algorithms

MDPI

Article

Improved Scheduling Algorithm for Synchronous Data Flow Graphs on a Homogeneous Multi-Core Systems

Lei Wang [1,2,*], Chenguang Wang [1] and Huabing Wang [2]

[1] Research Institute of Electronic Science and Technology, University of Electronic Science and Technology of China, Chengdu 611731, China; wangchenguang@std.uestc.edu.cn
[2] State Key Laboratory of Complex Electromagnetic Environment Effects on Electronics and Information System, Luoyang 471000, China; 13849916491@139.com
* Correspondence: wang_lei@uestc.edu.cn

Abstract: In order to accelerate the execution of streaming applications on multi-core systems, this article studies the scheduling problem of synchronous data flow graphs (SDFG) on homogeneous multi-core systems. To describe the data flow computation process, we propose the SDAG (Super DAG) computation model based on the DAG model combined with data-driven thoughts. Further, we analyze the current common SDFG scheduling algorithms and propose an improved SDFG scheduling algorithm, LSEFT (level-shortest-first-earliest-finish time). The LSEFT algorithm uses an inverse traversal algorithm to calculate the priority of tasks in the task-selection phase; the shortest-job-priority earliest-finish-time policy is used in the processor selection phase to replace the original long job priority policy. In the experimental part, we designed an SDFG random generator and generated 958 SDFGs with the help of the random generator as test cases to verify the scheduling algorithm. The experimental results show that our improved algorithm performs well for different numbers of processor cores, especially for 8 cores, where the speedup of our improved algorithm improves by 10.17% on average.

Keywords: homogeneous synchronous data flow graph; computational model; multi-core systems; scheduling algorithm

check for updates

Citation: Wang, L.; Wang, C.; Wang, H. Improved Scheduling Algorithm for Synchronous Data Flow Graphs on a Homogeneous Multi-Core Systems. *Algorithms* **2022**, *15*, 56. https://doi.org/10.3390/a15020056

Academic Editor: Marc Sevaux

Received: 10 January 2022
Accepted: 31 January 2022
Published: 9 February 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

In recent years, parallel computing has been widely popular in areas such as high-performance computing and deep learning. Streaming programs are one such class of computational tasks, which are widely popular in areas such as digital signal processing and multimedia [1]. SDFGs are an abstract representation of streaming applications. Where vertices represent tasks, directed edges represent dependencies between tasks, and vertices are also called *actors*. There is usually a directed loop structure in the SDFG, which needs to be converted into an HSDFG (Homogeneous SDFG). HSDFG is a special case of SDFG.

The common scheduling algorithm for SDFG is static scheduling [2]. In static scheduling, all information in the computational graph is known at the compilation stage. Static scheduling algorithms are divided into two main categories based on the scheduling object, namely, SDFG oriented and HSDFG oriented. SDFG-oriented scheduling algorithms study how to schedule actors in the original computational graph. HSDFG-oriented scheduling algorithms study the *instances* of actors. Instances are the execution units of an actor and have the same methods and properties as the actor, and different instances of the same actor process different data. Both SDFG-oriented and HSDFG-oriented algorithms are algorithms for scheduling SDFG. We use the algorithm in [3] to convert SDFG to HSDFG. HSDFG can be represented using the DAG model [4]. The study's object of our algorithm is HSDFG oriented.

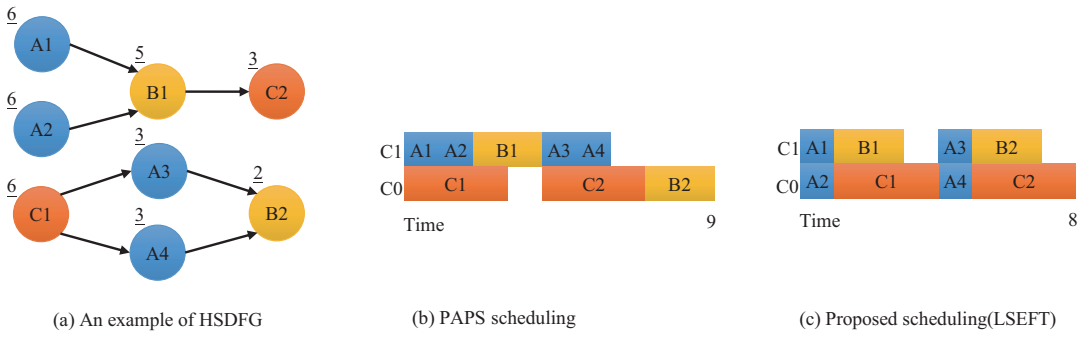
Among the shared storage computational models, the current computational model commonly used by parallel machines is APRAM [5]. The data flow model has a naturally

parallel representation compared to the APRAM computational model, which uses synchronization barriers to implement delayed waits, and must wait until all threads in a superstep have finished executing before synchronizing. A large part of the overhead in the APRAM model is in the delayed wait part, where the completion of the execution of a superstep depends on the thread with the longest execution time in that superstep. In the data-flow computing model, most methods of synchronization between tasks use peer-to-peer communication rather than using coarse-grained methods of synchronization barriers.

The Codelet [6] model is a data flow execution model applied to the von Neumann model. In this model, each non-preemptive task is the basic unit of scheduling and execution and is called a "codelet". All codelets are connected together to form a codelet graph. In the codelet graph, each codelet plays the role of a producer and/or a consumer. Corresponding to the SDFG model, each vertex is also called an actor, which has an input and output port attribute, and a port has a rate attribute. The port rate indicates the rate at which the actor produces/consumes data per execution. In the Codelet runtime model, each codelet has three states. Codelets are called dormant when they are waiting for their dependencies to be satisfied. When all dependencies are satisfied it is called the executable state, at which point the codelet is ready to be dispatched. When the codelet is dispatched to the processor core it is called the ignition state.

Although the DAG model can represent the HSDFG, the DAG model lacks a data-driven mechanism. To address this challenge, this paper adds a peer-to-peer communication and data-driven mechanism based on the DAG model, called SDAG (Super DAG), in which the communication between two instances is implemented through a specific FIFO (First-In, First-Out) queue.

Further, we analyze the current HSDFG scheduling algorithm, PAPS [3] (Figure 1b). We propose an improved HSDFG scheduling algorithm, LSEFT (Figure 1c). The PAPS is a Hu-level [7] based scheduling algorithm, where each instance has a level value (Figure 1a) top left corner of the vertex). Level value indicates the sum of the execution times of the instances contained in the longest path from the current instance to the endpoint instance. However, when an HSDFG has many endpoint instances, an additional instance with execution time 0 needs to be created to connect these endpoint instances. We have improved the method of calculating the level by using a reverse traversal algorithm that eliminates additional instances and calculate the longest path. Furthermore, we experimentally found that the scheduling policy in the PAPS algorithm does not perform well in some cases, without incorporating a core selection policy. Therefore, we use the short-job-first policy in the task-selection phase and the earliest-finish-time policy in the core selection phase.



(a) An example of HSDFG (b) PAPS scheduling (c) Proposed scheduling(LSEFT)

Figure 1. Scheduling for HSDFG.

When evaluating the SDFG scheduling algorithm, test cases need to be implemented using specific data flow languages such as SteamIt [8] or COStream [9]. Using these data flow graph languages specific data flow procedures such as FFT can be implemented. If only these specific test cases are used, a comprehensive evaluation of the scheduling algorithm is not possible. This paper, therefore, designs the SDFG random generator, which

can randomly generate SDFGs of a size that meets the actual demand size, as an input source for evaluating the scheduling algorithm.

In summary, the main contributions of this paper are divided into three components and are the novelty that this paper aims to highlight.

- Using data-driven thinking, the DAG computational model is extended to propose the SDAG model, which is applicable to the scheduling of dataflow programs under the current common processor architecture platforms;
- Improved the currently used SDFG scheduling algorithm to achieve better speedup;
- A new SDFG random generator was designed and used to create an SDFG dataset for evaluating the scheduling algorithm. The data are stored in matrix format, eliminating the need to create additional xml parsers and making it easier for researchers to use.

The remaining parts of this paper are organized as follows. Sections 2 and 3 introduce the SDFG background and related work, respectively. Section 4 introduces the SDAG model and the LSEFT scheduling algorithm. Section 5 presents the SDFG random generator and experimental results. Finally, Section 6 concludes this paper.

2. SDFG Background

As shown in Figure 2, an SDFG can be represented by a directed graph. Where the vertices represent computational tasks, also called *actors*. The actors have input and output ports, and the ports have data block processing rates, which are called *tokens*. The number of tokens consumed by the input port once the actor is executed is called *consumption rate*, similarly, the output port corresponds to *production rate*. In the synchronous data flow computational mode [2], the port rate of the actor is determined. The memory where the tokens are stored is called the *data buffer* (or called *channel*) and is implemented as a FIFO queue, which also specifies the data dependencies between the two actors, with the head of the queue connecting the actor's input port and the tail of the queue connecting the actor's output port.

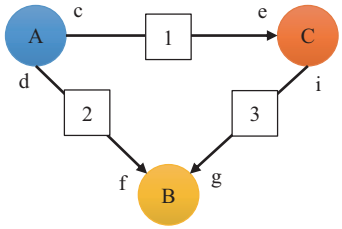


Figure 2. An example of SDFG.

Assume that the precursor actor is A, the successor actor is B, B depends only on A, and both A and B have only one port. *Minimum execution period* indicates that all actors in the SDFG are executed at least once. In a minimum execution period, the number of executions of A is q_A, the number of executions of B is q_B, the production rate of port A is $s(A)$, the consumption rate of port B is $r(B)$, the balancing equation of Equation (1) can be constructed.

$$q_A s(A) = q_B r(B) \tag{1}$$

where $s(A)$ and $r(B)$ are known and $s(A), r(B) \in \mathbb{N}^+$, then an infinite number of solutions (q_A, q_B) can be solved. With the minimum execution period, we take the minimum positive integer solution as the number of executions for actor A and actor B. In SDFG, each of the two actors with data dependencies constructs a balancing equation, and we represent the solution obtained for each actor in SDFG by a solution vector $q = (q_1, q_2, q_3, \ldots)$. If there is no solution, or if a positive integer solution cannot exist, the SDFG is said not to satisfy the consistency condition and cannot be scheduled for execution. Otherwise, we take the minimum positive integer solution as the number of times each actor has to be executed at its minimum period, which is referred to as the unit period in the following.

We represent an SDFG using the tuple (V, E), where $V = \alpha_1, \alpha_2, \ldots, \alpha_n$ denotes the set containing n actors, α_i, $i = 1 \ldots n$ denotes actor i, $E = e_1, e_2, \ldots, e_k \subseteq V \times V$ denotes the set containing k directed edges, and $e_i, i = 1, \ldots, k$ denotes the i-th directed edge. Use $src(e_i)$ to denote the production rate of directed edge e_i arc-tailed actors and $des(e_i)$ to denote the consumption rate of directed edge e_i arc-headed actors. The SDFG is represented using a topological matrix similar to the correlation matrix [3], where the rows represent directed edges and the columns represent actors, generating a $k \times n$ matrix with k directed edges and n actors. The topological matrix element (i, j) is given the value rule as in Equation (2).

$$(i,j) = \begin{cases} \text{src}(e_i), \text{ if the actor of arc tail of } e_i \text{ is } \alpha_j \\ -\text{des}(e_i), \text{ if the actor of arc head of } e_i \text{ is } \alpha_j \\ 0, \text{ other} \end{cases} \tag{2}$$

Thus, the topological matrix Γ for the example in Figure 1 can be obtained as (3):

$$\Gamma = \begin{bmatrix} c & -e & 0 \\ d & 0 & -f \\ 0 & i & -g \end{bmatrix} \tag{3}$$

With the help of the topological matrix Γ, we use Equation (4) to solve for the vector q.

$$\Gamma q = 0 \tag{4}$$

The sufficient condition for the existence of a minimum positive integer solution to Equation (4) is $rank(\Gamma) = n - 1$, where n is the number of actors in the SDFG. The corresponding number of instances of actors is created based on the q vector, and these instances are then constructed as an HSDFG.

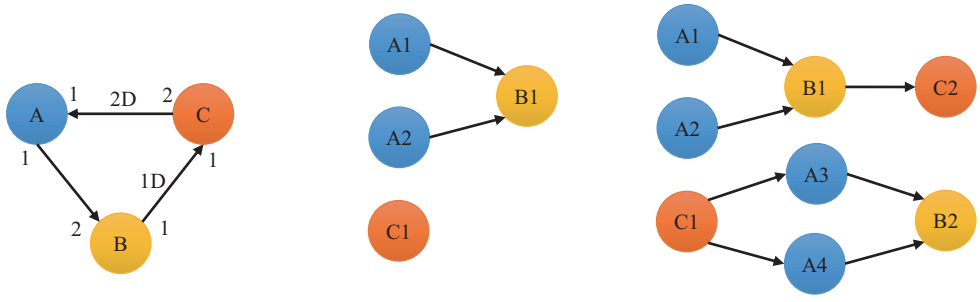
At the beginning of the execution of an instance, the tokens in the cache queue are consumed, and at the end of the execution of the instance, the resulting tokens are stored in the cache queue. Therefore the state of the cache queue needs to be updated throughout the lifetime of the instance, with the update formula as in Equation (5).

$$b(n+1) = b(n) + \Gamma v(n) \tag{5}$$

where vector b denotes the cache queue and $b(n+1)$ denotes the state of the cache queue at the $n+1$ moment. $v(n)$ denotes the actor at moment n. If the actor is α_i, then the i-th position of $v(n)$ is 1 and the remaining positions are 0.

Construction of the HSDFG Algorithm

In an SDFG, there is a directed cyclic structure, such as Figure 3a, where there is a directed cyclic ring in the directed graph. To avoid deadlocks, a cache of data, denoted "1D", is placed on the directed edge between actor B and actor C. Similarly, 2 caches of data, denoted "2D", are placed between actor C and actor A. In order to break this ring structure and to more naturally represent the parallelism between instances, we transform the SDFG into an HSDFG, i.e., a DAG, which in turn transforms the object of our study into a vertex in the DAG, without taking into account some of the properties of the SDFG.



(a) An example of SDFG (b) HSDFG J=1 (c) HSDFG J=2

Figure 3. HSDFG is constructed by SDFG.

Dataflow programs are periodic programs, so in converting to HSDFG we can consider the HSDFG constructed by all instances under a unit period, and we can also consider instances under multiple periods. We denote this variable by J. If $J = 1$, it denotes the HSDFG constructed by all instances under the unit period, as in Figure 3b. If $J = 2$, it denotes the HSDFG constructed for all instances under two periods, as in Figure 3c.

To describe the construction of the HSDFG algorithm, we first define some notation. j values, a repeat execution vector p, a random sequence of actors $node\_r$, a set of predecessor actors for each actor, and a topology matrix Γ. b(0) then denotes the state of the cache queue at the initial moment. j_p denotes the scheduling count vector for all actors, where j_p_α denotes the scheduling count for the actor α. $count_t$ denotes the vector of instantiation counts for all actors, where $count\_t_\alpha$ denotes the number of instantiations of the actor α. A detailed description is shown in Algorithm 1.

Algorithm 1: Constructing the HSDFG

Input: SDFG

Output: HSDFG

1: $j\_p = J * p$, $count_t = O$, O is zero vector.
2: Iterate over the actor α in the sequence $node\_r$
3: IF $count\_t_\alpha == j\_p_\alpha$, then perform Step 2, else perform Step 4.
4: Calculate $b + \Gamma v_\alpha$ to judge whether the actor α can be executed. If not, perform Step 2. If so, $count\_t_\alpha \leftarrow count\_t_\alpha + 1$, and create the $count\_t_\alpha$-th instance $ins\_{}_\alpha$. Then add to the instance sequence apg_ins.
5: Check whether the precursor set of the actor α has elements. If not, perform Step 9. Else perform Step 6.
6: Iterate through the precursor actor set of the actor α to obtain the precursor actor η and the connected directed edge a.
7: Calculate d value, using $d = \left\lceil \frac{-j\Gamma_{a\alpha} - b(0)_a}{\Gamma_{a\eta}} \right\rceil$. If $d < 0$, then $d = 0$. Establish a precursor relationship between the first d instances of actor η and instance $ins\_{}_\alpha$.
8: Update $b \leftarrow b + \Gamma v_\alpha$. If the current traversal is not complete, perform Step 2; else, preform Step 9.
9: For all actors, if $count\_t == j\_p$, then algorithm is terminated. Else perform Step 2 and start a new round of traversal.

3. Related Work

In this section, we introduce the work associated with the three contributions to this paper.

The study of the synchronous data flow computational model [2] is based on the data flow architecture [10], not the von Neumann architecture. In a data flow architecture, instructions are executed in a data-driven order, not according to a program counter. However, there is no commercially available data flow architecture computer, so the parallel

computational model based on the data flow computational model is based on a non-von Neumann architecture. The Wave Front Array (WFA) [11] computational model introduces the principle of data flow computing in an array of processors. This computational model is only applicable to a specific VLSI array. Furthermore, the WFA computational model does not use SDFG as an abstract representation of the application and loses this natural parallel representation structure.

The DAG computational model [12] is an architecture-independent computational model, particularly suitable for numerical computation, which expresses the internal parallelism of a program application by assigning processors to individual vertices according to a certain scheduling policy. Each vertex performs the operations specified by that vertex. Although HSDFG is based on the DAG model, the way in which tasks communicate with each other is missing from the DAG.

Static task scheduling methods can be divided into two main groups, heuristic-based strategies and guided random search-based ones. The heuristic-based strategies can be further divided into three groups: list scheduling strategies, clustering strategies and task replication strategies.

The list scheduling strategy first calculates the priority of a task based on a given task graph and then maintains a scheduling list based on the priority. The list scheduling algorithm is divided into two main phases: (1) the task-selection phase, which selects the ready task with the highest priority, and (2) the processor selection phase, which selects a suitable processor such that the objective function is minimized. MOELS algorithm was proposed in [13] and four list scheduling algorithms were extended in [14]. Both [13,14] study workflow scheduling methods, and although both SDFG and workflow can be represented using the DAG model, SDFG needs to consider its periodicity. Ref. [15] improves the list scheduling algorithm considering load balancing and proposes the LS-IPLB algorithm. In our experiments, we also compared load balancing strategies and showed that the results did not perform as well as the EFT strategy.

A clustering scheduling strategy maps the grouped tasks into an infinite number of computational units. In the steps of the clustering algorithm, each step of the clustering is an arbitrary task that may not be the ready task, so it is necessary to go back in each group to select the ready task for scheduling, where there is [16] formalized the scheduling problem as an integer planning clustering problem with grouped clustering scheduling [17]. Scheduling algorithms based on replication strategies are also task computational resources are unbounded and replication-based strategies use redundant tasks to reduce communication overhead [18,19]. Furthermore, the cluster scheduling strategy and the replication strategy have higher complexity compared to the list scheduling algorithm.

Guided random search techniques use random choices to guide themselves through a problem space, which is not a random search method [20]. These techniques combine the knowledge gained from previous search results with some random features to generate new search results. Refs. [21,22] use guided random search techniques to accomplish task scheduling, but they typically have much higher scheduling times than heuristic-based methods.

Scheduling methods for SDFG on multi-core systems are classified into static scheduling [23,24] and self-time scheduling [25–29]. The self-time scheduling method studies SDFG, which completes scheduling based on actor readiness and completion times, and is a scheduling strategy with many real-time requirements. Ref. [25] proposes an offline heuristic strategy for speeding up algorithm scheduling rather than optimizing scheduling quality. Ref. [26] is oriented towards a specific DSP multiprocessor architecture. Scheduling with other real-time tasks interfering is considered in [27], where the scheduling algorithm proposed in [27] relies on special cases. Ref. [28] considers multi-mode multi-processor scheduling techniques that can schedule all SDFGs in all modes simultaneously. Ref. [29] is targeted at specific embedded stream processing applications using a symbolic approach for SDFG analysis. DAG-based scheduling is a type of static scheduling method. PAPS [3] and DONF [4] are methods for scheduling SDFG on a multi-core systems based on the

DAG model. The PAPS method uses a Hu-level [7] based approach in the task-selection phase to calculate the level values of individual vertices. Priority is given to the vertex with the larger level value for scheduling. However, no core selection policy is specified with PAPS. The DONF method only calculates the priority of tasks that are locally in the ready state during the task-selection phase. The DONF method ignores the global information of the computed graph and also specifies the handling strategy when tasks of equal priority conflict with the number of processors. The DONF method studies single DAG, whereas HSDFG is transformed from SDFG and multiple DAG will exist.

SDF3 [30] studies the SDFG random generator, which generates SDFGs that satisfy consistency requirements for the evaluation of scheduling policies. However, the SDFG model save file of this generator is in xml format, and a corresponding xml parser is required to be used as the input source of the scheduling algorithm. We have designed the SDFG random generator without the use of an xml parser.

4. SDAG Model and HSDFG Scheduling

4.1. SDAG Computational Model

We have added a communication method and a data-driven mechanism to the DAG model, i.e., SDAG, which not only has the architecture-independent advantages of DAG, but also the data-driven advantages of the data flow computational model. In this subsection, we focus on the communication approach and the data-driven mechanism.

There are two actors A and B, and an SDFG consisting of a directed edge, as shown in Figure 3a. The production rate of port A is $s(A) = 1$, and the consumption rate of port B is $r(B) = 2$. According to the balancing Equation (1), at the minimum period, A needs to execute 2 times to satisfy the data demand of B. A needs to create 2 instances (A_1 and A_2) and B needs to create only 1 instance (B_1) to construct the HSDFG as shown in Figure 4b.

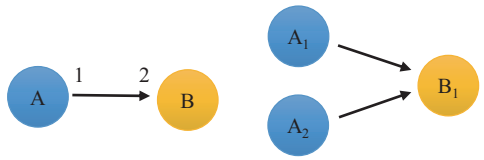


Figure 4. (**a**) SDFG, (**b**) HSDFG converted form (**a**).

In SDAG, each directed edge in the SDFG corresponds to a FIFO queue of finite capacity. We assume that there are two instances of α_1 and α_2, and a directed edge connecting these two instances. The predecessor instance is α_1, and the successor instance is α_2, and the directed edge is a FIFO queue with a shared memory area. Both α_1 and α_2 have only one port, the output port rate of α_1 is $s(\alpha_1)$, the input port rate of α_2 is $r(\alpha_2)$. Data communication between α_1 and α_2 is achieved through a FIFO queue with the following constraints on the operation of the FIFO queue. α_1 performs an *enqueue* operation on the FIFO queue after execution is complete, and α_2 performs a *dequeue* operation on the FIFO queue during execution. The data-driven mechanism is based on the FIFO queue. We set a threshold constant *thd* and a queue capacity variable *caps* for the FIFO queue, with the *caps* variable being jointly maintained by α_1 and α_2. The threshold constant *thd* and the queue capacity variable *caps* are defined as follows:

$$thd = r(\alpha_2) \tag{6}$$

$$caps = caps + s(\alpha_1) \tag{7}$$

$$caps = caps - r(\alpha_2) \tag{8}$$

After α_1 has finished updating the *caps* variable, when Equation (9) is satisfied, α_1 will access the waiting queue (Figure 5) to see if α_2 has been created. If not, α_2 is created, the

status of the port corresponding to α_2 is set to available and α_2 is placed in the waiting queue. After each instance is created, a status variable is also created for each input port of the instance and initialised to unavailable. If so, the state of the port corresponding to α_2 is updated. If α_2 has more than one input port, α_2 is created whenever one of them satisfies Equation (9).

$$caps \geqslant thd \tag{9}$$

The memory overhead can be excessive when creating a FIFO queue for each directed edge in the HSDFG. Therefore, we specify that instances created by the same actor share a FIFO queue, e.g., A_1 and A_2 in Figure 2b share a FIFO queue. To ensure consistency in reading data from the FIFO queue, a "pass lock" is set for both the enqueue and dequeue operations. The "pass lock" specifies the instance that can open the lock, and the instance cannot perform an enqueue or dequeue operation until the "pass lock" is opened. After the current instance has completed its queuing operation, the lock is set to be opened by the next instance in order of instance number, i.e., passed to the next instance. The last instance passes the lock to the first instance. The "pass lock" defaults to the first instance being able to open it at the initial stage.

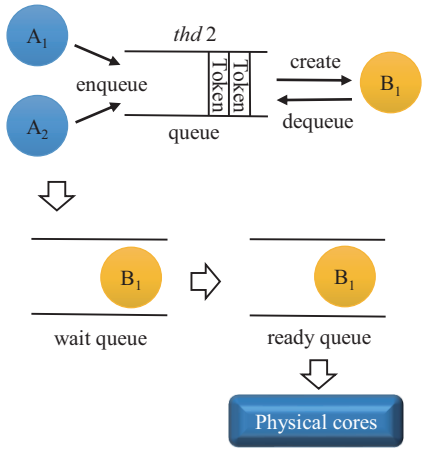


Figure 5. SDAG model overview.

An overview of the SDAG model is shown in Figure 5, where instance B_1 is created using a data-driven mechanism and then B_1 is added to the waiting queue. If B_1 is dependent on many FIFO queues, then the input port state of B_1 is updated whenever a FIFO queue satisfies Equation (9). When all input port states of B_1 are available, B_1 is added to the ready queue. All instances in the ready queue can be dispatched directly to the physical core for execution.

4.2. SDAG Computational Model Analysis

In the SDAG model, a computation consists of a series of thread-level tasks (instances). The dependencies between these tasks are represented using a DAG. The creation of tasks is data driven, each processor executes the task for which the data dependency has been satisfied, and communication between processors is done via a FIFO queue in a shared memory area. Figure 5 of the SDAG model shows that the cost function of an instance thus constructed is as follows.

$$T_{ins} = h_{ins} + w_{ins} + L \tag{10}$$

where h_{ins} is the time interval between the first FIFO queue at the input port of instance ins satisfying Equation (9) and the last FIFO queue satisfying Equation (9). w_{ins} is the computation time of instance ins and L represents the shared memory FIFO queue operation time (including enqueue, dequeue and "pass lock" time). So in the SDAG model, if there are s instances, the total execution time T_{SDAG} is:

$$T_{SDAG} = \sum_{i=0}^{s-1} h_i + \sum_{i=0}^{s-1} w_i + sL \tag{11}$$

SDAG belongs to the shared memory model of computing and adopts the data-driven idea of creating tasks in data flow, placing the communication of tasks in a limited shared memory space. To save shared memory space, we specify that different instances of the same actor share a FIFO queue. This is reflected in tasks that compute the same data differently, which is very common in data flow tasks. To ensure data consistency, we have added a "pass lock" mechanism so that the current FIFO queue can only be operated by an instance that can open the "pass lock". Instances are created sequentially, and if the execution environment is homogeneous with multi-core processors, then the processor computation times for instances of the same actor are the same, so that most instances finish sequentially. The biggest advantage of the SDAG model is that it does not depend on a specific hardware platform and can be implemented on all common hardware platforms.

4.3. HSDFG Scheduling

4.3.1. Problem Definition

HSDFG scheduling is a static scheduling, which allows all tasks to be scheduled during the compilation phase, and therefore HSDFG scheduling simulates the process of execution. First, we introduce the basic notations related to hardware and software, as shown in Table 1.

Table 1. Basic Notations.

Notation	Description
α	An instance in set V
$W(\alpha)$	The size of the instance α
$IN(\alpha)$	The set of input channels of the instance α
$OUT(\alpha)$	The set of output channels of the instance α
I^{α}	The ID of input channel of the instance α
O^{α}	The ID of output channel of the instance α
$r(I^{\alpha})$	The token producing rate of instance α of channel I^{α}
$s(O^{\alpha})$	The token consuming rate of instance α of channel O^{α}
$et(\alpha)$	The execution time of the instance α
e	A channel in set E
C	A set of all processing cores
c	A processing core in C
$p(c)$	The performance of processing core c
M	The number of processing cores

A data flow application consists of a series of periodic instances. These instances are represented using the HSDFG and we only consider instance scheduling per unit period. The HSDFG is denoted by a tuple (V, E), $V = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$ denoting the set containing n instances, α_i is the instance ID, $i = 1, \ldots, n$, and $W(\alpha_i)$ is the size of α_i. α_i has the set of input channels $IN(\alpha_i)$ and the set of output channels $OUT(\alpha_i)$, where $IN(\alpha_i) = \{I_1^{\alpha_i}, I_2^{\alpha_i}, \ldots\}$, $I_j^{\alpha_i}$ is the ID of the input channel of α_i, $j = 1, 2, \ldots, r(I_j^{\alpha_i})$ denotes the tokens consumption rate of $I_j^{\alpha_i}$, and $OUT(\alpha_i) = \{O_1^{\alpha_i}, O_2^{\alpha_i}, \ldots\}$, $O_j^{\alpha_i}$ is the ID of the output channel of α_i, $s(O_j^{\alpha_i})$ denotes the tokens production rate of $O_j^{\alpha_i}$. From the SDAG model, the total number of tokens consumed $sum(tokens)$ per unit period and the minimum space of the FIFO cache queue $min(FIFO)$ are defined as follows.

$$sum(tokens) = \sum_{i=1}^{n} \sum_{I^{\alpha_i} \in IN(\alpha_i)} r(I^{\alpha_i}) \tag{12}$$

$$min(FIFO) = \sum_{i=1}^{n} \sum_{O^{\alpha_i} \in OUT(\alpha_i)} s(O^{\alpha_i}) \tag{13}$$

$E = \{e_1, e_2, \ldots, e_k\} \in V \times V$ denotes the set containing k directed edges and e_i is the directed edge ID, $i = 1, \ldots, k$.

We use $c_i (i = 1, 2, \ldots)$ to denote the ID of the processor cores. For homogeneous multi-core systems, each ID is associated with a logical core. $p(c_i)$ denotes the performance of the processor core c_i, and since the processor cores are homogeneous, we use $p(c)$ denote the performance of the processor. $et(\alpha_i)$ denotes the execution time of the instance α_i at the processor core, including the FIFO queue operation time L, which is defined as follows.

$$et(\alpha_i) = \frac{W(\alpha_i)}{p(c)} + L \tag{14}$$

As instances communicate with each other in shared memory and instances are created sequentially, the FIFO queue operation time and "pass lock" time are very short compared to the execution time of instance α_i, so we use a constant L to represent the FIFO queue operation time for all instances.

$U(c_i)$ denotes all instances of c_i assigned to the processor. $cl(c_i)$ denotes the finish time at which processor c_i has executed all instances in $U(c_i)$. $l(c_i)$ denotes the load on processor c_i, defined as follows:

$$l(c_i) = \sum_{\alpha \in U(c_i)} et(\alpha) \tag{15}$$

When instance α_i is assigned to processor core c_i, $AFT(\alpha_i)$ denotes the actual execution end time of instance α_i. α_{exit} denotes the terminating instance of the HSDFG. If a given HSDFG has multiple terminating instances, then *makespan* is defined as follows.

$$makespan = max(AFT(\alpha_{exit})) \tag{16}$$

Our goal is to make full use of the limited number of processor cores to minimize s. Therefore, the objective function is defined as follows:

$$min\ makespan$$

4.3.2. Instance Selection

$level(\alpha_i)$ is the level value of instance α_i, which represents the sum of the execution times of all instances contained in the longest path from α_i to the endpoint instance, as defined below.

$$level(\alpha_i) = \sum_{\alpha \in S} et(\alpha) \tag{17}$$

S in Equation (16) denotes the set of all instances contained in the longest path from α_i to the endpoint instance. In the LSEFT algorithm, the level value of an instance is used as the priority when scheduling, and the instance with the higher level value among all instances is scheduled for execution first. In the level value calculation of the PAPS algorithm, if there are many endpoint instances, then additional vertices need to be created to connect these endpoint instances. We propose an improved method for level value calculation, the reverse traversal to solve for level. The main idea is to use the reverse traversal of multi-tree nodes and the Algorithm 2 is described.

Algorithm 2: Calculate Level Values of Instances

Input: HSDFG
Output: each instance level value $level(\alpha_i)$

```
Put all instances of HSDFG are in reverse order into queue F, each instance
has a set of precursor instance pre(ins)
level(ins) = et(ins), ins ∈ F
for all ins ∈ F do
    for all pins ∈ pre(ins) do
        new_level = et(pins) + level(ins)
        if level(pins) < new_level then
            level(pins) ← new_level
        end if
    end for
end for
```

The HSDFG is reconstructed from the SDFG, and the reconstruction process is also a simulated execution of the SDFG. Instances of actors in the SDFG need to be created, the order in which each actor is created is recorded, and all instances are added to the queue F in reverse order. Using this special structure, the exact level value can be obtained by traversing all instances in the HSDFG once.

If there are many instances with the same level value, we consider them to be at the same level and add them to the queue D. $num_t(D)$ represents the number of instances in the queue D at time t. For the instances in the to-be-scheduled queue D, we will use the short-job-first policy, defined as follows:

$$min(et(\alpha)),\ \alpha \in D \tag{18}$$

4.3.3. Core Selection

The PAPS algorithm does not use a policy for the core selection phase, but in our experiments we compare the load balancing policy with the earliest finish time(EFT) policy, and the results show that the EFT policy performs better. We therefore use the EFT policy in the core selection phase, defined as follows.

$$min(cl(c_i)),\ i = 1, 2, \ldots, M \tag{19}$$

4.3.4. Overall Algorithm

Algorithm 3 describes the overall process of the LSEFT scheduling algorithm.

Algorithm 3: LSEFT Algorithm

Input: queue F, set of cores M
Output: scheduling result

```
Create empty queue D
cl(c) = 0, c ∈ C
Compute level value using Algorithm 1
repeat
    Put instance α with maximum level into D
    Pick instance α_i from D using Equation 18)
    Assign instance α_i to core c_i using Equation 19)
    cl(c_i) ← cl(c_i) + et(α_i)
until queue F is empty
```

The LSEFT algorithm uses a combination of level values and a short-job-first policy to select instance. In Algorithm 3, the instances with the largest level value are first selected

from queue F each time and these instances are added to the to-be-scheduled queue D. Afterwards, the instance with the smallest execution time is selected from the to-be-scheduled queue D. If there are many instances with the same minimum execution time, the instance that enters the to-be-scheduled queue D earliest is selected first. Finally this instance is assigned to the processor with the minimum finish time, and if many processors have the same minimum finish time, the algorithm will select the one with the smallest processor ID.

For the scheduling problem of a given SDFG, the SDFG is first converted into an HSDFG using Algorithm 1, where the data structure of the converted HSDFG is a cross-linked table. Then, using Algorithm 2, the priority of each instance in the HSDFG is calculated, and finally, using Algorithm 3, a suitable processor is selected for each instance to execute based on the priority of each instance to optimise the objective function.

5. Evaluation

5.1. SDFG Random Generator

To evaluate the advantages of our proposed algorithm, we designed an SDFG random generator. The SDFG generator can randomly generate SDFGs that satisfy the consistency requirement, and then convert the SDFGs to equivalent HSDFGs using the algorithm in the [3]. In the conversion algorithm, how to choose the optimal *unfolding factor* [31] is not a problem we consider, so we only consider the case where the unfolding factor is 1, i.e., only the HSDFG consisting of all instances under a unit period is considered.

The main modules of our designed SDFG random generator are given in Figure 6, along with the relationships between the modules. The number of randomly generated nodes and directed edges obey a uniform distribution, and only the maximum values of the number of nodes and directed edges need to be set at the APIs. The number of nodes and directed edges are checked to ensure that both satisfy the boundedness of the directed graph. The key function implemented by the randomly generated directed graph module is to randomly construct the adjacency matrix and then do a connectivity test on it. Randomly generated port rates are used to make the SDFG satisfy the consistency requirement; randomly generated initial caches are used to resolve the deadlock situation where the SDFG has a directed loop. The parameters of the SDFG random generator that we use in this evaluation are shown in Table 2.

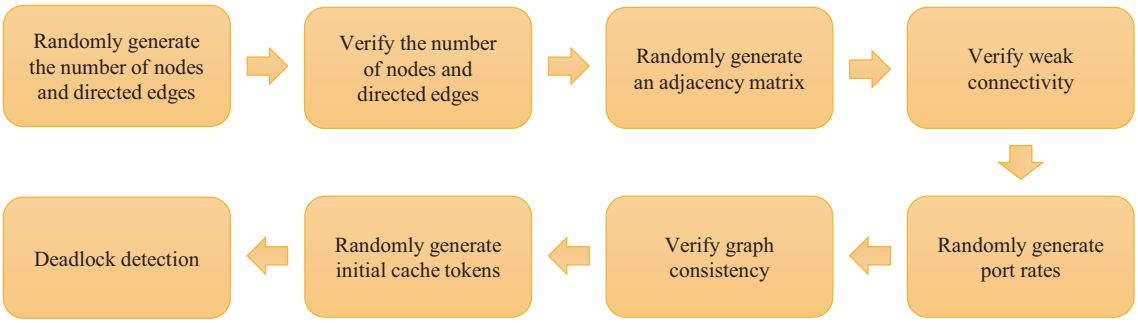


Figure 6. SDFG random generator process.

Table 2. Evaluation settings for SDFG random generator.

Parameters	SDFG
Number of SDFG	958
Number of nodes	$U(1,50)$
Number of edges	$U(1,80)$
Port rate	$U(1,3)$
Execution time	$U(1,3)$

5.2. SDFG Data

The data used in this study were open sourced on Github. The SDFG Randomizer generated a total of 958 SDFGs using the parameters shown in Table 2. A total of 958 SDFGs were divided into ten groups, each of which contained 90+ SDFGs. These were saved to a file in csv format. In a csv file, an SDFG consists of three attributes, namely, "et" "tm" and "buf". "et" is a one-dimensional vector representing the execution time of each role in the SDFG; "tm" is the topology matrix of the SDFG, which contains the port rate information of the roles in the SDFG, the dependency information with the rest of the roles, and is the core of the SDFG; "buf" is the topology matrix of the SDFG. It is the core of SDFG; "buf" is also a vector that represents the number of caches that all channels in SDFG have, which is used to break the deadlock structure. Github: https://github.com/wangcgzzu/SDFG-data.git (accessed on 9 January 2022).

When the LSEFT algorithm was executed, the data in the csv file were read sequentially by line and parsed to an SDFG. After the scheduling algorithm was completed, we obtained the SDFG's, and stored this makespan in the csv file.

5.3. Speed Up

For a given application and computing platform, the program execution time is the most direct reflection of the effectiveness of the program execution. Our computing platform is a simulated homogeneous multi-core platform, and the execution time of each instance is also randomly generated in the configuration shown in Table 2. Although *makespan* is the most commonly used metric, it does not provide an intuitive reflection of the strengths and weaknesses of parallel algorithms, so we used the *speedup*, which is defined as follows.

$$\eta = \frac{SFT}{makespan} \tag{20}$$

where SFT denotes the execution finish time of all instances of HSDFG for a single core per unit period. Based on the load we defined (15), we added a load-balancing policy to the PAPS algorithm. In the core selection phase, the core with the lowest current load is selected, this algorithm is called the PAPS-B algorithm. We first generated a specific number of SDFGs using the SDFG random generator and then converted the SDFGs to HSDFGs using the conversion algorithm in [3] to test the speedup at different processor counts.

Table 3 shows the speedup results under different scenarios. In this case, the number of instances of HSDFG is 8, 32, 64 and 128. Four different sizes of SDFG were used as test cases to test the speedup at different processor core counts. The number of simulated processor cores M is 2, 4 and 8, respectively. Three scheduling algorithms are compared, namely, the PAPS algorithm, the PAPS-B algorithm and our proposed LSEFT algorithm. The purpose of comparing the PAPS-B algorithm is to compare the advantages of the EFT policy with balancing policy. From the experimental results in Table 3, it can be seen that the speedup of the LSEFT algorithm are all improved to varying degrees compared to the algorithm before the improvement. In particular, when the number of instances is higher and the number of processors is greater, the speedup of the LSEFT algorithm is improved more significantly. Because instances are represented as task graphs, there are dependencies between instances and instances can only be scheduled for execution when all dependencies are satisfied. If the priority of the currently ready instances is consistent, then the short job priority policy is used, giving short execution times. At the end of the execution of the short instance, more instances of the ready state are available, which have parallelism between them and can further be assigned to execute in different processor cores. The parallelism is fully exploited, the *makespan* is reduced and therefore a higher speedup is obtained. The results in Table 3 also show that the LSEFT algorithm, with a higher number of processors, also achieves a higher speedup.

Table 3. Speed up vs. number of instances with different scheduling algorithms.

Number of Instances	M	Speed Up η of PAPS	Speed Up η of PAPS-B	Speed Up η of LSEFT
8	2	1.56	1.56	1.75
32	2	1.19	1.33	1.33
64	2	1.09	1.09	1.24
128	2	1.04	1.20	1.20
8	4	2.00	2.00	2.33
32	4	1.33	1.33	1.33
64	4	1.24	1.24	2.15
128	4	1.20	1.20	**2.24**
8	8	2.33	2.33	2.33
32	8	1.33	1.33	2.80
64	8	1.24	2.04	3.29
128	8	2.02	2.02	**3.61**

Without loss of generality, we used the SDFG random generator, configured as shown in Table 2, to generate 958 SDFGs. The same conversion algorithm in the [3] is used to convert to 958 HSDFGs. We processed 958 HSDFGs using the PAPS, PAPS-B and LSEFT algorithms, respectively. The PAPS, PAPS-B and LSEFT algorithms calculate the speedup using Equation (20) after each HSDFG is processed and save the speedup. When all the test case data were processed, the speedup for each test case were also calculated and stored. Finally, the average of these speedups was calculated; the results are shown in Figure 7. In Figure 7, we take the average of 958 speedups as the measurement data. The performance ranking of the algorithm based on the average speedup is LSEFT, PAPS-B, PAPS. For the generated SDFG, LSEFT outperforms the rest of the algorithms. Moreover, the average speedup of the LSEFT algorithm performs well for the 8-core simulation, with an average speedup improvement of 10.17% compared to the PAPS algorithm. Furthermore, Figure 7 shows that the EFTstrategy is superior to the load balancing strategy.

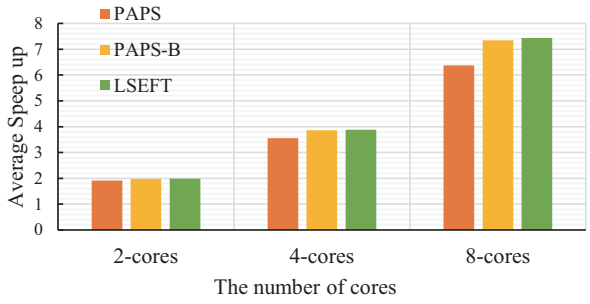


Figure 7. Average speed up with different scheduling algorithms.

5.4. Parallel Efficiency

Parallel efficiency reflects the practical effect of an increase in hardware resources in computing. On homogeneous processor system, parallel efficiency Eff is defined as follow.

$$Eff = \frac{\eta}{M} \tag{21}$$

For a homogeneous processor system, the execution performance of each processor core is the same. For a given number of HSDFGs and processor cores, the denominator of Equation (21) is constant, so the smaller the makespan, the larger the speed up η, and the higher the parallel efficiency, corresponding to the better the scheduling algorithm. We also tested the average parallelism efficiency using 958 HSDFGs for each scheduling algorithm. The performance ranking of the algorithms based on average parallel efficiency is LSEFT, PAPS-B, PAPS. The results are shown in Figure 8, where the parallel efficiency of the PAPS

algorithm decreases as the number of processor cores increases. Although the parallel efficiency of the LSEFT algorithm also decreases, the decrease in LSEFT is significantly lower than that of the PAPS algorithm. In terms of the decreasing trend, the PAPS algorithm has a steeper decreasing trend, implying that the parallel efficiency of the PAPS algorithm will decrease more when the number of processor cores increases. Therefore, the results in Figure 8 also clearly show that the LSEFT algorithm has a higher and more stable parallel efficiency.

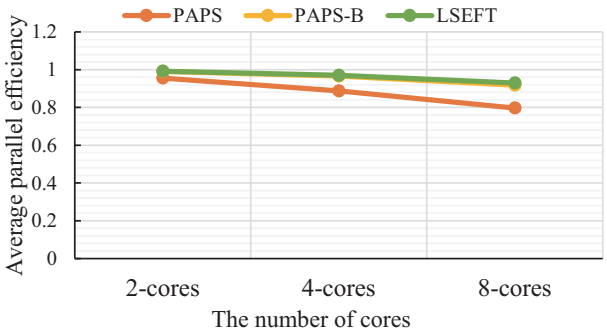


Figure 8. Average parallel efficiency with different scheduling algorithms.

6. Conclusions

SDFG is widely used to model streaming applications. With the current proliferation of dedicated homogeneous multi-core processors, it is important that streaming applications make full use of these computational resources.

We extend the DAG model using data-driven methods to propose the SDAG model as a model for parallel computation of SDFGs under the von Neumann architecture. Then, we improved the PAPS algorithm and proposed the LSETF algorithm for scheduling SDFGs on a homogeneous multi-core processor system. Based on an experimental study of 956 randomly generated SDFGs of different sizes, the LSEFT algorithm outperforms the pre-improvement algorithm in terms of performance metrics (speedup and parallel efficiency). The LSEFT algorithm is a feasible solution to the SDFG scheduling problem on homogeneous systems.

For the task priority calculation method of the LSEFT algorithm, we studied a new calculation method to avoid solving the longest path in the DAG. In the processor selection phase, we compared the load balancing strategy with the shortest completion time strategy. The disadvantage of the LSEFT algorithm is that it is based on HSDFG scheduling and cannot be scheduled in the original SDFG graph. Furthermore, the LSEFT algorithm does not take into account the communication overhead between processors when scheduling. We will extend the LSEFT algorithm to accommodate heterogeneous processor systems while incorporating communication overhead in the future work. We also working on future scheduling methods based on reinforcement learning and our dataset can be used for reinforcement learning as well.

Author Contributions: Conceptualization, L.W., C.W. and H.W.; methodology, L.W.; software, C.W.; validation, L.W., C.W. and H.W.; formal analysis, L.W.; data curation, C.W.; writing—original draft preparation, C.W.; writing—review and editing, L.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by State Key Laboratory of Complex Electromagnetic Environment Effects on Electronics and Information System (project no. CEMEE2018K0302B), Sichuan Science and Technology Program in China Grant Number 2020YFG0059 and Fundamental Research Funds for the Central Universities in China Grant Number ZYGX2019J128.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are openly available in [Github] at [https://github.com/wangcgzzu/SDFG-data.git] (accessed on 9 January 2022).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

DAG	Directed Acyclic Graphs
FIFO	First In First Out
HSDFG	Homogeneous SDFG
LSEFT	Level Shortest First Earliest Finish Time
SDAG	Super Directed Acyclic Graphs
SDFG	Synchronous Data Flow Graphs
PAPS	Periodic Admissible Parallel Schedule

References

1. Damavandpeyma, M.; Stuijk, S.; Basten, T.; Geilen, M.; Corporaal, H. Schedule-Extended Synchronous Dataflow Graphs. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **2013**, *32*, 1495–1508. [CrossRef]
2. Lee, E.; Messerschmitt, D. Synchronous data flow. *Proc. IEEE* **1987**, *75*, 1235–1245. [CrossRef]
3. Lee, E.A.; Messerschmitt, D.G. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. Comput.* **1987**, *100*, 24–35. [CrossRef]
4. Lin, H.; Li, M.F.; Jia, C.F.; Liu, J.N.; An, H. Degree-of-node task scheduling of fine-grained parallel programs on heterogeneous systems. *J. Comput. Sci. Technol.* **2019**, *34*, 1096–1108. [CrossRef]
5. Cole, R.; Zajicek, O. The APRAM: Incorporating Asynchrony into the PRAM Model. In Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA'89, Santa Fe, NM, USA, 18–21 June 1989; Association for Computing Machinery: New York, NY, USA, 1989; pp. 169–178. [CrossRef]
6. Suettlerlein, J.; Zuckerman, S.; Gao, G.R. An implementation of the codelet model. In Proceedings of the European Conference on Parallel Processing, Aachen, Germany, 26–30 August 2013; Springer: Berlin/Heidelberg, Germany, 2013, pp. 633–644. [CrossRef]
7. Hu, T.C. Parallel sequencing and assembly line problems. *Oper. Res.* **1961**, *9*, 841–848. [CrossRef]
8. Thies, W.; Karczmarek, M.; Amarasinghe, S. StreamIt: A Language for Streaming Applications. In *Compiler Construction*; Horspool, R.N., Ed.; Springer: Berlin/Heidelberg, Germany, 2002; pp. 179–196.
9. Zhang, W.; Wei, H.; Yu, J. COStream: A language for dataflow application and compiler. *Chin. J. Comput.* **2013**, *36*, 1993–2006. [CrossRef]
10. Veen, A.H. Dataflow Machine Architecture. *ACM Comput. Surv.* **1986**, *18*, 365–396. [CrossRef]
11. Kung, S.Y.; Arun.; Gal-Ezer.; Rao, B. Wavefront Array Processor: Language, Architecture, and Applications. *IEEE Trans. Comput.* **1982**, *100*, 1054–1066. [CrossRef]
12. JéJé, J. *An Introduction to Parallel Algorithms*; Addison-Wesley: Reading, MA, USA, 1992; Volume 10, p. 133889.
13. Dorostkar, F.; Mirzakuchaki, S. List Scheduling for Heterogeneous Computing Systems Introducing a Performance-Effective Definition for Critical Path. In Proceedings of the 2019 9th International Conference on Computer and Knowledge Engineering (ICCKE), Mashhad, Iran, 24–25 October 2019 ; pp. 356–362. [CrossRef]
14. Wu, Q.; Zhou, M.; Zhu, Q.; Xia, Y.; Wen, J. MOELS: Multiobjective Evolutionary List Scheduling for Cloud Workflows. *IEEE Trans. Autom. Sci. Eng.* **2020**, *17*, 166–176. [CrossRef]
15. Djigal, H.; Feng, J.; Lu, J. Performance Evaluation of Security-Aware List Scheduling Algorithms in IaaS Cloud. In Proceedings of the 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), Melbourne, Australia, 11–14 May 2020; pp. 330–339. [CrossRef]
16. Bo, Y.; Benfa, Z.; Feng, L.; Xiaoyu, Z.; Xinjuan, L.; Fang, W.; Zhitao, Z. List scheduling algorithm of improved priority with considering load balance. In Proceedings of the 2021 IEEE International Conference on Data Science and Computer Application (ICDSCA), Haikou, China, 29–31 October 2021; pp. 324–327. [CrossRef]
17. Suzuki, M.; Ito, M.; Hashidate, R.; Takahashi, K.; Yada, H.; Takaya, S. Fundamental Study on Scheduling of Inspection Process for Fast Reactor Plants. In Proceedings of the 2021 2020 9th International Congress on Advanced Applied Informatics (IIAI-AAI), 1–15 September 2020; pp. 797–801. [CrossRef]
18. Miao, C. Parallel-Batch Scheduling with Deterioration and Group Technology. *IEEE Access* **2019**, *7*, 119082–119086. [CrossRef]
19. Hu, Z.; Li, D.; Zhang, Y.; Guo, D.; Li, Z. Branch Scheduling: DAG-Aware Scheduling for Speeding up Data-Parallel Jobs. In Proceedings of the 2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS), Phoenix, AZ, USA, 24–25 June 2019; pp. 1–10. [CrossRef]
20. Wu, X.; Yuan, Q.; Wang, L. Multiobjective Differential Evolution Algorithm for Solving Robotic Cell Scheduling Problem with Batch-Processing Machines. *IEEE Trans. Autom. Sci. Eng.* **2021**, *18*, 757–775. [CrossRef]

21. Moi, S.H.; Yong, P.Y.; Weng, F.C. Genetic Algorithm Based Heuristic for Constrained Industry Factory Workforce Scheduling. In Proceedings of the 2021 7th International Conference on Control, Automation and Robotics (ICCAR), Singapore, 23–26 April 2021; pp. 345–348. [CrossRef]
22. Yang, S.; Xu, Z. Intelligent Scheduling for Permutation Flow Shop with Dynamic Job Arrival via Deep Reinforcement Learning. In Proceedings of the 2021 IEEE 5th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC), Chongqing, China, 12–14 March 2021; Volume 5, pp. 2672–2677. [CrossRef]
23. Jeong, D.; Kim, J.; Oldja, M.L.; Ha, S. Parallel Scheduling of Multiple SDF Graphs Onto Heterogeneous Processors. *IEEE Access* **2021**, *9*, 20493–20507. [CrossRef]
24. Bonfietti, A.; Benini, L.; Lombardi, M.; Milano, M. An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms. In Proceedings of the 2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010), Dresden, Germany, 8–12 March 2010; pp. 897–902. [CrossRef]
25. Ali, H.I.; Stuijk, S.; Akesson, B.; Pinho, L.M. Reducing the Complexity of Dataflow Graphs Using Slack-Based Merging. *ACM Trans. Des. Autom. Electron. Syst.* **2017**, *22*, 1–22. [CrossRef]
26. Bambha, N.; Kianzad, V.; Khandelia, M.; Bhattacharyya, S.S. Intermediate representations for design automation of multiprocessor DSP systems. *Des. Autom. Embed. Syst.* **2002**, *7*, 307–323. [CrossRef]
27. Choi, J.; Ha, S. Worst-Case Response Time Analysis of a Synchronous Dataflow Graph in a Multiprocessor System with Real-Time Tasks. *ACM Trans. Des. Autom. Electron. Syst.* **2017**, *22*, 1–26. [CrossRef]
28. Jung, H.; Oh, H.; Ha, S. Multiprocessor Scheduling of a Multi-Mode Dataflow Graph Considering Mode Transition Delay. *ACM Trans. Des. Autom. Electron. Syst.* **2017**, *22*, 1–25. [CrossRef]
29. Bouakaz, A.; Fradet, P.; Girault, A. Symbolic Analyses of Dataflow Graphs. *ACM Trans. Des. Autom. Electron. Syst.* **2017**, *22*, 1–25. [CrossRef]
30. Stuijk, S.; Geilen, M.; Basten, T. SDF3: SDF For Free. In Proceedings of the Sixth International Conference on Application of Concurrency to System Design (ACSD'06), Turku, Finland, 28–30 June 2006; pp. 276–278. [CrossRef]
31. Rajadurai, S.; Alazab, M.; Kumar, N.; Gadekallu, T.R. Latency Evaluation of SDFGs on Heterogeneous Processors Using Timed Automata. *IEEE Access* **2020**, *8*, 140171–140180. [CrossRef]