Základní paralelní operace

Tomáš Oberhuber
tomas.oberhuber@fjfi.cvut.cz

15. dubna 2024

Videa na Youtube:

Redukce

Prefix sum

Základní paralelní operace

Guy E. Blelloch, Vector Models for Data-Parallel Computing, Cambridge: MIT press, 1990.

Co je paralelní redukce?

Definition

Mějme pole prvků $a_1, \ldots, a_n$ určitého typu a asociativní operaci $\oplus$. **Paralelní redukce** je paralelní aplikace asociativní operace $\oplus$ na všechny prvky vstupního pole tak, že výsledkem je jeden prvek a stejného typu, pro který platí

$$a = a_1 \oplus a_2 \ldots a_n.$$

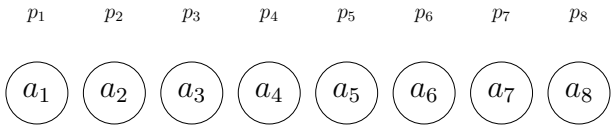
- porovnání polí, vektorů a řetězců
- skalární součin dvou vektorů
- l_p normy
- minimalální maximalní hodnota
- suma všech prvků pole

Co je paralelní redukce?

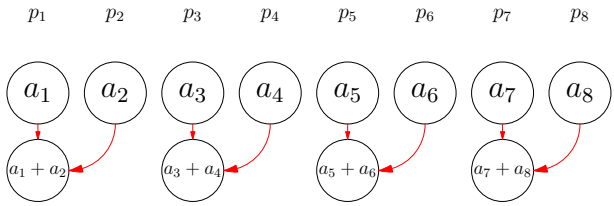
Příklad: Pro jednoduchost budeme uvažovat operaci sčítaní. Pak jde o paralelní provedení tohoto kódu:

```
a = a[ 1 ];
for( int i = 2; i <= n; i ++ )
    a += a[ i ];
```

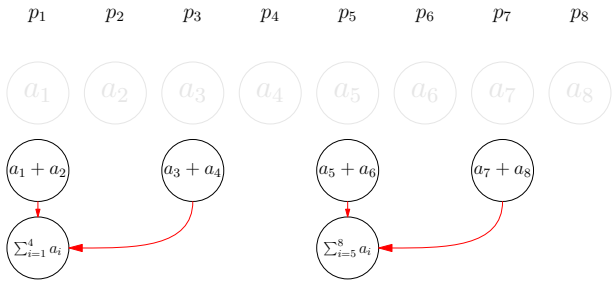
Paralelní provedení



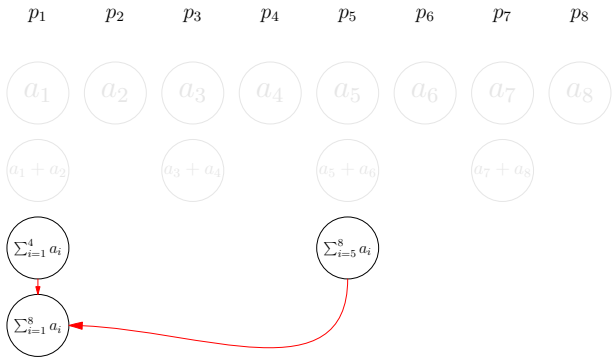
Paralelní provedení



Paralelní provedení



Paralelní provedení



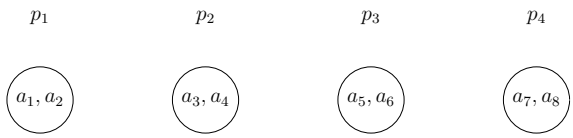
Analýza paralelní provedení

Za předpokladu $n = p$, snadno vidíme, že platí:

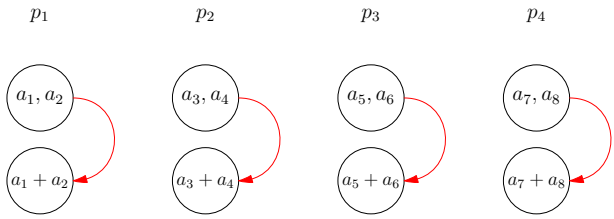
- $T_S(n) = \theta(n)$
- $T_P(n) = \theta(\log n)$
- $S(n) = \theta\left(\frac{n}{\log n}\right) = O(n) = O(p)$
- $E(n) = \theta\left(\frac{1}{\log n}\right)$
  - pro velká n efektivita klesá k nule
- $C(n) = \theta\left(n \log n\right) = \omega\left(T_S(n)\right)$
  - algoritmus není nákladově optimální

Provedeme úpravu algoritmu tak, aby byl nákladově optimální.

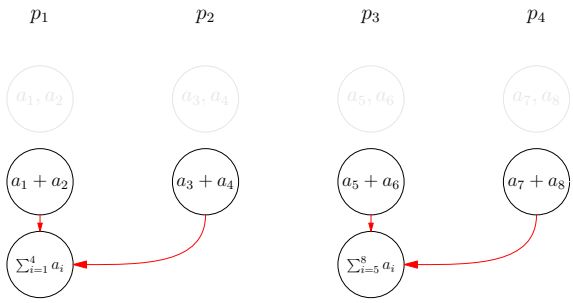
Nákladově optimální redukce



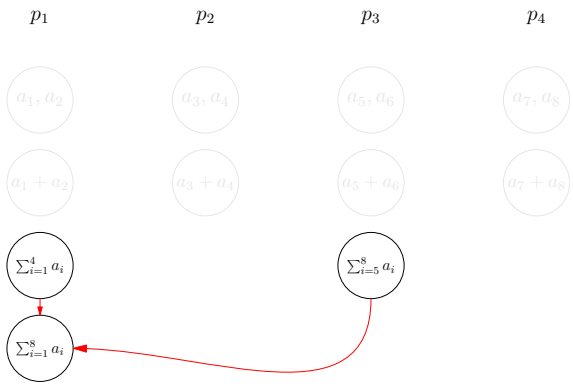
Nákladově optimální redukce



Nákladově optimální redukce



Nákladově optimální redukce



Nákladově optimální redukce

- nyní máme méně procesorů než prvků, tj. $p < n$.
- nejprve se provede sekvenční redukce $\frac{n}{p}$ prvků na každém procesoru$\rightarrow \theta\left(\frac{n}{p}\right)$
- a pak paralelní redukce na p procesorech $\rightarrow \theta\left(\log p\right)$
- celkem tedy $T_P\left(n,p\right) = \theta\left(\frac{n}{p} + \log p\right)$
- $S(n,p) = \frac{T_S(n)}{T_P(n,p)} = \theta\left(\frac{n}{\frac{n}{p}+\log p}\right) = \theta\left(\frac{p}{1+\frac{p\log p}{n}}\right)$
- $E(n,p) = \frac{S(n,p)}{p} = \theta\left(\frac{1}{1+\frac{p\log p}{n}}\right)$

Nákladově optimální redukce

Pak platí

- $C(n,p) = pT_P(n,p) = \theta\left(n + p\log p\right)$
- protože je $n = \Omega\left(p\log p\right)$, máme

$$C(n,p) = \theta(n) = \theta(T_S(n)).$$

Algoritmus je tedy nákladově optimální.

- toho jsme dosáhli na úkor počtu procesorů
- některé algoritmy nedokáží optimálně využít velký počet procesorů

Paralelní redukce na architekturách se sdílenou pamětí

- OpenMP má podporu pro redukci s základními operacemi – `reduction(sum:+)`
- pokud chceme dělat redukci s jinou operací (`min, max`) nebo jiným, než základním typem (matice), je nutné ji provést explicitně
- tyto architektury mají většinou maximálně několik desítek procesorů
- redukci lze často provádět sekvenčně
- mezivýsledky se ukládají do nesdílených proměnných
  - pozor na cache coherence a false sharing - tyto proměnné by neměly být alokované v jednom bloku
- nakonec se zapíší do sdíleného pole, na kterém se provede redukce třeba i sekvenčně nultým procesem

Paralelní redukce na architekturách s distribuovanou pamětí

- standard MPI má velmi dobrou podporu pro redukci
- podporuje více operací a to i pro odvozené typy a struktury

Paralelní redukce na GPU v CUDA

M. Harris, Optimizing Parallel Reduction in CUDA, Nvidia.

Paralelní redukce na GPU v CUDA

```
template <unsigned int blockSize>
__global__ void cudaDeviceReduceKernel(int size, int* dInput, int* dOutput)
{
    extern __shared__ volatile int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int gid = blockIdx.x*blockDim.x + threadIdx.x;
    unsigned int gridSize = blockDim.x*gridDim.x;
    sdata[tid] = 0;
    while (gid < size)
    {
        sdata[tid] += dInput[gid] + dInput[gid+blockSize];
        gid += gridSize;
    }
    __syncthreads();

    if (blockSize >= 1024) {
        if( tid<512 ) { sdata[ tid ] += sdata[ tid+512 ]; } __syncthreads(); }
    if (blockSize >= 512) {
        if( tid<256 ) { sdata[ tid ] += sdata[ tid+256 ]; } __syncthreads(); }
    if (blockSize >= 256) {
        if( tid<128 ) { sdata[ tid ] += sdata[ tid+128 ]; } __syncthreads(); }
    if (blockSize >= 128) {
        if( tid< 64 ) { sdata[ tid ] += sdata[ tid+ 64 ]; } __syncthreads(); }
    if (tid < 32) {
        if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
        if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
        if (blockSize >= 16) sdata[tid] += sdata[tid + 8];
        if (blockSize >= 8) sdata[tid] += sdata[tid + 4];
        if (blockSize >= 4) sdata[tid] += sdata[tid + 2];
        if (blockSize >= 2) sdata[tid] += sdata[tid + 1];
    }


    if (tid == 0) dOutput[blockIdx.x] = sdata[0];
}

bool cudaGlobReduce(int const size,
const int* deviceData,
const int blockSize,
const int deviceGridSize,
int* result)
{
    const int elementsPerBlock = 2 * blockSize;
    int gridSize = size / elementsPerBlock + ( size % elementsPerBlock != 0 );
    gridSize = gridSize < deviceGridSize ? gridSize : deviceGridSize;
    int* deviceOutput = nullptr; sizeof( int );
    int* deviceInput = deviceData;
    if( cudaMalloc( ( void** ) ( &deviceOut), sizeof(int) ) != cudaSuccess )
    cudaMalloc( ( void** ) ( &deviceOut), sizeof(int) ) != cudaSuccess )
    {
        cerr << "Not enough memory on device." << endl;
        return false;
    }

    int currentSize = size;
    const int* reductionInput = deviceData;
    while( currentSize > 1 )
    {
        dim3 cudaGridDim( 1, cudaGridSize( ) );
        cudaBlockDim.x = blockSize;
        gridDim = currentSize < elementsInBlock *
        ( currentSize % elementsInBlock != 0 );
        cudaGridDim.x = gridDim < deviceGridSize ? gridDim : deviceGridSize;

        size_t sharedMemory = blockSize * sizeof( int );
        switch( blockSize )
        {
            case 256:
            cudaBlockReduce<256 >>
            ( currentSize, cudaBlockDim, sharedMemory >>>
            ( currentSize, reductionInput, deviceOut );
            break;
            break;
        }

        cudaError error = cudaGetLastError();
        if( error != cudaSuccess )
        {
            cerr << "The CUDA kernel ended with error code " << error << "."
            << endl;
            return false;
        }

        currentSize = cudaGridDim. x;

        reductionInput = deviceOut;
        int* tmp = deviceOut;
        deviceOut = deviceOut;
        deviceOut = tmp;
    }
    if( cudaMemcpy( ( result,
reductionInput,
sizeof( int ),
cudaMemcpyDeviceToHost ) != cudaSuccess )
    {
        cerr << "An error occured while transfering data from device to host."
        << endl;
        return false;
    }
    cudaFree( deviceOut );
    cudaFree( deviceOut );
    return true;
```

Paralelní redukce v TNL

Jako příklad si vezmeme skalární součin:

```
float result( 0.0 );
for( int i = 0; i < size; i++ )
   result += a[ i ] * b[ i ];
```

Nyní ho přepíšeme pomocí lambda funkcí:

```
auto fetch = [=] __cuda_callable__ (int i)->float { return a[i]*b[i]; };
auto reduction = [] __cuda_callable__ (float x, float y) -> float {
    return x+y; };

float result( 0.0 );
for( int i = 0; i < size; i++ )
  reduction = reduction( result, fetch( i ) );
```

Paralelní redukce v TNL

V TNL, pak for cyklus nahradíme funkcí `reduction`:

```
1  auto a_view = a.getView();
2  auto b_view = b.getView();
3  auto fetch = [=] __cuda_callable__ ( int i)->float {
4          return a_view[ i ] * b_view[ i ]; };
5  auto reduction = [] __cuda_callable__ (float x, float y) -> float {
6     return x + y; };
7
8  result = TNL::Algorithms::reduce< Device >( 0,
9                                              a.getSize(),
10                                             fetch,
11                                             reduction,
12                                             0.0 );
```

Poslední parametr je identický element pro danou operaci.

Paralelní redukce v TNL

Porovnání dvou vektorů pak může být provedeno takto:

```
auto a_view = a.getView();
auto b_view = b.getView();
auto fetch = [=] __cuda_callable__ (int i)->bool {
        return a_view[ i ] == b_view[ i ]; };
auto reduction = [] __cuda_callable__ (float x, float y)->float {
   return x && y; };

result = TNL::Algorithms::reduce< Device >( 0,
                                            a.getSize(),
                                            fetch,
                                            reduction,
                                            true );
```

Paralelní redukce v TNL

Největší prvek v absolutní hodnotě lze najít takto:

```
auto a_view = a.getView();
auto fetch = [=] __cuda_callable__ (int i)->float {
   return abs( a_view[ i ] ); };
auto reduction = [] __cuda_callable__ (float x, float y)->float {
   return max( x, y ); };

result = TNL::Algorithms::reduce< Device >( 0,
                                  a.getSize(),
                                  fetch,
                                  reduction,
                                  std::numeric_limits< float >::lowest() );
```

Nebo kompatkněji:

```
auto a_view = a.getView();
result = TNL::Algorithms::reduce< Device >( 0, a.getSize(),
    [=] __cuda_callable__ (int i)->float { return abs( a_view[ i ] ); },
    TNL::Max() );
```

Paralelní redukce v TNL

Nalezení největšího prvku v absolutní hodnotě:

```
template< typename Device >
std::pair< double, int >
maximumNorm( const Vector< double, Device >& v )
{
   auto view = v.getConstView();

   auto fetch = [=] __cuda_callable__ ( int i ) {
      return abs( view[ i ] ); };
   auto reduction = [] __cuda_callable__ ( double& a,
                                           const double& b,
                                           int& aIdx,
                                           const int& bIdx ) {
      if( a < b ) {
         a = b;
         aIdx = bIdx;
      }
      else if( a == b && bIdx < aIdx )
         aIdx = bIdx;
   };
   return reduceWithArgument< Device >( 0, view.getSize(),
                                        fetch, reduction,
                                        std::numeric_limits< double >::max() );
}
```

Paralelní redukce v TNL

Volání předchozí funkce vypadá takto:

```
int main( int argc, char* argv[] )
{
   Vector< double, Devices::Host > host_v( 10 );
   host_v.forAllElements( [] __cuda_callable__ ( int i, double& value ) {
      value = i - 7; } );
   std::cout << "host_v = " << host_v << std::endl;
   auto maxNormHost = maximumNorm( host_v );
   std::cout << "The maximum norm of the host vector elements is "
             <<  maxNormHost.first
             << " at position "
             << maxNormHost.second << "." << std::endl;

#ifdef __CUDACC__
   Vector< double, Devices::Cuda > cuda_v( 10 );
   cuda_v.forAllElements( [] __cuda_callable__ ( int i, double& value ) {
      value = i - 7; } );
   std::cout << "cuda_v = " << cuda_v << std::endl;
   auto maxNormCuda = maximumNorm( cuda_v );
   std::cout << "The maximum norm of the device vector elements is "
             <<  maxNormCuda.first
             << " at position "
             << maxNormCuda.second << "." << std::endl;
#endif
   return EXIT_SUCCESS;
}
```

Paralelní redukce v TNL

Také je možné spojit dvě operace do jedné:

- přičtení vektoru
- výpočet normy tohoto vektoru.

Toto se často vyskytuje v Rungových-Kuttových metodách:

```
auto a_view = a.getView();
auto b_view = b.getView();
result = TNL::Algorithms::reduce< Device >( 0, a.getSize(),
   [=] __cuda_callable__ (int i)->float mutable {
            // mutable because we change a_view
            float update = 0.5 * ( a_view[ i ] + b_view[ i ];
            a_view[ i ] += update;
            return abs( update ); },
    TNL::Plus() );
```

Paralelní redukce v Thrust

```cpp
#include <thrust/reduce.h>
#include <thrust/device_vector.h>
#include <iostream>

int main() {
    // Example data
    thrust::device_vector<int> data{1, 2, 3, 4, 5};

    // Perform reduction
    int sum = thrust::reduce(data.begin(),
                             data.end(),
                             0,
                             thrust::plus<int>());

    std::cout << "Sum: " << sum << std::endl;

    return 0;
}
```

Paralelní redukce v Kokkos

```cpp
#include <Kokkos_Core.hpp>
#include <iostream>

int main(int argc, char* argv[]) {
    Kokkos::initialize(argc, argv);
    {
        int N = 5;
        Kokkos::View<int*> data("data", N);
        Kokkos::parallel_for("InitData", N, KOKKOS_LAMBDA(const int i) {
            data(i) = i + 1;
        });

        int sum = 0;
        // Use a lambda for the reduction
        Kokkos::parallel_reduce("SumReduce", N,
            KOKKOS_LAMBDA(const int i, int& lsum) {
                lsum += data(i);
            },
            sum);

        std::cout << "Sum: " << sum << std::endl;
    }
    Kokkos::finalize();

    return 0;
}
```

Paralelní redukce v SYCL

```
#include <CL/sycl.hpp>

void reduction( sycl::buffer<int>& inputBuffer,
                sycl::buffer<int>& outputBuffer )
{
   sycl::queue q;

   q.submit([&](sycl::handler& cgh) {
      auto inputData =
         inputBuffer.get_access<sycl::access::mode::read>(cgh);
      auto outputData =
         outputBuffer.get_access<sycl::access::mode::write>(cgh);

      cgh.parallel_for<class ReductionKernel>(
         sycl::range<1>(inputBuffer.get_count()),
         [=](sycl::item<1> item) {
            int localResult = inputData[item];
            localResult =
               sycl::ONEAPI::wgroup_reduce<int,sycl::ONEAPI::plus<int>>(
                  localResult,
                  cgh);

            if (item.get_local_id(0) == 0) {
                outputData[item.get_group(0)] = localResult;
            }
        } );
    } );
}
```

Paralelní redukce a MapReduce v Hadoop

*"Hadoop je framework obsahující sadu opensource softwarových komponent určených pro zpracování velkého množství nestrukturovaných a **distribuovaných** dat v řádech petabytů a exabytů."*[1]

Obsahuje tyto nástroje:

- **Hadoop Distributed File System (HDFS)** - distribuovaný filesystem.
- **MapReduce** - programovací model pro paralelní zpracování velkých datových sad na distribuyovaných klastrech.
- **YARN (Yet Another Resource Negotiator)** - distribuovaná plánovač úloh.

Ukážeme se příklad na MapReduce v Javě.

[1] Zdroj Wikipedie.

Paralelní redukce a MapReduce v Hadoop

Nejprve implementujteme `Mapper` ≡ obdoba funkce `fetch` v TNL.

```
1  import org.apache.hadoop.io.IntWritable;
2  import org.apache.hadoop.io.LongWritable;
3  import org.apache.hadoop.io.Text;
4  import org.apache.hadoop.mapreduce.Mapper;
5
6  import java.io.IOException;
7
8  public class WordCountMapper
9      extends Mapper<LongWritable, Text, Text, IntWritable> {
10
11     private final static IntWritable one = new IntWritable(1);
12     private Text word = new Text();
13
14     public void map(LongWritable key, Text value, Context context)
15         throws IOException, InterruptedException {
16             String[] words = value.toString().split("\\s+");
17             for (String str : words) {
18                 word.set(str);
19                 context.write(word, one);
20             }
21     }
22 }
```

- `Mapper` čte vstupní text `value`, dělí ho na slova a na každé slovo `word` vrací dvojici klíč-hodnota ve tvaru `word-1`.

Paralelní redukce a MapReduce v Hadoop

Dále implementujteme `Reducer` ≡ obdoba funkce `reduce` v TNL.

```
1  import org.apache.hadoop.io.IntWritable;
2  import org.apache.hadoop.io.Text;
3  import org.apache.hadoop.mapreduce.Reducer;
4
5  import java.io.IOException;
6
7  public class WordCountReducer
8      extends Reducer<Text, IntWritable, Text, IntWritable> {
9
10     private IntWritable result = new IntWritable();
11
12     public void reduce(Text key,
13                        Iterable<IntWritable> values,
14                        Context context)
15                 throws IOException, InterruptedException {
16         int sum = 0;
17         for (IntWritable val : values) {
18             sum += val.get();
19         }
20         result.set(sum);
21         context.write(key, result);
22     }
23 }
```

- `Reducer` dostává pro jednotlivé klíče množinu hodnot tak, je generoval `Mapper`. S hodnotami provede redukci.

Paralelní redukce a MapReduce v Hadoop

Nakonec implementujeme `Driver`, který inicializuje zpracování dat.

```java
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class WordCount {
    public static void main(String[] args) throws Exception {
        if (args.length != 2) {
            System.err.println("Usage: WordCount <input path> <output path>");
            System.exit(-1);
        }

        Configuration conf = new Configuration();
        Job job = Job.getInstance(conf, "word count");
        job.setJarByClass(WordCount.class);
        job.setMapperClass(WordCountMapper.class);
        job.setCombinerClass(WordCountReducer.class);
        job.setReducerClass(WordCountReducer.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Paralelní redukce a MapReduce v Hadoop

Příklad pak lze spustit následujícím příkazem v bashi:

```
hadoop jar WordCount.jar WordCount /path/to/input /path/to/output
```

Prefix sum

Definition

Mějme pole prvků $a_1, \ldots, a_n$ určitého typu a asociativní operaci $\oplus$. **Inkluzivní prefix sum** je aplikace asociativní operace $\oplus$ na všechny prvky vstupního pole tak, že výsledkem je pole $s_1, \ldots, s_n$ stejného typu, pro kterou platí

$$s_i = \oplus_{j=1}^{i} a_i.$$

Exkluzivní prefix sum je definován vztahy $\sigma_1 = 0$ a

$$\sigma_i = \oplus_{j=1}^{i-1} a_i,$$

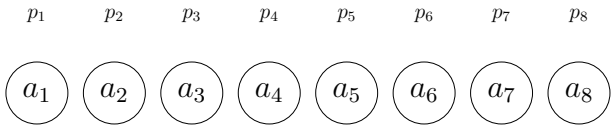
pro $i > 0$.

Prefix sum

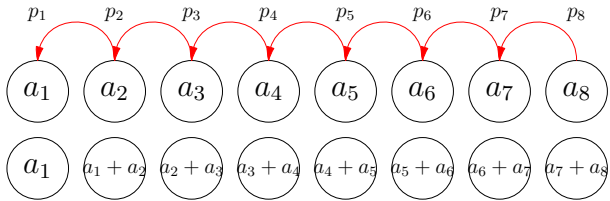
Poznámka: $\sigma_i = s_i - a_i$ pro $i = 1, \ldots, n$.
Příklad: Pro jednoduchost budeme uvažovat operaci sčítaní. Pak jde o paralelní provedení tohoto kódu:

```
s[ 1 ] = a[ 1 ];
for( int i = 2; i <= n; i ++ )
   s[ i ] += a[ i ] + s[ i - 1 ];
```

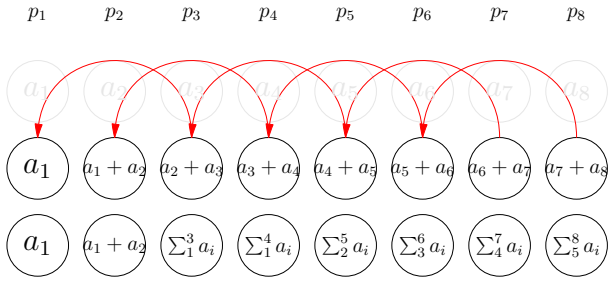
Paralelní prefix sum



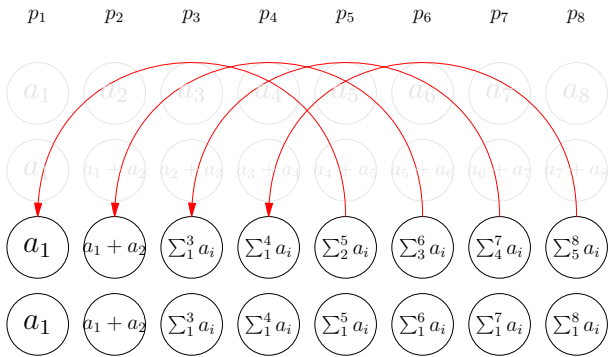
Paralelní prefix sum



Paralelní prefix sum



Paralelní prefix sum



Paralelní prefix sum

- $T_S(n) = \theta(n)$
- $T_P(n) = \theta(\log n)$
- $S(n) = \theta\left(\frac{n}{\log n}\right) = O(n) = O(p)$
- $E(n) = \theta\left(\frac{1}{\log n}\right)$

Paralelní prefix sum

- vidíme, že prefix sum má stejnou složitost jako redukce
- abysme získali nákladově optimální algoritmus, potřebovali bysme na jeden procesor mapovat celý blok čísel
- nyní je to o trochu složitější

Nákladově optimální paralelní prefix sum

- volíme $p < n$, pro jednoduchost tak, aby p dělilo n
- nechť každý procesor má blok A_k pro $k = 1, \ldots, p$
- v něm se sekvenčně napočítá částečný prefix sum

$$s_i^* = \sum_{j \in A_i \wedge j \leq i} a_i.$$

- definujeme posloupnost S_k pro $k = 1, \ldots, p$ tak, že $S_k = s_{i_k}$, kde i_k je index posledního prvku v bloku A_k.
- napočítáme exkluzivní prefix sum z posloupnosti $S_k \to \Sigma_k$
- každý procesor nakonec přičte Σ_k ke svým s_i
- potom je $T_P(n, p) = \theta\left(\frac{n}{p} + \log p\right)$ stejně jako u redukce

Segmentovaný prefix-sum

- jde o napočítání několik prefix-sum najednou
- například

$$[1, 2, 3], [4, 5, 6, 7, 8]$$

  dá výsledek

$$[1, 3, 6], [4, 9, 15, 22, 30]$$

- úlohu lze zakódovat jako jednu dlouhou řadu s restartovacími značkami
- sekvenční implementace pak může vypadat takto

Segmentovaný prefix-sum

```
1  void sequentialSegmentedPrefixSum( const int* a,
2                                     const int* segments,
3                                     const int n,
4                                     int* segmentedPrefixSum )
5  {
6     segmetedPrefixSum[ 0 ] = a[ 0 ];
7     for( int i = 1; i < n; i ++ )
8        if( segments[ i ] == 0 )
9           segmentedPrefixSum[ i ] =
10             segmentedPrefixSum[ i - 1] + a[ i ];
11       else
12          segmentedPrefixSum[ i ] = a[ i ];
13 }
```

Segmentovaný prefix-sum

- teoreticky lze segmentovaný prefix-sum implementovat stejně jako normální jen s jinak definovanou operací "sčítání"

$$(s_i, f_i) \oplus (s_j, f_j) = (f_j == 0 \ ? \ s_i + s_j \ : \ s_j, f_i | f_j),$$

kde $i < j$ a f_i je posloupnost jedniček na začátcích segmentů, jinak samé nuly