Throughput-Driven Partitioning of Stream Programs on Heterogeneous Distributed Systems

Vu Thien Nga Nguyen and Raimund Kirner, *Member, IEEE*

Abstract—Graph partitioning is an important problem in computer science and is of NP-hard complexity. In practice it is usually solved using heuristics. In this article we introduce the use of graph partitioning to partition the workload of stream programs to optimise the throughput on heterogeneous distributed platforms. Existing graph partitioning heuristics are not adequate for this problem domain. In this article we present two new heuristics to capture the problem space of graph partitioning for stream programs to optimise throughput. The first algorithm is an adaptation of the well-known Kernighan-Lin algorithm, called *KL-Adapted* (KLA), which is relatively slow. As a second algorithm we have developed the *Congestion Avoidance* (CA) partitioning algorithm, which performs reconfiguration moves optimised to our problem type. We compare both KLA and CA with the generic meta-heuristic *Simulated Annealing* (SA). All three methods achieve similar throughput results for most cases, but with significant differences in calculation time. For small graphs KLA is faster than SA, but KLA is slower for larger graphs. CA on the other hand is always orders of magnitudes faster than both KLA and SA, even for large graphs. This makes CA potentially useful for re-partitioning of systems during runtime.

Index Terms—Stream programs, throughput optimisation, graph partitioning, simulated annealing

1 Introduction

Stream programming is a paradigm where a program is structured by a set of computational nodes connected by streams. Focusing on data moving between computational nodes via streams, this programming model fits well for applications which process long sequences of data. In stream programming, concurrency is expressed implicitly via communication streams. This helps to reduce the complexity of parallel programming. For this reason, stream programming has gained popularity as a programming model for parallel platforms, and has been the research focus of several projects such as StreamIt [45], Brook [7], S-Net [19], etc.

To obtain good performance for stream programs on parallel platforms with shared memory, above mentioned projects have deployed numerous scheduling strategies. The main trend is to maximise the parallelism among all cores by mapping computational nodes of the stream program into individual cores. Different graph partitioning techniques have been taken on to provide load balance so far either regardless the communication cost or consider it only with a low priority [8], [12], [18].

On distributed platforms where communication cost has an enormous effect on performance, these techniques are no longer suitable. There are some approaches proposed to resolve this problem, as described in Section 2. However, all of them are limited to either a specific class of stream program, or a particular type of target platform.

• *The authors are with the University of Hertfordshire, United Kingdom.*

Manuscript received 30 May 2014; revised 9 Feb. 2015; accepted 14 Feb. 2015. Date of publication 25 Mar. 2015; date of current version 12 Feb. 2016. Recommended for acceptance by T. Hoefler.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2416726

In this paper, we propose a graph partitioning method particularly suitable to optimise the throughput of general stream programs on heterogeneous distributed platforms. As an old NP-hard problem [16], graph partitioning has a significant volume of existing work. The usual solutions are heuristic and approximation algorithms trying to divide a graph into separated partitions to optimise an objective. The most common and well investigated objective is to equalise the size of partitions while minimising the total cuts between them. However, this objective is not necessarily best suited for all problems. For stream programs, the throughput is decided not purely by the workload on each partition, but also by the communication cost between each pair of partitions. It is even more complicated when the distributed platform has heterogeneous resources and the communication bandwidth is not uniform among them. In Section 4 we introduce a method to calculate the throughput of stream programs based on these factors. In Section 5 we present two graph partitioning algorithms which use this formula as their objective function. The first one, called *KL-Adapted* (KLA), is a simple adaptation from the classic Kernighan-Lin (KL) graph partitioner [24]. It uses greedy heuristics to search for improved mappings by trying all possible move operations based on the current mapping. For each possible move, this method needs to re-calculate the objective function. For this reason, this method is rather slow although it achieves results of sufficient quality. Thus, we have developed a second algorithm, called *Congestion Avoidance* (CA), that narrows the search space. Instead of examining all possibilities, this method concentrates on moves around congestion points where the throughput is dimmed. The evaluation of KLA and CA in Section 6 shows that they achieve same or better results than the

genetic meta-heuristic Simulated Annealing (SA). The evaluation also shows that CA is significantly faster than KLA and also Simulated Annealing.

2 Related Work

2.1 General Graph Partitioning

Graph partitioning is the problem of dividing a graph's vertices into subsets that meet some requirements. Graph partitioning is NP-hard [16] and is commonly used in various applications such as VLSI design [42], image processing [43], distributing workloads for parallel computations [10], etc. The classic requirement of graph partitioning consists of two criteria: balancing vertex weight between subsets (called balance criterion), and minimising the total edge cut among these subsets (called total cut criterion).

In graph partitioning, algorithms using iterative improvement are the most common. Such an approach favours the balance criterion, i.e., divides the graph into subsets so that their weights of vertices are approximately equal and then applies refinement methods to move vertices between them to find the optimal total cut criterion. As a local search, the iterative improvement starts with an initial solution and repeatedly performs local perturbation of the current solution. For the local perturbation, iterative improvement can employ greedy heuristics such as Kernighan-Lin [24] and its algorithmic improvement Fiduccia-Mattheyses (FM) [15]. It can also use hill-climbing techniques such as simulated annealing [26].

As KL/FM was shown empirically to be efficient [22], [38], it is usually used in local refinements in recent multilevel partitioning schemes. Some examples are METIS [23], JOSTLE [46], KaFFPa [41], SCOTCH [36]. PaToH [9] is another example of using KL/FM for refinement in the multilevel approach for partitioning hyper-graphs. All these approaches first coarsen vertices according to some matching criterion to create a smaller graph at a new level. The coarsening stage is repeated until reaching the lowest level. An initial partitioning phase is used to generate partitions. The uncoarsening phase walks up each level, and applies local refinement based on KL/FM method. Employing the similar basic idea of multilevel partitioning, PARTY [33] instead uses another heuristic called Helpful-Set which is derived by theoretical analysis. KaFFPa also employs Max-Flow Min-Cut as another local refinement technique [41].

Another approach, called spectral partitioning, optimises the total edge cut by using the eigenvalues and eigenvectors of the graph. Unlike the iterative improvement, this approach aims to find the global optimal point. Examples of this approach can be found in [4], [39]. This approach is known to find reasonably solutions but is very slow to run compared to iterative improvement. To improve the spectral approach, there are some proposals to combine the balance and total cut criteria into a sole metric, for example Ratio Cut [20] and Sparest Cut [6], [25].

While most of the approaches focus on the balance and total cut criteria, some others claim that they are not efficient for some specialised domains. For example, Aspect Ratio [13] and the Partition Shape [14] are shown to be a better metric while partitioning solvers using the finite element method (FEM). Most of the partitioning strategies working on these criteria use iterative improvement. Starting with a set of seeds, a growing method is used to generate corresponding subsets. The centres of those subsets are used as seeds for the next iteration. The growing method can be based on a greedy breadth-first search [14], or based on the diffusive process [32].

2.2 Graph Partitioning in Stream Programming

In stream programming, graph partitioning is typically used to map a stream program onto a target platform. A stream program can be described as a graph $G_{\mathrm{SP}} = \langle T, S \rangle$ whose vertices (T) are computation nodes and edges are communication channels (S). The vertex weight is the load on each computational node, and the edge weight is the amount of transferred data on each communication channel. The target platform is a graph $G_{\mathrm{PF}} = \langle \mathrm{PE}, L \rangle$ with the nodes being the processing elements (PEs) and the edges being the communication links (L) that connect the PEs. Typically, the graph to model the target platform is undirected. Graph partitioning techniques are used to divide the stream program graph into subsets, each of which is mapped onto a PE. The stream program partitioning problem is thus to map G_{SP} onto G_{PF}, that is $\pi : G_{\mathrm{SP}} \rightarrow G_{\mathrm{PF}}$ such that a specified cost function is optimised. The mesh-based application partitioning problem [47] is equivalent to the stream program partitioning problem as they have the same type of graph mapping model. For the mapping objective, regardless of stream programs or mesh-based applications, the abstraction level between the real program and the graph model is relevant.

The most common optimisation goal in partitioning stream programs is to optimise the throughput. The throughput is determined by both the load/weight of vertex subsets, and the individual communications/cuts between each pair of vertex subsets. Throughput optimisation can be expressed into two criteria: balanced criterion and individual cut criterion. The balanced criterion is defined similarly to traditional graph partitioning problems; it is the balance of load among all subsets. The individual cut criterion, in contrast, involves not the total cut, but individual cuts between each pair of subsets. The cut between two subsets in fact represents the required communications between them. Moreover, unlike in traditional graph partitioning problems, the balance criterion and individual cut criterion should not be considered separately but always jointly, because the throughput is defined by the combination of both. For these above reasons, algorithms used for traditional graph partitions are not applicable to partition stream programs.

Compared to other stream programming models, synchronous data flow (SDF) [27] has attracted most of research for its synchronity property as the name suggests. SDF programs differ from conventional stream programs in that their nodes have static input and output rates. That means the amount of messages that each node consumes and produces during each invocation is constant and predefined statically. Generally, in an SDF program streams are unidirectional; node computation is deterministic; node communications are synchronous; and the program structure is static. These properties are used to generate periodic schedules at compile time [28]. The execution of an SDF program is simply an iteration of this schedule in the sense that data

required for the next iteration is generated in the previous iterations. In addition, nodes in SDF are stateless, i.e., their outputs do not change for the same input data. This property enables fusion operations, which combine multiple nodes into one; and fission operations, which are the reversing processes of fusion operations.

Taking advantage of the periodic schedule, there have been numerous approaches for mapping SDF programs onto parallel platforms, including shared-memory and distributed platforms. As the communication cost on shared-memory platforms is small and thus ignored by several approaches, we include here only those approaches that take account of communication cost and therefore are potentially extendible for distributed platforms. One example is described in [18] where the author attempts to divide the program's periodic schedule. Without considering the communication cost, the proposed technique aims for a balanced load by greedily applying fission and fusion operations based on the number of nodes and the number of PEs. The later approach from the same authors first obtains the load balance by using a simple greedy heuristic and then minimises the total communication by applying fusion operations [17]. In [12], although the role of individual communication cost is recognised, still the total communication cost is used to remove potential bottlenecks.

One of the first attempts to map SDF programs onto homogeneous multiprocessor architectures is the work of Sih and Lee [44]. The goal of this approach is to minimise the *make-span*, which is the execution time of a single periodic schedule and also is inversely proportional to the throughput. Based on graph analysis techniques, the authors propose a new clustering algorithm to form clusters of nodes. The algorithm takes into account the trade-off between benefits from parallelism and inter-PE communication costs. By considering the inter-cluster communication and the parallelism relationships, the approach hierarchically combines these clusters into a binary tree. It then traverses the binary tree from the top level to the bottom level, and decomposes the binary tree to a group of clusters that fits the target platform.

The approach in [8] is designed to map a class of stream programs, which are Kahn process networks with some SDF properties, onto heterogeneous multiprocessor systems. The approach uses a cost function which is inversely proportional to the throughput. The cost function is defined as the maximum of the computational cost of each PE; and the communication cost between each pair of PEs. The authors propose a two-phase partitioning algorithm to minimise the cost function. The first phase is to recursively bi-partition both the stream program and the set of PEs on the platform. Partitioning the stream program aims to minimise the cost function. Partitioning the set of PEs aims to maximise a function that harmonises the balance and the communication links. The second phase, refinement, tries to get rid of bottlenecks lying on the computation of PEs. This phase also considers some other constraints of convexity. The authors claim that the convexity is a guideline to avoid long pipelines and therefore to reduce the memory requirement as well as the latency. However, no proof has been provided and the experiments focus only on the total execution time instead of throughput and latency.

As the first approach using ILP to partition the periodic schedule of SDF programs onto heterogeneous architectures, the work in [29] aims to optimise the throughput. This work provides an ILP formulation based on the resource constraints, scheduling constraints and dependency constraints; whilst at the same time taking advantage of stateless nodes in granularity optimisation. The main goal of this work is to partition the periodic schedule not at the node level but at the node invocation level, i.e., a node can be executed by multiple different PEs within a periodic schedule. The optimisation goal is the *make-span* which is inversely proportional to the throughput. One drawback of this approach is that the ILP formulation does not model well simultaneous multi-thread execution. This leads to the limitation that node execution on each PE is sequential. This drawback does not occur in the later work from the same authors [30]. In this work, the authors use the Uppaal model checker [5] instead of ILP.

The work in [11] focuses on mapping router applications onto multi-core packet processing systems. These applications are structurally similar to stream programs with the constraint that their tasks do not hold a persistent state. Therefore their tasks can be duplicated to increase parallelism. The main target of the mapping problem is to obtain load balance and minimise the total communication cost. This work introduces two new algorithms and compares them with the well known Kernighan-Lin and Simulated Annealing algorithms. The first new algorithm is UDFS that duplicates tasks until the amount of tasks is equal to the amount of cores in the system, and uses a greedy mapping scheme to assign each task to each core while minimising the total communication cost among machines. The second is an extension of the Kernighan Lin algorithm where both the total communication cost and the load balance are incorporated into the gain function. The comparison in this work focuses on the quality of output mapping configuration rather than the execution time of the algorithms. This work also shows that applying task duplication and merging before mapping can help to find better mapping configurations for all algorithms.

To summarise, most of the above approaches are limited to a specific class of stream programs. Those are stream programs with static scheduling properties similar to StreamIt. Our work in this article, on the other hand, aims to partition general stream programs for heterogeneous distributed platforms. Also most of these approaches do no use the throughput as the direct optimisation target of the partitioning algorithm. They instead use similar optimisation targets as general partitioning algorithms, i.e., they obtain the load balance first while minimising the total communication cost. The approach in [8] is the only one that uses a cost function that is the reverse of our throughput formula. However, during the refinement phase of the partitioning algorithm, their approach considers only bottlenecks at the computational load of partitions while ignoring those at the communication costs between partitions.

3 Background

3.1 Stream Programming

In stream programming, a program is structured by a set of computation nodes and a set of communication channels

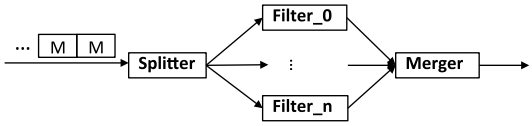


Fig. 1. Program structure of image filter application.

called streams. In this paper we refer to computation nodes simply as nodes. Streams are used to connect nodes in different patterns such as pipeline, parallel, etc. Some examples of stream programming can be found in [7], [19], [45]. The structure of stream programs can be illustrated as a graph whose vertices are nodes and edges are streams. In this work, we assume that one stream connects only two nodes. Data of stream programs is usually presented as an infinite sequence of input messages.

Fig. 1 shows an example of a stream program. This is the Image Filter application which includes node *Splitter* that reads in images and splits each of them into sub-images. The number of sub-images is varied depending on the size of the original image. All sub-images are scattered in different branches where nodes *Filter_i* $\mid i \in \{1..n\}$ carry out filtering on each sub-image. Then filtered sub-images are sent to node *Merger*. This node unifies them into a complete image and sends it out.

3.2 Nodes and Data in Stream Programs

In stream programs, data arrives from the environment as a virtually infinite sequence of messages. Nodes that receive messages from the environment are *entry nodes*. Nodes sending messages out to the environment are *exit nodes*. In the example of Fig. 1, *Splitter* is an entry node and *Merger* is an exit node. A stream program can have multiple entry nodes and multiple exit nodes. Input messages of entry nodes come from the external environment and therefore are called *external input messages*. Similarly output messages of exit nodes are called *external output messages*. Other messages inside the stream program are referred to as *intermediate messages*. An intermediate message can be input of a node and output of another one. An external input message can be an input of an entry node and can not be any node's output. Similarly, an external output message can only be output of an exit node, but not input of any other node.

Inside a stream program, messages are transferred from nodes to nodes via streams. A node gathers messages from a set of streams, called input streams. After processing these messages, the node produces new messages and scatters them to a set of streams, called output streams. Execution of a node consumes *n* input messages and produces *m* output messages, and *n-to-m* is the *multiplexity* of the node. This execution can occur only when *n* required input messages are available. As the node's behaviour can be context-dependent, the two values *n* and *m* can vary at runtime. Also *m* produced messages can be dynamically distributed to the output streams. In the Image Filter example, node *Splitter* can split one image into a number of sub-images depending on the image size. Depending on the implementation, this node can have a policy how to scatter these sub-images into different branches. For example, one can choose to scatter with a round-robin manner.

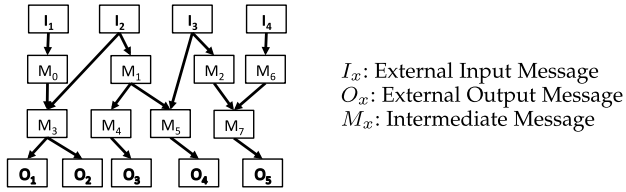


Fig. 2. An example of message derivation.

3.3 Execution Model

Conceptually the execution model of stream programs includes three layers: a compiler, a runtime system (RTS) and a scheduler. A stream program's source code first is passed through the compiler to generate object code. The RTS uses the object code to produce runtime objects including streams and tasks. Each task represents an instance of a node. The node's input streams now become the task's input streams and similarly its output streams become the task's output streams.

A task is an iteration of the node invocations, each of which includes reading messages from its input streams, processing them, and writing output messages to its output streams. Streams are implemented as FIFO buffers for the transfer of messages from tasks to tasks. We define the *stream transfer* of a message M as the activity of moving M across a stream from one task to another. The activity of a stream S therefore consists of stream transfer of all messages passing over S. Usually, on shared memory platforms, message transfer is implemented simply by memory access, while on distributed platforms it is implemented by using message passing techniques, for example MPI.

The graph of nodes now becomes the graph of tasks. Tasks associated with entry nodes are called *entry tasks*. Tasks associated with exit nodes are called *exit tasks*. Other tasks are called *middle tasks*.

The RTS also controls the state of tasks, i.e., when a task is ready to be scheduled. A task is ready to be executed if all required messages are available on their input streams. In this case, the task state is *ready*. Otherwise the task state is *blocked*.

Lying under the RTS, the scheduler employs a policy to execute ready tasks on a target platform consisting of processing elements. The scheduler's policy decides:

- which ready task will be executed
- which PE will perform the ready task
- the length of scheduling cycle, i.e., the period for which the PE will perform the ready task

3.4 Message Derivation

When a node invocation consumes an input message M_x (and possibly other messages) to produce an output message M_y (and possibly other messages), it is said that M_x derives M_y, or M_y is derived from M_x, formally written as $\mathrm{Drv}(M_x, M_y)$. In this case, M_x is a *predecessor* of M_y and M_y is a *successor* of M_x. The message derivation relation is transitive, i.e., $\mathrm{Drv}(M_x, M_y) \wedge \mathrm{Drv}(M_y, M_z) \Rightarrow Drv(M_x, M_z)$.

Fig. 2 shows an example of message derivation from external input messages to external output messages. In this example, we have from external input I_2 towards external output O_3 the derivations $\mathrm{Drv}(I_2, M_1)$, $\mathrm{Drv}(M_1, M_4)$ and

$\mathrm{Drv}(M_4, O_3)$. We also have $\mathrm{Drv}(M_1, M_5)$, $\mathrm{Drv}(I_3, M_5)$, and $\mathrm{Drv}(M_5, O_4)$, etc.

Given an external input message I, its derived external output message $derived\_outputs(I)$ is defined as the set of external output messages that are successors of I.

3.5 Message Completion

When processed by a stream program, an external input message I_i may derive multiple intermediate messages M_j before deriving any external output messages O_k, written as $\mathrm{Drv}(I_i, M_j)$, $\mathrm{Drv}(M_j, O_k)$ for each such M_j. An external input message I_i is said to be *completed*, i.e., completely processed, when none of its successor exists inside the stream program, i.e., all messages in $derived\_outputs(I_i)$ have been sent out. In the example of Fig. 2, the external input message I_2 is completed when the messages O_1, O_2, O_3, O_4 are all sent out.

The *completion of an external message* is defined as a set of node invocations that produce all its successor messages and the stream transfers of these messages. Put another way, the completion of an external message I_i is the process of performing node invocations to generate its successors, and stream transfers passing them to other tasks until all messages in $derived\_output(I_i)$ are sent out. In the example of Fig. 2, the completion of I_1 composes of three node invocations that produce M_0, M_2, O_1 and O_2; and two stream transfers that shift M_0 and M_2 to the corresponding tasks.

As mentioned above, a task executes a sequence of node invocations. The contribution of a task T to an external input message I_i is defined as the group of T's node invocations that belong to the completion of I_i. Similarly, contribution of a stream S to an external input message I_i is the group of S's stream transfers that belong to the completion of I_i.

Feedback loops. It is common that a stream program contains feedback loops, for example, to continuously maintain a program state. For each external input message, some of its successors will remain inside the stream program to form the program state that influences the processing of the next external input message.

For the extreme case where such successors remain with unbounded time in the system, the message derivation mechanism as presented above would lead to infinite completion time of external input messages. As irreducible loops occur rarely, we can assume that all feedback loops in a stream program are reducible [2]. In case of reducible loops, the back edge of a loop can be identified as a sequence of streams that connects a node n_1 to a node n_2 dominating n_1. A node n_{dom} is said to dominate a node n_i if all data flow has to go through n_{dom} in order to reach n_i. To avoid infinite completion times, all messages which are sent over back edges will be ignored by the message derivation relation used to calculate a message's completion time. This is a pragmatic approach that makes sense, since such unbounded feedback loops effectively model persistent state rather then the processing of any individual input message.

However, feedback loops where all successor messages remain only a finite time in the system are not a problem, as there the message completion times can be calculated by virtually unrolling the feedback loop.

4 Throughput of Stream Programs

Stream programs are similar to communication networks in the sense that they transfer messages from one end to another via interconnected nodes. Therefore, the throughput is one of the main metrics to evaluate stream programs. In similarity with communication networks, the throughput of stream programs is measured as the number of external input messages that are completed per time unit. In contrast to communication networks where each node has its own physical resource, tasks in a stream program perform their computations on a shared platform. Thus the throughput of stream programs depends highly on the scheduling policy.

4.1 Uniform Shared Memory Platforms

In our previous work [34], we have derived the throughput of stream programs on uniform shared memory platforms as follows:

$$\mathrm{TP} = \frac{(N - \widetilde{W} - \widetilde{O})}{\overline{C}}. \tag{1}$$

Where N is the number of homogeneous CPU cores on the shared memory platform, $\widetilde{W}$ is the relative idling of the system, $\widetilde{O}$ is the relative overhead of the system, and $\overline{C}$ is the average computational time required to complete one external input message.

4.2 Distributed Platforms

On shared memory platforms, the communication cost between nodes is negligible. The centralised scheduling approach is therefore quite efficient. When deploying stream programs on distributed platforms, the communication cost becomes significant and the central approach is no longer efficient.

In this section, we focus on deploying stream programs on distribute platforms where each PE is a uniform shared memory machine. An intuitive approach would be to divide the set of tasks of the stream program into multiple subsets, and assign them to separate PEs. Each PE has its own local scheduler for their assigned tasks. This approach only works for static stream programs with fixed structures during runtime. For dynamic stream programs, statistical observation of the structure changes can be used to stabilise the stream program before applying this approach. In this section, we present the quantitative evaluation of the throughput for distributed platforms.

A distributed platform is represented as an undirected graph $H = (R, L)$ where R is the set of vertices, each of which represents for a PE; and L is the set of edges, each of which is the communication link between PEs. Each PE, r, has weight $w(r)$, which equals to its number of cores. A communication link between PE r_i and r_j is denoted as $l_{r_i r_j}$ and has weight $w(l_{r_i r_j})$ which is equal to its bandwidth in Byte/s.

As presented in Section 3, each task represents an instance of a node and tasks communicate with each other via uni-directional streams. The stream program therefore can be represented as a directed graph $G = (T, S)$ where T is the set of tasks, and S is the set of directed streams.

Though a stream program G is directed, for the sake of throughput optimisation later described in this article, we will model as undirected graphs, based on the realistic assumption that communication cost is the same regardless of the direction of the stream.

The weight of each task t in T is defined as the average time that t requires to perform its processing contribution to an external input message. This amount of time depends on the PE that the task is mapped to. The weight of task t on PE r is notated as $w^r(t)$ representing the execution time in seconds. A stream connecting tasks t_i and t_j is denoted as $s_{t_it_j}$. Each stream has weight $w(s_{t_it_j})$, which is equal to the average amount of data to be transferred over $s_{t_it_j}$ for the completion of an external input message. This weight $w(s_{t_it_j})$ is the average size of all messages emerged in the contribution of $s_{t_it_j}$ for the completion of an external input message. The unit of $w(s_{t_it_j})$ is Byte/message. The values of $w^r(t)$ and $w(s_{t_it_j})$ can be easily obtained using an appropriate monitoring framework, for example the one we have developed for that purpose [35].

A mapping configuration of a stream program $G = (T, S)$ over a distributed platform $H = (R, L)$ is defined as group of partitions $\mathrm{MpC} = \{P_r \mid i = 1..|R|\}$. A partition P_r is the set of tasks that are mapped to the PE r.

For shared memory platforms we can ignore cost of message transfer, consequently the completion of an external input message is formed by a set of node invocations spread over the contributions of all tasks. On distributed platforms, the cost to transfer messages over a stream within a partition can still be considered to be minor. In contrast, message transfer over a stream across two partitions is costly. The completion of an external input message spreads over both tasks on partitions and the stream communication among them.

A partition P_r is considered to have *completed* its contribution to an external message I_x when its tasks have completed their contributions to I_x. The communication weight between two partitions P_{r_i} and P_{r_j} is defined as the total weight of all streams across them. This represents the average amount of data to be transferred between two partitions P_{r_i} and P_{r_j} within the completion of an external message,

$$\mathrm{Comm}(P_{r_i}, P_{r_j}) = \sum_{t_i \in P_{r_i}, t_j \in P_{r_j}} w(s_{t_it_j}). \tag{2}$$

Since completion of external input messages is stretched over all partitions $P_r \in MpC$ and the communications among them, the throughput of a stream program is determined based on two kinds of throughput: the computation and communication throughputs.

The computation throughput of a partition P_r is the average number of external input messages that P_r completes within a time unit. Since each PE r is a shared memory platform, Equation (1) can be used to derive the throughput of partition P_r as follows:

$$\mathrm{TP}_{comp}(P_r) = \frac{w(r) - \widetilde{W_r} - \widetilde{O_r}}{\overline{C_r}} = \frac{w(r) - \widetilde{W_r} - \widetilde{O_r}}{\sum_{t \in P_r} w^r(t)}, \tag{3}$$

where $\overline{C_r}$ is the average time that all tasks in the partition contribute to completely processing an external message. This equals the total weight of all tasks in the partition. $\widetilde{W_r}$ and $\widetilde{O_r}$ are the relative idling time and the relative overhead of the local scheduler of PE r, respectively.

The communication throughput between the two partitions P_{r_i} and Pr_j is amount of their communication weight that can be transferred via the physical link between PEs r_i and r_j within a time unit.

The physical link connecting two PEs is undirected, i.e., the bandwidth consumption of any data transfer is independent of its direction. Thus, when modelling the communication cost for throughput optimisation, we can abstract away from the directiveness of G.

The communication throughput is determined by dividing the bandwidth between PEs r_i and r_j by the communication weight between two partitions:

$$\mathrm{TP}_{comm}(P_{r_i}, P_{r_j}) = \frac{w(l_{r_ir_j})}{Comm(P_{r_i}, P_{r_j})}. \tag{4}$$

The communication weight $Comm(P_{r_i}, P_{r_j})$ is the amount of data that needs to be transferred between the two partitions P_{r_i} and P_{r_j} for the completion of an external input message. The communication throughput therefore can be considered as the number of external messages that can be completed by the communication link.

Since the computation of partitions and communication among them can occur in parallel, the throughput of the stream program with a mapping configuration MpC is intuitively the minimum of all computation throughputs and communication throughputs:

$$\mathrm{TP}(\mathrm{MpC}) = \min_{r_i, r_j, r_k \in R} (\mathrm{TP}_{comp}(P_{r_i}), \mathrm{TP}_{comm}(P_{r_j}, P_{r_k})). \tag{5}$$

5 Partitioning Algorithms to Optimise Throughput

The previous section presented a method to calculate the throughput of stream programs when deployed on a distributed platform with a mapping configuration. In this section, we focus on maximising the throughput. We assume that the local scheduler of each PE has predictable relative idling and overhead times. We also assume that these parameters are not affected by the partitioning method.

We introduce two algorithms to generate a mapping configuration so that its throughput is maximised. The first algorithm, called KL-Adapted, is a trivial adaptation from the well-known graph partitioning algorithm, Kernighan-Lin [24]. The second one, called Congestion Avoidance, operates in a similar way but instead of considering all possible move operations, the method detects the congestion point and examines only move operations that can help to improve the congestion point.

For convenience, we notate by $par(\mathrm{MpC}, t)$ the partition in MpC that task t belongs to.

5.1 KL-Adapted Algorithm

The original Kernighan-Lin algorithm [24] aims to divide a graph into two partitions such that they are balanced in terms of the number of vertices with minimum number of

edges across them. The algorithm introduces the gain of a vertex as the total of edge cut which will be decreased if the vertex is moved to the complimentary partition. The gain of each vertex is calculated based on internal edges connecting the vertex with vertices on the same partition, and external edges connecting the vertex with vertices on the different partition.

Starting with a randomly generated partition, the algorithm uses a greedy heuristic to find a sequence of locally optimal operations between two partitions which maximise the improvements. Each operation includes choosing a vertex from the first partition with the maximum gain to move to the second partition, and similarly choosing a vertex from the second partition with the maximum gain to move to the first partition. After each move, the gain of all vertices are updated locally by examined the moved vertex and its neighbours.

Fiduccia and Mattheyses algorithm [15] is an improvement of Kernighan-Lin by using an appropriate data structure. When gain values are integer and fall in a bounded range, a set of buckets can be used to store vertices. Each bucket is labelled with a value in the gaining range. Each vertex is stored in one bucket with the label matching with the vertex's gain value. This helps to choose the best vertex to move, that is one of the vertices in the bucket with the largest label. When the gain of a vertex changes, the vertex is moved to the new bucket according to its new gain.

We adapted the conception of the Kernighan-Lin algorithm in [24] to the multiple-way partitioning approach where each operation is to move one task to a new partition. Our new algorithm is called the KL-Adapted Partitioning Algorithm. In similarity to the original Kernighan-Lin algorithm, KLA starts with a randomly generated initial mapping configuration and applies greedy heuristic passes iteratively until the throughput stops increasing. We denote these passes as *KLA passes* to distinguish them from passes in the CA algorithm presented later. Each KLA pass searches for the best move operation which relocates a task to a new partition so that the throughput after the move operation is maximised. After being relocated, a task is locked so that it is moved only once during a KLA pass. This process is carried on until all tasks have been moved. At the end of the KLA pass, the sequence of move operations that creates the new mapping configuration with the highest throughput are chosen to be applied.

The original KL/FM algorithm keeps track of the gain of each vertex so that it is easy and quick to choose the best vertex to move. This is feasible since after moving a vertex, only the gain of its neighbours needs to be updated by simple arithmetic operations. In our problem, moving a vertex may cause changes of the gain values of all vertices and therefore tracking the gain value of each vertex is not beneficial. We instead keep track of the computation throughput of each partition and the communication throughput between each pair of partitions. We use a 1D-array to store computation throughputs and a 2D-array to store communication throughputs. The KLA pass in our algorithm needs to examine all possible move operations to find the best move operation. When trying a move operation, the computation and communication throughputs of involved partitions will be updated. The throughput of the new mapping configuration then is calculate as the minimum value of all computation and communication throughputs.

The details KLA are shown in Fig. 3, labelled as steps A to J. The input to the KLA algorithm includes the stream program $G = (T, S)$, and the distributed platform $H = (R, L)$ (step A). At the beginning, a randomly generated mapping configuration is generated and its throughput is evaluated as in step B. The KLA algorithm uses three lists $moved\_task$, $moved\_par$ and $moved\_tp$ to store the history of move operations. Starting with the empty history, each KLA pass operates on a temporary mapping configuration tmp_map which is a copy of the current mapping configuration cur_map (step C). The KLA pass then finds the task t not in $moved\_task$ and the partition P so that moving t to P carries out the highest throughput compared to all other possible move operations. If the move operation exists (step E with 'yes'), it is applied to generate a new mapping configuration. The move operation is also added to the history, i.e., t is appended to $moved\_task$, P is appended to $moved\_par$, and the new throughput value is added to $moved\_tp$ (step F). Note that some move operations in the history can reduce the throughput, i.e., an element in $moved\_tp$ can have smaller value than others before it. To avoid the case where a task is repeatedly exchanged between two partitions, one task is moved at most once within a pass. That means tasks in $moved\_task$ are not considered to move again within the current KLA pass. When all tasks have been moved once, i.e., no move operation can be found (step E with 'no'), the list $moved\_tp$ is examined to find the maximum throughput value max_tp and its index k (step G). The current KLA pass terminates and a new pass will be proceeded if max_tp is higher than the throughput of the current mapping configuration (step H with 'yes'). Before starting a new KLA pass, the current mapping configuration cur_map needs to be updated by applying all move operations in the history up to index k (step I). In the case where max_tp is not higher than the throughput of the cur_map (step H with 'no'), that means the heuristic KLA pass can not find any better mapping configuration. The KLA algorithm therefore returns cur_map as the final mapping configuration (step J) and terminates.

5.2 Congestion Avoidance Partitioning Algorithm

In similarity to KLA, the Congestion Avoidance partitioning algorithm begins with an initial mapping configuration and repeats a heuristic pass until the throughput reaches a locally optimal value. We denote the heuristic pass here as *CA pass*. Unlike a KLA pass, a CA pass does not examine all possible move operations, but focuses on only those around the congestion point identified by inspecting the throughput formula of Equation (5). Within each pass, the CA identifies the congestion point of the current mapping configuration and tries move operations that potentially improve the throughput.

From Equation (5), the throughput of a stream program with a mapping configuration MpC is the minimum value of a set of computation and communication throughputs. A congestion point is where the throughput is settled, i.e., where the minimum value occurs. If the minimum value is $\mathrm{TP}_{comp}(P_r)$, the congestion is said to lie on the computation

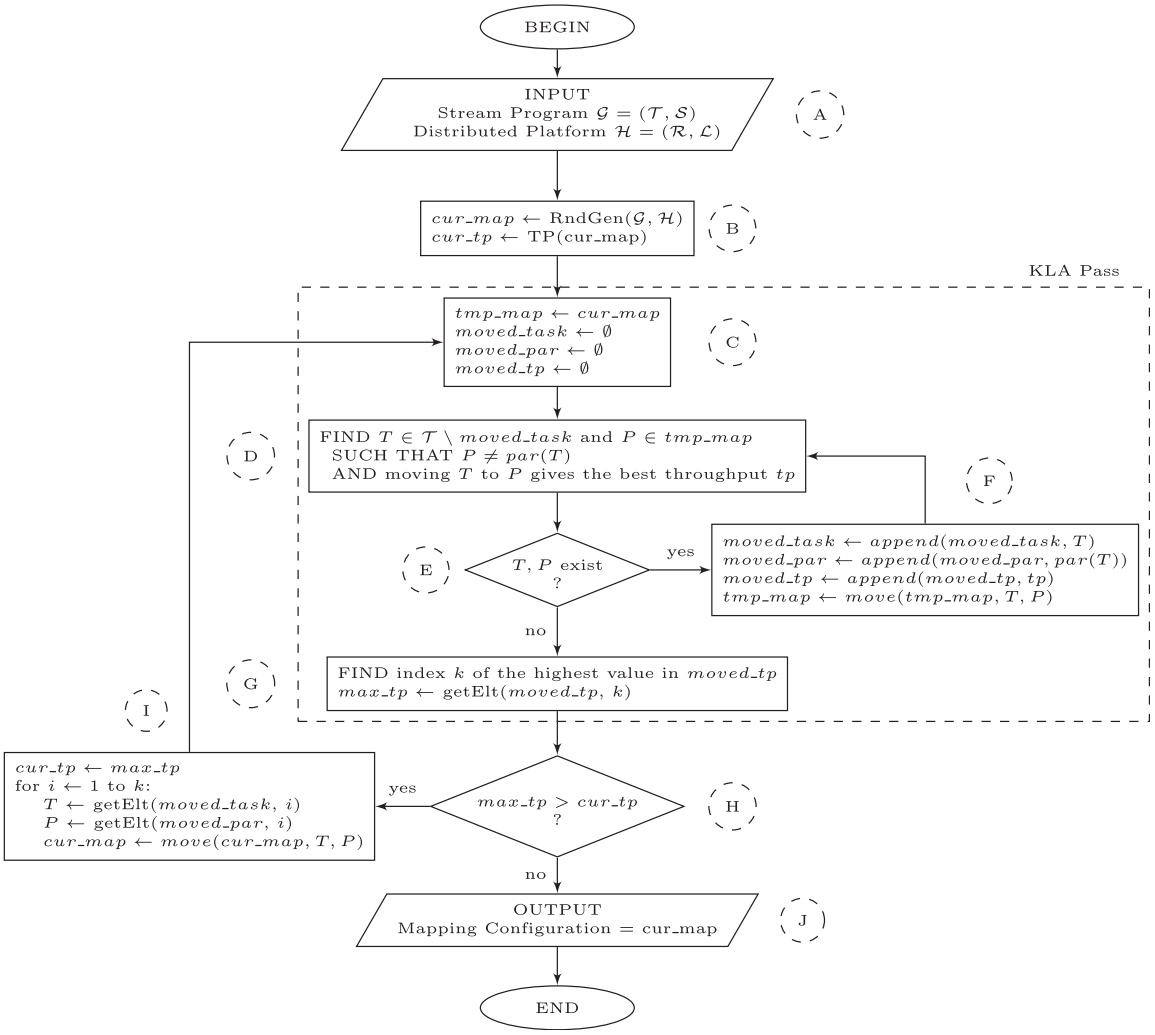


Fig. 3. Flowchart of KL-adapted partitioning algorithm.

of P_r. In this case, only tasks from P_r are considered to be moved to other partitions. This helps to reduce $\sum_{t \in P_r} w_r^t$ and therefore increase $\mathrm{TP}_{comp}(P_r)$. Thus the throughput $\mathrm{TP(MpC)}$ then improves.

Similarly if $\mathrm{TP}_{comm}(P_{r_i}, P_{r_j})$ is the minimum value, the congestion lies on the communication between partitions P_{r_i} and P_{r_j}. In this case, only move operations that reduce the communication weight between P_{r_i} to P_{r_j} are considered. Those are move operations involving tasks which have a stream connection across P_{r_i} and P_{r_j}. Relocating these tasks potentially reduces the communication weight between P_{r_i} and P_{r_j} and therefore potentially improves $\mathrm{TP}_{comm}(P_{r_i}, P_{r_j})$.

The details of the CA algorithm are shown in Fig. 4, labelled as steps A to N. Taking a stream program $G = (T, S)$ and a distributed platform $H = (R, L)$ as inputs (step A), CA starts by generating a randomly generated mapping configuration cur_map (step B) before it gets into heuristic passes. In similarity to KLA, CA stores the history of move operations in three lists $moved\_task$, $moved\_par$ and $moved\_tp$. Each CA pass also starts with an empty history and a temporary mapping configuration tmp_map which is copied from the current mapping configuration cur_map (step C). By evaluating the throughput formula (Equation (5)) on tmp_map, the CA pass identifies the congestion point. The type of congestion point is used to determine the set of tasks T_{try} so that reallocating them can potentially improve the congestion point (steps E, F and G). If the congestion point lies on the computation of P_r, T_{try} consists of tasks in P_r. If the congestion point lies on the communication of P_{r_i} and P_{r_j}, T_{try} includes pairs of tasks, one in P_{r_i} and one in P_{r_j}, which are connected by a stream. In step H, the algorithm examines move operations of re-allocating tasks in T_{try} which have not been moved during the current pass, i.e., not in the list $moved\_task$. The result of this step is the move operation of a task t to a another partition P that brings the highest throughput compared to other examined move operations. The remaining steps of the CA pass are similar as in a KLA pass. The move operation is applied and added to the history if it exists (step J). Otherwise, the algorithm scans the history of move operations to find the highest throughput value max_tp and its index k (step K). The algorithm decides to update cur_map and continues a new CA pass if max_tp is better than the throughput of cur_map (step M). If not, the algorithm terminates with cur_map as the output.

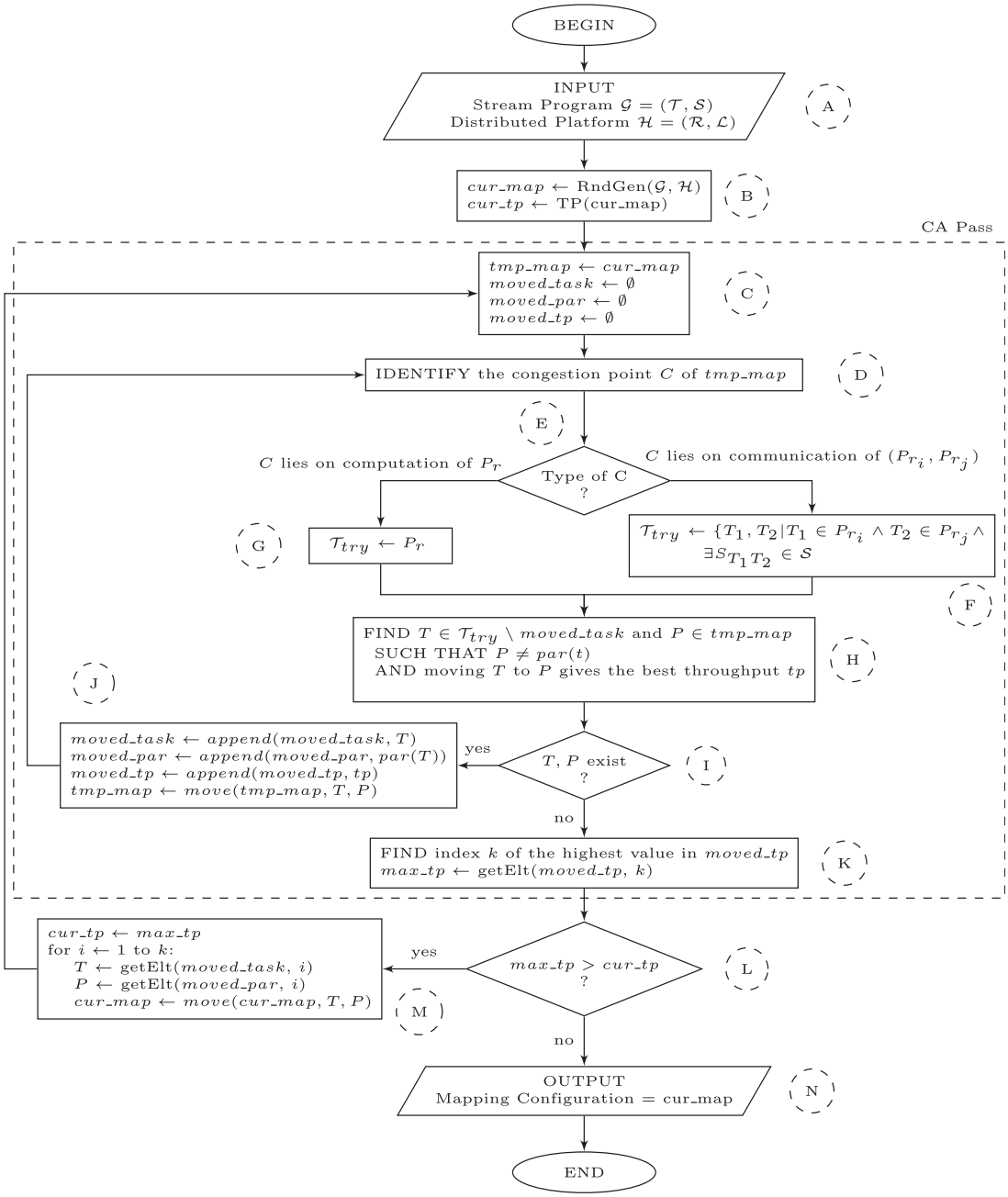


Fig. 4. Flowchart of congestion avoidance partitioning algorithm.

5.3 Local Optima in Heuristic Search

Since the proposed algorithms KLA and CA are local search heuristics, they can be trapped in local optima. There are a number of approaches to overcome this problem. One simple way is to run the partitioning algorithm multiple times with different initial mapping configurations. Another more complicated way is to integrate the local search heuristic into multilevel schemes. This way has been shown have been shown to be successful in overcoming the localized nature of KL/FM [23], [21]. In this work, we focus on the effectiveness of our new local search heuristic. We therefore choose to perform our algorithms multiple times with different initial mapping configurations. Integrating our approach into multilevel schemes will be considered in the future work.

6 Evaluation of The Partitioning Algorithms

In this section we evaluate the performance and efficiency of the two heuristic algorithms KLA and CA presented in Section 5. The implementation of these algorithms can be found in [1]. As discussed in Section 2, most of existing graph partitioning tools first obtained the load balance criterion and then use the total cut criterion as the optimised target. To show that these two criteria are not efficient in the domain of stream programs, we compare the CA algorithm with the existing tool METIS partitioner [31].

To the best of our knowledge, we are the first to use the throughput formula (Equation (5)) as the optimised target in graph partitioning. Despite using another version of this

formula, the approach in [8] does not consider congestion points lying on the communications but attempts to eliminate only those lying on the computations. Furthermore, this approach integrates this formula with other refinement methods for optimising other metrics in embedded systems. We therefore compare our proposed heuristics with Simulated Annealing as it is a generic technique of finding the global optima of a specific function.

As discussed in Section 5.3, CA and KLA will be run multiple times to overcome local optimum. When comparing with SA and METIS partitioner, we use 50 runs of CA and KLA.

6.1 Experimental Setup

We perform our experiments with S-Net stream programs [19] with its LPEL execution layer [40]. One convenience of using this execution model is that it provides an essential monitoring framework to obtain the required information for the stream program graph [35]. Another reason is that LPEL provides a local scheduler for each PE where the time overhead is predictable, and the waiting time is negligible when the system is fully working (i.e., the arrival rate of external input messages is high enough to keep the resources busy) [34]. The relative overhead $\widetilde{W}$ is one CPU core which is used as the conductor to manage the central task queue.

We use five different applications:

- DES: performs DES encryption
- OBD: detects four different types of objects from images
- HIST: calculates histogram of images
- MTI: detects moving objects on the ground from an aircraft [37]
- S500: synthetic stream graph with approximately 500 vertices
- S1,000: synthetic stream graph with approximately 1,000 vertices

Implemented in S-Net, each of the first four applications contains a primary stream structure that performs the application's main function. To increase the level of concurrency, S-Net provides a replication operation, called parallel replication, to create multiple instances of the primary stream structure. The number of instances is decided by the number of machines of the deployed targets. Therefore, each application when deployed on different targets will have a different number of nodes and streams. To diversify the experiment, we use the stream graph generator described in [3] to create two synthetic stream graphs with approximately 500 and 1,000 vertices. They are called S500 and S1,000, respectively. The generator makes sure these two synthetic graphs have key properties shared by most of stream applications. To partition graphs with 5,000 or more vertices, SA and KLA take too long to run. We therefore do not include such large benchmarks here.

The experimental targets are clusters of 4, 8 and 16 machines. Each machine has two sockets with Xeons E5520 CPUs. Each machine also has 24 GB of shared memory. The machines are connected via a 4xDDR Infiniband where the traffic between pairs of machines are guaranteed for a full bandwidth of 16 Gbits/s. For convenience, TARGET_i is used to denote the target with i machines. Note that although each machine has eight cores, only seven cores are used for the computation as one core is used as the conductor. Also, for each target, one of its machines needs to reserve two cores to simulate the source and sink for stream programs. The source is a process that continuously sends external input messages to the stream program while the sink continuously consumes its external output messages. We also include one *synthetic target* with 16 machines. The number of cores on each machine is chosen randomly from 1 to 7. The bandwidth of their connections is assigned arbitrarily from 0 to 16 Gbits/s. This synthetic target is denoted as TARGET_{S16}.

We use $A_i \mid A \in \{DES, HIS, OBD, MTI, S500, S1{,}000\}$ to denote for the benchmark of application A deployed on target TARGET_i.

Table 1 shows the number of vertices and edges for all benchmarks.

TABLE 1
Properties of Evaluation Benchmarks

Benchmark	# Vertices/Nodes	# Edges/Streams
DES_4	74	76
OBD_4	98	144
HIST_4	122	180
MTI_4	126	140
DES_8	146	152
OBD_8	194	288
HIST_8	242	360
MTI_8	250	280
$\mathrm{DES}_{16}, \mathrm{DES}_{S16}$	290	304
$\mathrm{OBD}_{16}, \mathrm{OBD}_{S16}$	386	576
$\mathrm{HIST}_{16}, \mathrm{HIST}_{S16}$	482	720
$\mathrm{MTI}_{16}, \mathrm{MTI}_{S16}$	498	560
$\mathrm{S500}_{16}, \mathrm{S500}_{S16}$	504	541
$\mathrm{S1000}_{16}, \mathrm{S1000}_{S16}$	992	1,067

6.2 METIS Partitioner for Stream Programs

We perform an experiment to show the efficiency of the METIS partitioner [31], an existing partitioning tool that use the balance criterion and total cut criterion as the objective function. In this experiment, we use the METIS partitioner version 5.1.0 with the default parameters. With these parameters, the METIS partitioner applies direct k-way partitioning to minimise the total edge cut, and uses the sorted heavy-edge matching scheme during the coarsening phase.

Fig. 5 shows the comparison in quality of outcome mapping configurations between our CA algorithm and the METIS partitioner. In all benchmarks except for DES_{S16} and $S10{,}000_{S16}$, CA produces mapping configurations with throughput at least 10 percent better than the METIS partitioner. For these two case, CA's output is around 2 percent better than the METIS partitioner's. The mapping configurations produced by CA are superior for OBD_{S16}, $HIST_{S16}$, and MTI_{S16}. In particular their throughputs are 755, 1,172, and 158 percent higher than the throughputs of mapping configurations produced by the METIS partitioner.

We do not focus on the comparison of the execution time here as the METIS partitioner uses a different objective function which aims to minimise the total cut. This objective function is simpler to compute compared to the throughput function in our partitioning algorithms. In addition, METIS

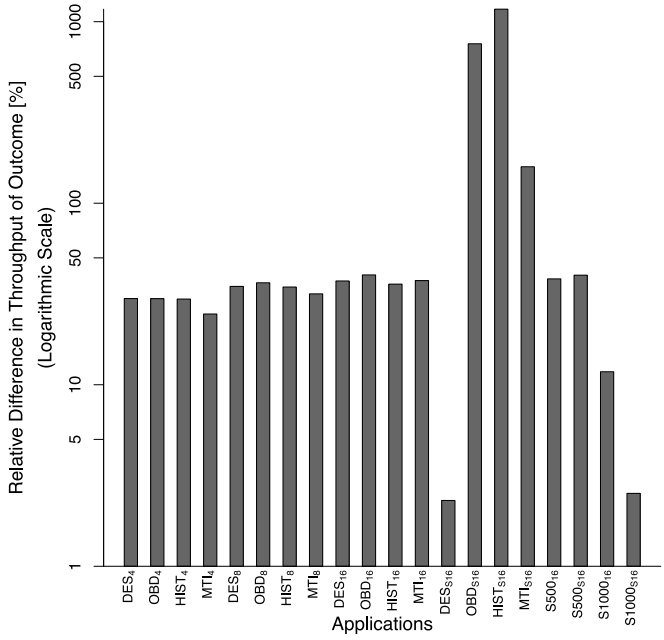


Fig. 5. Outcome quality of 50-run CA compared to METIS partitioner.

uses the multilevel partitioning scheme which is considerably faster than the direct partitioning scheme used in our algorithms. For these above reasons, the METIS partitioner takes significantly less time than the CA algorithm in all the benchmarks. For future work, we will consider integrating the CA algorithm into a multilevel partitioning scheme.

6.3 Convergence Speed of KLA and CA

In this section we evaluate the convergence in terms of the number of passes and the execution time of two proposed algorithms, KLA and CA. Fig. 6 shows the average number of passes of these two algorithms. The average value is calculated based on 100 runs. For all the evaluation benchmarks, both of the algorithms require a small number of passes with the highest value being 5. Comparing between KLA and CA, it is not clear which one is better in terms of number of passes. However, the difference is not significant, it is less than 1 for most of benchmarks.

TABLE 2
Execution Time (in Seconds) of SA

Benchmark	Execution time	Benchmark	Execution time
DES_4	1,452.40	OBD_4	159.43
$HIST_4$	1,870.27	MTI_4	3,902.52
DES_8	2,538.36	OBD_8	2,845.84
$HIST_8$	703.64	MTI_8	756.67
DES_{16}	594.17	OBD_{16}	1,087.71
$HIST_{16}$	1,644.43	MTI_{16}	1,788.80
DES_{S16}	349.24	OBD_{S16}	844.49
HIS_{S16}	1,555.94	MTI_{S16}	1,742.06
$S500_{16}$	1,823.34	$S1000_{16}$	13,945.55
$S500_{S16}$	1,825.83	$S1000_{S16}$	13,912.19

As presented in Section 5, each CA pass considers only move operations around the congestion point while KLA examines all possible move operations. The execution time of CA is therefore significantly less than KLA although they take similar numbers of passes. Table 2 shows the execution time of SA and Fig. 7 shows the execution time ratio of KLA to CA compared to SA. In general, KLA is significant slower than CA and it seems to be the trend that this ratio increases for benchmarks with larger numbers of vertices and edges. For example, KLA is 3 to 6 times slower for small benchmarks like DES_4, OBD_4, $HIST_4$ and MTI_4. KLA is 14 to 22 times slower for DES_8, OBD_8, $HIST_8$ and MTI_8. For very large benchmarks such as DES_{16}, OBD_{16}, $HIST_{16}$, MTI_{16}, $S500_{16}$, and $S1000_{16}$, KLA takes 62 to 335 times longer to execute than CA.

6.4 Comparison with SA

In this section, we compare our heuristic algorithms with the generic global search algorithm, Simulated Annealing [26].

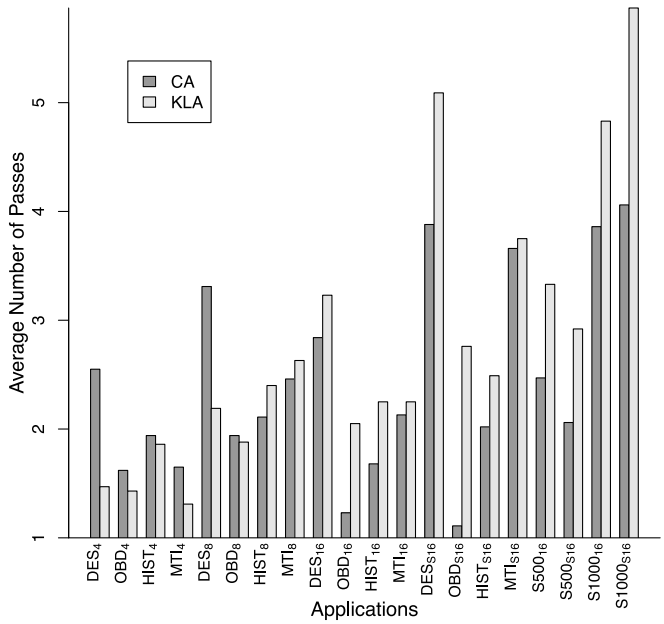


Fig. 6. Average number of passes in KLA and CA.

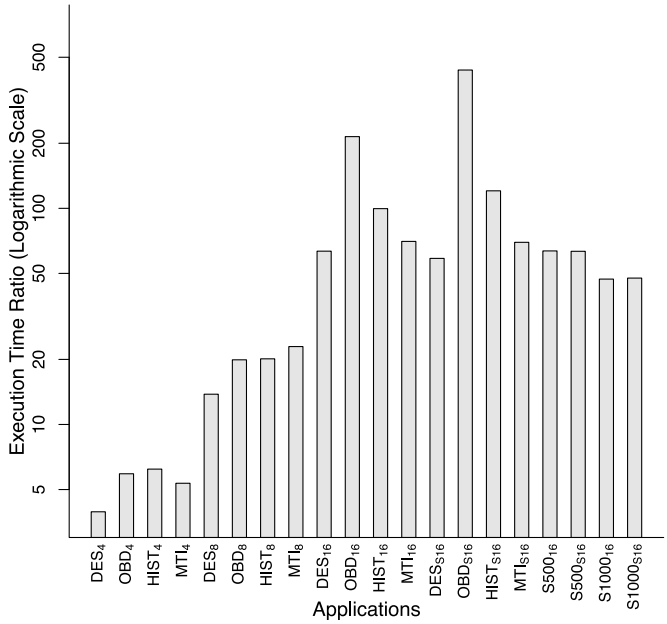


Fig. 7. Execution time ratio: et_{KLA}/et_{CA}.

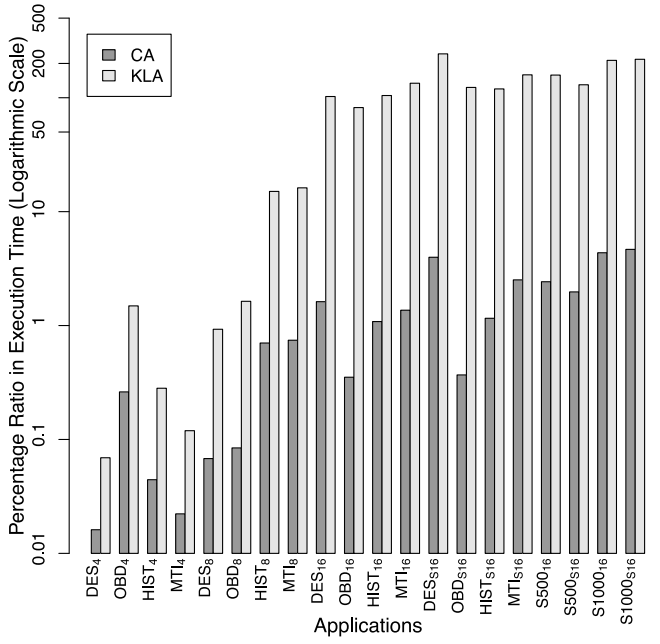


Fig. 8. Percentage ratio in execution time ratio of 50-run CA and KLA to SA ($et_{50\times CA} \times 100\%/et_{SA}$; $et_{50\times KLA} \times 100\%/et_{SA}$).

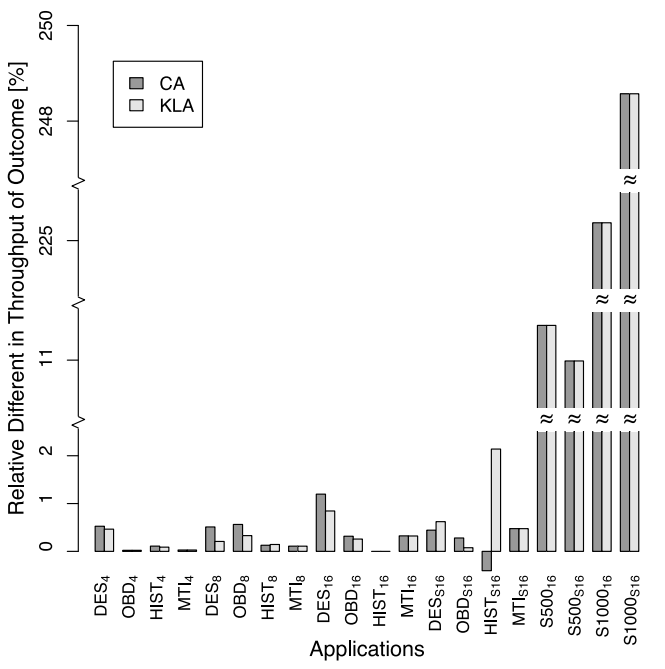
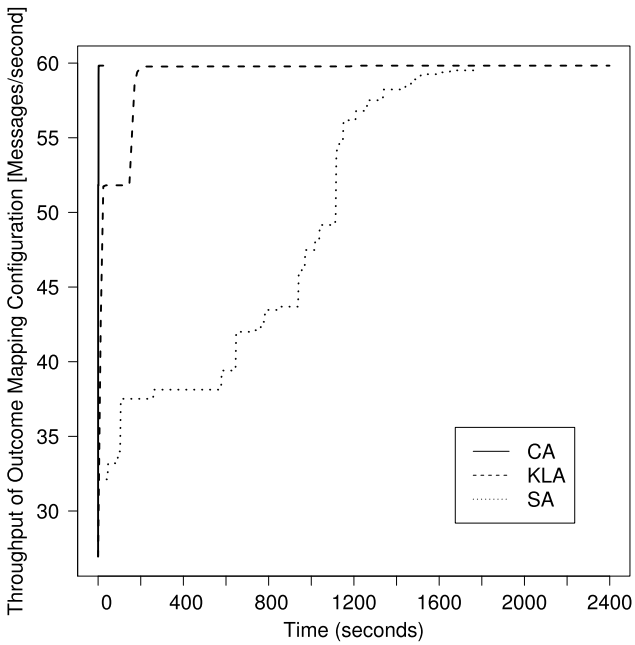


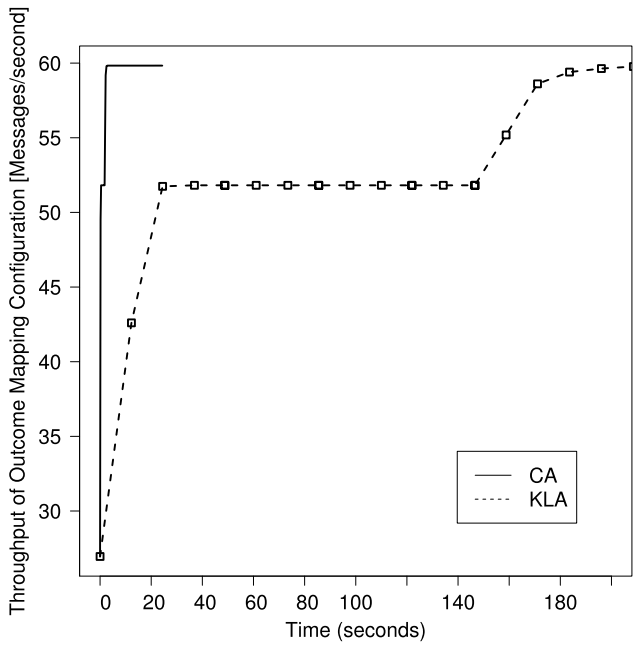
Fig. 9. Outcome Quality of 50-run CA and KLA Compared to SA.

The parameters SA are chosen so that good enough results can be reached within feasible time. By trying multiple combinations of the parameters, we found the following parameters where SA behaves well for all non-synthetic benchmarks: initial temperature $T_{init} = 109.00$, minimum/cooling temperature $T_{min} = 0.02$, cooling ratio $r_t = 0.985$, temperature length $L = 0.1 \times neighbour\_size$. Note that the temperature length is the number of iterations at each temperature. $neighbour\_size$ is the number of neighbours of each state and it is also the number of new mapping configurations that can be generated by relocating one task to a new partition: $neighbour\_size = |T|^{|R|-1}$.

Figs. 8 and 9 show the comparison in terms of execution time and quality of the outcome mapping configuration. For benchmarks small number of edges and vertices, 50 runs of KLA is always faster than SA. For example, to partition DES_4 the 50 runs of KLA take around 0.06 percent of SA's execution time. The ratio is 1.4, 0.2 and 0.11 percent for HIS_4, OBD_4 and MTI_4 respectively. When partitioning benchmarks with large number of edges and vertices, 50 runs of KLA can take more time than SA. For example, it takes 242, 158 and 217 of the execution time of SA to partition MTI_{16}, $HIS_{S16}$and $S1000_{S16}$.



(a) Convergence of SA, CA and KLA



(b) Convergence of CA and KLA (zoom of Figure 10a)

Fig. 10. Convergence of SA, CA and KLA on MTI_{16} over time.

In contrast, the 50 runs of CA are always significantly faster than SA. It takes less than 5 percent execution time of SA for all benchmarks.

For graphs with 5,000 or more vertices, it takes both CA and KLA more than 100 hours to run. For the reference purpose, we include here the execution time of 50 runs of CA on larger graphs. It takes 559.539996 and 4371.744931 seconds for 50 runs of CA on a 5,000-vertex graph and 10,000-vertex graph, respectively.

For the quality of outcome mapping configurations, all three partitioning algorithms provide similar results for the non-synthetic benchmarks. The difference in throughput of these mapping configurations is less than 1 percent, except for DES_{16} where CA's result is 1.19 percent better than SA. For synthetic benchmarks $S500_{16}$ and $S500_{S16}$, CA and KLA provide mapping configurations with throughput 11.72 and 10.98 percent higher than SA's outcome. For $S1000_{16}$ and $S1000_{S16}$, CA and KLA provide mapping configurations with throughput 225.37 and 248.57 percent higher than SA's outcome. Although one can search for a new set of parameters so that SA can provide comparable result to CA and KLA, this may cause worse behaviour or longer execution times for other benchmarks.

We also examine the convergence speed of all three partition algorithms. For all cases, CA is the fastest: it takes only a few milliseconds for small benchmarks like DES_4; and a few seconds for large ones like MTI_{16} and $S1000_{16}$. Though being a lot slower than CA, KLA still outperforms SA. For large benchmarks, KLA converges in a few hundreds seconds while SA takes thousands of seconds. Fig. 10a shows the convergence of these three algorithms on MTI_{16}. To better see the fast converge of CA, Fig. 10b shows the a zoomed version of Fig. 10a.

7 Conclusion

In this paper, we proposed two novel heuristic graph partitioning methods to partition the workload of stream programs to optimise throughput on heterogeneous distributed platforms. As explained in the article, traditional graph-partitioning problems with the optimisation criterion being formed by the total cuts, are not applicable to the throughput optimisation of stream programs.

The first graph-partitioning algorithm we developed is KLA, an adaptation of Kernighan-Lin. The second algorithm, called CA, narrows the search space compared to KLA, in particular by focusing on search points around the congestion, i.e., where the throughput is dimmed. Since KLA and CA are both local search heuristics, they have to be re-run multiple times in order to overcome local optima.

We experimentally evaluated KLA and CA with five applications on four different platform configurations. We compared both methods with the generic meta-heuristics *simulated annealing* as a reference method. Both KLA and CA achieve at least as good throughput results as SA, sometimes even better. For small benchmarks KLA with its multiple re-runs is up to more than 1,000 times faster than SA, but up to 2.5 times slower than SA for larger benchmarks. CA with its multiple re-runs on the other hand is always orders of magnitudes faster than both KLA and SA, even for large graphs. Depending on benchmark and platform, CA has been up to 335 times faster than KLA. The outstanding speed of CA makes it also potentially attractive for repartitioning of systems during runtime.

Multilevel graph partitioning schemes have the advantage of being fast and being able to overcome the local minima. We therefore also consider for the future work to integrate CA as the local search in the refinement phase in these schemes.

Acknowledgments

The research leading to these results has received funding from the IST FP7 research project "Asynchronous and Dynamic Virtualization through performance ANalysis to support Concurrency Engineering" (ADVANCE) under contract no IST-2010-248828 and the FP7 ARTEMIS-JU research project "ConstRaint and Application driven Framework for Tailoring Embedded Real-time Systems" (CRAFTERS) under contract no 295371. V. T. N. Nguyen is the corresponding author.

References

[1] The implementation of KLA and CA algorithms. [Online]. Available: https://github.com/nguyenvuthiennga/graph_partition, 2015.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Boston, MA, USA: Addison-Wesley, 1986.

[3] D. Ajwani, S. Ali, K. Katrinis, C. Li, A. J. Park, J. P. Morrison, and E. Schenfeld, "Generating synthetic task graphs for simulating stream computing systems," *J. Parallel Distrib. Comput.*, vol. 73, no. 10, pp. 1362–1374, 2013.

[4] C. J. Alpert, A. B. Kahng, and S.-Z. Yao, "Spectral partitioning with multiple eigenvectors," *Discrete Appl. Math.*, vol. 90, no. 13, pp. 3–26, 1999.

[5] T. Amnell, G. Behrmann, J. Bengtsson, P. DArgenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. Larsen, M. Mller, P. Pettersson, C. Weise, and W. Yi, "Uppaal-now, next, and future," in *Proc. 4th Summer School Model. Verification Parallel Proc.*, 2001, vol. 2067, pp. 99–124.

[6] S. Arora, S. Rao, and U. Vazirani, "Expander flows, geometric embeddings and graph partitioning," *J. ACM*, vol. 56, no. 2, pp. 5:1–5:37, Apr. 2009.

[7] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream computing on graphics hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, Aug. 2004.

[8] P. M. Carpenter, A. Ramirez, and E. Ayguade, "Mapping stream programs onto heterogeneous multiprocessor systems," in *Proc. Int. Conf. Compilers, Archit. Synthesis Embedded Syst.*, 2009, pp. 57–66.

[9] Ü. V. Çatalyürek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 10, no. 7, pp. 673–693, Jul. 1999.

[10] B. L. Chamberlain, "*Graph partitioning algorithms for distributing workloads of parallel computations*," Univ. of Washington, Seattle, WA, USATech. Rep. TR-98-10-03, 1998.

[11] W. Chen, "*Task partitioning and mapping algorithms for multi-core packet processing systems*," Master's Thesis, Dept. Electrical Comput. Eng., Univ. of Massachusettes-Amherst, Massachusettes, MA, USA, 2009.

[12] P. de Oliveira Castro, S. Louise, and D. Barthou, "Automatic mapping of stream programs on multicore architectures," in *Proc. Int. Workshop Compilers Parallel Comput.*, 2010, https://hal.archives-ouvertes.fr/hal-00551680/document

[13] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw, "Aspect radio for mesh partitioning," in *Euro-Par*, 1998, pp. 347–351.

[14] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw, "Shape-optimized mesh partitioning and load balancing for parallel adaptive fem," *Parallel Comput.*, vol. 26, no. 12, pp. 1555–1581, Nov. 2000.

[15] C. M. Fiduccia and R. M. Mattheyses, "A linear-time heuristic for improving network partitions," in *Proc. 19th Des. Autom. Conf.*, 1982, pp. 175–181.
[16] M. Garey, D. Johnson, and L. Stockmeyer, "Some simplified np-complete graph problems," *Theor. Comput. Sci.*, vol. 1, no. 3, pp. 237–267, 1976.
[17] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *SIGARCH Comput. Archit. News*, vol. 34, no. 5, pp. 151–162, Oct. 2006.
[18] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe, "A stream compiler for communication-exposed architectures," *SIGARCH Comput. Archit. News*, vol. 30, no. 5, pp. 291–303, Oct. 2002.
[19] C. Grelck, S.-B. Scholz, and A. Shafarenko, "A Gentle Introduction to S-Net: Typed stream processing and declarative coordination of asynchronous components," *Parallel Process. Lett.*, vol. 18, no. 2, pp. 221–237, 2008.
[20] L. Hagen and A. B. Kahng, "New spectral methods for ratio cut partitioning and clustering," *Trans. Comput.-Aided Des. Integrated Circuits Syst.*, vol. 11, no. 9, pp. 1074–1085, Nov. 2006.
[21] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Proc. ACM/IEEE Conf. Supercomput.*, article 28, 1995, Doi: 10.1145/224170.224228.
[22] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, "Optimization by simulated annealing: An experimental evaluation; part ii, graph coloring and number partitioning," *Oper. Res.*, vol. 39, no. 3, pp. 378–406, May 1991.
[23] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *J. Parallel Distrib. Comput.*, vol. 48, no. 1, pp. 96–129, Jan. 1998.
[24] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell Syst. Tech. J.*, vol. 49, no. 2, pp. 291–307, 1970.
[25] R. Khandekar, S. Rao, and U. Vazirani, "Graph partitioning using single commodity flows," in *Proc. 38th Annu. ACM Symp. Theory Comput.*, 2006, pp. 385–390.
[26] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, pp. 671–680, 1983.
[27] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proc. IEEE*, vol. 75, pp. 1235–1245, Sep. 1987.
[28] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, Jan. 1987.
[29] A. Malik and D. Gregg. (2012, Feb.). Executing synchronous data flow graphs on heterogeneous execution architectures using integer linear programming. School of Computer Science and Statistics, Trinity College Dublin, Ireland, Tech. Rep.. [Online]. Available: https://www.scss.tcd.ie/publications/tech-reports/tr-index.12.php
[30] A. Malik and D. Gregg, "Orchestrating stream graphs using model checking," *ACM Trans. Archit. Code Optimization*, vol. 10, no. 3, p. 19, 2013.
[31] Metis Partitioner Library. METIS-serial graph partitioning and fill-reducing matrix ordering. [Online]. Available: http://glaros.dtc.umn.edu/gkhome/metis/metis/overview, 2014.
[32] H. Meyerhenke, B. Monien, and S. Schamberger, "Graph partitioning and disturbed diffusion," *Parallel Comput.*, vol. 35, no. 10-11, pp. 544–569, Oct. 2009.
[33] B. Monien and S. Schamberger, "Graph partitioning with the party library: Helpful-sets in practice," in *Proc. 16th Symp. Comput. Archit High Perform. Comput.*, 2004, pp. 198–205.
[34] V. Nguyen and R. Kirner, "Demand-based scheduling priorities for performance optimisation of stream programs on parallel platforms," in *Proc. 13th Int. Conf. Algorithms Archit. Parallel Process.*, 2013, vol. 8285, pp. 357–369.
[35] V. Nguyen, R. Kirner, and F. Penczek, "A multi-level monitoring framework for stream-based coordination programs," in *Proc. 12th Int. Conf. Algorithms Archit. Parallel Process.*, 2012, vol. 7439, pp. 83–98.
[36] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *Proc. Int. Conf. Exhib. High-Perform. Comput. Netw.*, 1996, vol. 1067, pp. 493–498.
[37] F. Penczek, S. Herhut, C. Grelck, S.-B. Scholz, A. V. Shafarenko, R. Barrre, and E. Lenormand, "Parallel signal processing with s-net," in *Proc. Int. Conf. Comput. Sci.*, 2010, vol. 1, pp. 2085–2094.
[38] A. Pothen, "Graph partitioning algorithms with applications to scientific computing," in *Parallel Numerical Algorithms*. Norwell, MA, USA: Kluwer, 1997, pp. 323–368.
[39] A. Pothen, H. D. Simon, and K.-P. Liou, "Partitioning sparse matrices with eigenvectors of graphs," *SIAM J. Matrix Anal. Appl.*, vol. 11, no. 3, pp. 430–452, May 1990.
[40] D. Prokesch, "A light-weight parallel execution layer for shared-memory stream processing," Master's thesis, Faculty of Informatics, Technische Universität Wien, Vienna, Austria, Feb. 2010.
[41] P. Sanders and C. Schulz, "High quality graph partitioning," in *Proc. 10th Int. Conf. Graph Partitioning Graph Clustering*, 2012, pp. 1–18.
[42] D. G. Schweikert and B. W. Kernighan, "A proper model for the partitioning of electrical circuits," in *Proc. 9th Des. Autom. Workshop*, 1972, pp. 57–62.
[43] J. Shi and J. Malik, "Normalized cuts and image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 22, no. 8, pp. 888–905, Aug. 2000.
[44] G. C. Sih and E. A. Lee, "Declustering: A new multiprocessor scheduling technique," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 6, pp. 625–637, Jun. 1993.
[45] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *Proc. 11th Int. Conf. Compiler Construction*, 2002, pp. 179–196.
[46] C. Walshaw and M. Cross, "Mesh partitioning: A multilevel balancing and refinement algorithm," *SIAM J. Sci. Comput.*, vol. 22, no. 1, pp. 63–80, Jan. 2000.
[47] R. Wanschoor and E. Aubanel, "Partitioning and mapping of mesh-based applications onto computational grids," in *Proc. 5th IEEE/ACM Int. Workshop Grid Comput.*, Nov. 2004, pp. 156–162.

Vu Thien Nga Nguyen received the BcS degree in computer science from the HCM City University of Technology, Vietnam in 2007 and the MsC degree in grid computing from the University of Amsterdam, The Netherlands in 2010. She is currently working towards the PhD degree in computer science at the University of Hertfordshire, United Kingdom. Her current work focuses on efficient execution models for stream programs on parallel platforms. Her research interests include stream programming, parallel computing, scheduling methodologies, and runtime optimisation.

Raimund Kirner received the PhD degree in 2003 from the TU Vienna and his Habilitation in 2010. He is a reader in Cyberphysical Systems at the University of Hertfordshire. He has published more than 90 refereed journal and conference papers and received two patents. His research focus is on embedded computing, parallel computing, and system reliability. He currently works on adequate hardware and software architectures to bridge the gap between the many-core computing and embedded computing. He also published excessively on worst-case execution time analysis and served as a PC chair of WDES'06, WCET'08, and SEUS'13. He acted as the principal investigator of numerous national and European projects. He is a member of the IEEE, the ACM, and the IFIP Working Group 10.4 (Embedded Systems).