Lazy Tree Mapping: Generalizing and Scaling Deterministic Parallelism

Yu Zhang (yuzhang@ustc.edu.cn)
University of Science and Technology of China

Bryan Ford (bryan.ford@yale.edu)
Yale University

Abstract

Many parallel programs are intended to yield deterministic results, but unpredictable thread or process interleavings can lead to subtle bugs and nondeterminism. We are exploring a *producer-consumer* memory model—SPMC—for efficient system-enforced deterministic parallelism. However, the previous *eager page mapping* wastes physical memory, and cannot support large-size and real applications. This paper presents a novel *lazy tree mapping* approach to the model, which introduces "shadow page table" for allocating pages "on demand", and extends an SPMC region by a tree of lazily generated pages, representing an infinite stream on reusing a finite-size of virtual addresses. We build DLINUX to emulate the SPMC model entirely in Linux user space to make the SPMC more powerful. DLINUX uses virtual memory to emulate physical pages, and sets up page tables at user-level to emulate lazy tree mapping. Atop the SPMC, DetMP and DetMPI are explored and integrated into DLINUX, offering both thread- and process-level deterministic message passing programming. Experimental evaluations suggest *lazy tree mapping* improves memory use and address reuse. DLINUX scales close to ideal with 2048*2048 matrices for matmult, and better than MPICH2 for some workloads with larger input datasets.

1 Introduction

While many parallel programs are intended to be deterministic, unpredictable thread or process interleaving can lead to bugs and nondeterminism [1, 12, 13], which makes it difficult to write and debug parallel programs. Experimental languages such as DPJ [6] and SHIM [9] have explored the appeal and benefits of "deterministic by default" programming models, offering promising alternatives, but require developers to adopt unfamiliar coding styles, type systems or parallel constructs.

Deterministic runtimes [2, 3, 5, 7] can reproducibly execute code written in existing languages, but making them scale remains a challenge [16]. Grace [5] and Determinator [2] support only hierarchical synchronization such as *fork/join*, making interactions between child threads and their common parent a likely scalability bottleneck. CoreDet [3] and TERN [7] support arbitrary synchronization primitives, but a central *scheduler* in them synthesizes and imposes on all threads an artificial "time" schedule, becoming a scalability bottleneck. To be viable in the long term as core counts increasingly drive "the new Moore's Law", deterministic runtimes will need to support non-hierarchical, "peer-to-peer" communication and synchronization patterns, without introducing bottlenecks such as centralized schedulers.

As a possible foundation for more scalable deterministic parallelism, we introduced a *producer-consumer* virtual memory model—SPMC [19]. As with traditional shared memory, multiple processes, each with its own private address space, can directly "share" an SPMC memory region through memory mappings. To make shared memory access deterministic, however, only one *producer* ever has write access to an SPMC region, and the remaining *consumers* cannot read a location in the region until the producer explicitly *fixes* it, rendering it read-only. Multiple consumers can read the same region, supporting multicast communication.

At first attempt, we enhanced the Determinator OS kernel to prototype SPMC region primitives, denoted xDet, and established a user-level library—DetMP—atop it, offering a high-level *channel* abstraction for deterministic message passing [19]. A straightforward

APSys '13, July 29-30 2013, Singapore, Singapore

Acronym	Description
xDet	Determinator extended with SPMC virtual memory
DLINUX	Linux shielded with SPMC model to ensure determinism
DetMP	a thread-level deterministic message passing library based on SPMC channel abstraction
DetMPI	a core subset of MPI implemented based on SPMC model

Table 1: Acronyms and their explanation.

eager page mapping was used in xDet to maintain producer-consumer page sharing relationship. That is, the kernel must allocate all physical pages for an SPMC region once it is shared, even if many of those pages have not yet been—and perhaps may never be—actually "touched" by the producer or consumers. Since each process has finite address space, if the producer could fix each page in a region only once, SPMC regions used for inter-process communication (IPC) would thus finally "run out of space", and become unusable for further IPC.

To save physical memory and make the SPMC model more powerful for scalable deterministic parallelism, we propose a novel *lazy tree mapping* approach, which contains two orthogonal techniques, *lazy page mapping* and *space extension*. The former introduces a *shadow page table* to initially maintain producer-consumer relationship on a potentially large SPMC region and allocates physical pages for the region only "on demand". The latter extends an SPMC region by an arbitrarily deep *tree* of lazily-generated pages, which can represent an infinite stream on reusing a finite-size of virtual addresses.

Since Determinator is 32-bit and limited to using at most 1GB of physical memory, xDet is restricted to only support small programs even using lazy tree mapping. To generalize the SPMC model and support larger and more realistic applications, we built DLINUX, which retrofits SPMC into Linux, analogous to the way dOS [4] retrofitted CoreDet into Linux. DLINUX aims to offer deterministic thread-level and process-level parallelism on Linux via virtual memory technology. For quick prototyping, DLINUX emulates SPMC with lazy tree mapping entirely in Linux user space, via disciplined use of conventional `mmap` memory. Beyond porting DetMP into DLINUX, we built DetMPI atop DetMP, offering backward compatibility with legacy MPI [15] applications. Table 1 lists these acronyms, where xDet and DLINUX are foundation systems, while DetMP and DetMPI are libraries integrated into both xDet and DLINUX.

Experiments with parallel workloads suggest that the *lazy tree mapping* approach significantly improves memory use and address space reuse in SPMC regions on both DLINUX and xDet. DLINUX can run a larger set of practical applications deterministically than xDet, and exhibits performance and scalability comparable to nondeterministic environments, such as a popular MPI implementation—MPICH2-1.4 [14] on matmult, wordfreq and rmat workloads with larger input datasets.

This paper makes three main contributions. First, we enhance our SPMC model with *lazy tree mapping* to make it powerful for scalable deterministic parallelism. Second, we build DLINUX, retrofitting SPMC into Linux, to offer backward compatibility and support more realistic applications. Third, we integrate two deterministic message passing API layers, DetMP and DetMPI, into both DLINUX and xDet, offering convenient and backward-compatible parallel programming models. Performance results on these prototypes indicate that the SPMC virtual memory foundation may be a realistic and useful approach to deterministic parallelism.

The remainder of the paper is structured as follows. Section 2 presents the SPMC model and *lazy tree mapping* approach. Section 3 presents DLINUX. Section 4 evaluates the prototype, and Section 5 concludes.

2 SPMC Memory Model

The SPMC memory model is intended to offer a virtual memory foundation for scalable deterministic parallelism at OS-level, supporting programming in existing languages, such as C. In addition to allow read-only memory sharing among processes via *copy-on-write*, it offers a deterministic read-write memory sharing, allowing a process P to establish direct "peer-to-peer" SPMC regions between its arbitrary descendants, eliminating P as a scalability bottleneck in subsequent computation.

2.1 Basic Semantics of SPMC Regions

An SPMC region is a piece of restricted read-write sharing virtual memory. It is introduced by distinguishing two types of memory mappings mapped to the same physical pages, as shown in Fig. 1. A given shared physical page has only one *producer mapping* at a time in one process's address space, and any number of *consumer mappings* in other processes, however. An SPMC implementation should enforce a protocol that: 1) consumers have no access to an SPMC page while the producer is writing to it, 2) the producer can write to an SPMC page many times but then fix it at most once, and 3) once the producer explicitly *fixes* an SPMC page, it loses write permission and becomes a consumer, while all consumers then have read permission to it. The protocol constrains SPMC regions to guarantee determinism despite in the presence of this "peer-to-peer" communication. SPMC regions thus form communication primitives analogous to Kahn process networks [10].

An SPMC implementation can use page-based address translation to give each process an independent address space, enforcing inter-process protection and isolation. It can track and enforce memory access control through page-level protection bits in page table entries (PTE).

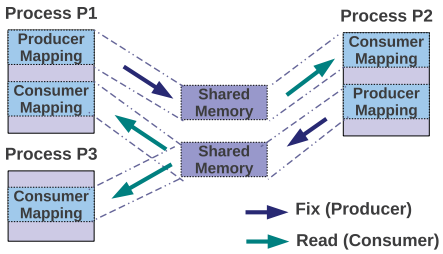

Figure 1: Processes sharing SPMC regions.

2.2 Eager vs. Lazy Page Mapping

One key design issue is how to efficiently maintain producer-consumer page sharing relationship. Our previous *eager page mapping* policy implemented in xDet requires that physical pages for an SPMC region should be allocated once it is shared, even if many of those pages have not yet been—and perhaps may never be—actually "touched" by the region's producer or consumers.

To conserve physical memory, we design a *lazy page mapping* policy, where physical pages for an SPMC region are allocated only "on-demand". When a process creates an SPMC region, mappings in the region are not filled with "real" pointers to physical pages (since they haven't been allocated yet), but rather with pointers to a common *shadow page table* (see Fig. 2(a)). The shadow page table is a page that mirrors the structure of a normal lowest-level page table, containing a list of page mappings—all initially null (unallocated)—to be filled incrementally as pages in the region are touched. For SPMC pages not yet allocated, the sharing relationship is among the set of processes with mappings pointing to the same shadow page table. Mappings pointing to shadow page tables always have their "valid" flags clear, so that application code can never access shadow page tables directly. Any attempt to access such a mapping from the producer or a consumer causes a page fault. If it comes from the producer, the touched unallocated page is allocated, and the faulting PTE as well as the corresponding shadow PTE are replaced with a regular page mapping of the newly allocated page (see P1 in Fig. 2(b)). When a consumer subsequently attempts to read the unfixed page, it still causes a page fault, at which point page fault handler looks up the actual page mapping in the shadow page table, and replaces consumer's mapping with a mapping of the actual page (see Fig. 2(c)).

If a consumer attempts to read an SPMC page before the producer has touched it, the page fault handler blocks the consumer and adds it to a wait list whose "head pointer" is kept in the corresponding shadow PTE—which is available for this purpose because it does *not* yet hold a pointer to an actual page. Once the producer touches an address corresponding to the shadow PTE, a new page is allocated and the wait list is moved from the shadow PTE into the per-page metadata structure associated with the new page. Once the producer fixes the page, all waiting consumers can then be awakened.

With this shadow page table mechanism, for an SPMC region of n-page size, initially only a shadow page need be allocated on creation, then actual pages are allocated on demand, avoiding allocating all n pages in advance. In the common case in which an application creates large SPMC regions for efficient communication, but often only ever uses a few pages in the region, the lazy page mapping policy can save large amounts of memory.

2.3 Space Extension via Tree Mapping

Since each process has finite address space, if the producer can only fix each virtual address at most once, SPMC regions used for IPC would thus eventually "run out of space", and become unusable for further IPC. Allowing a producer to "un-fix" and "re-fix" previously-fixed pages would introduce tricky synchronization challenges and make it harder to ensure determinism. To avoid these challenges, instead a producer is allowed to *extend* an SPMC region to represent as not just a "flat" list of pages, but an arbitrarily deep logical tree of lazily-generated pages, which can represent an infinite stream like a pipe or socket, or any lazy data structure.

To extend an SPMC region, a producer avoids writing to or fixing one or more pages in the region, reserving these pages instead as *extension pages*. When the producer needs more pages—*eg.,*when it has filled and fixed the rest of the region—the producer invokes an *extend* call to expand a reserved extension page in the existing region, to form a new set of producer mappings representing fresh pages. The producer can expand these fresh mappings either into a different part of the producer's address space, or into the same address range the original region occupied if the producer is finished with the existing region, thereby reusing the address range to represent a new "generation" of fresh SPMC pages. These fresh producer mappings inherit all the properties of the original ones, including their usability either as regular or (recursive) extension pages: hence the capability to represent infinite streams or trees.

Producers and consumers must agree on which pages are regular data pages and which are extension pages. A consumer must not attempt to read an extension page (doing so causes a trap), but expands the page into a corresponding, fresh set of consumer mappings—in either the same or a different range of the consumer's address space—just as the producer does. The expanded mappings then become regular consumer mappings, linked to the corresponding producer mappings, which the consumer can in turn read or recursively expand.

When a producer calls *extend*, the handler creates an

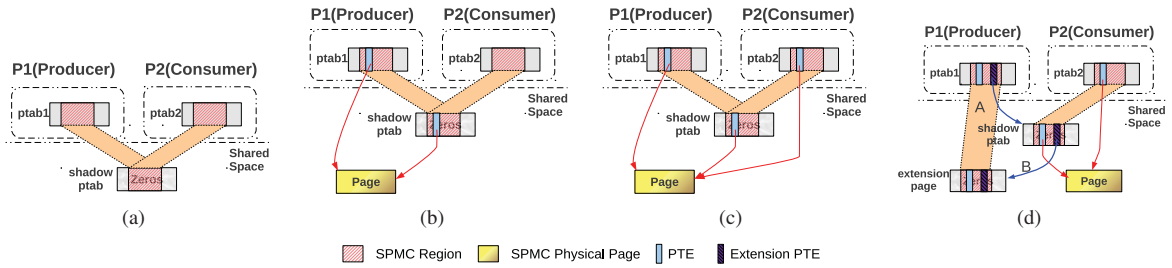

Figure 2: Mechanism of SPMC virtual memory with lazy page mapping and space extension.

extension page and links the old shadow page table to it (see B in Fig. 2(d)), then creates fresh producer mappings pointing to the new extension page (see A in Fig.(d)). The extension page henceforth becomes simply a "next-generation" shadow page table for the producer. The expanded mappings may overwrite previous SPMC mappings if the extension is reusing virtual addresses. Overwriting the original mappings, however, will not overwrite the original shadow page table or the original pages it pointed to, which consumers may still need to read.

When a consumer calls *extend*, the handler first tests whether the corresponding shadow PTE has been "produced" as either a data page or an extension page, and if not, blocks the consumer in the usual way until the page is produced. If the page was produced as a data page—indicating that producer and consumer "disagree" about the page's use—then the consumer takes a trap. If the page was produced as an extension page, however, the handler then creates a list of consumer mappings pointing to that extension page, just as the producer does. All pages, including data and extension pages, as well as shadow page tables, are reference-counted, so a shadow page table and the pages it refers to get garbage-collected once the producer and all consumers have overwritten or otherwise destroyed all mappings referencing them.

3 DLINUX:Linux-based Implementation

We now describe DLINUX, a system built on Linux to offer the SPMC virtual memory foundation for scalable deterministic parallelism. DLINUX (see Fig. 3) currently includes five parts, *i.e.,* SPMC-based thread/process—*space*—management, SPMC region primitives, the DetMP and DetMPI layers, and some microbenches. The former two form the core layer of DLINUX, and are emulated entirely in Linux user space for quick prototype. The latter three parts can be reused in either DLINUX or xDet with a few modifications. Overall, we wrote roughly 13500 lines of C code to implement DLINUX, including 3000 lines of code for the core layer, roughly 2000 lines for the DetMP, roughly 4000 lines for the DetMPI as well as an additional perl script for generating Fortran MPI interface and wrapper function files, and roughly 4500 lines for the microbenches.

Spaces. DLINUX executes application code within an arbitrary hierarchy of *spaces*. Term "space" is followed by Determinator to highlight differences from Linux processes or `pthreads`, avoiding confusion with "process" and "thread" abstractions DLINUX emulates. Determinator gives a *space* no physically shared memory but read-only sharing via copy-on-write; it emulates shared memory using "*copy-at-fork, merge-at-join*" techniques. While DLINUX allows sharing SPMC regions among spaces, not merely read-only sharing. Similar to Grace [5] DLINUX emulates *spaces* using Linux processes to achieve cross-space memory isolation by default. DLINUX requires to manage shared memory explicitly, and provides a set of API for thread-/process-level programming by wrapping spaces.

Deterministic Space Index. Linux does not guarantee the value of a process ID deterministic. To avoid exposing this nondeterminism to spaces running as Linux processes, DLINUX returns a unique internal space index to a space, which is managed using a counter field in a single global structure. The index is also used to manage per-space heaps and as offset into an array of space entries in the global structure. The global structure is kept in an `mmap` shared memory, and initialized in `spmc_init()`, which is directly or indirectly called by application code. For example, `spmc_init()` in DetMPI might be called by `MPI_Init()` or `mpiexec` to initialize DLINUX, depending on how to launch MPI processes. For simplicity, DLINUX fixes total number of spaces, specified by the argument of `spmc_init()`.

Heap. DLINUX supports memory allocation by giving each space a separate sub-heap, managed by a variant of Doug Lea's malloc [11] with about 170 modified lines of code. When receiving the first allocate request from application code before `spmc_init()` call or calling `spmc_init()` before any allocate requests, DLINUX initializes a fixed size of `mmap` private memory as the whole area for all sub-heaps. DLINUX considers the above two cases to ensure all sub-heaps occupy disjoint memory, and creates an *mspace* (used in Doug Lea's malloc) for each space to ensure addresses of the *mspace* are indeed in the space's sub-heap range.

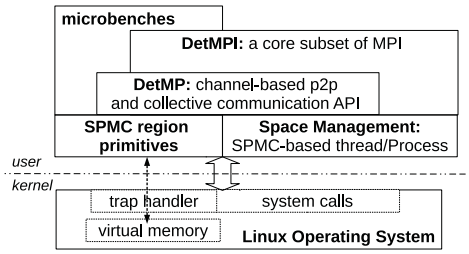

Figure 3: System overview.

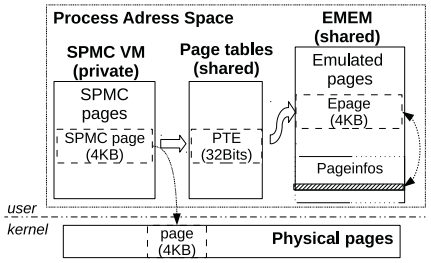

Figure 4: Emulating page mapping in Linux user space.

SPMC Shared Memory. DLINUX cannot control actual page mappings in Linux kernel due to emulating SPMC regions entirely in user space. To emulate the page mapping policies in Section 2, DLINUX creates an `mmap` shared memory—EMEM—to emulate physical memory, and establishes per-space page tables to map each space's SPMC regions to the EMEM (see Fig. 4).

To ensure an unfixed write to an SPMC region invisible to all consumers, a private `mmap` memory is created as SPMC VM in the root space, and copied to child spaces on space creation, ensuring SPMC VMs in all spaces occupy the same addresses and isolate each other. The shared EMEM consists of an array of emulated physical pages—*epages*, and a corresponding array of *pageinfos* containing per-page metadata, totally emulating physical memory management needed in Section 2. A 4KB epage can be allocated as an SPMC data page, a shadow page table, or an extension page.

For simplicity, DLINUX creates a shared `mmap` memory to manage per-space one-level page tables, mapping a space's SPMC VM to epages. An SPMC page has a corresponding 32-bit PTE in per-space page table, where Bit 31 is always 0, Bits 3 through 30 provide index of an *epage* in EMEM, and Bits 0 through 2 are P, W, O flags in turn to control access to an epage. P flag, *i.e.,*valid flag, indicates whether the epage is present or not, W flag indicates whether the epage is writable or not, and O flag indicates whether it is a producer PTE or not. We use 32-bit PTE just for memory saving, and based on this design, DLINUX can already support up to 2^{28} 4KB-epages.

Implementation of SPMC Regions. DLINUX provides a set of SPMC region primitives as follows.
- spmcR_alloc(*va*, *size*), which allocates a specified offset region [*va*, *va+size*) in the current space's SPMC VM.
- spmcR_transown(*sid*, *sva*, *dva*, *size*), which transfers producer mappings from the current space to space *sid*, making the current space a consumer.
- spmcR_copycons(*sid*, *sva*, *dva*, *size*), which copies consumer mappings from the current space to space *sid*.
- spmcR_setfix(*va*, *size*), which fixes the specified region in current space, setting it read-only, then awakens each waiting consumer.
- spmcR_extend(*extva*, *va*, *size*), which extends region [*va*, *va+size*) via an extension page at *extva*.

DLINUX elaborates access permission transitions for SPMC regions via disciplined use of `mprotect` system calls. Thus some read/write operations on SPMC regions from application code would trigger segmentation faults, the control is then transferred to the segmentation fault handler, emulating page faults and control transfer needed in Section 2. In response to SPMC primitive calls and segmentation faults, DLINUX performs page mapping and SPMC region protection operations.

When an SPMC region is created, it is set to PROT_NONE via `mprotect`. Any attempt to access PROT_NONE memory from application code triggers a segmentation fault, DLINUX then allocates a data epage mapped to the faulting PTE in the segmentation fault handler. Once a producer PTE has mapped to a data epage, the corresponding SPMC page in the producer's SPMC VM is upgraded to PROT_WRITE, so that the producer can directly write to the page. When the producer fixes an SPMC page, DLINUX first copies the content from the SPMC page to the corresponding epage, so that consumers can access later, then downgrades the SPMC page to PROT_READ, finally awakens each consumer waiting for the page using `kill()` system call.

For a consumer, an SPMC region is set to PROT_NONE initially. Any attempt to access the region from application code triggers a segmentation fault, then the handler checks whether the corresponding data epage has been fixed or not. If not, `sigsuspend()` system call is invoked to block the consumer. Once the blocked consumer is awakened or the data epage is fixed, DLINUX copies the content from the epage to the SPMC page, and sets the SPMC page PROT_READ. Thus the consumer can read data directly from its SPMC pages.

DetMP DetMP atop SPMC regions, offers a high-level *channel* abstraction for point-to-point and collective message passing programming.

We originally developed DetMP atop xDet [19]. Each channel is implemented as a fixed-size SPMC region holding a series of messages sent by the producer. Each message is internally page-aligned, occupying a contiguous range of pages in the region, and is fixed at page granularity. DetMP maintains metadata to record status

of active channels, such as offsets at which a producer puts or a consumer gets the next message in the SPMC region. For quick prototype, our algorithms on collective communications are straightforward and unoptimizable.

By means of space *extension* mechanism, we upgraded DetMP to support sending a message larger than available capacity in the channel. We also ported the updated DetMP into DLINUX by updating inconsistent SPMC region primitives and system calls used.

DetMPI We develop DetMPI atop DetMP, a deterministic runtime offering backward compatibility with legacy MPI programs in C or Fortran, hiding channels from users. DetMPI provides MPI object management, most MPI's *point-to-point* communication and *collective* communication functions within intra-communicators, process management simply implementing `mpiexec`, and Fortran binding module. It has been integrated into both xDet and DLINUX. Due to limitations on space, we only discuss channel management in DetMPI.

To manage channels implicitly used in communication for any MPI application, DetMPI classifies channels into several kinds: P2P (*process-to-process*), M2S (*master-to-one-slave*), S2M (*one-slave-to-master*), P2N (*process-to-next*), P2A (*process-to-all*), M2A (*master-to-all*) and S2A (*slave-to-all*). DetMPI allocates channels and assigns them to MPI processes as part of `MPI_Init()`. `mpiexec` in DetMPI allows users to configure channel kinds that might be used in running an MPI program, *eg.,*

mpiexec -p2p -p2a -s2m -n 4 is.W.4

starts is.W.4 with an MPI_COMM_WORLD whose group includes 4 processes, and creates P2P, P2A, S2M channels.

The default channel kinds include the universal set, *i.e.,* M2S, S2M, P2P, P2A, supporting all DetMPI communication operations. For n processes, the total number of channels is at most (n^2+n-1), including (n-1) M2S, (n-1) S2M, (n-1)^2 P2P and n P2A channels.

4 Evaluation

4.1 Workloads

NPB-MPI benchmark suite [8]. It contains two programs in C and seven programs in Fortran. Each workload includes S,W,A,B,C input dataset classes. We omit C since some (*i.e.,*FT.C.1) are built error in C class.

wordfreq and rmat. These two workloads come from MR-MPI library [17], which is an implementation of MapReduce written in C++ on top of the standard MPI. Input datasets for wordfreq come from Phoenix[18], including 10MB, 50MB and 100MB input files. Since wordfreq decides the number of mappers according to the number of input files, we use 16 copies of each dataset to run wordfreq with at most 16 processes. Input datasets for rmat are rmat-20 (around 8M edges) and rmat-24 (around 134M edges), both of which use parameters (a, b, c, d) = (0.57, 0.19, 0.19, 0.05) and generate 8 edges per vertex (on average).

matmult-mp, matmult-mpi. These two are parallel matrix multiplication workloads implemented in DetMP and standard MPI APIs, respectively.

The experiments ran on a 32-core 2.0GHz 4×Xeon E7-4820 server with 128GB RAM. The Linux distribution is Ubuntu 12.04. Workloads were built as 64-bit executable programs with gcc v4.4.7 for MPICH2 and DLINUX, and 32-bit for xDet. Each workload is executed 10 times. To reduce the effect of outliers, the lowest and highest runtimes for each workload are discarded, so each result is the average of the remaining 8 runs.

4.2 Capability

First, we examine the importance of *space extension*. Assume 252MB SPMC memory is set aside for channel inodes. If no space extension, channel size has to be set large enough to hold all messages sent to the channel, thus a channel may occupy one or more inodes. If size of a channel inode is fixed to 4MB, at most 63 channels can be created, and support communication among less than 8 processes with default channel kinds. With space extension, a channel occupies just one inode, which size can be set smaller (but reusable), *eg.,*512KB, allowing up to 504 ($> 16^2 + 16 - 1$) channels. For rmat-24 with 2 processes on DLINUX, a channel once saves a single message of size 32.2MB, and total messages of 923.6MB.

Second, all NPB and MR-MPI workloads can be built and run unmodified on DLINUX, except changing a static array in wordfreq to be dynamically-allocated due to large size. But only NPB workloads in small dataset can run on xDet, MR-MPI workloads fail in building due to incomplete libc and libc++ support in xDet.

Third, we examine input datasets of NPB supported by xDet with or without space extension, and DLINUX, using up to 16 processes. Only IS, FT, and EP on class S work on xDet without space extension with 1–16 processes. While on xDet with space extension, all S,W workloads except LU and SP work with 1–16 processes. On DLINUX, all S, W, A, B workloads work with 1–16 processes, illustrating that DLINUX is the most powerful, and xDet without space extension is the weakest.

4.3 Single-node Multicore Performance

We now compare the performance of xDet and DLINUX, against the mature but nondeterministic MPICH2. Table 2 shows speedup and runtime relative to MPICH2 comparisons of matmult among xDet, DLINUX and MPICH2. We cannot obtain results of 2048*2048 matrices on xDet due to the memory limitation of xDet. From the table, we

	NProc	1024*1024					2048*2048		
		mpich	dlinux		xDet		mpich	dlinux	
		mpi	mp	mpi	mp	mpi	mpi	mp	mpi
Speedup	1	1	1	1	1	1	1	1	1
	2	1.96	1.89	1.89	1.98	1.97	1.87	1.83	1.78
	4	3.88	3.59	3.6	3.88	3.6	3.7	3.7	3.66
	8	7.36	6.49	6.46	7.48	3.18	6.16	7.18	7.13
	16	12.03	10.53	9.84	12.02	3.02	8.84	13.01	13.09
T / T_{mpich}	1	1.00	1.00	1.00	1.05	1.01	1.00	0.99	0.99
	2	1.00	1.04	1.00	1.01	1.01	1.00	1.01	1.02
	4	1.00	1.08	1.00	0.98	1.09	1.00	1.00	1.01
	8	1.00	1.13	1.00	0.91	2.37	1.00	0.85	1.00
	16	1.00	1.14	1.07	0.86	4.02	1.00	0.67	0.99

Table 2: Speedup and overhead matmult.

NProc	wordfreq						rmat			
	10MB*16		50MB*16		100MB*16		rmat-20		rmat-24	
	S_M	S_D	S_M	S_D	S_M	S_D	S_M	S_D	S_M	S_D
1	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
2	2.03	1.89	1.61	1.54	1.50	1.49	2.17	2.14	1.80	1.81
4	3.70	3.29	2.96	2.76	2.85	2.68	3.85	3.76	3.89	3.78
8	8.47	6.49	5.35	5.26	4.29	4.16	10.55	8.48	6.91	6.23
16	13.31	8.66	5.86	6.04	4.87	4.69	14.35	11.77	9.41	9.78
	$T_M(s)$	R	$T_M(s)$	R	$T_M(s)$	R	$T_M(s)$	R	$T_M(s)$	R
1	15.1	0.97	53.9	1.01	134.1	1.02	57.3	0.97	942.3	1.06
2	7.4	1.05	33.6	1.06	89.3	1.03	26.4	0.99	522.8	1.06
4	4.1	1.09	18.2	1.09	47.1	1.08	14.9	1.00	242.0	1.10
8	1.8	1.27	10.1	1.03	31.3	1.05	5.4	1.21	136.4	1.18
16	1.1	1.50	9.2	0.98	27.5	1.06	4.0	1.19	100.2	1.02

* S_M, S_D-speedup on mpich and on dlinux
T_M, T_D-runtime in seconds on mpich and on dlinux, $R = T_D/T_M$

Table 3: Speedup and overhead wordfreq and rmat.

find the performance and scalability of matmult-mp on either xDet or DLINUX are always better than matmult-mpi, and matmult-mpi meets serious performance problem on xDet. MPICH2 has the best performance for 1024*1024 matrices, but scales worse than DLINUX for larger input dataset—2048*2048 matrices.

Table 3 compares DLINUX against MPICH2 using wordfreq and rmat. DLINUX scales worse than MPICH2 for smaller dataset, *eg.,* rmat-20, but a little better than MPICH2 for larger dataset. Runtimes on DLINUX are a little longer but acceptable (mostly $< 1.1 \times T_{\text{mpich}}$) than MPICH2, since DLINUX emulates page mapping in user space, while MPICH2 directly uses shared memory for communication on multicore systems.

For NPB, due to space limitations, we do not present detailed results here, but summarize some conclusions we found. Currently only EP exhibits close to ideal scalability on DLINUX, and others scale lower than MPICH2. By modifying algorithms of `Alltoall` and `Alltoallv`, performance of IS and FT on DLINUX are improved a lot, illustrating room for improving algorithms on channel based collective communications. With input datasets becoming larger, runtimes on DLINUX move a step closer to those on MPICH2.

5 Conclusion

We believe that SPMC offers a promising virtual memory foundation for scalable deterministic parallelism. DetMP and DetMPI on DLINUX have shown the capability of lazy tree mapping, and good performance on some workloads with large input datasets. We also realize that communication algorithms in DetMP are very unoptimizable, causing poor performance on most NPB workloads. In the future we expect to optimize DLINUX and explore more efficient programming models atop SPMC.

Acknowledgments. The authors thank Huifang Cao for testing prerelease versions. This work was supported in part by the National High Technology Research and Development Program of China (863 Program) grant (No.2012AA010901), and the National Natural Science Foundation of China grant (No.61229201).

References

[1] Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. In *VVEIS*, pages 82–93, April 2003.

[2] Amittai Aviram et al. Efficient system-enforced deterministic parallelism. In *9th OSDI*, October 2010.

[3] Tom Bergan et al. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *15th ASPLOS*, March 2010.

[4] Tom Bergan et al. Deterministic process groups in dOS. In *9th OSDI*, October 2010.

[5] Emery D. Berger et al. Grace: Safe multithreaded programming for C/C++. In *24th OOPSLA*, October 2009.

[6] Robert L. Bocchino et al. A type and effect system for deterministic parallel Java. In *OOPSLA*, October 2009.

[7] Heming Cui et al. Stable deterministic multithreading through schedule memoization. In *9th OSDI*, October 2010.

[8] Rob F. Van der Wijngaart. NAS parallel benchmarks version 2.4. Technical Report NAS-02-007, NASA Ames Research Center, October 2002.

[9] Stephen A. Edwards et al. Programming shared memory multiprocessors with deterministic message-passing concurrency: Compiling SHIM to Pthreads. In *DATE*, March 2008.

[10] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing*, pages 471–475, Amsterdam, Netherlands, 1974. North-Holland.

[11] Doug Lea. A memory allocator, 2000.

[12] E.A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[13] Shan Lu et al. Learning from mistakes — a comprehensive study on real world concurrency bug characteristics. In *13th ASPLOS*, pages 329–339, March 2008.

[14] Mathematics and Computer Science Division Argonne National Laboratory. MPICH2-1.4: a high-performance and widely portable implementation of the MPI standard, June 2011.

[15] Message Passing Interface Forum. MPI: A message-passing interface standard version 2.2, September 2009.

[16] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Scaling deterministic multithreading. In *2nd WoDet*, March 2011.

[17] Steven J. Plimpton et al. MapReduce in MPI for large-scale graph algorithms. *Parallel Comput.*, 37(9):610–632, September 2011.

[18] Colby Ranger et al. Evaluating MapReduce for multi-core and multiprocessor systems. In *13th HPCA*, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[19] Yu Zhang and Bryan Ford. A virtual memory foundation for scalable deterministic parallelism. In *2nd APSys*, July 2011.