HIGH PERFORMANCE COMPUTING

Advanced Methodologies for Profiling and Partitioning Stream Programs on Many-Core Architectures

Published: November 24, 2025 | Index ID: #121

In the contemporary landscape of high-performance computing, the transition from single-core processors to many-core architectures has necessitated a fundamental shift in how software is designed, profiled, and executed. Stream programming models, characterized by their focus on data-parallelism and task-level concurrency, have emerged as a dominant paradigm for developing applications in domains such as digital signal processing, video encoding, and real-time data analytics. However, the efficient execution of these programs on many-core systems—often comprising dozens or hundreds of processing elements interconnected by a Network-on-Chip (NoC)—remains a formidable challenge. The core of this challenge lies in two critical phases: **profiling** the application to understand its behavior and **partitioning** the workload to balance execution across available hardware resources.

The Emergence of Stream Programming and Many-Core Systems

Stream programming represents a departure from traditional von Neumann execution models. In a stream program, the application is represented as a directed graph where nodes, often called **actors** or **kernels**, perform computations on streams of data tokens flowing along the edges. This model is inherently modular and exposes significant parallelism, making it an ideal candidate for many-core platforms such as Transport Triggered Architecture (TTA) processors and GPGPUs.

Despite the theoretical advantages, mapping these logical graphs to physical hardware is an NP-hard optimization problem. Many-core platforms introduce complex constraints, including limited local memory, varied communication latencies between cores, and the overhead of synchronization. To achieve optimal performance, engineers must employ sophisticated methodologies for **profiling** (measuring execution characteristics) and **partitioning** (allocating actors to specific cores).

The Theoretical Framework of Dataflow Programs

Most stream programs are rooted in dataflow formalisms. The most common is **Synchronous Dataflow (SDF)**, where the number of tokens produced and consumed by an actor is fixed and known at compile time. However, modern applications often require **Dynamic Dataflow (DDF)**, where actor behavior depends on the data values themselves (e.g., the RVC-CAL standard). DDF programs are more flexible but significantly harder to profile because their execution patterns can change dynamically based on input data.

Methodology for Accurate Profiling

Profiling is the process of collecting data regarding the execution of a program. In the context of stream programming on many-core architectures, profiling must capture not only the execution time of individual actors but also the dependencies and communication overheads between them.

The Execution Trace Graph (ETG)

One of the most robust methods for profiling dynamic stream programs is the construction of an **Execution Trace Graph (ETG)**. An ETG is a directed acyclic graph (DAG) that represents a single execution instance of a program. Each node in the ETG corresponds to a specific firing (execution) of an actor, and each edge represents a data

dependency or a state dependency between firings.

- Nodes: Represent the computation time of a specific actor firing, measured in clock cycles.
- Edges: Represent the precedence constraints. A node cannot begin execution until all its predecessor nodes have completed and their data is available.
- Critical Path: The longest path through the ETG, which determines the minimum possible execution time (makespan) regardless of the number of processors.

Profiling Challenges in Many-Core Environments

Accurate profiling on many-core systems faces several hurdles:

1. Non-determinism: Shared resources like NoC and global memory can introduce variance in execution times.
2. Probe Effect: The act of measuring performance (instrumentation) can alter the timing behavior of the system.
3. State Dependencies: Actors often maintain internal states, meaning the n^{th} firing might depend on the $(n-1)^{th}$ firing, limiting parallel execution.

The Partitioning Problem: Strategies and Algorithms

Partitioning involves grouping the actors of a stream program into sets, where each set is assigned to a specific Processing Element (PE). The goal is generally to minimize the **makespan**(the time between the start of the first actor and the completion of the last) while staying within resource constraints such as memory and power.

Heuristic Partitioning for Large Design Spaces

Given the exponential number of possible partitions in a many-core system, exhaustive search is impossible. Heuristic methods are required to explore the design space effectively. These methodologies typically use a cost function to evaluate the quality of a partition:

$$C = \max_{p \in PEs} \left(\sum_{a \in p} T_{exec}(a) + \sum_{e \in comm(p)} T_{comm}(e) \right)$$

Where T_{exec} is the execution time of actors assigned to a core, and T_{comm} is the communication overhead incurred when data crosses core boundaries.

Tabu Search for Dynamic Dataflow

Tabu Searchis a meta-heuristic local search algorithm used for mathematical optimization. It is particularly effective for partitioning because it can escape local optima by allowing "downhill" moves. The algorithm maintains a "Tabu List" of recently visited configurations to prevent the search from cycling back to previous states.

Genetic Programming and Evolutionary Approaches

As suggested in recent research (e.g., Genetic Programming Theory and Practice XV), evolutionary algorithms can be used to evolve partitioning strategies. By treating a partition as a "chromosome," the algorithm uses crossover and mutation operators to find high-performing configurations over several generations.

Technical Analysis of Partitioning Methodologies

The following table compares the most common partitioning approaches used in modern stream processing research:

Methodology	Description	Pros	Cons
Static Partitioning	Allocation decided at compile time based on estimated costs.	Low runtime overhead; predictable behavior.	Poor handling of dynamic workloads or variable data rates.
Tabu Search	Meta-heuristic that explores neighborhoods and avoids cycles via a Tabu list.	Excellent at escaping local optima; highly tunable.	Can be computationally expensive for very large graphs.
Genetic Algorithms	Population-based search using crossover and mutation.	Robust search over diverse design spaces.	Slow convergence; requires careful parameter tuning (NIND, GGAP).
Greedy Heuristics	Assigns actors to the least-loaded core at each step.	Extremely fast; easy to implement.	Frequently gets stuck in local optima; ignores long-term dependencies.
Simulation-based	Uses a virtual model of the architecture to estimate performance.	High accuracy; accounts for NoC contention.	Very high computational cost for the profiling phase.

Transport Triggered Architecture (TTA) as a Validation Platform

Research by M. Michalska et al. (2015) utilized an array of **Transport Triggered Architecture (TTA)** processors to validate profiling measures. TTA is a unique processor design where the programmer or compiler explicitly manages data movement between functional units. This transparency makes TTA an ideal target for profiling because the cost of every operation and data move is explicitly known, leading to highly accurate Execution Trace Graphs.

Practical Implementation: A Step-by-Step Workflow

For engineering teams looking to implement a profiling and partitioning pipeline, the following procedural guide outlines the best practices based on current technical literature.

Step 1: Program Specification and Instrumentation

Define the application using a dataflow language like RVC-CAL or C-based stream libraries. Instrument the code to log actor firing start/end times and token counts. On many-core platforms, hardware counters (e.g., cycle counters) should be used to ensure high-resolution data collection.

Step 2: Generation of the Execution Trace Graph (ETG)

Execute the program on a simulator or a reference hardware setup. Capture the trace of all firings. Post-process this trace to build the ETG. This graph must account for both **data dependencies** (tokens) and **logical dependencies** (actor state).

Step 3: Cost Modeling and Estimation

Develop a model for the target hardware. This includes the execution time for each actor on a specific PE and the latency of the communication fabric (NoC). If the architecture is heterogeneous, the cost model must reflect that an actor may take longer on a "little" core than on a "big" core.

Step 4: Neighborhood Exploration via Tabu Search

Initialize a partition (e.g., using a simple Round-Robin distribution). Define a "neighborhood" as any partition that can be reached by moving one actor from one core to another. Use the ETG and cost model to evaluate the neighbors. Select the best neighbor that is not on the Tabu List.

Step 5: Mapping and Deployment

Once the optimization algorithm converges, generate the mapping table or configuration files required by the runtime environment. Deploy the partitioned application to the many-core target.

Mobile Cloud Partitioning: A Special Case

As noted in the research by L. Yang and LJ Yao, partitioning is not limited to multi-core chips; it also applies to **Mobile Cloud Computing (MCC)**. In this context, the "partition" determines which parts of a mobile application run locally on the device and which are offloaded to the cloud.

Offloading Decision Factors

- Wireless Bandwidth: High-bandwidth environments favor cloud offloading for compute-heavy tasks.
- Energy Constraints: If the energy cost of transmitting data to the cloud is less than the energy cost of local execution, the task should be partitioned to the cloud.
- Latency Sensitivity: Real-time tasks (e.g., Augmented Reality) may require local execution regardless of energy costs to meet latency bounds.

The methodology for MCC partitioning often involves **online profiling**, where the device monitors current network conditions and CPU load in real-time to adjust the partitioning configuration dynamically.

Troubleshooting Common Failure Modes in Partitioning

Partitioning is rarely perfect on the first attempt. Engineers must be aware of common pitfalls that can degrade performance:

1. Load Imbalance

If one core is assigned a disproportionate amount of work, it becomes a bottleneck, causing other cores to idle while waiting for data. **Solution:**Re-evaluate the actor execution weights in the ETG and run the Tabu Search with a higher emphasis on load balancing.

2. Communication Saturation

Aggressively splitting actors across many cores can lead to excessive NoC traffic, where the time spent moving data exceeds the time saved through parallel execution. **Solution:**Include a communication-to-computation ratio (CCR) threshold in the partitioning heuristic to encourage "clustering" of highly connected actors.

3. Deadlocks in Dynamic Dataflow

In dynamic programs, a specific partition might lead to a circular dependency if buffer sizes are finite. **Solution:** Use formal verification or simulation-based deadlock detection during the partitioning phase to ensure the validity of the mapping.

Synthesis and Broader Implications

The methodologies for profiling and partitioning stream programs represent a critical intersection of compiler theory, operations research, and hardware architecture. By leveraging Execution Trace Graphs and meta-heuristic search algorithms like Tabu Search, it is possible to unlock the massive parallel potential of many-core

architectures. The transition from manual, intuition-based partitioning to automated, data-driven methodologies is essential as the complexity of both software and hardware continues to scale.

Looking forward, the integration of Machine Learning (ML) into the profiling phase offers promising avenues. Predictive models could estimate actor execution times across different inputs, further refining the accuracy of ETGs without the need for exhaustive simulation. As we move toward 1000-core processors and seamless mobile-to-cloud integration, the ability to intelligently partition workloads will remain a cornerstone of efficient computing. Engineers and researchers must continue to refine these frameworks, ensuring that the software can effectively "tame" the increasingly complex hardware landscapes of the future.

Baca Juga (Artikel Terkait)

- **The Comprehensive Guide to Grid and Cluster Computing: Architectures, Algorithms, and Technical Evolution**

  URL: http://alghafgolden.com/comprehensive-guide-grid-cluster-computing-architectures-evolution.pdf

Tags: #Many core Architecture #Stream Programming #Tabu Search #Dataflow Analysis #Profiling Tools