The Relentless Execution Model for Task-uncoordinated Parallel Computation in Distributed Memory Environments

Lucas A. Wilson
Texas Advanced Computing Center
The University of Texas at Austin
Austin, Texas, U.S.A.
lucaswilson@acm.org

Jeffery von Ronne
Department of Computer Science
The University of Texas at San Antonio
San Antonio, Texas, U.S.A.
vonronne@acm.org

ABSTRACT

This work demonstrates the feasibility of executing tightly-coupled parallel algorithms in a task-uncoordinated fashion, where that tasks do not use any explicit interprocess communication (e.g., messages, shared memory, semaphores, etc.). The proposed model achieves this through the use of dataflow program representation, and the use of a distributed, eventually-consistent key/value store to record intermediate values and their associated state. This work also details a new domain-specific language called StenSAL which allows for simple description of stencil-based scientific applications. Experiments performed on the Stampede supercomputer in Austin, Texas, demonstrate the ability of task-uncoordinated parallel execution models to scale efficiently in both shared memory and distributed memory environments.

1. INTRODUCTION

Modern High Performance Computing (HPC) systems consist of many thousands of individual servers linked together with high bandwidth, low latency interconnection networks. As these systems become larger and more complex it will become increasingly difficult, if not impossible, for their nodes to maintain sufficient availability for worthwhile scientific calculations to be performed.

The thesis highlighted in this work details the formalization and development of a dataflow-inspired model for resilient, task-uncoordinated parallel execution called the Relentless Execution Model, as well as the domain-specific language StenSAL for writing programs for the model.

2. PROBLEM STATEMENT

Current models assume a level of reliability that will not be achievable as systems approach exascale. As such, any execution model that hopes to be effective on exascale systems will need to be capable of handling the loss of processing elements mid-computation, as well as adding elements should more become available (*elasticity*). While some execution models have been developed to make use of elastic, volatile hardware environments (e.g. MapReduce, Hadoop, Azure), their data model is significantly more static than that of traditional scientific simulation codes, which by necessity exchange data frequently in order to find suitable approximations to partial differential equations and other high order mathematical equations.

While techniques aimed at recovering from faults (e.g., checkpoint/restart, redundant processes) have been the subjects of study for decades, they do not address the underlying problem:

Existing execution and programming models are not well suited to future distributed memory parallel systems where hardware reliability cannot be guaranteed.

3. CONTRIBUTIONS

This work has made several contributions to the current state of knowledge with regards to task-uncoordinated parallel program execution:

- This work provides a theoretical model for the execution of parallel algorithms in a task-uncoordinated fashion, which is capable of executing tightly-coupled, data-dependent algorithms in distributed memory environments where hardware volatility may result in occasional-to-frequent loss of computational processes [1–3, 5].
- This work describes a domain-specific language, StenSAL, which allows for rapid, easy-to-understand encoding of explicit stencil algorithms targeting the proposed task-uncoordinated model [4].
- This work describes a series of offline optimizations, which can be performed mechanically by the StenSAL compiler, which enable multi-threaded worksharing and vectorization without explicit input from the programmer [5].
- This work defines a mathematical means of determining the minimum amount of memory (and therefore the minimum number of nodes) required to be available in order for a particular stencil algorithm to maintain computational progress.

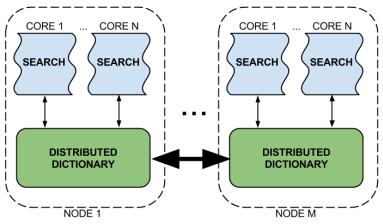


Figure 1: The Relentless Execution Model

4. RELENTLESS EXECUTION MODEL

The Relentless Execution Model (REM) [1–5] is a distributed dataflow model for task-uncoordinated execution on distributed memory parallel architectures where hardware volatility is assumed. REM uses uncoordinated scheduling processes running in parallel to translate dataflow program graphs into individual compute tasks which interact through a distributed, eventually-consistent single-assignment key-value store called the dictionary (see Figure 1).

The advantages of this approach are that new scheduling processes can be added to a computation elastically, and processes which exist on failing hardware can fall out of the execution pool without fatally effecting other processes.

Deterministic, Single-Assignment Tasks. REM works on programs described as explicit data dependency descriptions [3], which consist of a set of deterministic imperative operations chained together by single assignment dictionary labels. By restricting the labels in the dictionary to single assignment and computations to be deterministic, REM can effectively schedule unexecuted tasks without explicit coordination between parallel processes. This is because the only external effect of executing a task is to perform a deterministic computation and update the store to include the computation's result(s). Since the execution of each task is idempotent, there is no need to prevent the same task from being executed by multiple agents.

Uncoordinated Task Execution. REM uses autonomous search agents to locate tasks capable of being executed based only on the data available within the attached distributed dictionary at any given time. This is done by dynamically generating a limited depth subgraph of the unexecuted tasks in the data dependency description and pruning out those branches for which outputs have already been generated. It is possible for tasks to have no input dependencies, thus allowing computation to begin.

REM does not need to generate or store the entire graph, making this solution feasible for use on large-scale applications for which the generation and storage of a complete graph would be impossible (as would be the case for programs requiring conditional loops) or impractical. Each agent repeatedly performs a random walk over the dependency description until all labels in the result set have been associated with values and execution ends. Since the labels generated by the execution of any task must be unique, the existence of a task's output labels provides a guarantee that the task in question has already been executed, and there is no need to re-execute it or its dependencies. This allows a label's existence to be used to prune the graph and reduce search time for subsequent explorations by other agents. This technique, which we call ***cooperative pruning***, is what enables distributed dataflow tasks to perform parallel calculations without the need for explicit task-level coordination.

```
1: for all r ∈ Result do
2:     SOLVE(r)
3: end for
4: function SOLVE(label)
5:     if label ∉ dom(Dictionary) then
6:         task ← PRODUCER(label)
7:         Missing ← {l | l ∈ REQUIRES(task) ∧
                       l ∉ dom(Dictionary)}
8:         for all m ∈ Missing do
9:             SOLVE(m)
10:        end for
11:        Inputs ← {(l ↦ v) | (l ↦ v) ∈ Dictionary ∧
                      l ∈ REQUIRES(task)}
12:        Dictionary ← Dictionary∪
                        COMPUTES(task)(Inputs)
13:    end if
14: end function
```

Figure 2: Pseudocode for REM Search

Agents traverse the graph based solely on the provided dependency descriptions, and only look forward a single level within the dependency description. Each agent autonomously schedules computation to execute without consideration of the current schedules of other agents. Because only the existing set of labels determines which tasks to schedule next, no task-level coordination between agents is necessary. This allows agents to be elastically added/removed from the execution pool without adversely affecting other agents.

REM agents perform this task search by following the algorithm provided in Figure 2. For any appropriate data dependency description, the search algorithm selects at random a label r which is contained within the result set of the problem. It then randomly walks various paths which start with r by repeatedly applying the *producer* and *requires* functions until a task with no inputs or whose inputs already exist in the global dictionary is located. Once the candidate task is identified, the computation associated with that task is performed and its outputs recorded in the dictionary to be used by other processes. The search then walks back up the stack, checking if intermediate labels have been generated. If they have been, it moves up another level, looking for an intermediate label which needs to be generated. When the path has been walked back to the label in the result set, random walks can begin on other result labels.

5. STENSAL

StenSAL is a single-assignment language developed to easily and efficiently build stencil-based codes for evaluating the relentless execution model. StenSAL provides a simple means of expressing explicit stencils as collections of independent tasks tied together with data dependencies.

```
task updateTemp
  creates u(x)(t) as newVal
  for
    x=1:maxX-1
    t=1:maxT
  using
    u(x-1)(t-1) as left
    u(x)(t-1)   as center
    u(x+1)(t-1) as right
  code
    newVal = center + alpha *
      (left + right - 2 * center)
  end code
end task
```

Figure 3: StenSAL task for solving 1-D heat equation using explicit (FTCS) 3-point update

StenSAL programs are collections of independent tasks which are chained together into a directed acyclic graph (DAG) through a series of single-assignment label/value pairs (*labels*). Tasks can range from very fine-grained to very coarse-grained. Within each task, variables can be overwritten as many times as desired. However, the labels used to connect tasks must adhere to the single assignment restriction.

StenSAL uses a natural language-inspired grammar to allow programmers to express task dependency graphs in a very simple manner. Each StenSAL program consists of a program block, which contains a set of tasks that can be used to generate the desired labels. Figure 3 is an example of an internal update for a 1-dimensional stencil algorithm coded in StenSAL.

6. COMPILER-BASED COALESCING

The StenSAL compiler generates C++ code which uses the Relentless Computing API (RCPAI) to describe task outputs, inputs, and reuse across multiple domain values. The StenSAL compiler can identify and optimize simple stencil algorithms in order to more efficient use registers and vector units on modern processors, as well as automatically inject OpenMP 4.1 directives to enable SMP multi-threaded work sharing. The compiler will also fuse label/value pairs into contiguous label/array pairs, which reduces the number of dictionary transactions while simultaneously maximizing network bandwidth.

Figure 4 shows the results of tests where the compiler condenses single-element updates into large $120k^2$ blocks on a single node of Stampede and across multiple nodes. The size was chosen because 120k double-precision values is the maximum storage size of a single value in the currently used key-value storage system (memcached). Distributed processes communicated over Ethernet for these tests. Additional scaling tests were performed for $16k^2$ blocks, which are more efficiently transferred over Gigabit Ethernet (Figure 5).

7. RESILIENT PROGRAM EXECUTION

REM uses stateless search/compute agents to search a distributed dictionary of label/value pairs. These label/value pairs encode both the values emitted by dataflow tasks, but also the state of the task at the time the value was emitted.

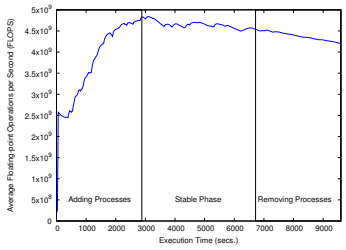


Figure 6: Avg. FLOPS during Elasticity Tests

Elastic Program Execution. Experiments were performed to demonstrate REM's ability to elastically add and remove computational agents in an efficient manner.

Figure 6 shows the average instruction throughput in floating-point operations per second (FLOPS) for the elasticity experiment. The first 2,880 seconds ("Adding Processes") are the insertion phase of the test, the "Stable Phase" is the period when no processes are being added or removed, and the last 2,880 seconds ("Removing Processes") is the phase when processes are being terminated using the `kill` command.

Maintaining Computational Progress. The end goal of REM is to avoid halting the progress of a computation, even in situations where hardware volatility results in the termination of processes. For our purposes, *computational progress* is the ability of an algorithm to progress from one state to the next. As an example, a time stepping simulation must progress from time step k to time step $k+1$.

This means that the dictionary D must contain enough space to store a sufficient quantity of label/value pairs that, given all labels generated by tasks at time step k ($\mathcal{T}_k$) exist in D, all labels generated by tasks at time step $k+1$ ($\mathcal{T}_{k+1}$) can be generated, even if k step pairs are replaced by $k+1$ step pairs, assuming all tasks emit a single label/value pair upon completion. In short, the minimum size of the dictionary $\lfloor D \rfloor$ is less than the space required to store all label/value pairs in time step k and time step $k+1$.

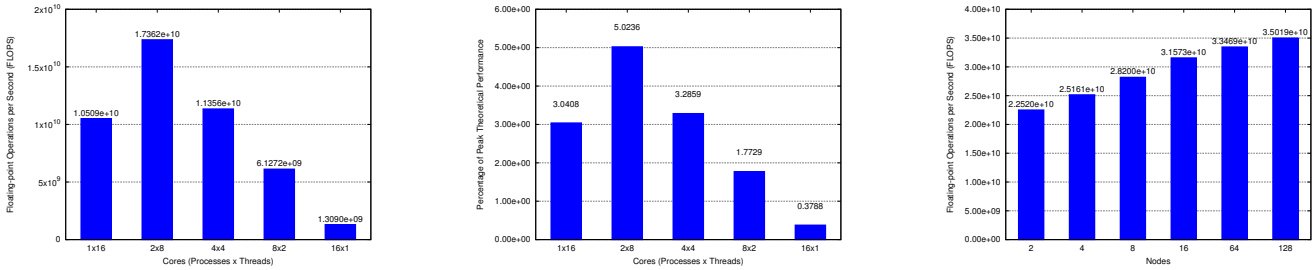
We define $\lfloor D \rfloor$ as:

$$\lfloor D \rfloor = \alpha \prod_{i=1}^{|c|} \left(\mid \mathcal{T}_{k+1}^{c_i} \mid + \Gamma \right)$$

$$\Gamma = \max_{\tau \in \mathcal{T}_{k+1}} deg^-(\tau)$$

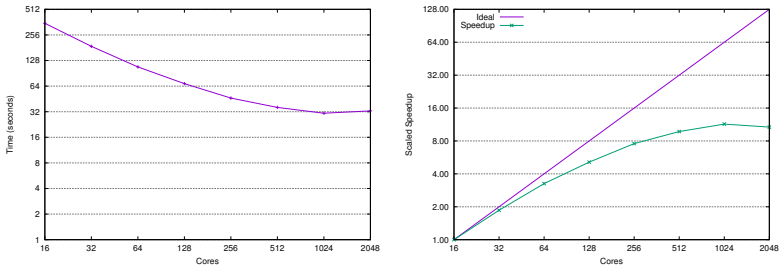
Here, $\mathcal{T}_{k+1}^{c_i}$ is the set of tasks when traversing the i^{th} dimension of coordinate tuple c for sequencing index $k+1$, Γ is the retention factor (which is defined as the maximum in-degree of any task τ in time step $k+1$), and α is an additional term added for curve fitting.

Figure 7 shows the amount of recomputation the occurs for different restricted dictionary sizes. The vertical bars show



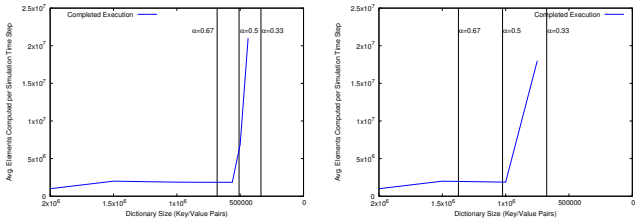
(a) Single-node FLOPS (b) Single-node % of Peak (c) Multi-node FLOPS

Figure 4: REM Performance Using Coalesced ($120k^2$) Tasks



(a) Wall Time (b) Scaled Speedup

Figure 5: REM Performance Using Coalesced ($120k^2$) Tasks



(a) 9-point 2-D stencil (b) 27-point 3-D stencil

Figure 7: Results of Dictionary Restriction Tests

the calculated points of failure for different α values.

As can be seen in Figure 7, both stencils fail when the α value approaches one third. The reasoning for this is under continued investigation, but our current theory is that this is related to the width of the stencil (3 elements per dimension). Future experiments will test this hypothesis.

8. CONCLUSION

This work has demonstrated that it is possible to effectively and efficiently execute tightly-coupled parallel algorithms in distributed memory environments without the need for explicit task coordination. As a result, computational tasks can be easily added and removed from the execution pool without forcing program abort and restart from prior checkpoint.

The proposed model allows for continuous execution of parallel algorithms in environments where hardware volatility can be expected, including cloud environments, volunteer computing environments, and exascale HPC systems.

9. REFERENCES

[1] Lucas A Wilson and John A Lockman III. Poster: The relentless computing paradigm: a data-oriented programming model for distributed-memory computation. In *Proceedings of the 2011 companion on High Performance Computing Networking, Storage and Analysis Companion*, pages 53–54. ACM, 2011.

[2] Lucas A. Wilson and John A. Lockman III. Relentless Computing: Enabling fault-tolerant, numerically intensive computation in distributed environments. In *Proceedings of the 2011 International Parallel and Distributed Processing Techniques and Applications Conference (PDPTA'11)*, 2011.

[3] Lucas A. Wilson and Jeffery von Ronne. A distributed dataflow model for task-uncoordinated parallel program execution. In *Parallel Processing Workshops (ICPPW), 2014 43rd International Conference on*, pages 321–330, Sept 2014.

[4] Lucas A. Wilson and Jeffery von Ronne. Stensal: A single assignment language for relentlessly executing explicit stencil algorithms. In *Proceedings of the Second Workshop on Optimizing Stencil Computations*, WOSC '14, pages 17–24, New York, NY, USA, 2014. ACM.

[5] Lucas A. Wilson and Jeffery von Ronne. A task-uncoordinated distributed dataflow model for scalable high performance parallel program execution. *Parallel Computing: Systems and Applications*, to appear.