

Quantifying Locality In The Memory Access Patterns of HPC Applications

Jonathan Weinberg¹
jonw@sdsc.edu

Michael O. McCracken¹
mike@cs.ucsd.edu

Allan Snaveley¹
allans@sdsc.edu

Erich Strohmaier²
estrohmaier@lbl.gov

¹San Diego Supercomputer Center
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0505

²Future Technology Group
Lawrence Berkeley National Laboratory
One Cyclotron Road, CA 94720

Abstract

Several benchmarks for measuring the memory performance of HPC systems along dimensions of spatial and temporal memory locality have recently been proposed. However, little is understood about the relationships of these benchmarks to real applications and to each other. We propose a methodology for producing architecture-neutral characterizations of the spatial and temporal locality exhibited by the memory access patterns of applications. We demonstrate that the results track intuitive notions of locality on several synthetic and application benchmarks. We employ the methodology to analyze the memory performance components of the HPC Challenge Benchmarks, the Apex-MAP benchmark, and their relationships to each other and other benchmarks and applications. We show that this analysis can be used to both increase understanding of the benchmarks and enhance their usefulness by mapping them, along with applications, to a 2-D space along axes of spatial and temporal locality.

©2005 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the [U.S.] Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. SC-05 November 12-18, 2005, Seattle, Washington, USA (c) 2005 ACM 1-59593-061-2/05/0011\$5.00

1 Introduction

Machine performance in the arena of supercomputing has traditionally been tracked by The Top500 List [2], a ranking of the world's fastest supercomputers based on the peak floating point operations per second that each can achieve on the LINPACK benchmark [17]. This ranking system has dominated performance comparisons of supercomputers for over a decade and today serves as a unique historical record of the technological evolution of supercomputing.

Despite its longevity, the Top500 list has garnered an increasingly vocal and growing set of critics who justifiably protest that the LINPACK benchmark only stresses machines in a very limited and unrealistic manner, that is, by measuring the peak floating point rate on a compute bound code. Since 1993 when the Top500 project was established, the exponentially widening gap between processor and memory speeds has only exacerbated the problem. Application runtimes today are increasingly dominated by memory operations, a phenomenon that the LINPACK benchmark has failed to capture.

To provide a more complete understanding of comparative machine performance, particularly memory subsystem performance, several new benchmarks have been proposed including components of the HPC Challenge (HPCC) Benchmarks [1] and Apex-Map [30]. A guiding principle in the design of these benchmarks is that the memory subsystem should be stressed along two dimensions, representing various

degrees of spatial and temporal memory locality. The HPCC benchmark suite addresses this requirement with multiple memory benchmarks, each of which presumably represents extreme combinations of spatial and temporal locality in its memory access patterns. Apex-MAP is a synthetic probe that directly tackles the locality concept by performing data movement operations in accordance with parameterized degrees of spatial and temporal locality.

These benchmarking efforts represent important progress towards the establishment of performance probes that more accurately mimic the behavior of today’s applications. Ironically, little is known concretely of the relationships between these probes and the applications they are intended to help us understand. While some components of the HPCC benchmark suite are intended to exhibit “low” spatial locality and another “high”, we cannot quantify “how much” each exhibits or how that quantity compares to a real application of interest. While Apex-MAP can hypothetically be parameterized to imitate the spatial and temporal locality of any application, no satisfactory method exists for easily determining what those parameters might be for a given code. The ability to measure the degree of spatial and temporal locality exhibited by an application or a benchmark, and to relate that to these benchmarks, could go a long way towards understanding these relationships and making the benchmarks more practically useful.

In addition to enabling benchmark evaluation, measurable metrics of spatial and temporal locality can have much to offer the greater understanding of application performance, particularly with respect to the deep memory hierarchies of today’s machines [4, 3, 7, 25]. Such analysis can lend insight into the memory requirements of given applications as input parameters or processor counts scale, be used as part of a performance model to predict memory subsystem performance, or even expose load imbalance issues with respect to workload difficulty on parallel codes.

Section 2 of this paper proposes quantifications of spatial and temporal locality scores whereby we start with classic definitions and arrive at concrete metrics that can be measured of real parallel applications.

Section 3 describes how we actually measure the locality of applications using the Metasim tracer and describes the performance of these techniques.

Section 4 displays results from our locality analysis performed on several application-based and synthetic benchmarks; included are spatial and tempo-

ral scores for memory benchmarks from HPCC and then a “recipe” for finding the mapping between an arbitrary HPC program and Apex-MAP input parameters.

Section 5 explores additional uses for locality measurement of HPC applications.

Section 6 reviews and summarizes the area’s rich literature on which we have built. Finally, we discuss our conclusions and ongoing work.

2 Quantifying Locality

To analyze applications for memory locality, we desire simple, measurable, and architecture-independent metrics for quantifying both spatial and temporal locality. Locality is well defined as a notion in the computer architecture literature. Bunt recently summarized findings from his 20- year career [12], during which he and colleagues have defined locality metrics and also proposed ways to measure them. We start from these definitions and are guided by much previous work further described in the related work section. Nevertheless, it will become apparent that some concrete and arbitrary choices have to be made when implementing the formal definitions about how to count locality statistics, and some approximations have to be made to make the data acquisition process tractable for HPC applications. We work through those details.

In our study of parallel applications, we are focused on the memory reference patterns of individual processors to their local memory (addresses sent through the load-store units). It is true that spatial and temporal locality exist in messages and inter-processor communication and some components of the HPCC test these. We do not treat that here, although a straightforward extension can be considered for future work.

After gathering detailed statistics about spatial and temporal locality, there is a temptation to reduce this information into a single score per-loop or even per application so that one can make broad comparisons such as “application A has more temporal locality than B”. This reduction can be useful but may potentially oversimplify things and throw information away. We propose single-number spatial and temporal locality scores and show their uses and limitations in what follows.

Philosophically, we think it is important to suppress the urge to customize these metrics to proportionately track some observable phenomena such

as application performance or theoretical cache hit rates. This sort of mapping between locality and observable phenomena on a particular machine can be performed more effectively at a later stage; the metrics need only track intuitive notions of locality in an internally consistent fashion. We explore the implication below.

2.1 Spatial Locality

Spatial locality is the tendency of applications to access memory addresses near other recently accessed addresses [11, 19]. We use a definition similar to Bunt’s [11] to define a spatial locality metric based on the average length of non-zero strides performed by an application. The first practical, choice we have to make stems from the interleaved nature of address streams under dynamic execution. For example, a loop could have a stride 1 reference to an index array followed by a random access (indirect) reference to another array using the index; if we only look at two consecutive dynamic addresses for spatial locality, we may miss the fact that the stride 1 address stream (every-other address) has spatial locality. We therefore define the stride of a memory access to be the minimum absolute distance, in 64-bit words, of that memory reference to its nearest neighbor among the \mathbf{W} previously accessed addresses. The “look back window” that contains these previous \mathbf{W} addresses should be sized large enough to capture the memory behavior of most reasonably sized application loops but small enough that we can search it with each new reference in a reasonable amount of time. In our study, we have chosen to set $\mathbf{W} = 32$. This choice is admittedly arbitrary and small but improves the performance of our data collection methods described in section 3.

Once we have defined stride, we can formulate the following simple summation to represent a single-number spatial locality score where $stride_i$ denotes the fraction of total dynamic memory operations that are of stride length i :

$$\sum_{i=1}^{\infty} stride_i/i \quad (1)$$

The idea is simply to generate a normalized score in the range [0,1] that is inversely proportional to the average stride length. An application that performs only stride 1 references receives a score of 1, an application that performs only stride 2 references receives a score of .5, an application whose memory

references are evenly split among stride 1 and 2 receives a score of .75, and so forth. An application without any strided references receives a score of 0.

Notice that the summation does not include stride 0 references. By this definition, spatial locality is the tendency of an application to reference memory addresses near *other* recently referenced addresses, not the same ones. We consider stride 0’s to be the simplest case of temporal locality, not a degenerate case of spatial. This is another concrete but arbitrary choice.

Even though there is technically no termination for the series, it should converge at some i where all subsequent terms in the series are zero for real applications. In practice, the summation can actually be terminated within any number of terms if we accept that the value calculated is within \mathbf{U}/i of the actual score, where \mathbf{U} is the fraction of memory operations that are unstrided. The choice of termination point is therefore dictated by the degree of accuracy desired. However, there is little point in choosing very large values with this scoring definition because the maximum size of each successive element decreases by harmonic series. Further, in our experience, few real codes regularly exhibit much longer strides. For the studies presented in this paper, we have chosen to terminate the summation at $i=8$, an admittedly arbitrary value with whose error bound we are comfortable.

2.2 Temporal Locality

Temporal locality is the tendency of an application to reference the same memory addresses that it referenced recently [19]. Many memory locality studies, further detailed in the related work section, have focused on this type of locality by analyzing statistics related to reuse distance. The reuse distance of some reference to memory address A is the number of unique memory addresses that have been accessed since the last access to A .

We begin our analysis by collecting information about the distribution of reuse distances in an application run. We graph this distribution in what we call the application’s temporal reuse function. The reuse function plots reuse distances against the percentage of an application’s dynamic memory operations with reuse distances less than or equal to that distance. Section 4 contains several examples of such graphs, such as Figure 2.

Producing a single score from this data is less straightforward than is the case for spatial locality.

While in the spatial case the arrangement of memory suggests intuitive meanings and consequently a natural weighting for “nearest-neighbor” and “next-nearest neighbor”, no such natural meaning attaches to reuse distance bins. One approach is to choose a point on the graph that corresponds to a particular size of cache and formulate the temporal score as the fraction of total memory accesses with reuse distance less than that size. However, we would then be bound to a specific architecture and forced to always qualify an application’s temporal score with a cache size. This may be of some limited use for predicting cache hit rates [22] but does not adequately summarize the whole distribution. Further, such a metric is difficult to apply to cache-less, vector-based machines.

An application’s reuse function contains information that is inevitably lost if that function is summarized into a single score. This is certainly also true of stride information with spatial locality scoring, but in that case, we can utilize a natural weighting to summarize the information intuitively. In the absence of such a natural weighting, we have generally chosen to maintain the entire distribution. However, in the interest of a single normalized metric for comparing reuse functions, we formulate a metric similar to that of spatial locality where the cumulative fraction of memory operations at each reuse distance is scaled by some increasing function of that reuse distance and summed. For illustrative purposes, one can visualize this as the area under the temporal reuse function, plotted on some scale, and normalized by total area on the graph. The fundamental idea here is that since the reuse function is monotonically increasing, each memory access increases the application’s total score in a manner inversely proportional to its reuse distance. The result is a metric that recognizes more temporal locality as an application contains more memory references with lower reuse distances. Again the range of these scores is [0,1] with 0 indicating no temporal locality and 1 indicating that all memory references are within the shortest reuse distance measured.

An important parameter is the weighting to assign to memory references at each reuse distance. As discussed earlier, a best weighting is not immediately obvious. There are many intriguing choices for this scale, including one that scales references at each reuse distance by its respective expected or typical cache latency. Our goal however, need not be to create metrics that proportionately track some observable phenomenon like performance or cache hits, but ones that simply track our intuitive perceptions

of locality. The mapping onto observable phenomena can always be performed at a later stage. For simplicity this initial study employs a log scale where each memory reference is weighed by the log of its reuse distance with respect to the largest distance considered. A simple mechanism for visualizing this is the space under the reuse curve, where the reuse distance axis is plotted on a log scale. Again, this choice is subjective and we have provided full temporal analyses should the reader choose to investigate an alternative scale.

The other variable we must consider before practically measuring or approximating such a metric is the size of the largest reuse distance to include. This choice will affect the scoring because it dictates the normalizing value, i.e. the total area on the graph. We must choose to integrate from 0 to some reuse distance N , where there is no compelling value at which to standardize N . This consideration is of only limited concern when we compare applications on the same graph, but it does preclude our using the metric as an unqualified application attribute if no standard exists.

We consequently formulate the following summation, where $reuse_i$ denotes the fraction of dynamic memory operations with reuse distance less than or equal to i .

$$\frac{\sum_{i=0}^{\log(N)-1} ((reuse_{2^{i+1}} - reuse_{2^i}) * \log_2(N) - i)}{\log_2(N)} \quad (2)$$

One factor to remember is that this metric does not *necessarily* track memory subsystem performance. It is indeed possible for two applications to yield identical temporal scores but perform very dissimilarly on a particular machine. For example, on a cache-based machine, the application whose reuse function has a higher value at the reuse distance corresponding to the machine’s cache size would have a performance edge. To predict performance on a known cache-based architecture, a better approach might be to examine the value of each application’s reuse curve at the point corresponding to that architecture’s cache size [22].

3 Measuring Locality

We gather memory access characteristics using the Metasim tracer [27], a tool for dynamic analysis of a program’s memory references, developed as part

of a framework for memory-centric performance prediction. The tracer collects statistics about memory strides and simulates cache behavior as the program runs.

In Metasim, instrumentation is added around each memory reference using a binary rewriting tool such as Atom [29] or Dyninst [10]. The instrumentation is automated and requires only a few seconds for smaller codes. For larger codes, we create multiple binaries with instrumentation on different phases of the program so that runtimes more readily fit in a batch queueing system.

As the instrumented binary runs, the tracer compares each address in a basic block to a window of previous addresses to capture stride information. The stride of an address is defined as in section 2.1. The size of the look-back window, \mathbf{W} , is configurable.

Metasim produces a detailed report of the collective stride information in each basic block. We use this count to generate the spatial locality score as discussed in section 2.1 for each basic block in the program. The score for the whole program is a sum of each block’s score, weighed by the percentage of dynamic memory references the block had generated.

In addition to the stride calculation described above, Metasim runs each address through a set of cache simulators as each is encountered. For performance prediction, this is used to generate cache statistics for each target architecture. We exploit this feature to collect reuse distance distributions by simulating a series of variously sized *temporal caches*, caches with a 1-word block size. The percentage of memory operations with reuse distance less than or equal to \mathbf{N} is the hit rate in an \mathbf{N} -line temporal cache. One should note that our implementation is somewhat inexact because Metasim simulates caches with a random replacement policy instead of LRU (Least Recently Used). This may create some small noise in the data, i.e. it is possible that an address is evicted before \mathbf{N} memory references to other addresses have transpired. Practically however, we observed the hit rates of these caches are not very different on real address streams whether LRU or random replacement policy is used while random is faster and uses less memory in our simulations.

The slowdown yielded by the Metasim tracer has been shown to be within two orders of magnitude of the original code [26]. We benefit from existing work done to improve tracing speed in Metasim via sampling methods that reduce the overhead of memory instrumentation [13]. Current research on Metasim performance enhancement is continually

lowering tracing time and today, traces are usually within a single order of magnitude. The Performance Modeling and Characterization (PMAc) group regularly employs Metasim to trace large-scale, highly parallel, scientific codes [14].

4 Results

To evaluate the methodology, we have applied it to the analysis of relevant serial benchmarks from the HPCC and Apex-MAP. We used an Atom version of Metasim to instrument all binaries and performed the traces on *Lemieux*, an Alpha SC45 machine at the Pittsburgh Supercomputer Center. To obtain the following results, we configured Metasim to use a look back window of size $\mathbf{W}=32$ and a maximum detectable stride length of $\mathbf{S}=8$. For our temporal analysis, we measure reuse distances from \mathbf{N} in the range 16 to 131072 8- byte words by doubling, distances that correspond to temporal cache sizes of 128Bytes to 1MB. When we present an application’s temporal score, we refer to the integral of its logscaled locality curve from 0 to 131072, normalized as a percentage of total area as given by Formula 2. We performed no custom tuning on any of the benchmarks.

4.1 HPCC

To analyze the spatial and temporal locality exercised by the HPCC benchmarks, we examine the following four benchmarks from the suite:

STREAM - a simple synthetic benchmark program that measures sustainable memory bandwidth and the corresponding computation rate for a vector kernel

RandomAccess (GUPS) - measures the rate of integer random updates of memory

FFT - measures the floating point rate of execution of double precision complex one-dimensional Discrete Fourier Transform.

HPL - the Linpack TPP benchmark that measures the floating point rate for solving a linear system of equations.

Figure 1 shows the locality scores of the four applications and Figure 2 displays their temporal reuse functions. To place these scores within some context, we have performed a similar analysis on CG and MG, two of the NAS Parallel Benchmarks [8].

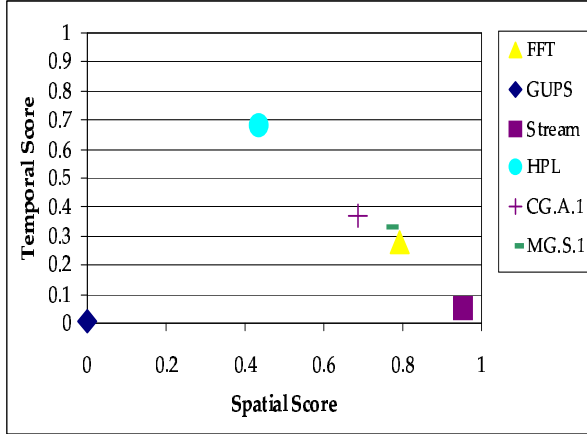


Figure 1: Locality scores of the HPCC and NPB benchmarks

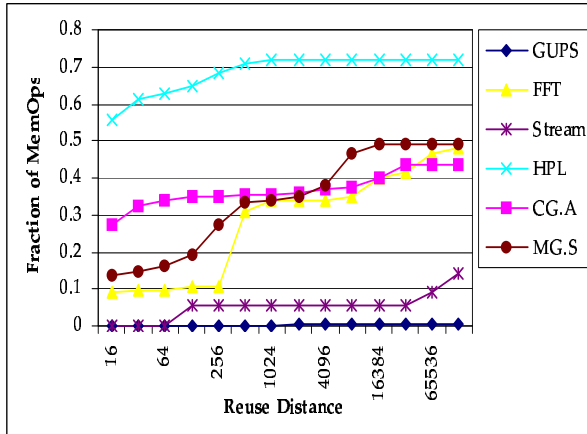


Figure 2: Temporal locality functions of HPCC and NPB benchmarks

These benchmarks were run at problem class A and S respectively.

Notice that STREAM and GUPS are scored intuitively. GUPS performs random updates to a large memory and therefore displays neither spatial nor temporal locality. STREAM on the other hand, performs only regularly strided memory access to a large memory and therefore displays a great degree of spatial locality but very little temporal. STREAM’s access pattern exemplifies the motivation for differentiating locality along these dimensions. If we were to predict a cache hit rate for STREAM based on reuse distance alone [22], then we would under-predict, having not accounted for cache hits induced by prefetching.

HPL and FFT are both meant to represent compute-bound codes and are therefore expected to exhibit high degrees of locality. The results bear this out with HPL exhibiting especially high levels of temporal locality while FFT does similarly with spatial locality.

The end result is that these metrics allow us to compare benchmarks via two single-number scores and make comparisons such as “STREAM has more spatial locality than CG” or “FFT has lower temporal locality than HPL” in a straightforward and meaningful way.

4.2 Apex-MAP

Apex-MAP [30] is a synthetic benchmark that stresses a machine’s memory subsystem according to parameterized degrees of spatial and temporal locality. Along with other parameters, the user specifies L and K , parameters related to spatial locality and temporal reuse respectively. Apex-MAP then chooses a configurable number of indices into a data array that are distributed according to K , using a non-uniform random number generator. The indices are most dispersed when $K=1$ (uniform random distribution) and become increasingly crowded as K approaches 0. Apex-MAP then performs L stride 1 references starting from each index. This process is repeated a configurable number of times.

Parameter sweeps of Apex-MAP have been used to map the locality space of certain systems with respect to L and K . Figures 3 and 4 show how performance in cycles per memory operation varies as a function of L and K on two very different architectures.¹

There has been some effort to determine the L and K values of certain applications using back fitting

¹The parameter K is sometimes referred to as **alpha**.

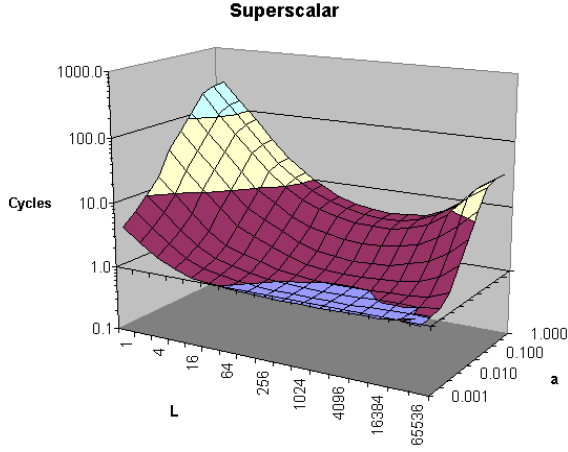


Figure 3: Apex-MAP performance surface on a superscalar machine

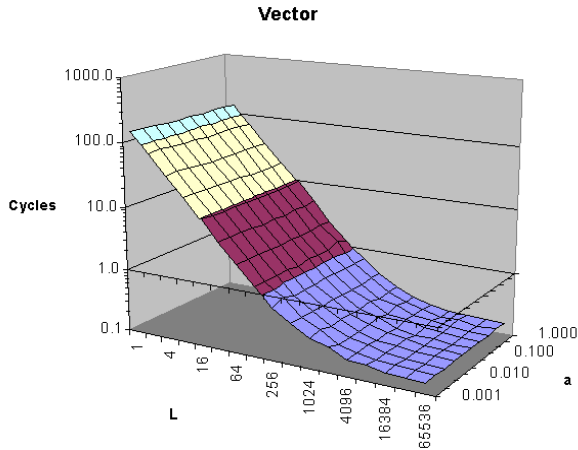


Figure 4: Apex-MAP performance surface on a vector machine

[30]. Theoretically, if we could more easily obtain the L and K value of some application, we could use Apex-MAP as a lightweight probe to mimic the memory access behavior of that application and give some indication of its possible memory performance on a target architecture.

4.2.1 Locality Scoring of Apex-MAP

Our first results aim to determine the extent to which the locality parameters of Apex-MAP actually affect the benchmark’s memory behavior. To determine this, we performed a parameter sweep of Apex-MAP, tracing all combinations of K and L formed from the sets $\{.001, .01, .05, .1, .5, 1\}$ and $\{1, 2, 4, 8, 16, 32, 64, 128, 512, 1024\}$ respectively.

Figure 5 plots the temporal reuse functions of all combinations of K while L is held constant at 1 (i.e. no runs of stride 1). The results clearly indicate that our metrics track the temporal locality of Apex-MAP as a monotonic function of K . The smaller K is, the faster a larger fraction of memory operations falls into smaller temporal caches due to small reuse distances.

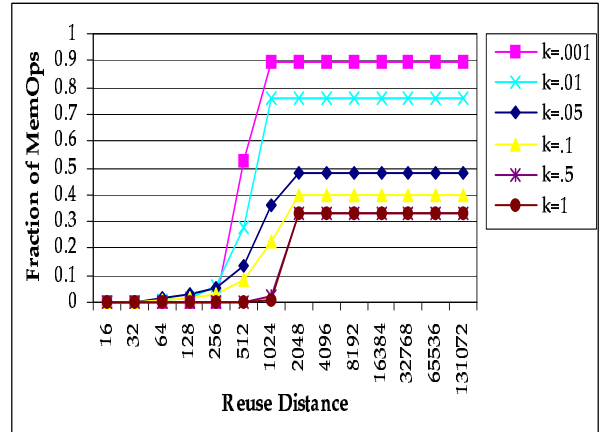


Figure 5: As K increases, the temporal locality of Apex-MAP decreases

Figure 6 plots the mapping functions between K and our temporal scores for each value of L . The reason that different values of L yield separate functions is that as L increases, it introduces more strided, non-temporal, references which in turn lower the percentage of total memory operations that the temporal hits constitute at each point on the curve. This phenomenon is most clear in the case of $K=1$ where temporal scores are in proportion to $1/L$. As the value of K increases however, the relationship between the

curves is blurred, which could indicate some interdependence between \mathbf{L} and \mathbf{K} with respect to temporal reuse.

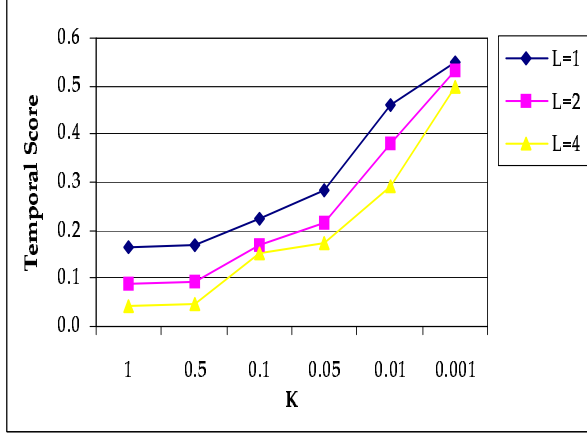


Figure 6: Mapping of \mathbf{K} to temporal score

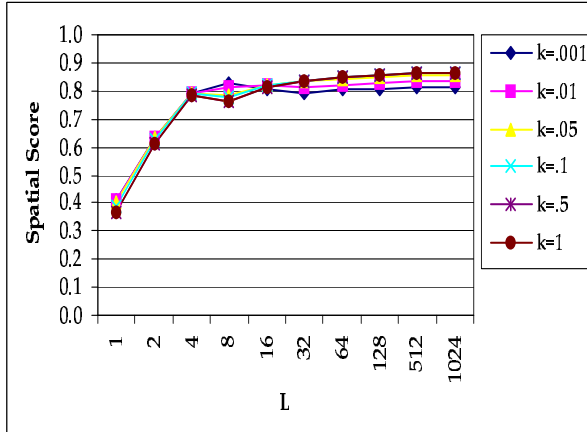


Figure 7: Mapping of \mathbf{L} to spatial score

Figure 7 charts the mapping function between \mathbf{L} and our spatial score under various assumptions for \mathbf{K} . These results indicate that the spatial score is, for the most part, an increasing function of \mathbf{L} that asymptotically approaches a value near .9, chiefly independent of \mathbf{K} . The sporadically anomalous behavior starting at $\mathbf{L}=8$ stems from a compiler optimization that begins to exercise a new basic block only when \mathbf{L} grows larger than four.

Based on these results, Apex-MAP covers a spatial score range of approximately .35-.85 and a temporal score range of approximately .02-.55. It is possible that the high end of the temporal score could be in-

creased using yet smaller values of \mathbf{K} with which we have not experimented.

Figure 8 plots the performance of Apex-MAP on the SC45 as a function of its spatial and temporal scores. We interpolate this surface using results from Apex-MAP runs parameterized by all combinations of $\mathbf{L}=\{1, 2, 4\}$ and $\mathbf{K}=\{.001, .01, .05, .1, .5, 1\}$. The surface confirms that our spatial and temporal scores relate intuitively to the performance of Apex-MAP and its notions of locality.

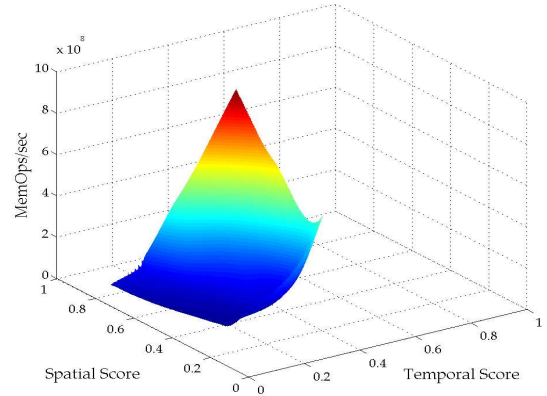


Figure 8: Locality scores track performance on Apex-MAP

4.2.2 Apex-MAP To Other Codes

In the previous section, we developed a mapping between the abstract spatial and temporal parameters of Apex-MAP and our observable locality metrics. These mappings enable us to measure the locality score of a given application and find the corresponding values of \mathbf{L} and \mathbf{K} . To find the \mathbf{L} and \mathbf{K} values for an application of interest:

1. Use Metasim on the code of interest to trace and calculate its temporal and spatial locality scores.
2. Consult the interpolations of Figure 7 to find the \mathbf{L} value corresponding to the application's spatial score.
3. Select the curve on Figure 6 that corresponds to the \mathbf{L} value derived in the previous step. To find \mathbf{K} , select the point on the curve that corresponds to the application's temporal score.

Using the above procedure, we can configure Apex-MAP to perform memory accesses with the locality

S, T	L, K	Apex-MAP Performance (memOps/s)	CG.A.1 Per- formance (memOps/s)	Error
.68, .33	3, .01	3.25×10^8	3.75×10^8	7.5%

Table 1: Mapping Apex-MAP to CG.A Yields Similar Performance

of an application of interest, creating a lightweight memory performance probe to represent it.

The tracer code for Metasim, and instructions for running it, along with source code and instructions for Apex-MAP are at www.sdsc.edu/pmac.

One should not assume that an application will get the same performance in memory operations per second as its Apex-MAP representative. This may be true though if memory performance is the limiting factor.

Table 1 displays an analysis of CG.A, a serial benchmark we assume may be memory bound, using Apex-MAP. The spatial and temporal scores of this application are .68 and .33 respectively, as displayed in Figure 1. Using the linear interpolation shown in Figure 7, we can interpolate a spatial score of .68 to map to an L -value of 3. Based on the $L=2$ and $L=4$ curves presented in Figure 6, we could choose $K=.01$ as the approximate value for K . On our test system, a single 1-GHz Alpha processor completed CG.A at a memory operation rate of 3.75×10^8 operations per second. Apex-MAP at $L=3$ and $K=.01$ performs at a rate of approximately 3.25×10^8 operations per second, a 7.5% discrepancy.

The end-result of this section is that we now have a recipe for determining L and K parameter settings for Apex-MAP that should cause it to mimic the temporal and spatial locality behavior of a chosen application.

5 Other Uses

The presented methodology for locality analysis has many potential uses for helping us understand application performance. One example is using the analysis to understand how applications scale as problem size or processor counts grow. Figure 9 plots the locality scores of CG as the problem size scales between classes S, W, and A. The graph also shows the average spatial and temporal locality of the memory work assigned to each processor as processor counts scale.

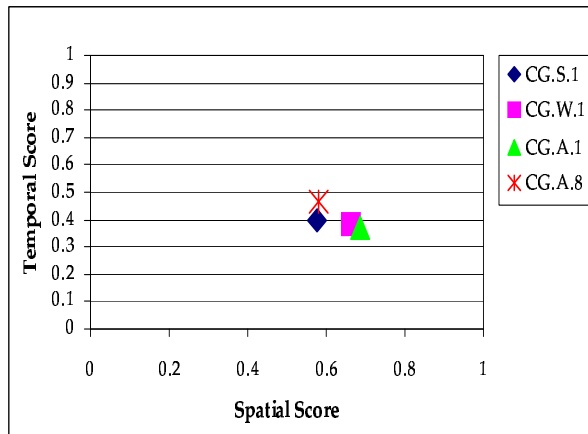


Figure 9: Locality scoring captures how the memory work of CG changes with problem size

We see that as input size increases, temporal score decreases, because the array sizes and size of the working set increases. At the same time, spatial score increases with long runs of stride 1 references through bigger arrays. In this simple case, everything matches intuition although the behavior of large applications could be less intuitive.

Splitting the CG class A problem across 8 CPUs (comparing CG.A.1 to CG.A.8) reduces the size of the working set per-processor and thus increases the temporal score while spatial score drops due to diminishing array/loop bounds.

6 Related Work

Temporal locality, perhaps because it is a little subtle to reason about, has a rich history in the literature. In 1970, Mattson et al. studied stack algorithms in cache management and defined the concept of stack distance [1]. Reuse distance is simply the same as LRU stack distance or stack distance using LRU replacement policy. In particular, Bunt has produced a host of publications studying both kinds of locality ranging from his seminal papers [11, 24, 23] to various investigations of metrics and measurement techniques. He recently summarized his work [12], 20 years after first publishing in the area [23]. Our approach is to adapt these ideas and make them practical and suitable for analysis of HPC parallel applications via dynamic tracing.

In the early 1980s Smith's seminal work points out the opportunities for cache to take advantage of both

kinds of locality [25]. This either presaged or, to some extent, sparked the cache-based system revolution. Subsequently, in the late 1980s Agarwal and Snir published several papers modeling deepening memory hierarchies using metrics of locality [4, 3]. Likewise, Carter and Alpern [7] made contributions in this area around the same time. We reviewed those works and consulted with Carter in adapting those ideas to this work.

Trace-driven simulation and analysis has its own rich literature; Uhlig and Mudge provide a good summary up progress until late 1990s [32]. Our work has leveraged and extended the state-of-the-art to make parallel HPC applications tracing fast enough to be practical [27].

Concepts of temporal and spatial locality are now so well developed they are taught to undergraduates in the classic Hennessy and Patterson text [19]. Nevertheless, as mentioned, the details of exactly how to measure them are left to the reader. We drew on more detailed ideas as of Bunt, Agarwal, Carter etc. above to devise metrics that are concrete and practical to measure for HPC applications.

Using reuse distance to reason about and improve performance is a well developed area [9, 16] and there is also work in exploiting spatial locality for performance as in the work of Torellas [31], Johnson [20], and Kumar [21]. From the standpoint of this work these provide more evidence that the capability of measuring and benchmarking locality of HPC applications is important.

Work such as Song's [28] is representative of a whole area that uses compilers to improve code locality, especially temporal locality, to improve performance. We refer the reader to Allen and Kennedy's work [5] for more in depth background on this area; there simply is not room here to cite all the good compiler analysis for locality papers.

Darema et al. characterized a scientific workload with respect to memory access patterns in 1987 [15]. This work aims to update that kind of capability. Harrison did similar work on pointer-chasing and numeric programs of the day [18]. Ding et al. predict locality based on reuse distance [16]; our focus is rather to measure locality directly.

Almasi et al. have proposed ways to calculate stack distance efficiently [6]. Our approach is somewhat similar in spirit; we propose a tractable approximation.

7 Conclusions and Future Work

We have proposed and implemented a concrete methodology whereby benchmarks and applications can be scored for spatial and temporal locality. We used it to confirm that the HPC Challenge Benchmarks cover an interesting space along these dimensions. We provided a recipe for tracing an arbitrary application and determining the **L** and **K** values that can be used to configure Apex-MAP into a succinct benchmark proxy for the spatial and temporal characteristics of the application. The tracer and Apex-MAP are available at www.sdsc.edu/pmac.

This coming year we are commissioned by the DoD HPC Modernization Program to acquire memory statistics on the following full-scale applications and inputs.

AVUS Standard - The Air Force Research Laboratory (AFRL) developed AVUS to determine the fluid flow and turbulence of projectiles and air vehicles.

HYCOM Standard - The Naval Research Laboratory (NRL), Los Alamos National Laboratory (LANL), and the University of Miami developed HYCOM as an upgrade to MICOM (both well known ocean modeling codes). HYCOM's standard test case models all of the world's oceans as one global body of water at a resolution of one-fourth of a degree when measured at the Equator.

OVERFLOW2 Standard - NASA Langley and NASA Ames developed this application to solve CFD equations on a set of overlapping, adaptive grids, such that the grid resolution near an obstacle is higher than that of other portions of the scene.

RFCTH - Sandia National Laboratories (SNL) developed CTH to model complex multidimensional, multiple-material scenarios involving large deformations or strong shock physics. RFCTH is a non-export-controlled version of CTH.

As part of that work we will provide spatial and temporal locality scores for these strategic codes, plot them on axes of spatial and temporal locality, and explore the benchmarking implications of representing them by Apex-MAP. Potentially, benchmarking

for procurement could be much cheaper and easier if, instead of deploying all these codes on prospective machines, one could run Apex-MAP with a few judicious parameters and get the same performance information concerning the memory subsystems.

We are currently experimenting with a spatial scoring methodology that folds the size of the look back window into an application's spatial score in much the same manner as reuse distance is folded into its temporal score. Scoring each memory reference as a function of both its stride length and the size of the look back window required to identify such lengths would allow us to plot spatial reuse functions in a manner very much analogous to the temporal reuse functions we have presented here. We can modify Metasim to collect this information without noticeable slowdown in the tracing performance.

8 Acknowledgements

This work was supported in part by NSF awards CNS-0406312, the DOE Office of Science through the award entitled HPCS Execution Time Evaluation, by NSF NGS Award #0406312: Performance Measurement & Modeling of Deep Hierarchy Systems, and by the Department of Energy Office of Science through SciDAC award High-End Computer System Performance: Science and Engineering. Computer time was provided by the Pittsburgh Supercomputing Center via an NSF NRAC award. We would also like to thank Bob Lucas, John McAlpin, David Koester, Jeffrey Vetter, Pedro Diniz, and Larry Carter for their invaluable input on this work.

References

- [1] Hpc challenge: <http://icl.cs.utk.edu/hpcc/>.
- [2] Top500: <http://www.top500.org>.
- [3] A. Aggarwal, B. Alpern, A. Chandra, and M. Snir. A model for hierarchical memory. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 305–314, 1987.
- [4] A. Aggarwal, A. Chandra, and M. Snir. Hierarchical memory with block transfer. In *FOCS '87: Proceedings of the 28th Annual IEEE Symposium on Foundations of Computer Science*, pages 204–216, 1987.
- [5] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 2001.
- [6] G. Almasi, C. Cascaval, and D. Padua. Calculating stack distances efficiently. In *Proceedings of the first ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, 2002.
- [7] B. Alpern, L. Carter, E. Feig, and T. Selker. The uniform memory hierarchy model of computation. In *Tech. Rep. Cornell University*, pages 93–119, 1993.
- [8] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The nas parallel benchmarks. *International Journal of Supercomputer Applications*, 5(2):63–73, 1991.
- [9] K. Beyls and E. Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of PDCS*, pages 617–662, 2001.
- [10] B. Buck and J. Hollingsworth. An api for runtime code patching. *Journal of High Performance Computing Applications*, 14(4), 2000.
- [11] R. Bunt and J. Murphy. Measurement of locality and the behaviour of programs. *The Computer Journal*, 27(3):238–245, 1984.
- [12] R. Bunt and C. Williamson. Temporal and spatial locality: A time and a place for everything. In *Proceedings of the International Symposium in Honour of Professor Guenter Haring's 60th Birthday*, 2003.
- [13] L. Carrington, A. Snively, X. Gao, and N. Wolter. Performance prediction framework for scientific applications. *Lecture Notes in Computer Science*, 2659:926–935, January 2003.
- [14] L. Carrington, N. Wolter, A. Snively, and C. B. Lee. Applying an automated framework to produce accurate blind performance predictions of full-scale hpc applications. In *Proceedings of the 2004 Department of Defense Users Group Conference*. IEEE Computer Society Press, 2004.
- [15] F. Darema-Rogers, G. Pfister, and K. So. Memory access patterns of parallel scientific programs. In *SIGMETRICS 87: Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 46–58, New York, NY, 1987. ACM Press.

- [16] C. Ding and Y. Zhong. Predicting wholeprogram locality through reuse distance analysis. In *PLDI 03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 245–157. ACM Press, 2003.
- [17] J. Dongarra, P. Luszczek, and A. Petitet. The linpack benchmark: Past, present and future. *Concurrency: Practice and Experience*, 15:803–820, 2003.
- [18] L. Harrison. Examination of a memory access classification scheme for pointer-intensive and numeric programs. In *Proceedings of the 10th International Conference on Supercomputing*, pages 131–140, New York, NY, 1996. ACM Press.
- [19] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, CA, 1990.
- [20] T. Johnson, M. Merten, and W. Hwu. Runtime spatial locality detection and optimization. In *Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture*, pages 57–64, 1997.
- [21] S. Kumar and C. Wilkerson. Exploiting spatial locality in data caches using spatial footprints. In *ISCA 98: Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 357–368. IEEE Computer Society, 1998.
- [22] G. Marin and J. Mellor-Crummey. Crossarchitecture performance predictions for scientific applications using parameterized models. In *SIGMETRICS 2004 / PERFORMANCE 2004: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 2–13, New York, NY, 2004. ACM Press.
- [23] J. Peachey, R. Bunt, and C. Colbourn. Towards an intrinsic measure of program locality. In *Proceedings of the Sixteenth Annual Hawaii International Conference on System Sciences*, pages 128–137, 1983.
- [24] J. M. R.B. Bunt and S. Majumdar. A measure of program locality and its application. In *Proceedings of the 1984 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 28–40, 1984.
- [25] A. Smith. Cache memories. *ACM Computing Survey*, 14(3):473–530, 1982.
- [26] A. Snaveley, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for application performance modeling and prediction. In *Supercomputing 02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–17, Los Alamitos, CA, 2002. IEEE Computer Society Press.
- [27] A. Snaveley, N. Wolter, and L. Carrington. Modeling application performance by convolving machine signatures with application profiles. In *Proceedings of IEEE 4th Annual Workshop on Workload Characterization*, pages 128–137, December 2001.
- [28] Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *PLDI 99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 215–228. ACM Press, 1999.
- [29] A. Srivastava and A. Eustace. Atom: A flexible interface for building high performance program analysis tools. In *Proceedings of the Winter 1995 USENIX Conference*, January 1995.
- [30] E. Strohmaier and H. Shan. Architecture independent performance characterization and benchmarking for scientific applications. In *International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, Volendam, The Netherlands, 2004.
- [31] J. Torrellas, M. Lam, and J. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.
- [32] R. A. Uhlig and T. N. Mudge. Trace-driven memory simulation: A survey. *ACM Computing Surveys*, 29(2):128–170, 1997.