Modern BSP: Scientific Computing, HPC, and 'Big Data'

Jan-Willem Buurlage, CWI, Amsterdam

Prepared for MasterMath course *Parallel Algorithms*, 05-10-2016

Outline

Modern parallel computing

- End of Moore's law
- Bulk

BSP on Exotic Systems

- Parallella
- Epiphany BSP
- Streams
- Examples

Modern parallel computing

Moore's law and the need for parallel

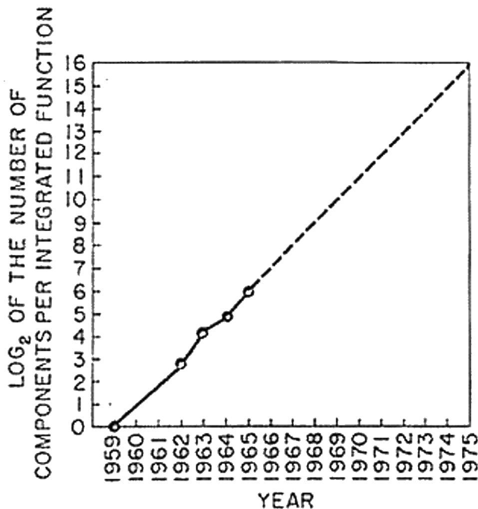


Fig. 2 Number of components per Integrated function for minimum cost per component extrapolated vs time.

Moore's law in recent years

- (1999) 'Physical limits reached for processor chips' [1]
- (2003) 'Intel scientists say the end of Moore's law is near' [2]
- (2006) 'Gordon Moore: Moore's law is no more' [3]
- (2016) 'Industry struggling to keep up with Moore's law' [4]

---

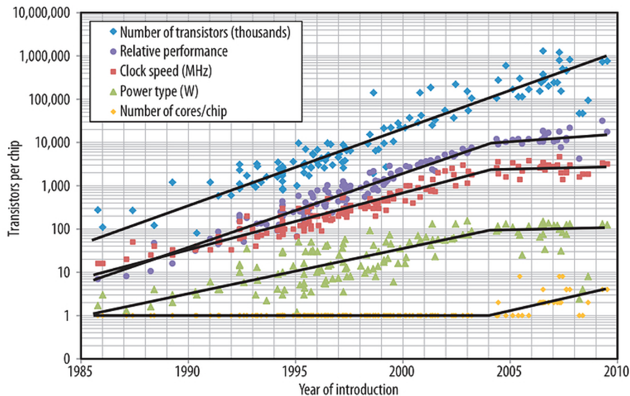
[1] https://tweakers.net/nieuws/5951/ontwikkelingen-chip-industrie-bereiken-fysieke-limieten.html

[2] https://tweakers.net/nieuws/29942/intel-wetenschappers-schetsen-het-einde-van-moores-law.html

[3] https://tweakers.net/nieuws/45230/gordon-moore-de-wet-van-moore-is-niet-meer.html

[4] https://tweakers.net/reviews/4787/is-de-wet-van-moore-dood.html

Moore's law in recent years (cont.)[5]



[5]Samuel H. Fuller; Lynette I. Millett: *Computing Performance: Game Over or Next Level?*, 2011

Formal models of parallel computing

- Parallel random access machine (PRAM)[6] allows to model parallel performance.
- The BSP model[7] builds on these ideas, puts restrictions on parallel algorithms, fundamental model for modern high-performance algorithms.
- *Others:*
  - Actor model (standard model in many modern programming languages)
  - LogP (similar to BSP)
  - Dataflow (big data, tensorflow)
  - Functional, skeleton models.

[6] Wyllie, James C. *The Complexity of Parallel Computations*, 1979
[7] Leslie G. Valiant. *A bridging model for parallel computation*, 1990

BSP today

- BSP is still the leading model for *distributed* computing. BSP is used throughout industry. The Big Data methodologies introduced by Google (MapReduce, Pregel) and used by all major tech companies (such as Facebook) are built on (restricted) BSP algorithms.
- However, BSPlib is not that popular (anymore). BSP programming usually happens through MPI or the various Apache projects *(Hama, Giraph, Hadoop)*. But it provides an accessible way to familiarize yourself with parallel programming.

Google's MapReduce[8]

- Array of **key-value pairs** $D_1 = [\{k_1, v_1\}, \ldots]$.
- The *map* M emits a *number of* **mapped key-value pairs**:

$$M(k_i, v_i) = [\{\tilde{k}_1, \tilde{v}_1\}, \{\tilde{k}_2, \tilde{v}_2\}, \ldots].$$

- The mapped key-value pairs form a new data set: $\bigcup_i M(k_i, v_i)$. Each pair in this set with the same key is gathered together, so that we obtain:

$$D_2 = [\{\tilde{k}_1, [v_{11}, v_{12}, \ldots]\}, \{\tilde{k}_2, [v_{21}, v_{22}, \ldots]\}, \ldots].$$

- The *reduce* R works on pairs in $\tilde{D}$ and usually emits a single value per pair leading to a final dataset D_3.

[8] MapReduce: Simplified Data Processing on Large Clusters, Jeffrey Dean and Sanjay Ghemawat (2004)

Google's MapReduce (Example)

- Classic example: word count. The map takes (file, content) pair, and emits (word, 1) pairs for each word in the content. The 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!

Google's Pregel

BSP for graph processing, used by Google[9] and Facebook[10]:

> *"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. During a superstep the framework invokes a user-defined function for each vertex, conceptually in parallel. The function specifies behavior at a single vertex V and a single superstep S. It can read messages sent to V in superstep $S - 1$, send messages to other vertices that will be received at superstep $S + 1$, and modify the state of V and its outgoing edges."*

[9] Pregel: A System for Large-Scale Graph Processing – Malewicz et al. (2010)

[10] One Trillion Edges: Graph Processing at Facebook-Scale - Avery Ching et al (2015).

Other parallel models

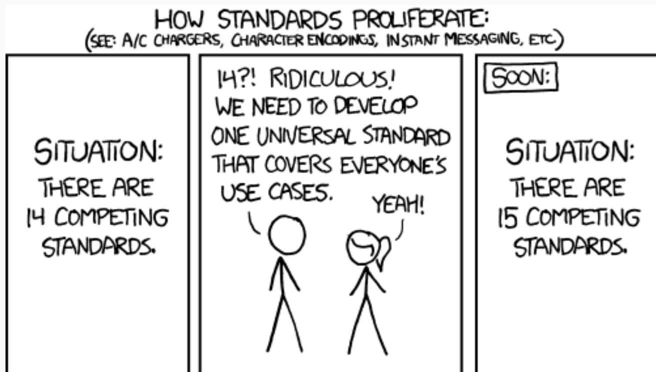
- Asynchronous, thread-based computing (primary way of programming for shared memory systems).
- 'Stream processing', used for GPGPU programming (BSP on coarse level).
- Functional programming, restrictive but often trivial to parallelize.

Libraries for parallel computing

- MPI
- CUDA
- OpenCL
- OpenMP
- Apache Hadoop, Apache Giraph, Apache Hama, ...
- Functional languages (Erlang, Haskell, ...)
- ...
- *BSPlib*

New library for modern parallel scientific computing: Bulk.

Bulk



5th most cited xkcd[11]

[11] https://xkcdref.info/statistics/

Introduction

- Not trying to replace directly any of the current systems
- These other libraries should simply not be used where they are used right now! Need a modern interface, user should not have to worry about the *transport layer*.
- In a way, BSPlib already improves upon MPI in this regard.

Goals

- Unified and *modern* interface for distributed and parallel computing.
- Works accross a wide variety of platforms, flexible *backends*.
- We support initially already:
  - multi-core desktop computers (on top of C++ threading support)
  - a many-core coprocessor (custom backend)
  - and distributed clusters (using MPI).

Why is it needed

- A single syntax for parallel application across platforms. There is no general library or interface for writing BSP programs for modern heterogeneous systems.
- Maintainable and generic code because of modern C++ constructs and idioms.
- Easy to write performant code, freedom to optimize.

What does it look like (cont.)

Communication using variables:

Bulk:

```
auto x = bulk::create_var<int>(world);

bulk::put(world.next_processor(), 1, x);
auto y = bulk::get(world.next_processor(), x);

world.sync();
auto value = y.value();
world.sync();
```

BSPlib:

```
int a = 0;
bsp_push_reg(&a, sizeof(int));
bsp_sync();

int b = s;
bsp_put((s + 1) % p, &b, &a, 0, sizeof(int));
bsp_sync();

a = s;
int b = 0;
bsp_get((s + 1) % p, &a, 0, &b, sizeof(int));
bsp_sync();
```

There is also a short-hand syntax in Bulk:

```
x(processor) = value;
auto y = x(processor).get();
```

What does it look like (cont.)

Co-arrays:

```
auto xs = bulk::create_coarray<int>(world, 5);
xs(3)[2] = 1;

world.sync();

// generic (local) container
int result = 0;
for (auto x : xs)
    result += x;
```

What does it look like (cont.)

Message queues :

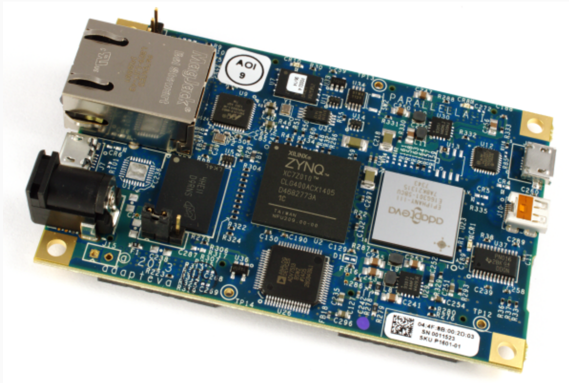
```
bulk::create_queue<int, int>(world);
q(world.next_processor()).send(1, 1);
q(world.next_processor()).send(2, 3);
q(world.next_processor()).send(123, 1337);

auto q2 = bulk::create_queue<int, float>(world);
q2(world.next_processor()).send(5, 2.1f);
q2(world.next_processor()).send(3, 4.0f);

world.sync();

// read queue
for (auto& msg : q) {
    std::cout << "the first queue received a message:" << msg.tag << ", "
              << msg.content << "\n";
}

for (auto& msg : q2) {
    std::cout << "the second queue received a message:" << msg.tag << ", "
              << msg.content << "\n";
}
```

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 (you can `std::sort` an incoming message queue).
- Works across more platforms than any competing library (because of the backend mechanism).
- Joint work with Tom Bannink (CWI). Open-source, MIT licensed. Tentative documentation at `http://www.codu.in/bulk/docs`.

BSP on Exotic Systems

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 $\mathrm{GFLOPs/W}$ (single precision), in 2011, top GPUs about $5\times$ 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[12].

[12] JB, Tom Bannink, Abe Wits. *Bulk-synchronous pseudo-streaming algorithms for many-core accelerators*. arXiv:1608.07200 [cs.DC], 2016

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)

```c
// 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
             );
    // ...
```

```c
// kernel

int main(void) {
    const char    ShmName[] = "hello_shm"
          ;
    const char          Msg[] = "Hello 
          World from core 0x%03x!";
    char                buf[256] = { 0 };
    e_coreid_t    coreid;
    e_memseg_t       emem;
    unsigned           my_row;
    unsigned           my_col;


    // Who am I? Query the CoreID from
          hardware.
    coreid = e_get_coreid();
    e_coords_from_coreid(coreid, &my_row
          , &my_col);

    if ( E_OK != e_shm_attach(&emem,
          ShmName) ) {
        return EXIT_FAILURE;
    }

    snprintf(buf, sizeof(buf), Msg,
          coreid);
    // ...
```

Hello World: Epiphany BSP (18 LOC)

```c
// 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;
}
```

```c
// 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**.

BSP accelerator

- 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

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

(*): 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 ith stream for the sth processor:

$$\Sigma_i^s = (\sigma_1, \sigma_2, \ldots, \sigma_n)$$

- Tokens fit in local memory: $|\sigma_i| < L$.
- We call the BSP programs that run on the tokens loaded on the cores hypersteps.

Structure of a program

- In a hyperstep, while the computation is underway, the next tokens are loaded in (asynchronously).
- The time a hyperstep takes is either **bound 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 ith stream, and T_h is the (BSP) cost of the hth hyperstep.

Pseudo-streaming

- 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

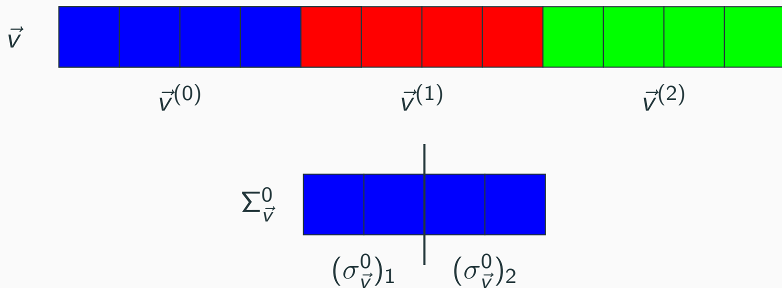
```
// host
void* bsp_stream_create(
        int processor_id,
        int stream_size,
        int token_size,
        const void* initial_data);

// kernel
int bsp_stream_open(int stream_id);
int bsp_stream_close(int stream_id);
```

BSPlib extension for streaming (2)

```
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:
$$\Sigma^s_{\vec{v}} = ((\sigma^s_{\vec{v}})_1, (\sigma^s_{\vec{v}})_2, \ldots, (\sigma^s_{\vec{v}})_H),$$
$$\Sigma^s_{\vec{u}} = ((\sigma^s_{\vec{u}})_1, (\sigma^s_{\vec{u}})_2, \ldots, (\sigma^s_{\vec{u}})_H).$$
3. Maintain a partial answer α_s throughout the algorithm, add $(\sigma^s_{\vec{v}})_h \cdot (\sigma^s_{\vec{u}})_h$ in the hth hyperstep. After the final tokens, sum over all α_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.

$$AB = \left(\begin{array}{c|c|c|c} A_{11} & A_{12} & \dots & A_{1M} \\ \hline A_{21} & A_{22} & \dots & A_{2M} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline A_{M1} & A_{M2} & \dots & A_{MM} \end{array}\right) \left(\begin{array}{c|c|c|c} B_{11} & B_{12} & \dots & B_{1M} \\ \hline B_{21} & B_{22} & \dots & B_{2M} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline B_{M1} & B_{M2} & \dots & B_{MM} \end{array}\right)$$

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:

$$A_{ij} = \left(\begin{array}{c|c|c|c} (A_{ij})_{11} & (A_{ij})_{12} & \ldots & (A_{ij})_{1N} \\ \hline (A_{ij})_{21} & (A_{ij})_{22} & \ldots & (A_{ij})_{2N} \\ \hline \vdots & \vdots & \ddots & \vdots \\ \hline (A_{ij})_{N1} & (A_{ij})_{N2} & \ldots & (A_{ij})_{NN} \end{array}\right).$$

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.

$$\Sigma^A_{st} = \underbrace{(A_{11})_{st}(A_{12})_{st}\dots(A_{1M})_{st}}_{\circlearrowright\ M\ \text{times}}\ \underbrace{(A_{21})_{st}(A_{22})_{st}\dots(A_{2M})_{st}}_{\circlearrowright\ M\ \text{times}}$$

$$\dots\underbrace{(A_{M1})_{st}(A_{M2})_{st}\dots(A_{MM})_{st}}_{\circlearrowright\ M\ \text{times}},$$

$$\Sigma^B_{st} = (B_{11})_{st}(B_{21})_{st}\dots(B_{M1})_{st}(B_{12})_{st}(B_{22})_{st}$$

$$\underbrace{\dots(B_{M2})_{st}(B_{13})_{st}\dots(B_{1M})_{st}(B_{2M})_{st}\dots(B_{MM})_{st}}_{\circlearrowright\ M\ \text{times}}.$$

Example 2: Matrix multiplication (cont.)

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:

$$\tilde{T}_{\text{cannon}} = \max\left(2\frac{n^3}{N^2} + \frac{2Mn^2}{N}g + NM^3 l,\ 2\frac{Mn^2}{N^2}e\right).$$

Thanks

If you want to do your final project on something related to Epiphany BSP and/or Bulk, let me know!