INCA: In-Network Compute Assistance

Whit Schonbein∗†
SANDIA National Laboratories
Center for Computational Research
Albuquerque, New Mexico
wwschon@sandia.gov

Matthew G.F. Dosanjh
SANDIA National Laboratories
Center for Computational Research
Albuquerque, New Mexico
mdosanj@sandia.gov

Ryan E. Grant
SANDIA National Laboratories
Center for Computational Research
Albuquerque, New Mexico
regrant@sandia.gov

Dorian Arnold
Department of Computer Science
Emory University
Atlanta, Georgia
dorian.arnold@emory.edu

ABSTRACT

Current proposals for in-network data processing operate on data as it streams through a network switch or endpoint. Since compute resources must be available when data arrives, these approaches provide deadline-based models of execution. This paper introduces a deadline-free general compute model for network endpoints called INCA: In-Network Compute Assistance. INCA builds upon contemporary NIC offload capabilities to provide on-NIC, deadline-free, general-purpose compute capabilities that can be utilized when the network is inactive. We demonstrate INCA is Turing complete, and provide a detailed design for extending existing hardware to support this model. We evaluate runtimes for a selection of kernels, including several optimizations, and show INCA can provide up to a 11% speedup for applications with minimal code modifications and between 25% to 37% when applications are optimized for INCA.

1 INTRODUCTION

As network speeds increase to hundreds of gigabits per second and beyond, message processing overheads have become increasingly onerous—sufficiently so that many manufacturers now provide network interface cards (NICs) with offload processing capabilities. These capabilities typically include DMA, hardware for performing MPI-style message matching based on rank and tag addressing, line-rate atomic operations (for example, sum or compare-and-swap), and support for issuing new messages independently of a host application (‘triggered operations’). All of these capabilities operate independently of the host processor, significantly reducing latencies associated with communication by admitting parallelism in the form of ‘communication-computation overlap’.

Researchers have proposed stream-based approaches that leverage general-purpose compute cores and instruction level parallelism in combination with network offloading [20]. For example, in the ‘stream processing in network’ (sPIN) approach, users write and load kernels onto the NIC. A header kernel inspects incoming messages, and when payload processing is required, each subsequent packet is directed to a compute core where it is processed by a packet kernel. The results are written to memory via DMA. By working on data where it resides rather than moving it to the host CPU for processing, approaches like sPIN can achieve speedups for many use cases. However, fundamentally, kernel computation complexity is limited by the fact compute cores must be available to process packets when they arrive. For example, for a data rate of 200Gb/s, 64B packets, and 32 2.5GHz cores with 1 IPC, a stream-based approach is limited to packet-processing kernels of less than 500 instructions. This instruction limit can be increased by adding cores, introducing additional message processing latency through buffering, or increasing packet size (for example), but doing so does not remove the deadline.

We propose In-Network Compute Assistance (INCA), an alternate and complementary approach for enabling general-purpose, on-NIC compute capabilities. INCA is built on top of a computational model that directly leverages the matching, atomic, and triggered operation capabilities mentioned previously. In contrast to stream-based approaches, this model is deadline-free and preemptible, and can process data even when the network is idle. This flexibility typically comes at the cost of serial compute speed as a NIC’s operation speed is typically limited to that required by the network data rate.
However, INCA provides opportunities for significant data parallelism that can more than make up for lost serial computational speed. Under INCA, then, NICs may be viewed somewhat analogously to co-processors in the model of compute assistance they can provide to host processors.

In this paper, we make the following contributions:

1. The design of a new, deadline-free preemptible general computational model for network endpoints;
2. A demonstration that this new model can be made Turing complete;
3. An architecture for a NIC capable of providing this new type of computation;
4. A quantitative evaluation of various use cases, including the potential speedups for a range of kernels and applications.

We describe an INCA implementation that requires minimal changes to existing NIC architectures and propose multiple architectural enhancements that can render speedups of over 7200× for some computational kernels. We show that INCA can provide between 25% to 37% application speedup for three widely representative mini-applications and one production molecular application. In summary, we show that optimized and even basic INCA-based kernels can yield excellent performance that in turn significantly reduces application runtimes.

In the remainder of this paper, we elaborate the details: after surveying the relevant background on in-network processing (Section 2), we present the ‘triggered operation machine’ compute model and demonstrate it can be made Turing complete given the appropriate atomic operations (Section 3). Guided by these results, we provide an implementation of this model – INCA – on the Portals 4.0 networking programming API, with only minor changes to the specification (Section 4). Then using a LogGP-based model, we assess runtimes for a variety of kernels, including data-parallel versions, and applications (Section 5). Finally, we conclude with a discussion of common questions regarding INCA.

## 2 BACKGROUND AND RELATED WORKS

Network architects first introduced offload or in-network processing to minimize CPU involvement in packet processing, and to mask packet processing overheads by overlaying them with other CPU computations. Modern network technologies with sophisticated packet processing engines (PPEs) include Mellanox’s Bluefield [23] devices (SoC-based), Broadcom’s Stingray SmartNICs [6], and other technologies like CORE-Direct [16], SHARP [15], FlexNIC [21], Azure SmartNICs (FPGA-based) [14], Mellanox Innova FPGA SmartNICs and sPIN [20]. These modern PPEs succeed a previous generation of such solutions from the early 2000s like Myrinet [4], Quadrics [24] and early Portals [2] NICs like Seastar [5].

While PPE technologies whether they are SoC-based like SmartNIC and Bluefield or FPGA-based like Azure and Innova SmartNICs, have been used for higher-level stream processing, Ethernet-based PPEs have been used to offload processing for lower-level network protocols like TCP and UDP [8, 10, 11, 17, 27]. Researchers have also explored the performance impacts of offloading collective operations [7, 18, 31], with recent work centering on generalizability [28] and offloading at the NIC [16] and switch [15]. Most recently, researchers have proposed using offload capacities for purposes other than originally intended, e.g., using a hardware matching engine to offload key-value store processing [22]. The latter solutions are interesting in that they illustrate the possibility of exploiting existing NIC offload features for novel ends, but do not aim for general-purpose compute capacities as is the case with INCA.

### 2.1 PPE Deadlines

The works surveyed above share a common strategy of processing data streams as they reside on a NIC. This has two implications. First, when there is no data flowing into a NIC, no work is performed, i.e., no processing occurs when the network is idle. Second, when data does arrive, computational resources must be available to process that data. Consequently, these approaches work under an implicit deadline determined by network speed, buffer availability, number of compute elements, etc. Failing to meet the deadline means the task has failed, and may invalidate the data stream where the failure occurred, resulting in an unrecoverable, undefined data state, retransmission of data, or even termination of the application. Because of this deadline, the computational complexity (e.g., number of instructions) of the work NIC compute resources can do for any given flow is limited.

PPE deadlines are potentially problematic for kernel developers. As network data speeds increase, kernel execution speeds must also increase, so PPE deadlines become shorter. For example, recent PPEs typically comprise a collection of ARM processors. With many cores, a device may be able process 32 packets simultaneously, meaning that the deadline is the time to process a packet through the NIC times 32. Assuming 2.5 GHz cores with an IPC of 1, 256 byte packets, and a 200Gb/s network, each core can execute at most 819 instructions before it must be available to process the next packet. As network speeds increase, we cannot reasonably expect core clock speeds or the number of cores to increase proportionally. Consequently, as network speeds increase and deadlines decrease, packet processing kernels will not necessarily be forward-compatible.

One way to avoid PPE deadline problems is by eschewing general-purpose cores in favor of efficient yet task-specific hardware [2, 12]. For purpose of this work, we emphasize three such specialized offload capacities. First, some systems provide the capacity to perform atomic operations (summation, compare-and-swap, etc.) on a single incoming operand and local operand. These basic operations provide building blocks for offloading reductions, distributed mutexes, etc. Second, hardware support may be provided to assist in message matching, as required by messaging libraries such as MPI [32]. Finally, hardware supporting triggered operations – i.e., outgoing messages that are generated automatically in response to events (e.g., buffer updates) caused by incoming traffic – provides primitives for constructing offloaded collectives and (potentially) rendezvous messaging [3, 28, 31].

In this work, we address the fundamental and practical constraints of deadline-based approaches by introducing a generalized, deadline-free model built on top of these specialized offload capacities. Being deadline-free, our INCA model can admit more complex computational kernels. Additionally, to the best of our knowledge, INCA is the first network offload model that supports preemptible kernels. Lastly, unlike previous approaches, INCA allows NIC PPEs to be used even when the network is idle, that is one can leverage the network’s computational capabilities for non-network computations.
3 ENABLING GENERAL-PURPOSE COMPUTATION

Contemporary network hardware typically provides offload support for (i) message demultiplexing (e.g., MPI tag matching), (ii) atomic operations, and (iii) generating new messages contingent on buffer modifications (triggered operations). Although these capabilities can be realized on a general-purpose CPU, current hardware also exists using purpose-built ASIC circuitry [12]. Since we are proposing these primitives can be leveraged to provide general-purpose compute capabilities (i.e., without relying on standard CPUs), in this section we offer a proof of this claim. Specifically, we present the triggered operation machine (TOM), a novel computational model based on these capacities. TOM serves two purposes: first, it demonstrates that when properly organized, these primitives are indeed capable of general-purpose computing (i.e., are Turing complete); and second, by specifying the basic capabilities required of any practical realization, the model serves as a guide for the implementation given in section 4.

3.1 The model

In a typical NIC with matching, atomic, and triggered operation capabilities, an incoming message is interpreted (e.g., through a tag); this can result in an atomic operation being applied to its contents and local data, and a triggering signal generated. If there are additional operations attached to this signal, new messages are generated (figure 1). The basic strategy behind the TOM is to treat pairs of generated messages and atomic operations (linked by unique tags) as primitive instructions, where these instructions share a common trigger signal functioning as a program counter.

Formally, an incoming message \( m_j \) is a pair comprising an operand \( (o_j) \) and a tag \( (\mu_j) \). The atomic unit (AU) maintains a list of 4-tuples, each comprising a tag \( (\mu_i) \), an operand \( (o_i) \), an atomic operation (\( \alpha_i \)), and a trigger \( (T_i) \). The triggering unit (TU) also maintains a list of 4-tuples, each comprising a trigger \( (T_k) \), a threshold \( (\theta_k) \), an operand \( (o_k) \), and a tag \( (\mu_k) \).

In this notation, operands, triggers, and out are locations, and angled brackets designate the contents of those locations. Let the set of tuples of the AU be \( A \). Given an incoming message \( m_j = (o_j, \mu_j) \), the operation of the AU can be represented as updating the state of all operands \( o_i \) referenced by the members of \( A \):

\[
\forall a_i \in A : \langle o_i \rangle \leftarrow \begin{cases} \alpha(\langle o_i \rangle, \langle o_j \rangle) & \mu_i = \mu_j \\ \langle o_i \rangle & \text{otherwise} \end{cases}
\]

(1)

Unary operations are accommodated by operating only on \( o_i \), ignoring \( o_j \).

The contents of the trigger specified in the AU tuple are updated according to some function; here we use the successor function to count the number of atomics initiated using the same trigger, where \( R \) is the set of triggers:

\[
\forall R_i \in R : \langle R_i \rangle \leftarrow \begin{cases} \langle R_i \rangle + 1 & \mu_i = \mu_j \\ \langle R_i \rangle & \text{otherwise} \end{cases}
\]

(2)

Finally, letting \( T \) be the set of 4-tuples of the TU, the operation of the TU can then be represented as updating the state of out based on the status of the triggers identified by each tuple:

\[
\forall t_k \in T : \langle \text{out} \rangle \leftarrow \begin{cases} \langle o_k, \mu_k \rangle & \theta_k = \langle T_k \rangle \\ \epsilon & \text{otherwise} \end{cases}
\]

(3)

3.2 Turing completeness

To establish what is required of the TOM to secure Turing completeness, we reduce the well-known universal register machine (URM) model to TOM [29]. A simple URM is defined as follows. First, let \( R = \{r_0, r_1, \ldots\} \) be an unbounded set of registers, each capable of storing a member of \( \mathbb{N}_0 \). Second, the behavior of the machine is determined by a finite list of indexed instructions \( I = \{i_1, i_2, \ldots, i_k\} \), each of which is one of the following four possibilities, where \( r_i, r_j \in R \):

(1) Zero: \( Z(r_i) \): \( \langle r_i \rangle \leftarrow 0 \).

(2) Successor: \( S(r_i) \): \( \langle r_i \rangle \leftarrow \langle r_i \rangle + 1 \).

(3) Transfer: \( T(r_i, r_j) \): \( \langle r_i \rangle \leftarrow \langle r_j \rangle \).

(4) Jump: \( J(r_i, r_j, q) \), where \( i_q \in I \): letting \( p \) be the index of the current instruction, if \( \langle r_i \rangle = \langle r_j \rangle \), jump to instruction \( i_q \); otherwise, proceed to instruction \( i_{p+1} \).

For all instructions but jump, the instruction index is incremented by one after the instruction is executed. The machine begins at instruction \( i_1 \), and continues until there is no instruction with the current index to execute. The result of the computation is the contents of the registers.

URMs are reduced to the TOM as follows. First, for each URM register \( r_0, r_1, \ldots \) designate a corresponding TOM operand location \( o_0, o_1, \ldots \). Second, let all AU and TU tuples utilize the same trigger, \( R_{\text{PC}} \). \( R_{\text{PC}} \) can be viewed as a program counter storing the index of the current URM instruction in the list of instructions comprising a URM program. Third, each TOM operation has two parts: an entry in the TU and a matching entry in the AU (connected by the message generated by the TU). A TOM instruction \( i_j \) is therefore a pair \( (t_k, a_i) = ((\theta_k, R_{\text{PC}}, o_k, \mu_k), (o_i, \mu_i, a, R_{\text{PC}})) \). This can be simplified as follows. First, since the tag for a message cannot vary between the AU tuple and the TU tuple, \( \mu_i = \mu_k \), so can be dropped. Second,
since all instructions use the same trigger, it is also omitted. Finally, rearranging terms for clarity and dropping superfluous parentheses gives a simplified definition: $ij = (\theta_k, o_i, o_k, \alpha)$. Fourth, a TOM program is defined as a list of $n \geq 1$ TOM instructions $i_1, i_2, \ldots, i_n$ beginning with $\theta_k = 1$, and concluding with $\theta_k = n$. Since under this definition, $\langle \theta_k \rangle = j$, the TU threshold can be omitted, and instruction $j$ of a TOM program – corresponding to instruction $j$ of a URM program – is represented simply as: $ij = (o_i, o_k, \alpha)$.

The remainder of the reduction involves the selection of an appropriate set of primitive operations for possible values of $\alpha$. Recall incoming messages are PUTs from location $o_j$ to location $o_i$, where the contents of the latter are replaced with the contents of some atomic operation performed on one or both operands. Consequently, if $\alpha$ is identity, the result is merely $⟨o_i⟩ ← ⟨o_j⟩$. Therefore, for each URM instruction of the form $Z(r_i)$, let the corresponding TOM instruction be $(o_i, 0, \Rightarrow)$; and for each URM instruction of the form $T(r_i, r_k)$, let the corresponding TOM instruction be $(o_i, o_k, \Rightarrow)$.

Addition is a standard atomic operation in current hardware, so the URM successor function can be simulated by letting $\alpha$ be addition, and setting $ ⟨o_k⟩ = 1$. Each URM instruction of the form $S(r_i)$ thus has a corresponding TOM instruction $(o_i, 1, +)$.

To reduce the URM jump instruction, first note that compare-and-swaps is a typical atomic operation, so testing whether the contents of two operand locations are equal is a reasonable atomic operation for the TOM. Consequently, the equality test of a URM jump can be simulated by a TOM instruction of the form $(o_i, o_k, \Leftrightarrow)$, where $⟨o_i⟩ \Leftrightarrow ⟨o_k⟩$ is defined as $⟨o_i⟩ \Leftrightarrow \left\{ \begin{array}{ll} 1 & \text{if } o_i = o_k \\ 0 & \text{otherwise} \end{array} \right.$ Second, since triggers are ‘registers’ in the same sense as operand locations, we define a special atomic operation that takes advantage of this fact: $[o_i, q, > 0]$, interpreted as follows:

$$\langle R_{PC} \rangle \left\{ \begin{array}{l} q \\ (R_{PC}) + 1 \end{array} \right. \begin{array}{l} \text{if } o_i > 0 \\ \text{otherwise} \end{array} \right. \quad (4)$$

Note that since each URM jump results in two TOM instructions, the values of $q$ and $j$ for all instructions must be adjusted appropriately. To simplify exposition, we exclude these details.

Table 2 summarizes the reduction of URMs to TOMs. This reduction tells us the TOM model can be made Turing complete if, at the minimum, it supplies atomic operations for identity (i.e., PUT), addition, and equivalence, as well as the capacity to conditionally modify the contents of a location (the trigger) depending on whether the contents of another are greater than zero. In the next section, we use these requirements to guide the construction of a concrete implementation of the TOM model.

### 4 IMPLEMENTATION

As a proof-of-concept, we implemented the TOM model on top of the Portals network programming interface [2]. Portals is designed for implementation in hardware, is supported by at least one current vendor [12], and includes all the core building blocks of the TOM: atomic operations, message matching, semantics, and triggered operations. However, several modifications are necessary to fully enable general-purpose computation under the TOM model.

#### 4.1 Modifying Portals

Figure 2 illustrates Portals functionality. Each communicating host process is associated with a Portals table entry (PTE). Incoming messages are demuxed to host processes by matching against these entries. Attached to each PTE is a match list comprising a collection of match elements (MEs), each of which specifies a set of match bits, a buffer, and an optional event counter. If the match bits of an incoming message match those of an ME, an optional atomic operation is performed using the contents of the incoming message and those of the ME buffer, and the result is written to the ME buffer via DMA. The modification to the buffer contents is noted by incrementing a counter, and associated with each counter is a collection of triggered operations, each with a threshold. For each triggered operation, if the new value of the counter is greater-than-or-equal to its threshold, the operation is triggered, generating a new message on the wire.

For example, a simple use case is offloaded reduction. If a node has $k$ children, $k$ appropriate MEs can be registered, each with the same buffer and counter. Likewise, a triggered operation (e.g., PUT_ATOMIC with a sum operation) is posted, with a threshold of $k$, and the processes’ parent as the destination. As data arrives from each child, the value of the buffer is modified, and the counter incremented. When the counter reaches $k$, all child data has been received, the triggered operation is discharged, and the Portals system automatically sends the result to the parent.

Much of the TOM model maps straightforwardly onto Portals. A TOM instruction corresponds to a pair of a triggered operation ($PtlTriggeredPut$) and a matching ME, linked by unique match bits. All instructions (i.e., all MEs and triggered operations) use the same trigger, i.e., a program counter. One TOM operand ($o_i$) is specified as the ME buffer, and the other ($o_k$) is given by the message generated by the triggered operation. TOM transfers and the zero operation are thus standard PUT operations. The ME can also specify an atomic
Algorithm 1 Dot Product

1: \( k \leftarrow \text{length}(A) \)
2: \( i \leftarrow 0 \)
3: \( \text{while } i < k \text{ do} \)
4: \( c \leftarrow c + A[i] \times B[i] \)
5: \( i \leftarrow i + 1 \)
6: \( \text{end while} \)

operation \( (\alpha) \) to be performed. Note that by having MEs specify atomics, INCA departs from typical atomic operation semantics in that the receiver (rather than the sender) determines the operation to be performed. The atomics provided by Portals include addition, so it is trivial to simulate the URM successor function; however, rather than limit ourselves to URM operations, our implementation allows \( \alpha \) to be any of the atomics provided by the Portals specification. In addition to arithmetic operations (addition, subtraction, multiplication), these include bitwise, logical, and comparison operators.

Fully implementing TOM on top of Portals requires three substantive modifications to the Portals specification. First, a Portals triggered operation is discharged when its threshold is \( \leq \) the counter value; we modify this behavior to allow ‘strict’ indexing (i.e., \( = \)), as required by the TOM model (equation 3). Second, while MEs can be used multiple times, triggered operations are by default ‘use once’. Since TOM supports backwards jumps, we extend Portals to include persistent triggered operations (Portals already allows for persistent MEs). Finally, while Portals provides an atomic for testing equality, it does not support modifying a counter value conditionally on the outcome of such a test. To address this, we introduce a new atomic operation, ‘branch if \( \leq \) zero’. An ME specifying this atomic also provides an instruction index. If the value contained in the operand identified by the ME is zero or less, the counter is updated to this instruction index; otherwise, it is incremented by one (equation 4).

4.2 INCA code

When extended as just described, Portals provides a ‘microcode’ for performing on-NIC program execution, where these programs are sequences of instructions as defined by the TOM model. Since these programs provide in-network compute assistance to host programs, we refer to them as INCA codes.

For example, Algorithm 1 shows pseudocode for calculating a dot product, and the same algorithm rendered in INCA is shown in Algorithm 2. Microcode instructions have a standard syntax, where the first argument is the instruction location, the following argument (unary operation) or arguments (binary operation) specifies locations of operands, and the final argument is the type (which corresponds to one of the primitive datatypes of Portals, exclusive of complex numbers). The exceptions are the branching instructions \( \text{JMP} \) and \( \text{BLEZ} \), which update the program counter.

In Algorithm 2, instruction 1 initializes \( i \) to 0 via a local \( \text{PUTL} \). Instructions 2–4 evaluate the loop condition by transferring the current contents of \( i \) to a temporary location \( r_0 \) (instruction 2), evaluating whether \( (r_0) \) is less than \( k \) (instruction 3), and conditionally adjusting the program counter to 5 or 10 depending on the result of that evaluation (instruction 4). Instructions 5–9 comprise the loop body.

Note that in this code, binary operations (\( \text{LT} \), \( \text{MUL} \)) are represented as having three arguments – a destination location and the locations of two operands. However, Portals atomics utilize two locations,
The memory fetching penalty is imposed once for unary operations (because the operand specified in the ME must be retrieved), and twice for binary operations (because the operand in the triggered PUT must also be fetched). No fetches are required for JUMP and BLEZ since the data necessary for evaluation is available during matching.

Our second model configuration, scratchpad, accounts for several optimizations, as diagrammed in Figure 4. First, a fast path is available for message loopback, eliminating the trip through the switch and providing buffering (in the form of a low-priority queue) for the pending operation such that incoming data transmissions can take precedence over executing INCA programs. Second, we assume the Ports system is extended with a scratchpad memory similar to fast cache (SRAM) as employed in contemporary stream models [6, 20]. This memory provides a low-latency, NIC-local space for caching current data, and operands specified by an INCA program can be pre-loaded, e.g., by traversing the matching list.

In both configurations, Ports overhead and gap are bounded by network speed, because a Ports-enabled NIC is expected to process incoming requests and issue outgoing data at least at message rate [31]. Figure 5 shows the ideal processing speeds at network bandwidths targeted by the InfiniBand roadmap. The three message sizes correspond to 8B operands, 48B messages containing the minimum Ports header information (according to the current Ports reference implementation) and an operand, and the common 64B cache line. Message rates reported by Intel (100Gbs OmniPath) and Mellanox (100Gbs EDR and 200Gbs HDR) are shown in colored
triangles. For current hardware, Portals overhead can therefore be expected to fall somewhere between 5ns (gap for 200M messages/s and 1.76ns (gap for 44B messages). For the present evaluation, we use the conservative larger overhead value.

Figure 5: INCA instruction rates for 8, 44, and 64 byte packets, and message rates for several current network products.

Configuration parameters are summarized in Table 4. Loopback latency for base reflects the time to traverse the local switch, and Portals overhead is assessed as just described. Memory latency for base estimates the cost of retrieving an operand via gen 5 x32 PCIe [30], while scratchpad memory latency is based on the value adopted in [20].

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>Scratchpad</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loopback latency</td>
<td>50 ns</td>
<td>0</td>
</tr>
<tr>
<td>Portals overhead</td>
<td>5 ns</td>
<td>5 ns</td>
</tr>
<tr>
<td>Memory latency</td>
<td>250 ns</td>
<td>1 ns</td>
</tr>
<tr>
<td>gap</td>
<td>5 ns</td>
<td>5 ns</td>
</tr>
</tbody>
</table>

Table 4: Model parameters.

5.3 Performance Results

Using the LogP performance model, parameters and configurations described in the previous sections, we first calculated INCA runtimes for the base and the scratchpad configurations. We then calculated runtimes for a series of optimizations. These optimizations are presented incrementally so that benefits of each can be assessed independently, e.g., as regards potential implementation costs. Results for all configurations and optimizations are collected in Table 5.

5.3.1 Baseline Performance. Our base configuration most reflects INCA performance if current generation HPC networks were extended to support the TOM model as previously described. Perhaps unsurprisingly, the base runtimes range from slow (42 µs for 128 B matrix transpose) to very slow (100 ms for 8192 B matrix multiplication). While below we present a series of significant optimizations, we observe that even the performance of the base configuration may not be problematic. An application running on the host with sufficient latent parallelism still can take advantage of relatively slow compute assistance. Moreover, since INCA programs are intrinsically preemptible, a host application can always assume control over NIC compute resources. For instance, Portals includes an event notification queue that can be polled by host applications to determine the status of posted requests. This event queue can be used to provide information regarding the current state of an executing INCA program. Consequently, given a mechanism for signaling the NIC to halt INCA program execution, an idle host could steal partially-completed work from the NIC, picking up where the INCA program left off.

5.3.2 Basic Optimizations. We considered two basic optimizations to our INCA kernels: utilizing scratchpad memory and working on data in place by minimizing data copying.

Scratchpad Memory. Our scratchpad configuration assumes an INCA NIC is equipped with a local scratchpad memory to minimize memory access penalties, and a fast loopback path to avoid routing instructions through the switch. Unsurprisingly, enabling these features has a significant performance impact, reducing runtimes by one or two orders of magnitude across all kernels considered.

Clobbering. As noted, atomic operations overwrite the contents of the first operand, so, to preserve standard program semantics, INCA programs copy operands before operating on them. However, this data copy may introduce unnecessary overhead. Furthermore, working in place has been shown to be a useful optimization in research on processing-in-memory. We modified several kernels to work in place by utilizing traditional atomic semantics, when possible. For example, we modify the dot product kernel (Algorithm 2) to directly use and modify the vector’s contents. (Some copying is still necessary since overwriting the contents of the index during the loop condition evaluation would invalidate the loop.)

Table 5 shows runtimes for the kernels that could be modified straightforwardly modified to allow clobbering, namely dot-product and hadamard-product. In comparison to scratchpad, which shows an average speedup relative to base of 30.0×, and 32.7× for the two kernels, respectively, clobber brings this speedup to 38.13× and 37.18× (1.27× and 1.14× speedup with respect to scratchpad), demonstrating that when possible, avoiding data movement through clobbering is an effective optimization.

5.3.3 Parallelization-based Optimizations. While the "basic" optimizations we propose promise significant boosts to INCA kernel performance, reasonable hardware enhancements are also promising. In particular, ALUs of the sort utilized by a Portals NIC occupy approximately 1/8th of the die space typically occupied by the compute logic circuitry of a modern CPU core [13]. Consequently, ALUs may be used in parallel to provide SIMD or simple MIMD functionality.

As an illustrative parallelization example, consider the INCA microcode for dot product in Algorithm 2. We can utilize SIMD multiplication instructions for the innermost loop (line 6). This moves the multiplication outside of the that loop, substituting \([k/A] SIMD multiplications – where \(A\) is the width of the SIMD instruction – for the original \(k\) single-operand instructions. The innermost addition cannot be SIMD-parallelized easily because it accumulates to a single memory location, so the loop is not entirely eliminated. The resulting parallel algorithm for inputs with 256 or fewer operands is shown in Algorithm 3. The contents of array \(A\) are copied to a temporary location (Line 1), \(T\), which is then clobbered (Line 2) by a new SIMD INCA instruction, MULM (‘multiply multiple’). The remaining code accumulates the result.

We parallelized dot-product and hadamard-product in the manner just described. We also parallelized matrix-multiplication
Algorithm 3 INCA Parallel Dot Product (n ≤ 256)

1: PUTLM T[0], A[0], f, 256
2: MUL T[0], T[0], B[0], f, 256
3: PUT L i, 0, i16
4: LT r0, r0, b, i16
5: BLEZ r0, 9
6: ADD c, c, T[i], f
7: ADD i, i, i16
8: JMP 3
9: END

under the assumption data is provided in a format that facilitates SIMD operations, e.g., the second input matrix is transposed, and distinct matrices corresponding to rows of the first input matrix are provided (so that the Hadamard product can be used to concurrently multiply all columns by a row). For comparison (see below), we also implemented parallel versions with clobbering for each of these three kernels, which eliminates the need to copy vectors or matrices to temporary locations. The other kernels were excluded from parallelization considerations: matrix-transpose and matrix-unpack are primarily data-movement kernels. filter is parallelizable assuming a ternary compare-and-swap operation; currently this is not an available atomic operation under Portals. linear-interpolation contains a backwards dependency, and convolution invokes non-trivial logic for handling edge data.

Modeling Hardware Optimizations. To consider parallelization, we extended the INCA interpreter to include SIMD versions of arithmetic instructions (MULM, ADDM, etc.). To estimate execution time, we incorporated inter-byte gap for 'multiple' instructions (G, in the LogGP model [1]), because those instructions involve payloads with multiple operands. Runtime calculations use whichever parameter (G or g) gives a longer execution time.

We assume 256 parallel ALUs, corresponding to approximately half the die space of a common 32-core CPU, leaving enough die space for NIC logic, processing pipelines and other miscellaneous circuitry. For payloads with more than 256 operands, multiple instances of the INCA SIMD instructions are used, unrolled rather than iterated. A diagram of the proposed SIMD parallel ALU design is shown in Figure 6. It should be noted that due to the reactive nature of computation used by INCA, data for the local operands can be staged to scratchpad or even local ALU cache (when predictable) prior to the remote operands arriving in network messages. Local operand caches can be small (a few operands) up to a size of a few KiB. For example, a reasonable design may use 4KiB caches. This would typically require at least a 1MiB scratchpad to feed 256 ALU operands with more than 256 operands, multiple instances of the INCA SIMD instructions are used, unrolled rather than iterated. A diagram of the proposed SIMD parallel ALU design is shown in Figure 6. It should be noted that due to the reactive nature of computation used by INCA, data for the local operands can be staged to scratchpad or even local ALU cache (when predictable) prior to the remote operands arriving in network messages. Local operand caches can be small (a few operands) up to a size of a few KiB. For example, a reasonable design may use 4KiB caches. This would typically require at least a 1MiB scratchpad to feed 256 ALU 4KiB caches at a reasonable rate. It is possible to design a scratchpad to

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Optimization</th>
<th>Payload size</th>
<th>Average speedup wrt base</th>
<th>Average speedup wrt scratchpad</th>
</tr>
</thead>
<tbody>
<tr>
<td>matrix-transpose</td>
<td>base</td>
<td>128B</td>
<td>4.16</td>
<td>283.96</td>
</tr>
<tr>
<td></td>
<td>scratchpad</td>
<td>256B</td>
<td>3.04</td>
<td>512B</td>
</tr>
<tr>
<td>filter</td>
<td>base</td>
<td>1024B</td>
<td>2.18</td>
<td>1024B</td>
</tr>
<tr>
<td></td>
<td>scratchpad</td>
<td>2048B</td>
<td>1.81</td>
<td>2048B</td>
</tr>
<tr>
<td>matrix-unpack</td>
<td>base</td>
<td>4096B</td>
<td>1.71</td>
<td>4096B</td>
</tr>
<tr>
<td>convolution</td>
<td>base</td>
<td>8192B</td>
<td>1.64</td>
<td>8192B</td>
</tr>
<tr>
<td>linear-interpolation</td>
<td>base</td>
<td>32B</td>
<td>1.32</td>
<td>32B</td>
</tr>
<tr>
<td>hadamard-product</td>
<td>base</td>
<td>256B</td>
<td>1.13</td>
<td>256B</td>
</tr>
<tr>
<td>dot-product</td>
<td>base</td>
<td>1024B</td>
<td>1.14</td>
<td>1024B</td>
</tr>
<tr>
<td>matrix-multiplication</td>
<td>base</td>
<td>2048B</td>
<td>1.15</td>
<td>2048B</td>
</tr>
</tbody>
</table>

Table 5: Kernel runtimes in µs. See main text for a description of the optimizations.

Hardware Optimization Performance Results. Revisiting Table 5, for the parallel-clobber kernels we observe average speeds up to 254.01× for matrix multiplication, 43.33× for dot_product, and 7208.41× for hadamard_product. Compared to scratchpad, the respective speeds are 7.92×, 1.33×, and 240.11×.
While this phase cannot be fully parallelized, it can be restructured well in advance of the operations in the scratchpad memory ALU hardware for affecting this process, we extended the INCA (so the gap and memory latency costs are only incurred once, at the initiation of the instruction). Again referring to Table 5, these stages occur in a cascade, without requiring re-injection of new instructions (so the gap and memory latency costs are only incurred once, at the initiation of the instruction). Again referring to Table 5, these advanced parallel dot-product and matrix-multiplication kernels achieve average speedups over base of 2807.45× and 1354.69×, respectively (85.81× and 42.09× over scratchpad).

5.3.5 Increasing Network Speeds. As illustrated in figure 5, INCA execution times are expected to decrease as network speeds increase. To investigate the impact of network bandwidth on INCA kernel runtimes, we modified model parameters to execute kernels on the scratchpad configuration with 64B messages at 400Gb/s and 1000Gb/s network speeds; gap and Portals overhead becomes 1.28ns and 0.512ns, respectively. These parameters reflect the assumption that loopback for INCA program execution is optimized to avoid overhead. Table 6 shows results for kernel execution on an initial base of 1067.5 650.24 216.16 131.28 60.85 37.25 1067.5 650.24 216.16 131.28 60.85 37.25. The large speedup for Hadamard product is due to the fact that loopback for INCA program execution is optimized to avoid overhead. Table 6 shows results for kernel execution on an initial base of 1067.5 650.24 216.16 131.28 60.85 37.25.

5.4 Application Performance Impact

The previous results analyze INCA’s potential performance for the on-NIC kernels themselves. While this provides some quantitative context for appreciating INCA’s benefits, we also studied the potential impact this approach can have on the higher-level applications. We studied a set of proxy applications identified by the Exascale project [26] and a full application.

MiniAMR [19] is an adaptive mesh refinement code designed to represent a range of applications. It should be noted that a general solver is used in this application, as the target of the proxy application is to capture the behavior of an adaptive mesh, rather than the computational kernels. MiniAMR uses pack/unpack and interpolation routines similar to the INCA kernels described above.

MiniMD [19] is a molecular dynamics code that is a subset of the LAMMPS code. This code runs the Lennard Jones Liquid solver from LAMMPS that is based on a halo exchange pattern. This application was selected because it runs a well used simulation mode of a major application. To analyze the potential for INCA, we adapted the computation to separate internal data dependencies, the overlap target, from external dependencies, the INCA kernel target. This process also allows for communication and computation to be overlapped.

MiniFE [19] is a finite elements code that solves a conjugate gradient. This application was selected for its broad applicability to major production applications. Particularly, MiniFE’s solver features a matrix vector multiply that can be split into two separate computation phases, one handles intra-rank data dependencies and the other handles inter-rank data dependencies. As these split computation phases don’t directly depend on the other, the inter-rank computation can be directly translated into an INCA kernel without restructuring the application. MiniFE spends an observable amount of time in its setup phase that would otherwise be negligible in a production application. Therefore, to examine the impact on real applications we only consider the major time consuming section for production codes, the solver time.
LAMMPS [25] is a full molecular dynamics application which MiniMD is based on. For this test, we ran the same problem as with miniMD. The adaptations here mirror those of MiniMD discussed above. Many of the solvers in LAMMPS share the same structure and can be converted to INCA in a similar manner.

For each application, we identified code that could potentially be offloaded as an INCA kernel, subject to several criteria. First, these potential "INCA targets" should come directly before or after a communication region; second, the target needs to take a significant portion of the application’s runtime; and finally, the target has to be separable from other computation. For MiniFE, MiniMD, and LAMMPS, there were computational regions that could easily be separated to internally dependent (intrinode) and externally dependent (internode) computation. As an initial evaluation, we measured the target kernel as just the external computation. Additionally, for these applications we analyzed the maximum possible speedup if we could also refactor the code to better split the computation between the INCA processor and the CPU. This analysis has the caveat that it doesn’t account for variations in the throughput between an INCA processor and a CPU. In contrast to the other three apps, MiniAMR does not have easily separable regions, so MiniAMR required a refactor to stagger data dependencies to allow INCA to run in parallel with the CPU.

For code modification, we set a limit of no more than 10 lines of source code (not including the INCA kernel) to meet our definition of not refactoring the code. Non-refactored code represents a ‘first pass’ INCA implementation expected to take a few hours to several days of developer time. This requires modifications to allow for communication overlap with internal computation, to prevent external computation from occurring, and implementing the INCA kernel. The effort required to address the first two is relatively minimal, e.g., MiniFE already addresses communication overlap, and required removing nine lines to address the external computation. Similarly, MiniMD required under 10 lines to affect communication overlap, and two additional lines to remove externally dependent computation from the CPU code. Converting one of the LAMMPS solvers requires a similar effort to MiniMD.

For each application, we timed the INCA target kernel, communication, and the computation phase the target kernel would overlap. Table 7 shows the results of profiling these applications averaged over 10 runs. For non-refactored code, we observe a potential runtime improvement of up to 2.98% in MiniFE, up to 11.0% in MiniMD, and up to 11.5% in the LJ_cut solver for LAMMPS. With a structural refactoring, we can leverage the INCA’s processing power to further parallelize these applications. With MiniFE and MiniMD our refactor predictions are the result of moving some of the internal computation to the INCA kernel. For MiniAMR the process is more complex, where computation starts and continues as an unpack INCA kernel is making incoming data available and a pack INCA kernel is preparing completed data for transfer. With refactoring, speedups 26%, 37.2%, 25.7%, and 28.9% for MiniAMR, MiniMD, MiniFE, and LAMMPS respectively.

There are complexities and nuances that this analysis doesn’t account for, namely performance variation from laggard processes and network congestion. With delays caused in data arrival, the amount of time available to run INCA kernels before the host needs the data may be reduced such that the data must be handed off to the host before processing is complete, necessitating a lengthened overall runtime due to reduced overlap potential. However, even in the unlikely case where extreme performance variation halved the performance improvement shown in this analysis, the impacts of leveraging INCA kernels could have significant impact on the runtime of applications at scale.

### Table 7: Potential Application Impact

<table>
<thead>
<tr>
<th></th>
<th>MiniAMR</th>
<th>MiniMD</th>
<th>MiniFE</th>
<th>LAMMPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Runtime</td>
<td>174 s</td>
<td>43.1 s</td>
<td>37.0 s</td>
<td>39.2 s</td>
</tr>
<tr>
<td>Communication</td>
<td>52.8 s</td>
<td>6.98 s</td>
<td>4.36 s</td>
<td>6.23 s</td>
</tr>
<tr>
<td>INCA Target</td>
<td>45.4 s</td>
<td>4.73 s</td>
<td>1.10 s</td>
<td>3.80 s</td>
</tr>
<tr>
<td>Overlap Target</td>
<td>76.3 s</td>
<td>34.3 s</td>
<td>22.3 s</td>
<td>21.5 s</td>
</tr>
<tr>
<td>Potential speed-up without refactor</td>
<td>N/A</td>
<td>11.0%</td>
<td>2.98%</td>
<td>11.5%</td>
</tr>
<tr>
<td>Potential speed-up with refactor</td>
<td>26%</td>
<td>37.2%</td>
<td>25.7%</td>
<td>28.9%</td>
</tr>
</tbody>
</table>

The host before processing is complete, necessitating a lengthened overall runtime due to reduced overlap potential. However, even in the unlikely case where extreme performance variation halved the performance improvement shown in this analysis, the impacts of leveraging INCA kernels could have significant impact on the runtime of applications at scale.

### 6 DISCUSSION

In this section we answer several common questions about INCA and its use in future generation NICs.

**How hard would it be to build an INCA NIC?** INCA builds upon existing NIC architectures that support atoms, tag matching and triggered operations such that the majority of the requirements of INCA can be met by current designs. Examples of such NICs include the Portals-based Atos-Bull BXI interconnect [12], and any NIC that supports atoms and tag matching compliant with the OpenFabrics interface. Existing InfiniBand solutions could also be adapted with the addition of triggered operations. Overall the difficulty of creating an INCA NIC could reasonably be predicted to be easier than or equal to in difficulty to that of a deadline-based compute approach.

**What benefits does a preemptible model provide?** Preemptibility allows INCA to temporarily suspend work, permitting high priority incoming data to be delivered without being slowed down by computation occurring on the NIC. This ability allows for work to be done beyond typical deadlines in packet processing as a separate set of resources can be allocated to accommodate INCA kernels that will not be immediately needed by other incoming packets. The number of overall INCA programs that can be run simultaneously can be limited, ensuring that these resources are never exhausted. Preemptibility also avoids re-injecting packets back into the network, which could cause congestion and head-of-line blocking. This allows for longer and more complex computation to take place on the NIC and makes NIC idle time a useful compute resource.

**Can my host be forced to wait on compute completion from the NIC?** While it would be possible to design an INCA NIC that could block network progress until INCA kernels complete, we do not recommend such a design. It is possible to design INCA kernels with an accompanying interrupt function that can be triggered from the host CPU or other co-processors via the PCIe bus. A proposed interrupt function could work as follows for a matrix multiplication. First, the INCA NIC immediately returns the current values of the i, j, and k for each of the matrices being worked on, representing the current status of the computation. Next, the INCA NIC can...
move the data to where the host can continue the computation. When triggered, such interrupts can contain enough information about the requesting compute resource that direct cache injection is possible into modern CPUs. This has the effect of drastically reducing the time between the interrupt and data delivery. As the INCA NIC can stream the scratchpad contents out to memory while injecting the data immediately needed to complete the computation to the requested CPU/GPU. With an efficiently designed INCA NIC interrupt mechanism, the time difference between a memory read request and the INCA data injection to a CPU could be comparable, allowing for very fast handover from the NIC kernel execution.

**Why does the INCA model sacrifice serial execution speed?** INCA provides preemptibility at the cost of serial processing speed as each instruction is re-injected at the beginning of the NIC processing pipeline. INCA will always work at a speed in proportion to the network data rate or atomic operations will not be supported on the underlying network. Such primitives are important for existing communication libraries, so it is reasonable to expect that manufacturers will continue to provide such support. In addition, the preemptible nature of INCA means that kernels will continue to be forward compatible with successive NIC generations. Although some performance optimization may be required, legacy kernels will work, unlike deadline-based solutions where kernels may not be forward-compatible due to shrinking deadlines, or work in compatibility modes that introduce extra latency for all traffic on the network to enable legacy handler code to execute. An example of this forward-compatibility challenge is illustrated in Figure 7 where the number of instructions available declines between 200Gb/s and 400Gb/s networks.

However, the goal of INCA is not to provide full rate speeds of data processing, it is to exploit periods of time where the host can overlap computation with the work occurring on the INCA NIC, gaining application speedup equivalent to the amount of time needed to perform the work performed by the NIC kernel.

**When can INCA outperform a CPU or GPU?** The data parallel nature of the INCA architecture allows for speedup over the host itself. A Hadamard product on 8KiB worth of data can occur in only 440 cycles of a 2.5 GHz host processor, which is faster than a host CPU can read the data for the compute from main memory (where the NIC would deposit it). With direct L1 cache injection it would be possible to load the data in approximately 5 cycles and perform a multiplication in 3 cycles. Accounting for pipelining and out of order execution, it is possible (but difficult) to obtain 1 cycle per instruction of throughput. Assuming this, INCA would still be significantly faster, the CPU could perform 440 multiplications at the time that INCA performed 1024. With multiple CPU cores and perfect data injection layout a 32-core CPU could perform approximately 14K multiplications in the time that INCA performs 1K. Therefore, while INCA is not equivalent in raw computing power to a CPU, it is a valuable resource that can provide several cores worth of compute power assistance to a host.

**Can INCA be used in combination with stream-based (deadline) approaches?** Certainly. INCA can be used alongside stream-based approaches, using the stream processing to handle small compute jobs that require immediate response or very low latency from the host (i.e. the host is waiting for the data). For solutions like Broadcom’s Stingray NIC [6] or sPIN [20] which both work on packets, INCA can co-exist with the caveat that the die space for both approaches would be reduced and INCA would likely have less parallelism available to it and the stream-based packet processor would have a shorter deadline. The approaches could even be used in combination, with the stream-based processing serving as a pre-processor for INCA kernels (working on data before it is stored in the scratchpad). With stream-based message processing solutions like Mellanox’s Bluefield [23], INCA would work similarly, with the distinction that it would not function as a pre-processor due to the placement of the processing units in the data pipeline.

![Figure 7: Number of instructions available per core under sPIN.](image)

**How does INCA relate to FPGA SmartNIC Designs?** While we have proposed a microarchitecture for INCA that uses Portals-capable NICs as a foundation, the design of INCA matches the microarchitecture of an FPGA quite easily for some logical elements. The parallel and advanced parallel optimizations for INCA could be implemented on an FPGA quite easily as the logic block and block RAM structure of an FPGA matches with the many ALU with local cache type architecture for INCA. However, we are proposing ASIC solutions as they are already developed for the vast majority of INCA functionality and have higher potential frequencies, enabling an INCA NIC to provide line-rate processing.

**Why discuss the INCA optimizations separately when they seem straightforward?** While at least some of these optimizations are common in standard CPUs, they are non-trivial in the context of a new architecture such as INCA. Since designing this hardware requires significant effort, we explore the optimizations separately so that designers can perform a cost-benefit analysis on each optimization independently.

**Traditional Atomics have operations dictated by the source, why does INCA make the target dictate the operations?** This is a significant diversion from current NIC designs and one of the most important aspects of INCA. Running a program on a remote node is difficult if the remote node is unaware of the execution as it may not have the correct receive side information posted to accommodate program execution at all times. This makes a significant change in the control flow difficult. By moving the control of the operations to the location where the operations take place, INCA can control the flow of data without impacting the ordering of data. In addition, this receiver side focus allows the NIC scratchpad to be intelligently pre-loaded with the operands needed for a given program whenever such operands become available. As some operands are known from the beginning of execution time, this is advanced knowledge of the program execution can be leveraged for better performance (see previous scratchpad speedups).
7 CONCLUSION
In this paper, we have presented a new model for processing in network. INCA, INCA provides NIC-local compute assistance by building upon offload capabilities of state-of-the-art NICs, including atomic operations, message matching, and trigged operations. The resulting computational model is Turing complete, and can be implemented within the Ports network programming API with minor changes and additions to the specification. Unlike traditional stream-processing approaches, computational work under INCA is not limited by caps on instruction counts or processing time; instead, the model is inherently preemptible by virtue of utilizing the same data paths as other network traffic. For the same reason, INCA kernels can be forward-compatible with successive network generations. The cost incurred by this solution is that INCA programs can be slow relative to stream-processing approaches, as we demonstrated in the ‘base’ and ‘scratchpad’ evaluations (section 5). In comparison to full-featured compute cores, however, the processing units performing INCA operations are simple and small; e.g., 256 ALUs occupy approximately half the space of a 32 core CPU. Taking advantage of this possibility for data parallelism, application codes can exhibit speedups of up to 11% with no modifications beyond an INCA kernel and up to 37% when optimized for INCA. Hand-tuned INCA kernels can outperform stream-processing for some tasks that are possible with both methods and can perform computations not possible with stream-processing. We conclude by noting that INCA and stream-processing approaches are complementary, with INCA handling larger more complex computations and stream processing providing quick response times for small compute tasks.

REFERENCES
SUMMARY OF THE EXPERIMENTS REPORTED

We ran MiniFE 2.1.0, MiniMD 2.0, MiniAMR 1.0, and LAMMPS Stable Release on 12 Dec 2018. These were compiled using the Intel 19.0.3.199 and OpenMPI 3.0. The system these were run on is a dual socket 2.1 GHz Intel Broadwell E5-2695 v4 with 18 cores per processor, 128 GB of RAM per node, and an Intel Omni-Path Network. There were minor modification to the applications to separate internal versus external computation (MiniMD and LAMMPS) and for timing purposes (MiniMD, LAMMPS, and MiniFE).

ARTIFACT AVAILABILITY

Software Artifact Availability: Some author-created software artifacts are NOT maintained in a public repository or are NOT available under an OSI-approved license.

Hardware Artifact Availability: There are no author-created hardware artifacts.

Data Artifact Availability: There are no author-created data artifacts.

Proprietary Artifacts: No author-created artifacts are proprietary.

List of URLs and/or DOIs where artifacts are available:
- https://github.com/Mantevo/miniMD
- https://github.com/arm-hpc/miniAMR
- https://github.com/Mantevo/miniFE

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: The system these were run on is a dual socket 2.1 GHz Intel Broadwell E5-2695 v4 with 18 cores per processor, 128 GB of RAM per node, and an Intel Omni-Path Network.

Operating systems and versions: RHEL7

Compilers and versions: Intel 19.0.3.199

Applications and versions: MiniFE 2.1.0, MiniMD 2.0, MiniAMR 1.0, and LAMMPS Stable Release on 12 Dec 2018

Libraries and versions: OpenMPI 3.0

Key algorithms: Conjugate Gradient, Lennard Jones, Adaptive Mesh Refinement

Input datasets and versions: MiniFE input ‘-nx 800 -ny 800 -nz 800 -mv_overlap_comm_comp 1’ MiniAMR input ‘-num_refine 4 -max_blocks 1000 -init_x 1 -init_y 1 -init_z 1 -npix 8 -npyp 8 -npz 8 -nx 8 -ny 8 -nz 8 -num_vars 32 -inbalance 0 -num_tsteps 2000 -refine_freq 3 -checksum_freq 1 -stages_per_ts 1 -num_objects 3 -object 3 0.375 0.0 0.83 0.00015 0.0 0.00004 0.0 0.00015 0.00008 -object 14 0.375 0.0 0.75 0.00012 0.0 0.00017 0.0 0.00012 0.034 0.034 0.00017 0.00034 0.00034 -object 14 0.375 0.0 0.75 0.00012 0.0 0.00017 0.0 0.00012 0.034 0.034 0.00017 0.00034 0.00034

LAMMPS input file: #Lennard Jones inpute file for LAMMPS units lj variable s index 420 variable x index $s variable y index $s variable z index $s variable n index 50 atom_style atomic variable gn index 0 if "${gn} > 0" then & "newton on" & else & "newton off" & lattice fcc 0.8442 region box block 0 $[x] 0 $[y] 0 $[z] create_box 1 box create_atoms 1 box mass 1 1 velocity all create 1.44 376847 loop geom pair_style lj cut 2.5 pair_coeff 1 1 1.1 neighbor 0.3 bin neigh_modify every 20 delay no fix 1 all nve timestep 0.005 thermo 100 run $n

Paper Modifications: The following application modifications were made for this paper:

For MiniMD and LAMMPS the LJ computation loop was duplicated, and an if statement was added to each separating the loops to the first evaluating local neighbors, and the second evaluating ghost cells.

For MiniFE, MiniMD, and LAMMPS timers were placed around the internal computation, external computation, and communication sections. After the run, the laggard process was identified (maximum internal + external computation as the communication would have overlapped the internal computation) and that processes times were printed.

MiniFE input ‘-nx 800 -ny 800 -nz 800 -mv_overlap_comm_comp 1’ MiniAMR input ‘-num_refine 4 -max_blocks 1000 -init_x 1 -init_y 1 -init_z 1 -npix 8 -npyp 8 -npz 8 -nx 8 -ny 8 -nz 8 -num_vars 32 -inbalance 0 -num_tsteps 2000 -refine_freq 3 -checksum_freq 1 -stages_per_ts 1 -num_objects 3 -object 3 0.375 0.0 0.83 0.00015 0.0 0.00004 0.0 0.00015 0.00008 -object 14 0.375 0.0 0.75 0.00012 0.0 0.00017 0.0 0.00012 0.034 0.034 0.00017 0.00034 0.00034 -object 14 0.375 0.0 0.75 0.00012 0.0 0.00017 0.0 0.00012 0.034 0.034 0.00017 0.00034 0.00034 -lb_opt 2 -block_change 1’ LAMMPS input ‘-n 50 -s 420 -gn 0 -half_neigh 0 -t 1’ LAMMPS input file: #Lennard Jones inpute file for LAMMPS units lj variable s index 420 variable x index $s variable y index $s variable z index $s variable n index 50 atom_style atomic variable gn index 0 if "${gn} > 0" then & "newton on" & else & "newton off" & lattice fcc 0.8442 region box block 0 $[x] 0 $[y] 0 $[z] create_box 1 box create_atoms 1 box mass 1 1 velocity all create 1.44 376847 loop geom pair_style lj cut 2.5 pair_coeff 1 1 1.1 neighbor 0.3 bin neigh_modify every 20 delay no fix 1 all nve timestep 0.005 thermo 100 run $n