iThreads: A Threading Library for Parallel Incremental Computation

Pramod Bhatotia
MPI-SWS
bhatotia@mpi-sws.org

Pedro Fonseca
MPI-SWS
pfonseca@mpi-sws.org

Umut A. Acar
Carnegie Mellon University & Inria
umut@cs.cmu.edu

Björn B. Brandenburg
MPI-SWS
bbb@mpi-sws.org

Rodrigo Rodrigues
NOVA University of Lisbon / CITI / NOVA-LINCS
rodrigo.rodrigues@fct.unl.pt

Abstract
Incremental computation strives for efficient successive runs of applications by re-executing only those parts of the computation that are affected by a given input change instead of recomputing everything from scratch. To realize these benefits automatically, we describe iThreads, a threading library for parallel incremental computation. iThreads supports unmodified shared-memory multithreaded programs: it can be used as a replacement for pthreads by a simple exchange of dynamically linked libraries, without even re-compiling the application code. To enable such an interface, we designed algorithms and an implementation to operate at the compiled binary code level by leveraging MMU-assisted memory access tracking and process-based thread isolation. Our evaluation on a multicore platform using applications from the PARSEC and Phoenix benchmarks and two case-studies shows significant performance gains.

Categories and Subject Descriptors D.1.3 [Programming Techniques]: Concurrent Programming; D.4.1 [Operating Systems]: Process Management; F.1.2 [Computation by Abstract Devices]: Modes of Computation

General Terms Algorithms, Design, Performance

Keywords Incremental computation, shared-memory multithreading, self-adjusting computation, memoization, Concurrent Dynamic Dependence Graph (CDDG), Release Consistency (RC) memory model

1. Introduction
Many applications operate incrementally by repeatedly invoking an algorithm or a program over input data that differs slightly from that of the previous invocation [25, 30, 37, 38, 42, 46, 68]. In such a workflow, small, localized changes to the input often require only small updates to the output, creating an opportunity to update the output incrementally instead of recomputing it from scratch. Since the work done is often proportional to the change size [6], rather than the total input size, incremental computation can achieve significant performance gains and efficient use of computing resources.

To obtain the benefits of incremental computation automatically, researchers in the programming-languages community have proposed compiler- and language-based approaches. While this has been an area of active research for several decades (§7), almost all prior work targets sequential programs—with the notable exceptions of two recent approaches for parallel incremental computation [27, 48].

Importantly, by leveraging a language-based approach, these two prior proposals [27, 48] have demonstrated substantial speedups, thus establishing that the promise of incremental computation can be realized in parallel programs. Specifically, these systems enable efficient and correct incremental updates to the output through the use of new programming languages with special data types (§7), and by requiring a strict fork-join programming model, where threads communicate only at end points (i.e., when forking/joining).

These choices reflect a difficult design tradeoff: they provide the compiler and runtime system with the information required to maximize reuse, but they also impose a cost on the programmer, who has to provide appropriate type annotations and, in some cases, also application-specific functions to safely implement the new type system. Furthermore, due to the restricted programming model, they also preclude support for many existing shared-memory multithreaded programs.
and synchronization primitives (such as R/W locks, mutexes, semaphores, barriers, and conditional wait/signal).

In this paper, we instead target increased generality, and to this end propose an OS-based approach to parallel incremental computation. More specifically, we present iThreads, a threading library for parallel incremental computation, which achieves the following goals.

- **Practicality**: iThreads supports the shared-memory multi-threaded programming model with the full range of synchronization primitives in the POSIX API.
- **Transparency**: iThreads supports unmodified programs (e.g., C/C++) without requiring the use of a new language with special data types.
- **Efficiency**: iThreads achieves efficiency, without limiting the available application parallelism, as its underlying algorithms are parallel as well.

The iThreads library is easy to use (see Figure 1 for the workflow): the user just needs to preload iThreads to replace pthreads by using the environment variable LD_PRELOAD. The dynamically linkable shared library interface allows existing executables to benefit from iThreads.

For the first run of a program (or the initial run), iThreads computes the output from scratch and records an execution trace. All subsequent runs for the program are incremental runs. For an incremental run, the user modifies the input and specifies the changes; e.g., assuming that the program reads the input from a file, the user specifies the offset and len for the changed parts of the file. Thereafter, iThreads incrementally updates the output based on the specified input changes and the recorded trace from the previous run.

Our approach relies on recording the data and control dependencies in a computation during the initial run by constructing a Concurrent Dynamic Dependence Graph (CDDG). The CDDG tracks the input data to a program, all sub-computations (a sub-computation is a unit of the computation that is either reused or recomputed), the data flow between them, and the final output. For the incremental run, a (parallel) change propagation algorithm updates the output and the CDDG by identifying sub-computations that are affected by the changes and recomputing only those sub-computations.

We make the following main contributions:

- We present parallel algorithms for incremental multi-threading (§4).

Our experience with iThreads shows that significant performance gains (time savings) and efficient resource utilization (work savings) can be achieved in many parallel applications without requiring any effort from the programmer.

### Design Overview

We base our design on POSIX threads, or pthreads, a widely used threading library for shared-memory multithreading with a rich set of synchronization primitives.

#### 2.1 The Basic Approach

Our design adapts the principles of self-adjusting computation [6] for shared-memory multithreading, and also makes use of techniques from record-replay systems employed for reliable multithreading (§7). At a high level, the basic approach proceeds in the following three steps:

1. Divide a computation into a set of sub-computations $N$.
2. During the initial run, record an execution trace to construct a Concurrent Dynamic Dependence Graph (or CDDG). The CDDG captures a partial order $O = (N, \rightarrow)$ among sub-computations with the following property: given a sub-computation $n$ (where $n \in N$) and the subset of sub-computations $M$ that precede it according to $\rightarrow$, i.e., $M = \{M \subset N \mid \forall m \in M, m \rightarrow n\}$, if the inputs to all $m \in M$ are unchanged and the incremental run follows the partial order $\rightarrow$, then $n$’s input is also unchanged and we can reuse $n$’s memoized effect without recomputing $n$.
3. During the incremental run, propagate the changes through the CDDG. That is, the incremental run follows an order that is consistent with the recorded partial order $\rightarrow$, reusing sub-computations whose input is unchanged and re-computing those whose input has changed.
2.2 Example

We next use a simple example, shown in Figure 2, to explain how our basic approach works. The example considers a multi-threaded execution with two threads ($T_1$ & $T_2$) modifying three shared variables ($x$, $y$, & $z$) using a lock.

**Step #1: Identifying sub-computations.** We divide a thread execution into sub-computations at the boundaries of lock() / unlock(). (We explain this design choice in §3.) We identify these sub-computations as $T_1.a$ for thread $T_1$, and $T_2.a$ & $T_2.b$ for thread $T_2$. For the initial run, let us assume that thread $T_2$ acquired the lock after the execution of sub-computation $T_1.a$. This resulted in the following thread schedule for sub-computations: $T_1.a → T_2.a → T_2.b$.

**Step #2: Construct the CDDG.** To understand the dependencies that need to be recorded to build the CDDG, we consider incremental runs with changes either in the input data or the thread schedule (shown in Figure 3).

We first consider the case of change in the input data. An important function of the CDDG is to propagate the changes through the graph by determining whether a sub-computation is transisitively affected by the input change. For example consider case A in Figure 3, when the value of variable $y$ is changed—in this case, we need to recompute $T_1.a$ because it reads the modified value of $y$. In contrast, we can still reuse $T_2.a$ because it is independent of $y$ and also not affected by the writes made by $T_1.a$. However, we need to recompute $T_2.b$ even though it does not directly depend on $y$, since it is still transitively affected (via modified $z$) by the writes made by $T_1.a$. Therefore, the CDDG needs to record data dependencies (meaning which sub-computations modify a value that is read by another sub-computation) to determine whether a sub-computation can be reused or if it has to be recomputed.

Next, we consider the case of a change in the thread schedule. In general, multi-threaded programs are non-deterministic because the OS scheduler is free to interleave sub-computations in different ways. As a result, a problem can arise if the initial and the incremental runs follow different schedules. This might alter the shared state, and therefore cause unnecessary re-computations even without any input changes. For example consider case B in Figure 3: if thread $T_1$ acquires the lock after the execution of $T_2.b$ (i.e., a changed thread schedule of $T_2.a → T_2.b → T_1.a$) then sub-computations $T_1.a$ and $T_2.b$ need to be recomputed because of the changed value of $y$. Therefore, (and as observed by prior work on reliable multithreading (§7)) the partial order captured by the CDDG (→) is a happens-before order among synchronization events, which ensures that, given unchanged input and that all threads acquire locks in the same order as dictated by →, all sub-computations remain unchanged, as shown in case C in Figure 3. We explain how to build this partial order in §4.

**Step #3: Change propagation.** The previous observations allow us to reach a refined explanation of our basic algorithm (see Algorithm 1). The starting point is the CDDG that records the happens-before order (→) between sub-computations, according to the synchronization events. Furthermore, data dependencies are recorded implicitly in the CDDG by recording the read and write sets: if we know what data is read and written by each sub-computation, we can determine whether a data dependency exists, i.e., if a sub-computation is reading data that was modified by another sub-computation. Therefore, the incremental run visits sub-computations in an order that is compatible with →, and, for each sub-computation, uses the read and write sets to determine whether part of its input was modified during the incremental run. If the read-set is modified then the sub-computation is re-computed, otherwise we skip the execution of the sub-computation, and directly write the memoized value of the write-set to the address space.

### Algorithm 1: Basic algorithm for the incremental run

```plaintext
 Algorithm 1: Basic algorithm for the incremental run

```

<table>
<thead>
<tr>
<th>Case</th>
<th>Input</th>
<th>Thread schedule</th>
<th>Sub-computations</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$x, y, z$</td>
<td>$T_1.a → T_2.a → T_2.b$</td>
<td>$T_2.a$</td>
</tr>
<tr>
<td>B</td>
<td>$x, y, z$</td>
<td>$(T_2.a → T_2.b → T_1.a)$</td>
<td>$T_2.a$</td>
</tr>
<tr>
<td>C</td>
<td>$x, y, z$</td>
<td>$T_1.a → T_2.a → T_2.b$</td>
<td>$T_2.a$</td>
</tr>
</tbody>
</table>

Figure 3. For the incremental run, some cases with changed input or thread schedule (changes are marked with *)
inter-thread communication would be prohibitively expensive, essentially requiring tracking of shared-memory accesses at the granularity of individual load/store instructions.

In contrast, the RC model only requires writes made by one thread to become visible to other threads at synchronization points, thus restricting inter-thread communication to such points. This allows us to define the granularity of a sub-computation at the boundaries of synchronization points, which is essential to achieving feasible runtime overheads.

Note that the RC model still guarantees correctness and liveness for applications that are data-race-free [9]. Consequently, iThreads assumes that the programs are data-race-free w.r.t. pthreads synchronization primitives. In fact, the semantics provided by iThreads is no more restrictive than pthreads semantics [4], which mandate that all accesses to shared data structures must be properly synchronized using pthreads synchronization primitives, and which guarantees only that any updates become visible to other threads when invoking a pthreads synchronization primitive.

Synchronization model. We support the full range of synchronization primitives in the pthreads API. However, due to the weakly consistent RC memory model, our approach does not support ad-hoc synchronization mechanisms [82] such as user-defined spin locks. (We will revisit this limitation in §8.) Our current implementation also does not handle C/C++ atomic synchronization constructs.

4. Algorithms

In this section we present two parallel algorithms for incremental multithreading. The first algorithm is for the initial run that executes the program from scratch and constructs the CDDG. The second algorithm is for the incremental run that performs change propagation through the CDDG. Both algorithms rely on the CDDG, which we explain first.

4.1 Concurrent Dynamic Dependence Graph (CDDG)

The CDDG is a directed acyclic graph with vertices representing sub-computations (or thunks), and two types of edges to record dependencies between thunks: happens-before edges and data-dependence edges. We next explain how to derive vertices and edges.

Thunks (or sub-computations). We define a thunk as the sequence of instructions executed by a thread between two pthreads synchronization API calls. We model an execution of thread \( t \) as a sequence of thunks \( L_t \). Thunks in a thread are totally ordered based on their execution order using a monotonically increasing thunk counter \( \alpha \). We refer to a thunk of thread \( t \) using the counter \( \alpha \) as an index in the thread execution sequence \( L_t \), i.e., \( L_t[\alpha] \).

Happens-before edges. There are two types of happens-before edges: control edges, which record the intra-thread execution order; and synchronization edges, which record explicit inter-thread synchronization events.

Control edges are simply derived by ordering thunks of the same thread based on their execution order. Synchronization edges are derived by modeling synchronization primitives as acquire and release operations. In particular, during synchronization, a synchronization object \( s \) is released by one set of threads and subsequently acquired by a corresponding set of threads blocked on the synchronizing object. For example, an unlock(s) operation releases \( s \) and a corresponding lock(s) operation acquires it. Similarly, all other synchronization primitives can also be modeled as acquire and release operations [43, 71].

Under the acquire-release relation, a release operation happens-before the corresponding acquire operation. Given that a thunk’s boundaries are defined at synchronization points, the acquire and release operations also establish the happens-before ordering between thunks of different threads. Formally, two thunks \( L_{(t_1)}[\alpha] \) and \( L_{(t_2)}[\beta] \) are connected by a

- control edge iff they belong to the same thread \( (t_1 = t_2) \) and \( L_{(t_1)}[\alpha] \) was executed immediately before \( L_{(t_2)}[\beta] \);
- synchronization edge iff \( L_{(t_1)}[\alpha] \) releases a synchronization object \( s \) and \( L_{(t_2)}[\beta] \) is a thunk that acquires \( s \) next.

Data-dependence edges. Data dependencies are tracked to establish the update-use relationship between thunks. Intuitively, such a relationship exists between two thunks if one reads data written by the other. More formally, for a thunk \( L_t[\alpha] \), the read-set \( L_t[\alpha] \cap R \) and the write-set \( L_t[\alpha] \cap W \) are the set of addresses that were read-from and written-to, respectively, by the thread \( t \) while executing the thunk. Two thunks \( L_{(t_1)}[\alpha] \) and \( L_{(t_2)}[\beta] \) are then connected by a

- data-dependence edge iff \( L_{(t_2)}[\beta] \) is reachable from \( L_{(t_1)}[\alpha] \) via happens-before edges and \( L_{(t_1)}[\alpha] \cap W \cap L_{(t_2)}[\beta] \cap R = \emptyset \).

4.2 Algorithm for the Initial Run

During the initial run, we record the execution of the program to construct the CDDG. Algorithm 2 presents the high-level overview of the initial run algorithm, and details of the subroutines used in the algorithm are presented in Algorithm 3. The algorithm is executed by threads in parallel. The algorithm employs run-time techniques to derive the information needed for the CDDG. In particular, during a thread execution, the thread traces memory accesses on load/store instructions (using routine onMemoryAccess()), and adds them to the read and the write set of the executing thunk. (Our implementation, described in §5, derives the read and write sets at the granularity of memory pages using the OS memory protection mechanism.) The thread continues to execute instructions and perform memory tracing until a synchronization call is made to the pthreads library. At the synchronization point, we define the end point for the executing thunk and memoize its end state (using routine endThunk()). Thereafter, we let the thread perform the synchronization. Next, we start a new thunk and repeat the process until the thread terminates.
Algorithm 2: The initial run algorithm

```plaintext
/* Let S be the set of synchronization objects and T be the number of threads in the system. */
∀s ∈ S; ∀i ∈ {1, ..., T} : C_{s}[i] ← 0; // All sync clocks set to zero
executeThread(t)
begin
    initThread(t);
    while t has not terminated do
        startThunk(); // Start new thunk
        repeat
            Execute instruction of t;
            if (instruction is load or store) then
                onMemoryAccess();
            end
        until t invokes synchronization primitive;
        endThunk(); // Memoize the end state of thunk
        α ← α + 1; // Increment thunk counter
        // Let s denote invoked synchronization primitive
        onSynchronization(s);
    end
end
```

To infer the CDDG, control and synchronization edges are derived by ordering thunks based on the happens-before order. To do so, we use vector clocks \(C\) [65] to record a partial order that defines the happens-before relationship between thunks during the initial run, and in the incremental run we follow this partial order to propagate the changes. Our use of vector clocks is motivated by its efficiency for recording a partial order in a decentralized manner, rather than having to serialize all synchronization events in a total order.

Our algorithm maintains one vector clock for each thread, thunk, and synchronization object. These vector clocks are an array of size \(T\), where \(T\) denotes the number of threads in the system, which are numbered from 1 to \(T\).

Each thread \(t\) has a vector clock, called its "thread clock" \(C_t\), to track its local logical time, which is updated at the start of each thunk (using routine \(\text{startThunk}()\)) by setting \(C_t[L_t[α]]\) to the thunk index \(α\). Further, each thunk \(L_t[α]\) has a "thunk clock" \(L_t[α].C\), which stores a snapshot of \(C_t[L_t[α]]\) to record the thunk’s position in the CDDG.

Finally, each synchronization object \(s\) has a "synchronization clock" \(C_s\) that is used to order release and acquire operations (see \(\text{onSynchronization}()\)). More precisely, if a thread \(t\) invokes a release operation on \(s\), then \(t\) updates \(C_s\) to the component-wise maximum of its own thread clock \(C_t\) and \(C_s\). Alternatively, if \(t\) invokes an acquire operation on \(s\), it updates its own thread clock \(C_t\) to the component-wise maximum of \(C_t\) and \(s\)'s synchronization clock \(C_s\). This ensures that a thunk acquiring \(s\) is always ordered after the last thunk to release \(s\).

At the end of the initial run algorithm, the CDDG is defined by the read/write sets and the thunk clock values of all thunks.

4.3 Algorithm for the Incremental Run

The incremental run algorithm takes as input the CDDG \(∀t : L_t\) and the modified input (named the dirty set \(M\)), and performs change propagation to update the output as well as the CDDG for the next incremental run. As explained in the basic change propagation algorithm (Algorithm 1), each thread transitions through its list of thunks by following the recorded happens-before order to either reuse or recompute thunks. To make this algorithm work in practice, however, we need to address the following three challenges.

(1) Missing writes. When a thunk is recomputed during the incremental run, it may happen that the executing thread no longer writes to a previously written location because of a data-dependent branch. For such cases, our algorithm should update the dirty set with the new write-set of the thunk as well as the missing writes. These consist of the set of memory locations that were part of the thunk’s write-set in the previous run, but are missing in the current write-set.

(2) Stack dependencies. As mentioned previously, we transparently derive read and write sets by tracking the global memory region (heap/globals) using the OS memory protection mechanism (detailed in §5). Unfortunately, this mechanism is inefficient for tracking the per-thread stack region

Algorithm 3: Subroutines for the initial run algorithm

```plaintext
initThread(t)
begin
    α ← 0; // Initializes thunk counter (α) to zero
    ∀i ∈ {1, ..., T} : C[i][i] ← 0; // T's clock set to zero
end

startThunk()
begin
    C[α] ← α; // Update thread clock
    ∀i ∈ {1, ..., T} : L[α].C[i] ← C[i][α]; // Update thunk clock
    L[α].R/W ← ∅; // Initialize read/write sets to empty set
end

onMemoryAccess()
begin
    if load then
        L[α].R ← L[α].R ∪ \{memory-address\}; // Read
    else
        L[α].W ← L[α].W ∪ \{memory-address\}; // Write
end

endThunk()
begin
    memo (L[α].W) ← content(L[α].W); // Globals & heap
    memo (L[α].Stack) ← content(Stack);
    memo (L[α].Reg) ← content(CPU_Registers);
end

onSynchronization(s)
begin
    switch Synchronization type do
        case release(s):
            // Update s's clock to hold max of its and t's clocks
            ∀t ∈ {1, ..., T} : C[t][i] ← max(C[i][t], C[t][i]);
            sync(s); // Perform the synchronization
        case acquire(s):
            sync(s); // Perform the synchronization
            // Update t's clock to hold max of its and s's clocks
            ∀t ∈ {1, ..., T} : C[i][t] ← max(C[i][t], C[i][t]);
        end
end
```
Algorithm 4: The incremental run algorithm

Data: Shared dirty set \( M \leftarrow \{\text{modified pages}\} \) and \( L_t \)

\( \forall s \in S, \forall i \in \{1, ..., T\}: C_i[s] \leftarrow 0 \); All sync clocks set to 0

executeThread(i)
begin
    initThread(i); // Same as initial run algorithm
    while (t has not terminated and isValid(L_t[α]) ) do
        // Thread t is valid
        await isEnabled(L_t[α]) or !isValid(L_t[α])
        if (isEnabled(L_t[α]) then
            resolveValid(L_t[α]);
            C_t[i] ← α; // Update thread clock
            α ← α + 1; // Increment thunk counter
        end
    // The thread has terminated or a thunk has been invalidated
    L_t'[i] ← L_t; // Make a temp copy for missing writes
    while (t has not terminated or α < \( |L_t'| \) do
        // Thread t is invalid
        if (α < \( |L_t'| \)) then
            M ← M ∪ L_t'[α], W; // Add missing writes
            C_t[i] ← α; // Update thread clock
        end
        if (t has not terminated) then
            resolveInvalid(L_t[α]);
        end
        α ← α + 1; // Increment thunk counter
    end
    // The thread has terminated
end

(which usually resides in a single page storing local variables) because the stack follows a push/pop model, where the stack is written (or gets dirty) when a call frame is pushed or popped, even without a local variable being modified. To avoid the overheads of tracking local variables, we do not track the stack. Instead, we follow a conservative strategy to capture the intra-thread data dependencies. In our design, once a thunk is recomputed (or invalidated) in a thread, all remaining thunks of the thread are also invalidated in order to capture a possible change propagation via local variables.

(3) Control flow divergence. During the incremental run, it may happen that the control flow diverges from the recorded execution. As a result of the divergence, new thunks may be created or existing ones may be deleted. As in the previous challenge, the algorithm we propose takes a simple approach of only reusing a prefix of each thread (before the control flow diverges), and subsequently recording the new CDDG for enabling change propagation in subsequent runs.

Details. Algorithm 4 presents the overview of the incremental run algorithm, and details of subroutines used in the algorithm are presented in Algorithm 5. The incremental run algorithm allows all threads to proceed in parallel, and associates a state with each thunk of every thread. The state of each thunk follows a state machine (shown in Figure 4), which enforces that each thread waits until all thunks that happened-before its next thunk to be executed are resolved (i.e., either recomputed or reused), and only when it is certain that reusing memoized results is not possible will it start to re-execute its next thunk. In particular, the state of a thunk is either resolved or unresolved. The state of a thunk is resolved when the thunk has either been reused (resolved-valid) or re-executed (resolved-invalid). Otherwise, the thunk is still unresolved. An unresolved thunk is in one of the following states: pending, enabled or invalid.

Initially, the state of all thunks is pending, except for the initial thunk, which is enabled. A pending thunk is not “ready” to be considered for re-computation or reuse. A pending thunk of a thread is enabled (state transition ①) when all thunks of any thread that happened-before are resolved (either resolved-valid or resolved-invalid). To check for this condition (using routine isEnabled()), we make use of the strong clock consistency condition [65].

![Figure 4. State transition for thunks during incremental run](image-url)
5. Implementation

We implemented iThreads as a 32-bit dynamically linkable shared library for the GNU/Linux OS (Figure 5). iThreads uses two mechanisms of the Dthreads implementation [63]: the memory subsystem (§5.1) and a custom memory allocator (§5.4). Additionally, our implementation also includes the iThreads memoizer, which is a stand-alone application. We next describe the implementation in detail.

5.1 iThreads Library: Memory Subsystem

The iThreads memory subsystem implements the RC memory model and per-thunk read/write sets.

Release consistency memory model. To implement the RC memory model, iThreads converts threads into separate processes using a previously proposed mechanism [17]. This “thread-as-a-process” approach provides each thread with its own private address space, and thus allows iThreads to restrict inter-thread communication. In practice, iThreads forks a new process on pthread_create() and includes a shared memory commit mechanism [28, 56] that enables communication between processes at the synchronization points, as required by the RC memory model.

At a high level, throughout the application execution, iThreads maintains a copy of the address space contents in a (shared) reference buffer, and it is through this buffer, with instrumentation provided by iThreads at the synchronization points, that the processes transparently communicate (Figure 6). Communication between processes is implemented

by determining the thunk write-set, as explained next, which is then used to calculate a byte-level delta [63].

To compute the byte-level delta for each dirty page, iThreads performs a byte-level comparison between the dirty page and the corresponding page in the reference buffer, and then applies atomically the deltas to the reference buffer. In case there are concurrent writes by different processes to the same memory location, iThreads resolves the conflict by using a last-writer wins policy.

Furthermore, for efficiency reasons, the implementation of the communication mechanism relies on private memory-mapped files—this allows different processes to share physical pages until processes actually write to the pages, and still keeps performance overheads low by virtue of the OS copy-on-write mechanism.

Read and write set. Besides serving as the foundation for the RC memory model, the adopted thread-as-a-process mechanism is also essential for easily deriving per-thread read and write sets. More specifically, iThreads uses the OS memory protection mechanism to efficiently track the read and write sets. In particular, iThreads renders the address space inaccessible by invoking mprotect(PROT_NONE) at the beginning of each thunk, which ensures that a signal is triggered the first time a page is read or written by the thunk. Hence, within the respective signal handler, iThreads is able to record the locations of the accesses made to memory at the granularity of pages. Immediately after recording a memory access, the iThreads library proceeds to reset the page protection bits, allowing the thunk to resume the read/write operation as soon as the handler returns. In addition, resetting the permissions also ensures that subsequent accesses proceed without further page faults. In this way, iThreads incurs at most two page faults (one for reads & one for writes) for each accessed page during a thunk execution.

5.2 iThreads Library: Recorder and Replayer

The iThreads library executes the application in either recording or replaying mode. We next describe the two sub-components, recorder and replayer, that realize these modes of execution by implementing the algorithms described in §4.

Recorder. Since iThreads reuses the Dthreads memory subsystem, which serializes memory commit operations from different threads, the implementation of the recording algorithm is greatly simplified. Due to the resulting implicit serialization of thunk boundaries, the employed thread, thunk,
As practical applications depend on OS services, there are
iThreads
finally), subsequent thunks according to the replayer algorithm.
iThreads
returns, oizes the thunk state, and immediately after the system call
iThreads
mediately before a system call takes place, im-
calls, iThreads
calls thus cannot be memoized or replayed. To support system
has no access to kernel data structures. The effects of system

Figure 6. Overview of the RC model implementation

and synchronization vector clocks effectively reduce to scalar
sequence numbers, which allows the recorder to simply encode the thread schedule using thunk sequence numbers.

The recorder is further responsible for memoizing the state of the process at the end of each thunk. To this end, using an assembly routine, iThreads stores the register values on the stack, takes a snapshot of the dirty pages in the address space, and stores the snapshot in the memoizer (§5.4). In addition, the recorder also stores the CDDG, consisting of thunk identifiers (thread number and thunk sequence number) and their corresponding read/write sets, to an external file.

Replayer. Similarly to the recorder, the replayer relies on thunk sequence numbers to enforce the recorded schedule order. The replayer first reads the file with the input changes and the CDDG to initialize the replay algorithm. During an incremental run, whenever memoized thunks can be reused, the replayer retrieves the appropriate state from the memoizer, patches the address space and restores the state of registers.

5.3 iThreads Library: OS Support

As practical applications depend on OS services, there are
two important aspects related to the OS that iThreads needs
to address. First, system calls are used by the application to communicate with the rest of the system, so the effects of system calls (on the system and application) need to be addressed; in particular, input changes made by the user need to be handled. Second, there are OS mechanisms that can unnecessarily change the memory layout of the application across runs, preventing the reuse of memoized thunks.

System calls and input changes. Since iThreads is a user-space library running on top of an unmodified Linux kernel, it has no access to kernel data structures. The effects of system calls thus cannot be memoized or replayed. To support system calls, iThreads instead considers system calls to be thunk delimiters (in addition to synchronization calls). Hence, immediately before a system call takes place, iThreads memoizes the thunk state, and immediately after the system call returns, iThreads determines whether it still can reuse the subsequent thunks according to the replayer algorithm.

To ensure that system calls take effect (externally and internally), iThreads invokes system calls in all executions, even during replay runs. To guarantee that effects of system calls on the application (i.e., the return values and writes made to the address space) are accounted for by the thunk invalidation rules, iThreads infers the write-set of the system calls and checks whether the write-set contents match previous runs by leveraging knowledge of system call semantics (e.g., some system call parameters return pointers where data is written).

An important special case is that of reading the potentially large input to the computation (e.g., using mmap). In this case, iThreads efficiently identifies the content that does not match across runs by allowing the user to specify input changes explicitly. This relies on an external file, either written manually by users or produced by external tools, that lists the modified offset ranges (Figure 1).

In practice, our implementation intercepts system calls through wrappers at the level of glibc library calls.

Memory layout stability. To avoid causing unnecessary data dependencies between threads, iThreads reuses the custom memory allocator of Dthreads, which is based on HeapLayer [16]. The allocator isolates allocation and deallocation requests on a per-thread basis by dividing the application heap into a fixed number of per-thread sub-heaps. This ensures that the sequence of allocations in one thread does not impact the layout of allocations in another thread, which otherwise might trigger unnecessary re-computations.

In addition, iThreads disables Address Space Layout Randomization (ASLR) [1], an OS feature that deliberately randomizes the memory layout.

5.4 iThreads Memoizer

The memoizer is responsible for storing the end state of each thunk so that its effects can be replayed in subsequent incremental runs. The memoizer is implemented as a separate program that stores the memoized state in a shared memory segment, which serves as the substrate to implement a key-value store that is accessible by the recorder/replayer.

6. Evaluation

Our evaluation answers the following three main questions:

- What performance gains does iThreads provide for the incremental run? (§ 6.1)
- How do these gains scale with increases in the size of the input, the computation, and the input change? (§ 6.2)
- What overheads does iThreads impose for memoization and for the initial run? (§ 6.3)

Experimental setup. We evaluated iThreads on a six-core Intel(R) Xeon(R) CPU X5650 platform with 12 hardware threads running at 2.67 GHz and 32 GB of main memory.

Applications and datasets. We evaluated iThreads with applications from two benchmark suites: PARSEC [24] and Phoenix [74]. Table 1 lists the applications evaluated and their
When comparing the performance, we use the same number of threads in iThreads which re-compute everything from scratch. We report the normalized input size is also shown by the line plot in §6.3. Overall, the results show that iThreads is effective across a wide range of applications, but also that the library is not a one-size-fits-all solution.

As expected, we observed that increasing the number of threads tended to yield higher speedups. This is because, for a fixed input size, a larger number of threads translates to less work per thread. As a result, iThreads is forced to recompute fewer chunks when a single input page is modified.

Note that work speedups do not directly translate into time speedups. This is because even if just a single thread is affected by changes, the end-to-end runtime is still dominated by the (slowest) invalidated thread’s execution time.

6.1 Performance Gains

We first present a comparison of iThreads’s incremental run with pthreads and Dthreads, as shown in Figures 7 and 8 respectively. In this experiment, we modified one randomly chosen page of the input file prior to the incremental run. We then measured the work and time required by iThreads’s incremental run, as well as by pthreads and Dthreads, which re-compute everything from scratch. We report the work and time speedups (i.e., iThreads’s performance normalized by the performance of pthreads/Dthreads) for a varying number of threads ranging from 12 to 64 threads. When comparing the performance, we use the same number of threads in iThreads and pthreads/Dthreads.

The experiment shows that the benefits of using iThreads vary significantly across applications. In over half of the evaluated benchmarks (7 out of 11), iThreads was able to achieve at least 2X time speedups. In contrast, for applications such as canneal and reverseindex, iThreads can be very inefficient, by a factor of more than 15X, an effect that we explain in further detail in §6.3. Overall, the results show that iThreads is effective across a wide range of applications, but also that the library is not a one-size-fits-all solution.

As expected, we observed that increasing the number of threads tended to yield higher speedups. This is because, for a fixed input size, a larger number of threads translates to less work per thread. As a result, iThreads is forced to recompute fewer chunks when a single input page is modified.

Note that work speedups do not directly translate into time speedups. This is because even if just a single thread is affected by changes, the end-to-end runtime is still dominated by the (slowest) invalidated thread’s execution time.

6.2 iThreads Scalability

In a second experiment, we investigated the scalability of iThreads w.r.t. increases in the size of the input, the amount of computation (work), and the size of the input change.

Input size. We first present the performance of iThreads as we increase the input data size for the three application benchmarks (histogram, linear regression, and string match) that are available in three input sizes: small (S), medium (M), and large (L). (We used the large size in §6.1.) Figure 9 shows a bar plot of the work and time speedups w.r.t. pthreads for different input sizes (S, M, L) with a single modified page for 64 threads. For reference, the normalized input size is also shown by the line plot in the same figure. In summary, this result shows that speedups increase with the input size due to increased work savings.
Computation (work). We next present iThreads’s incremental run performance for two applications (swapations and blackscholes) that allow the amount of work required to be tuned with a parameter. Figure 10 reports work speedups as the normalized work is increased (from 1X to 16X) for a single modified page and 64 threads. The result shows the gap between pthreads and iThreads widens as the total work increases, which directly translates to higher speedups.

Input change. Finally, we present iThreads’s incremental run performance in the case of multiple modified input pages. To avoid confining changes to a single thread, we modified multiple non-contiguous pages of the input that are read by different threads. Figure 11 shows speedups w.r.t. pthreads with different change sizes (ranging 2 to 64 dirty pages) for 64 threads. As expected, the results show that speedups decrease as larger portions of the input are changed because more threads are invalidated.

6.3 Overheads

iThreads imposes two types of overheads: (1) space overheads; and (2) performance overheads during the initial run.

Space overheads. Table 1 shows the space overheads for memoizing the end state of the thunks and storing the CDDG. We report the overheads in terms of 4KB pages for 64 threads (space overhead grows with the number of threads). To put the overheads into perspective, we also report overheads as a percentage of the input size.

The space overheads varied significantly across applications. We found that three applications (canneal, swapations and reverse-index) incur very high overheads (exceeding 1000% of the input size), but, interestingly, nearly half of the applications (5/11) have a very low overhead (ranging from 0.1% to 10% of the input size).

Performance overheads. We measured iThreads’s performance overheads during the initial run (in terms of work and time) by comparing it against both pthreads and Dthreads (Figures 12 and 13). Our results show that most of the applications (7/11) incur modest overheads when compared with either pthreads (i.e., lower than 50%) or Dthreads (i.e., lower than 25%). In fact, linear-reg and string-match even performed better during the initial run of iThreads than with pthreads, which is explained by the fact that private address space mechanism avoid false sharing, as previously noted by Sheriff [62]. At the other end of the spectrum, applications such as canneal and reverse-index incur high overheads mainly due to the high number of memory pages written by these applications (as shown in Table 1).

When compared to Dthreads as the baseline, iThreads incurs work overheads of up to 3.58X and time overheads of up to 3.13X. iThreads incurs additional overheads on

![Figure 9. Scalability with data (work and time speedups)](image)

![Figure 10. Scalability with work](image)

![Figure 11. Scalability with input change compared to pthreads for 64 threads](image)

<table>
<thead>
<tr>
<th>Application</th>
<th>Input size</th>
<th>Memoized state</th>
<th>CDDG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Histogram</td>
<td>230400</td>
<td>347 (0.15%)</td>
<td>57   (0.02%)</td>
</tr>
<tr>
<td>Linear-reg.</td>
<td>132436</td>
<td>192 (0.14%)</td>
<td>33   (0.02%)</td>
</tr>
<tr>
<td>Kmeans</td>
<td>586</td>
<td>1145 (195.39%)</td>
<td>27   (4.61%)</td>
</tr>
<tr>
<td>Matrix-mul.</td>
<td>41609</td>
<td>4162 (10.00%)</td>
<td>64   (0.15%)</td>
</tr>
<tr>
<td>Swapations</td>
<td>143</td>
<td>1473 (103.07%)</td>
<td>1    (0.70%)</td>
</tr>
<tr>
<td>Blackscholes</td>
<td>155</td>
<td>201 (129.68%)</td>
<td>1    (0.65%)</td>
</tr>
<tr>
<td>String match</td>
<td>132436</td>
<td>128 (0.10%)</td>
<td>33   (0.02%)</td>
</tr>
<tr>
<td>PCA</td>
<td>140625</td>
<td>3777 (2.69%)</td>
<td>43   (0.03%)</td>
</tr>
<tr>
<td>Canneal</td>
<td>9</td>
<td>15381 (17900.00%)</td>
<td>4   (44.44%)</td>
</tr>
<tr>
<td>Word count</td>
<td>12811</td>
<td>10191 (79.55%)</td>
<td>24   (0.19%)</td>
</tr>
<tr>
<td>Rev-index</td>
<td>359</td>
<td>260679 (72612.53%)</td>
<td>64  (17.83%)</td>
</tr>
</tbody>
</table>
top of Dthreads mainly from two sources: memoization of the intermediate address space state and read page faults (Dthreads incurs write faults only). We show the work overheads along with a breakdown of these two sources of overheads with respect to Dthreads for 64 threads in Figure 14. The overheads are dominated by read page faults (around 98%) for most applications. For instance, histogram incurs overheads of roughly 3.5X due the large number of page faults while reading a large input file (as shown in Table 1). In contrast, some application such as canneal and reverse-index suffer a significant overhead for memoization (around 24%) due to a large number of dirtied pages.

6.4 Case-study Applications
In addition to the benchmark applications, we report the performance gains for two case-study applications: (1) Pigz [3], a parallel gzip compression library compressing a 50MB file, and (2) a monte-carlo simulation [2]. To compute speedups, we modified a random input block and compared the performance of the DThreads incremental run with the pthreads run. Figure 15 shows the work and time speedups with a varying number of threads (from 12 to 64). The performance gains peak at 24 threads for both applications. In particular, DThreads achieves a time speedup of 1.45X and a work speedup of 4X for Pigz, and a time speedup of 2.28X and a work speedup of 22.5X for the monte-carlo simulation.

To conclude, while there exist specific workloads for which our OS-based approach is not suitable, our evaluation is overall positive: DThreads is able to achieve significant time and work speedups both for many of the benchmark applications and also for the two considered case-studies.

7. Related Work
Researchers in the algorithms community proposed several dynamic algorithms, a class of algorithms that take advantage of application-specific properties to incrementally update the output. Dynamic algorithms have been shown to be asymptotically more efficient than their conventional non-dynamic versions (e.g., [25, 30, 37, 38, 42, 46, 68]). However, they can be difficult to design, implement, and maintain even for simple problems [10, 45]. Moreover, most dynamic algorithms are sequential, and cannot be easily parallelized due to their highly specialized nature. In contrast, DThreads provides incremental computation in a transparent way.

Incremental computation is a well-studied area in the programming languages community; see [73] for a classic survey. Earlier work on incremental computation was primarily based on dependence graphs [36, 53] and memoization [5, 51, 72]. In the past decade, with the development of self-adjusting computation [6–8, 29, 49, 50, 60, 61], the efficiency of incremental computation has much improved. In contrast to DThreads, however, most prior work in this area targets sequential programs only. Nonetheless, DThreads’s central data structure, the CDDG, is based on these foundations.

For supporting parallel incremental computation, existing proposals [27, 48] require a strict fork-join programming model without supporting other synchronization primitives. Furthermore, these proposals rely on the use of a new programming language with special data types (e.g., isolation,
versioned, cumulative, merge function [27] and read, write, mod, and letpar [48]). In contrast, our approach targets unmodified multithreaded programs supporting the full range of synchronization primitives in the POSIX API.

In very recent work, Tseng and Tullsen [78] proposed compiler-based whole-program transformations to eliminate redundant computation, which can be leveraged for faster incremental computation. The transformed programs rely on underlying hardware [76] and software [77] support to dynamically identify redundant code that can be skipped. In contrast, our approach directly operates at the binary level without requiring access to source code. A further design difference is that iThreads realizes incremental computation based on explicit change propagation, and that iThreads memoizes and reuses intermediate results of previous runs.

Incremental processing of “big data” is an active area of research [18–20, 22, 23, 26, 31, 47, 64, 70]. These “big data” systems exploit the underlying data-parallel programming model such as MapReduce [35] or Dryad [55] for supporting incremental computation. In contrast to these structured approaches where the dependence graph is explicitly available based on the programming model, iThreads is designed to support general shared-memory multithreaded programs.

In the context of increased reliability, prior work has yielded a large range of solutions to eliminate non-determinism from multithreaded programs. Most relevant to iThreads are the wide range record and replay techniques (e.g., [12, 41, 52, 57, 59, 69, 75, 79–81]) and deterministic multithreading approaches (e.g., [13–15, 17, 32–34, 39, 40, 54, 63, 67]). As described throughout the paper, these proven techniques are leveraged by iThreads, which applies them in a novel context, namely transparent parallel incremental computation.

8. Conclusion

We have presented iThreads, a practical, transparent, and efficient solution for parallel incremental computation. Our approach targets unmodified, multithreaded programs and supports the full range of synchronization primitives in the POSIX API. The iThreads library is easy to use: it simply replaces the pthreads library. Our experience with iThreads shows that significant performance gains (time savings) and efficient resource utilization (work savings) can be achieved for incremental workflows in many applications.

Limitations and future work. While iThreads is a significant step towards general and practical support for parallel incremental computation, plenty of opportunities remain to further increase the range of supported workloads.

For one, iThreads’s memory model currently lacks support for ad-hoc synchronization mechanisms [82]. While such mechanisms are error-prone [82], they are nonetheless used for either flexibility or performance reasons in some applications. Replacing the ad-hoc synchronization calls with equivalent pthread calls might solve the problem in some cases, but perhaps a better solution would be to extend iThreads with an interface for annotating ad-hoc synchronization primitives (e.g., at the level of gcc’s built-in atomic primitives).

Another interesting research challenge is improving support for small, localized insertions and deletions in the input data. Since iThreads is currently tuned for in-place modifications of the input data, insertions and deletions lead to the displacement of otherwise unchanged data, which causes an excessively large dirty set. Prior work has solved the displacement problem in the context of data-deduplication by replacing fixed-size input chunking with variable-size, content-based chunking [21, 66]. We plan to explore similar approaches in the context of iThreads.

Lastly, our current implementation assumes the number of threads in the system remains the same. However, our approach can be extended to handle dynamically varying number of threads by considering newly forked threads or deleted threads as invalidated threads, where the writes of deleted threads are handled as “missing writes”. The happens-before relationship for dynamically varying number of threads can be detected using interval tree clocks [11].

Acknowledgments

We are thankful to Remzi Arpaci-Dusseau, Rose Hoberman, Tongping Liu, and the reviewers for their valuable feedback. Umut Acar is partially supported by ERC (ERC-2012-StG-308246) and NSF (CCF-1320563 and CCF-1408940). Rodrigo Rodrigues is partially supported by ERC (ERC-2012-StG-307732).
References


[57] O. Laadan, N. Viennot, and J. Nieh. Transparent, lightweight application execution replay on commodity multiprocessor operating systems. In proceedings of the ACM SIGMETRICS
of the ninth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP), 2003.


