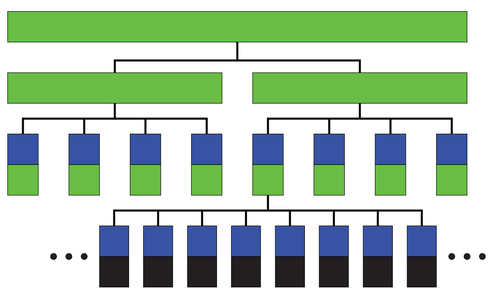
Automatic deployment of parallel algorithms

Albert-Jan N. Yzelman ExaScience Life HPC Lab / Department of Computer Science, KU Leuven, Belgium; albert-jan@yzelman.net

The Multi Bulk Synchronous Parallel (Multi-BSP) model

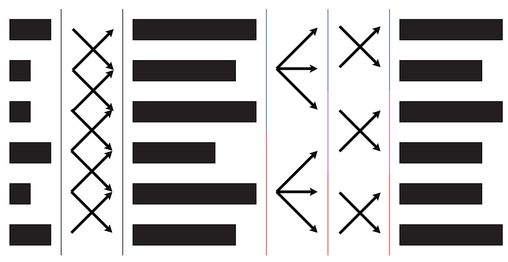
Modern supercomputers consist of shared-memory parallel compute nodes, interconnected by increasingly hierarchical networks. This gives rise to **Non-Uniform Memory Access** (NUMA) effects on all levels of the compute hierarchy. The recently proposed Multi-BSP [Val11] extends the Bulk Synchronous Parallel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

1. **a supercomputer** as a tree. Internal nodes correspond to communication networks while leaf nodes correspond to compute cores:



This allows **precise modeling of multi-core architectures**, as well as of distributed compute nodes, and combinations.

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



This exposes the **h-relation** of a parallel algorithm; i.e., the maximum number a processor either sends or receives. In the above, from left to right, we have $h=(2,3,1)$.

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

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

Here, N is the total number of supersteps, while L equals 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 a single h-relation, and l_e the latency cost.
This allows for **NUMA-aware algorithm design**, by distinguishing local from remote data movement, and subsequently the ability to appropriately **penalise large-distance data movement**.

The Multi-BSP fast Fourier transform

The discrete Fourier transform F_n with complex-valued input and output vectors x, y of length n computes

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

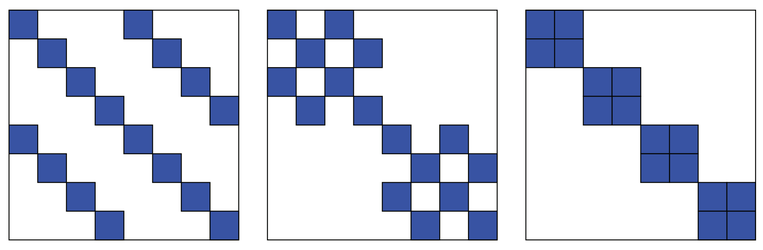
The radix-2 decimation-in-time FFT computes the same in loglinear time by decomposing the above as:

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

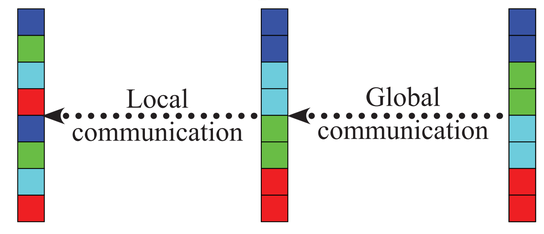
An optimal BSP FFT algorithm splits this in two [IB01]:

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

The following illustrates the three stages of an FFT with $n=8$, where matrix-vector multiplication progresses from the right-most matrix to the left-most one:



Should the vector x be **block distributed** over four processes, then matrix-vector multiplication with the right-most matrix is an entirely processor-local endeavour. For the left-most matrix, however, we require x to **switch to a cyclic distribution** instead. For large n, there is some freedom in selecting the splitting stage t.
The BSP algorithm thus requires a complete redistribution of x once, and two stages of multiplication with R_n and L_n. For a smart choice of t, these stages can be expressed as an element-wise multiplication of vector elements (twiddling), followed by multiple shorter-length FFTs; i.e., the parallel FFT is **decomposable** in many sequential FFTs.
Since only leaf nodes may compute, a Multi-BSP FFT algorithm only adapts the communication stage of this optimal BSP FFT algorithm; **data movement during redistribution follows the Multi-BSP computer tree**:



The above illustrates the block-to-cyclic redistribution run on a Multi-BSP computer with four cores laid out in a binary tree (e.g., a two-socket machine with dual-core processors).

Collective communication

Collective communications are common patterns in parallel programming that involve communication over the entire parallel machine. Examples of such collectives include the
- broadcast (one-to-all),
- gather (all-to-one),
- reduce (reducing multiple values to one), and
- allreduce (reduction combined with a broadcast).

Their use for scientific computing naturally arises during the implementation of parallel iterative solvers, mesh boundary exchanges, matrix computations, and so on.

Writing collectives and other numerical templates using Multi-BSP benefits parallel scientific computation in two ways:
1. programs will be highly efficient and will **automatically adapt** to the layout of the target machine, whether it be a supercomputer or your home laptop;
2. Multi-BSP programming is **transparant**, **easily learnt**, and relatively **robust** to programming errors; this enables more focus on science, as opposed to implementation.

The below code snippets sketch a Multi-BSP broadcast implementation. This is work in progress; for updates, see

www.multicorebsp.com

Automatic deployment

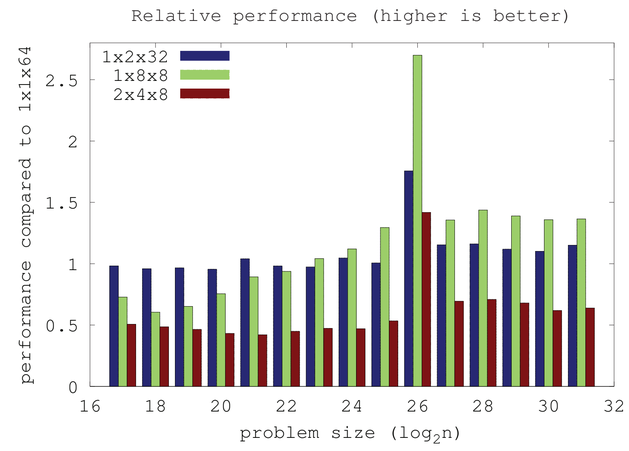
A Multi-BSP algorithm written using MulticoreBSP for C will, at compile time, automatically expand to adapt to a given MultiBSP computer model.
In the below, a Multi-BSP algorithm that generates two random vectors and calculates their inner product in a distributed fashion is declared, expanded, and executed:

```
#define COMPUTER_MODEL 2, 2, 1

int main() {
    std::cout << "MultiBSP computer configuration:\n" <<
    MultiBSP< COMPUTER_MODEL >::description() << "\n";

    MultiBSP_IP algorithm;
    MultiBSP< COMPUTER_MODEL >::bsp_begin( algorithm );

    std::cout << "\nResult: " << algorithm.result;
    std::cout << ", expected: " << (N/4.0) << "\n";
    return 0;
}

$ ./multibsp_ip
MultiBSP computer configuration:
( 2, 8 GB) -> ( 2, 32 MB) -> ( 1, 32 kB)

Result: 65548.7, expected: 65536
```

Note that the target computer model is changed simply by **modifying the first program line** only.

Preliminary results: FFT

Our **64-core** machine combines two QuickPath Interconnects (QPIs); each QPI connects four sockets, each socket contains an eight-core processor. This may be modelled in Multi-BSP in various ways: $L=1$ (1x1x64), $L=2$ (1x2x32), $L=2$ (1x8x8), or $L=3$ (2x4x8).
A prototype Multi-BSP FFT was implemented and run on each of these models:



These results indicate the two-level eight by eight model fits this machine best.
The below code snippets illustrate the programming effort involved, and illustrate how Multi-BSP templates, such as that of the broadcast collective (on the far left), may be reused within other code snippets and end-user programs.

Code snippets: broadcast

```
void bcast( const ValueType * const &toBroadcast ) {
    if( bsp_pid() == 0 )
        for( size_t k = 1; k < bsp_nprocs(); ++k )
            bsp_put( k, toBroadcast, &receive_buffer );
}

void send_to_root() {
    const unsigned int s = bsp_pid();
    if( root && s > 0 )
        bsp_put( 0, &receive_buffer, &receive_buffer );
    bsp_sync();
    if( s == 0 )
        root = true;
    bsp_up();
}
```

```
void nested() {
    send_to_root();
    bcast( &receive_buffer );
    bsp_down();
}

ValueType leaf( const ValueType * const toBroadcast ) {
    if( toBroadcast == NULL )
        root = false;
    else {
        receive_buffer = *toBroadcast;
        root = true;
    }
    send_to_root();
    bcast( &receive_buffer );
    bsp_sync();
    return receive_buffer;
}
```

Code snippets: FFT

```
void init( const size_t n ) {
    //initialise broadcast collective
    BCAST::init();
    //do FFTW autotuning sequentially
    if( bsp_leaf() && bsp_lid() == 0 ) {
        initFFTW( n );
        //broadcast initialised FFTW plans
        BCAST::leaf( &fftw_stage1 );
        BCAST::leaf( &fftw_stage2 );
    } else {
        if( bsp_leaf() ) {
            //receive broadcasted FFTW plans
            fftw_stage1 = BCAST::leaf(NULL);
            fftw_stage2 = BCAST::leaf(NULL);
        } else {
            //execute nested broadcast code
            BCAST::nested(); //for stage 1
            BCAST::nested(); //for stage 2
        }
    }
}
```

```
void leaf( double * const x ) {
    //do stage 1
    stage1( x );
    //perform higher-level redistributions
    bsp_up();
    //do leaf-level redistribution,
    //followed by stage 2 computations
    stage2( x );
}

void nested( double * const x ) {
    //first do global redistributions
    bsp_up();
    //now do local redistribution
    nested_redistribute( x );
    //do lower-level redistributions
    //and 2nd stage leaf computations
    bsp_down();
}
```

References:
- [IB01] M. A. Inda, R. H. Bisseling, “A simple and efficient parallel FFT algorithm using the BSP model”, Parallel Computing, Vol. 27, 2001.
- [Val90] L. G. Valiant, “A bridging model for parallel computation”, Comm. of the ACM, Vol. 33(8), 1990.
- [Val11] L. G. Valiant, “A bridging model for multi-core computing”, J. of Computer and System Sciences, Vol. 77(1), 2011.
- [YBRM13] A. N. Yzelman, R. H. Bisseling, D. Roose, K. Meerbergen, “MulticoreBSP for C: a high-performance library for shared-memory parallel programming”, Intl. J. Parallel Programming, Vol. 24(4), 2013.