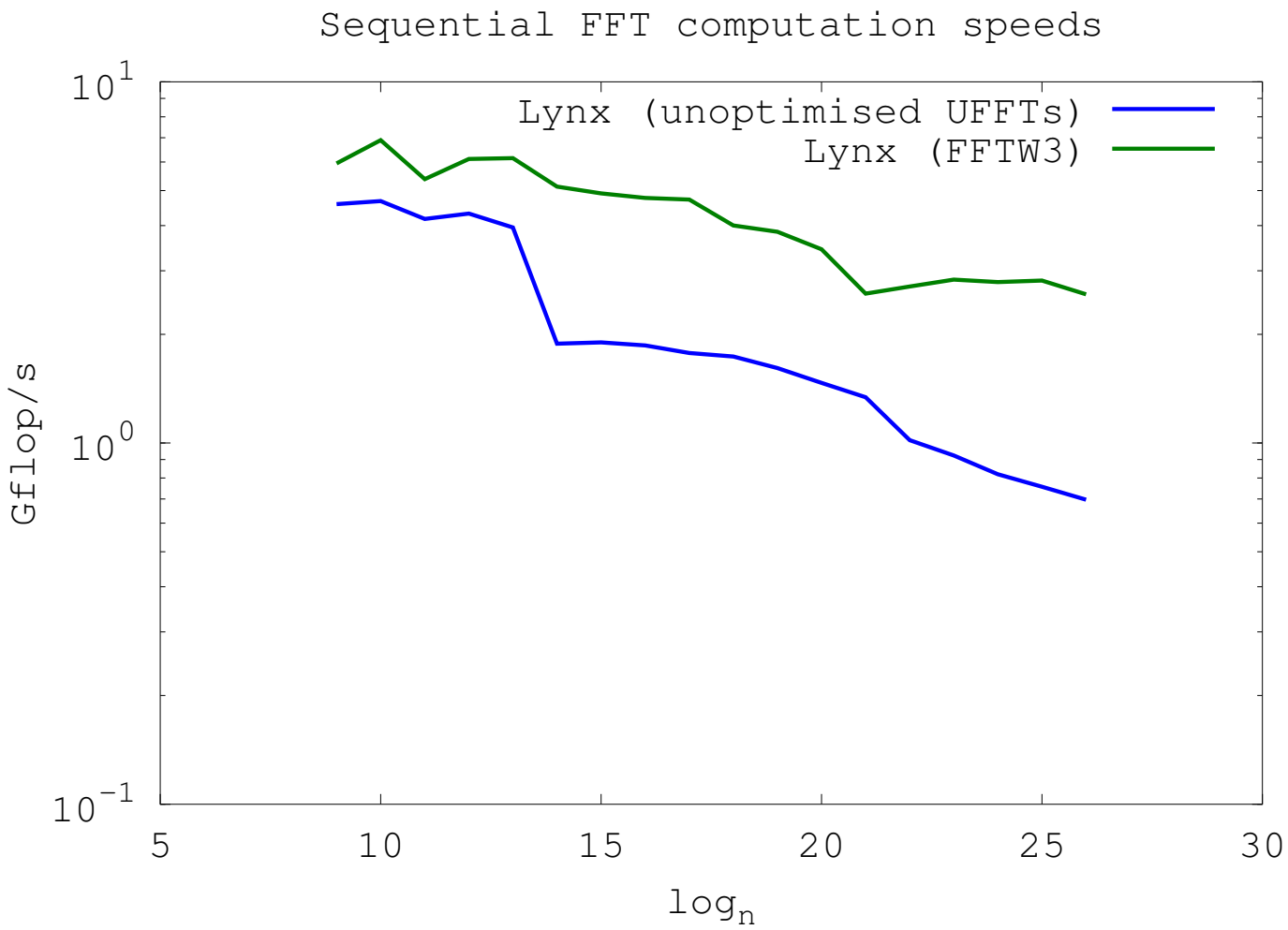


FFT – BSP vs. BSPonMPI



A dedicated shared-memory library is indeed faster than BSPonMPI.

FFT – Mortals vs. FFTW



But how fast is the original UFFT implementation?

Fast Fourier Transform

Consider an entry with global index $(1001101)_2$. Its bit-reversed index is $(1011001)_2$.

If $p = 4$ and x is distributed block-wise:

- the entry is on process $(10)_2$ with local index $(01101)_2$.
- local bit-reversal yields the local index $(10110)_2$ (still at process $(10)_2$).

If $p = 4$ and x is distributed cyclically:

- the entry is on process $(01)_2$ with local index $(10011)_2$.
- Local bit-reversion results in local index $(11001)_2$.

Fast Fourier Transform

Consider an entry with global index $(1001101)_2$. Its bit-reversed index is $(1011001)_2$.

If $p = 4$ and x is distributed block-wise:

- the entry is on process $(10)_2$ with local index $(01101)_2$.
- local bit-reversal yields the local index $(10110)_2$ (still at process $(10)_2$).

If $p = 4$ and x is distributed cyclically:

- the entry is on process $(01)_2$ with local index $(10011)_2$.
- Local bit-reversion results in local index $(11001)_2$.

Thus local bit-reversion of x yields a global bit-reversion with x block-distributed and with bit-reversed process numbers.