Modern BSP

Jan-Willem Buurlage, CWI, Amsterdam
Prepared for MasterMath course Parallel Algorithms, 2017-09-27
BSP today

• BSP is still the leading model for *distributed* computing, used in industry.
  • MapReduce
  • Pregel

• BSP programming usually done using MPI or the various Apache projects (*Hama*, *Giraph*, *Hadoop*).

• BSPlib provides an accessible way to familiarize yourself with parallel programming.
Google’s MapReduce (Example)\textsuperscript{1}

- Classic example: word count. The \textit{map} takes (file, content) pair, and emits (word, 1) pairs for each word in the content. The \textit{reduce} function sums over all mapped pairs with the same word.

- The Map and Reduce are performed in parallel, and are both followed by communication and a bulk synchronization, which means MapReduce $\subset$ BSP!

\textsuperscript{1}MapReduce: Simplified Data Processing on Large Clusters, Jeffrey Dean and Sanjay Ghemawat (2004)
Google’s Pregel

BSP for graph processing, used by Google\textsuperscript{2} and Facebook\textsuperscript{3}:

“The high-level organization of Pregel programs is inspired by Valiant’s Bulk Synchronous Parallel model. Pregel computations consist of a sequence of iterations, called supersteps ... It can read messages sent to V in superstep $S - 1$, send messages to other vertices that will be received at superstep $S + 1$ ...”

\textsuperscript{2}Pregel: A System for Large-Scale Graph Processing – Malewicz et al. (2010)
\textsuperscript{3}One Trillion Edges: Graph Processing at Facebook-Scale - Avery Ching et al (2015).
Modern BSP

- These frameworks are good for big data analytics, too limiting for general purpose scientific computing
- Most scientific software built with MPI
- Modern languages have features (safety, abstractions) which can aid parallel programming. Full power of BSP not yet available in such a language.
5th most cited xkcd\textsuperscript{4}

\textsuperscript{4}https://xkcdref.info/statistics/
• **Bulk** is a BSP library for modern C++
• Provides a safe and simple layer on top of low-level technologies, user can avoid dealing with the *transport layer*.
• BSPlib already improves upon MPI in this regard.
Goals of Bulk

- Unified and *modern* interface for distributed and parallel computing.
- Works across a wide variety of platforms, flexible *backends*.
- Shorter, safer, makes it easier to write (correct) programs.
```c
#include <bsp.h>

int main() {
    bsp_begin(bsp_nprocs());
    int s = bsp_pid();
    int p = bsp_nprocs();
    printf("Hello World from processor %d / %d", s, p);
    bsp_end();

    return 0;
}
```
#include <bulk/bulk.hpp>
#include <bulk/backends/mpi/mpi.hpp>

int main() {
    bulk::mpi::environment env;
    env.spawn(env.available_processors(), [](auto& world) {
        auto s = world.rank();
        auto p = world.active_processors();

        world.log("Hello world from processor %d / %d!", s, p);
    });
}
// BSPlib
int x = 0;
bsp_push_reg(&x, sizeof(int));
bsp_sync();
...
bsp_pop_reg(&x);

// Bulk
auto x = bulk::var<int>(world);
// BSPlib
int b = 3;
bsp_put(t, &b, &x, 0, sizeof(int));

int c = 0;
bsp_get(t, &x, 0, &c, sizeof(int));

bsp_sync();

// Bulk
x(t) = 3;
auto c = x(t).get();

world.sync();
// BSPlib
int* xs = malloc(10 * sizeof(int));
bsp_push_reg(xs, 10 * sizeof(int));
bsp_sync();

int ys[3] = {2, 3, 4};
bsp_put(t, ys, xs, 2, 3 * sizeof(int));
int z = 5;
bsp_put(t, &z, xs, 0, sizeof(int));

bsp_sync();

...

bsp_pop_reg(xs);
free(xs);
// Bulk
auto xs = bulk::coarray<int>(world, 10);
xs(t)[{2, 5}] = {2, 3, 4};
xs(t)[0] = 5;

world.sync();
// BSPlib
int s = bsp_pid();
int p = bsp_nprocs();

int tagsize = sizeof(int);
bsp_set_tagsize(&tagsize);
bsp_sync();

int tag = 1;
int payload = 42 + s;
bsp_send((s + 1) % p, &tag, &payload, sizeof(int));
bsp_sync();

int packets = 0;
int accum_bytes = 0;
bsp_qsize(&packets, &accum_bytes);

int payload_in = 0;
int payload_size = 0;
int tag_in = 0;
for (int i = 0; i < packets; ++i) {
    bsp_get_tag(&payload_size, &tag_in);
    bsp_move(&payload_in, sizeof(int));
    printf("payload: %i, tag: %i", payload_in, tag_in);
}
// Bulk
auto s = world.rank();
auto p = world.active_processors();

auto q = bulk::queue<int, int>(world);
q(world.next_rank()).send(1, 42 + s);
world.sync();

for (auto [tag, content] : queue) {
    world.log("payload: %i, tag: %i", content, tag);
}
```cpp
// Generic queues
auto q = bulk::queue<int, int, int, float[]>(world);
q(t).send(1, 2, 3, {4.0f, 5.0f, 6.0f});
world.sync();

for (auto [i, j, k, values] : queue) {
    // ...
}

// Standard containers
std::sort(q.begin(), q.end());

auto maxs = bulk::gather_all(world, max);
max = *std::max_element(maxs.begin(), maxs.end());

// Skeletons
// result_1 + result_2 + ... + result_p
auto alpha = bulk::foldl(result, std::plus<int>());
```
Summary of Bulk

- Modern interface for writing parallel programs, safer and clearer code
- Works together with other libraries because of generic containers and higher-level functions.
- Works across more (mixed!) platforms than competing libraries (because of the backend mechanism).
Parallella

- ‘A supercomputer for everyone, with the lofty goal of “democratizing access to parallel computing’
- Crowd-funded development board, raised almost $1M in 2012.
Epiphany co-processor

- $N \times N$ grid of RISC processors, clocked by default at 600 MHz (current generations have 16 or 64 cores).
- Efficient communication network with ‘zero-cost start up’ communication. Asynchronous connection to external memory pool using DMA engines (used for software caching).
- Energy efficient @ 50 GFLOPs/W (single precision), in 2011, top GPUs about 5× less efficient.
Epiphany memory

- Each Epiphany core has 32 kB of **local memory**, on 16-core model 512 kB available in total.
- On each core, the kernel binary and stack already take up a large section of this memory. Duplication.
- On the Parallella, there is 32 MB of **external RAM** shared between the cores, and 1 GB of additional RAM accessible from the ARM host processor.
Many-core co-processors

- **Applications**: Mobile, Education, possibly even HPC.
- Specialized (co)processors for AI, Computer Vision gaining popularity.
- **KiloCore** (UC Davis, 2016). 1000 processors on a single chip.
- Bulk provides the same interface for programming the Epiphany co-processor as for programming distributed computer clusters! BSP algorithms can be used for this platform when modified slightly for streamed data\(^5\).

Epiphany BSP

- Parallella: powerful platform, especially for students and hobbyists. Suffers from poor tooling.
- Epiphany BSP, implementation of the BSPlib standard for the Parallella.
- Custom implementations for many rudimentary operations: memory management, printing, barriers.
Hello World: ESDK (124 LOC)

// host

const unsigned ShmSize = 128;
const char ShmName[] = "hello_shm";
const unsigned SeqLen = 20;

int main(int argc, char *argv[])
{
    unsigned row, col, coreid, i;
    e_platform_t platform;
    e_epiphany_t dev;
    e_mem_t mbuf;
    int rc;

    srand(1);

    e_set_loader_verbosity(H_D0);
    e_set_host_verbosity(H_D0);

    e_init(NULL);
    e_reset_system();
    e_get_platform_info(&platform);

    rc = e shm_alloc(&mbuf, ShmName, ShmSize);
    if (rc != E_OK)
        rc = e shm_attach(&mbuf, ShmName);
    // ...
Hello World: Epiphany BSP (18 LOC)

// host

#include <host_bsp.h>
#include <stdio.h>

int main(int argc, char** argv) {
    bsp_init("e_hello.elf", argc, argv);
    bsp_begin(bsp_nprocs());
    ebsp_spmd();
    bsp_end();
    return 0;
}

// kernel

#include <e_bsp.h>

int main() {
    bsp_begin();
    int n = bsp_nprocs();
    int p = bsp_pid();
    ebsp_printf("Hello world from core %d/%d", p, n);
    bsp_end();
    return 0;
}
BSP on low-memory

- Limited local memory, *classic* BSP programs can not run.
- Primary goal should be to minimize communication with external memory.
- Many known performance models can be applied to this system (EM-BSP, MBSP, Multi-BSP), no *portable way to write/develop algorithms.*
We view the Epiphany processor as a BSP computer with limited local memory of capacity $L$.

We have a shared external memory unit of capacity $E$, from which we can read data asynchronously with inverse bandwidth $e$.

Parameter pack: $(p, r, g, l, e, L, E)$. 
Parallella as a BSP accelerator

- $p = 16$, $p = 64$
- $r = (600 \times 10^6)/5 = 120 \times 10^6$ FLOPs$^(*)$
- $l = 1.00$ FLOP
- $g = 5.59$ FLOP/word
- $e = 43.4$ FLOP/word
- $L = 32$ kB
- $E = 32$ MB

$^(*)$: In practice one FLOP every 5 clockcycles, in theory up to 2 FLOPs per clockcycle.
External data access: streams

- **Idea**: present the input of the algorithm as **streams** for each core. Each stream consists of a number of **tokens**.
- The $i$th stream for the $s$th processor:
  \[ \Sigma^s_i = (\sigma_1, \sigma_2, \ldots, \sigma_n) \]
- Tokens fit in local memory: $|\sigma_i| < L$.
- We call the BSP programs that run on the tokens loaded on the cores **hypersteps**.
• In a hyperstep, while the computation is underway, the next tokens are loaded in (asynchronously).

• The time a hyperstep takes is either bounded by bandwidth or computation.

• Our cost function:

\[ \tilde{T} = \sum_{h=0}^{H-1} \max \left( T_h, e \sum_i C_i \right) . \]

Here, \( C_i \) is the token size of the \( i \)th stream, and \( T_h \) is the (BSP) cost of the \( h \)th hyperstep.
In video-streaming by default the video just ‘runs’. But viewer can skip ahead, rewatch portions. In this context referred to as pseudo-streaming.

Here, by default the next logical token is loaded in. But programmer can seek within the stream.

This minimizes the amount of code necessary for communication with external memory.

We call the resulting programs bulk-synchronous pseudo-streaming algorithms.
BSPlib extension for streaming

// host
void* bsp_stream_create(
    int processor_id,
    int stream_size,
    int token_size,
    const void* initial_data);

// kernel
int bsp_stream_open(int stream_id);
int bsp_stream_close(int stream_id);
int bsp_stream_move_down(
    int stream_id,
    void** buffer,
    int preload);

int bsp_stream_move_up(
    int stream_id,
    const void* data,
    int data_size,
    int wait_for_completion);

void bsp_stream_seek(
    int stream_id,
    int delta_tokens);
Example 1: Inner product

- **Input:** vectors \( \vec{v}, \vec{u} \) of size \( n \)
- **Output:** \( \vec{v} \cdot \vec{u} = \sum_i v_i u_i \).
Example 1: Inner product (cont.)

- **Input**: vectors $\vec{v}, \vec{u}$ of size $n$
- **Output**: $\vec{v} \cdot \vec{u} = \sum_i v_i u_i$.

1. Make a $p$-way distribution of $\vec{v}, \vec{w}$ (e.g. in blocks), resulting in subvectors $\vec{v}^{(s)}$ and $\vec{u}^{(s)}$.

2. These subvectors are then split into tokens that each fit in $L$. We have two streams for each core $s$:

   $\Sigma^s_v = \left((\sigma^s_{\vec{v}})_1, (\sigma^s_{\vec{v}})_2, \ldots, (\sigma^s_{\vec{v}})_H\right)$,

   $\Sigma^s_u = \left((\sigma^s_{\vec{u}})_1, (\sigma^s_{\vec{u}})_2, \ldots, (\sigma^s_{\vec{u}})_H\right)$.

3. Maintain a partial answer $\alpha_s$ throughout the algorithm, add $\left(\sigma^s_{\vec{v}}\right)_h \cdot \left(\sigma^s_{\vec{u}}\right)_h$ in the $h$th hyperstep. After the final tokens, sum over all $\alpha_s$. 
Example 2: Matrix multiplication

- **Input**: Matrices $A, B$ of size $n \times n$
- **Output**: $C = AB$

We decompose the (large) matrix multiplication into smaller problems that can be performed on the accelerator (with $N \times N$ cores). This is done by decomposing the input matrices into $M \times M$ outer blocks, where $M$ is chosen suitably large.

\[
AB = \begin{pmatrix}
A_{11} & A_{12} & \cdots & A_{1M} \\
A_{21} & A_{22} & \cdots & A_{2M} \\
\vdots & \vdots & \ddots & \vdots \\
A_{M1} & A_{M2} & \cdots & A_{MM}
\end{pmatrix}
\begin{pmatrix}
B_{11} & B_{12} & \cdots & B_{1M} \\
B_{21} & B_{22} & \cdots & B_{2M} \\
\vdots & \vdots & \ddots & \vdots \\
B_{M1} & B_{M2} & \cdots & B_{MM}
\end{pmatrix}
\]
Example 2: Matrix multiplication (cont.)

We compute the **outer blocks** of $C$ in row-major order. Since:

$$C_{ij} = \sum_{k=1}^{M} A_{ik} B_{kj},$$

a complete outer block is computed every $M$ hypersteps, where in a hyperstep we perform the multiplication of two outer blocks of $A$ and $B$.

Each block is again decomposed into **inner blocks** that fit into a core:

$$A_{ij} = \begin{pmatrix}
(A_{ij})_{11} & (A_{ij})_{12} & \ldots & (A_{ij})_{1N} \\
(A_{ij})_{21} & (A_{ij})_{22} & \ldots & (A_{ij})_{2N} \\
\vdots & \vdots & \ddots & \vdots \\
(A_{ij})_{N1} & (A_{ij})_{N2} & \ldots & (A_{ij})_{NN}
\end{pmatrix}. $$
Example 2: Matrix multiplication (cont.)

The streams for core \((s, t)\) are the inner blocks of \(A\) that belong to the core, laid out in row-major order, and the inner blocks of \(B\) in column-major order.

\[
\sum^A_{st} = \left( A_{11} \right)_{st} \left( A_{12} \right)_{st} \ldots \left( A_{1M} \right)_{st} \left( A_{21} \right)_{st} \left( A_{22} \right)_{st} \ldots \left( A_{2M} \right)_{st} \bigcirc M \text{ times} \\
\ldots \left( A_{M1} \right)_{st} \left( A_{M2} \right)_{st} \ldots \left( A_{MM} \right)_{st} \bigcirc M \text{ times}
\]

\[
\sum^B_{st} = \left( B_{11} \right)_{st} \left( B_{21} \right)_{st} \ldots \left( B_{M1} \right)_{st} \left( B_{12} \right)_{st} \left( B_{22} \right)_{st} \ldots \left( B_{M2} \right)_{st} \left( B_{13} \right)_{st} \ldots \left( B_{1M} \right)_{st} \left( B_{2M} \right)_{st} \ldots \left( B_{MM} \right)_{st} \bigcirc M \text{ times}
\]
In a hyperstep a suitable BSP algorithm (e.g. Cannon's algorithm) is used for the matrix multiplication on the accelerator.

We show that the cost function can be written as:

$$\tilde{T}_{\text{cannon}} = \max \left( 2 \frac{n^3}{N^2} + \frac{2Mn^2}{N} g + NM^3 l, \ 2 \frac{Mn^2}{N^2} e \right).$$
If you want to do your final project on something related to Epiphany BSP and/or Bulk, let me know!