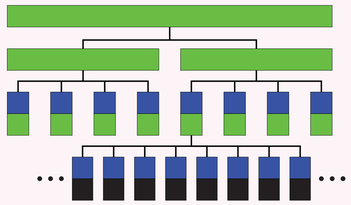
Towards a highly scalable Multi-BSP fast Fourier transform

Albert-Jan N. Yzelman Flanders ExaScience Lab / Department of Computer Science, KU Leuven, Belgium; albert-jan.yzelman@cs.kuleuven.be

The Multi-BSP model

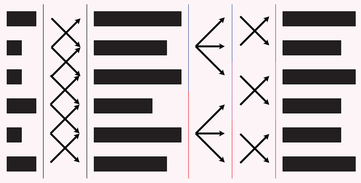
Modern supercomputers consist of shared-memory parallel compute nodes, interconnected by increasingly hierarchical networks. This gives rise to **Non-Uniform Memory Access** (NUMA) effects on all levels of the compute hierarchy.
The recently proposed Multi-BSP [Val11] extends the Bulk Synchronous Parallel model to account for this by modelling

1. **a supercomputer as a tree**. Internal nodes correspond to hardware interconnects and leaf nodes to compute elements:



An example Multi-BSP computer consisting of two four-socket nodes. Each node contains four eight-core processors. Interconnects are coloured green, local memories blue, and compute elements black.

2. a parallel algorithm as an SPMD program with a **strict separation of computation from communication**:



An example of a Multi-BSP algorithm. Vertically, six processes are shown; horizontally, time is depicted. Black bars indicate computation, arrows indicate communication. This is a *conceptual* view on algorithm *design*.

3. the cost of a specific algorithm on a specific computer, expressed in **compute**, **data movement**, and **latency**:

$$T = \sum_{i=0}^{N-1} T^i_{\text{comp}} + \sum_{e=0}^{L-1} \sum_{i=0}^{N_e-1} (h_i g_e + l_e)$$

The Multi-BSP cost. N is the total number of supersteps, and L the number of levels in the Multi-BSP tree. For the eth level, N_e is the number of supersteps, g_e the cost of an h-relation, and l_e the latency.

This allows for **NUMA-aware design** by distinguishing local from remote data movement, and **appropriately penalises large-distance data movement**.

Fast Fourier transform (FFT)

The discrete Fourier transform on complex input computes

$$y = F_n x, \quad \text{with } y_i = \sum_{j=0}^{n-1} x_j e^{-2\pi i i j/n}$$

The radix-2 decimation-in-time FFT computes the same and can be succinctly written as

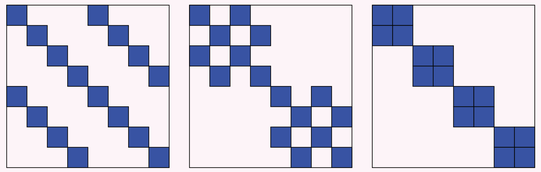
$$F_n x = \left[\Pi_{i=0}^{m-1} \left(I_{2^i} \otimes B_{2^{m-i}}\right)\right] V_n x$$

An efficient BSP FFT splits this computation in two [IB01]:

$$F_n = \left[\underbrace{\Pi_{i=0}^{t-1} \left(I_{2^i} \otimes B_{2^{m-i}}\right)}_{L_n} \underbrace{\Pi_{i=t}^{m-1} \left(I_{2^i} \otimes B_{2^{m-i}}\right)}_{R_n}\right] V_n$$

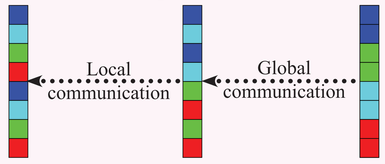
For integer m, q *let* $n=2^m$ *and* $p=2^q$. *The default value for* t *is* m-q.

The structure of the first stages of the FFT computation is well-suited to the **block distribution**; the computation then can proceed without intermittent communications. The latter stages require a **cyclic distribution** instead:



The $\log_2 n$ *stages of a complete FFT for* n=8. *Computation progresses from right to left. For* p=4 *the middle matrix requires a cyclic distribution to avoid communication; for* p=2, *redistribution can occur one stage later.*

This BSP FFT thus has only one communication phase, where data is redistributed by switching the input vector from a block distribution to a cyclic distribution.
Since only leaf nodes may compute, a Multi-BSP FFT algorithm only adapts data movement by mapping the data redistribution stage onto a Multi-BSP computer tree:



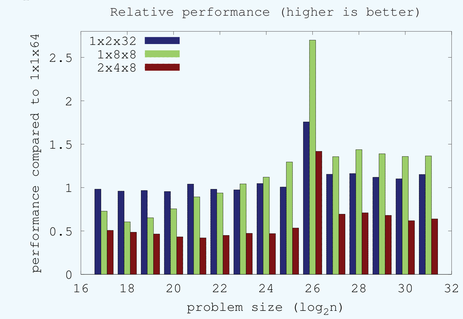
An example redistribution for n=8 *and a two-level Multi-BSP computer.*

Results

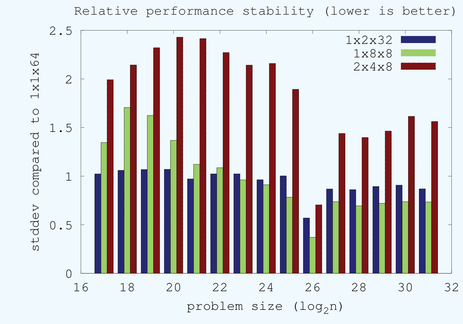
The **64-core HP DL980** machine combines two Intel QuickPath Interconnects (QPIs) through a custom HP interface. Each QPI connects four sockets, and each socket contains an eight-core Intel Xeon Sandy Bridge processor.
There are **several ways to model** this computer as a Multi-BSP tree:

- flat (1x1x64),
- two-level by QPI (1x2x32).
- two-level by sockets (1x8x8), or
- three-level (2x4x8).

A prototype Multi-BSP FFT was implemented and run on each of these models. These are **preliminary results**. The performances, normalised to the flat model, follow:



The observed standard deviation over 10x30 runs indicates the stability of this performance:



Conclusions

The current Multi-BSP FFT
- is derived from the regular BSP algorithm [IB01] and its high-performance implementation [YBRM13],
- does not change the parallel work distribution, but
- does change the communication patterns, and
- is implemented on top of a Multi-BSP programming framework that uses MulticoreBSP for C [YBRM13].

Although still work in progress, early results show that
- adapting the communication pattern to a target architecture leads to **faster communication**, while
- this approach works for any recursively-defined Multi-BSP algorithm and **automatically adapts itself** to a target Multi-BSP computer model; explicitly programming all levels of the tree is not required.
- molding an algorithm to a target architecture also helps to **reduce variability** in algorithm execution speeds.

All code will soon be **freely available** under the terms of the LGPL; see MulticoreBSP.com for updates.

Future work

Several aspects for improvements are readily identified:
- increasing the efficiency of the Multi-BSP framework; the high-performance BSP FFT implemented directly on MulticoreBSP for C [YRM13] performs better in absolute performance. In particular,
- every downwards traversal through the compute tree unnecessarily allocates new system resources that are freed with every upwards traversal, while a static allocation would **incur overhead only once** per Multi-BSP enabled algorithm.
- achieve the **bounds of optimality** for the FFT within the (Multi-)BSP model, cf. Valiant [Val90,Val11].
- account for the **local memory** sizes that Multi-BSP also models. This will adapt computation, and allows for more efficient thread-local use of libraries such as FFTW and Spiral. This leads to more stable and higher performance.

```
//code that runs on the leaf nodes
template< size_t _P, size_t ... _tail >
class FFT : public MultiBSP_program< _P > {

        //the input vector
        Distributed_Vector< double, _P > &x;

        void superstep( const size_t i ) {
            if( i == 0 ) {
                //assume input is cyclic; do local
                //multiplication with R_n and V_n
                stage1();
            } else {
                //do redistribution back to cyclic
                redistribute();
                //perform final multiplication with L_n
                stage2();
            }
        }
};
```

```
//code that runs on internal nodes
template< size_t _P, size_t _subP, size_t ... _tail >
class FFT< _P, _subP, _tail... > :
    public MultiBSP_program< _P, _subP, _tail... > {

        //the input vector
        Distributed_Vector< double, _P, _subP, _tail... > &x;

        void superstep( const size_t i ) {
            if( i == 0 ) {
                //do stage1 computes further down the tree
                bsp_recurse();
            } else {
                //do global redistribution
                redistribute();
                //do local redistributions and final computes
                bsp_recurse();
            }
        }
};
```

High level pseudocode for the Multi-BSP FFT

Multi-BSP programmability

C++ **template recursion** simplifies Multi-BSP programming. The current implementation framework allows **SPMD** programming and the use of the transparent and robust **one-sided communication**, both trademarks of BSP. It only requires **explicit implementation on two levels**; code automatically expands to the full Multi-BSP computer hierarchy.
The FFT pseudocode (left) demonstrates this approach. It employs a **separation of data from algorithm** which benefits modularity (right). In these code snippets, a distributed vector has its elements distributed over all leaf nodes, whereas a replicated vector has a fixed size at both internal and leaf nodes.

```
//the 2 QPI, 4 socket, 8 core model (one-line model switching!)
#define MULTIBSP_COMPUTER 2, 4, 8

int main( int argc, char ** argv ) {
    ...
    //declare input/output vector (our FFT is in-place)
    Distributed_Vector_Root< double, MULTIBSP_COMPUTER > x(2*n);

    //declare FFT data (contains weights, permutations, ...)
    FFT_data< MULTIBSP_COMPUTER > fft_data( n, FORWARD_FFT );

    //initialise distributed data structures (variable arguments)
    Initialiser< MULTIBSP_COMPUTER >::initialise( x, fft_data );

    //declare FFT program
    FFT< MULTIBSP_COMPUTER > fft( fft_data, x );

    ...
    fft.begin(); //execute a FFT in parallel
    ...

    return 0;
}
```

Modularity in Multi-BSP programming

http://www.multicorebsp.com

References:

- [IB01] M. A. Inda, R. H. Bisseling, “A simple and efficient parallel FFT algorithm using the BSP model”, Parallel Computing, Volume 27, 2001; pages 1847 to 1878.
- [Val90] L. G. Valiant, “A bridging model for parallel computation”, Communications of the ACM, Volume 33, Issue 8, 1990; pages 103 to 111.
- [Val11] L. G. Valiant, “A bridging model for multi-core computing”, Journal of Computer and System Sciences, Volume 77, Issue 1, 2011; pages 154 to 166.
- [YBRM13] A. N. Yzelman, R. H. Bisseling, D. Roose, K. Meerbergen, “MulticoreBSP for C: a high-performance library for shared-memory parallel programming”, International Journal of Parallel Programming, in press (2013).