Characterization of Locality in Loop-Parallel Programs

(Student Paper)

Seon Wook Kim Michael Voss Rudolf Eigenmann
School of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907-1285
(765) 494-3557, 0634, 1741
{seon,mjvoss,eigenman}@ecn.purdue.edu

Abstract

Data locality is one of the most important characteristics of programs. Its study has significant influence on the future development of architectures and compilers. Shared-memory multiprocessor (SMP) machines and their applications have become widely available, but there are few studies in the classification of data locality in parallel programs. Most studies have focused on temporal and spatial locality, and false sharing as metrics by which to optimize cache-coherence actions.

In this paper, we propose a classification of data locality in loop-based parallel programs on SMP machines. The classification is expressed in terms of the cacheline sharing, processor sharing, memory reference instruction reuse, and parallel/serial region sharing. This classification helps us understand the characteristics of complex parallel programs. For analysis, we have implemented a tool, called ParaSim which consists of two parts: a code augmenter and a memory simulator. It can augment highly optimized parallel code written using the Sun native parallel directives, OpenMP and in thread-based form, and runs in parallel with supporting parallel execution on SMP machines. As a case study, for this extended abstract, we discuss the data locality of one Perfect Benchmarks, parallelized by the Polaris compiler.

In our measurements we have found that (1) the characteristics of data locality in loop-parallel programs is very similar to that in sequential ones, (2) the cachelines are well localized to processors in highly parallelized code, (3) most variables are reused in the same region type, (4) hits usually occur in the same region, and misses occur across regions, (5) capacity and conflict misses are more important factors than compulsory and coherence misses, (6) spatial locality is the dominant factor, and (7) coherence misses occur more often for temporal locality than spatial locality, and these accesses are by different instructions in the same region.

I. Introduction

One of the fundamental means to characterize an application is to analyze its *data locality*. Data locality primarily helps researchers understand the data movement of applications, and suggests future directions for hardware and software development. Understanding data locality in parallel programs is crucial for improving performance on SMP systems. However, understanding data locality and its implications in parallel programs is difficult, since data movement is frequent and not easily predicted on shared-memory machines.

Woo and *et al.* in [1] characterized the SPLASH-2 programs, which is a suite of parallel applications used for the study of centralized and distributed shared address space multiprocessors. They studied various characteristics such as concurrency and load balance, communication to computation ratio and traffic needs, working set sizes, temporal and spatial locality, and scalability with respect to problem size and number of processors.

Dubois and *et al.* in [2] classified misses based on interprocessor communication into essential misses and useless misses. Essential misses are the smallest set of misses required for correct execution. The other misses can be ignored without affecting the correctness of the execution. Based on this classification, they compared the effectiveness of several protocols which delay and combine invalidations leading to useless misses.

McKinley and Temam in [3] modeled data locality using four categories on loop nests in sequential programs: *spatial*, *temporal*, *self*, and *group* depending on the cacheline sharing and the memory reference instruction reuse. The authors quantified the data locality trends in numerical applications using the SPEC95 and Perfect Benchmarks. They found that although most reuse is within loop nests, most misses are inter-nest capacity misses, and correspond to potential reuse between nearby loop nests. They show that temporal and spatial reuse are important within a loop nest and most reuse across the nests and the entire program is temporal.

Even though data locality has been studied extensively, there are many remaining questions about data locality in parallel programs. Most research in this area has focused on studying temporal and spatial locality to improve cache-coherence actions. In this paper we introduce a new cache characterization methodology that will allow us to address the following questions about data locality in loop-based parallel programs on SMP machines:

1. *Are the data locality characteristics of parallel programs the same as those of sequential programs?* For example, in sequential programs, most reuse occurs within a nest rather than across nests. In a whole program, spatial reuse is the dominant form of reuse [4], but in loop nests, temporal locality is the dominant factor [3]. Is this the same in parallel programs?

2. *How well are variables localized to processors in highly parallelized code?* Can we say that certain variables or certain ranges of arrays are always used in a specific processor? Or can we classify variables into serial and parallel variables?

3. *Are coherence misses a more important factor than the other misses, such as compulsory, capacity, and conflict misses?* Unlike sequential programs, parallel programs have one more miss type, a coherence miss [5]. Do coherence misses occur more often than other misses?

Our classification is an extension of the scheme proposed in [3]. We have extended this scheme to categorize data locality in loop-based parallel programs. We will discuss how each classification helps understand parallel applications. For analysis purposes, we have developed a data locality tool, called ParaSim based on the Polaris preprocessor. As a case study, we will discuss the data locality in the Perfect Benchmark `FLO52`[1].

In Section II, we propose our new classification of data locality in parallel programs, and in Section III, we present our data locality tool, ParaSim. The tool was implemented for collecting the proposed locality information, and it can run parallel programs written using the Sun iMPACT directive language [6], the Polaris/MOERAE thread-based form [7], and the OpenMP API [8]. Section IV gives the case study presenting the analysis of the Perfect Benchmark `FLO52`, and Section V concludes the paper.

II. Classification of Data Locality in Parallel Programs

We extend the classification in [3] for loop-based parallel applications executed on SMP machines. Unlike serial applications, loop-parallel programs consist of sequences of serial and parallel regions. The master processor executes serial regions, whereas parallel regions are executed by all processors. Data is moved across serial/parallel regions and processors. We classify the data locality in parallel programs in terms of cacheline sharing, processor sharing, memory reference instruction reuse, and region sharing:

- cacheline sharing:
  - *temporal*: the word is/was in the cache and the last previous reference accessed the same word in the same cacheline.
  - *spatial*: the word is/was in the cache and the last previous reference accessed another word in the same cacheline.
- processor sharing:
  - *local*: the current and last previous references were made by the same processor.
  - *nonlocal*: the current and last previous references were made by different processors.
- memory reference instruction reuse:
  - *self*: the last previous reference to the word or cacheline came from the same memory reference instruction.
  - *group*: the last previous reference to the word or cacheline came from a different memory reference instruction.
- region sharing:
  - *same*: the last previous reference to the word or cacheline came from the same type of region (serial/parallel).

[1] The full paper will characterize the entire Prefect Benchmark suite.

– *other*: the last previous reference to the word or cacheline came from a different type of region (serial/parallel).

We name each classification as follows: "cacheline sharing"-"processor sharing"-"memory reference instruction reuse"-"region sharing." For example, the category `temporal-local-self-same` means the current and the last previous memory reference instructions accessed the same word in the same cacheline (temporal). These references were issued by the same processor (local) and were generated by the same instruction (self). Since they were generated by the same instruction, it is obvious that they were in the same type of region (same). The symbol * is used as wildcard and is refers to both categories. We further extend each classification by breaking them down into cache *hits* and *misses*.

Our classification is based upon a 4-state (MESI) Write-Back Invalidation Protocol [9], which includes the state transitions (Modified, Exclusive, Shared, Invalid). Variants of this protocol are used in many modern microprocessors, including the Intel Pentium, PowerPC 601, and the MIPS R4400. In a write invalidation protocol, such as MESI, misses can be divided into *compulsory misses*, *capacity/conflict misses*, and *coherence misses* [5]. A compulsory miss occurs at the first reference to a data block by a given processor. A capacity/conflict miss is a replacement miss, and a coherence miss results from the invalidation of the cacheline. A block that has been never accessed before (compulsory miss) is classified by default as `temporal-local-self-same`.

Each memory reference is classified according this scheme, and three distances characterizing the reference is collected: *region*, *instruction*, and *memory reference instruction distances*. The region distance is the number of serial and parallel regions between the last previous and the current references to the same data block. The instruction distance is the number of instructions between these accesses and the memory reference instruction distance is the number of memory references between the previous and the current references to the block.

Using the classification scheme described above, we can make the following observations:

1. `temporal-nonlocal-*-*`: This is true sharing, since more than one processor accesses the same word in the same cacheline. If the current access is a hit, then the sharing was read-only in the other processor. A coherence miss shows communication.

2. `spatial-nonlocal-*-*`: These accesses may be, or may be the cause of, false-sharing. More than one processor accesses different words of the same cacheline. In a hit, writes can cause false-sharing. Coherence misses show that false-sharing has occurred.

3. `temporal-local-*-*`: These accesses are local to a single processor. In a hit, a previously cached word is reused as though it was placed in a buffer. In a miss, only capacity/conflict or compulsory misses can occur.

4. `spatial-local-*-*`: These accesses are also local to a single processor. In a hit, a previously cached line is reused, but a different word is accessed. In a miss, only capacity/conflict or compulsory misses can occur so this does not imply false sharing.

5. `spatial-*-self-*`: The same instruction is accessing different words in the same cache line. If there is a hit, this shows good use of spatial locality. This usually occurs in loops that walk through arrays, accessing contiguous regions.

6. `*-*-self-other`: This category never occurs, since it is impossible that the same instruction appears both in a serial and a parallel region.
7. `*-local-group-same`: In a hit, variables are used like a regional buffer. Data is brought into the cache by one instruction and is later reused by another instruction in the same type of region. If the region distance is zero, this implies that the variables are reused within a loop nest. A nonzero region distance shows good data distribution across regions, since a processor reuses data that was cache while it was executing a different section of code.
8. `*-local-group-other`: This can occur only for the master processor. The master processor executes both in serial and parallel regions. If the number of hits are high in this category, and static scheduling is used, there is a potential load imbalance, since the master processor may experience much fewer cache misses than the other processors.
9. `*-nonlocal-group-same`: If the access is a miss, this suggests poor distribution across regions.
10. `*-*-group-other`: The cacheline is accessed in both types of regions. Coherence misses suggest a region conflict. That is, that the movement of data across region types (from a parallel to serial region or from a serial to parallel region) caused invalidations.

A summary of all classifications are given in Table 1, and Figure 1 shows an example of the data locality classification of a program section on 2 processors.

III. ParaSim: Data Locality Tool

For analyzing programs according to our proposed classification, we developed the tool called ParaSim. Unlike other simulators like RSIM [10], WWT-II [11], PROTEUS [12], and Tango Lite [13] which emulate the parallel machines, our data locality simulator ParaSim actually runs the programs in parallel on SMP machines. The codes can be compiled by various parallelizing compilers such as the Sun parallelizing compiler (AutoPar) [6], and Polaris [14], using the Sun directive sets [14], thread-based codes generated from Polaris with MOERAE backend [7] and the OpenMP API[8]. ParaSim traces and collects real addresses generated on the SMPs.

We augment the parallel code at the assembly level. Unfortunately few tools for such augmentation have been implemented for SPARC multiprocessor machines [15]. Two tools are widely used for augmentation on SPARC machines, not for parallel code but for sequential code: EEL and Shade. The EEL (Executable Editing Library) is a library for building tools to analyze and modify executable programs [16]. It supports an editing model that enables tool builders to modify an executable without awareness of the details of deletion or addition of instructions. The current version of EEL cannot modify linked libraries, and the executable objects should be linked statically, which implies that EEL cannot be used for parallel programs using threads. Shade is an instruction set simulator using dynamic compilation [15]. Shade reads only statically linked executable codes, interprets them, and generates traces controlled by an analyzer interface. It supports flexible and extensive trace generation capabilities. There

Table 1: Classification of Data Locality in Parallel Programs.

Cacheline Sharing	Processor Sharing	Instr. Reuse	Serial/ Parallel	Cache	Characteristics
temporal	local	self	same	hit	Local read-only buffer.
				miss	Capacity/conflict or compulsory only.
			other	hit	Not applicable.
				miss	
		group	same	hit	Local region buffer. Nonzero region distance shows good distribution across regions.
				miss	Capacity/conflict only.
			other	hit	Inter-region reuse by master processor. It may cause load imbalance by favoring master processor in parallel region.
				miss	Capacity/conflict misses on master processor across regions.
	nonlocal	self	same	hit	Read-only sharing.
				miss	Capacity/conflict misses for read-only sharing.
			other	hit	Not applicable.
				miss	
		group	same	hit	Read-only sharing.
				miss	All types of misses but compulsory possible. Nonzero region distance shows poor distribution across regions.
			other	hit	Inter-region/inter-processor sharing.
				miss	Coherence misses show region conflict.
spatial	local	self	same	hit	Exploit spatial locality (read or write).
				miss	Capacity/conflict only.
			other	hit	Not applicable.
				miss	
		group	same	hit	Shows good distribution. It may be a regional buffer.
				miss	Capacity/conflict only.
			other	hit	Inter-region reuse by master processor. It may cause load imbalance by favoring master processor in parallel region.
				miss	Capacity/conflict misses on master processor across regions.
	nonlocal	self	same	hit	Write can cause false sharing.
				miss	Coherence misses are false sharing.
			other	hit	Not applicable.
				miss	
		group	same	hit	Write can cause false sharing.
				miss	Coherence misses are false sharing. Nonzero region distance shows poor distribution across regions.
			other	hit	Inter-region/inter-processor sharing.
				miss	Coherence misses show region conflict.

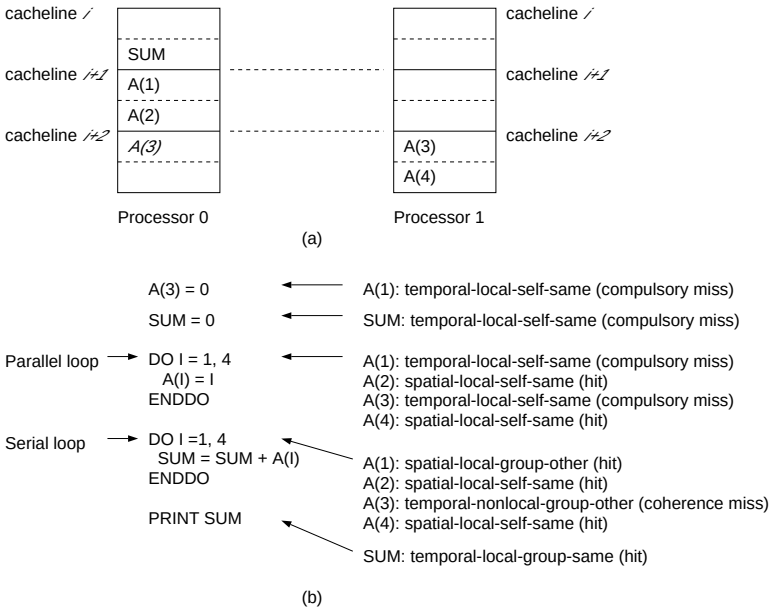

Figure 1: An Example of the Classification of Data Locality. (a) Cachelines. (b) An Example Program and Its Data Locality. The blocks for `A(1)`, `A(3)`, and `SUM` have never been brought into the cache on processor 0 (compulsory miss), so the category is be default `temporal-local-self-same`. The variable `A(2)` is in the same cacheline as `A(1)`, but are different words. The cacheline was accessed by the same processor, the same instructions, and in the same region. Therefore the category is `spatial-local-self-same`. Since `A()` is in category `*-local-self-same`, it is used as a buffer as shown in Table 1. In the first iteration of the serial loop, `A(1)` is accessed by the same processor but now in a different region. The previous and the current instructions are different and access the different words in the same cacheline. Therefore, the category is `spatial-local-group-other`. `A(3)` was invalidated by processor 1, and later read by processor 0. Since this variable is accessed as the same word in the same cacheline by two processors, in different instructions, the category is `temporal-nonlocal-group-other` and is a coherence miss.

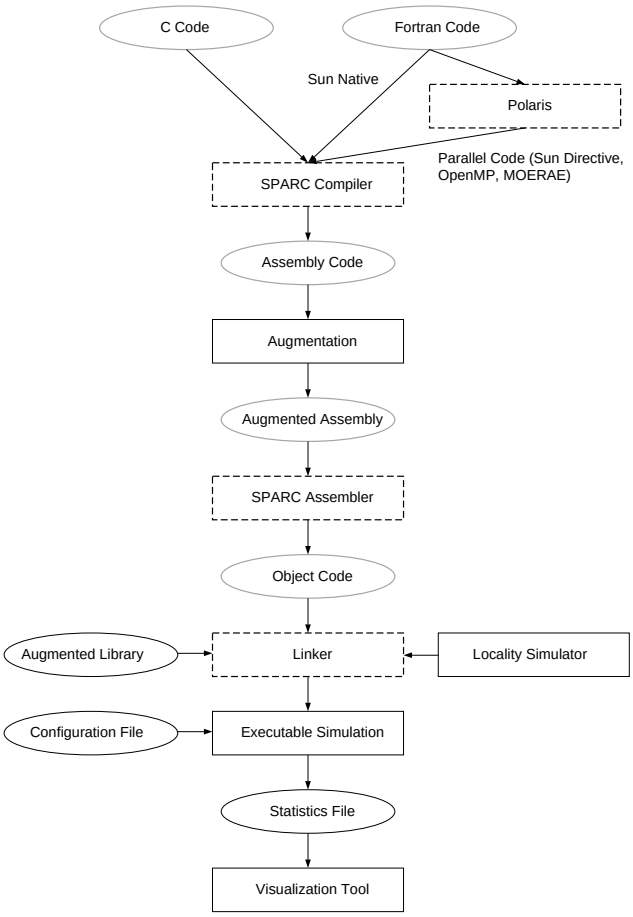

Figure 2: Compilation Process to Build Applications in the ParaSim Simulator.

are several memory reference tracing tools on other machines, and their detailed comparisons are in [15].

Figure 2 shows the process for compiling a program to be analyzed by ParaSim. First the serial and parallel assembly code, generated by the SPARC compiler, is augmented, and the augmented code is linked with the simulator modules and augmented standard libraries. Even though augmentation of assembly codes can instrument only the available assembly code (cannot cover all codes completely) the results are acceptable [13]. ParaSim consists of two major parts: an augmenter and a memory simulator. The augmenter inserts instructions in the application assembly code to pass information to the memory simulator, and the memory simulator analyzes the data locality.

At each call to the memory simulator, the augmented code passes the following information: memory reference, the number of bytes accessed, the number of instructions executed between two consecutive calls to the simulator, a memory reference instruction identification number, and the type of the current code region (serial or parallel). ParaSim can augment applications compiled by various parallelizing compilers. Augmentation of highly-optimized code is difficult because it requires code reordering, hoisting, and compaction. In general, simulators solve this problem by first augmenting the code at a low optimization level, and then further optimize the augmented code [12]. However, our system needs to augment parallel code, which is generated using an optimization level of `-O3` or higher. Because of this, the augmentation module performs advanced program analysis and transformations.

The augmenter employs several techniques such as instruction reordering, control flow analysis, register live analysis, and branch and conditional code hoisting. The instruction reordering relaxes the code compaction generated by the high optimization of the compiler. The control flow and register liveness analysis are done to determine which registers are live at calls to the simulator. The purpose of this analysis is to store and restore the live registers at each call to the memory simulator, since the simulator program may change the contents of the live registers. The new blocks are created by hoisting annulled and not-annulled branches, and integer and floating-point conditional codes are modified. In addition, the mathematical and C library functions are renamed and augmented by the augmentation libraries, and the number of instructions between calls to the simulator are counted. Finally the augmentation code is inserted for storing and restoring live-in and live-out registers and to call the simulator, and the augmented routines are printed out. The detailed description of the implementation is in [17].

IV. Case Study: The Perfect Benchmark FLO52

The data locality of `FLO52` is measured [18, 19] in this extended abstract. We use the following parameters: 16K byte L1 data cache, 32 byte cacheline size, direct map, and LRU replacement scheme. The program is parallelized by the Polaris parallelizing compiler using the MOERAE thread-based postpass [7], and compiled using -O5 optimization by the Sun iMPACT compiler [6]. The run was performed on a Sun Enterprise 4000 using 4 processors [20]. The Sun Enterprise 4000 has 6 UltraSPARC Version 9 processors, and each has a 16K data cache, with 32 byte cachelines, and the system has a shared 1.5 GB main memory.

A. *Basic Statistics*

Table 2 shows the total number of load and store instructions executed and their ratios on each processor in serial and parallel regions. In `FLO52`, about 40% of all instructions are memory reference instructions. Most of the execution time in spent in parallel regions, and the processors are well load-balanced. In parallel regions, the number of load instructions are 2 times larger than those of store instructions. Tables 3 shows the cache hit and miss ratio in serial and parallel regions. It shows that in parallel regions, capacity/conflict misses are the dominant factor over the other misses, which is the same as in sequential programs.

Table 2: Instruction Executed in Serial and Parallel Regions of `FLO52`. The percentages are relative to the total number of instructions executed on each processor.

Benchmark	Processor	Total Instructions		Memory Instructions			
				Serial		Parallel	
		Serial	Parallel	Load	Store	Load	Store
FLO52	0	7.72e+06 (1.72%)	4.41e+08 (98.28%)	1.92e+06 (0.04%)	9.71e+05 (0.22%)	1.28e+08 (28.61%)	5.28e+07 (11.75%)
	1	0 (0%)	4.26e+08 (100.00%)	0 (0%)	0 (0%)	1.23e+08 (28.95%)	5.07e+07 (11.91%)
	2	0 (0%)	4.26e+08 (100.00%)	0 (0%)	0 (0%)	1.23e+08 (28.95%)	5.07e+07 (11.92%)
	3	0 (0%)	4.34e+08 (100.00%)	0 (0%)	0 (0%)	1.25e+08 (28.66%)	5.13e+07 (11.82%)

Table 3: The Cache Hit and Miss Ratio (%) in Serial and Parallel Regions on Each Processor.

Benchmark	Region	Processor	Instruction	Hit	Misses		
					Compulsory	Capacity/ Conflict	Coherence
FLO52	Serial	0	Load	91.64	0.07	8.13	0.15
			Store	90.81	0.29	8.90	0.00
	Parallel	0	Load	93.31	0.01	5.92	0.76
			Store	94.44	0.07	5.49	0.00
		1	Load	92.82	0.01	6.37	0.80
			Store	94.11	0.04	5.85	0.00
		2	Load	93.27	0.00	5.99	0.74
			Store	94.63	0.03	5.34	0.00
		3	Load	93.06	0.00	6.20	0.74
			Store	94.36	0.05	5.59	0.00

B. *Data Locality in FLO52*

Figure 3 shows the measurements of data locality in `FLO52` using our proposed classification. In `FLO52`, the highest category for hits is `*-local-group-same`. This is high for both temporal and spatial locality. As noted in Table 1 these correspond to use of cache lines as local regional

buffers. Figure 4 shows two loops found inside of DFLUX DO30, a parallel loop, and the most time consuming loop in FLO52. The J variable is the index of DFLUX DO30 and so its range is distributed across the processors.

In the DO 28 loop, the load of FS(I,J,N) causes the store to FS(I,J,N) to be a temporal-local-group-same hit. The following iteration will access the adjacent element of FS(I,J,N) yielding a spatial-local-group-same hit. Similarly DIS2(I,J), DIS4(I,J), and DW(I,J,N) will provide spatial-local-group-same hits as the loop iterates. Since processors have disjoint values for J no coherence misses are seen. In the DO 18 loop, the loads of W(I,J,N) and W(I+1,J,N) will generate a spatial-local-group-same hit within the same iteration. There will be further spatial hits across iterations for all three accesses. The access to W(I+1,J,N) can also cause W(I,J,N) to have a temporal-local-group-same hit from the previous iteration. There are many such loops found throughout FLO52 and these explain the large amount of *-local-group-same hits.

Figure 5 shows the breakdown of the hits in the major loops in FLO52 into temporal-local-group-same and spatial-local-group-same. Three of the four loops show that spatial locality is dominant as is reflected by the total numbers in Figure 3. However, in PSMOO DO40 temporal locality dominates. The characteristics of this loop nest are clearly seen in Figure 6. In each inner loop, the same array element is accessed as many as three times in each of the Fortran statements. In the assembly code these accesses will be separate instructions and so will fall into the spatial-local-group-same category.

The dominant form of miss in Figure 3 is capacity, showing that working set size exceeds the cache size. Over 90% of all misses are capacity misses. Approximately 55% of all misses are spatial-local-group-same capacity misses and these have large region distances. The fact that these misses are spatial may be misleading. Figure 7 explains this phenomenon.

About 20% of misses are spatial-nonlocal-group-same capacity misses. Misses that fall in the spatial-nonlocal-group-same category can signal poor data distribution if the region distances are nonzero. This is because a different word in the same cache line was last accessed on a different processor. In the case of coherence misses this is false sharing. FLO52 shows very few misses due to false sharing (approximately 1%). The reason that so many misses fall in the spatial-nonlocal-group-same grouping can explained by looking at Figure 8, which shows EFLUX DO30, the second most time consuming loop in FLO52.

The values of JL and IL vary during the program execution from 4 to 32 and 20 to 160 respectively. The J loop is distributed across the processors. This means, that as the program progresses, different ranges of the arrays in EFLUX DO30 are accessed by each processor. This redistribution of the arrays is shown by the high number in the spatial-nonlocal-group-same category.

Finally, about 8% of misses are due to coherence with a large majority of these falling into the temporal-nonlocal-group-same category. As Table 1 states, this again is an indication of distribution problems when the region distance is large. As with the spatial-nonlocal-group-same capacity misses, EFLUX DO30 is a good example of where such distribution problems occur.

Table 4 summarizes the data locality in FLO52. From the Tables, we conclude the following:

- In parallel regions, the behavior of all processors is very similar.
- The characteristics of the benchmark is similar to those of sequential programs.

cacheline sharing	processor	instruction	serial/parallel	hit	miss		
					compulsory	capacity/conflict	coherence
temporal	local	self	same	(0.58, 0.54, 0.30, 0.30) (0.69, 0.65, 0.09, 0.09) (0.63, 0.59, 0.05, 0.05) (0.63, 0.59, 0.05, 0.05)	(0.00, 0.00, 1.86, 1.26) (0.00, 0.00, 206.04, 204.97) (0.00, 0.00, 179.14, 178.06) (0.00, 0.00, 180.61, 179.51)	N/A	N/A
			other	N/A	N/A	N/A	N/A
		group	same	(18.47, 17.37, 0.03, 0.03) (18.00, 16.90, 0.03, 0.03) (18.17, 17.06, 0.03, 0.03) (18.18, 17.09, 0.03, 0.03)	N/A	N/A	N/A
			other	(0.03, 0.03, 2.03, 1.76) N/A N/A N/A	N/A	N/A	N/A
	nonlocal	self	same	(0.13, 0.12, 0.11, 0.11) (0.13, 0.12, 0.12, 0.12) (0.13, 0.12, 0.11, 0.11) (0.13, 0.12, 0.11, 0.11)	N/A	N/A	N/A
			other	N/A	N/A	N/A	N/A
		group	same	(0.12, 0.12, 3.95, 3.85) (0.10, 0.10, 6.31, 6.20) (0.11, 0.10, 5.34, 5.31) (0.12, 0.11, 5.97, 5.82)	N/A	N/A	(3.32, 0.20, 3.08, 3.20) (2.50, 0.15, 2.78, 2.75) (3.20, 0.20, 2.96, 2.97) (2.78, 0.17, 3.18, 3.18)
			other	(0.01, 0.00, 1.96, 3.16) (0.00, 0.00, 6.76, 5.67) (0.00, 0.00, 25.51, 24.42) (0.00, 0.00, 2.99, 1.86)	N/A	N/A	(0.00, 0.00, 37.97, 39.07) (0.05, 0.00, 3.15, 1.78) (0.05, 0.00, 2.34, 1.00) (0.04, 0.00, 2.49, 1.33)
spatial	local	self	same	(5.09, 4.79, 0.08, 0.08) (5.14, 4.82, 0.07, 0.07) (4.86, 4.57, 0.04, 0.04) (4.87, 4.58, 0.05, 0.05)	N/A	(0.04, 0.00, 0.00, 0.00) (0.01, 0.00, 0.00, 0.00) (0.05, 0.00, 0.00, 0.00) (0.04, 0.00, 0.00, 0.00)	N/A
			other	N/A	N/A	N/A	N/A
		group	same	(75.39, 70.92, 0.05, 0.04) (75.85, 71.22, 0.04, 0.04) (75.99, 71.32, 0.04, 0.04) (75.97, 71.40, 0.05, 0.05)	N/A	(87.04, 5.16, 1.29, 1.29) (89.84, 5.49, 1.42, 1.43) (89.16, 5.48, 1.40, 1.41) (89.61, 5.39, 1.39, 1.40)	N/A
			other	(0.09, 0.08, 2.99, 2.80) N/A N/A N/A	N/A	(2.47, 0.15, 5.71, 5.81) N/A N/A N/A	N/A
	nonlocal	self	same	(0.01, 0.01, 1.36, 1.36) (0.01, 0.01, 5.28, 5.28) (0.02, 0.01, 2.60, 2.60) (0.02, 0.01, 3.69, 3.69)	N/A	(0.03, 0.00, 0.00, 0.00) (0.04, 0.00, 0.00, 0.00) (0.04, 0.00, 0.07, 0.07) (0.00, 0.00, 1.03, 1.03)	N/A
			other	N/A	N/A	N/A	N/A
		group	same	(0.09, 0.08, 1.33, 1.27) (0.08, 0.08, 1.71, 1.64) (0.09, 0.08, 1.51, 1.46) (0.09, 0.08, 2.05, 1.99)	N/A	(6.40, 0.38, 3.23, 3.27) (7.05, 0.43, 4.05, 4.08) (6.99, 0.43, 4.54, 4.55) (6.09, 0.42, 4.27, 4.27)	(0.43, 0.03, 2.57, 2.65) (0.39, 0.02, 3.93, 3.90) (0.39, 0.02, 4.23, 4.22) (0.52, 0.03, 3.06, 3.05)
			other	(0.01, 0.01, 3.58, 4.95) (0.00, 0.00, 4.15, 2.94) (0.00, 0.00, 3.88, 2.73) (0.00, 0.00, 2.28, 1.21)	N/A	(0.26, 0.02, 9.55, 10.84) (0.06, 0.00, 28.09, 26.85) (0.06, 0.00, 40.43, 39.34) (0.05, 0.00, 21.34, 20.16)	(0.02, 0.00, 18.52, 19.86) (0.05, 0.00, 4.16, 3.14) (0.05, 0.00, 2.54, 1.52) (0.04, 0.00, 1.57, 0.56)

Figure 3: Data Locality Characteristics of `FLO52`. In each category the number of references is given for each processor in the form (X, Y, A, B), where X is the ratio of the category to the total hit or miss memory references, Y is the ratio of the category to the total memory references, and A and B are the serial and parallel region distances respectively.

```
      DO 28 I=1,IL
      FS(I,J,N) = DIS2(I,J)*FS(I,J,N)-DIS4(I,J)*DW(I,J,N)
   28 CONTINUE

  ...

      DO 18 I=1,IL
      FS(I,J,N) = W(I+1,J,N)-W(I,J,N)
   18 CONTINUE
```

Figure 4: Inner Loops of `DFLUX DO30`. These loops show a high number of `*-local-group-same` hits.

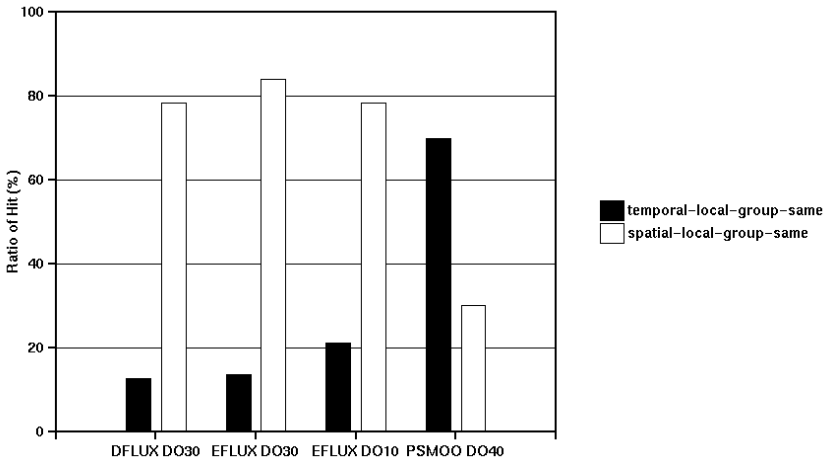

Figure 5: The Breakdown of Hits in the Major Loops in `FLO52`.

```
      DO 40 N=1,4
      DO 10 J=2,JL
      DW(1,J,N) = 0.
   10 CONTINUE
      DO 20 I=2,IL
      DO 20 J=2,JL
      DW(I,J,N) = DW(I,J,N)-R*(DW(I,J,N)-DW(I-1,J,N))
   20 CONTINUE
      I= IL
      DO 30 J=2,JL
      DW(I,J,N) = T*DW(I,J,N)
   30 CONTINUE
      DO 40 II=3,IL
      I = I-1
      DO 40 J=2,JL
      DW(I,J,N) = DW(I,J,N)-R*(DW(I,J,N)-DW(I+1,J,N))
   40 CONTINUE
```

Figure 6: The `PSMOO DO40` Loop Nest in `FLO52`.

```
PARALLEL DO = 1,N
    A(I) = ...
ENDDO

...

PARALLEL DO = 1,N
    ... = A(I)
ENDDO
```

Figure 7: Two Parallel Loops. The first loop accesses `A(1)`, `A(2)`, ... and then second loop accesses `A(1)`, `A(2)`, ... If the `A` is evicted due to capacity between the two loops, `spatial-local-group-same` capacity misses will result, since `A(2)` was accessed last in the first loop and the miss is on `A(1)` in the second loop.

```
   DO 30 J=2,JL
   DO 30 I=2,IL
   XX        = X(I,J,1)-X(I-1,J,1)
   YX        = X(I,J,2)-X(I-1,J,2)
   PA        = P(I,J+1)+P(I,J)
   QSP       = (XX*W(I,J+1,3)-YX*W(I,J+1,2))/W(I,J+1,1)
   QSM       = (XX*W(I,J,3)-YX*W(I,J,2))/W(I,J,1)
   FS(I,J,1) = QSP*W(I,J+1,1)+QSM*W(I,J,1)
   FS(I,J,2) = QSP*W(I,J+1,2)+QSM*W(I,J,2)-YX*PA
   FS(I,J,3) = QSP*W(I,J+1,3)+QSM*W(I,J,3)+XX*PA
   FS(I,J,4) = QSP*(W(I,J+1,4)+P(I,J+1))+QSM*(W(I,J,4)+P(I,J))
30 CONTINUE
```

Figure 8: `EFLUX DO30`. The second most time consuming loop in `FLO52`.

- Cacheline sharing: Spatial locality is more important than temporal locality. Temporal locality is the main factor in the occurrence of coherence misses, and spatial locality is the main factor in the occurrence of capacity/conflict misses.
- Processor sharing: Cachelines are highly privatized to processors. The local category suffers from capacity/conflict miss, but the nonlocal category results in both capacity/conflict and coherence misses. Interestingly most evicted cachelines are later accessed by the same processor.
- Instruction reuse: There are few misses in the self category, and most cachelines are used by different memory references.
- Region sharing: Most variables are always used either in a serial region or in a parallel region, not in both. There is little data movement between regions.

C. Locality Distances

Figure 5 shows the distance of instructions, memory reference instructions, serial and parallel regions. It shows that misses have larger distances than hits, and hits generally occur inside the same loop. Most misses occur across regions. This result corresponds to that on single processor machines [3].

V. Conclusion

Data locality is an important characteristic of programs, and its analysis helps determine future directions in the development of architectures and compilers. Even if shared-memory multiprocessor machines and their applications have become widely available, there are few studies in the classification of data locality in parallel programs. Most studies have focused on temporal and spatial locality, and false sharing to optimize cache-coherence actions.

In this extended abstract, we have proposed a new classification of the data locality in parallel programs on shared-memory multiprocessors. This classification is based on the cache-line sharing (temporal and spatial), processor sharing (local and nonlocal), memory reference instruction reuse (self and group), and region sharing (same and other). Also each category is classified into hit, miss, cache-coherence state transitions, and we further divide misses into compulsory, capacity/conflict, and coherence misses. For the analysis of our proposed classification, we developed the tool called ParaSim. Unlike other existing tools, ParaSim can run various versions of parallel programs compiled by several parallelizing compilers. Using the proposed classification scheme, we have discussed the data locality of the Perfect Benchmark `FLO52`.

Our results show that (1) The characteristics of the data locality in `FLO52` is very similar to that in sequential programs. (2) The cachelines are well localized to processors in highly parallelized code. (3) Most variables are reused in the same type of region, not across the different regions, (4) A hit occurs in the same region, and a miss occurs across the regions, (5) Capacity and conflict misses are more important than compulsory and coherence misses in highly parallelized code. (6) Like on single processor systems, spatial data locality is also the

Table 4: The Summary of the Locality (%) in `FLO52`.

Processor	Locality		Hit	Misses		
				Compulsory	Capacity/ Conflict	Coherence
0	cacheline sharing	temporal	18.37	0.03	0.59	0.44
		spatial	75.23	0.00	5.25	0.08
	processor sharing	local	92.57	0.03	4.13	0.00
		nonlocal	1.04	0.00	1.71	0.52
	instruction reuse	self	6.02	0.03	0.25	0.00
		group	87.59	0.00	5.59	0.53
	region sharing	same	93.48	0.03	5.73	0.53
		other	0.12	0.00	0.11	0.00
1	cacheline sharing	temporal	17.90	0.02	0.58	0.47
		spatial	75.30	0.00	5.64	0.09
	processor sharing	local	92.13	0.02	4.35	0.00
		nonlocal	1.07	0.00	1.87	0.56
	instruction reuse	self	5.73	0.02	0.25	0.00
		group	87.46	0.00	5.97	0.57
	region sharing	same	93.19	0.02	6.21	0.56
		other	0.00	0.00	0.01	0.01
2	cacheline sharing	temporal	17.94	0.01	0.49	0.43
		spatial	75.73	0.00	5.31	0.08
	processor sharing	local	92.59	0.01	4.06	0.00
		nonlocal	1.07	0.00	1.74	0.52
	instruction reuse	self	5.70	0.01	0.26	0.00
		group	87.97	0.00	5.54	0.52
	region sharing	same	93.67	0.01	5.79	0.52
		other	0.00	0.00	0.01	0.00
3	cacheline sharing	temporal	17.79	0.02	0.52	0.42
		spatial	75.65	0.00	5.50	0.09
	processor sharing	local	92.38	0.02	4.30	0.00
		nonlocal	1.06	0.00	1.73	0.52
	instruction reuse	self	6.03	0.02	0.26	0.00
		group	87.41	0.00	5.77	0.52
	region sharing	same	93.43	0.02	6.02	0.52
		other	0.00	0.00	0.01	0.01

Table 5: The Distance of Instructions, Memory References, Serial and Parallel Regions on Each Processor.

Processor	Distance	FLO52		
		Total	Hit	Miss
0	Instructions	440868.80	136.64	6891008.79
	Memory References	406075.87	124.95	6347190.33
	Serial Regions	12.52	0.05	194.90
	Parallel Regions	12.52	0.05	194.92
1	Instructions	422110.33	136.16	6201290.73
	Memory References	388862.73	124.24	5712861.08
	Serial Regions	24.67	0.05	361.82
	Parallel Regions	24.67	0.05	361.82
2	Instructions	319719.12	135.96	5048614.66
	Memory References	294536.81	124.21	4650981.65
	Serial Regions	1.16	0.05	17.56
	Parallel Regions	1.16	0.05	17.56
3	Instructions	283347.47	163.25	4314397.80
	Memory References	261068.02	147.53	3975200.32
	Serial Regions	16.53	0.05	251.13
	Parallel Regions	16.53	0.05	251.13

dominant factor on shared-memory multiprocessor systems. (7) Coherence misses occur more for temporal locality than spatial locality, and these access are by different instructions in the same region.

VI. References

[1] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. *Proceedings of the 22nd Annual Symposium on Computer Architecture (ISCA'95)*, pages 24–36, June 1995.

[2] Michel Dubois, Jonas Skeppstedt, Livio Ricciulli, Krishnan Ramamurthy, and Per Stenström. The detection and elimination of useless misses in multiprocessors. *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA'93)*, pages 88–97, 1993.

[3] Kathryn S. McKinley and Olivier Temam. A quantitative analysis of loop nest locality. *The 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'96)*, October 1996.

[4] A. J. Smith. Line (block) size choice for cpu caches. *IEEE Transactions on Computers*, 36(9):1063–1075, September 1987.

[5] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantative Approach.* Morgan Kaufmann Publishers, Inc., San Francisco, CA, 2nd edition, 1996.

[6] Sun Microsystems Inc., Mountain View, CA. *Fortran Programmer's Guide*, 1996. SC23-0431-0.

[7] Seon Wook Kim, Michael Voss, and Rudolf Eigenmann. Performance analysis of parallel compiler backends on shared-memory multiprocessors. Technical Report ECE-HPCLab-99202, HPCLAB, Purdue University, School of Electrical and Computer Engineering, 1999.

[8] OpenMP Forum, http://www.openmp.org/. *OpenMP: A Proposed Industry Standard API for Shared Memory Programming*, October 1997.

[9] David Culler, Jaswinder Pal Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, 1998.

[10] Vijay S. Pai, Parthasarathy Ranganathan, and Sarita V. Adve. RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. *Proceedings of the Third Workshop on Computer Architecture Education*, February 1997.

[11] S. Mukherjee, S. Reinhardt, B. Falsafi, M. Litzkow, M. Hill S. Huss-Lederman, J. Larus, and D. Wood. Wisconsin Wind Tunnel II: A fast and portable parallel architecture simulator. *Workshop on Performance Analysis and Its Impact on Design (PAID)*, June 1997.

[12] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, MIT, September 1991.

[13] Stephen Alan Herrod. *Tango Lite: A Multiprocessor Simulation Environment, Introduction and User's Guide.* Computer Systems Laboratory, Stanford University, Stanford, CA 94305, November 1993.

[14] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. Parallel programming with Polaris. *IEEE Computer*, pages 78–82, December 1996.

[15] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. *Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 128–137, May 1994.

[16] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*, June 1995.

[17] Seon Wook Kim and Rudolf Eigenmann. ParaSim: The parallel program simulator on shared-memory multiprocessors. Technical Report ECE-HPCLab-99203, HPCLAB, Purdue University, School of Electrical and Computer Engineering, 1999.

[18] M. Berry, D. Chen, P. Koss, D. Kuck, L. Pointer, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[19] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer Performance Evaluation and the Perfect Benchmarks^{TM}. *Proceedings of the 4th International Conference on Supercomputing (ICS'90), Amsterdam, Netherlands*, pages 254–266, March 1990.

[20] Sun Microsystems Inc., Mountain View, CA, http://www.sun.com/servers/enterprise/e4000/index.html. *Sun Enterprise 4000.*