RWTH Aachen University

Aachen Institute for Advanced Study in Computational Engineering Science
High-Performance and Automatic Computing Group

Master's Thesis

Parallel Prefix Algorithms for the Registration of Arbitrarily Long Electron Micrograph Series

Marcin Copik

First Supervisor: Prof. Paolo Bientinesi
AICES
RWTH Aachen University

Second Supervisor: Prof. Benjamin Berkels
AICES
RWTH Aachen University

Abstract

Recent advances in the technology of transmission electron microscopy have allowed for a more precise visualization of materials and physical processes, such as metal oxidation. Nevertheless, the quality of information is limited by the damage caused by an electron beam, movement of the specimen or other environmental factors. A novel registration method has been proposed to remove those limitations by acquiring a series of low dose microscopy frames and performing a computational registration on them to understand and visualize the sample. This process can be represented as a prefix sum with a complex and computationally intensive binary operator and a parallelization is necessary to enable processing long series of microscopy images. With our parallelization scheme, the time of registration of results from ten seconds of microscopy acquisition has been decreased from almost thirteen hours to less than seven minutes on 512 Intel IvyBridge cores.

Contents

List of symbols and abbrevations

Image Registration

A	The registration function for two neighboring frames
B	The registration function for two non-neighboring frames
NCC$[f, g]$	The normalized cross–correlation of images f and g
$\phi_{i,j}$	Deformation matching image f_j to f_i
f_i	Image with index i

Parallel prefix sum

$\mathcal{SP}$	The measured speedup of an algorithm, defined as a ratio of serial and parallel exeuction time
P	Total number of allocated process cores
$S(N, P)$	Span, length of critical path of an algorithm for input data of length N and P workers
$SP(N, P)$	The theoretical speedup of an algorithm, defined as a ratio of serial and parallel span
T_P	Execution time of a parallel execution
T_S	Execution time of a serial execution
$W(N, P)$	An amount of work performed by an algorithm for input data of length N and P workers

Chapter 1

Introduction

Modern electron microscopes allow an observation of specimens at a nanometer resolution. Recent advances in the technology of scanning transmission electron microscopy (STEM) allowed for a more precise visualization of materials and physical processes such as metal oxidation. Nevertheless, the quality of information obtained during the acquisition is limited by the damage caused by an electron beam, movement of the specimen or other environmental factors. Restricting the electron dose to avoid the damage results in obtaining data with an undesirably low *signal–to–noise* ratio. Image processing algorithms have been successively applied to extract reliable information from a noisy electron microscopy data.

Berkels et.al.[1] have proposed a new approach to increase the amount of information gathered by observation with STEM. Instead of using a single high–dose frame, a series of low–dose noisy frames $f_0, f_1, \ldots, f_n$ is acquired. The quality of frames is affected not only by the noise but also by the movement of observed object during the acquisition. Therefore, frames are aligned to the first image f_0 to represent only the physical change of the observed object and not the movement of the specimen.

The information encoded in images is extracted in a two–step series *registration* method. First, for each pair of neighboring images (f_i, f_{i+1}) an image deformation $\phi_{i,i+1}$ is approximated such that $f_i \approx f_{i+1} \circ \phi_{i,i+1}$. The process of finding a good estimation for $\phi_{i,i+1}$ is based on a *multilevel gradient flow* minimizing a *normalized cross–correlation* between f_i and $f_{i+1} \circ \phi_{i,i+1}$. A composition of two deformations $\phi_{i,j}$ and $\phi_{j,k}$ with a common center j can be used as a starting to point for registration of frame f_k to f_i. A recursive application of this procedure allows aligning each frame f_i to the first one f_0. An example of the process is presented on Figure 1.1. A series of deformed images $f_i \circ \phi_{0,i}$ is averaged to produce a single frame representing the observed object.

The theoretical background of the new method and a formal statement of the problem is introduced in the the Chapter 2. The process of image registration is computationally intensive. As a matter of fact, the convergence of a gradient flow on three levels 8, 9 and 10, usually takes a few seconds. In our experiments, a serial registration of 4096 frames requires almost 13 hours of computation. Since each second of data acquisition generates 400 frames, the registration algorithm becomes impractical for an acquisition

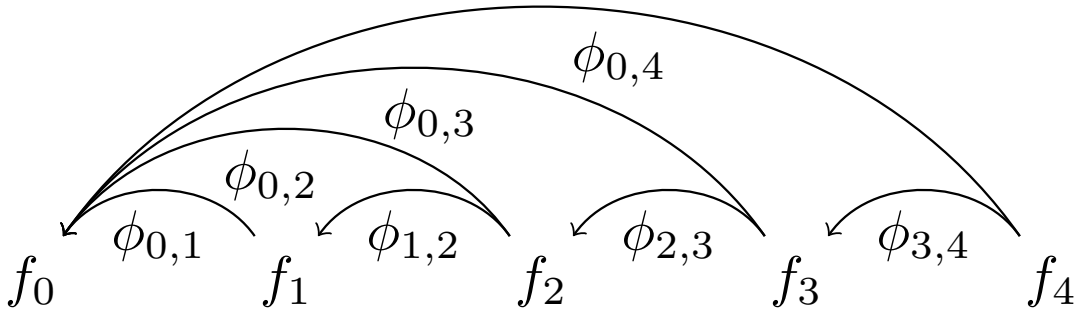


Figure 1.1: An example of series registration for images $f_0, f_1, \ldots, f_4$.

running longer than a few dozen seconds. This problem becomes even more apparent in applications such as the *series averaging procedure*, where the registration algorithm is repeated many times to improve the quality of averaged frame. The computation time could be reduced by manipulating algorithm parameters to perform fewer iterations, but this raises the likelihood of gradient flow finding a local minimum which decreases the quality of results. In this dissertation, we discuss another approach to speed up the registration procedure which is to employ parallel computing techniques.

In the Chapter 3, we prove that the image registration process can be represented as a *prefix sum*. The prefix sum, also known as *scan* or *cumulative sum*, accepts a sequence of input data $x_0, x_1, \ldots, x_n$ with a binary operator $\odot$ and for each element x_i, computes a sum of all preceding elements and the selected item, such as

$$x_i = x_0 \odot x_1 \odot \cdots \odot x_i$$

Parallelization strategies for a prefix sums have been researched for decades to construct fast and efficient prefix adders, a class of digital circuits performing binary addition. In the parallel programming, the scan primitive has been proposed as a basic block for building parallel applications. Therefore, we intend to use the parallel prefix sum as a basis for parallelization strategy of the image registration problem.

However, properties of the image registration process are entirely different from prefix sum problems discussed in the literature. Previous work is focused on parallelization strategies for memory–bound operators where the cost of accessing and moving data is significantly higher than an application of the operator. Furthermore, the iterative nature of registration does not allow to predict a total cost of computation, and we have observed huge variances in execution time between different pairs of frames. We have not found any examples of a parallel scan with an operator of unpredictable runtime. Thus, we construct new guidelines for an efficient parallelization and reevaluate existing strategies.

For an arbitrarily long series of data acquisition, we need a distributed implementation of a parallel prefix sum. We derive a general strategy which attempts to minimize the

synchronization between workers. The distributed approach for a parallel prefix sum is discussed in the Chapter 4.

We implemented the general strategy for a parallel distributed prefix sum as an extension of QuocMesh[45], a library for Finite Elements computations on Cartesian grids developed at the Institute for Numerical Simulation at the University of Bonn. The strategy has been applied to the image registration process, and a comparative evaluation of different algorithms is presented in Chapter 5. Our results prove that we cannot solve a fixed size problem efficiently on an arbitrary number of processors. However, we can use more hardware to register longer series without a significant increase in the execution time. Further improvements are provided by parallelization of the registration algorithm.

Finally, we present conclusions and suggestions for future work in Chapter 6.

Chapter 2

Image registration

In this chapter, we introduce the problem of image registration for series of frames. We use names *image* and *frame* interchangeably for a single item of data acquired by an electron microscope. Furthermore, we use terms *transformation* and *deformation* interchangeably for a function defining the deformation of an image. Definitions provided here are based on those given by Modersitzki in [2][3].

2.1 Problem statement

An image is a function which assigns a gray value to each position in the region of interest, as defined below

Definition 2.1.1. Image A d–dimensional image is a function f

$$f : \Omega \longrightarrow \mathbb{R} \tag{2.1}$$

where $\Omega \in \mathbb{R}^d$ is a region of interest and $d \in \mathbb{N}$. Img(d) is a set of all d–dimensional images.

In this chapter, we introduce methods for *registration* of two–dimensional images where $d = 2$. We denote two particular kinds of image: $\mathcal{R}$ known as the reference and $\mathcal{T}$ known as the template image. In the image registration problem, we want to find the transformation $\phi : \mathbb{R}^d \longrightarrow \mathbb{R}^d$ of $\mathcal{T}$ such that the deformed image is aligned to the reference image and

$$\mathcal{T} \circ \phi \approx \mathcal{R} \tag{2.2}$$

For the sake of simplicity, we restrict ourselves to rigid transformations in the description, but techniques described below allow to approximate both rigid and non–rigid deformations. A rigid deformation is defined as follows

Definition 2.1.2. Rigid transformation A transformation is called rigid when only rotation and translation are allowed. A rigid transformation is represented by the equation

$$\phi(x) \; = \; R(\alpha) \cdot x + G \tag{2.3}$$

where $R(\alpha) \in \mathbb{R}^{d\times d}$ is an orthogonal matrix and $G \in \mathbb{R}^d$.

The name refers to a movement of a rigid body which can not be deformed through a shear, scaling or any other non–affine transformation. In the two–dimensional case, the deformation is given by a translation vector b of length two and a single rotation angle α. Then the transformation shall take the form

$$\phi(x) \;=\; \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} + \begin{pmatrix} t_0 \\ t_1 \end{pmatrix} \tag{2.4}$$

In practice, however, it is usually impossible to obtain a deformation providing an ideal match for images. Therefore, a proper metric has to be defined to estimate the similarity between images and serve the role of an objective functional in the minimization process. We intend to find a transformation such that given a metric, the distance between a reference $\mathcal{R}$ and a transformed template $\mathcal{T} \circ \phi$ is minimal. Finally, we define the image registration problem

Definition 2.1.3. Image registration problem Given a metric $\mathcal{M} :\longrightarrow \mathbb{R}$ and two images $\mathcal{R}, \mathcal{T}$, find a transformation ϕ such that

$$\mathcal{M}(\mathcal{R}, \phi \circ \mathcal{T}) \tag{2.5}$$

is minimized.

An example of a trivial transformation is presented in Figure 2.1. For both images, $\Omega = [0,1]^2$. In the image f_1, the rectangle has been translated but not rotated. We intend to find a deformation $\phi_{0,1}$ such that

$$\begin{gathered} f_1 \circ \phi_{0,1} = f_0 \\ \forall x \in \Omega \; f_1(\phi_{0,1}(x)) = f_0(x) \end{gathered}$$

To find a transformation, we observe that no shift is performed on vertical axis and we solve for the lower–left corner of the rectangle $(0.25, 0.25)$ and $(0.5, 0.25)$

$$\begin{gathered} f_0(x) = f_1(\phi_{0,1}(x)) \\ f_0(x) = f_1(x + G) \\ f_0((0.25, 0.25)^T) = f_1((0.25, 0.25)^T + (g_0, 0)^T) \\ \Longleftrightarrow \\ (0.5, 0.25) = (0.25, 0.25)^T + (g_0, 0)^T \\ G = (0.25, 0)^T \end{gathered}$$

Thus, the transformation does not deform the template image f_1 to match f_0. This operation would require a translation vector $-G$. The deformation is applied to coordinates before computing an image value at given position. ϕ represents a geometrical change from frame f_0 to f_1, not the other way around. With both images representing exactly the same object, the transformation encodes a correspondence between coordinate systems of two images.

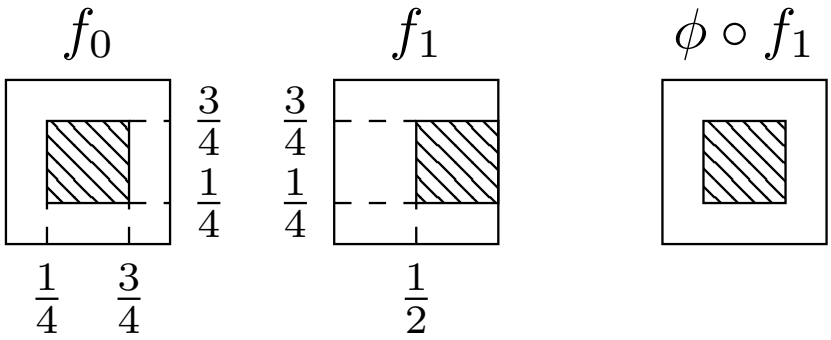


Figure 2.1: An example of two images containing a rectangle of size $(0.5, 0.5)$, located in the center of image f_0 and on right side of image f_1. Deformation ϕ produces an ideal match of f_1 to f_0.

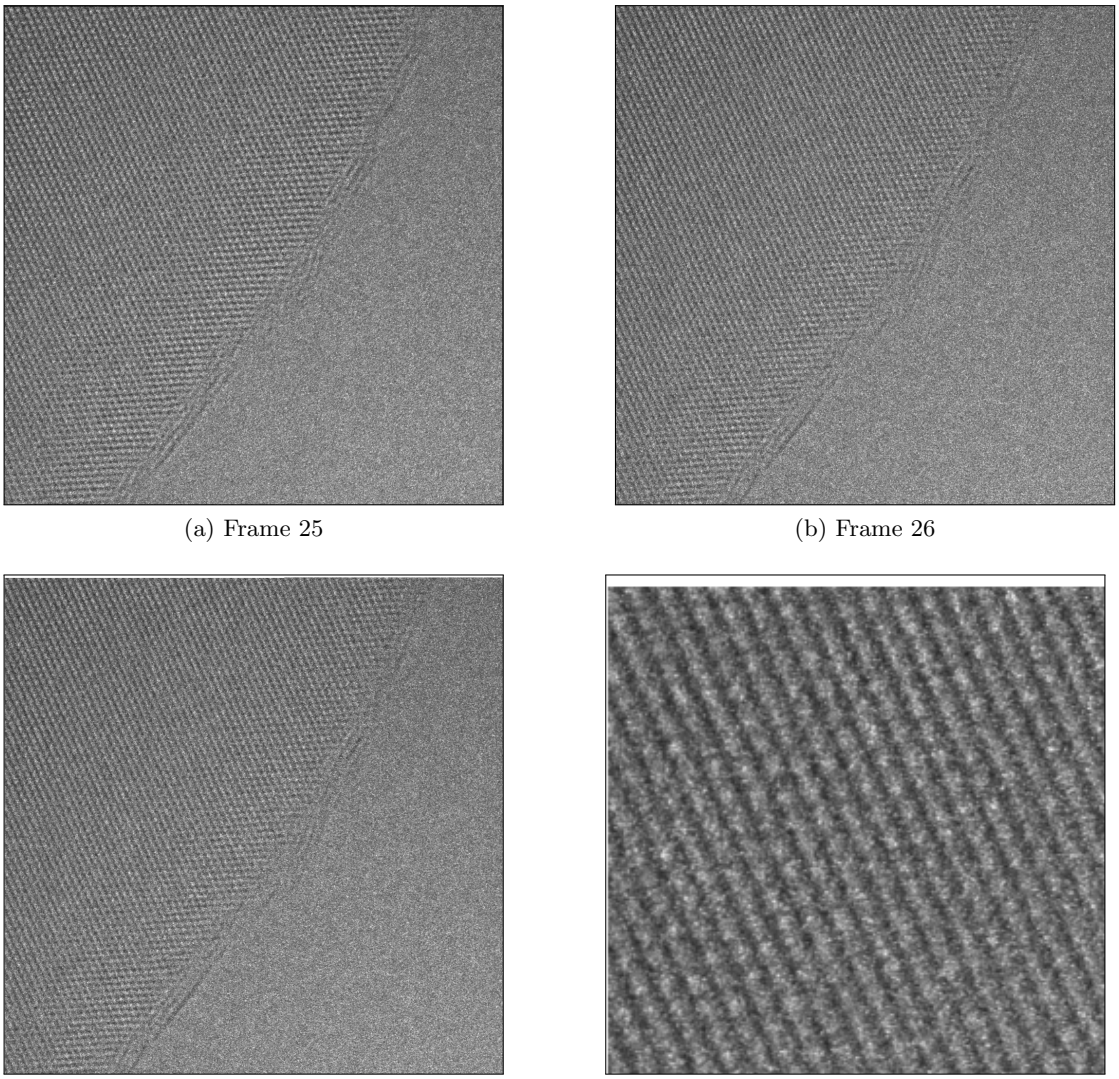
2.2 Electron microscopy data

For the image registration, we consider data from an experiment where ultrahigh vacuum high–resolution transmission electron microscopy (UVH HRTEM) has been applied to capture the process of aluminum oxidation[4]. The images have been acquired at the rate of 400 frames per second, and each experiment has lasted for up to 4 minutes, producing up to 96,000 frames. The quality of images is lowered by *sample drift*, a movement of the aluminum sample between taking consecutive frames, and the presence of low–contrast frames.

An example of an electron microscopy image is presented on Figure 2.2. The regular structure on the left represents an atomic grid of aluminum. The approximated deformation for this pair of images is

$$\phi(x) = \begin{pmatrix} 0.99 & -4.18 \cdot 10^{-4} \\ 4.18 \cdot 10^{-4} & 0.99 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} + \begin{pmatrix} -4.53 \cdot 10^{-4} \\ -0.004909 \end{pmatrix} \tag{2.6}$$

A key feature of these pictures is a very low variability between two images with neighbor indices, presented in Figures 2.2a and 2.2b. It is clearly seen in the deformation, the estimated angle of rotation is approximately equal $4.18 \cdot 10^{-4}$ radians and the translation on the horizontal axis is insignificant as well. Not surprisingly, it is a hard task to notice a change in the deformed frame presented on Figure 2.2c. A magnification of the upper-left corner of this frame is presented on Figure 2.2d. There, a movement of the frame along the vertical axis can be spotted.



(a) Frame 25

(b) Frame 26

(c) Deformed frame 26

(d) An upper–left corner of deformed frame 26

Figure 2.2: The first two figures of the top row present frames 25 and 26, respectively, from twentieth–first second of first minute of the acquisition. The registration has been performed and the two figures of the bottom row depict the deformed frame 26 and a magnification of an upper–left corner of this frame. The movement of frame is visible on the last image.

2.3 Image registration for electron microscopy

This section provides a brief description of the selected method for image registration. This technique has been proposed for electron microscopy images by Berkels et al. in [1]. For more details on the method, please refer to the mentioned paper or to [5].

We begin the description with a definition of selected metric. The mean value of an image $\bar{f}$ is defined as

$$\bar{f} = \frac{1}{|\Omega|} \int_{\Omega} f \mathrm{d}x \tag{2.7}$$

And the standard deviation is defined as follows

$$\sigma_f = \sqrt{\frac{1}{|\Omega|} \int_{\Omega} (f - \bar{f})^2 \mathrm{d}x} \tag{2.8}$$

The *normalized cross–correlation* of reference $\mathcal{R}$ and template $\mathcal{T}$ is given as

$$\mathrm{NCC}[\mathcal{R}, \mathcal{T}] = \frac{1}{|\Omega|} \int_{\Omega} \frac{\mathcal{R} - \bar{\mathcal{R}}}{\sigma_{\mathcal{R}}} \frac{\mathcal{T} - \bar{\mathcal{T}}}{\sigma_{\mathcal{T}}} \mathrm{d}x \tag{2.9}$$

The value of a normalized cross–correlation is bounded from below by -1 and from above by 1. Obviously, for a perfect transformation $\mathrm{NCC}[\mathcal{R}, \phi \circ \mathcal{T}] = 1$. This function is combined with a regularization term consisting of Dirichlet energy of the displacement $\phi(x) - x$ to form the objective functional for the optimization process

$$\mathrm{E}[\phi] \;=\; -\mathrm{NCC}[\mathcal{R}, \phi \circ \mathcal{T}] + \lambda \cdot \frac{1}{2} \int_{\Omega} \|D(\phi(x) - x)\|^2 \mathrm{d}x \tag{2.10}$$

The additional term, controlled by a regularization parameter $\lambda > 0$, has been introduced because of an ill–posedness of the problem. This function is characterized by a presence of multiple local minima which make finding a unique solution a very non–trivial problem. A key requirement for well–posedness of a problem is an existence of a unique solution, and the additional regularization is expected to make the problem well–posed by leading to a more convex functional with a single minimum[3]. As we are going to see later, it is usually not the case, and the computed deformation may vary not only between different starting points for the minimization but also among various implementations of the same algorithm.

The proposed approach for minimization of the functional is a hybrid one, based on a combination of a multilevel scheme with a gradient flow minimization process. The multilevel process and the gradient flow with its spatial discretization are introduced in the following sections.

2.3.1 Multilevel

The idea of a multilevel algorithm comes from multigrid[6], a major technique developed for solving partial differential equations. There, a scheme consisting of multiple grids is used to reduce high–frequency errors by applying a smoother at a fine grid and iteratively solving the problem on coarser grids. Each step down to a coarser grid requires *restricting* the residual, solving problem there and *prolongating* the solution to a finer grid.

A multilevel scheme applies those concepts in image registration to minimize the likelihood of a gradient solver stopping at local minima. The minimization process operates on grid levels $m_0, m_0+1, \ldots, m_1$ and $m_0 < m_1$. For each level l, a grid of size $2^l \times 2^l$ or $(2^l+1) \times (2^l+1)$ is created. For this implementation, we focus on the latter. A coarser grid $\mathcal{G}_l$ of size $(2^l+1) \times (2^l+1)$ is extended with a new node between each pair of nodes. The new finer grid $\mathcal{G}_{l+1}$ has $(2^{l+1}+1) \times (2^{l+1}+1)$ nodes. An example is presented in Figure 2.3.
The *prolongation* operator $\mathcal{I}_l^{2l}$ is responsible for copying values from a coarser grid to corresponding nodes in the finer grid and for computing a bilinear interpolation for each new node, as defined below

$$\mathcal{I}_l^{2l} f(x,y) = \begin{cases} \frac{1}{4}(f(x,y-l)+f(x,y+l) & \text{for } x \bmod 2 = 0 \\ \quad +f(x-l,y)+f(x+l,y)) & \quad \wedge\, y \bmod 2 = 0 \\ \frac{1}{2}(f(x-l,y)+f(x+l,y)) & \text{for } x \bmod 2 = 0 \\ \frac{1}{2}(f(x,y-l)+f(x,y+l)) & \text{for } y \bmod 2 = 0 \\ f(x,y) & \text{otherwise} \end{cases} \tag{2.11}$$

Since the direction is from a coarse to fine grid, there is no procedure of going back to the starting level like in the V–cycle in multigrid. However, the *restriction* operator is required in the initialization to represent images and the initial guess of deformation on the coarse grid. This operation is performed by a scaled average of neighbors around the coarse node and a stencil representation of this operator is

$$\frac{1}{16}\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} \tag{2.12}$$

On coarser levels, fewer features of an image are preserved which eradicates local minima created by small structures in the image. Furthermore, the multigrid strategy has been known as an efficient iterative solver because of less computationally intensive computations on coarse levels.

2.3.2 Gradient flow

A gradient flow solver is a generalization of the gradient descent. The idea behind this optimization technique is the same - moving in the direction of the negative gradient - but the gradient is computed with respect to a scalar product G. The update is defined as an ordinary differential equation

$$\frac{\partial \phi}{\partial t} = -\text{grad}_G \text{E}[\phi] \tag{2.13}$$

The scalar product G is selected in a way to help the minimization process avoid local minima. It has been shown that this ordinary differential equation can be reformulated as

$$\frac{\partial \phi}{\partial t} = -A^{-1}\text{E}'[\phi] \tag{2.14}$$

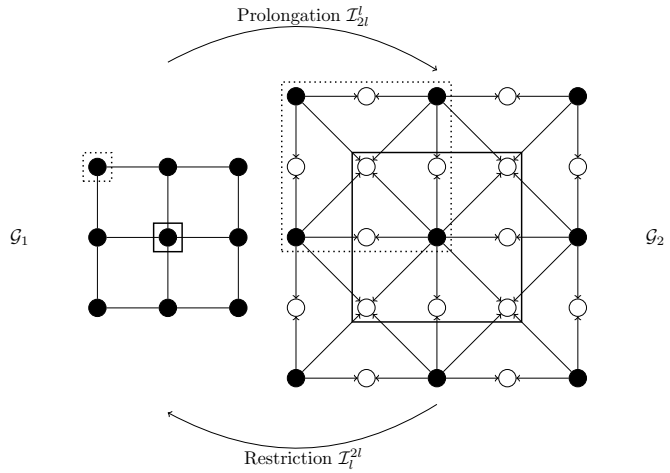


Figure 2.3: An example of a coarse grid $\mathcal{G}_1$ and a finer grid $\mathcal{G}_3$. On a fine grid, nodes copied from a coarse grid are depicted as black, and new nodes are white. For each new node, arrows depict nodes contributing to the bilinear interpolation $\mathcal{I}_{2l}^{l}$. Dashed and solid frames symbolize the restriction process. On the finer grid, a frame identifies nodes to which the stencil operator $\mathcal{I}_{l}^{2l}$ is applied. The result of a restriction is stored in a node indicated by a corresponding frame on the coarse grid.

E' refers to the first variation of the functional E i.e. the generalization of the first derivative of a function of one variable to functionals. The other component A^{-1} is selected to smooth the functional. The smoother is applied only in the non–rigid registration.
The equation is discretized in both spatial and time domain. For the latter, a forward Euler scheme is applied

$$\frac{\partial \phi}{\partial t} = \frac{\phi^{k+1} - \phi^{k}}{\tau} \tag{2.15}$$

Where τ is a step size and ϕ^k is an approximation of deformation from k-th iteration. The discrete form of equation 2.14 is obtained

$$\phi^{k+1} = \phi^{k} - \tau A^{-1} \mathrm{E}'[\phi^{k}] \tag{2.16}$$

The step size is selected to ensure the convergence and make the update process more efficient by choosing in each iteration the largest τ still guaranteeing a decrease of energy. For this problem, an Armijo rule with widening is applied to the energy function of a new deformation $\mathrm{E}(\phi^{k+1})$

$$\Phi(\tau) = \mathrm{E}[\phi^{k} - \tau A^{-1} \mathrm{E}'[\phi^{k}]] \tag{2.17}$$

A τ value is selected such that all conditions below are satisfied for $0 < \sigma < 1$

$$\begin{cases} \frac{\Phi(\tau) - \Phi(0)}{\Phi'(\tau)\tau} > \sigma \\ \tau \leq \tau_{max} \end{cases} \tag{2.18}$$

This ensures that the decay of energy Φ is at least σ times larger than the expected decrease in energy, given by the derivative Φ'. In the implementation employed for this problem, $\sigma = 0.5$.

2.3.3 Spatial discretization

The image domain Ω is mapped into a uniform rectangular mesh i.e. a mesh consisting of M equal nodes N_i in shape of a rectangle. Then, a canonical basis needs to be constructed. For this problem, piecewise bilinear have been selected as basis functions φ_i. The image function f expressed in the nodal basis is presented in the next equation

$$f = \sum_{j=1}^{M} f_j \varphi_j \tag{2.19}$$

An FE discretization allows introducing the mass matrix M to discretize the integration over domain Ω

$$M_{i,j} = \int_{\Omega} \varphi_i \varphi_j \mathrm{d}x \tag{2.20}$$

As a result, a new equation for a mean of an image can be derived

$$\begin{aligned} \bar{f} &= \frac{1}{\Omega} \int_{\Omega} f \mathrm{d}x \\ &= \sum_{i=1}^{M} \sum_{j=1}^{M} \int_{\Omega} f_i \varphi_i \varphi_j \mathrm{d}x \\ &= MF\mathbb{1} \end{aligned} \tag{2.21}$$

where F denotes a vector representation of f in the nodal basis and $\mathbb{1}$ is a vector of ones. Furthermore, the standard deviation becomes

$$\begin{aligned} \sigma_f &= \sqrt{\frac{1}{|\Omega|} \int_{\Omega} (f - \bar{f})^2 \mathrm{d}x} \\ &= \sqrt{M(F - \bar{F})^2} \end{aligned} \tag{2.22}$$

And the normalized cross–correlation can be formulated as

$$\begin{aligned} \mathrm{NCC}[\mathcal{R}, \mathcal{T}] &= \frac{1}{|\Omega|} \int_{\Omega} \frac{\mathcal{R} - \bar{\mathcal{R}}}{\sigma_{\mathcal{R}}} \frac{\mathcal{T} - \bar{\mathcal{T}}}{\sigma_{\mathcal{T}}} \mathrm{d}x \\ &= M \tilde{\mathcal{R}} \tilde{\mathcal{T}} \end{aligned} \tag{2.23}$$

where $\tilde{\mathcal{R}}$ and $\tilde{\mathcal{T}}$ are FE representations of normalized reference and template images.

2.3.4 Summary

Algorithm 1 Multilevel gradient flow for image registration.

for i **from** $m_1 - 1$ **to** m_0 **step** -1 **do** ▷ Initialize coarse grids
  $\mathcal{R}^i \leftarrow restriction(\mathcal{R}^{i+1})$
  $\mathcal{T}^i \leftarrow restriction(\mathcal{T}^{i+1})$
  $\phi^i \leftarrow restriction(\phi^{i+1})$ ▷ Only for a non–rigid deformations
end for
for i **from** m_0 **to** m_1 **do**
  $\phi^i \leftarrow solve(\mathcal{R}^i, \mathcal{T}^i, \phi^i)$ ▷ Gradient flow
  if $i < m_1$ **then** ▷ Prolongate deformation to a finer grid
    $\phi^{i+1} \leftarrow prolongate(\phi^i)$ ▷ Only for a non–rigid deformations
  end if
end for

The method is summarized on Algorithm 1. The deformation ϕ^i is prolongated and restricted only in the non–rigid case. A rigid transformation consists only of three parameters, and it does not depend on grid size.
Given relatively low differences between two consecutive frames, setting an identity transformation as the initial guess should not prevent the algorithm from finding a decent solution. Multilevel start and end levels are selected by the user, and they provide a firm boundary on the outer loop. On the other hand, the inner loop requires proper stopping criteria. We employ two criteria:

- convergence ϵ
  stop the computation if a difference in energy $\mathrm{E}[\phi^{k+1}] - \mathrm{E}[\phi^k]$ is less than the ϵ
- maximum number of iterations
  perform at most $iter\_max$ iterations

In the following chapters, we refer to an implementation of this technique as the function **A**. It accepts two images with subsequent indices, f_i and f_{i+1}, and the initial guess for deformation ϕ_0 with a default value of this parameter equal to an identity transformation I_ϕ. The function applies the proposed algorithm to estimate a deformation $\phi_{i,i+1}$

$$\forall i \in \mathbb{N}\ \phi_{i,i+1} = \mathbf{A}(f_i, f_{i+1}, \phi_0 = I_\phi) \tag{2.24}$$

2.4 Registration for series of images

The previous section introduced a multilevel gradient flow for registration of two consecutive frames. However , we operate on a sequence of $n + 1$ images $f_0, f_1, \ldots, f_n$. Therefore, a strategy for registration of series of frames is necessary.
Given deformation $\phi_{0,1}$ which estimates $f_1 \circ \phi_{0,1} \approx f_0$, and deformation $\phi_{1,2}$ providing an

approximation $f_2 \circ \phi_{1,2} \approx f_1$, we can safely assume that a composition of deformations $\phi_{1,2} \circ \phi_{0,1}$ is a decent guess of deformation registering f_0 and f_2, since

$$\begin{aligned} f_2 \circ (\phi_{1,2} \circ \phi_{0,1}) &= (f_2 \circ \phi_{1,2}) \circ \phi_{0,1} \\ &\approx f_1 \circ \phi_{0,1} \\ &\approx f_0 \end{aligned} \tag{2.25}$$

Thus, we can approximate the deformation for two non–consecutive frames by using a specific initial guess. We reuse the function $\mathbf{A}$ defined in the previous section to define a new function $\mathbf{B}$ such that

$$\begin{aligned} \forall i, k \in \mathbb{N}, |k-i| > 1 \; \forall j \in \mathbb{N}, i < j < k \; \phi_{i,k} &= \mathbf{B}(\phi_{i,j}, \; \phi_{j,k}) \\ &= \mathbf{A}(f_i, f_k, \phi_{j,k} \circ \phi_{i,j}) \end{aligned} \tag{2.26}$$

In particular, if we iterate consecutively from the first image

$$\forall i \in \mathbb{N}, i > 1 \; \phi_{0,i} = \mathbf{B}(\phi_{0,i-1}, \phi_{i-1,i}) \tag{2.27}$$

The algorithm (2.27) is depicted in Figure 2.4. A series of aligned images may be

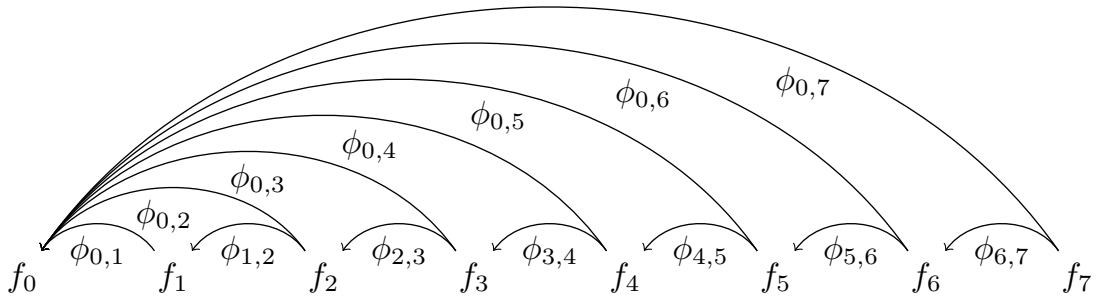


Figure 2.4: An image registration process for a series of frames. For an image f_i, the partial result from its predecessor $\phi_{0,i-1}$ is combined with a neighbor deformation $\phi_{i-1,i}$ to register the image to the reference frame f_0.

averaged to obtain a single image storing all information acquired in the experiment. Later, the *series averaging procedure* may be used to obtain results of a better quality by performing multiple iterations of the algorithm (2.27). For details, please refer to [1].

We formally define the problem as consisting of two steps - a preprocessing stage to generate neighbor transformations and the general registration for non–consecutive frames. In next sections, we refer to the second stage as the *image registration problem.*

Definition 2.4.1. Image series registration Given a sequence of images $f_0, f_1, \ldots, f_n$, perform a *preprocessing* step to register each pair of frames

$$\phi_{i,i+1} = \mathbf{A}(f_i, f_{i+1}, I_\phi)$$

and the *series registration* step to align each image f_i to the reference f_0.

2.5 A note on the associativity

The standard iterative algorithm outlined in the previous section would require three applications of function **B**

$$\begin{aligned}\phi_{0,2} &= \mathbf{B}(\phi_{0,1}, \phi_{1,2}) \\ \phi_{0,3} &= \mathbf{B}(\phi_{0,2}, \phi_{2,3}) \\ \phi_{0,4} &= \mathbf{B}(\phi_{0,3}, \phi_{3,4})\end{aligned} \tag{2.28}$$

It is not the only way of computing $\phi_{0,4}$. The process can be split between two processors computing independently $\phi_{0,2}$ and $\phi_{2,4}$. Then, results are merged to estimate $\phi_{0,4}$

$$\begin{aligned}\phi_{0,2} &= \mathbf{B}(\phi_{0,1}, \phi_{1,2}) \\ \phi_{2,4} &= \mathbf{B}(\phi_{2,3}, \phi_{3,4}) \\ \phi_{0,4} &= \mathbf{B}(\phi_{0,2}, \phi_{2,4})\end{aligned} \tag{2.29}$$

The first strategy (2.28) returns such deformation

$$\phi_{0,4}(x) \;=\; \begin{pmatrix} 0.999 & 1.074 \cdot 10^{-5} \\ -1.074 \cdot 10^{-5} & 0.999 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} + \begin{pmatrix} 4.743 \cdot 10^{-4} \\ 3.431 \cdot 10^{-3} \end{pmatrix} \tag{2.30}$$

The alternative approach (2.29) produces a result $\phi'_{0,4}$ which is clearly different from the previous one

$$\phi'_{0,4}(x) \;=\; \begin{pmatrix} 0.999 & -1.424 \cdot 10^{-4} \\ 1.424 \cdot 10^{-4} & 0.999 \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \end{pmatrix} + \begin{pmatrix} 5.158 \cdot 10^{-4} \\ 4.325 \cdot 10^{-3} \end{pmatrix} \tag{2.31}$$

Deformed images have been verified to represent images indistinguishable by a human operator. For further analysis, we compare these deformations by computing energy along a line going through both solutions. We evaluate the energy functional for different deformations

$$\mathrm{E}[t\phi_{0,4} + (1-t)\phi'_{0,4}] \tag{2.32}$$

Obviously, for $t = 0$ we have $\mathrm{E}[\phi'_{0,4}]$ and for $t = 1$ the computed functional is $\mathrm{E}[\phi_{0,4}]$. The function is plotted against different values of t in Figure 2.5. In the second case, the gradient solver has not been able to reach $\phi_{0,4}$ because it got stuck at a local minimum $\phi'_{0,4}$. An important conclusion here is that the difference between deformations is small enough to not have any impact on the image. Thus, we consider two solutions to be identical if the difference between them appears only at the sub–pixel level. We formally define it by introducing the concept of *approximate associativity*, a weaker form of the associativity

Definition 2.5.1. Approximate associativity A binary operation $\odot$ defined on a set S is called *approximately associative* if

$$\forall a, b, c \in S \;\; (a \odot b) \odot c \approx a \odot (b \odot c) \tag{2.33}$$

where $\approx$ defines two objects as equal if they are indistinguishable in the context of the operation represented by $\odot$ i.e. they represent the same final result.

Thus, changing the order of operations during evaluation may affect the exact representation of the result. Obviously, each associative operator is approximately associative at the same time. We conclude this chapter with a final remark.

Remark 2.5.1. Function **B** is approximately associative.

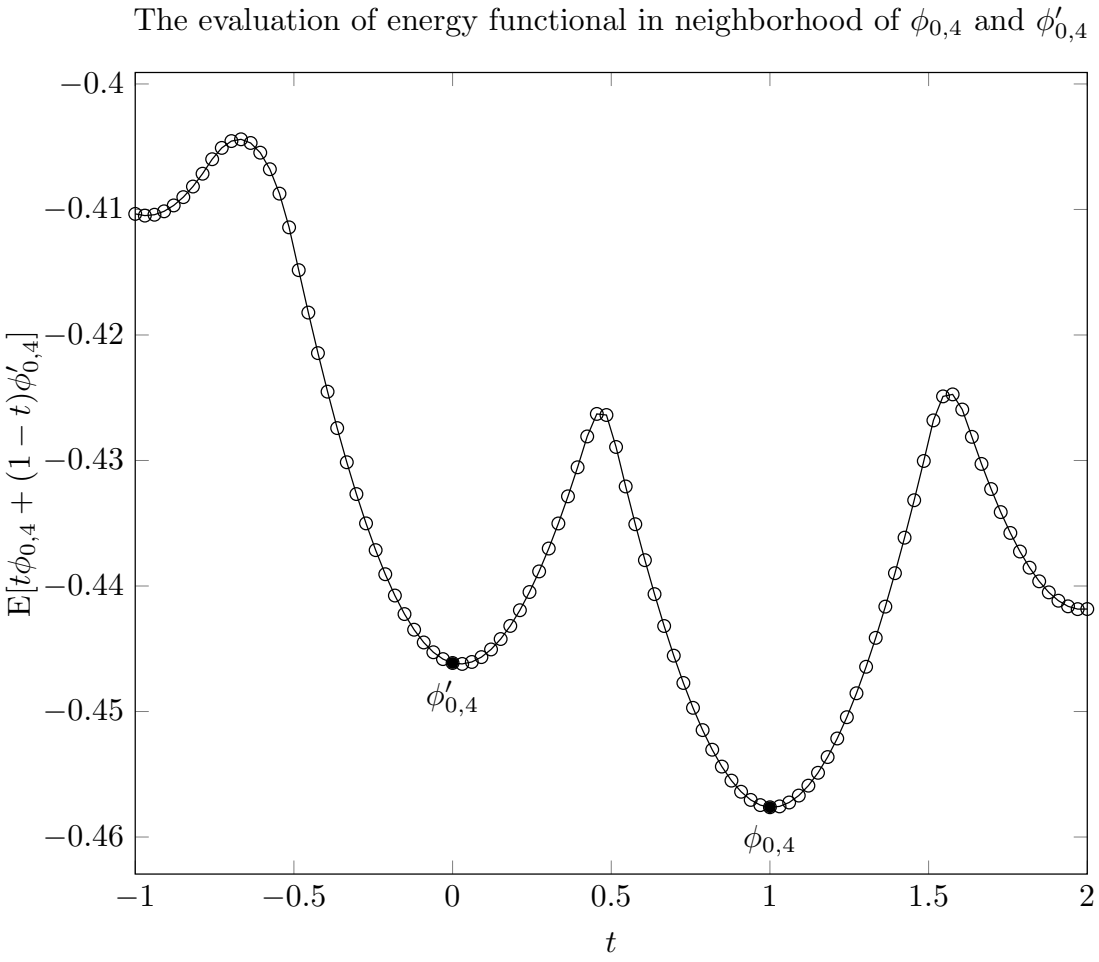


Figure 2.5: An evaluation of the energy function along a line passing through $\phi_{0,4}$ and $\phi'_{0,4}$. Both solutions appear to be a local minimum and the problem is ill–posed.

Chapter 3

Prefix sum

In this chapter, we introduce the prefix sum and discuss parallelization strategies known from the literature. We use names *prefix sum* and *scan* interchangeably.

3.1 Introduction

A prefix sum is an operation accepting a sequence of elements and generating a new sequence of partial sums. The operation has been described under many names in literature, including names such as inclusive prefix sum, scan, cumulative sum. The definition is as follows:

Definition 3.1.1. Prefix sum A prefix sum operation applies a binary approximately associative operator $\odot$ to a sequence (x_i)

$$x_1, x_2, x_3, \ldots, x_{n-1}$$

generating a new sequence (y_i)

$$\begin{aligned} y_1 &= x_1 \\ y_2 &= x_1 \odot x_2 \\ y_3 &= x_1 \odot x_2 \odot x_3 \\ &\cdots \\ y_{n-1} &= x_1 \odot x_2 \odot \cdots \odot x_{n-1} = \odot_{i=1}^{n-1} x_i \end{aligned}$$

Our definition differs from the one commonly used in the literature in that we permit approximately associative operators. With such operator, the result may be affected by changes in the order of operator evaluations, but it is guaranteed to be correct. This extension is required to treat the image registration problem as a prefix sum.
We use the notation $\odot_{i=1}^{n-1}$ to describe an iterative application of the binary operator $\odot$ and $x_{1,n-1}$ to denote the product of such application. The definition above presents an *inclusive* prefix sum where each new element with index i is a sum of first i elements

of input sequence. In an alternative approach, the new i-th value is a sum of first $i-1$ input values. The i-th value is *excluded*, therefore this approach is known as an *exclusive* prefix sum. The method is also known under a name prescan.

Definition 3.1.2. Exclusive prefix sum An exclusive prefix sum operation applies a binary approximately associative operator $\odot$, with an identity element $I_\odot$, to a sequence (x_i)

$$x_1, x_2, x_3, \ldots, x_{n-1}$$

generating a new sequence (y_i)

$$\begin{aligned} y_1 &= I_\odot \\ y_2 &= x_1 \\ y_3 &= x_1 \odot x_2 \\ &\cdots \\ y_{n-1} &= x_1 \odot x_2 \odot \cdots \odot x_{n-2} = \odot_{i=1}^{n-2} x_i \end{aligned}$$

Inclusive and exclusive prefix sums are closely connected with each other. After all, for an input sequence of length n, $n-1$ output elements are exactly the same in both scans, only placed in different positions. Obtaining an exclusive result from an inclusive prefix sum is trivial because all necessary values are already computed, and it is sufficient to shift results by one position to the right and place the identity element $I_\odot$ on the very first position. On the other hand, computing an inclusive prefix sum from an exclusive one may require additional computation. Results are shifted by one position to the left, and the binary operator is applied to y_{n-2} and x_{n-1} to compute the last reduction value y_{n-1}. In some algorithms, such as Blelloch parallel prefix sum described in section 3.3.1, this step is unnecessary because the full reduction has already been computed.

The scan algorithm has been originally proposed for APL programming language[7]. Prefix sum is a natural representation for a binary addition of two numbers in digital circuits[8] and a significant progress have been achieved to improve the performance and internal design of arithmetical circuits known as parallel prefix adders. Modern parallel prefix algorithms are usually based on circuit design research.
The parallel prefix sum has been described as a common pattern and fundamental building block in parallel applications[9][10]. It has been found to be helpful and useful for implementation and parallelization of multiple computer science problems with serial dependencies, including, but not limited to, polynomial evaluation, various sorting algorithms, solving recurrence equations, graph and tree algorithms[11][12][13]. The usefulness of prefix sum has motivated a proposal of *scan vector model* for parallel computations, where prefix sums are given as a unit time primitive[14].

A proof for the importance of this algorithm is the prevalence of various implementations of the prefix sum in major programming languages. C++ Standard Template Library[15] includes a sequential implementation of an inclusive prefix sum `std::partial_sum`. The order of summation is explicitly defined by the standard and associativity

of a binary operator is not required. The most recent update of standard[1][16] provides several overloads of `std::inclusive_scan` and `std::exclusive_scan`, both sequential and parallel via appropriate execution policies. Associativity of the binary operator is required, otherwise the behavior is nondeterministic. A similar set of overloads `std::transform_inclusive_scan` and `std::transform_exclusive_scan` transforms input range with an unary operator before a prefix sum is computed. Other parallel implementations are provided with an `inclusive_scan` and `exclusive_scan` in OpenCL-based Boost.Compute[46] and and CUDA-based Thrust[17], a multithreaded implementation in Intel TBB[18] and a distributed implementation of `MPI_Scan` and `MPI_Exscan`[19].
The prefix sum has been generalized to perform independently multiple scans on *segments*, disjoint subsequences of input data, therefore this operation is known as a segmented prefix sum[14]. An additional input is required to distinguish how segments are located in the input sequence (x_i). It could be a bit sequence of the same length as input data for prefix sum, indicating where new segment starts, or a shorter sequence of integers containing lengths of consecutive segments. We provide a formal definition for the former case.

Definition 3.1.3. Segmented prefix sum A segmented prefix sum operation takes as an input two sequences of the same length, a data sequence (x_i) and a flag sequence of bits (b_i), and applies a binary approximately associative operator $\odot$, generating a new sequence (y_i)

$$\begin{aligned} y_1 &= x_1 \\ y_2 &= (y_1 \otimes b_2) \odot x_2 \\ &\dots \\ y_{n-1} &= (y_{n-2} \otimes b_{i-1}) \odot x_{n-1} \end{aligned}$$

where binary operator $\otimes$ is defined as follows

$$x \otimes b = \begin{cases} I_{\odot}, & \text{if } b = 1. \\ x, & \text{otherwise.} \end{cases}$$

Segmented prefix sum has been found useful in parallelization of the quicksort algorithm[14]. An example of an implementation is `MPI_Scan` which computes k independent prefix sums for input array of length k.
A further variation of this technique is known as multiprefix[20]. There, multiple exclusive scans are computed for data represented by pairs (k, a), where k is a key encoding in which subsequence is the value a located. Segments are required to be neither contiguous nor disjoint. The multiprefix operation can be performed with work–efficient algorithm of $O(\sqrt{n})$ span[21].

[1]At the time of writing, C++17 was feature-complete but an official ISO standard has not been published.

3.2 Prefix sum for registration problem

In the previous chapter, we have defined the process of applying distinct operators **A** and **B** to register a sequence of images. In this chapter, we focus on the second phase of the process where neighbor deformations are processed to align all images to the first one. The initial application of function **A** to images can be performed independently and it is not relevant to the analysis.
A short investigation of the approach reveals a striking similarity between registration process and prefix sum. Indeed, for any final deformation $\phi_{0,i}$ we observe

$$\begin{aligned} \phi_{0,i} &= \mathbf{B}(\phi_{0,i-1},\ \phi_{i-1,i}) \\ &= \mathbf{B}(\mathbf{B}(\phi_{0,i-2}, \phi_{i-2,i-1}), \phi_{i-1,i}) \\ &= \mathbf{B}(\mathbf{B}(\mathbf{B}(\dots), \phi_{i-2,i-1}), \phi_{i-1,i}) \\ &= \ \phi_{0,1} \odot_B \phi_{1,2} \odot_B \cdots \odot_B \phi_{i-2,i-1} \end{aligned} \tag{3.1}$$

where binary operator $\odot_B$ is defined as follows

$$\phi_{i,j} \odot_B \phi_{j,k} \ = \ \mathbf{B}(\phi_{i,j}, \phi_{j,k}) \tag{3.2}$$

The new operator inherits approximate associativity from function **B** and by the definition 3.1.1, we prove that the problem (3.1) may be represented in terms of a prefix sum. A stronger formulation of prefix sum with a regular associativity would not allow using function **B** as an operator for prefix sum. This result allows to express the parallelization of image registration as a parallelization of prefix sum, and the parallel prefix pattern has been found to be applicable in yet another problem.

3.3 Parallel prefix sum

Upon initial inspection, the inevitable sequential nature of prefix sum is a bad indicator for finding an efficient parallelization strategy. Undoubtedly, in this case, it is not possible to achieve a perfect linear scaling, but several decades of research have produced numerous procedures with varying efficiency.
Below we describe distinct approaches for parallelization of prefix sum. A lot of recent work has been done on researching and optimizing different algorithms for SIMD architectures such as GPGPU[22][23][24]. New, hybrid strategies have been developed to fit their execution model[25][26]. Design goals and improvements are, however, related to the specific execution and memory models, such as removing bank conflicts or proper utilization of hierarchical memory.

For the simplicity of analysis, we assume that there are exactly as many workers as data elements and that the length of input data is a power of two. The Chapter 4 presents the general scheme for prefix sum without those assumptions. For simplicity's sake, for all discussed algorithms we assume a constant running time $C_\odot$ of the binary operator $\odot$.
Our comparative analysis is based on the *span*[10] or *depth*[27] of an algorithm i.e. length

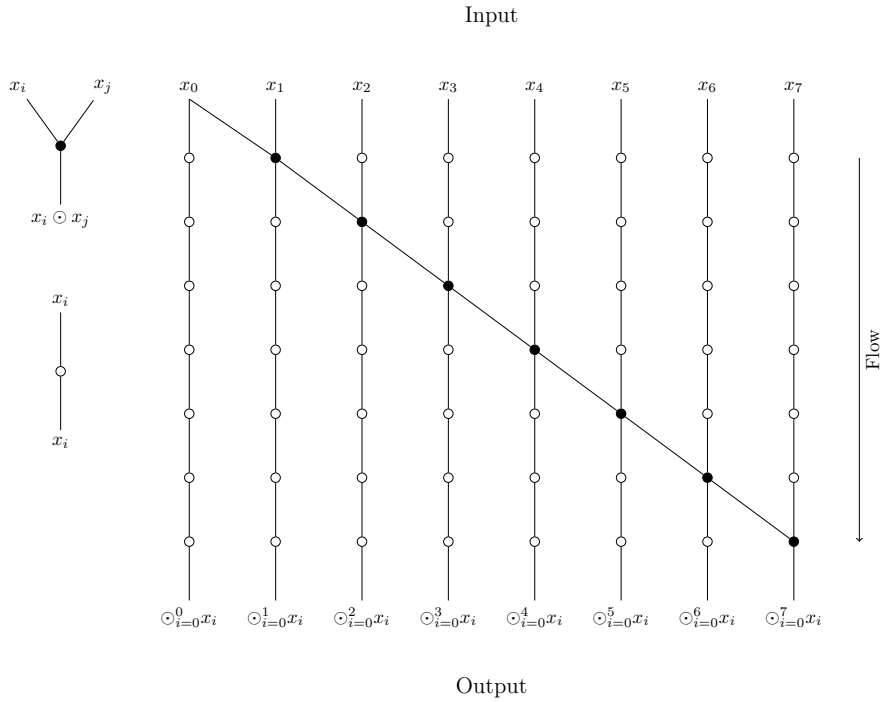


Figure 3.1: An example of serial prefix sum on 8 data elements. Results are produced in 7 steps and 7 applications of binary operator.

of the critical path determined by the longest sequence of computation performed during the execution. A comparison of a span between serial and parallel algorithm gives an upper bound on attainable parallelism. We provide work complexity as well, which estimates the span in a case of a fully serialized execution. An algorithm is considered to be *work–efficient* if its work complexity scales linearly with the size of input data. In our PRAM algorithms, we assume the input data to be allocated in a single block of memory with zero-based indexing. The for loop has an inclusive upper bound in our notation. As an example, a for loop iterating from 1 to N executes N iterations.

We begin the description by introducing the serial algorithm for prefix sum computation on Listing 2. Figure 3.1 presents an example of serial prefix sum. Each layer of nodes depicts a single iteration of the algorithm and filled nodes represent an application of binary operator. The lines joining nodes represent communication between workers. The algorithm is a direct mapping from definition 3.1.1. It is worth noting that this solution is the most optimal in terms of work–efficiency. Total span of the algorithms is simply equal to $N - 1$ applications of the operator

$$\begin{aligned} S_S(N) &= N - 1 \\ W_S(N) &= N - 1 \end{aligned} \tag{3.3}$$

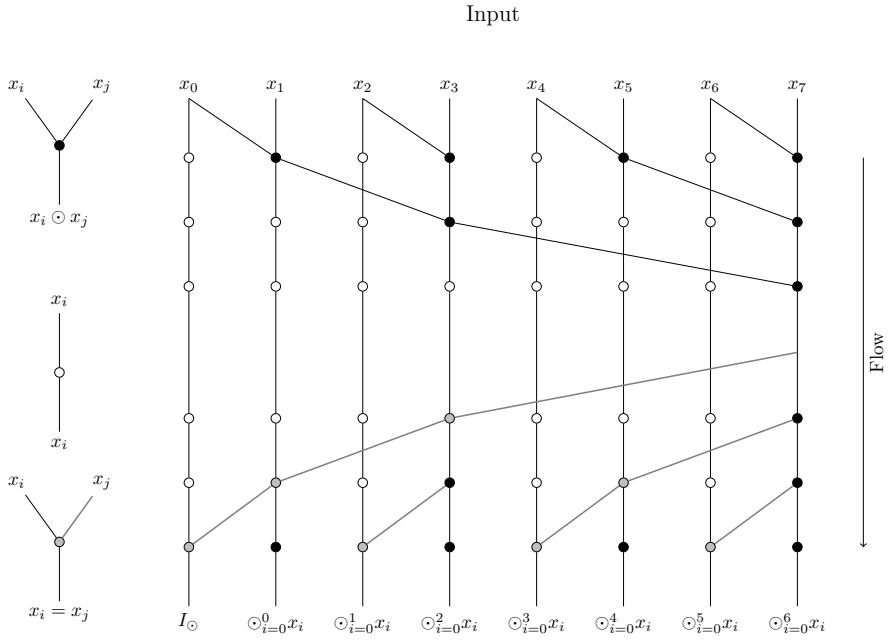


Figure 3.2: An example of Blelloch parallel prefix sum on 8 data elements. Results are produced in 6 steps and 14 applications of the binary operator. A black node represents an application of the binary operator during either up–sweep or down–sweep phase (lines 3 and 14 of Algorithm 3, respectively). A gray node represents a part of down–sweep where a left child receives a value from its parent and no computation is performed. This node corresponds to line 12 of Algorithm 3.

Algorithm 2 A pseudocode for serial prefix sum of N deformations.

1: **for** i **from** 1 **to** $N-1$ **do**
2: $\quad data[i] = data[i-1] \odot data[i]$
3: **end for**

3.3.1 Blelloch scan

One of the most popular parallel prefix sum strategies has been researched and presented by Guy Blelloch[28]. The tree–based approach correlates with Brent–Kung parallel prefix adder[29], described separately later. Figure 3.2 depicts an example of Blelloch prefix sum on eight image deformations.
The algorithm is usually defined as two sweeps on a binary tree. An up-sweep, from leaves to the root, produces a reduction of all data elements, in our case $x_{0,7}$. This procedure requires no more steps than a height of three which is equal to $\log_2 N$. Then, a down–sweep is performed, from the root to leaves, and in each iteration workers proceed

in a triple of a parent, left and right child. The left child, illustrated in the figure with a gray node, receives a partial result from its parent and the right child, depicted in the figure with a black node, computes another partial result. In practice, parent and right child refer to the same worker in different iterations. Once again, a full sweep requires $\log_2 N$ steps to finish. Listing 3 presents an example of the algorithm.

Algorithm 3 A pseudocode for Blelloch parallel prefix sum.

```
1: for i from 0 to log2 N − 1 do                          ▷ Up-sweep traversal of the tree.
2:     for j from 2^i to N step 2^(i+1) in parallel do
3:         data[j] = data[j − 2^(i+1)] ⊙ data[j]
4:     end for
5: end for
6:
7: data[N − 1] = I⊙                                       ▷ data[N − 1] stored the final reduction
8:
9: for i from log2 N − 1 to 0 do                          ▷ Down-sweep traversal of the tree.
10:    for j from 0 to N − 1 step 2^(i+1) in parallel do
11:        temp = data[j + 2^i − 1]                       ▷ Save value of left child.
12:        data[j + 2^i − 1] = data[j + 2^(i+1) − 1]      ▷ Copy value to left child.
13:        data[j + 2^(i+1) − 1] = temp ⊙ data[j + 2^(i+1) − 1]   ▷ Apply left
14:                                                       ▷ child's value to right child.
15:    end for
16: end for
```

The algorithm computes an exclusive prefix sum, but an inclusive prefix sum may be computed after a small modification. For a i-th worker, the inclusive value may be obtained either through one additional application of operator or by receiving the value of exclusive scan from $i + 1$-th worker. The former approach is preferable for applications where it is less expensive to apply the operator rather than communicate with other workers. This is especially significant on message-passing systems. On the other hand, the latter approach is desirable for examples where the binary operator is computationally intensive. In this scenario, the missing value for last worker $x_{0,N-1}$ is already computed by the same worker at the end of an up-sweep.
The span of Blelloch prefix sum is bounded by a double traversal of binary tree which scales logarithmically with the number of workers

$$S_B(N) \;=\; 2 \cdot \log_2 N \tag{3.4}$$

A number of applications of the operator scales linearly with the number of data ele-

ments. This makes the Blelloch algorithm work–efficient

$$\begin{aligned} W_B(N) &= 2 \cdot \sum_{i=0}^{\log_2 N - 1} \frac{N}{2^{i+1}} \\ &= 2 \cdot \frac{N}{2} \frac{1 - (\frac{1}{2})^{\log_2 N}}{1 - \frac{1}{2}} \\ &= 2 \cdot N \cdot (1 - \frac{1}{N}) \\ &= 2 \cdot (N - 1) \end{aligned} \tag{3.5}$$

3.3.2 Brent–Kung

Brent–Kung adder[29] is a work-efficient circuit design for parallel prefix sum. Figure 3.3 presents an example of Brent–Kung strategy on eight image deformations. The up–sweep traversal is exactly the same as in Blelloch prefix sum and produces a reduction of all data elements. In the down–sweep, a breadth–first search (BFS) is performed where only right child adds partial result from its parent. An implementation may not always synchronize between workers and given lack of data dependencies, some parts of second tree traversal may be performed earlier, even during the first tree visit. This improvement, however, won't improve algorithm's span and a less greedy version simplifies the description.
The strategy allows computing results in $2 \cdot \log_2 N - 1$ steps, making it slightly more efficient than Blelloch method. Listing 4 presents the algorithm.
Contrary to the Blelloch algorithm, Brent–Kung produces an inclusive prefix sum. An exclusive version of Brent–Kung[24], with an integrated rejection of final reduction and propagation of operator's identity, exhibits a design very similar to the Blelloch algorithm with a slightly better depth of Brent–Kung strategy.

Algorithm 4 A pseudocode for Brent–Kung parallel prefix sum.

```
1: for i from 0 to log2 N − 1 do                    ▷ Up-sweep traversal of the tree.
2:     for j from 2^i to N step 2^(i+1) in parallel do
3:         data[j] = data[j − 2^(i+1)] ⊙ data[j]
4:     end for
5: end for
6:     ▷ Iterations of an inner loop may be executed concurrently with the loop above
7: for i from log2 N − 2 to 0 do                    ▷ Down-sweep traversal of the tree.
8:     for j from 2^(i+1) to N − 1 step 2^(i+1) in parallel do
9:                                  ▷ Left child is visited without any computation.
10:        data[j + 2^i − 1] = data[j − 1] ⊙ data[j + 2^i − 1]
11:    end for
12: end for
```

The span of Brent–Kung prefix sum is bounded by a full traversal of the binary tree

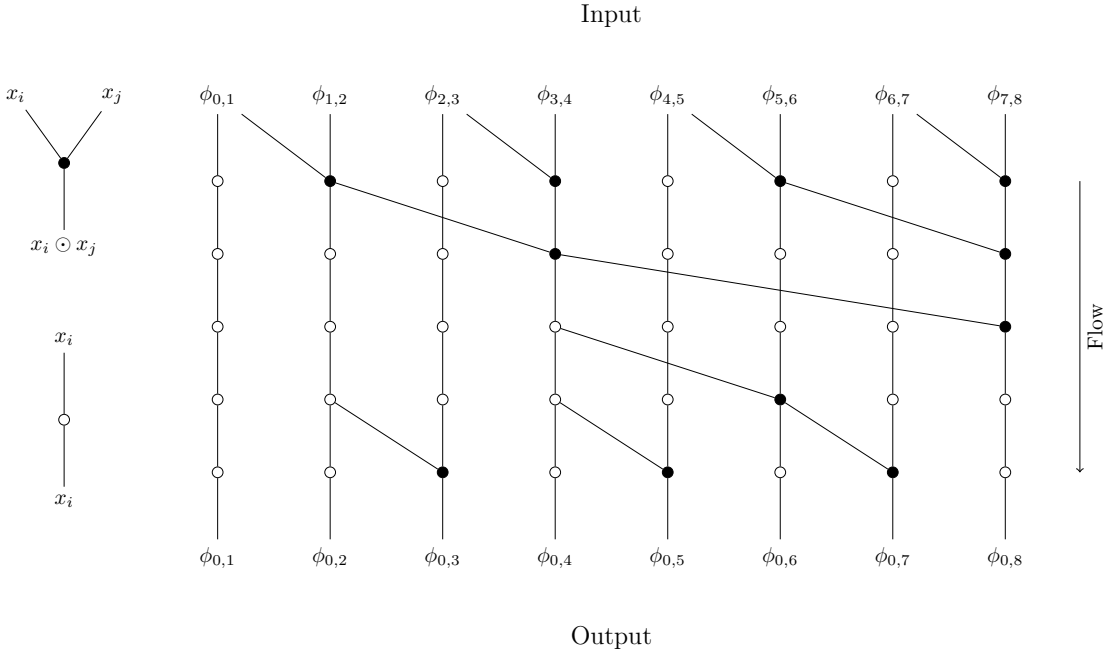


Figure 3.3: An example of Brent–Kung parallel prefix sum on 8 data elements. Results are produced in 5 steps and 11 applications of the binary operator.

and a breadth–first search where root does not perform any computation

$$S_B(N) \;=\; 2\log_2 N - 1 \tag{3.6}$$

Estimation of performed work is similar to Blelloch case. Brent–Kung strategy is work–efficient due to linear scaling of performed work

$$\begin{aligned} W_B(N) &= \sum_{i=0}^{\log_2 N-1} \frac{N}{2^{i+1}} + \sum_{i=0}^{\log_2 N-2} (\frac{N}{2^{i+1}} - 1) \\ &= N - 1 + \sum_{i=0}^{\log_2 N-2} \frac{N}{2^{i+1}} - (\log_2 N - 1) \\ &= N - 1 + \frac{N}{2}\frac{1 - (\frac{1}{2})^{\log_2 N-1}}{1 - \frac{1}{2}} - (\log_2 N - 1) \\ &= N - 1 + N - 2 - (\log_2 N - 1) \\ &= 2 \cdot N - \log_2 N - 2 \end{aligned} \tag{3.7}$$

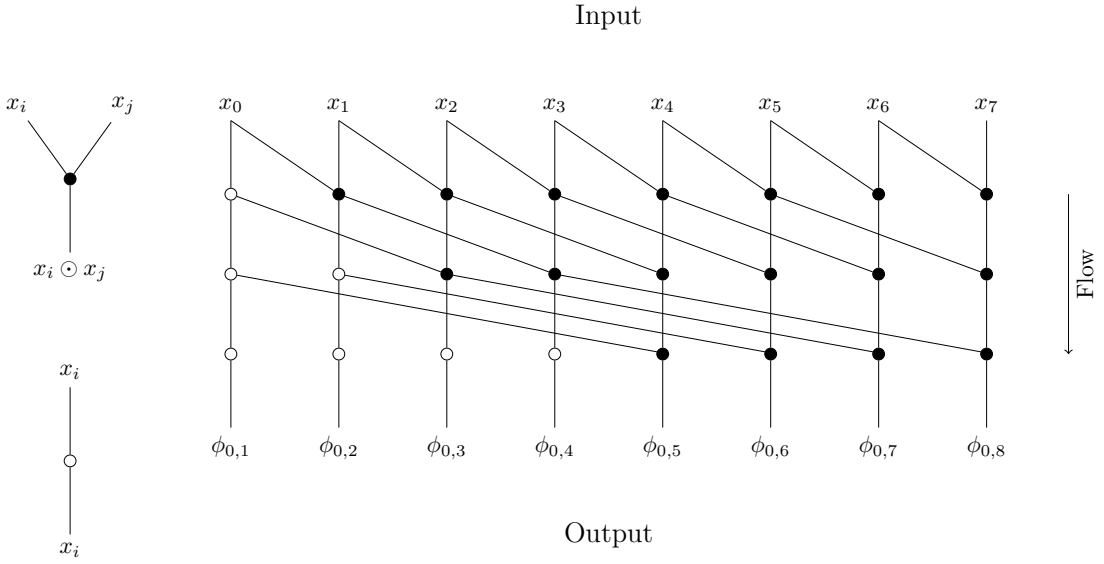


Figure 3.4: An example of Kogge–Stone a.k.a. Hillis–Steele parallel prefix sum on 8 data elements. Results are produced in 3 steps and 17 applications of the binary operator.

3.3.3 Kogge–Stone

This parallelization strategy is based on a Kogge–Stone parallel prefix adder, firstly proposed by Peter Kogge and Harold Stone in 1973[30]. In 1986, Hillis and Steele[31] described an application of the adder to PRAM model and the algorithm has been known under a different name *Hillis–Steele parallel prefix sum*. This algorithm has also been discussed under a name *recursive doubling algorithm*[32].
Figure 3.4 presents an example of parallel prefix sum with Kogge–Stone approach on eight image deformations. The strategy allows to compute results in $\log_2 N$ steps and each step produces $2^N - 1$ partial results. In each iteration, workers apply the binary operator to its own result and a partial result obtained from one of the workers on their left. Listing 5 presents the algorithm.
Kogge–Stone produces an inclusive prefix sum. An obvious difference with the Blelloch prefix sum is a much larger amount of work performed in all steps. Furthermore, the huge work intensity requires excessive communication. Internally, the algorithm is synchronous and each level depends on results from the previous iteration. This particular feature makes it sensitive to deviations in execution time between different applications of the binary operator $\odot$.
Another characteristic feature of the algorithm is the presence of Write-After-Read anti-dependencies, where a worker should not overwrite the previous result until it has been accessed. A common technique to resolve this problem is *double-buffering* where two distinct arrays are used to store partial results[23].

Algorithm 5 A pseudocode for Kogge–Stone parallel prefix sum.

```
1: for i from 0 to log2 N − 1 do
2:     for j from 2(i + 1) to N in parallel do
3:         data[j] = data[j − 2^i] ⊙ data[j]
4:     end for
5: end for
```

An estimation of time complexity for Kogge–Stone parallel prefix sum is trivial and the span is simply determined by the outer loop

$$S_{KS}(N) \;=\; \log_2 N \tag{3.8}$$

Work complexity is assessed by multiplying the number of steps with a number of active workers in each step

$$\begin{aligned} W_{KS}(N) &= \sum_{i=0}^{\log_2 N-1} N - 2^i \\ &= N \cdot \log_2 N - \sum_{i=0}^{\log_2 N-1} 2^i \\ &= N \cdot \log_2 N - \frac{1 - 2^{log_2 N}}{1 - 2} \\ &= N \cdot \log_2 N - N + 1 \end{aligned} \tag{3.9}$$

3.3.4 Sklansky

This strategy is based on a first parallel prefix adder described in 1960 by Sklansky[33]. This inclusive parallel prefix sum is similar to Kogge–Stone in work inefficiency and purely logarithmic span. The recursive nature of algorithm is visible on Figure 3.5. A divide–and–conquer approach splits the problem in half at each step, instating twice the same task for two halves of input data.
The strategy generates results in $\log_2 N$ steps. According to a recent report[24], their proposed algorithmic description for Sklansky prefix sum is the first iterative version of Sklansky prefix adder. We present a more verbose version of the algorithm on Listing 6. In our opinion, it is easier to follow the flow of execution with a triple–nested loop.
Comparing to previously introduced prefix adders, Sklansky is the only one to have a non-constant number of *fan-outs* i.e. outbound wires in a logical gate applying the operator. The example on Figure 3.5 shows how a number of outputs in a node changes from two to four. Furthermore, the algorithm involves a constant number of tasks per each iteration which simplifies the mapping of work to hardware in implementations such as SIMD architectures.

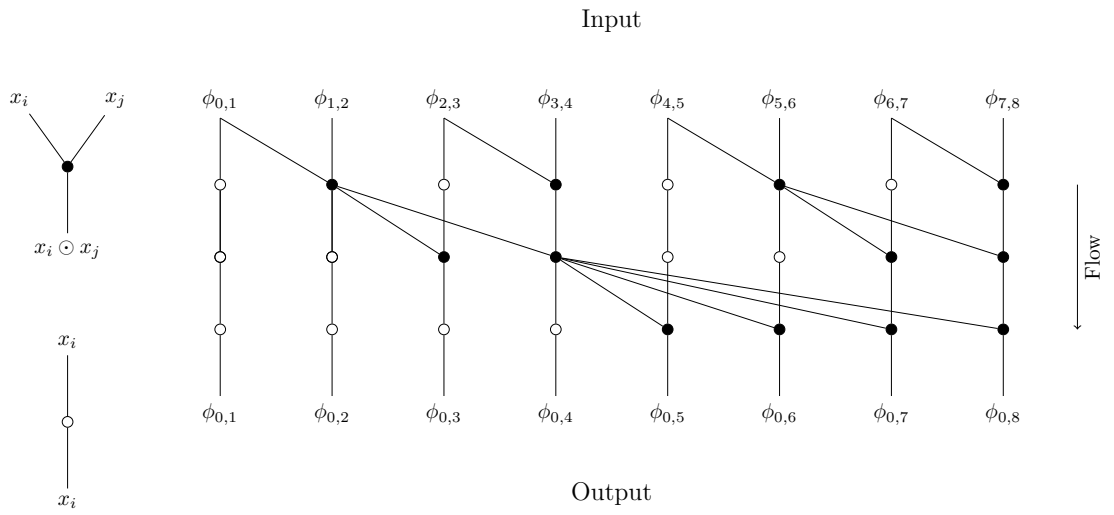


Figure 3.5: An example of Sklansky parallel prefix sum on 8 image deformations. Results are produced in 3 steps and 12 applications of the binary operator.

Algorithm 6 A pseudocode for Sklansky parallel prefix sum.

```
1: for i from 0 to log2 N − 1 do
2:     ▷ Iterate over sources of result from previous step.
3:     for j from 2^i − 1 to N step 2^(i+1) in parallel do
4:         ▷ Iterate over destinations for result from previous step.
5:         for k from 0 to 2^i in parallel do
6:             data[j + k + 1] = data[j] ⊙ data[j + k + 1]
7:         end for
8:     end for
9: end for
```

The span of the Sklansky prefix sum is given by a divide–and–conquer method needing $\log_2 N$ steps for N input elements

$$S_{SK}(N) \;=\; \log_2 N \tag{3.10}$$

Work complexity is straightforward as well. Each recursive call creates twice as many problems with a half of the original size, and therefore each level processes the same number of tasks

$$W_{SK}(N) = \frac{N}{2} \cdot \log_2 N \tag{3.11}$$

3.4 Relation between span and work

Introduced parallel prefix sum algorithms have different work complexities, but neither of them can match the purely linear complexity of a serial application. The intuition

suggests that the sequential nature of prefix sum allows for parallelization only by performing more work but in parallel. This intuition has been formalized and certain bounds for the relation between span and work of a parallel prefix sum has been proven. These results have been described with terminology appropriate for prefix adder circuits, but semantics stay the same and we can apply directly these results to PRAM algorithms.

An important theorem has been proposed and proved for prefix circuits by Snir[34] in 1986. The theorem introduces a relation between size and depth of a prefix circuit. The former property describes the number of nodes inside the circuit and corresponds to work complexity $W(N)$ in a parallel algorithm. The latter represents a delay introduced by the circuit and it primarily depends on the critical path. Therefore, this attribute corresponds to span in parallel computation model.

Theorem 3.4.1. Let $x_0, x_1, \ldots, x_{N-1}$ be a sequence of inputs and f_i be prefix sums computed with i first elements of input sequence. Let G be a prefix circuit that computes $f_1, \ldots, f_{N-1}$, with a size $s(G)$ and depth $d(G)$. Then

$$s(G) + d(G) \geq 2N - 2$$

Proof. The original proof can be found in[34]. Alternative proof not requiring induction can be found in[35]. □

An immediate corollary of this theorem is that each gain in improving parallelism, which reduces critical path of the algorithm, has to be compensated by performing more work. The question remains whether the excessive work is justified by an improvement in depth. The concept of zero–deficiency measures if there exists a linear trade-off between size and depth:

Definition 3.4.1. The deficiency of a prefix circuit is defined as

$$\mathrm{def}(G) = 2N - 2 - s(G) - d(G)$$

A parallel prefix circuit is said to be of zero–deficiency if $\mathrm{def}(G) = 0$

A method for constructing zero–deficiency prefix sums has been proposed for depths in the range $2\log_2 N - 2 \leq d(G) \leq N - 1$. In 2006 a lower bound for depth of zero–deficiency prefix circuit for a given N was proven[35]. Zero–deficiency prefix circuits do not exist below this boundary, including prefix circuits of minimal depth $\log_2 N$, such as Sklansky or Kogge–Stone. Therefore, the most span–optimal parallel algorithm for prefix sum can not achieve a linear work complexity.

A trivial example of zero-deficiency prefix sum is the serial algorithm with exactly $N - 1$ applications of a binary operator and $N - 1$ span.

3.5 Other work

A hybrid parallel prefix strategy has been proposed by Han and Carlson[36], where Brent–Kung and Kogge–Stone prefix adders are merged into a single algorithm. The

goal of a new layout is to leverage an optimal span of Kogge–Stone and a linear work complexity of Brent–Kung. The algorithm is parameterized by a non-zero constant k which controls the balance between span and work complexity. Span is equal to $k+\log_2 N$ and it has been proved[25] that a proper choice of k allows to bound work complexity by $\mathcal{O}(N \cdot \log_2 N)$ or even $\mathcal{O}(N)$. Thus, Han–Carlson strategy achieves asymptotically optimal span and work complexity.

Ladner and Fischer[8] proposed a general recurrence method of designing circuits for prefix adders. An interesting application of their method involves a prefix sum using Sklansky and Brent–Kung strategies to attain a minimum span and a slightly better work complexity[37][38]. However, some literature describes Sklansky prefix sum under the name of Ladner–Fischer scan[39].

3.6 Summary

We have described several different algorithms that have been developed and researched for parallelization of prefix sum. The next chapter defines and explains the methodology for choosing a right parallel prefix strategy for the problem of image registration.

Table 3.1 presents a comparison of discussed parallel prefix sum algorithms. Kogge–Stone with Hills–Steele remain a widely-used strategy due to its minimal span. On the other hand, Sklansky prefix adder does not seem to be as popular. Blelloch and Brent–Kung strategies are still highly influential and prevalent. Less popular prefix adders, such as Han–Carlson, Ladner–Fischer or Hockney–Jesshope have been recently researched for GPGPU architectures.

Name	Type	Span	Work
Sequential	Inclusive	$N-1$	$N-1$
Blelloch	Exclusive	$2 \cdot \log_2 N$	$2(N-1)$
Brent–Kung	Inclusive	$2 \cdot \log_2 N - 1$	$2 \cdot N - \log_2 N - 2$
Kogge–Stone	Inclusive	$\log_2 N$	$N \cdot \log_2 N - N + 1$
Sklansky	Inclusive	$\log_2 N$	$\frac{N}{2} \cdot \log_2 N$

Table 3.1: Comparison of major strategies for parallel prefix sum.

Chapter 4

Distributed prefix sum

In this chapter, we consider a distributed implementation of an inclusive parallel prefix sum described in chapter 3. Presented strategies have been developed to attain a best theoretical speedup when executing in a cluster environment. Furthermore, we demonstrate how the distributed prefix sum can be implemented within MPI programming model.
Algorithms accept input data of length N and operate on P workers with separate address spaces. A worker is expected to obtain multiple data elements. We use the terms *worker* and *process* interchangeably. For MPI–based implementations we use the term rank as well. Workers are allocated in a one–dimensional grid and for a worker with index I, we use terms *successor* and *right neighbor* for a worker with index $I + 1$, if it has one. Similarly, names *predecessor* and *left neighbor* are used interchangeably to refer to a worker with index $I - 1$.

4.1 General strategy

We have seen in the previous chapter that the span–optimal solution scales logarithmically with the number of workers when each one is responsible for one data element. Logarithmic complexity may be desired for a time complexity of a serial algorithm, but in a parallel algorithm it prevents any major improvements by spawning more workers on a larger set of cluster nodes. In a distributed setting, we expect to have significantly fewer workers than data. Hence, it is preferable to reduce the intra–process part to $\log_2 P$. Thus, this distributed stage requires one input value per a worker and a reduction step is required to transform the input data from length N to length P.

We name the first part of the general strategy a *local stage one*. It should accept the whole sequence of input data and end with a one value per process. For the purpose of this description, we do not assume any a priori knowledge which could suggest a specific data distribution policy. Without any hints on the actual running time of the binary operator on different operands, the safest choice is to split data as equally as possible over all workers. Each process is assigned $K = \frac{N}{P}$ input elements, if P evenly divides N. In other case, K shall be equal to $\lfloor \frac{N}{P} \rfloor + 1$, first $N \bmod P$ workers are assigned

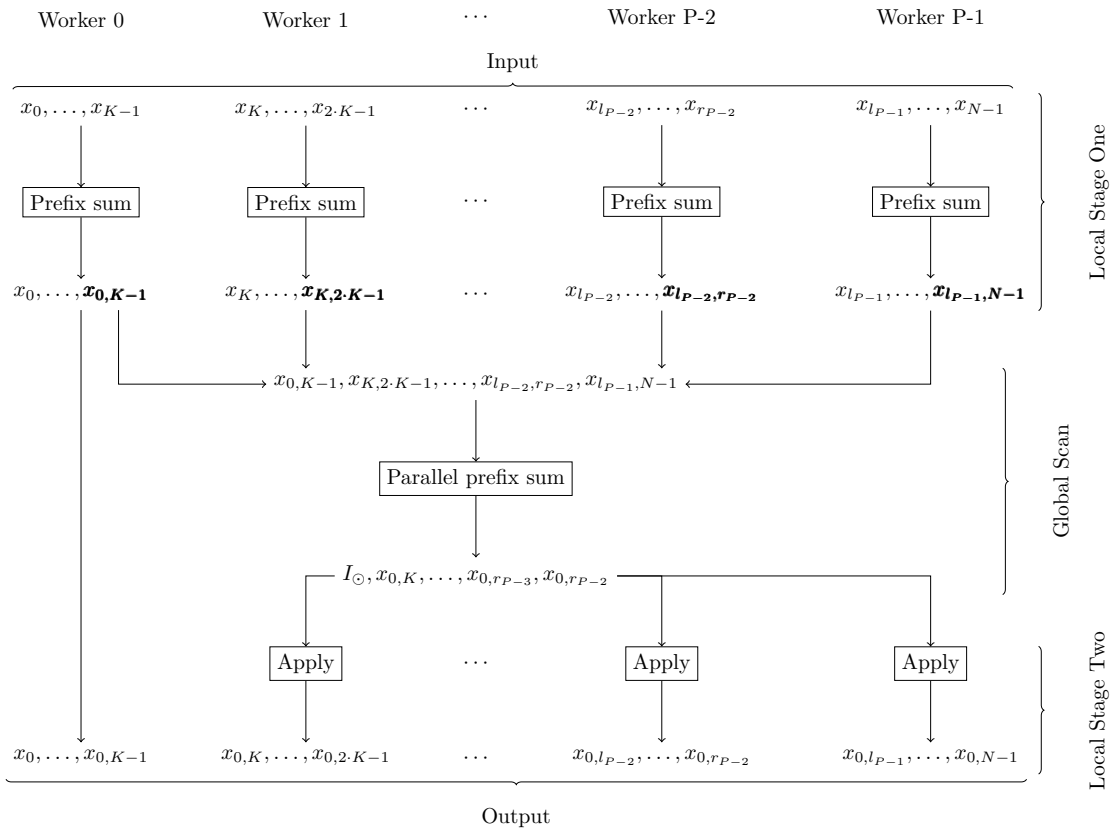


Figure 4.1: The general strategy for distributed prefix sum of N elements on P workers. An evenly distributed data across workers is passed to a local sequential prefix sum and the last computed value becomes an input to the global stage. Results from a global parallel prefix sum are applied on each worker except the first one.

K elements, and the rest obtains $K - 1$ data elements. For the sake of simplicity, we assume an evenly distributed data.
The logical distribution of work across processors follows a 1D grid where I-th worker is responsible for K input elements from $x_{K \cdot I}$ to $x_{K \cdot (I+1)-1}$. To simplify notation, helper variables l_I and r_I are introduced to store left and right boundary for a worker, with l_I equal to the index of first data element $K \cdot i$ and r_I equal to the index of last data element $K \cdot (i + I) - 1$. Data layout is depicted on Figure 4.1.

The first local stage is presented on lines 1–3 in Listing 7. Each worker performs performs independently a sequential prefix sum on the assigned chunk of data $x_{l_i}, x_{l_i+1}, \ldots, x_{r_i}$. Local data is overwritten with partial results and the last item is a reduction of all K data items $\odot_{i=0}^{K-1} x_{l_I+i}$.
Span is estimated as for a serial prefix sum

$$S_{LS1}(N, P) = S_S(\frac{N}{P}) = \frac{N}{P} - 1 \tag{4.1}$$

Work complexity is estimated as P workers performing a serial prefix sum

$$\begin{aligned} W_{LS1}(N,P) &= P \cdot S_S(\frac{N}{P}) \\ &= N - P \end{aligned} \tag{4.2}$$

Algorithm 7 Distributed parallel prefix sum of N data elements on worker I.

```
1: for i from 1 to K − 1 do
2:     data[i] ← data[i − 1] ⊙ data[i]                ▷ Local Stage One
3: end for
4: excl_scan ← parallel_scan(data[K − 1])             ▷ An exclusive prefix sum
5: if I > 0 then
6:     for i from 0 to K − 1 do                       ▷ Local Stage Two
7:         data[i] = excl_scan ⊙ data[i]
8:     end for
9: end if
```

The second stage, a global parallel scan, computes a prefix sum over local reductions. After the parallel scan, processor I should receive a value which allows combining its local results $\odot_{i=0}^{j} x_{l_I+j}$ with a reduction of all values assigned to preceding workers $0, 1, \ldots, I-1$. Consequently, the global prefix sum has to produce value $\odot_{i=0}^{r_{I-1}} x_i$ for worker I and the scan should be exclusive. Some of the proposed algorithms are by default inclusive, but they can be applied here without modifications changing their behavior, as described in section 4.1.2.
The last stage, presented on lines 5–9 in algorithm listing, operates again locally and independently from other processes. As soon as a result from the parallel stage has arrived, it is applied to each data item, and local results are transformed into partial results for a global prefix sum. The corner case here is the first worker who does not have any predecessors, and it does not perform any computation after first local stage. This stage is a single loop with K iterations, and therefore span and work analysis are trivial:

$$S_{LS2}(N,P) = \frac{N}{P} = S_{LS1}(N,P) + 1 \tag{4.3}$$

$$\begin{aligned} W_{LS2}(N,P) &= (P-1) \cdot \frac{N}{P} \\ &= N - \frac{N}{P} \end{aligned} \tag{4.4}$$

We summarize the strategy by combining estimations for local stages and an unknown span S_{GS} and work W_{GS} of a global scan which depends only on the number of workers, not on the input size. The span and work for a distributed scan is given as follows

$$\begin{aligned} S_{DS}(N,P) &= S_{LS1}(N,P) + S_{GS}(P) + S_{LS2}(N,P) \\ &= 2 \cdot \frac{N}{P} - 1 + S_{GS}(N,P) \end{aligned} \tag{4.5}$$

$$
\begin{aligned}
W_{DS}(N,P) &= W_{LS1}(N,P) + W_{GS}(P) + \cdot W_{LS2}(N,P) \\
&= 2 \cdot N - P - \frac{N}{P} + W_{GS}(N,P)
\end{aligned}
\tag{4.6}
$$

We must remark that this analysis of critical path is possible only in the condition of an even distribution of data. In the opposite case, a simple summation of the span for two local stages might yield an incorrect result if critical paths for those stages are provided by different workers. An example of such situation may be a prefix sum where $N \bmod P = 1$. There, the span of the first stage is given by the first worker who has one more data element than other processes, but it is inactive in the second local stage.

This general strategy attempts to minimize the asymptotically logarithmic global stage and perform locally as much computation as possible. The speedup of a local stage should scale linearly with an increase in a number of workers, and it is expected that for large values of P and small values of K, the global part is going to dominate the runtime of application because of its poor scaling.
Moving work from global to local stages may improve the runtime not only because of a better scalability of local stage. For all parallel prefix sum algorithms introduced in chapter 3, it holds that for the first step, each worker has to receive a value from its left neighbor. In this strategy, this value is the last result computed in the local stage which creates a Read–After–Write (RAW) dependency of the global scan on the local reduction stage. Any computational imbalance in the latter may influence the former, and the flow dependency increases the negative influence of time deviations on total runtime. Besides that, global scan requires sending partial results after each iteration which makes it even more sensitive to variations in execution time of the binary operator. Local stages are free of those dependencies.

4.1.1 Scan versus reduce

The general strategy presented above is not the only possible way of organizing work in a distributed prefix sum. One can notice that the global stage requires only the last value of local prefix sum which is also a result of performing a reduction on input data. Hence, preparing input for global stage does not require storing prefix sum, and those intermediate results can be recomputed in second local stage with a little cost.
After the global stage, a worker operates on received result from an exclusive scan and unmodified input data. To transform an input value x_{l_I+j} to $x_0 \odot x_1 \odot \cdots \odot x_{l_I+j}$, scan result is merged with the first element x_{l_I} and a new partial result $x_{0,l_I} = x_{0,r_{I-1}} \odot x_{l_I}$ is computed. Then, a sequential prefix sum would propagate changes from global scan and desired results are computed as $x_{0,l_I+j} = x_{0,l_I} \odot (x_{l_I+1} \odot x_{l_I+2} \odot \cdots \odot x_{l_I+j})$. Alternative strategy is presented in Figure 4.2 and in Listing 8. An alternative second local stage performs the same number of loop iterations as the old one but one additional application of binary operator is necessary

$$
S_{ALS2}(N,P) = \frac{N}{P} + 1 \tag{4.7}
$$

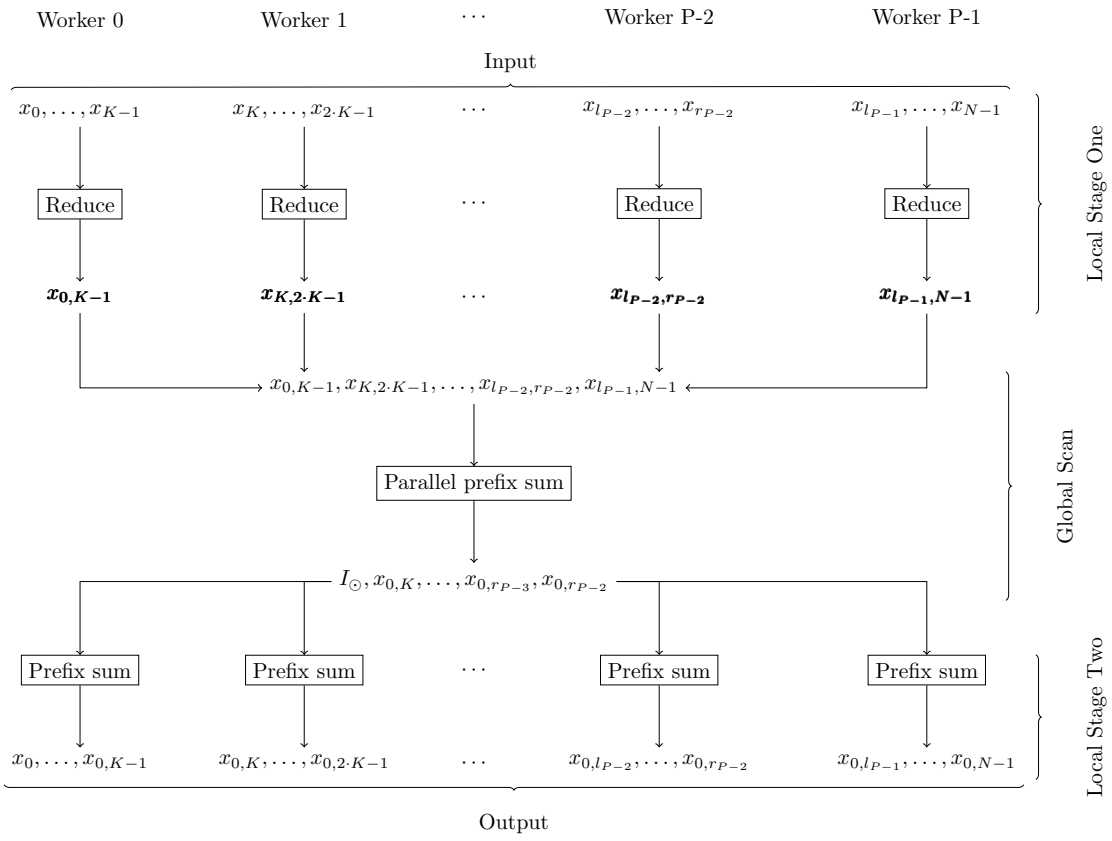


Figure 4.2: An alternative strategy of a distributed prefix sum of N elements on P workers. A local prefix sum is replaced by a reduction generating just one value as a result. In the second stage, the global scan value is applied to the first element of input data and a sequential prefix sum is performed.

More work is performed because the worker with index 0 is active in all stages, hence

$$\begin{aligned} W_{ALS2}(N,P) &= P \cdot (\frac{N}{P} + 1) \\ &= N + P \end{aligned} \tag{4.8}$$

And the span and work for an alternative distributed scan is as follows

$$\begin{aligned} S_{ADS}(N,P) &= S_{LS1}(N,P) + S_{GS}(P) + S_{ALS2}(N,P) \\ &= 2 \cdot \frac{N}{P} + S_{GS}(N,P) \end{aligned} \tag{4.9}$$

$$\begin{aligned} W_{ADS}(N,P) &= W_{LS1}(N,P) + W_{GS}(P) + W_{ALS2}(N,P) \\ &= 2 \cdot N + W_{GS}(N,P) \end{aligned} \tag{4.10}$$

Algorithm 8 An alternative distributed parallel prefix sum of N data elements on worker I.

1: $red \leftarrow data[0]$
2: **for** i **from** 1 **to** $K-1$ **do**
3: $\quad red \leftarrow red \odot data[i]$ ▷ Local Stage One
4: **end for**
5: $excl\_scan \leftarrow parallel\_scan(red)$ ▷ An exclusive prefix sum
6: $data[0] = excl\_scan \odot data[0]$ ▷ Apply scan result to first value
7: **for** i **from** 1 **to** $K-1$ **do** ▷ Local Stage Two
8: $\quad data[i] \leftarrow data[i-1] \odot data[i]$
9: **end for**

4.1.2 Inclusive global scan

In both strategies, an exclusive partial sum is required to start processing the second local stage. Hence, an exclusive prefix sum is preferred as an algorithm for the global scan. This, however, does not exclude the possibility of using an inclusive scan. All exclusive results are already computed, but they are not placed correctly on workers. Thus, an additional round of communication is necessary. A worker I sends its result to its successor $I+1$, if it has one, and receives a new result from the predecessor $I-1$.
A downside of this solution is that additional communication forces each worker to wait for its left neighbor and adds another flow dependency. Sadly, many prefix sum algorithms are inclusive by default. However, it is possible to benefit from this situation by reducing the computation cost in last local stage. An inclusive result for x_{0,r_I} is exactly the result of last loop iteration of the second local stages. Thus, one can directly assign this value and skip one loop iteration. Algorithm 9 formally defines the case for an inclusive global scan.

Algorithm 9 The general strategy with an inclusive scan. The inclusive result replaces the last loop iteration in the second local stage.

1: **for** i **from** 1 **to** $K-1$ **do**
2: $\quad data[i] \leftarrow data[i-1] \odot data[i]$ ▷ Local Stage One
3: **end for**
4: $incl\_scan \leftarrow parallel\_scan(data[K-1])$ ▷ An exclusive prefix sum
5: $excl\_scan \leftarrow blocking\_receive(I-1)$ ▷ Receive from left neighbor
6: **if** $I > 0$ **then**
7: $\quad$ **for** i **from** 1 **to** $K-2$ **do** ▷ Local Stage Two
8: $\quad\quad data[i] = excl\_scan \odot data[i]$
9: $\quad$ **end for**
10: $\quad data[K-1] \leftarrow incl\_scan$
11: **end if**

The span of a second local stage combined with an inclusive global scan is reduced by a constant factor. Nevertheless, this small improvement may be reduced or even eliminated by an additional synchronization after the global scan

$$\begin{aligned} S_{ILS2}(N,P) &= S_{LS2}(N,P) - 1 \\ &= \frac{N}{P} - 1 \end{aligned} \tag{4.11}$$

This optimization may be explored for exclusive prefix sums as well, especially if non–blocking communication is available. Scan result from a successor may reduce the computation in the second local stage by one loop iteration. Non–blocking communication can be applied to avoid synchronization because this value is not required until the second local stage is finished. Algorithm 10 presents the application of optimization to an exclusive scan.

Algorithm 10 An optimized general strategy with an exclusive scan. Function *probe_receive* is a simplified notation for a function intended to return a boolean true value only if a specific message has been received. In an actual MPI implementation, `MPI_Test` or `MPI_Wait` may be used.

```
1:  for i from 1 to K − 1 do
2:      data[i] ← data[i − 1] ⊙ data[i]                    ▷ Local Stage One
3:  end for
4:  excl_scan ← parallel_scan(data[K − 1])                 ▷ An exclusive prefix sum
5:  if I > 0 & I < P − 1 then                  ▷ Last worker does not have a successor
6:      incl_scan ← non_blocking_receive(I + 1)            ▷ Receive from neighbor
7:  end if
8:  if I > 0 then
9:      for i from 1 to K − 2 do                           ▷ Local Stage Two
10:         data[i] = excl_scan ⊙ data[i]
11:     end for
12:     if (I > 0 & I < P − 1) & probe_receive(incl_scan) then
13:         data[K − 1] ← incl_scan
14:     else
15:         data[K − 1] ← excl_scan ⊙ data[K − 1]
16:     end if
17: end if
```

We do not expect this change to reduce the theoretical span because it does not apply to the last worker which may leave the critical path unaffected. However, it is worth considering this small improvement in applications where differences in computation time lead to large imbalances. There, it is not uncommon for processes to finish much later than the last one, even if lengths of their computation paths are exactly the same.

4.2 Related work

Literature review reveals that an approach similar to the general strategy has been evaluated for GPGPU architectures. Our general strategy is known there under the name *scan–then–propagate*[40]. This strategy has also been presented as an algorithm for prefix sum where a limited number of processors is available[32].
The idea behind an alternative strategy has been presented in the algorithm for vectorization of prefix sum on CRAY Y-MP by Chatterjee et. al.[11]. In addition, this approach has been described as the *reduce–then–scan* strategy for GPGPU architectures[26]. There, it may outperform the general strategy because of a smaller global memory footprint in reduction phase.
Pipelined binary trees have been proposed for a distributed implementation of MPI scan collective[41]. Later, the performance of prefix sum in message–passing systems has been improved by exploiting a bidirectional communication[42][43]. This research has been focused on reducing the communication cost and improving bandwidth. Furthermore, only simple memory–bound operators have been evaluated. As explained in the next section, the requirements for a distributed prefix sum in image registration problem are quite different.

4.3 Registration problem

In the previous chapter, we have proved that the problem of image registration can be represented as a prefix sum with the function **B** as a binary operator. This function has several important properties which make our problem much more specific and different from applications and case studies analyzed in the literature.

We begin with the actual cost of applying the operator. The simplest case found in the literature, which happens to be the one most frequently evaluated, is an integer addition which should not take more than one CPU cycle on modern processors. More complex examples still involve relatively cheap operations, such as polynomial evaluation with floating–point multiplication and addition or summed area table where the binary operator performs multiple additions.
As a result, parallel prefix sum algorithms tend to be optimized for memory–bound applications with a rather low execution time of the operator. Image registration does not fit into this category. Figure 4.3 presents the execution time for a serial registration process. Each single image registration is much more computationally expensive and the actual execution time is of a completely different order of magnitude than a simple integer of floating–point number operation. It is very likely that different parallelization strategies may be required for a prefix sum with an operator taking several seconds to compute a single result.

Each non–rigid deformation stores only three floating–point values and the cost of sending this amount of data between cluster nodes is dominated by the latency, not the bandwidth. As a matter of fact, each execution of the registration function requires corresponding image data which is stored on the disk. We assume that each worker has

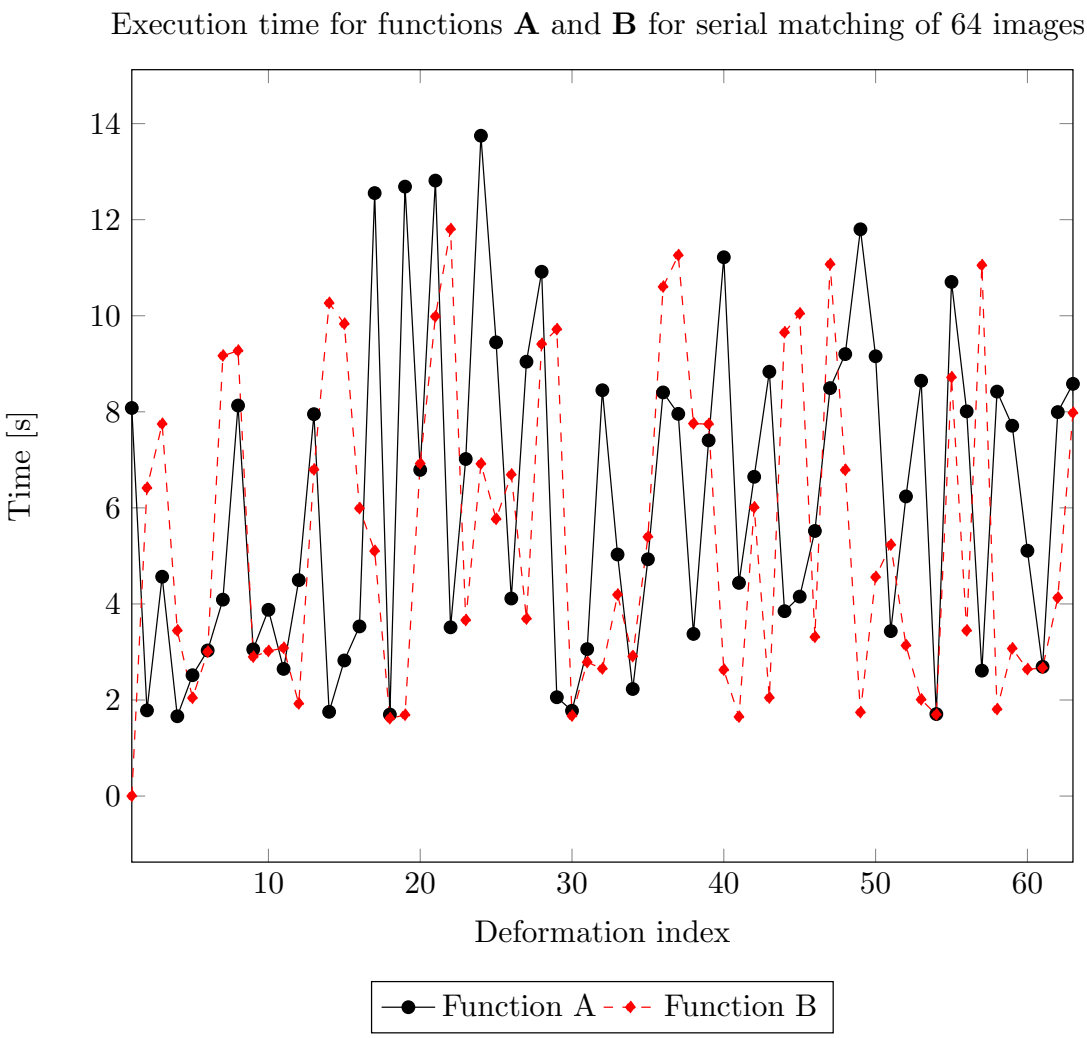


Figure 4.3: An example of serial registration for 64 images. The function **A** has been applied 64 times to create neighbor deformations $\phi_{i,i+1}$ and a serial prefix sum applied function **B** 63 times to generate final deformations $\phi_{0,i}$.

access to each image file through usual and serial I/O operations.
The cost of sharing data between workers is negligible when compared with the computation time. Because of this difference, our problem will not benefit from prefix sum algorithms developed for message–passing systems which concentrated on minimizing the number of communication cycles.

In contrast to operations with a deterministic execution time, here in both functions **A** and **B** the actual computation cost is not only unpredictable but highly variant. Due to the iterative nature of problem solved by the two functions, as explained in Chapter 2, we can not foresee for a given input data how many iterations are necessary to reach a

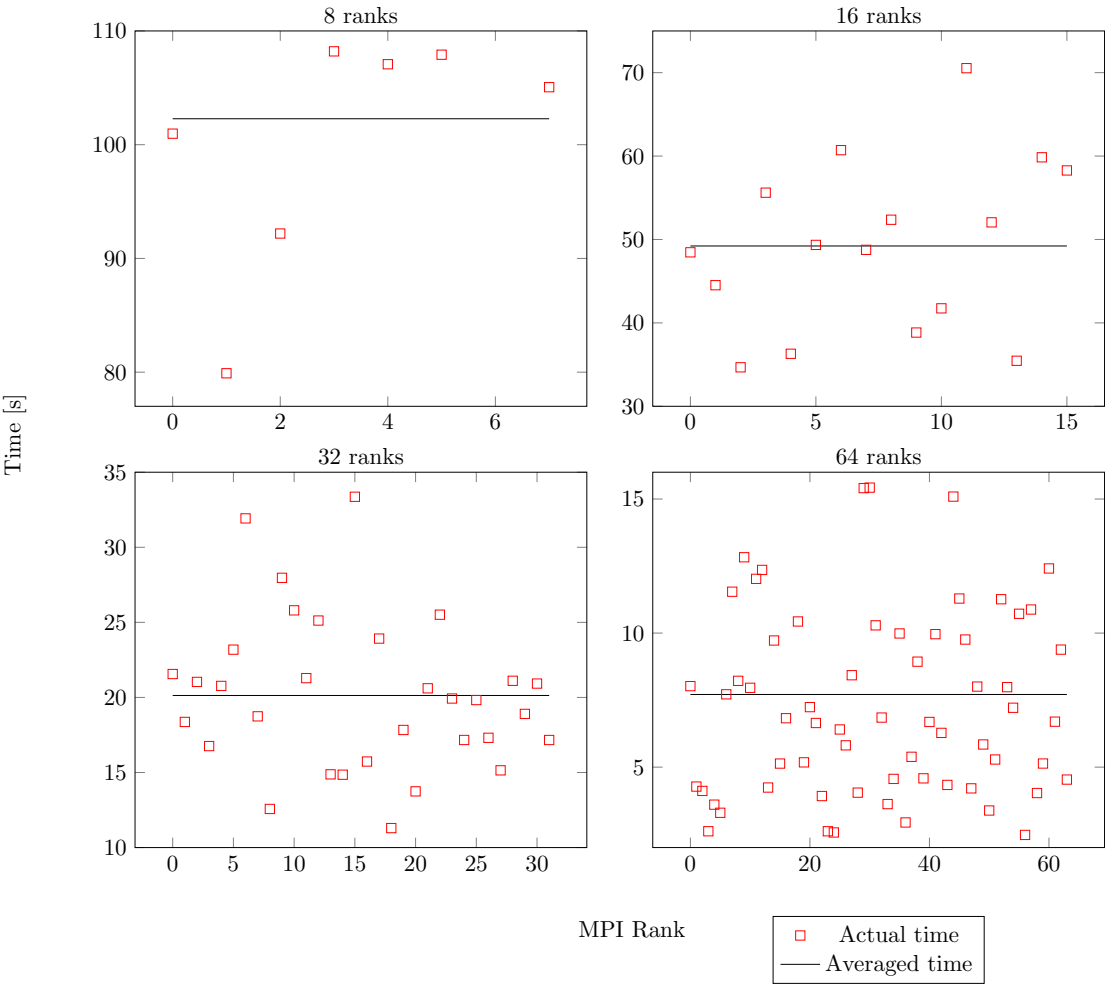


Figure 4.4: A comparison of the execution time of the first local stage for $N = 128$ image registrations within an MPI implementation. Four cases are presented, with P values varying from 8 to 64. Red squares represent execution time for each worker and the black line represents a theoretical mean, an ideal distribution of work in an imaginary case where all operator applications take the same amount of time.

stopping criterion. Figure 4.4 presents execution times of the first local stage for different workers. The black line represents an average computation time across workers. This scenario is not feasible, but it allows us estimating the delay introduced by an unequal load balance. In the first iteration of the global stage, workers are required to wait for a result from its predecessor, and it is not unusual that this idle time can be as large as the total runtime of local stage.

Regarding actual implementation, our operator depends on external objects supplying data structures such as multiple levels of a multigrid. Since constructing these objects in each application of the operator adds an undesired overhead, it is preferred to implement an operator able to capture external objects. We summarize the characteristics of our problem in several suggestions. We have used these pieces of advice to choose which parallel prefix algorithms, strategies, and optimizations may be interesting to our problem.

- **Prefer replacing computation with communication**
  In contrast to usual design principles for prefix sum in a distributed environment, we do not optimize to reduce the latency of data transmission – quite the opposite, computational intensiveness of our application suggest that we should prefer sharing partial results between workers if it allows reducing the amount of work to perform.

- **Do more work but in parallel**
  A lot of research have been done to design prefix sums with a logarithmic span and optimal, linear work complexity. For image registration, the most promising algorithms are the one who attains a minimum span with a non–linear work complexity.

- **Reduce dependencies when computation time is not predictable**
  There is a very little we could do with work distribution when no a priori knowledge on time imbalance is present. We can, however, look for parallel algorithms with the minimum number of flow dependencies.

4.4 MPI implementation

In this section, we present an MPI implementation of the general strategy combined with various algorithms for a global parallel prefix sum. Work, span and speedup estimations are provided.

4.4.1 MPI scan

MPI defines two functions to perform parallel prefix sum, `MPI_Scan` for an inclusive and `MPI_Exscan` for an exclusive prefix sum. Besides that, non–blocking alternatives `MPI_Iscan` and `MPI_Iexscan` have been added in MPI 3.0. The C declaration of `MPI_Scan` function is presented on Listing 4.1.

```
int MPI_Scan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype
    datatype, MPI_Op op, MPI_Comm comm);
```

Listing 4.1: An interface of MPI function for an inclusive parallel prefix sum.

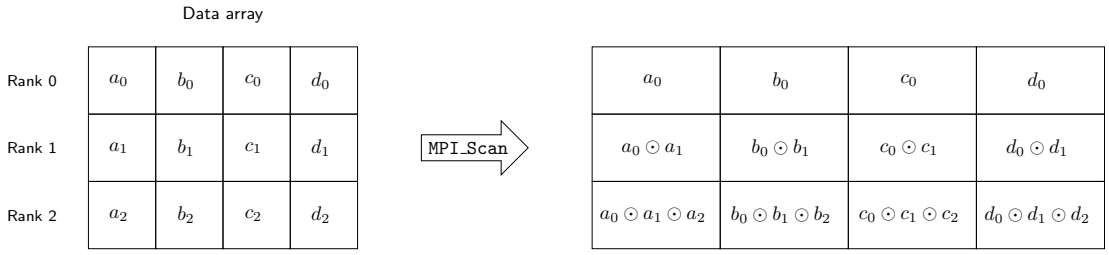


Figure 4.5: An example of executing `MPI_Scan` on three ranks with four data elements per rank. After the execution, each array element stores a result corresponding to one of four different scans.

The operation requires all ranks to pass the same arguments for *count*, *datatype* and *comm*. Input data is provided in *sendbuf* and results are written in *recvbuf*. A user–defined operator is assumed to be associative and can be declared as commutative to enable selection of more optimized algorithms. A C prototype of an operator is presented on Listing 4.2.

```
void MPI_User_function(void* invec, void* inoutvec, int *len,
    MPI_Datatype *datatype);
```

Listing 4.2: An interface of MPI user–defined operator for reducion function.

The function is expected to apply *len* times the operation $inoutvec[i] = invec[i] \odot inoutvec[i]$. The scan operator is applied elementwise to each element of the input buffer. Consequently, for *count* data elements passed to an MPI scan on each rank, *count* different prefix sums are computed. Given this limitation, existing MPI algorithms can not be used outside the general strategy for parallel prefix sum unless there is exactly one data item per rank. Otherwise, a specific type of segmented prefix sum is computed with only one data item per scan allowed on each rank. Figure 4.5 demonstrates this problem. MPI literature does not describe the problem in detail, nor it proposes any idea alternative to already described general scheme.
The MPI specification does not put performance requirements for an implementation. We provide a brief overview of algorithms present in two evaluated MPI implementations.

OpenMPI

OpenMPI documentation does no provide any information on actual implementation. An investigation of current source code[47] revealed an `MPI_Scan` implementation in *ompi/mca/coll/basic/coll_basic_scan.c* and an `MPI_Exscan` in *ompi/mca/coll/basic/coll_basic_exscan.c*. Both have been implemented with a fully serial algorithm where an MPI rank receives a value from its predecessor, computes a new value and sends it to its successor.

The same algorithm can be found in an implementation of `MPI_Iscan` in *ompi/mca/coll/libnbc/nbc_iscan.c* and `MPI_Iexscan` in *ompi/mca/coll/libnbc/nbc_iexscan.c*.

IntelMPI

IntelMPI documentation shortly mentions types of algorithms implemented for the prefix sum[44]. The selection of algorithm can be manipulated by setting environment variable `I_MPI_ADJUST_SCAN` or `I_MPI_ADJUST_EXSCAN` with an integer value corresponding to algorithm selection.
For an inclusive scan, two algorithms are offered to the user: *Partial results gathering* and *Topology aware partial results gathering*. An exclusive scan can be computed with two algorithms, the first one is the same as in inclusive scan and the other one is called *Partial results gathering regarding layout of processes*.

Unfortunately, the documentation does not describe algorithms in detail and the available information is limited only to these names.

4.4.2 Serial

With an exclusive sequential global prefix sum in the global stage, the span of a distributed scan becomes

$$\begin{aligned} S_{DS}(N,P) &= S_{LS1}(N,P) + S_{GS}(P) + S_{LS2}(N,P) \\ &= \frac{N}{P} - 1 + P - 2 + \frac{N}{P} \\ &= 2 \cdot \frac{N}{P} + P - 3 \end{aligned} \tag{4.12}$$

Then, the theoretical speedup is given as

$$SP_{Serial}(N,P) = \frac{N-1}{2 \cdot \frac{N}{P} + P - 3} \tag{4.13}$$

This term reaches its maximum value when denominator reaches its minimum. Removing constants simplifies the term to minimize to

$$\frac{2N}{P} + P \tag{4.14}$$

and as the literature suggests[10], it may be interpreted as doubled arithmetic mean of $\frac{2N}{P}$ and P. Arithmetic mean is bounded from below by the geometric mean

$$\frac{\frac{2N}{P} + P}{2} \geq \sqrt{\frac{2N}{P} \cdot P} \tag{4.15}$$

Arithmetic and geometric means are equal if and only if both operands are equal which implies that for a fixed N the minimum is obtained for

$$\begin{aligned} P &= \frac{2N}{P} \\ P &= \sqrt{2N} \end{aligned} \tag{4.16}$$

This result proves that for any N the best speedup is obtained for a number of processors much smaller than N. Thus, the maximum speedup is

$$\begin{aligned}\max_{P=1}^{P=N} SP_{Serial}(N,P) &= \frac{N-1}{\frac{2N}{\sqrt{2N}} + \sqrt{2N} - 3} \\ &= \frac{\sqrt{2N}\cdot(N-1)}{4N - 3\sqrt{2N}}\end{aligned} \tag{4.17}$$

As we can see, even for quite large input the attainable speedup is very low, no matter how many processor cores are available. The limit on scalability and very poor upper bound on speedup make this solution not attractive for a distributed implementation.

4.4.3 Blelloch

With a double sweep of a tree, the span is equal to

$$\begin{aligned}S_{DS}(N,P) &= S_{LS1}(N,P) + S_{GS}(P) + S_{LS2}(N,P) \\ &= \frac{N}{P} - 1 + 2\cdot\log_2 P + \frac{N}{P} \\ &= \frac{2N}{P} - 1 + 2\cdot\log_2 P\end{aligned} \tag{4.18}$$

and the theoretical speedup of distributed scan becomes

$$SP_{Blelloch}(N,P) = \frac{N-1}{\frac{2N}{P} - 1 + 2\cdot\log_2 P} \tag{4.19}$$

The maximum of this function is found by minimizing the denominator

$$\begin{aligned}f(P) &= \frac{2N}{P} - 1 + 2\cdot\log_2 P \\ \frac{df}{dP} &= -\frac{2N}{P^2} + \frac{2}{P\ln 2} \\ &= \frac{2P - 2N\ln 2}{P^2\ln 2}\end{aligned} \tag{4.20}$$

The derivative reaches zero when

$$P_0 = N\ln 2 \tag{4.21}$$

To find out whether it is a maximum or minimum, second derivative is tested

$$\begin{aligned}\frac{d^2 f}{dP^2}|_{P=N\ln 2} &= \frac{4N}{P^3} - \frac{2}{P^2\ln 2}|_{P=N\ln 2} \\ &= \frac{4N\ln 2 - 2P}{P^3\ln 2}|_{P=N\ln 2} \\ &= \frac{2N\ln 2}{N^3\ln^4 2}\end{aligned} \tag{4.22}$$

Obviously, the second derivative is greater than zero for any positive value of N. Hence, P_0 is a local minimum of $f(P)$ and a local maximum of S. The important result here is that critical point is always greater than N, implying that for practical values of P it is always possible to improve the result by adding more processors until the number of processors reaches the number of data elements.

Work performed at all stages is equal to

$$\begin{aligned} W_{DS}(N,P) &= 2 \cdot N - P - \frac{N}{P} + W_{GS}(P) \\ &= 2 \cdot N - \frac{N}{P} - P + 2 \cdot (P-1) \\ &= 2 \cdot N - \frac{N}{P} + P - 2 \end{aligned} \tag{4.23}$$

which proves that work complexity is linear in both input size and number of workers.

4.4.4 Kogge–Stone

With an inclusive scan, we apply the optimization and the span of distributed scan becomes

$$\begin{aligned} S_{KS}(N,P) &= S_{LS1}(N,P) + S_{GS}(P) + S_{ILS2}(N,P) \\ &= \frac{N}{P} - 1 + P - 2 + \frac{N}{P} - 1 \\ &= \frac{2N}{P} - 2 + \log_2 N \end{aligned} \tag{4.24}$$

The analysis of speedup is similar to Blelloch algorithm, with the derivative

$$\frac{df}{dP} = \frac{2P - N \ln 2}{P^2 \ln 2} \tag{4.25}$$

Here the local maximum is reached for

$$P_0 = 2 \cdot N \ln 2 \tag{4.26}$$

Which is, again, an impractical value. Work performed at all stages is

$$\begin{aligned} W_{DS}(N,P) &= 2 \cdot N - P - \frac{N}{P} + W_{GS}(P) \\ &= 2 \cdot N - \frac{N}{P} - P + P \cdot \log_2 P - P + 1 \\ &= 2 \cdot N - \frac{N}{P} + P \cdot (\log_2 P - 2) + 1 \end{aligned} \tag{4.27}$$

which proves that work complexity is linear in input size but $\mathcal{O}(n \log n)$ in number of workers.

4.4.5 Sklansky

Sklansky parallel prefix adder has the same span as Kogge–Stone algorithm. Therefore, attainable speedup is exactly the same. The work complexity is slightly different but asymptotically equal to work performed by Kogge–Stone prefix sum

$$\begin{aligned} W_{DS}(N,P) &= 2 \cdot N - P - \frac{N}{P} + W_{GS}(P) \\ &= 2 \cdot N - \frac{N}{P} - P + \frac{P}{2} \cdot \log_2 P \\ &= 2 \cdot N - \frac{N}{P} + P \cdot (\frac{\log_2 P}{2} - 1) \end{aligned} \tag{4.28}$$

4.4.6 Summary

We select for evaluation several instantiations of our general strategy

- **MPI–based**
  Global scan implementation provided by the MPI library. Evaluate inclusive and exclusive variants against OpenMPI and different implementations of the scan in IntelMPI.
- **Blelloch**
- **Kogge–Stone, Sklansky**

Figure 4.6 presents a theoretical prediction of attained speedup for selected variants. Clearly, the small constant factor in a span between Blelloch and Kogge–Stone prefix sums may lead to a large difference. Figure 4.7 compares selected strategies against a linear scaling to demonstrate that even with an ideal implementation, the parallel prefix sum is a problem which can not be efficiently parallelized on a large number of processor cores.

We expect to find answers for key questions:

- How these algorithms perform with a computationally intensive operator?
- How load imbalance influences different algorithms? Does the theoretical promise of scalability holds up?
- Is there a gain in selecting an algorithm with a slightly better span but noticeably worse work complexity?
- What is the quality of prefix sum provided by MPI implementations?
- How does an inclusive and exclusive MPI algorithm perform in the general strategy?

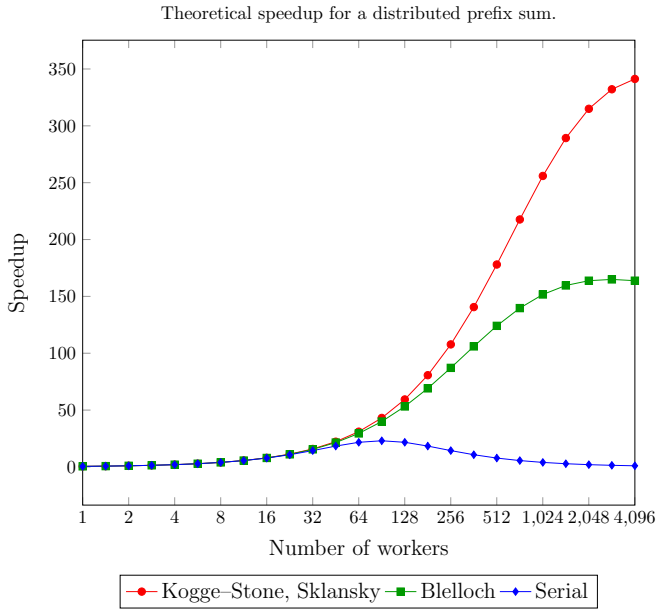


Figure 4.6: Theoretical speedup attainable by the general strategy with various implementation of the global stage. The horizontal axis is logarithmic with a base of two.

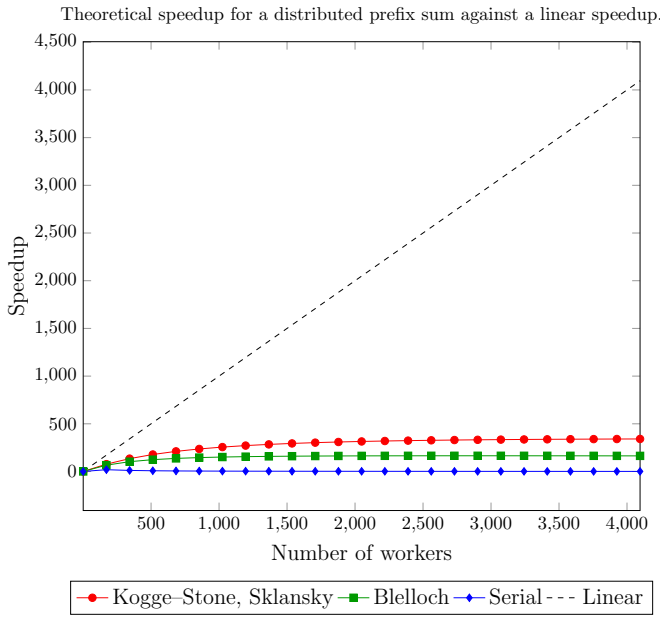


Figure 4.7: A comparison of an ideal linear speedup against the theoretical speedup attainable by the general strategy with various implementation of the global stage.

Chapter 5

Results

This chapter presents and analyses evaluation of parallel prefix sum algorithms for image registration. Experiments have been performed on cluster nodes consisting of Intel Xeon E5-2680 v2 CPUs. Each node contains two ten–core processors with frequency of 2.80 GHz and 3.60 GHz in TurboBoost mode. Our compiler of choice is GCC in version 5.3.0. MPI libraries selected for evaluations are OpenMPI in version 1.10.4 and IntelMPI in version 2017.1. Each measurement has been performed with the help of `MPI_Wtime` function, preceded by an `MPI_Barrier` as it is advised in the MPI standard[19]. Timings have been averaged over five repetitions and the standard deviation has been calculated to ensure the quality of results.
A comparison of different parallel prefix strategies in terms of strong and weak scalability is presented in sections 5.1 and 5.2, respectively. The section 5.3 provides a detailed view of MPI performance in the parallel prefix sum problem. In the section 5.4, a brief comparison of the general and alternative strategy is provided. Results of experimental shared–memory parallelization of the operator are presented in section 5.5.

5.1 Strong scaling

In this section, we investigate the parallel performance of a distributed prefix sum when the number of processor cores is increased from 1 to 512 and the problem size is fixed. Thus, the size of work chunk for each worker is decreased. An ideal linear speedup is not possible to obtain in our problem because of an increased workload in the parallel execution. For the comparison, we select results from following parallel prefix sums

- **Kogge–Stone, Sklansky**
  our implementation of work–inefficient prefix adders with a logarithmic span
- **Blelloch**
  our implementation of a classical work–efficient scan
- **OpenMPI**
  an inclusive MPI collective function `MPI_Scan`

- **IntelMPI**
  an exclusive MPI collective `MPI_Exscan` with the algorithm *Partial results gathering regarding layout of processes*

For MPI libraries, we have selected the best performing implementation for this comparison. A detailed overview is presented in section 5.3.
The speedup is measured as a ratio of serial and parallel execution time. For all results, the standard deviation of measured speedup $\mathcal{SP}$ has not exceeded 0.1. Thus, we can safely compare various algorithms even when the difference in speedup is rather small. A standard deviation of speedup $\mathcal{SP}$ is given as a

$$\sigma_{\mathcal{SP}} = \mathcal{SP}\sqrt{(\frac{\sigma_{t_1}}{t_1})^2 + (\frac{\sigma_{t_2}}{t_2})^2} \tag{5.1}$$

where times t_1 and t_2 with their respective deviations σ_{t_1} and σ_{t_2} are measured runtime of a serial and parallel execution.

Figure 5.1 presents the image registration of 4096 frames. We compare our implementations against the theoretical upper bound, with Sklansky and Kogge–Stone implementations having the highest bound due to the lowest span of the global stage. We expect that an ideal implementation should be able to achieve such speedup for problems with a low variation in execution time for the binary operator of prefix sum.
We do not observe any significant difference in experiments with less than 32 MPI ranks, where each process has a large work chunk and the cost of the global stage is relatively low. For larger runs, the best performance is attained by a Kogge–Stone implementation. Interestingly enough, the Sklansky implementation performs worse even though both algorithms have the same span. Moreover, the Sklansky prefix sum has a lower work complexity which should indicate a better performance. A possible explanation of this phenomena may be the non–constant *fan–out* of Sklansky algorithm, as explained in section 3.3.4. In both algorithms, $\frac{N}{2}$ workers are active in the last iteration but their dependencies are different. In the Kogge–Stone prefix sum, each active worker depends on a result from a different worker. In the Sklansky algorithm, all active workers have to wait for a result from the same worker with index $\frac{N}{2} - 1$. Therefore, a significant delay on this worker has a much serious influence on the total performance. The work–efficient Blelloch scan performs slightly worse than a Kogge–Stone approach. A large work complexity seems to not be a problem in the image registration, as long as the span is minimal.
We do not have any expectations about the performance of an exclusive IntelMPI implementation because of a lack of details about the algorithm in the documentation. It behaves similarly to the Blelloch prefix sum, which might suggest that the tree–based scan has been implemented as `MPI_Exscan`. At the same time, the source code analysis of OpenMPI suggested a serial implementation of the scan operation and measurements confirm this hypothesis. This implementation performs very poorly and it is simply unable to scale beyond a certain upper limit.

Results of a smaller experiment are presented in Figure 5.2. Here, the execution time is even more dominated by the global stage, because of a much smaller workload in local

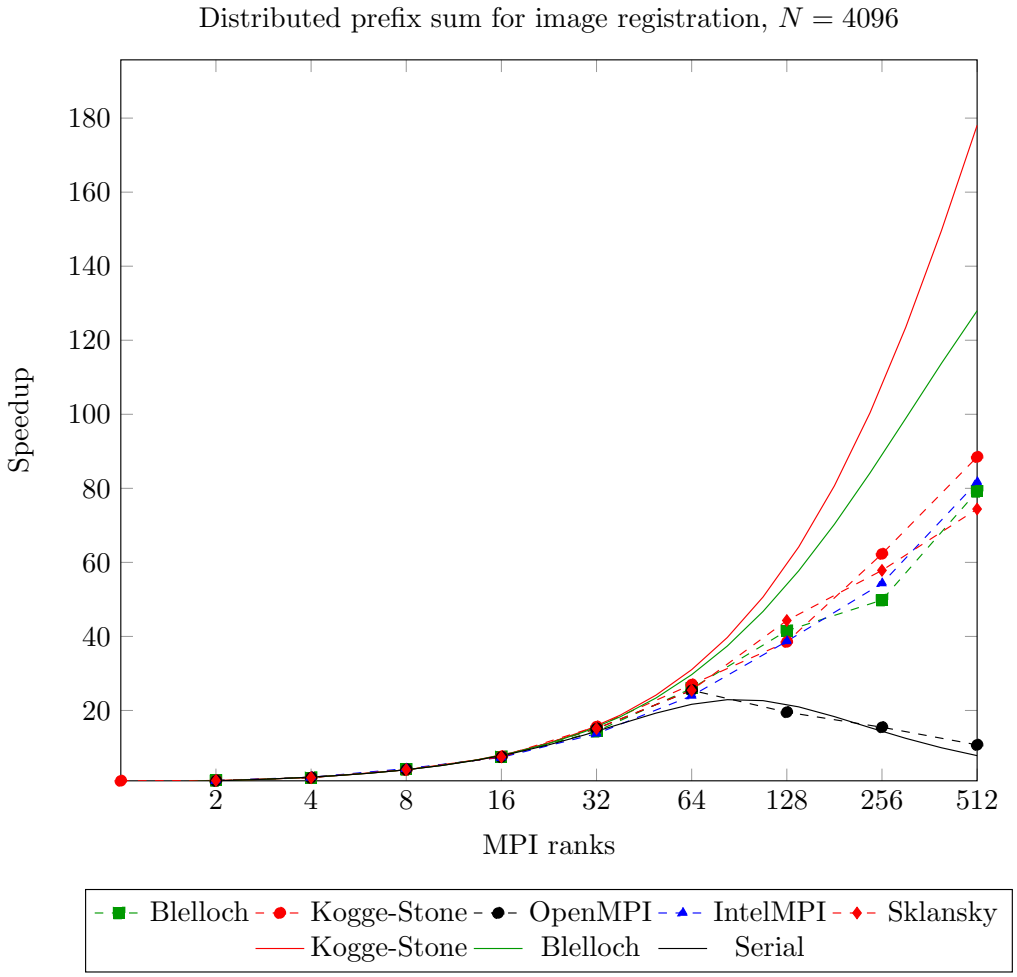


Figure 5.1: Speedups of different implementations of a distributed prefix sum, plotted as dashed lines, are compared with the theoretical upper bounds, presented with solid lines. The horizontal axis is logarithmic with a base of two.

stages. A corner case is $P = 512$ where each worker obtains only one data element, and no computation is done in local phases. Three implementations - Kogge–Stone, Sklansky and IntelMPI - attain the best speedup for $P = 256$. Surprisingly, the Blelloch algorithm improves even for $P = 512$. Although this result proves that the work–efficient Blelloch algorithm can perform better in some scenarios than a span–optimal Kogge–Stone algorithm, we note that this is a pathological case of $N = P$. Thus, Kogge–Stone is a preferable choice for real–world applications.

It is not surprising that the efficiency of all algorithms drops quickly on a larger number of MPI ranks. An increased number of workers means that more iterations are performed in the global stage and each one involves a blocking receive of results from

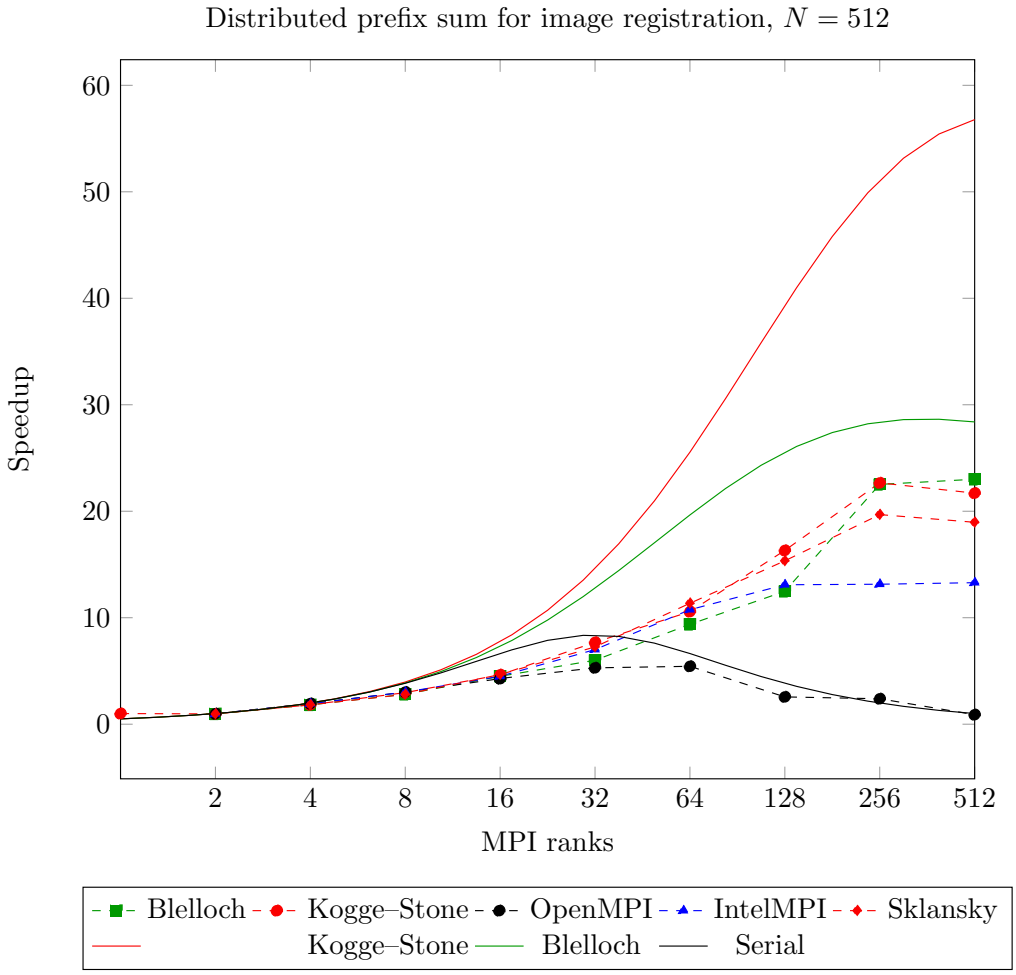


Figure 5.2: Speedups of different implementations of a distributed prefix sum, plotted as dashed lines, are compared with the theoretical upper bounds, presented with solid lines. The horizontal axis is logarithmic with a base of two.

another worker. This synchronization increases the likelihood of a rank idle waiting due to an unequal distribution of workload.

5.2 Weak scaling

In the case of weak scaling, the performance is measured for a fixed amount of work performed by each worker. In contrary to strong scaling, where the algorithm is stressed to utilize all available parallelism, weak scaling answers the question: is the algorithm able to solve a bigger problem without a decrease in efficiency, when more hardware is available? For an ideal linear scaling, the algorithm should be able to match an available parallelism of more processors with an increased size of the problem and the execution should stay constant.
In a distributed prefix sum, the span is given in a general form

$$S(N, P) = \frac{2N}{P} + C_1 \log_2 P + C_2 \tag{5.2}$$

for some constants $C_1, C_2 \in \mathbb{N}$. For an equal increase in the problem size and the number of processing elements, the span of a prefix sum is given as

$$S(2N, 2P) = \frac{4N}{2P} + C_1 \log_2 2P + C_2 \tag{5.3}$$

$$= \frac{2N}{P} + \log_2 P + C_2 + C_1 \tag{5.4}$$

Thus, we can not expect that the execution time stays constant. Nevertheless, as long as the amount of work per worker is relatively high, we should observe that the increase in span does not cause a significant growth of execution time.

We begin with considering the case $\frac{N}{P} = 8$. The smallest and largest experiments have been performed with 128 and 4096 images, respectively. Results are presented on Figure 5.3. The IntelMPI implementation has the lowest increase in time of 34% but it has been the slowest solution at the beginning. Kogge–Stone and Blelloch perform similarly with 50% and 51.6% increase, respectively. Sklansky algorithm outperforms all competitors most of the time but the sudden decrease in the least measurement gives it the worst result - 77% increase in execution time. On average, the increase in execution time is approximately equal to 42.8%. The OpenMPI solution is excluded from this analysis due to a large difference between its execution time and all other algorithms. It is sufficient to say that the runtime increases from 196 to 2041 seconds i.e. by 941%.
The analysis reveals more interesting information about the algorithms. Although Kogge–Stone provided the best performance in large runs, it is the only algorithm to not have a monotonic increase in execution time[1]. The significant increase from 16 to 64 ranks, followed by a sharp decrease on 128 ranks is a very unusual behavior. The variance of measurements is relatively low and we have found no reason to doubt the quality of conducted experiments. A possible explanation may be a huge sensitivity to load imbalance, causing a serious performance degradation for specific distributions of work.

[1]The Sklansky algorithm exhibits a small decrease between 64 and 128 ranks but the decrease is lower than the standard deviation of measurements.

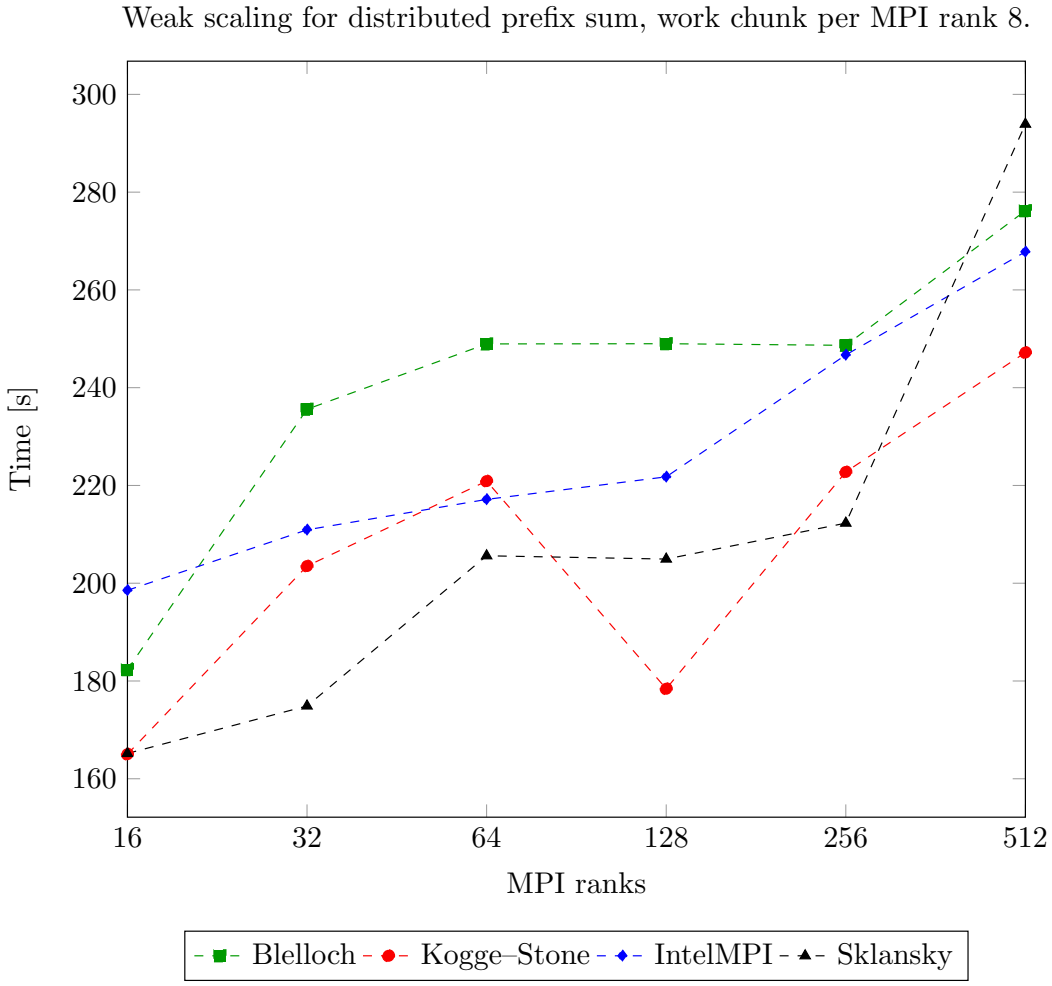


Figure 5.3: The smallest and largest experiments have been performed with 128 and 4096 images, respectively. The OpenMPI solution is excluded from this plot due to comparatively large values of execution time. The horizontal axis is logarithmic with a base of two.

Another results are presented on Figure 5.4. An experiment with the larger amount of work per rank $\frac{N}{P} = 32$ is expected to perform better because a constant increase should have less effect when the span is dominated by $\frac{2N}{P}$. All algorithms show a high increase in execution time from 4 to 8 ranks and a slow and rather stable increase for other values. Although Blelloch is the slowest algorithm on 128 images, its 14.3% increase in execution time is the lowest one, followed by 27.4% increase for Sklansky, 29.3% for the IntelMPI implementation and 34.4% for the Kogge–Stone algorithm. The average 28% increase can not be compared with a 165% increase for OpenMPI.

We observe that some image registration problem algorithms are not able to match theoretical bounds. For Kogge–Stone and Sklansky algorithms, the increase is much

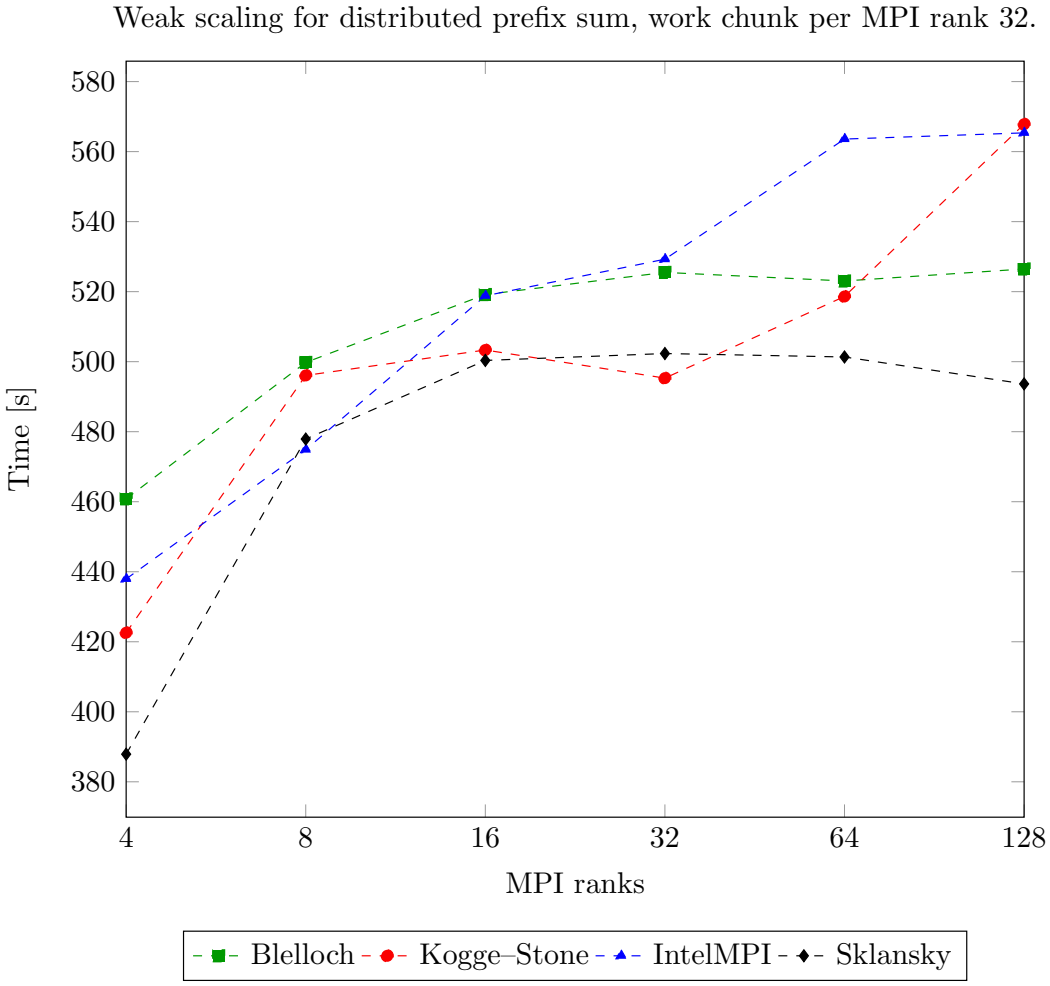


Figure 5.4: The smallest and largest experiments have been performed with 128 and 4096 images, respectively. The OpenMPI solution is excluded from this plot due to comparatively large values of execution time. The horizontal axis is logarithmic with a base of two.

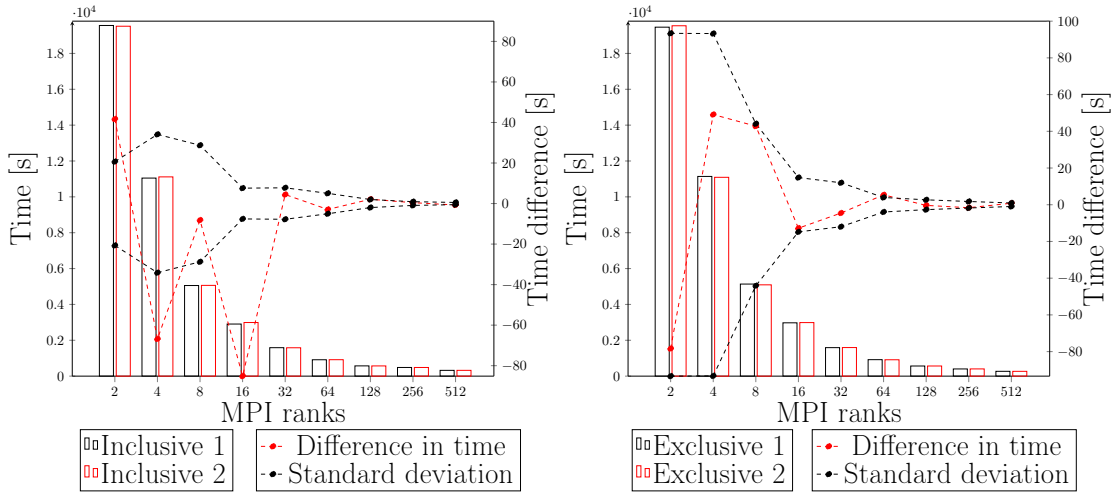
higher than theoretical limits of 27.7% and 7.8% increase, obtained by dividing the total increase with a span for the first measurement. This time, the Kogge–Stone algorithm exhibits a sudden increase in execution time at the last stage. On the other hand, Blelloch performs better than theoretical limits of 52.6% and 14.9%.

5.3 MPI implementation

In this section, we compare the performance of a distributed image registration with the global stage wrapped over a built–in MPI parallel scan. First, we present various im-

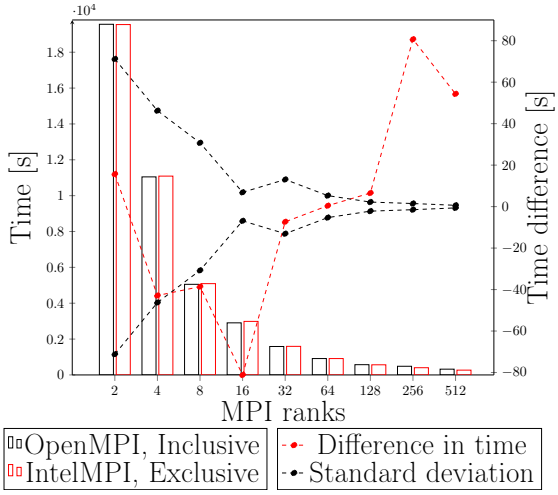
plementations in the IntelMPI library. Then, we measure the performance of OpenMPI scan and compare it against the best available IntelMPI scan.

5.3.1 IntelMPI



(a) Inclusive

(b) Exclusive



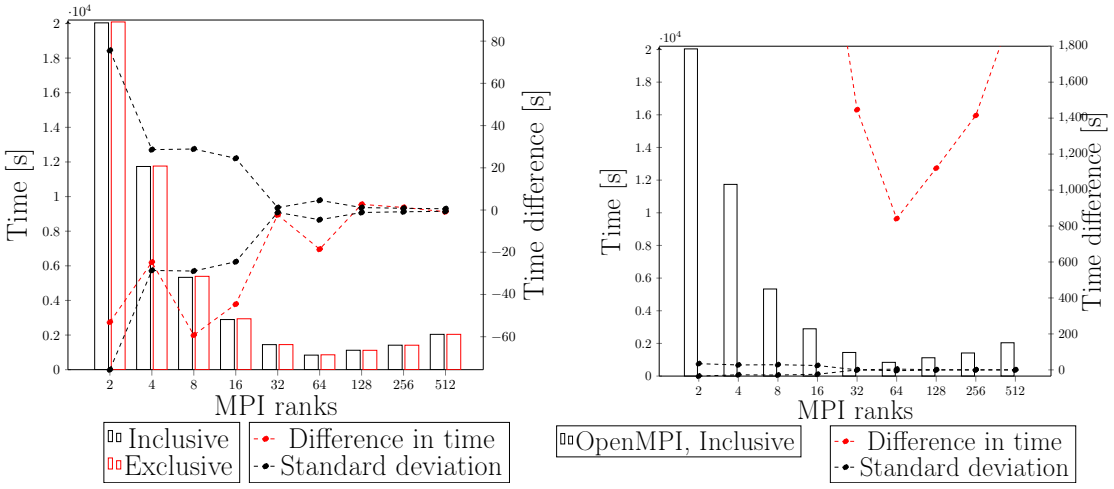
(c) Inclusive vs exclusive

Figure 5.5: An IntelMPI–based implementation of a distributed prefix sum for $N = 4096$. The left vertical axis corresponds to the bar plot presenting the execution time for algorithms. The right vertical axis corresponds to the plot of a difference in execution time between the first and second algorithm. The standard deviation is plotted with a reflection over the horizontal axis.

Inclusive and exclusive IntelMPI algorithms are compared on Figures 5.5a and 5.5b,

respectively. Interestingly, the inclusive *partial results gathering* performs better than its *topology–aware* counterpart for runs which do not span across nodes. For computations requiring intra–node communication, the difference is lower than a combined standard deviation. On the other hand, results with an exclusive MPI algorithm are much less clear, since inter–node results are not consistent, and they are too close to each other to justify a verdict. However, we prefer the second algorithm because of a slightly better stability of measurements.
The figure 5.5c compares a default implementation of an inclusive scan and the *topology–aware* implementation of the exclusive scan. The inclusive implementation provides a lower runtime when the number of MPI ranks is low. There, the difference in execution time is only an insignificant percentage of the total execution time. The exclusive scan surpasses the other algorithm on 128, 256 and 512 processor cores where this improvement has a noteworthy influence on the performance.

5.3.2 OpenMPI vs IntelMPI



(a) OpenMPI (b) Comparison

Figure 5.6: An MPI–based implementation of a distributed prefix sum for $N = 4096$. The left vertical axis corresponds to the bar plot presenting the execution time for algorithms. The right vertical axis corresponds to the plot of a difference in execution time between the first and second algorithm. The standard deviation is plotted with a reflection over the horizontal axis.

Figure 5.6a compares the performance of two scan implementations in OpenMPI. A negative difference suggests that the inclusive prefix sum tends to perform better for all runs utilizing less than 128 cores. Later, the difference is too small to differentiate between them. We have not found any explanation why the exclusive prefix sum performs worse in a serial implementation. We have decided to use the inclusive scan for comparisons with other prefix sums.

Finally, we compare best–performing implementations for IntelMPI and OpenMPI. Although the Figure 5.6b clearly shows that IntelMPI provides a superior performance for executions spanning multiple cluster nodes, results for $16, 32$ and 64 ranks are surprising. Again, a likely explanation is that multiple iterations of a parallel scan intensify delays created by ab unequal computation time between workers.

5.4 Alternative strategy

Section 4.1.1 introduced a formulation of the distributed prefix sum problem alternative to *scan–then–apply* general strategy. Using the notation from the previous chapter, we can express the default strategy in terms of how result x_i, located on worker I, is computed

$$x_{0,i} = \overbrace{\underbrace{x_{0,l_I}}_{\text{Global scan}} \odot \underbrace{\left((x_{l_I} \odot x_{l_I+1} \odot \cdots \odot x_{i-1}) \odot x_i\right)}_{\text{First local stage}}}^{\text{Second local stage}} \tag{5.5}$$

The final result is obtained by two applications of operator $\odot$ to x_i. On the other hand, the *reduce–then–scan* strategy achieves the same goal with a single application of the binary operator. Result from the first local stage is passed only to the global prefix sum

$$x_{0,i} = \overbrace{\left(\underbrace{x_{0,l_I}}_{\text{Global scan}} \odot x_{l_I} \odot x_{l_I+1} \odot \cdots \odot x_{i-1}\right) \odot x_i)}^{\text{Second local stage}} \tag{5.6}$$

Within the context of image registration, the approximately associative operator with an unpredictable runtime creates a possibility where a different ordering of execution may lead to improved initial guesses for consecutive calls to function **B**. Thus, we have to compare these two strategies to find out which one is better suited for this task.

Figure 5.7 presents the execution time of two strategies for the distributed image registration of 4096 frames, with a Kogge–Stone implementation of global prefix sum. For all measurements, up to the 256 cores, the default strategy performs better but the difference is monotonically decreasing. The alternative strategy is slightly faster than the default strategy for $P = 128$. We have seen in the section 5.2 that the default strategy with Kogge–Stone global scan exhibits a strange drop in performance for $N = 4096$ and $P = 128$. In the execution on 512 cores, the difference becomes smaller than the combined standard deviation.

To get a better picture, we analyze as well a measurement with a smaller data chunk per worker. Results presented on Figure 5.8 are not drastically different from the previous experiment.

We have not found any indication that investigating the alternative strategy may improve the algorithm's performance.

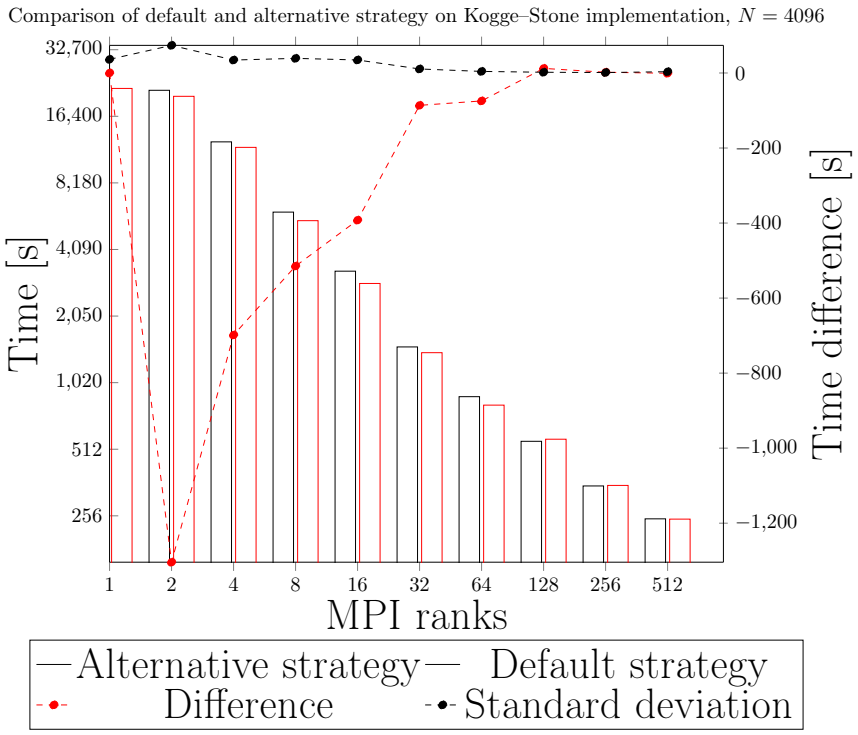


Figure 5.7: The left vertical axis corresponds to runtime for two strategies plotted as bars. The right vertical axis corresponds to a difference between the default and alternative strategy, presented with the standard deviation of difference. Both vertical and the left horizontal axes are logarithmic with base two.

5.5 Multithreading

To overcome the poor scalability of prefix sum on a large number of cores, we investigated the possibility of a hybrid parallelization. Instead of allocating P MPI ranks performing the distributed prefix sum, one could allocate $\frac{N}{2}$ or even $\frac{N}{4}$ ranks with two of four threads per rank. There, a better performance could be achieved by utilizing hardware to parallelize the image registration process.

A performance analysis of the image registration revealed two parallelizable functions which are responsible for approximately two-thirds of the execution time. According to Amdahl's law, it should allow for a speedup of 1.67 times on two threads and 3 times on four threads.

An experimental result for with GOMP, a GNU implementation of OpenMP for GCC compiler, is presented in Figure 5.9. We observe that a shared–memory parallelization of the operator is beneficial on computations on a large number of cores and with small chunk of work per MPI rank, even if the speedup of operator parallelization is not linear. An investigation of OpenMP runtime for Intel compiler did not reveal significantly different results.

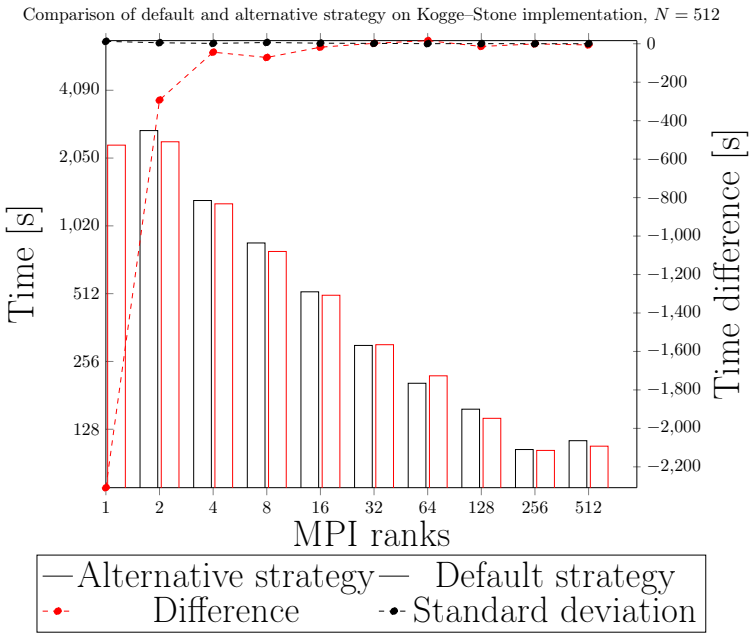


Figure 5.8: The left vertical axis shows runtime for two strategies plotted as bars. The right vertical axis displays a difference between the default and alternative strategy, presented with the standard deviation of difference. Both vertical and the horizontal ax are logarithmic with base two.

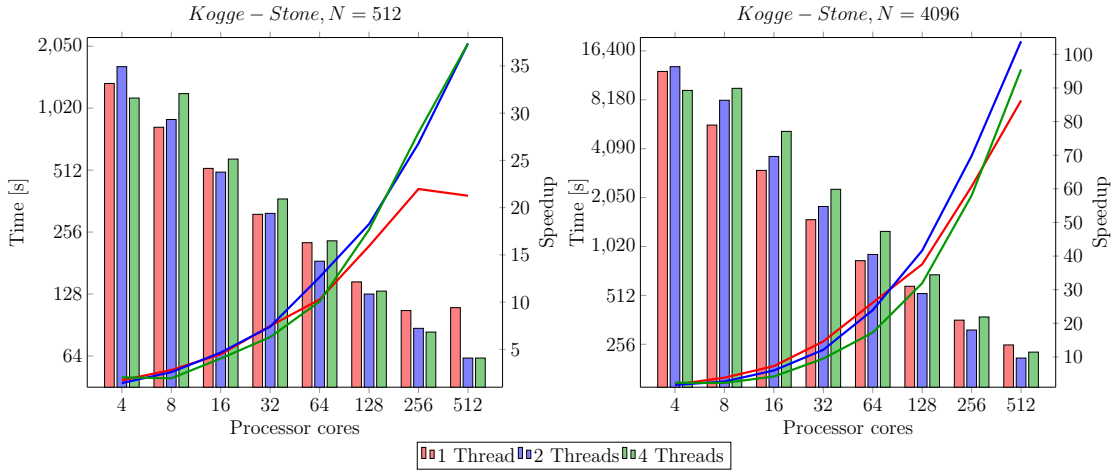


Figure 5.9: The left vertical axis corresponds to runtime of a hybrid parallelization, plotted as bars. The right vertical axis corresponds to speedup of a hybrid parallelization, plotted as solid lines. Both vertical and the left horizontal ax are logarithmic with base two. The last value for 4 threads is hardly visible because it is too small when compared with other data points.

Chapter 6

Summary

In this dissertation, we have discussed a parallelization strategy for registration of series of electron microscopy images. We have approached the problem by representing the registration procedure as a prefix sum. Several parallel prefix sum algorithms have been implemented, evaluated and compared against each other and their respective theoretical predictions. In this chapter, we present the conclusions and suggest future improvements.

Scalability

Strong scaling analysis of our problem reveals that an upper boundary on attainable speedup makes a massively parallel execution quite inefficient. Furthermore, a load imbalance induced by variances in execution time makes it even more wasteful in terms of computational resources.
Instead of trying to speedup fixed size problems, we focus on utilizing available resources to solve larger problems. A weak scaling analysis indicates there a more efficient use of available hardware.

MPI facilities

Our results show a stunning difference between MPI libraries in the quality of collective operations. A review of the literature suggests that this state of affairs may be caused by a relatively low popularity of the scan primitive in MPI community. Multiple papers have been written about optimal algorithms for collective operations such as barrier, broadcast, scatter, gather or even reduce. The IntelMPI library offers at least nine algorithms with variants for `MPI_Allreduce`, `MPI_Barrier`, `MPI_Bcast` and `MPI_Reduce`[44]. Sadly, the same cannot be said about the `MPI_Scan`.
The image registration problem demonstrates that there is a need for high quality implementations of distributed prefix sum algorithms.

Our contribution

We believe that the parallel image registration is the first example of the parallel prefix sum applied to a problem where the binary operator is

- not associative
- computationally intensive
- of iterative nature with huge variances in convergence time

We have investigated algorithms existing in the literature and derived alternative formulations of the problem. The research on parallel prefix adders has been concentrated on constructing work–efficient strategies with a minimal span. Parallel and distributed prefix sum algorithms have been designed for memory–bound operations and, as a result, they are optimized to minimize the cost of communication and memory access. Our work shows that there is a class of problems where better alternatives exist as neither work–efficiency nor optimized communication is desired.

Limitations and future work

The scalability of our parallelization scheme is inherently limited by the logarithmic nature of a parallel prefix sum and an unpredictable runtime for image registration operators. Our results suggest that a parallelization of the prefix sum operator is the only way of significantly improving the efficiency of a distributed prefix sum.

However, there are optimizations which can be applied to our problem. A major cause of a poor efficiency on many MPI ranks is a huge variation in execution times between workers. This impacts the global stage where each iteration involves an implicit synchronization through point–to–point communication. A shared–memory parallel implementation could decrease effects of a load imbalance by allocating one MPI rank per node and performing work–stealing inside a node. Negative effects of synchronization and communication in the global stage are decreased because they grow with the number of nodes, not the number of workers. Such improvement could boost the efficiency of computations spanning among a limited number of nodes, but it would not enable efficient, massively parallel computations.

Another way of improving performance is by redistributing the work after the global stage. In parallel prefix sum algorithms, many workers are not required to perform exactly $\log_2 P$ iterations, and they are allowed to start computing final deformations earlier. Furthermore, in the general strategy the very first worker is not performing any work at all after the first local phase.
Sadly, an analysis of results suggests that workers with a largest theoretical span are not always the slowest ones. Besides, the second local phase tends to require much less computational effort than other stages. Even a successful reduction of the last phase on the slowest worker would only slightly decrease the total execution time. And there is evidence to suggest that a redistribution policy based on a theoretical prediction of span could cause more harm than good.

References

[1] Benjamin Berkels, Peter Binev, Douglas A. Blom, Wolfgang Dahmen, Robert C. Sharpley, and Thomas Vogt. "Optimized imaging using non-rigid registration". In: *Ultramicroscopy* 138 (2014), pages 46–56. ISSN: 0304-3991. DOI: http://dx.doi.org/10.1016/j.ultramic.2013.11.007. URL: http://www.sciencedirect.com/science/article/pii/S0304399113002994.

[2] Jan Modersitzki. *Numerical methods for image registration*. 2004.

[3] Jan Modersitzki. *FAIR: flexible algorithms for image registration*. 2009.

[4] Jonathan Ngiam. "On the Processing and Analysis of HRTEM Data of Al-Oxidation". Master's thesis. University of Manchester, 2015.

[5] U. Clarenz, M. Droske, and M. Rumpf. "Towards Fast Non-Rigid Registration". In: *IN INVERSE PROBLEMS, IMAGE ANALYSIS AND MEDICAL IMAGING, AMS SPECIAL SESSION INTERACTION OF INVERSE PROBLEMS AND IMAGE ANALYSIS*. AMS, 2002, pages 67–84.

[6] Ulrich Trottenberg and Anton Schuller. *Multigrid*. Orlando, FL, USA: Academic Press, Inc., 2001. ISBN: 0-12-701070-X.

[7] Kenneth E. Iverson. *A Programming Language*. New York, NY, USA: John Wiley & Sons, Inc., 1962. ISBN: 0-471430-14-5.

[8] Richard E. Ladner and Michael J. Fischer. "Parallel Prefix Computation". In: *J. ACM* 27.4 (Oct. 1980), pages 831–838. ISSN: 0004-5411. DOI: 10.1145/322217.322232. URL: http://doi.acm.org/10.1145/322217.322232.

[9] "Scan, Reduce and". In: *Encyclopedia of Parallel Computing*. Edited by David Padua. Boston, MA: Springer US, 2011, pages 1787–1787. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_2099. URL: http://dx.doi.org/10.1007/978-0-387-09766-4_2099.

[10] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123914439, 9780124159938.

[11] Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. "Scan Primitives for Vector Computers". In: *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. Supercomputing '90. New York, New York, USA: IEEE Computer Society Press, 1990, pages 666–675. ISBN: 0-89791-412-0. URL: http://dl.acm.org/citation.cfm?id=110382.110597.

[12] Guy E. Blelloch. *Prefix Sums and Their Applications*. Technical report CMU-CS-90-190. School of Computer Science, Carnegie Mellon University, Nov. 1990.

[13] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. 2nd. McGraw-Hill Higher Education, 2001. ISBN: 0070131511.

[14] Guy E. Blelloch. *Vector Models for Data-parallel Computing*. Cambridge, MA, USA: MIT Press, 1990. ISBN: 0-262-02313-X.

[15] *Programming Language C++*. Standard. Geneva, CH: International Organization for Standardization, Mar. 2014.

[16] *Working Draft, Standard for Programming Language C++*. Standard. Mar. 2017.

[17] Jared Hoberock and Nathan Bell. *Thrust: A parallel template library*. 2010.

[18] James Reinders. *Intel Threading Building Blocks*. First. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007. ISBN: 9780596514808.

[19] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*. Specification. Sept. 2009. URL: `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`.

[20] Guy E. Blelloch and Bruce M. Maggs. "Algorithms and Theory of Computation Handbook". In: edited by Mikhail J. Atallah and Marina Blanton. Chapman & Hall/CRC, 2010. Chapter Parallel Algorithms, pages 25–25. ISBN: 978-1-58488-820-8. URL: `http://dl.acm.org/citation.cfm?id=1882723.1882748`.

[21] Thomas J. Sheffler. "Implementing the Multiprefix Operation on Parallel and Vector Computers". In: *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*. SPAA '93. Velen, Germany: ACM, 1993, pages 377–386. ISBN: 0-89791-599-2. DOI: `10.1145/165231.166115`. URL: `http://doi.acm.org/10.1145/165231.166115`.

[22] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. "Scan Primitives for GPU Computing". In: *Proceedings of the 22Nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. GH '07. San Diego, California: Eurographics Association, 2007, pages 97–106. ISBN: 978-1-59593-625-7. URL: `http://dl.acm.org/citation.cfm?id=1280094.1280110`.

[23] Mark Harris, Shubhabrata Sengupta, and John Owens. "Parallel prefix sum (scan) with CUDA". in: *GPU Gems 3*. Edited by Hubert Nguyen. Addison Wesley, 2007, pages 851–876.

[24] Duane Merrill and Andrew Grimshaw. *Parallel Scan for Stream Architectures*. Technical Report. Geneva, CH: Department of Computer Science, University of Virginia, Dec. 2009.

[25] Sang-Won Ha and Tack-Don Han. "A Scalable Work-Efficient and Depth-Optimal Parallel Scan for the GPGPU Environment". In: *IEEE Trans. Parallel Distrib. Syst.* 24.12 (Dec. 2013), pages 2324–2333. ISSN: 1045-9219. DOI: `10.1109/TPDS.2012.336`. URL: `http://dx.doi.org/10.1109/TPDS.2012.336`.

[26] Yuri Dotsenko, Naga K. Govindaraju, Peter-Pike Sloan, Charles Boyd, and John Manferdelli. "Fast Scan Algorithms on Graphics Processors". In: *Proceedings of the 22Nd Annual International Conference on Supercomputing*. ICS '08. Island of Kos, Greece: ACM, 2008, pages 205–213. ISBN: 978-1-60558-158-3. DOI: `10.1145/1375527.1375559`. URL: `http://doi.acm.org/10.1145/1375527.1375559`.

[27] Guy E. Blelloch. "Programming Parallel Algorithms". In: *Commun. ACM* 39.3 (Mar. 1996), pages 85–97. ISSN: 0001-0782. DOI: 10.1145/227234.227246. URL: http://doi.acm.org/10.1145/227234.227246.

[28] G. E. Blelloch. "Scans As Primitive Parallel Operations". In: *IEEE Trans. Comput.* 38.11 (Nov. 1989), pages 1526–1538. ISSN: 0018-9340. DOI: 10.1109/12.42122. URL: http://dx.doi.org/10.1109/12.42122.

[29] R. P. Brent and H. T. Kung. "The Chip Complexity of Binary Arithmetic". In: *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*. STOC '80. Los Angeles, California, USA: ACM, 1980, pages 190–200. ISBN: 0-89791-017-6. DOI: 10.1145/800141.804666. URL: http://doi.acm.org/10.1145/800141.804666.

[30] Peter M. Kogge and Harold S. Stone. "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations". In: *IEEE Trans. Comput.* 22.8 (Aug. 1973), pages 786–793. ISSN: 0018-9340. DOI: 10.1109/TC.1973.5009159. URL: http://dx.doi.org/10.1109/TC.1973.5009159.

[31] W. Daniel Hillis and Guy L. Steele Jr. "Data Parallel Algorithms". In: *Commun. ACM* 29.12 (Dec. 1986), pages 1170–1183. ISSN: 0001-0782. DOI: 10.1145/7902.7903. URL: http://doi.acm.org/10.1145/7902.7903.

[32] Ömer Eğecioğlu, Cetin K. Koc, and Alan J. Laub. "A recursive doubling algorithm for solution of tridiagonal systems on hypercube multiprocessors". In: *Journal of Computational and Applied Mathematics* 27.1 (1989). Special Issue on Parallel Algorithms for Numerical Linear Algebra, pages 95–108. ISSN: 0377-0427. DOI: http://dx.doi.org/10.1016/0377-0427(89)90362-2. URL: http://www.sciencedirect.com/science/article/pii/0377042789903622.

[33] J. Sklansky. "Conditional-Sum Addition Logic". In: *IRE Transactions on Electronic Computers* EC-9.2 (June 1960), pages 226–231. ISSN: 0367-9950. DOI: 10.1109/TEC.1960.5219822.

[34] Marc Snir. "Depth-size Trade-offs for Parallel Prefix Computation". In: *J. Algorithms* 7.2 (June 1986), pages 185–201. ISSN: 0196-6774. DOI: 10.1016/0196-6774(86)90003-9. URL: http://dx.doi.org/10.1016/0196-6774(86)90003-9.

[35] Haikun Zhu, Chung-Kuan Cheng, and Ronald Graham. "On the Construction of Zero-deficiency Parallel Prefix Circuits with Minimum Depth". In: *ACM Trans. Des. Autom. Electron. Syst.* 11.2 (Apr. 2006), pages 387–409. ISSN: 1084-4309. DOI: 10.1145/1142155.1142162. URL: http://doi.acm.org/10.1145/1142155.1142162.

[36] T. Han and D. A. Carlson. "Fast area-efficient VLSI adders". In: *1987 IEEE 8th Symposium on Computer Arithmetic (ARITH)*. May 1987, pages 49–56. DOI: 10.1109/ARITH.1987.6158699.

[37] D. Harris. "A taxonomy of parallel prefix networks". In: *The Thrity-Seventh Asilomar Conference on Signals, Systems Computers, 2003.* Volume 2. Nov. 2003, 2213–2217 Vol.2. DOI: 10.1109/ACSSC.2003.1292373.

[38] Ralf Hinze. "An Algebra of Scans". In: *In Mathematics of Program Construction*. Springer, 2004, pages 186–210.

[39] A. P. Diéguez, M. Amor, and R. Doallo. "Efficient Scan Operator Methods on a GPU". in: *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. Oct. 2014, pages 190–197. DOI: 10.1109/SBAC-PAD.2014.23.

[40] Shubhabrata Sengupta, Mark Harris, and Michael Garland. *Efficient Parallel Scan Algorithms for GPUs*. Technical Report. NVIDIA, 2008.

[41] Peter Sanders and Jesper Larsson Träff. "Parallel Prefix (Scan) Algorithms for MPI". in: *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 13th European PVM/MPI User's Group Meeting Bonn, Germany, September 17-20, 2006 Proceedings*. Edited by Bernd Mohr, Jesper Larsson Träff, Joachim Worringen, and Jack Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pages 49–57.

[42] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. "Full Bandwidth Broadcast, Reduction and Scan with Only Two Trees". In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 14th European PVM/MPI User's Group Meeting, Paris, France, September 30 - October 3, 2007. Proceedings*. Edited by Franck Cappello, Thomas Herault, and Jack Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pages 17–26. ISBN: 978-3-540-75416-9. DOI: 10.1007/978-3-540-75416-9_10. URL: http://dx.doi.org/10.1007/978-3-540-75416-9_10.

[43] Peter Sanders, Jochen Speck, and Jesper Larsson Träff. "Two-tree algorithms for full bandwidth broadcast, reduction and scan". In: *Parallel Computing* 35.12 (2009). Selected papers from the 14th European PVM/MPI Users Group Meeting, pages 581–594. ISSN: 0167-8191. DOI: http://dx.doi.org/10.1016/j.parco.2009.09.001. URL: http://www.sciencedirect.com/science/article/pii/S0167819109000957.

[44] *Intel MPI Library for Linux OS*. Developer Reference. May 2017.

Webpages

[45] *QuocMesh Library*. 2017. URL: http://numod.ins.uni-bonn.de/software/quocmesh/ (visited on June 14, 2017).

[46] Kyle Lutz. *Boost.Compute C++ Library*. 2017. URL: https://github.com/boostorg/compute (visited on May 29, 2017).

[47] *OpenMPI*. 2017. URL: https://github.com/open-mpi/ompi (visited on June 14, 2017).