Bulk-synchronous pseudo-streaming for many-core accelerators

Jan-Willem Buurlage\textsuperscript{1}  Tom Bannink\textsuperscript{1,2}  Abe Wits\textsuperscript{3}

\textsuperscript{1}Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands

\textsuperscript{2}QuSoft, Amsterdam, The Netherlands

\textsuperscript{3}Utrecht University, The Netherlands
Overview

Parallella

Epiphany BSP

Extending BSP with streams

Examples

  Inner product

  Matrix multiplication

  Sort
Parallella
‘A supercomputer for everyone, with the lofty goal of democratizing access to parallel computing’

• Crowd-funded development board, raised almost $1M in 2012.
• ‘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), each with limited local memory.
- 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 co-processor

- $N \times N$ grid of RISC processors, clocked by default at 600 MHz (current generations have 16 or 64 cores), each with limited local memory.
- 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.
• $N \times N$ grid of RISC processors, clocked by default at 600 MHz (current generations have 16 or 64 cores), each with limited local memory.

• 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 $5 \times$ less efficient.
Epiphany memory

- Each Epiphany core has 32 kB of local memory, on 16-core model 512 kB available in total. There are no caches.
- On each core, the kernel binary and stack already take up a large section of this memory.
- 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.
Epiphany memory

• Each Epiphany core has 32 kB of local memory, on 16-core model 512 kB available in total. There are no caches.
• On each core, the kernel binary and stack already take up a large section of this memory.
• 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.
Epiphany memory

- Each Epiphany core has 32 kB of **local memory**, on 16-core model 512 kB available in total. There are no caches.
- On each core, the kernel binary and stack already take up a large section of this memory.
- 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.
- There are also specialized (co)processors on the market for e.g. machine learning, computer vision.
- *KiloCore* (UC Davis, 2016). **1000 processors** on a single chip.
Many-core co-processors

- **Applications**: Mobile, Education, possibly even HPC.
- There are also specialized (co)processors on the market for e.g. machine learning, computer vision.
Many-core co-processors

- **Applications**: Mobile, Education, possibly even HPC.
- There are also specialized (co)processors on the market for e.g. machine learning, computer vision.
- *KiloCore* (UC Davis, 2016). **1000 processors** on a single chip.
Epiphany BSP
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.
• 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.
• 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_loaderverbosity(H_D0);
    e_set_hostverbosity(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);
    // ...

// kernel

int main(void) {
    const char ShmName[] = "hello_shm";
    const char Msg[] = "Hello World from core_0x%03x!";
    char buf[256] = { 0 };  
coreid;  
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)

// host

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

int main(int argc, char** argv) {
    bsp_init("e_hello.e1f", argc, argv);
    bsp_begin(bsp_nprocs());
    bsp_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 computers

- The BSP model [Valiant, 1990] describes a general way to perform parallel computations.
- An abstract BSP computer is associated to the model that has $p$ processors, which all have access to a communication network.
• The BSP model [Valiant, 1990] describes a general way to perform parallel computations.

• An abstract BSP computer is associated to the model that has $p$ processors, which all have access to a communication network.
BSP programs consist of a number of supersteps, that each have a computation phase, and a communication phase. Each superstep is followed by a barrier synchronisation.

- Each processor on a BSP computer has a processing rate $r$. It has two parameters: $g$, related to the communication speed, and $l$ the latency.
- The running time of a BSP program can be expressed in terms of these parameters! We denote this by $T(g, l)$. 
BSP programs consist of a number of supersteps, that each have a computation phase, and a communication phase. Each superstep is followed by a barrier synchronisation.

Each processor on a BSP computer has a processing rate $r$. It has two parameters: $g$, related to the communication speed, and $l$ the latency.

The running time of a BSP program can be expressed in terms of these parameters! We denote this by $T(g, l)$. 
• BSP programs consist of a number of supersteps, that each have a computation phase, and a communication phase. Each superstep is followed by a barrier synchronisation.

• Each processor on a BSP computer has a processing rate $r$. It has two parameters: $g$, related to the communication speed, and $l$ the latency.

• The running time of a BSP program can be expressed in terms of these parameters! We denote this by $T(g, l)$. 
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 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 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: $(p, r, g, l, e, L, E)$. 
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: $(p, r, g, l, e, L, E)$.
• 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.
Extending BSP with streams
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**.
**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**.

---

**External data access: streams**
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**.
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:

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

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

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 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.

• 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.
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.
- 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.
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.
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.
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.
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);
void 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);
Examples
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 $\mathbf{v}, \mathbf{u}$ of size $n$
- **Output**: $\mathbf{v} \cdot \mathbf{u} = \sum_i v_i u_i$.

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

2. These subvectors are then split into tokens that each fit in $L$.
   We have two streams for each core $s$:
   
   $$\Sigma^s_v = ((\sigma^s_v)_1, (\sigma^s_v)_2, \ldots, (\sigma^s_v)_H),$$
   
   $$\Sigma^s_u = ((\sigma^s_u)_1, (\sigma^s_u)_2, \ldots, (\sigma^s_u)_H).$$

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

   \[
   \sum_\vec{v}^s = ((\sigma_\vec{v}^s)_1, (\sigma_\vec{v}^s)_2, \ldots, (\sigma_\vec{v}^s)_H),
   \]
   \[
   \sum_\vec{u}^s = ((\sigma_\vec{u}^s)_1, (\sigma_\vec{u}^s)_2, \ldots, (\sigma_\vec{u}^s)_H).
   \]

3. Maintain a partial answer $\alpha_s$ throughout the algorithm, add $(\sigma_\vec{v}^s)_h \cdot (\sigma_\vec{u}^s)_h$ in the $h$th hyperstep. After the final tokens, sum over all $\alpha_s$. 

21
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.

\[
AB = \begin{pmatrix}
A_{11} & A_{12} & \ldots & A_{1M} \\
A_{21} & A_{22} & \ldots & A_{2M} \\
\vdots & \vdots & \ddots & \vdots \\
A_{M1} & A_{M2} & \ldots & A_{MM}
\end{pmatrix}
\begin{pmatrix}
B_{11} & B_{12} & \ldots & B_{1M} \\
B_{21} & B_{22} & \ldots & B_{2M} \\
\vdots & \vdots & \ddots & \vdots \\
B_{M1} & B_{M2} & \ldots & B_{MM}
\end{pmatrix}
\]
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 one outer blocks of $A$, and one of $B$.

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

$$A_{ij} = \begin{pmatrix}
(A_{ij})_{11} & (A_{ij})_{12} & \cdots & (A_{ij})_{1N} \\
(A_{ij})_{21} & (A_{ij})_{22} & \cdots & (A_{ij})_{2N} \\
\vdots & \vdots & \ddots & \vdots \\
(A_{ij})_{N1} & (A_{ij})_{N2} & \cdots & (A_{ij})_{NN}
\end{pmatrix}.$$
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 one outer blocks of $A$, and one of $B$.

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

$$A_{ij} = \begin{pmatrix}
(A_{ij})_{11} & (A_{ij})_{12} & \cdots & (A_{ij})_{1N} \\
(A_{ij})_{21} & (A_{ij})_{22} & \cdots & (A_{ij})_{2N} \\
\vdots & \vdots & \ddots & \vdots \\
(A_{ij})_{N1} & (A_{ij})_{N2} & \cdots & (A_{ij})_{NN}
\end{pmatrix}. $$
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 one outer blocks of \( A \), and one of \( B \).

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

\[
A_{ij} = \begin{pmatrix}
(A_{ij})_{11} & (A_{ij})_{12} & \cdots & (A_{ij})_{1N} \\
(A_{ij})_{21} & (A_{ij})_{22} & \cdots & (A_{ij})_{2N} \\
\vdots & \vdots & \ddots & \vdots \\
(A_{ij})_{N1} & (A_{ij})_{N2} & \cdots & (A_{ij})_{NN}
\end{pmatrix}.
\]
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.

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

\[
\sum_{st}^B = \left( (B_{11})_{st} (B_{21})_{st} \ldots (B_{M1})_{st} (B_{12})_{st} (B_{22})_{st} \right) \odot M \text{ times} \\
\cdots (B_{M1})_{st} (B_{M2})_{st} \ldots (B_{1M})_{st} (B_{2M})_{st} \ldots (B_{MM})_{st} \odot M \text{ times}
\]
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.

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

\[
\sum^B_{st} = (B_{11})_{st} (B_{21})_{st} \ldots (B_{M1})_{st} (B_{22})_{st} \\
\quad \circ M \times \text{times} \\
\quad \ldots (B_{M2})_{st} (B_{13})_{st} \ldots (B_{1M})_{st} (B_{2M})_{st} \ldots (B_{MM})_{st}.
\]

\[
\quad \circ M \times \text{times}
\]
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:

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

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

- **Input**: An array $A$ of comparable objects.
- **Output**: The sorted array $\tilde{A}$.

1. **Parallel bucket sort**: create $p$ buckets, put each element of $A$ in the appropriate bucket, let the $s$th core sort the $s$th bucket.
2. **Sample sort** samples elements of $A$ in order to balance the buckets.
Example 3: Sorting

- **Input**: An array $A$ of comparable objects.
- **Output**: The sorted array $\tilde{A}$.

1. Parallel bucket sort: create $p$ buckets, put each element of $A$ in the appropriate bucket, let the $s$th core sort the $s$th bucket.
2. Sample sort samples elements of $A$ in order to balance the buckets.
1. Split the input array to create $p$ equally sized streams. Also create $p$ initially empty streams that will be the buckets.

2. We adapt the sample sort algorithm, first we need to find the buckets, which is **Phase 1** of our algorithm.

3. Each core samples $k$ elements randomly from its stream. We do this using a classic streaming algorithm called **reservoir sampling**. These samples are then sorted.
1. Split the input array to create $p$ equally sized streams. Also create $p$ initially empty streams that will be the buckets.

2. We adapt the sample sort algorithm, first we need to find the buckets, which is **Phase 1** of our algorithm.

3. Each core samples $k$ elements randomly from its stream. We do this using a classic streaming algorithm called *reservoir sampling*. These samples are then sorted.
1. Split the input array to create $p$ equally sized streams. Also create $p$ initially empty streams that will be the buckets.

2. We adapt the sample sort algorithm, first we need to find the buckets, which is **Phase 1** of our algorithm.

3. Each core samples $k$ elements randomly from its stream. We do this using a classic streaming algorithm called *reservoir sampling*. These samples are then sorted.
Each core chooses $p$ equally spaced elements and sends these to the first core.

The first core sorts its $p^2$ values, and chooses $p - 1$ equally spaced *global splitters*

The global splitters are communicated to the other cores, and define the bucket boundaries.
Each core chooses \( p \) equally spaced elements and sends these to the first core.

The first core sorts its \( p^2 \) values, and chooses \( p - 1 \) equally spaced *global splitters*

The global splitters are communicated to the other cores, and define the bucket boundaries.
Each core chooses $p$ equally spaced elements and sends these to the first core.

The first core sorts its $p^2$ values, and chooses $p - 1$ equally spaced *global splitters*.

The global splitters are communicated to the other cores, and define the bucket boundaries.
• In **Phase 2** of the algorithm we fill the buckets with data.

  • In a hyperstep, we run a BSP sort on the current tokens. Next, each core will have consecutive elements that can be sent to the correct buckets efficiently.

  • These buckets are the $p$ additional streams that were created, which were initially empty.
• In **Phase 2** of the algorithm we fill the buckets with data.

• In a hyperstep, we run a BSP sort on the current tokens. Next, each core will have consecutive elements that can be sent to the correct buckets efficiently.

• These buckets are the $p$ additional streams that were created, which were initially empty.
• In **Phase 2** of the algorithm we fill the buckets with data.

• In a hyperstep, we run a BSP sort on the current tokens. Next, each core will have consecutive elements that can be sent to the correct buckets efficiently.

• These buckets are the $p$ additional streams that were created, which were initially empty.
• In Phase 3, the $s$th core sorts the $s$th bucket stream using an external sort algorithm.

• We use a merge sort variant for this.
In Phase 3, the s<sup>th</sup> core sorts the s<sup>th</sup> bucket stream using an external sort algorithm.

We use a merge sort variant for this.
Summary

- Parallella and the Epiphany: great platform for BSP.
- Pseudo-streaming algorithms are a convenient way to think about algorithms for this platform.
- Can often (re)use BSP algorithms, and generalize them to this streaming framework, even if local memory is limited.
Thank you for your attention. Questions?
Sources

1. Parallella, Adapteva Epiphany:  
   http://www.adapteva.org
2. Epiphany BSP: http://www.codu.in/ebsp