A Load Balancing Inspired Optimization Framework for Exascale Multicore Systems: A Complex Networks Approach

Yao Xiao, Yuankun Xue, Shahin Nazarian, Paul Bogdan
Department of Electrical Engineering
University of Southern California, Los Angeles, CA, USA
{xiaoyao, yuankunx, shahin.nazarian, pbogdan}@usc.edu

***Abstract*—Many-core multi-threaded performance is plagued by on-chip communication nonidealities, limited memory bandwidth, and critical sections. Inspired by complex network theory of social communities, we propose a novel methodology to model the dynamic execution of an application and partition the application into an optimal number of clusters for parallel execution. We first adopt an LLVM IR compiler analysis of a specific application and construct a dynamic application dependency graph encoding its computational and memory operations. Next, based on this graph, we propose an optimization model to find the optimal clusters such that (1) the intra-cluster edges are maximized, (2) the execution times of the clusters are nearly equalized, for load balancing, and (3) the cluster size does not exceed the core count. Our novel approach confines data movement to be mainly inside a cluster for power reduction and congestion prevention. Finally, we propose an algorithm to sort the graph of connected clusters topologically and map the clusters onto NoC. Experimental results on a 32-core NoC demonstrate a maximum speedup of 131.82% when compared to thread-based execution. Furthermore, the scalability of our framework makes it a promising software design automation platform.**

I. Introduction

The tight power and thermal constraints call for fine-grained exploration of chip multiprocessors (CMPs) and data-center-on-a-chip [2] to provide performance improvement in exascale computing. To make use of CMPs, software paradigms have been shifted from sequential programming to multi-threading. However, three fundamental inefficiency issues can appear if threads are spawned without careful consideration of the underlying hardware.

(1) **Non-negligible on-chip communication overhead**. With applications being randomly partitioned, plenty of inter-core communications are generated, leading to many flits injected into the network. Those flits are unwanted as none of them would exist with running the application in only one core. Therefore, intelligent application partitioning is required to minimize inter-core communication overhead.

(2) **Limited off-chip memory bandwidth**. Due to limited off-chip memory bandwidth, the performance of data-intensive multi-threaded programs is negatively affected due to frequent updates in the main memory. A critical thread may be delayed due to race conditions to access the main memory simultaneously. Increasing the number of threads to the point of off-chip bandwidth saturation degrades the power consumption with no performance increase.

(3) **Increased critical sections**. Locks are used to prevent multiple threads from writing to shared variables simultaneously to guarantee the correctness of a multi-threaded program. In other words, due to synchronization, the serial portions of the program increase. According to Amdahl's Law, speedup is limited by the sequential parts. Therefore, as more threads are spawned, thread-based execution could potentially become slower compared to sequential execution.

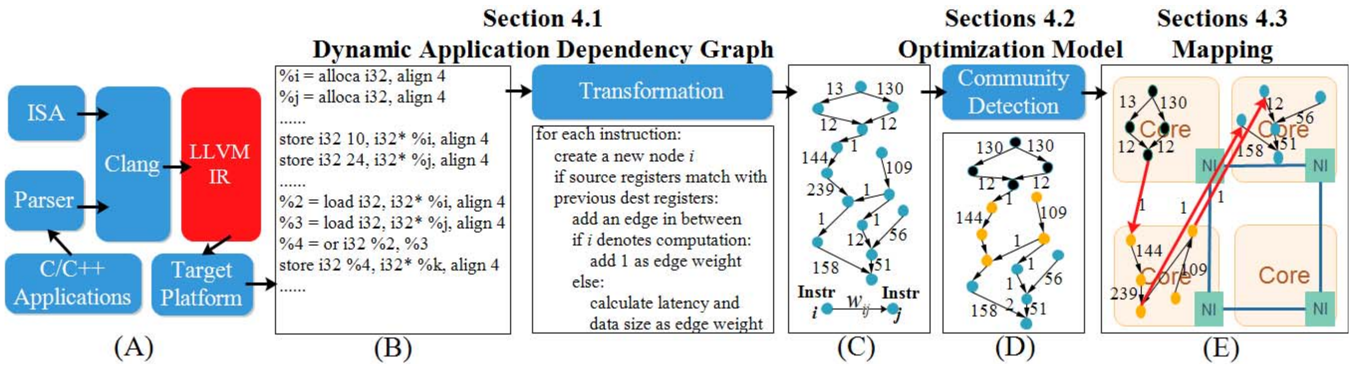
In this paper, **the goal** is to design a novel methodology to automatically parallelize complex programs without increasing the programmer's efforts. Considering the three pitfalls mentioned previously, we propose a complex network based parallelization framework to partition applications into **highly interdependent clusters of tasks** representing **communities**[1] in a graph rather than threads such that the amount of data transferred among communities is minimized. As shown in Figure 1, we first construct the weighted dynamic application dependency graph where the nodes denote individual low level virtual machine (LLVM) intermediate representation (IR) [13] instructions generated by *Clang* compiler, and edges represent data dependencies between different instructions on the same virtual registers. Edge weights represent latency (L1 hits, L1 misses, or L2 misses assuming there is a shared L2 cache among cores) and data sizes (1, 2, 4, 8 bytes, cache line, or memory page). Second, based on the constructed graph of IR instructions we present the mathematical optimization model to detect community structures ensuring that (1) **the number of inter-cluster communication flits is minimal**; (2) **communities reach approximately equalized execution times**; (3) **the number of communities is smaller than or equal to the number of cores**. Third, having calculated the optimal communities and their dependencies, we construct a cluster graph where nodes indicate communities. We then use topological sort to map the clusters onto the NoC, while ensuring that the clusters at the same depth are executed in parallel. In case the number of clusters is smaller than the core count, the rest of the cores are shut off using power gating.

There are three primary issues that cause performance degradation in parallel computing: (1) **load imbalance**, (2) **resource sharing**, (3) **synchronization**. Our framework mitigates these three bottlenecks by a robust real-time aware optimization approach that (1) prevents the difference of execution times between two consecutive clusters from being too large considering cache miss cycles and data sizes for memory instructions, (2) confinies most of data movement within each cluster as the framework tries to partition the dependency graph into clusters with maximized intra-cluster communications, making better use of caches. Moreover, mesh-based NoC is used to route flits efficiently for cases where a core requires variables stored in another core's caches, and (3) applies pipeline parallelism to parallelize sequential applications rather than multi-threading to reduce synchronization overhead caused by threads with locks and barriers.

Towards this end, our main contributions are as follows.

- We present an architecture-independent application profiling approach that identifies the IR-level data dependency of one application and represents it as a dynamic application dependency

[1]Of note, throughout the manuscript, we use the terms "community", "cluster", and "subcomponent" as synonyms.



```
%i = alloca i32, align 4
%j = alloca i32, align 4
......
store i32 10, i32* %i, align 4
store i32 24, i32* %j, align 4
......
%2 = load i32, i32* %i, align 4
%3 = load i32, i32* %j, align 4
%4 = or i32 %2, %3
store i32 %4, i32* %k, align 4
......
```

```
for each instruction:
    create a new node i
    if source registers match with
    previous dest registers:
        add an edge in between
        if i denotes computation:
            add 1 as edge weight
        else:
            calculate latency and
            data size as edge weight
```

Figure 1: **Overview of the framework**. **A**: Input C/C++ applications. **B**: A compiler based parser obtains the dynamic LLVM IR instructions from C/C++ applications. **C**: Construct a DADG by analyzing the dependencies between LLVM IR instructions. The DADG nodes represent IR instructions; edges represent dependencies between instructions; weights are collected by instrumenting a lightweight function *rdtsc*() and some inlined code to find the latency and data size for memory operations. Weights for the rest of instructions are set to 1. **D**: We develop a mathematical optimization model to partition the DADG into several clusters considering maximum intra-cluster edge weights, load balancing, and availability of hardware resources. **E**: We map clusters onto NoC based on Topological Sort for parallel execution.

graph (DADG).

- We propose a complex-network inspired framework to automatically parallelize the execution of the application with minimized inter-core traffic overhead by intelligently partitioning DADGs into networked (distributed) processing communities.
- We develop a power-aware scheduling and mapping algorithm that parallelizes the execution of the clusters on an NoC-based multi-core platform with optimized power consumption via power-gating.

The rest of the paper provides the related work, preliminaries, complex network based parallelization framework, simulation results, and conclusions in successive sections.

II. Related Work

In recent years, there has been a surge of efforts in exploiting fine-grained parallelism and minimizing the execution overhead (e.g., energy, runtime, thermal and communication cost) of applications on multi-core systems by optimizing the task-to-core mapping and scheduling based on the application task structures. To take advantage of the graphical models of parallel applications, prior efforts focus on both runtime and static extraction of execution dependency structures at task [18][21][23] or dataflow [14] level for both conventional and NoC-based multi-core systems. Based on the knowledge of the inter-dependencies among different tasks or threads, the optimal modeling, parallelization, scheduling, mapping and data routing of an application on a given architecture is further investigated from a wide spectrum of perspectives. For high-level modeling and control of multi-core systems, the authors in [3] propose a complex dynamics approach to characterize the workloads for dynamic optimization of performance, power and thermal overhead. Leveraging the learned dependency structures, fine-tuned scheduling and mapping strategies are also well studied for task and thread execution [7][9][22], memory access [26], energy efficiency [4] on both homogeneous and heterogeneous multi-core systems at both design and execution time [19]. For automatic parallelization at the task level on general purpose platforms, Fonseca et al. [10] propose a framework to automatically perform parallelization of sequential Java programs by identifying data access patterns and control flow information. Similar idea is proposed in Li et al. [15] by discovering code sections which can run in parallel, and ranking those parallel targets to provide feedback to users. Cordes et al. [5] extend the automatic parallelization to embedded systems by leveraging a hierarchical task dependency graph based on inter-linear programming (ILP). On a different direction, fine-grained parallelism is also exploited by synthesis or reconfiguration of the communication and computation architectures best-fit to the learned application structures. Xue et al. [24] unify the learning of application task structures with optimized reconfiguration of NoC-based multi-core system in a general mathematical model that provides guaranteed optimality. Despite significant research developments that take advantage of applications parallelism at different levels or on different platforms, we propose a radical approach that explores the IR instruction-level fine-grained parallelism in order to guide the design and optimization of multi-core platforms.

III. Preliminaries

In this paper, we describe a complex network based framework to parallelize sequential programs by constructing the corresponding dynamic application dependency graph (DADG), identifying edge weights between dependent instructions, and guiding the cluster-to-core scheduling and mapping while balancing the workload and minimizing the data movement. Therefore, in this section we provide preliminaries on pipeline parallelism to speed up applications and mathematical models to detect communities.

A. Pipeline Parallelism

Pipeline parallelism [20] is a technique where instructions inside the outer loop are split into stages preserving serial semantics, which are assigned to different cores in order to be executed in time-sliced fashion like hardware pipelining.

In Figure 2, a sequential program is partitioned into three stages with distinct features. For example, to compress files, first stage is to open one file; second stage is to compress all words in the file; and the third segment is to write compressed words into another file. Without parallelism, we can only compress one file in three time units assuming all three stages have the same execution times. However, with the help of the compilers and programmers, pipeline parallelism can be achieved by inserting software buffers between two stages to store the intermediate results. For each iteration, stage 1 enqueues data necessary for stage 2 in the current buffer (pipe). The data, when required, is dequeued into the stage 2 for processing but at the same time stage 1 operates on the next iteration. This way, the overall speedup is nearly the number of stages in each program. One caveat is that producing too many stages increases communication overhead including enqueue and dequeue operations.

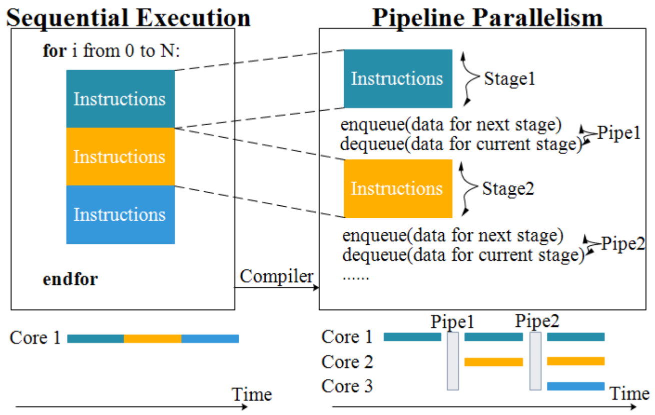


Figure 2: From sequential execution to pipeline parallelism

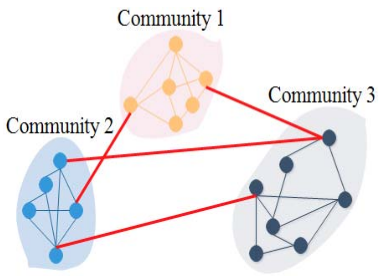


Figure 3: Community detection. Three communities are detected based on the maximum number of intra-community edges.

B. Community Detection

Community structures [16] refer to the grouping of nodes with high density of edges inside each community, shown in Figure 3. It is of great significance to detect those structures for which scientists can find self-similarity [25] of groups of nodes, for example, in social networks.

To detect common structures, *hierarchical clustering* [11] is applied based on similarity of nodes. This technique is categorized into two classes: *agglomerative* and *divisive* [17]. In an agglomerative method, initially each node belongs to one cluster. Later more and more nodes are merged together depending on the high connectivity between nodes. In a divisive approach, starting from the original network, only edges between nodes with low similarity are removed to form clusters. However, *modularity* [11] has been proposed and proven to be more efficient compared to the above-mentioned methods. Modularity-based community detection is basically an optimization problem where given a complex graph, we want to maximize the following quality function Q to get the best fitting clusters [17]:

$$Q = \sum_{i=1}^{n} (e_{ii} - {a_i}^2) \tag{1}$$

where n denotes the number of communities in a network; e_{ij} denotes the number of edges cluster i connects with cluster j; and a_i denotes the number of all edges cluster i connects with all clusters including itself, which can be expressed as $a_i = \sum_j e_{ij}$. Therefore, modularity is based on a quality function to calculate the number of edges which falls within clusters minus the expected number of edges in a random graph with the same expected degree for each node [11].

Despite its benefits, community detection strategies have not been exploited in parallel computing to discover the communities of intensive processing and inter-dependencies among instructions. Nevertheless, we think that due to synchronization overhead, limited off-chip bandwidth, and complex on-chip communication traffic, in order to achieve better performance, one cannot assume to use all of available cores. This paper proposes a new parallelization framework based on concepts in complex network to obtain the optimal number of clusters with minimal inter-cluster edge weights based on the data-dependent structures of each application.

IV. Complex Network Based Parallelization Framework

The proposed parallelization framework consists of three stages. In the first stage, an application is first analyzed by the LLVM compiler to construct a DADG by collecting the dynamic LLVM IR traces and identifying the interdependencies between computation and memory operations. In the second stage, the constructed DADG is partitioned into clusters based on the identified network communities with the balanced workloads. Finally, the partitioned application is scheduled and mapped to different cores in an NoC-based multi-core system for parallel execution.

A. Spatio-Temporal Interdependency Description of Computations and Communications

To describe the spatio-temporal interdependencies between the computations and memory operations, we adopt an architecture independent LLVM IR [13][18]. The rationale for adopting this compiler framework is that it is a language-independent type-system that exposes the primitives used to implement high-level language (HLL) features. It includes an instruction for typed address arithmetic, and a mechanism for implementing the exception handling HLL features. Furthermore, IR is crucial in LLVM. It is an abstract machine language which mimics the basic computations, memory operations, and branch instructions with unlimited virtual registers to prevent register spilling. Therefore, backends can easily produce from IR, machine code suited for any target platform regardless of ARM in portable mobiles or x86 in laptops and high-end servers. As shown in Figure 1, there are several features for our approach:

IR instructions are collected dynamically. Static compilation has several drawbacks. (1) Memory operations are difficult to detect dependencies, which could potentially increase communication overhead if we map dependent memory operations onto different cores. (2) The number of iterations in one loop sometimes cannot be statically determined. Depending on how many iterations one loop has, load imbalance appears between different clusters. Therefore, rather than static compilation, dynamic execution traces are collected to reflect true dependencies and break one loop into several iterations executing sequentially, increasing the chances of grouping different iterations into clusters.

Memory operations are instrumented to get correct values for latency and data sizes. The store and load instructions have different execution times and data sizes if data required to fetch reside in L1, L2, L3, or main memory. Taking into account those values could potentially group computations and memory operations with the same registers into one cluster, leading to more efficient use of caches and less communication overhead. In this way, load balancing is achieved by explicitly formulating weight constraints in an optimization model.

The parser collects C/C++ essential instructions within an outer loop and constructs a DADG from dynamic traces generated by the compiler. In the parser, we maintain three hash tables called source table, destination table, and dependency table respectively. The source/destination tables are used to keep track of source/de-stination registers with keys being source or destination

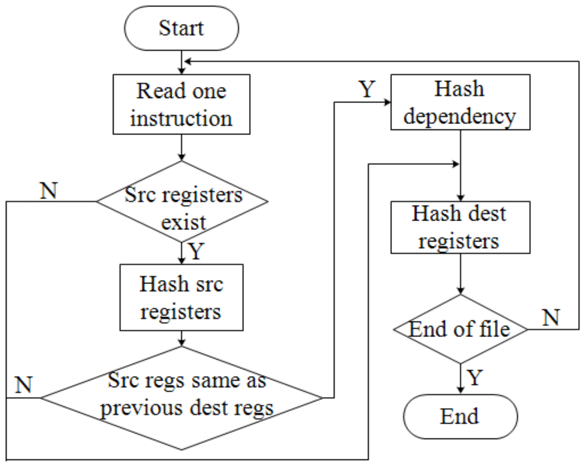
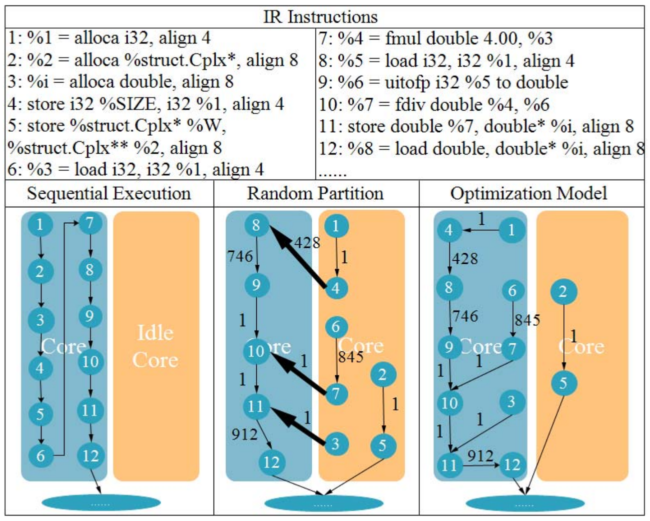


Figure 4: The flow chart of constructing a DADG


IR Instructions

```
1: %1 = alloca i32, align 4
2: %2 = alloca %struct.Cplx*, align 8
3: %i = alloca double, align 8
4: store i32 %SIZE, i32 %1, align 4
5: store %struct.Cplx* %W,
%struct.Cplx** %2, align 8
6: %3 = load i32, i32 %1, align 4
7: %4 = fmul double 4.00, %3
8: %5 = load i32, i32 %1, align 4
9: %6 = uitofp i32 %5 to double
10: %7 = fdiv double %4, %6
11: store double %7, double* %i, align 8
12: %8 = load double, double* %i, align 8
......
```

Figure 5: Comparison among different partitions

registers and values being the corresponding line number. The dependency table is to store dependencies between nodes with keys being the line number for current instruction, and values being clock cycles, data sizes and line numbers of previous instructions dependent on the same virtual register. The parser instruments the lightweight *rdtsc* function and some inlined code to collect the attributes of memory operations, i.e., data sizes and clock cycles as edge weights.

Example: In Figure 5, twelve IR instructions are generated by the compiler *Clang*. As soon as the parser reads the first instruction in Figure 5, it checks source registers as indicated in Figure 4. Since this instruction does not have the source register, then only the destination register is hashed into the destination table with keys being %1 and values being 1. Instructions 2 and 3 follow the same procedure as the first instruction. When the parser reads the fourth instruction[2], it checks whether the source registers in the instruction match with any destination registers in previous instructions. In this case, the source register %1 matches with the same destination register in node 1. Thus, this is hashed into the dependency table with keys being 4 (the line number of the current instruction), values being 1 (the line number of the previous instruction which depends on the source register %1), and weights being 1 (non-memory operations). The dependency table can be regarded as a DADG in which keys represent nodes and key-value pairs indicate directed edges.

B. Mathematical Optimization Model

In order to propose a rigorous mathematical strategy for parallelizing applications, we build on our architecture independent spatio-temporal DADG representation of an application and formulate a novel community detection problem that seeks to: (1) Determine a strongly connected subcomponent of the DADG that encapsulates strong causal dependencies between computations and memory operations dependent on registers in computations yet is complex enough to require localized specialized processing elements (functional units in cores or accelerators) and corresponding memory systems; (2) Perform load balancing by distributing computations and related memory operations among the identified computational communities to improve system performance under uncertain inputs on average; (3) Minimize the deviations between the number of strongly connected

[2]Registers %SIZE and %1 are hashed into the source table to prevent from violating true memory dependencies. Register %1 is hashed into the destination table.

communities (subgraphs of the DADG) and the hardware resources (cores or accelerators). To make the discussion more concrete, we introduce a series of definitions that help us construct the community detection problem as follows.

Definition 1: A dynamic application dependency graph (DADG) is a *weighted directed* graph $\mathcal{G} = G(n_i, e_{ij}, w_{ij}|i, j \in |N|)$ where each node n_i represents one LLVM IR instruction, and each edge e_{ij}, associated with different weights w_{ij}, characterizes dependency from the current node n_i to the previous node n_j or control flow such as jump and branch to guarantee the strict program order.

Definition 2: A weight w_{ij} between node i and j is defined as latency function $\mathcal{T}(e_{ij})$ times data size $\mathcal{D}(n_i)$. Latency function $\mathcal{T}(e_{ij})$ calculates the latency from node i to node j based on the timing information for memory operations provided by the compiler. Likewise, data size $\mathcal{D}(n_i)$ calculates the number of bytes node i requires to transfer from one location to another location (possible locations are disk, main memory, caches, processor registers).

Definition 3: A quality function Q for DADGs is an indicator of how good a grouping of clusters for parallel execution is based on load balancing, available hardware resources, and data movement.

Using these definitions, the mathematical optimization model in terms of intelligently partitioning DADGs can be formulated as follows:

Given a DADG $\mathcal{G}$, **find** non-overlapping clusters n_c which **maximize** a quality function Q [3]:

$$Q = \sum_{c=1}^{n_c}[\frac{W^{(c)}}{W} - (\frac{S^{(c)}}{2W})^2] - R_1 - R_2 \tag{2}$$

$$R_1 = \frac{\lambda_1}{W^2}\sum_{c=1}^{n_c}[W_c - W_{neighbor(c)}]^2 \tag{3}$$

$$R_2 = \frac{\lambda_2}{n_c^2}(n_c - N)^2 H(n_c - N) \tag{4}$$

where n_c denotes the cluster size; N denotes the core count; $W^{(c)}$ denotes the sum of weights all connected within cluster c ($W^{(c)} = \sum_{i\in c}\sum_{j\in c} w_{ij}$); W is the sum of weights of all edges

[3]Note that the formula is used for undirected networks. We think this is a simple way to model directed networks.

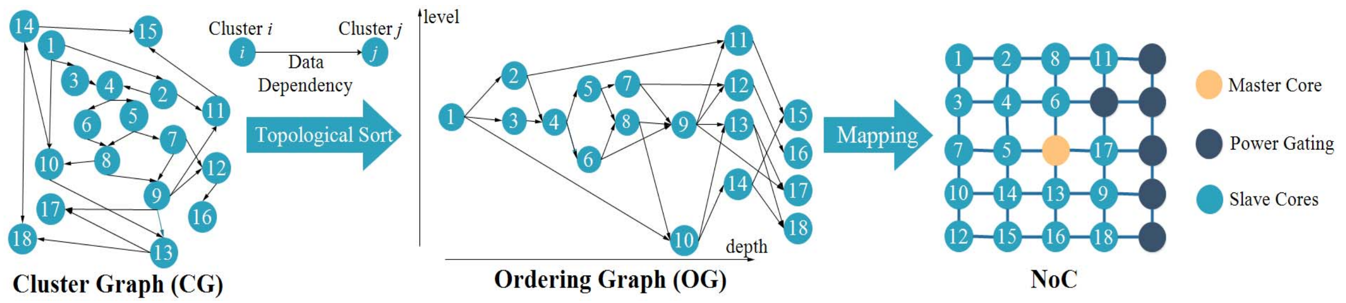


Figure 6: Mapping clusters onto NoC for parallel execution. First, we convert the representation of the cluter graph into the ordering graph by Topological Sort. Topological Sort, essentially, reorders a directed acyclic graph (DAG) based on the rule that for every direct edge e_{ij} between nodes i and j, i comes before j in the ordering graph. Then, we map nodes with no incoming edges in the ordering graph onto NoC, making sure nodes and their neighbors should be adjacent to each other to reduce the transmission distance.

($W = \sum_i \sum_j w_{ij}$); $S^{(c)}$ is the sum of weights of all the edges adjacent to cluster c; λ_1 and λ_2 are *regularization* parameters; $neighbor(c)$ denotes clusters connected to the cluster c; $H(x)$ is the *Heaviside step function* ($H(x) = \int_{-\infty}^{x} \delta(s)ds$); $\delta(x)$ is the *Dirac delta function*.

Intuitively, the first term in equation (2) aims to maximize the intra-cluster weights and reduce the communication requirements among different clusters. The second and third terms aim to balance the computational (processing) requirements for each core and account for the limited number of hardware resources imposed by pipeline parallelism. Similar to prevention of over-fitting in machine learning, R_1 and R_2 are regularization terms and $\frac{1}{W^2}$, $\frac{1}{n_c^2}$ are used to make sure those terms have the same unit:

(1) The first term in equation (2) limits data movement almost within each cluster. It measures the difference between the sum of edge weights in a cluster and the sum of edge weights adjacent to the cluster. Through maximization of this term, we try to find partitions where data movement is constrained.

(2) R_1 is used for *load balancing*. By measuring the sum of the deviation squared between the total weights in a cluster c and its neighbors, R_1 is magnified and the value of Q is reduced if clusters have unbalanced weights. Therefore, by maximizing Q, R_1 is meant to be minimized by equalizing works in clusters c and its neighbor. Therefore, we try to balance the weights/work between different clusters, making stages in pipeline parallelism have roughly equalized execution times.

(3) R_2 is used to ensure the number of clusters doesn't exceed the core count. $(n_c - N)^2$ means if the number of clusters n_c is different from the number of available cores N in a system, Q is further reduced. However, $H(n_c - N)$ takes a value of 0 until n_c equals N, and then has a value of 1 if n_c is greater than N. Hence, R_2 is large **only** when n_c exceeds the number of cores in the system. If n_c is less than N, $H(n_c - N) = 0$ and the rest of idle cores are turned off to save energy while providing the best performance. Therefore, in order to maximize Q, R_2 should be minimized by making sure that n_c (the number of communities) is slightly less than N (core count).

(4) For regularization parameters λ_1 and λ_2, both can be adjusted during run-time. If $\lambda_1 = \lambda_2 = 0$, the quality function Q is reduced to a standard model without considering balanced works and available resources. If λ_1, λ_2 are very large numbers, the first term in equation (2) can be ignored, and the model tries to detect communities such that balanced works and n_c less than N are achieved without evaluating the maximum communication messages restricted within one community. Therefore, values of λ_1 and λ_2 should be somewhere in-between. In section 5.2.2, we will show some results regarding the effects of different values of parameters on the number of clusters and execution times.

The advantages of applying the mathematical optimization model to partition sequential programs are: (1) **Minimal programmer's efforts to write parallel programs to exploit the speedup provided by multi-core chips**. (2) **Easy to detect independencies at the granularity of IR instructions and balanced load among clusters**. (3) **Limited communication overhead in NoC leading to small chances of congestion**. As shown in Figure 5, mapping the entire application into one core would mean no communication overhead among cores, but this approach cannot improve performance as the other core is being idle all the time. The second method is just to partition the graph randomly. However, this random partitioning can cause significant communication overhead among cores, making cache utilization and performance poor. The last one can group many instructions into clusters such that the number of inter-cluster flits is minimized. Data movement is restricted by keeping data locally as much as possible to save energy and improve performance.

C. Mapping

Determining the optimal number of clusters raises the question on how to map them onto the NoC such that (1) hop count of communicated flits among clusters is minimized and (2) independent clusters can be executed in parallel.

A cluster graph (CG) is constructed where nodes represent clusters and edges indicate data dependencies. There are two properties associated with CG. (1) CG is *directed*: As there should be an order in which tasks are executed due to program sequential semantics, one task waits data provided by the other tasks until it is executed, leading to a *directed* graph. (2) CG is *acyclic*: One cluster depends on data which are generated from its previous clusters. Based on the *directed* and *acyclic* graph, we sort CG topologically to ensure that for any directed edge (v, w) in $E \in CG$, v precedes w in the ordering, which can be expressed as an Ordering Graph (OG). Based on OG, clusters are mapped into NoC for pipeline parallelism. In conclusion, we propose the following algorithm which is a combination of Topological Sort and mapping.

Algorithm 1 exploits parallelism and pipelining. We define depth of cluster v in OG as the number of edges from v to its root, and level of cluster v as the number of clusters at the same depth as v. Therefore, (1) **Depth of i represents a stage $i+1$ in pipelining**. In Figure 6, clusters 2 and 3 in OG at the depth of 1 represent 2nd

stage while cluster 4 at the depth of 2 represents 3rd stage. Moreover, cluster 4 cannot be executed before clusters 2 and 3 as it waits data generated by 2nd stage. (2) **Different levels at the same depth of i represent the number of clusters which can be executed in parallel**. In Figure 6, clusters 2 and 3 at the stage 2 can be executed in parallel as they both only depend on availability of data produced by cluster 1. After mapping, if there are still idle cores, to save power consumption, those cores are turned off using power gating.

Algorithm 1 Mapping Algorithm

```
Input: Clusters and their dependencies in $CG = (V, E)$ from
    Section 4.2
Output: Mapping of clusters in $CG$ into NoC
 1: $Counter = 0$
 2: while $CG$ is not empty do
 3:     $V_{partial}$ = No_Incoming_Edges ($CG$)
 4:     if $Counter == 0$ then
 5:         Map $V_{partial}$ to $(0, 0)$
 6:     else
 7:         Map $V_{partial}$ to their nearest parent clusters based on
 8: Greedy Heuristic
 9:     end if
10:     Running_In_Parallel ($V_{partial}$)
11:     Delete $V_{partial}$ from $CG$
12:     $Counter$++
13: end while
14: if There still exist idle cores $C$ in NoC then
15:     Power_Gating ($C$)
16: end if
```

V. Evaluation

In this section, we provide simulation configurations and experimental results to demonstrate the validity of our framework.

A. Simulation Configurations

We simulate a symmetric CMP with all out-of-order cores in NoC with the parameters shown in Table 1. Three different types of execution are considered and compared: sequential execution where all instructions are executed in one core, thread-based execution where the number of threads spawned is equal to core count, and optimization-based execution discussed in this paper. Thread-based and optimization-based executions are evaluated on a 32-core NoC as all of our workloads can be configured to 32 communities if proper λ_2 is applied. Table 2 shows the simulated workloads.

B. Experimental Results

1) Complex Network and Basic Properties: Figure 7 shows the DADG connectivity structure for several applications (e.g., *MM.1*, *Dijkstra*, and *Blackscholes*). Table 3 summarizes their main attributes.

In Figure 7, DADGs can be classified into 3 categories: **high/medium/low discernibility**. High discernible DADGs are clearly seen as some interconnected clusters. One of examples is *MM.1*. Medium discernible DADGs may be seen as some regular patterns by humans. One of examples is *Dijkstra*. Those applications mentioned above can be parallelized by programmers without many efforts. The similarity in high and medium discernible DADGs is that there are some visible patterns to humans. The reason is that array declaration in C/C++ programs corresponds to one IR instruction called "getelementptr". Therefore, all array-related operations depend on this instruction. This node is becoming central and betweenness is very high due to array operations. One cluster should be centered on at least one of those nodes. As we can see in Figure 7, we can infer that we have at least three distributed arrays and operations on one array barely depend on those on another arrays. However, low discernible DADGs are difficult to be seen as interconnected clusters clearly to humans. Those applications are hard to be parallelized by programmers. One of examples is *blackscholes*. Therefore, we apply the parallelization framework to those applications to reduce programmer's efforts, making it practical to be executed in parallel.

In Table 3, average degree measures the sum of the number of edges incident on each node divided by the total number of edges. The reason why average degree is around 2 is that the number of edges in DADGs depend on the number of source registers. Almost all IR instructions except call functions have only one or two source registers, making DADGs sparse.

2) Effects of λ_1 and λ_2 on Load Balancing and Cluster Count: To illustrate the effects of different values of λ_1 and λ_2 on load balancing and cluster count, results regarding 10 applications are shown in Figure 10. $|R_1|$ is the second term used in equation 2, meaning sum of the difference of loads between two consecutive clusters. The higher $|R_1|$, the worse load balancing.

In Figure 8, from figures in the first column, $|R_1|$ has its peak when $\lambda_1 = 0$. However λ_1 can be adjusted to fine-tune the load difference among clusters, making tasks more balanced. After λ_1 approaches some threshold, $|R_1|$ plummets to 0 as R_1 is becoming the dominant term in equation (2) compared to the first term. It is

Table I: Configuration parameters

CPU	cores	OOO, 2-wide issue, 16 MSHRs
	L1 private caches	64KB, 4-way associative, 32-byte blocks
	L2 shared caches	256KB, distributed across nodes
Network	Topology	Mesh
	Routing Algorithm	XY routing
	Flow Control	Virtual channel flit-based

Table II: Benchmarks and descriptions

Benchmark	Description	Source
Mandelbrot	Calculate Mandelbrot Set	OmpSCR[8]
MM.1	Simple matrix multiplication	
Stencil	2D nine point stencil operation	SHOC[6]
MD	Simulate molecular dynamics	OmpSCR[8]
FFT	Compute Fast Fourier Transform	OmpSCR[8]
Dijkstra	Find the shortest path	MiBench[12]
Blackscholes	Calculate European options	PARSEC[1]
FFT6	Compute 1D FFT	OmpSCR[8]
MM.2	Strassen's matrix multiplication	
qSort	Quicksort algorithm	OmpSCR[8]

Table III: The main properties of DADGs

Benchmark	Nodes	Edges	Avg degree	Avg path length
Mandelbrot	1,045,696	1,315,168	2.549	7.718
MM.1	1,489,656	1,957,420	2.701	14.517
Stencil	2,107,098	2,847,856	2.549	13.703
MD	1,498,210	2,070,557	2.571	19.307
FFT	610,011	799,933	2.486	13.425
Dijkstra	398,674	550,462	2.433	11.179
Blackscholes	1,236,128	1,516,241	2.35	23.296
FFT6	338,920	458,273	2.513	21.753
MM.2	1,514,622	2,063,665	2.422	16.33
qSort	1,118,977	1,442,563	2.353	29.312

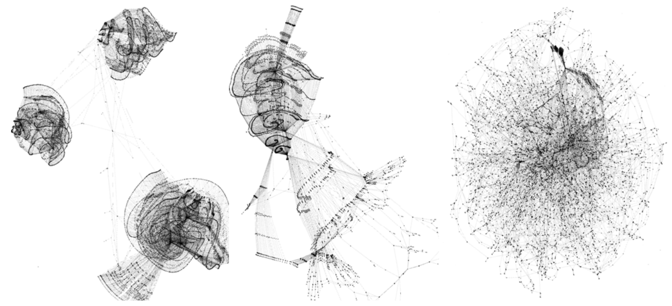


Figure 7: The three DADGs representing *MM.1*, *Dijkstra*, and *Blackscholes* respectively

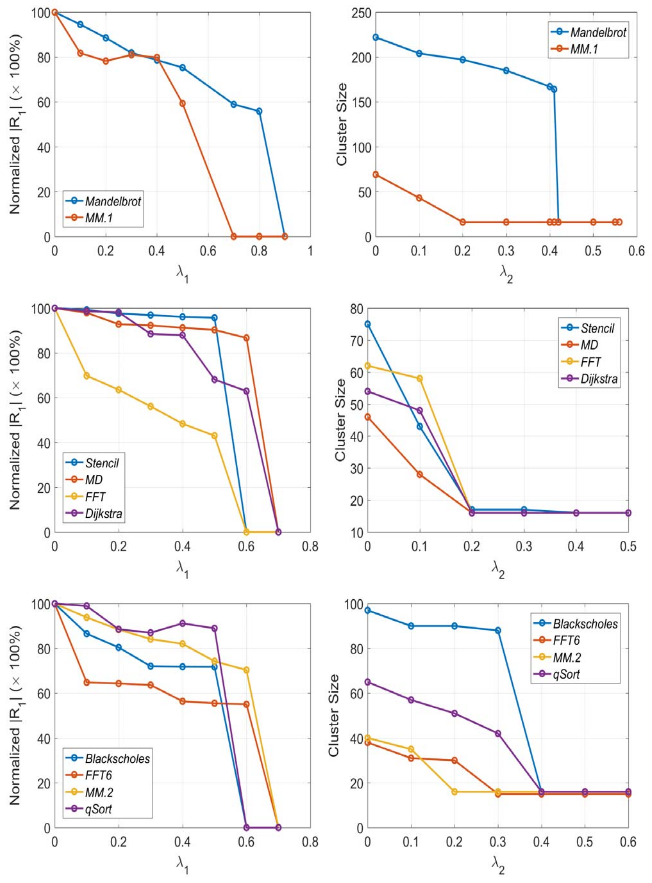


Figure 8: Load balancing and cluster count. Lower is better.

natural that to balance load among clusters to its best, one node is assigned to one cluster and $|R_1| = 0$. Therefore there is no point in increasing the value of λ_1 to the threshold. In the rest of figures, Since in this case N is set to be 0, we want to make sure that all clusters generated by the framework can be fully mapped into NoC by increasing λ_2. All applications, although some have large cluster counts at the beginning, can be confined within core count if proper λ_2 is applied.

3) Comparisons With Sequential Execution and Thread-based Execution: We simulated all 10 applications on the 32-core NoC[4]. In Figure 9, for embarrassingly parallel programs such as *Mandelbrot*, speedup given by thread-based execution is 10% higher than optimization-based execution as threads are independent from each other whereas communities still have load imbalance and inter-community communications. In such a case, each thread can be mapped to different cores to execute faster due to no communication overhead. Nevertheless, for most non-embarrassingly parallel programs, in general, optimization-based execution achieves better performance. Locks and barriers are applied to ensure correctness of multi-threaded programs, giving rise to more synchronization overhead. Besides, on-chip inter-thread or intra-thread communication appears to cause congestion, slowing down the entire program. The optimization-based execution speedup varies from 10.2% to 131.82% when compared to thread-based execution. The degree of data movement influences significantly the overall speedup.

[4]In optimization-based execution, we set both λ_1 and λ_2 to be 0.5.

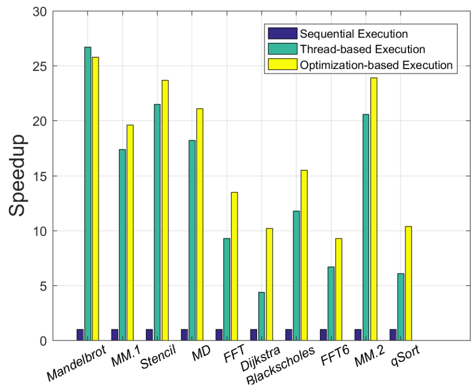


Figure 9: Speedup on the 32-core NoC

4) Scalability: We evaluate the cluster count and cycle counters for the slowest and fastest stages for two applications *FFT* and *qSort* based on different input sizes. In Figure 10, input size is linearly proportional to the cycle counter in the slowest stage and inversely proportional to the counter in the fastest stage. But when adjusting λ_1 to be 0.5, the cycle counter in the slowest stage is reduced by nearly 10% as increasing λ_1 reorganizes the graph by distributing some nodes in a dense cluster into a sparse cluster, taking into consideration the difference between loads in clusters. In terms of cluster size, although input size varies, cluster size only changes slightly, making it easy for λ_2 to reduce the size within core count.

VI. Conclusions

The main goal of this paper is to try to run a sequential program in NoC such that we can gain the maximum speedup in multi-core systems. Considering the deficiencies of multi-threading, we propose a complex network inspired parallelization framework to efficiently partition sequential programs. We first construct the DADG of an application where nodes indicate LLVM IR instructions and edges represent dependencies on virtual registers. Next, we formulate the optimization model by considering not only the inter-cluster edges, but also load balancing and cluster size. We can adjust the load among different clusters and cluster size by fine-tuning the regularization parameters λ_1 and λ_2 used in equation (2). In order to save energy and prevent NoC congestion, data communications are mainly constrained in each cluster. Finally, we construct a CG where nodes denote clusters and edges indicate data dependencies. Having constructed the CG, we propose an algorithm based on topological sort, to identify clusters for parallel execution and map them onto the NoC. Our evaluation with 10 workloads performed on a 32-core

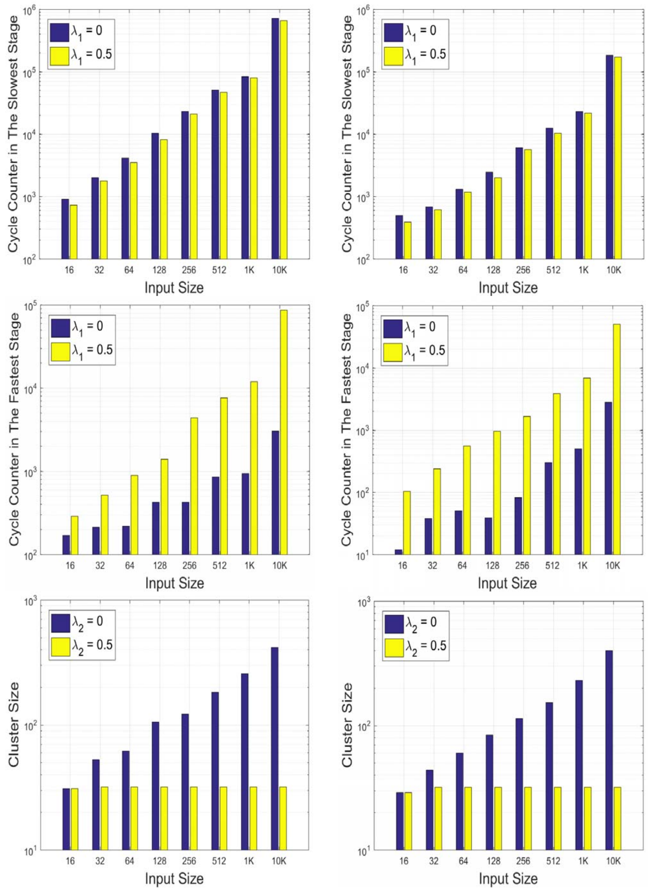


Figure 10: Scalability. *FFT*: left column; *qSort*: right column

NoC shows that load balancing and core count can be alleviated by the framework under various input sizes. Overall speedup of most applications is 10.2% to 131.82% higher compared to the thread-based execution.

VII. Acknowledgement

The authors are thankful to the reviewers for their valuable feedback. The authors gratefully acknowledge the support by the National Science Foundation under CAREER Award CPS-1453860, the U.S. Army Defense Advanced Research Projects Agency (DARPA) under grant no. W911NF-17-1-0076, DARPA Young Faculty Award under grant no. N66001-17-1-4044, the Okawa Foundation Award and the University of Southern California support.

References

[1] C. Bienia et al. “The PARSEC benchmark suite: Characterization and architectural implications”. In: *PACT*. 2008.
[2] P. Bogdan. “Mathematical modeling and control of multifractal workloads for data-center-on-a-chip optimization”. In: *NOCS*. 2015.
[3] P. Bogdan and Y. Xue. “Mathematical models and control algorithms for dynamic optimization of multicore platforms: A complex dynamics approach”. In: *ICCAD*. 2015.
[4] J. Cong et al. “Energy-efficient scheduling on heterogeneous multi-core architectures”. In: *ILSPED*. 2012.
[5] D. Cordes et al. “Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming”. In: *CODES+ISSS*. 2010.
[6] A. Danalis et al. “The scalable heterogeneous computing (SHOC) benchmark suite”. In: *GPGPU-3*. 2010.
[7] G. F. Diamos et al. “Harmony: an execution model and runtime for heterogeneous many core systems”. In: *HPDC*. 2008.
[8] A. J. Dorta et al. “The OpenMP source code repository”. In: *PDP*. 2005.
[9] M. Fattah et al. “Adjustable contiguity of run-time task allocation in networked many-core systems”. In: *ASP-DAC*. 2014.
[10] A. Fonseca et al. “Automatic parallelization: Executing sequential programs on a task-based parallel runtime”. In: *IJPP*. 2016.
[11] S. Fortunato. “Community detection in graphs”. In: *Physics reports*. 2010.
[12] M. R. Guthaus et al. “MiBench: A free, commercially representative embedded benchmark suite”. In: *Workload Characterization, 2001*.
[13] C. Lattner et al. “LLVM: A compilation framework for lifelong program analysis & transformation”. In: *CGO*. 2004.
[14] F. Li et al. “Automatic extraction of coarse-grained data-flow threads from imperative programs”. In: *IEEE Micro*. 2012.
[15] Z. Li et al. “Unveiling parallelization opportunities in sequential programs”. In: *Journal of Systems and Software*. 2016.
[16] M. E. J. Newman. “The structure and function of complex networks”. In: *SIAM review*. 2003.
[17] M. E. J. Newman and M. Girvan. “Finding and evaluating community structure in networks”. In: *Physical review E*. 2004.
[18] B. P. Railing et al. “Contech: Efficiently generating dynamic task graphs for arbitrary parallel programs”. In: *TACO*. 2015.
[19] A. K. Singh et al. “Mapping on multi/many-core systems: survey of current and emerging trends”. In: *DAC*. 2013.
[20] M. A. Suleman et al. “Feedback-directed pipeline parallelism”. In: *PACT*. 2010.
[21] K. S. Vallerio and N. Jha. “Task graph extraction for embedded system synthesis”. In: *VLSI Design*. 2003.
[22] J. A. Winter et al. “Scalable thread scheduling and global power management for heterogeneous many-core architectures”. In: *PACT*. 2010.
[23] Y. Xue et al. “Scalable and realistic benchmark synthesis for efficient NoC performance evaluation: A complex network analysis approach”. In: *CODES+ISSS*. 2016.
[24] Y. Xue and P. Bogdan. “Improving NoC performance under spatio-temporal variability by runtime reconfiguration: a general mathematical framework”. In: *NOCS*. 2016.
[25] Y. Xue and P. Bogdan. “Reliable Multi-fractal Characterization of Weighted Complex Networks: Algorithms and Implications”. In: *Scientific Reports*. Vol. 7. 7487. 2017.
[26] G. L. Yuan et al. “Complexity effective memory access scheduling for many-core accelerator architectures”. In: *MICRO*. 2009.