Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 2, No. 1, April 2016.

CHARACTERIZATION OF SHARED-MEMORY MULTI-CORE APPLICATIONS

Mohammed Sultan Mohammed[1] and Gheith A. Abandah[2]

Computer Engineering Department, the University of Jordan, Amman, Jordan
m.s.mohammed@ieee.org[1], abandah@ju.edu.jo[2]

(Received: 29-Nov.-2015, Revised: 21-Jan.-2016, Accepted: 01-Feb.-2016)

ABSTRACT

The multicore processor architectures have been gaining increasing popularity in the recent years. However, many available applications cannot take full advantage of these architectures. Therefore, many researchers have developed several characterization techniques to help programmers understand the behavior of these applications on multicore platforms and to tune them for better efficiency. This paper proposes an on-the-fly, configuration-independent characterization approach for characterizing the inherent characteristics of multicore applications. This approach is fast, because it does not depend on the details of any specific machine configuration and does not require repeating the characterization for every target configuration. It just keeps track of memory accesses and the cores that perform these accesses through piping memory traces, on-the-fly, to the analysis tool. We applied this approach to characterize eight applications drawn from SPLASH-2 and PARSEC benchmark suites. This paper presents the inherent characteristics of these applications, including memory access instructions, communication characteristics patterns, sharing degree, invalidation degree, communication slack and communication locality. The results show that two of the studied applications have high parallelization overhead, which are Cholesky and Fluidanimate. The results also indicate that the studied applications of SPLASH-2 have higher communication rates than the studied applications of PARSEC and these rates generally increase as the number of used threads increases. Most of the sharing and invalidation occurs in small degrees. However, two of SPLASH-2 applications have significant fraction of communication with high sharing degrees involving four or more threads. Most of the applications have some uniform communication component and the initial thread is generally involved in more communication compared to the other threads.

KEYWORDS

Multi-core processor, On-the-fly analysis, Shared memory applications, Communication patterns, Performance evaluation.

1. INTRODUCTION

The multicore architecture is the current and the foreseeable future approach that processor manufacturers use to build high-performance and low-power processors [1]-[2]. Most of the current processors are multicore processors; i.e., there are multiple processors on the processor chip. This approach is also the preferred approach in mobile devices [3]. Moreover, the number of cores on one chip increases with time [4]. To take advantage of the increasing number of cores, many parallel programming approaches were developed. One of these approaches is multithreading. Using parallel-multithreaded approach, various numbers of threads of one application can be concurrently executed on multiple cores and shared memory, thus facilitating implementing the algorithms that solve data-intensive problems such as searching and sorting [5]-[6]. Also, there are many applications that are developed to run on multicore systems, including some popular *benchmarks*. Nevertheless, there are many aspects that need to be tackled to improve multicore performance. Characterizing the benchmarks that represent

multicore applications on multicore systems is important to tune such applications for better performance and to design better multicore systems.

This paper proposes an on-the-fly configuration independent characterization approach for characterizing the inherent characteristics of multicore applications. This approach is carried out to characterize eight representative applications drawn from the popular SPLASH-2 and PARSEC application benchmarks. The proposed approach characterizes the inherent characteristics that are independent from any particular multicore configuration. The inherent characteristics are divided into two parts. The first part is the characteristics of the memory access instructions, which include the numbers of memory accesses and the percentages of memory accesses by type of access and access data size. This characterization is useful to find the amount of parallelization overhead. The second part is the communication characteristics, which include communication patterns, sharing degree, invalidation degree, communication slack and communication locality. It is important to characterize the communication characteristics of the multicore applications, as high communication overhead is often responsible of bad parallelization efficiency [7].

The rest of this paper is organized as follows: Section 2 presents a short survey of some related work. Section 3 summarizes our methodology in characterizing multicore applications and the experimental setup used. Section 4 presents the characterization results. Finally, Section 5 presents some conclusions and future work.

2. RELATED WORK

Several studies have proposed different techniques to characterize parallel applications. These techniques are summarized in the following four categories.

2.1 Hardware-Assisted Characterization

Many characterization studies have used hardware performance counters, which are special registers on the processor that count hardware events to characterize various aspects of running applications.

Dongarra *et al.* used these counters to characterize data cache and translation lookaside buffer (TLB) behaviors of their microbenchmarks [8]. Bhadauria *et al.* characterized PARSEC on multiple aspects, including cache performance, sensitivity to DRAM speed and bandwidth, multithread scalability and micro-architecture design choices on a variety of real multicore systems [9]. Ferdman *et al.* used these counters to study the micro-architectural behavior of their CloudSuite benchmarks [10]. They concluded that existing processor micro-architectures are inefficient for running their benchmarks. Jia *et al.* also used these counters to characterize eleven data analysis workloads of a data center to determine their micro-architectural characteristics on systems equipped with modern superscalar, out-of-order processors [11]. They also developed a benchmark suite called DCBench to mimic typical data center workloads.

2.2 Message-Passing Characterization

Instrumented message-passing libraries are often used to characterize parallel applications running on multi-computer systems.

Cohen and Mahafzah proposed a utility to characterize NAS benchmarks, which are a group of programs developed by NASA Ames to help evaluate the performance of parallel supercomputers. Their results provide a deep look at how NAS benchmarks work on parallel computers [12]. Alam *et al.* characterized the scaling behavior of a set of micro-benchmarks, kernels and scientific workloads on HPC systems [13]. They used AMD Opteron multicore

processors and concluded that the Opteron cache coherence protocol is insufficient to exploit the full bandwidth capability of the memory interface. Chai *et al.* characterized micro-benchmarks and application-level benchmarks on an Intel dual-core cluster [14]. They suggested that the communication middleware and applications should be multicore to optimize intra-node and inter-node communication.

2.3 Configuration Dependent Analysis

Configuration dependent characterization techniques characterize applications depending on simulating the application execution on a specific system configuration. This approach is widely used in characterizing applications.

Abandah developed a configuration dependent analysis tool (CDAT) to characterize shared memory behavior, including cache misses and sharing that depend on system configuration parameters such as cache block size [15]. CDAT is a multiprocessor system simulator that has memory, cache, bus and interconnection models. By using a configuration file, users can specify the system configuration, including the coherence protocol, size and speed of the system components, as well as processors and memory banks interconnections. Jaleel *et al.* used a dynamic binary instrumentation tool as an alternative to the trace-driven and execute-driven approaches [16]. They proposed a memory system simulator to characterize memory performance of x86 workloads on multicore systems. Contreras and Martonosi characterized a subset of PARSEC benchmark applications that were compiled with Intel TBB on AMD dual-core processors in order to determine the sources of overhead within the TBB [17]. Bienia *et al.* characterized PARSEC applications and found that they have various types of multithreaded behaviors. Bhattacharjee and Martonosi characterized TLB behavior of the PARSEC [18]. Dey *et al.* also characterized PARSEC and measured the effect of shared resource contention on performance [19]. They classified resource contention into intra-application contention, which is the contention among threads from the same application, and inter-application contention, which is the contention among threads from different applications. Natarajan and Chaudhuri characterized a set of multithreaded applications selected from PARSEC, SPEC OMP and SPLASH-2 to understand last-level cache (LLC) behavior of multithreaded applications [20]. They proposed a generic design that introduces sharing-awareness in LLC replacement policies. They showed that their design could significantly improve the performance of LLC replacement policies.

2.4 Configuration Independent Analysis

The configuration independent characterization technique is a unique technique for characterizing the inherent application characteristics that do not change when changing the system configuration.

Abandah and Davidson developed a configuration independent analysis tool (CIAT) to characterize the configuration independent characteristics, such as memory access instructions, concurrency, communication patterns and sharing behavior of shared-memory applications that run on multiprocessor systems [21]. CIAT analyzes the memory traces of shared-memory applications to find these characteristics. It is faster than detailed simulators, as it only keeps track of accesses to each memory location and does not include detailed models of a specific system's components and protocols [15]. Moreover, its characterization is general; it gives the inherent characteristics of the application that do not depend on a machine configuration. Thus, CIAT gives us a more general understanding of the application behavior.

This work ported CIAT, which was originally developed for RISC multiprocessor systems, to commodity multicore systems. The ported tool was used to characterize the inherent characteristics of representative multicore applications.

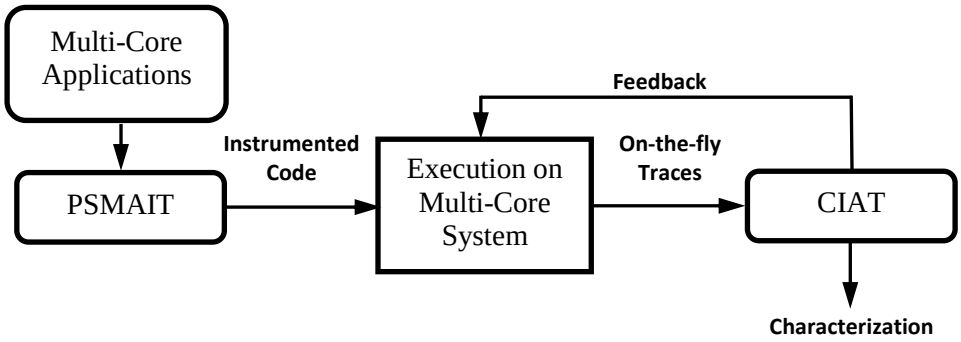


Figure 1. Methodology used to characterize multi-core applications.

3. CHARACTERIZATION METHODOLOGY

This section summarizes the methodology used in this study for monitoring and characterizing multicore applications and describes the tools developed to characterize these applications.

This methodology involves generating detailed memory traces and sending these traces on-the-fly, as the application is being executed, to the analysis tool, as shown in Figure 1. The Pin shared-memory application instrumentation tool (PSMAIT) is used to instrument the multicore application and pipe traces to the ported CIAT that analyzes these traces.

The following sub-sections present more details about the developed tools and the applications being characterized.

3.1 Instrumentation Tool (PSMAIT)

PSMAIT is a tool based on Pin [22], a dynamic binary instrumentation tool for Linux and Windows. Pin is a just in time (JIT)-based dynamic instrumentation tool. It uses dynamic compilation techniques to instrument applications while they are running. Pin instruments single and multithreaded applications and supports Intel IA-32 and x86-64 instruction-set architectures [23]. It has a rich set of API's that can be used to instrument applications without the need to master the underlying instruction set.

PSMAIT is a tool written in C++. It consists of a set of instrumentation and analysis routines as shown in Figure 2. The instrumentation routines determine where instrumentation is inserted and the analysis routines determine what to do when instrumentation is activated. PSMAIT is designed to collect traces of multithreaded parallel applications and to send these traces directly, on-the-fly, to CIAT. PSMAIT is a run-time binary instrumentation tool, which means that it does not need the source code of the parallel application. It instruments both the parallel application's user code and all the user-level libraries that are called during the application execution.

Figure 2 shows the implementation overview of PSMAIT. PSMAIT uses Pin instrumentation routine to capture memory accesses that are performed by user code and user-level libraries only; it does not measure operating system events. Subsequently, it uses `MemRead` or `MemWrite` analysis routine, depending on memory access type, to send a simple trace record to CIAT for every memory access. This trace record contains the type of the access (load or store, integer or floating point), its size and the starting virtual address of the memory location accessed. PSMAIT sends these memory access records to CIAT on-the-fly by using pipes and waits for receiving confirmation feedback from CIAT. On-the-fly analysis enables analyzing large problems fast without needing huge storage medium.

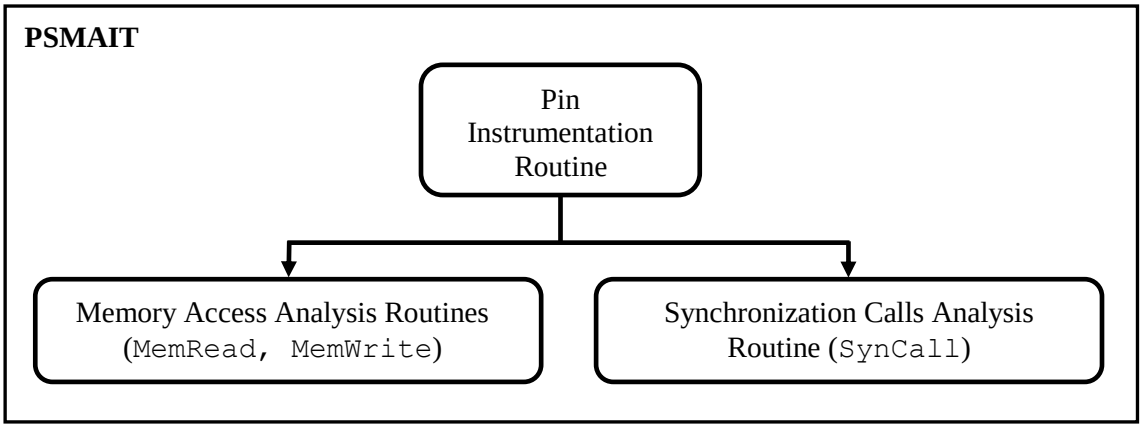


Figure 2. PSMAIT implementation overview.

Additionally, PSMAIT uses Pin instrumentation routine to capture the synchronization calls such as thread spawn and join, mutex lock and unlock, barrier and conditional wait and uses `SynCall` analysis routine to pipe their trace records to CIAT. Through these records and the response feedback from CIAT, the two tools control the parallel application execution and avoid any non-deterministic behavior of the instrumented application due to the instrumentation overhead.

3.2 Analysis Tool (CIAT)

Our analysis tool is ported from CIAT that was developed for the RISC multiprocessor systems by Abandah [15]. CIAT characterizes the inherent application characteristics, such as memory access instructions, communication patterns and sharing behavior of parallel applications that are independent from one multicore configuration to another. A multicore configuration includes the hierarchy of cores, the interconnection topology, the coherence protocol, the cache configuration, as well as the sizes and speeds of the multicore system components. CIAT does not characterize the application characteristics that depend on configuration parameters, such as cache misses and false sharing. However, CIAT's characterization of the communication characteristics gives a basic understanding of applications execution and helps explain the dependent characteristics such as cache misses; e.g. high number of RAW accesses means that load misses are higher than store misses.

CIAT uses many variables to count the various events by tracking the memory load and store operations. It accepts traces from PSMAIT, which generates *n* trace pipes for the *n* executing threads. CIAT supports various execution phases; it assumes that the traces come from a parallel application either in a serial or in a parallel phase. In a serial phase, there is only one thread active, while the other threads are idle. In a parallel phase, more than one thread can be active. CIAT uses the special records of the thread spawn and thread join calls to identify switches between serial and parallel phases. At the end of each phase, CIAT generates statistics and saves them in a report file. At the end of the last phase, CIAT reports the aggregate statistics in the report file.

CIAT assumes that *n* cores in multicore processor can execute *n* instructions at the same time and each instruction takes a fixed time. Therefore, a *pseudo clock* in instruction units is used to keep track of the execution time. However, CIAT currently only sees the memory accesses and advances the clock by one for each thread whenever it receives a memory access record. This is an approximation of the instruction stream. CIAT interleaves the analysis of multiple thread traces on the processors according to the thread spawn and join calls and follows the constraints

of the lock, conditional wait and barrier synchronization calls. More details about CIAT are found in Ref. [24].

3.3 Case Study Applications

We have chosen a set of parallel applications that are representative of multicore applications and are widely used in recent multicore research. This set consists of eight applications from two benchmark suits. The first four of these applications are from SPLASH-2 suite [25], which are Radix, FFT, LU and Cholesky. The second four are from PARSEC suite [26], which are Canneal, Blackscholes, Fluidanimate and Swaptions. These eight particular applications are selected, because they represent a wide range of applications and are often used in multicore research. More details about these selected applications are found in Ref. [24].

To study the impact of the application problem size on the communication behavior, we use two problem sizes of each application: Size I and Size II, where the problem solved in Size I is smaller than that of Size II. Table 1 shows these problem sizes and the abbreviations that are used for naming these applications.

We have conducted many analysis experiments of these studied applications with various numbers of threads for the two problem sizes. To validate our results, we repeated these experiments on two machines that have different types of multicore processors. The first machine has dual-core Core i5 2520M processor (3-MB Cache, 3.20 GHz) and the second has quad-core Core i5 2400 processor (6-MB Cache, 3.40 GHz). The characterization results on the two machines are identical. Therefore, this is one validation check that our characterization tools do not depend on the hardware configuration. Moreover, the number of instructions that have been obtained from running the applications with Pin is similar to that obtained by the developed tools.

Table 1. The applications' problem sizes.

Suite	Application	Abbreviation	Size I	Size II
SPLASH-2	Radix	Radix	256K integers	2M integers
	FFT	FFT	64K points	1M points
	LU	LU	256×256	512×512
	Cholesky	Chole	`tk15.O` file	`tk29.O` file
PARSEC	Canneal	Cann	simsmall	simmedium
	Blackscholes	Black	simsmall	simmedium
	Fluidanimate	Fluid	simsmall	simmedium
	Swaptions	Swap	simsmall	simmedium

4. CHARACTERIZATION RESULTS AND EVALUATION

This section presents the results of the inherent characteristics of the multicore applications that are measured and reported by CIAT. Due to paper length limitations, we present here the results of Size II; interested readers can find the results of Size I in Ref. [24].

4.1 Memory Access Instructions

As mentioned in the previous section, the developed tools capture the user code and user-level libraries operations on the memory and report the number of load and store operations as shown in Table 2.

Table 2 shows the number of memory accesses in billions for the eight studied applications when using one thread. These numbers represent the number of operations on memory and not the memory access instructions, where some memory access instructions may do more than one operation on the memory.

Table 2. The counts and percentages of load and store operations for one thread.

	Radix	**FFT**	**LU**	**Chole**	**Cann**	**Black**	**Fluid**	**Swap**
No. of Loads (in 10^9)	0.285 (66.7%)	0.190 (57.1%)	0.109 (68.2%)	0.381 (77.6%)	3.577 (57.7%)	0.213 (61.1%)	1.146 (81.9%)	2.246 (75.2%)
No. of Stores (in 10^9)	0.143 (33.3%)	0.144 (42.9%)	0.051 (31.8%)	0.110 (22.4%)	2.625 (42.3%)	0.136 (38.9%)	0.253 (18.1%)	0.742 (24.8%)

In all applications, loads are more frequent. The load operations ratio is about twice the store operations in most of the studied applications. Some applications, such as Cholesky, Fluidanimate and Sawptions have even larger percentages of load operations. Cholesky has about four times more of load operations than store operations, because it operates on sparse matrices and needs to find the indices of non-zero elements in these matrices. Fluidanimate has about five times more load operations than store operations. Sawptions has about three times more load operations than store operations. The load and store operations in Canneal are relatively equi-frequent.

Figure 3 shows the percentage of memory accesses for running the eight applications with various numbers of threads. The percentages are normalized to the number of memory accesses when running the respective applications with single thread. Thus, we can notice the parallelization overhead. As obvious, the parallelization overhead is negligible in most of the studied applications. However, there are two of the eight studied applications that have a high percentage of parallelization overhead, which are Cholesky from SPLASH-2 and Fluidanimate from PARSEC. Cholesky has about 50% of memory accesses as overhead when running 16 threads. This overhead is due to Cholesky's work on sparse matrices, which have a larger communication to computation ratio. Fluidanimate has about 33% of memory accesses as overhead when running 16 threads. This overhead is due to Fluidanimate's partitioning of the work among the threads and each thread handles its portion and interacts with other threads to handle the shared data.

Figure 4 shows the percentage of the byte, half-word (2 bytes), word (4 bytes), double-word (8 bytes), float (single-precision floating-point) and double-float (double-precision floating-point) load and store operations when running 16 threads. All the studied applications do not have any quad-word (16 bytes) or extended-float (extended-precision floating-point) memory accesses. All the studied applications are scientific benchmarks, which have a large percentage of floating-point operations except Radix and Canneal, which are integer kernel applications. The percentages of byte and half-word accessed data is insignificant in almost all the studied applications except Radix and FFT that have 25% and 6% half-word load and store operations, respectively. These relatively large percentages are because they have a large portion of integer computation.

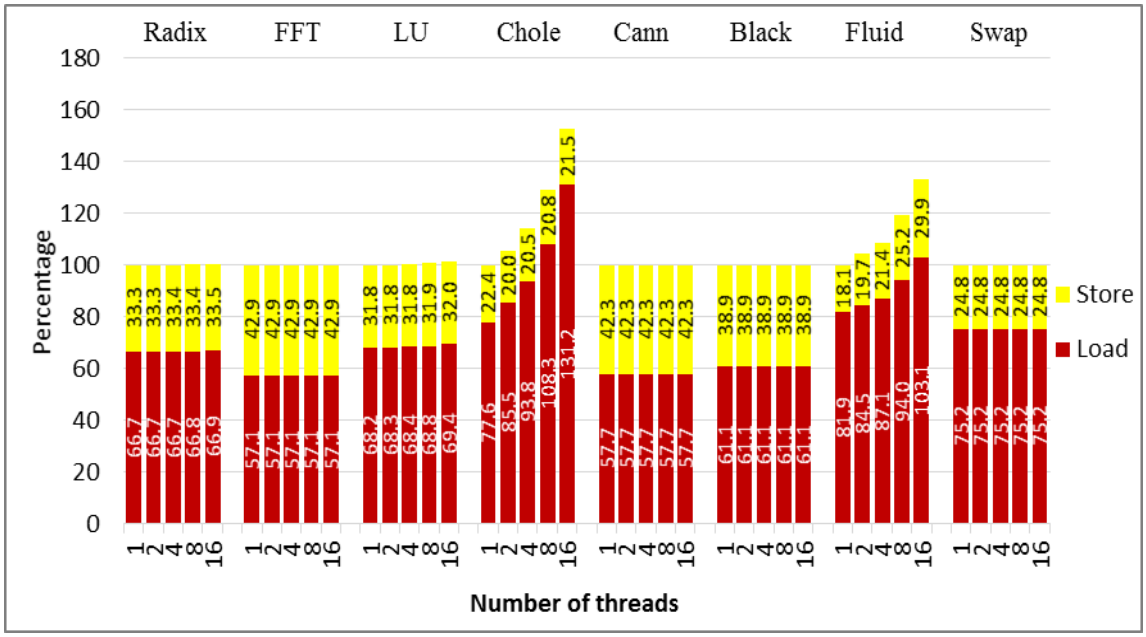


Figure 3. Percentage of memory accesses for 1-16 threads normalized to the memory accesses of one thread.

4.2 Communication Characteristics

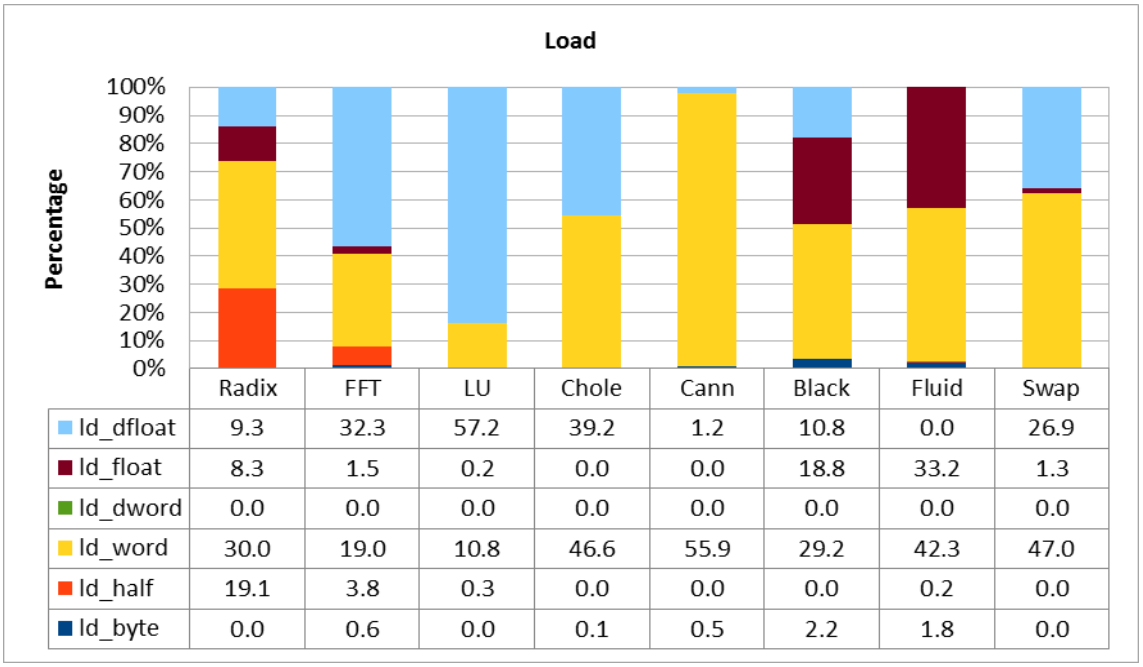
This sub-section presents the inherent communication characteristics that are reported by CIAT.

4.2.1 Communication Patterns

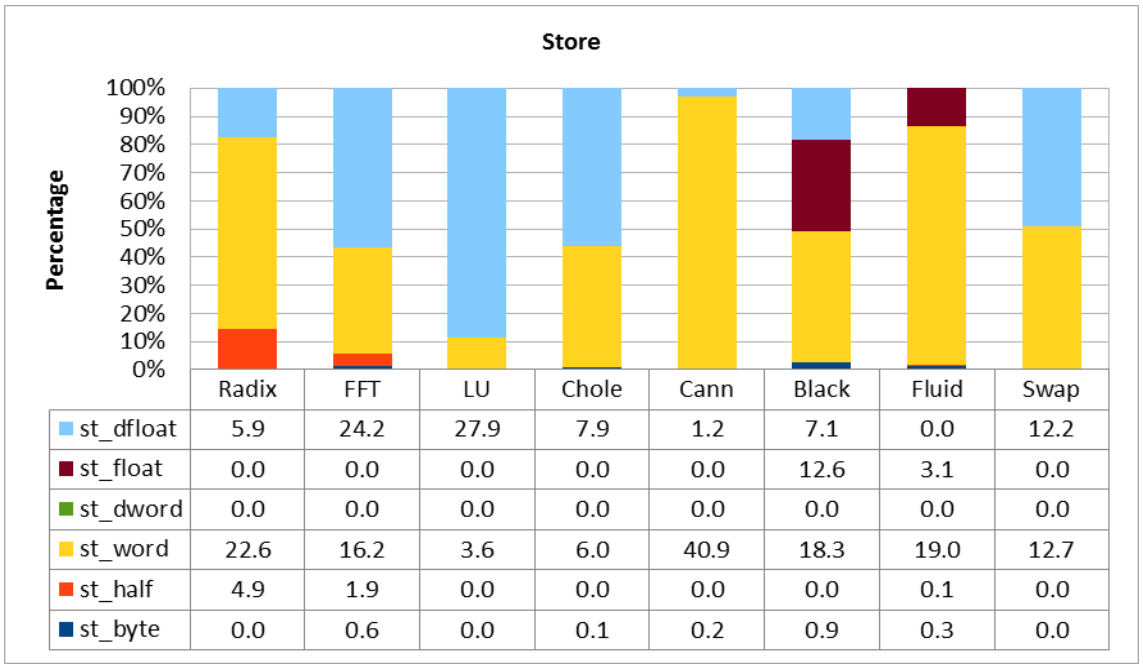
The communication among the cores of a multicore processor occurs when those cores access same-shared memory locations. Characterizing the *communication patterns* is important to know which of the communication patterns are common, thus, facilitating the design of system that supports these patterns efficiently in the current applications and facilitating tuning applications to have less expensive patterns. For each memory location, CIAT keeps track of the type of accesses and the cores that perform these accesses. Consequently, CIAT can report the numbers of the following four types of communication patterns:

- Read after write (RAW) occurs when one core writes to a memory location and other core(s) read from this location. This is the main producer/consumer(s) communication pattern and usually involves copying the written data from the producer's cache.
- Write after read (WAR) occurs when a core writes to a memory location that was read by other core(s). This pattern usually involves invalidating the data copies in the other cores' private caches.
- Write after write (WAW) occurs when a core writes to a memory location that was written by another core. This pattern also involves invalidation and occurs when cores take turns on updating some shared locations.
- Read after read (RAR) occurs when a core reads from a memory location that was read by another core and the first visible access to this location is a read. Here, the data is usually replicated in the cores' caches.

Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 2, No. 1, April 2016.



	Radix	FFT	LU	Chole	Cann	Black	Fluid	Swap
ld_dfloat	9.3	32.3	57.2	39.2	1.2	10.8	0.0	26.9
ld_float	8.3	1.5	0.2	0.0	0.0	18.8	33.2	1.3
ld_dword	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
ld_word	30.0	19.0	10.8	46.6	55.9	29.2	42.3	47.0
ld_half	19.1	3.8	0.3	0.0	0.0	0.0	0.2	0.0
ld_byte	0.0	0.6	0.0	0.1	0.5	2.2	1.8	0.0



	Radix	FFT	LU	Chole	Cann	Black	Fluid	Swap
st_dfloat	5.9	24.2	27.9	7.9	1.2	7.1	0.0	12.2
st_float	0.0	0.0	0.0	0.0	0.0	12.6	3.1	0.0
st_dword	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
st_word	22.6	16.2	3.6	6.0	40.9	18.3	19.0	12.7
st_half	4.9	1.9	0.0	0.0	0.0	0.0	0.1	0.0
st_byte	0.0	0.6	0.0	0.1	0.2	0.9	0.3	0.0

Figure 4. Percentages of the load and store operations according to the type and size of accessed data.

Figure 5 shows the percentages of these four communication patterns of the total number of memory accesses as function of the number of threads used. In the studied applications, PARSEC applications have less communication rates compared to SPLASH-2 (0.5% or lower). The communication rates generally increase as the number of threads increases, except for Blacksholes and Swaptions that have negligible rates.

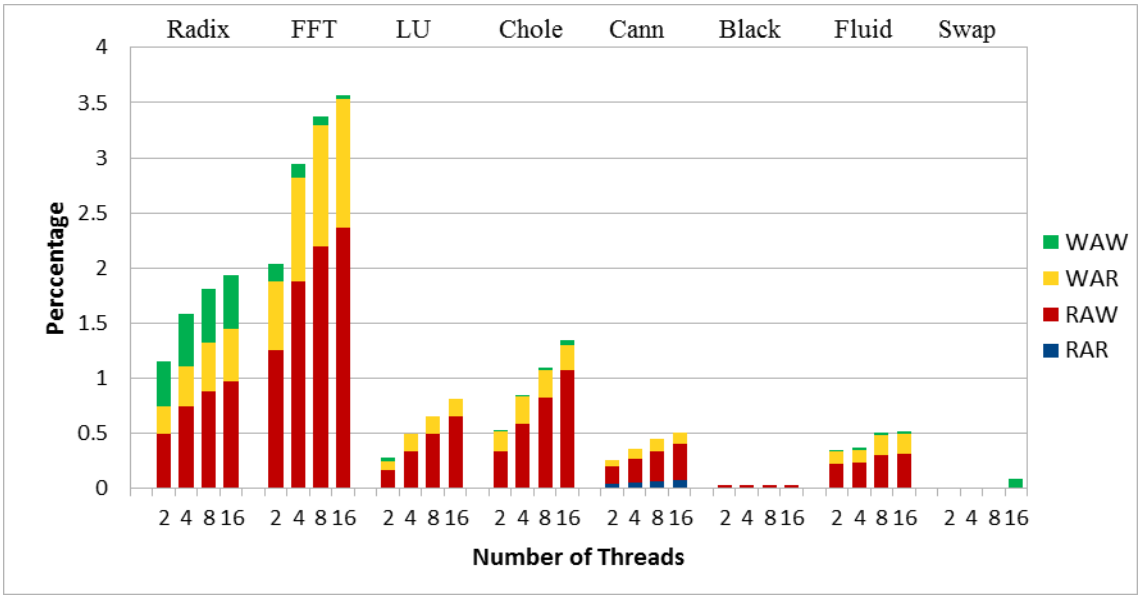


Figure 5. Percentages of the four communication patterns as a function of the number of threads.

Swaptions only has some WAW accesses when using 16 threads due to the reuse of some limited shared locations by these many threads.

Most of the remaining communication accesses are RAW and WAR. FFT has the largest percentage of WAR accesses, which results in large coherence traffic. In addition, almost all the communication accesses of Blacksholes are RAW; the shared memory locations of this application are generally not updated by WAR accesses after the first initialization. The figure shows that only Canneal has a small RAR rate due to reading shared data that is not initialized by the application's user code.

4.2.2 Sharing Degree

The *sharing degree* is the number of threads that read a memory location in the RAW pattern. Figure 6 shows the distributions of sharing degrees for the RAW accesses. It presents the percentages of sharing degrees when using 16 threads. These percentages are calculated by using the following formula:

$$\frac{S[p]}{\sum_{i=1}^{16} S[i]} \times 100\% \quad ; for \quad p = 1,...,16$$

where $S[p]$ is the number of times that p threads read from a memory location after being previously written. Radix, FFT and Blackscholes have small sharing degrees, where almost all the shared locations are shared with only one thread each. Fluidanimate and Swaptions have two sharing degrees. In Fluidanimate, 76% of shared locations are shared with one thread and 23% with two threads. In Swaptions, 78% of shared locations are shared with one thread and 22% with two threads. LU, Cholesky and Canneal have also some sharing degrees higher than two. In LU, 97% of shared locations are shared with four threads and the remaining shared locations are shared with one, two or three threads. In Cholesky, about 58% of shared locations are shared with one thread and 42% are shared with two or more threads. In Canneal, 76% of shared locations are shared with one thread and 24% with two or more threads.

4.2.3 Invalidation Degree

The *invalidation degree* is the number of threads that have read a shared memory location in the WAR pattern.

Figure 7 shows the distributions of invalidation degrees for the WAR accesses. It presents the percentages of invalidation degrees when using 16 threads. These percentages are calculated by using the following formula:

$$\frac{I[p]}{\sum_{i=1}^{16} I[i]} \times 100\% \quad ; for \quad p = 1,...,16$$

where $I[p]$ is the number of times that a memory location was updated after being previously read by p threads. The invalidation degrees in Radix, FFT, LU, Fluid and Swaptions are almost similar to their sharing degree, because the memory locations are iteratively shared and updated by RAW and WAR accesses. Choleskey and Canneal's invalidation degrees drop to one, because the locations that are shared with high degree are not updated by WAR accesses. Blackscholes has some invalidations of degree two, however, its WAR accesses are negligible compared to its RAW accesses.

4.2.4 Communication Slack

The *communication slack* is a measure to know how much time is present between writing a value to a memory location and referencing it by either read or write operation. CIAT measures this time by counting the number of instructions from writing the value until referencing it. The communication slack is distributed into eight ranges from less than ten instructions to more than ten million instructions.

Figure 8 shows the percentages of the communication slack distributions using 16 threads. These percentages are the number of instructions in each range over the total number of memory accesses. In all the studied applications, the communication has most of the slack in the range of tens of thousands of instructions and more. These ranges are enough to make use of prefetching.

4.2.5 Communication Locality

The *communication locality* is a measure of how the cores communicate with each other. Characterizing the communication locality helps both software developers in assigning threads to the cores and hardware designers in selecting a suitable system topology.

CIAT characterizes the communication locality by counting the number of communication events for each thread pair. CIAT maintains a 2D matrix for the communication events, where the rows represent the data producer threads and the columns represent the data consumer threads. For example, the value in Row i and Column j is the number of communication events from Thread T_i to T_j. This value is incremented by one whenever T_j reads from a location after T_i write (RAW), T_j writes to a location after T_i write (WAW) or T_i updates a location after T_j read (WAR).

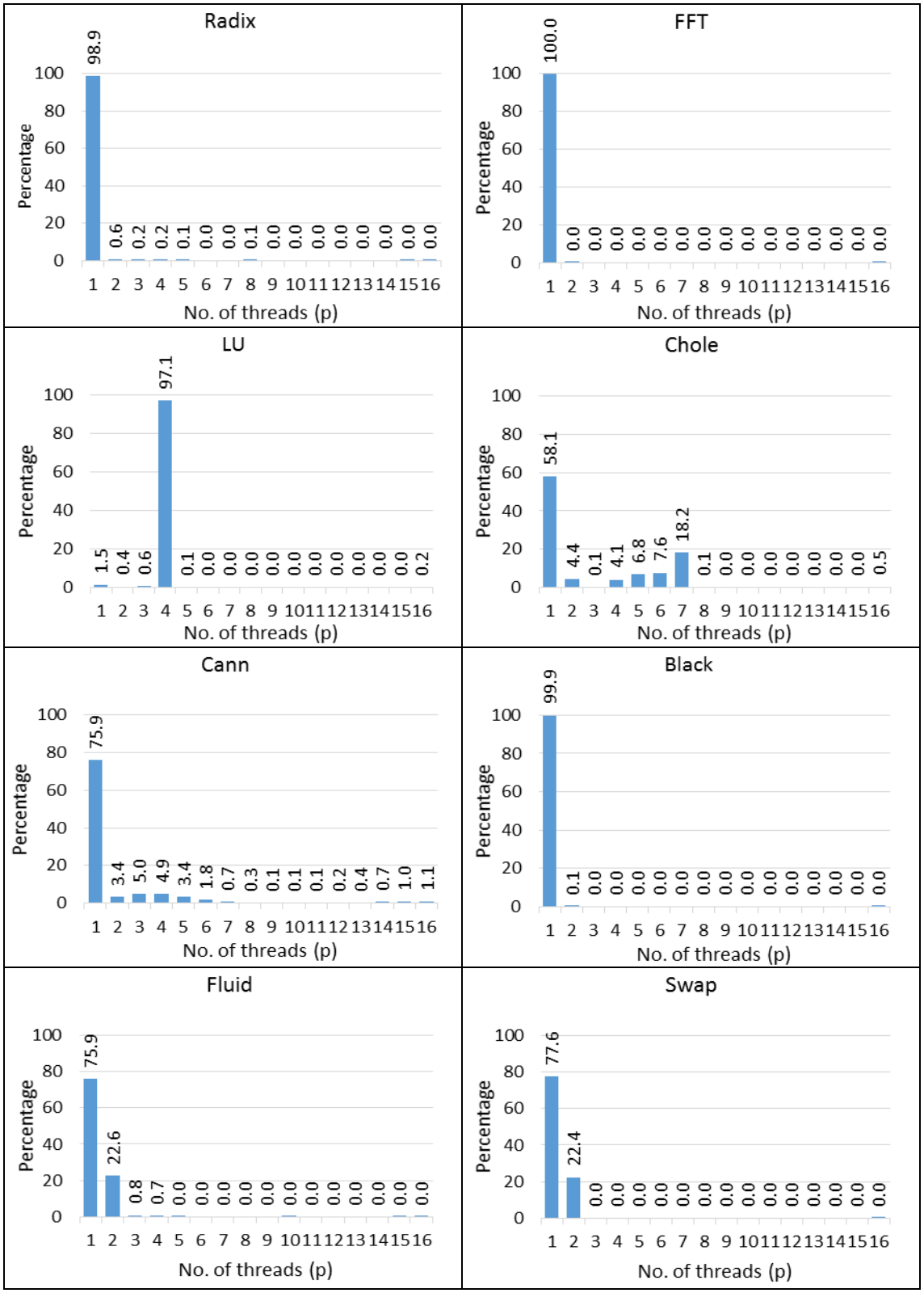


Figure 6. RAW sharing degree for 16 threads.

Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 2, No. 1, April 2016.

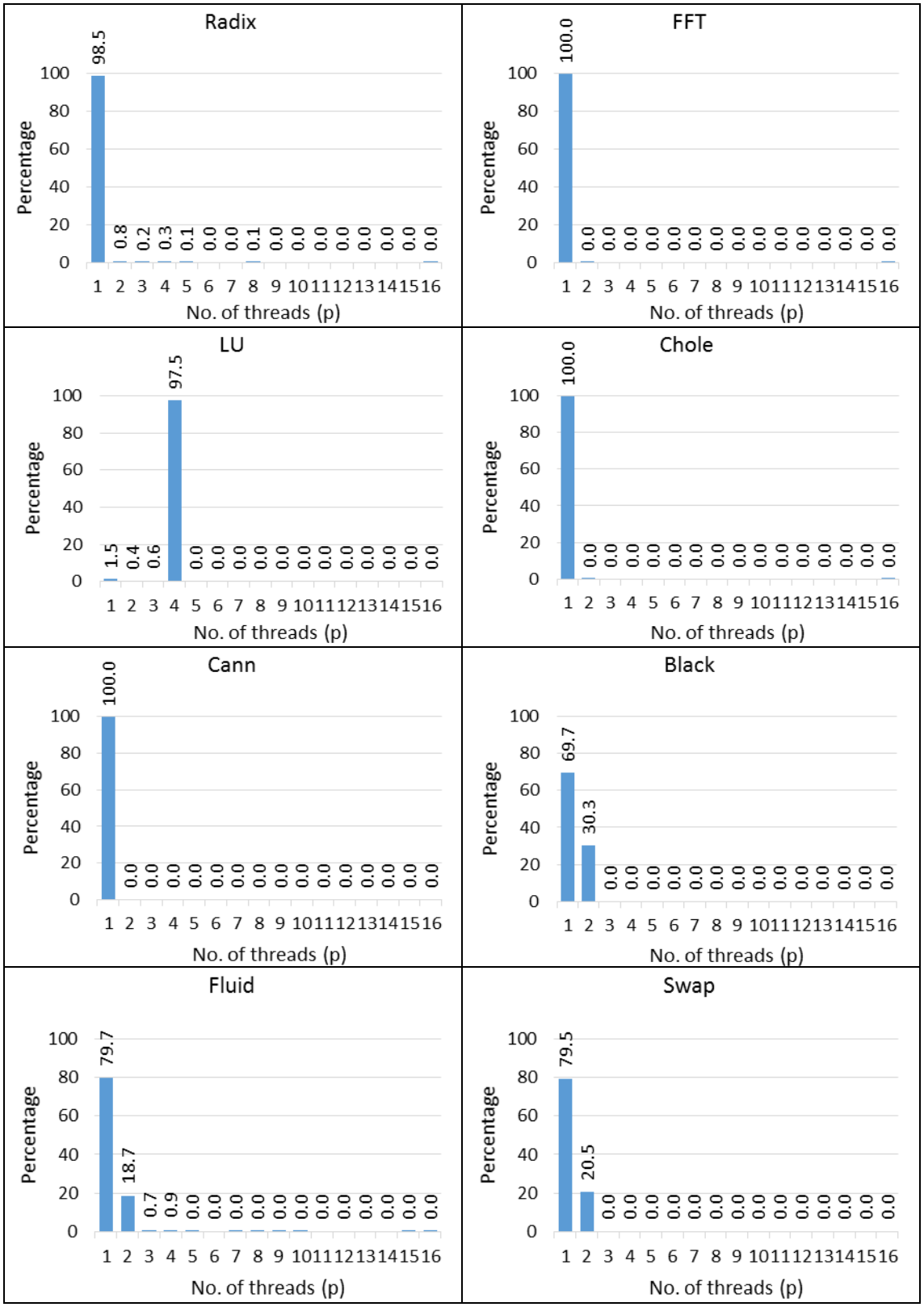
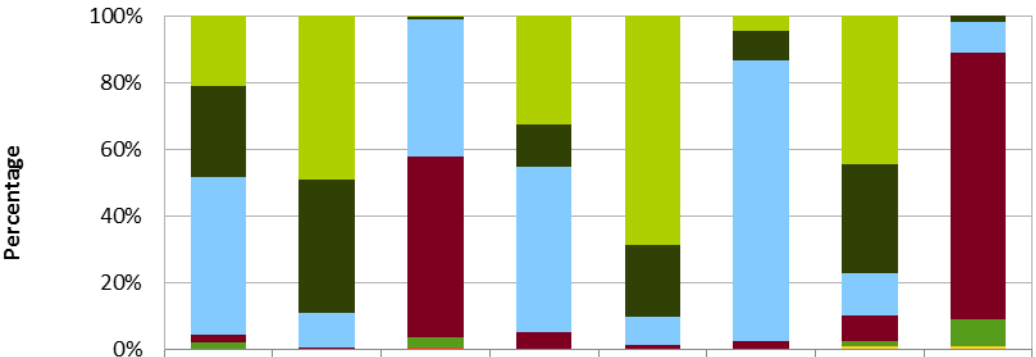


Figure 7. WAR invalidation degree for 16 threads.



	Radix	FFT	LU	Chole	Cann	Black	Fluid	Swap
>=1E+7	20.9	49.3	0.3	32.7	68.9	4.5	44.7	0.1
1E+6-9999999	27.5	39.9	0.7	12.6	21.5	9.0	32.5	1.9
1E+5-999999	47.4	10.4	41.3	49.5	8.5	83.9	12.8	9.1
1E+4-99999	2.1	0.3	54.1	5.1	1.0	2.5	7.5	80.0
1000-9999	2.1	0.0	3.1	0.2	0.1	0.0	1.8	8.1
100-999	0.0	0.0	0.1	0.0	0.0	0.0	0.6	0.6
10-99	0.0	0.0	0.4	0.0	0.0	0.0	0.2	0.1
<10	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.1

Figure 8. Communication slack distributions for 16 threads.

Figure 9 presents the communication locality when using 16 threads. In Radix, each thread communicates with all other threads. Also, there are some additional communications with the neighbors, where some odd threads communicate with only the next thread and some even threads communicate with more than one thread.

FFT has some uniform communication component and has also a large amount of communication from the initial thread to every other thread. In LU, the communication is clustered within groups of $g = n/4$ threads where n is the number of threads that are used to run the application. For example, when running LU using 16 threads, $g = 16/4 = 4$ threads. Additionally, each thread communicates to and from the thread that is located after multiple of g threads from it. For example, if $g = 4$, Thread 1 communicates with Threads 5, 9 and 13. Moreover, the initial thread communicates to all other threads and from the last g threads.

In Cholesky, the communication is non-uniform and each thread communicates with itself; i.e., each thread reads from or writes to memory locations that it previously wrote to and shared with other threads. The initial thread communicates with all other threads. Canneal has some uniform communication component and each thread communicates with itself and the initial thread communicates with all other threads. In Blackscholes, the communication is only with the initial thread; there is no data sharing among the other threads.

In Fluidanimate, the communication is non-uniform and each thread communicates with itself and the initial thread communicates with all other threads. Swaptions has low communication rates and each thread communicates with itself and there is some additional communication due to WAW accesses.

Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 2, No. 1, April 2016.

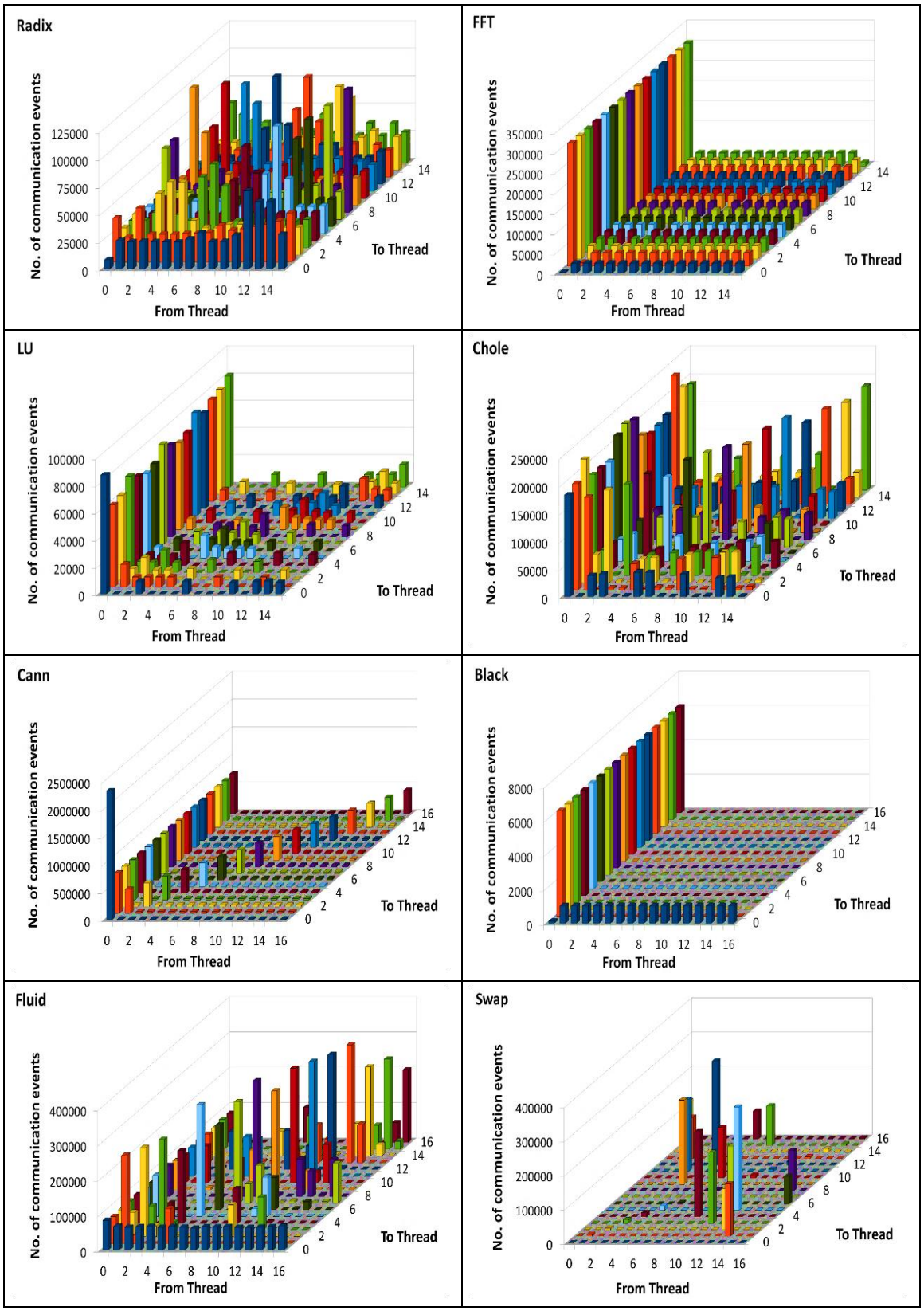


Figure 9. Number of communication events per thread pair for 16 threads.

5. CONCLUSIONS AND FUTURE WORK

Characterizing the inherent characteristics of multicore applications is important to help the programmers in tuning the current applications and developing future parallel applications, as well as to help designers in developing multi-core architectures that efficiently run parallel applications.

In this work, we have used on-the-fly configuration-independent analysis approach to characterize the inherent characteristics of eight multicore applications. Four applications are from SPLASH-2, which are: Radix, FFT, LU and Cholesky, and four from PARSEC, which are: Canneal, Blackscholes, Fluidanimate and Swaptions. The used on-the-fly approach is fast and enables analyzing large problems without needing a huge storage medium.

The obtained results show that the number of memory accesses, in the studied applications, does not change significantly as the number of threads increases. However, some applications such as Cholesky and Fluidanimate show high parallelization overhead, which is about 50% in Cholesky and about 33% in Fluidanimate. This overhead is due to the synchronization operation. Therefore, the speedup of these applications is limited by the increasing parallelization overhead. As expected, the largest percentages of memory accesses in the scientific applications are of floating point accesses.

The most common communication patterns are RAW and WAR except in Radix that has 36% of its communication in the WAW pattern and Swaptions that has 100% of its communication in the WAW pattern when using 16 threads. Therefore, designers must design systems that support these common patterns efficiently. Also, programmers must tune the applications to reduce these patterns. In general, the communication rates increase with more threads and PARSEC applications have rates smaller than SPLASH-2 applications.

Almost all the sharing in Radix, FFT and Blackscholes is with only one thread. In Fluidanimate and Swaptions, thare are about 23% of sharing with two threads. In LU, Cholesky and Canneal, there are 97, 42 and 24% of sharing with two or more threads, respectively. The invalidation degrees in most of the applications are similar to their sharing degrees. There is considerable diversity in the communication locality of the studied applications. Some applications show uniform communication components such as FFT, Canneal and Blackscholes. Others show non-uniform communication and almost in all applications, the initial thread communicates with the other threads. Therefore, it is advisable to assign the initial thread to a central core to reduce the communication cost.

As future work, we plan to extend CIAT to capture the instruction stream in addition to capturing the data stream. Moreover, we need to develop CIAT to handle additional parallelization schemes such as the pipeline parallelization scheme that is used in three PARSEC applications: Dedup, Ferret and X264.

REFERENCES

[1] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson and K. Chang, "The Case for a Single-chip Multiprocessor," ACM Sigplan Notices, vol. 31, no. 9, pp. 2–11, 1996.

[2] D. Geer, "Chip Makers Turn to Multicore Processors," Computer, vol. 38, no. 5, pp. 11–13, 2005.

[3] C. van Berkel, "Multi-core for Mobile Phones," in Design, Automation Test in Europe Conference Exhibition, pp. 1260–1265, 2009.

[4] G. Blake, R. G. Dreslinski and T. Mudge, "A Survey of Multicore Processors," Signal Processing Magazine, IEEE, vol. 26, no. 6, pp. 26–37, 2009.

[5] B. A. Mahafzah, "Performance Assessment of Multithreaded Quicksort Algorithm on Simultaneous Multithreaded Architecture," The Journal of Supercomputing, vol. 66, no. 1, pp. 339–363, 2013.

[6] B. A. Mahafzah, "Parallel Multithreaded IDA* Heuristic Search: Algorithm Design and Performance Evaluation," International Journal of Parallel, Emergent and Distributed Systems, vol. 26, no. 1, pp. 61–82, 2011.

[7] G. A. Abandah and E. S. Davidson, "Origin 2000 Design Enhancements for Communication Intensive Applications," in Proc. of the International Conference Parallel Architectures and Compilation Techniques (PACT'98), pp. 30–39, 1998.

[8] J. Dongarra, S. Moore, P. Mucci, K. Seymour and H. You, "Accurate Cache and TLB Characterization Using Hardware Counters," in Computational Science-ICCS 2004, Springer, pp. 432–439, 2004.

[9] M. Bhadauria, V. M. Weaver and S. A. McKee, "Understanding PARSEC Performance on Contemporary CMPs," in IEEE Int'l Symp.Workload Characterization, pp. 98–107, 2009.

[10] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," ACM SIGARCH Computer Architecture News, vol. 40, no. 1, pp. 37–48, 2012.

[11] Z. Jia, L. Wang, J. Zhan, L. Zhang and C. Luo, "Characterizing Data Analysis Workloads in Data Centers," in IEEE Int'l Symp. Workload Characterization (IISWC), pp. 66–76, 2013.

[12] W. E. Cohen and B. A. Mahafzah, "Statistical Analysis of Message Passing Programs to Guide Computer Design," in Proceedings of the IEEE Thirty-First Hawaii International Conference on System Sciences , vol. 7, pp. 544–553, 1998.

[13] S. R. Alam, R. F. Barrett, J. A. Kuehn, P. C. Roth and J. S. Vetter, "Characterization of Scientific Workloads on Systems with Multi-core Processors," in IEEE International Symposium on Workload Characterization, pp. 225–236, 2006.

[14] L. Chai, Q. Gao and D. K. Panda, "Understanding the Impact of Multicore Architecture in Cluster Computing: A Case Study with Intel Dual-core System," in 7th IEEE Int'l Symp. Cluster Computing and the Grid, 2007, pp. 471–478.

[15] G. A. Abandah, Reducing Communication Cost in Scalable Shared Memory Systems, Ph.D. dissertation, The University of Michigan, 1998.

[16] A. Jaleel, R. S. Cohn, C.-K. Luk and B. Jacob, "CMP$im: A Pin-based on-the-fly Multi-core Cache Simulator," in Proc. 4th Annual Workshop on Modeling, Benchmarking and Simulation, pp. 28–36, 2008.

[17] G. Contreras and M. Martonosi, "Characterizing and Improving the Performance of Intel Threading Building Blocks," Proc. of the IEEE International Symposium on Workload Characterization (IISWC 2008), pp. 57–66, 2008.

[18] A. Bhattacharjee and M. Martonosi, "Characterizing the TLB Behavior of Emerging Parallel Workloads on Chip Multiprocessors," in 18th Int'l Conf. Parallel Architectures and Compilation Techniques, pp. 29–40, 2009.

[19] T. Dey, W. Wang, J. W. Davidson and M. L. Soffa, "Characterizing Multi-threaded Applications Based on Shared-resource Contention," in IEEE Int'l Symp. Performance Analysis of Systems and Software, pp. 76–86, 2011.

[20] R. Natarajan and M. Chaudhuri, "Characterizing Multi-threaded Applications for Designing Sharing-aware Last-level Cache Replacement Policies," in IEEE International Symposium on Workload Characterization, pp. 1–10, 2013.

[21] G. A. Abandah and E. S. Davidson, "Configuration Independent Analysis for Characterizing Shared-memory Applications," in Proc. of the 12th International Parallel Processing Symp. (IPPS), pp. 485–491, 1998.

[22] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," SIGPLAN Not., vol. 40, no. 6, pp. 190–200, 2005.

[23] Intel, "Pin-A Dynamic Binary Instrumentation Tool," https://software.intel.com/en-us/articles/pin-a-dynamic-binaryinstrumentation-tool/, 2015, [Online; accessed 22-March-2015].

[24] M. S. Mohammed, Hardware Configuration-independent Characterization of Multi-core Applications, Master's Thesis, The University of Jordan, Amman, 2015.

[25] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," in ACM SIGARCH Computer Architecture News, vol. 23, no. 2, pp. 24–36, 1995.

[26] C. Bienia, S. Kumar, J. P. Singh and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in Proc. of the 17th Int'l Conf. Parallel Architectures and Compilation Techniques, pp. 72–81, 2008.

ملخص البحث:

اكتسبت معماريات المعالج متعدد النوى انتشاراً متزايداً في السنوات الأخيرة. إلا أنّ العديد من التطبيقات المتاحة لا تستفيد تمام الاستفادة من هذه المعماريات. لذا، فقد طوّر باحثون كُثُر تقنيات عديدة لوصف الخصائص من أجل مساعدة المبرمجين على فهم سلوك هذه التطبيقات على المنصّات متعددة النوى وضبطها للحصول على فاعلية أفضل. تقترح هذه الورقة منحىً لوصف الخصائص، مباشرة، من دون الاعتماد على التكوين؛ لوصف الخصائص الكامنة للتطبيقات متعددة النوى. وهذا المنحى يمتاز بالسرعة؛ لأنه لا يعتمد على تفاصيل أي تكوين آلة بعينه، ولا يتطلب إعادة وصف الخصائص لكل تكوين مستهدف. فهو يتتبّع فقط إمكانيات الوصول إلى الذاكرة والنوى التي تؤدي تلك الإمكانيات من خلال إيصال بيانات تتبع الذاكرة على نحو فوري إلى أداة التحليل.

لقد تم تطبيق هذا المنحى على ثمانية تطبيقات لوصف خصائصها، مأخوذة من مجموعتي المقارنة (SPLASH-2) سبلاش 2 و(PARSEC) بارسيك. تعرض هذه الورقة الخصائص الكامنة لهذه التطبيقات، بما في ذلك تعليمات الوصول إلى الذاكرة، وأنماط خصائص الاتصال، ودرجة التشارُك، ودرجة الإبطال، وتراخي الاتصال، ومحلية الاتصال. وتُظهر النتائج أن إثْنين من التطبيقات المدروسة لهما سقف توازٍ عال، وهما: (Cholesky) تشولسكي و (Fluidanimate) فلويدانيميت. كما تشير النتائج إلى أنّ التطبيقات المدروسة من مجموعة "سبلاش 2" تمتلك معدّلات اتصال أعلى مقارنة بالتطبيقات المدروسة من مجموعة "بارسيك"، وأنّ هذه المعدّلات تزداد بشكل عام كلما ازداد عدد المسارات (Threads) المستخدمة. ويحدث التشارك والإبطال في معظمه بدرجات قليلة. ومع ذلك، كان لإثْنين من تطبيقات مجموعة "سبلاش 2" جزء مهم من الاتصال بدرجات تشارُك عالية باستخدام أربعة مسارات أو أكثر. وكانت لمعظم التطبيقات مركّبة اتصال موحّدة، وكان المسار الابتدائي بشكل عام منخرطاً في قدر أكبر من الاتصال مقارنة بالمسارات الأخرى.