A role for Pareto optimality in mining performance data

***Joël M. Malard*[*§]**

Pacific Northwest National Laboratory[†]
Battelle Boulevard, P.O. Box 999
Richland, WA 99352

Abstract

Improvements in performance modeling and identification of computational regimes within software libraries is a critical first step in developing software libraries that are truly agile with respect to the application as well as to the hardware. It is shown here that Pareto ranking, a concept from multi-objective optimization, can be an effective tool for mining large performance datasets. The approach is illustrated using software performance data gathered using both the public domain LAPACK library and an asynchronous communication library based on IBM LAPI active message library.

1. Introduction

The porting of a software library across computers of different architectures entails more than just running the existing code on the target machine. Performance can suffer much from a straightforward port. The first line of attack is to seek optimal values of implementation parameters, such as block sizes, unrolling loop level, etc. Deep memory hierarchies ranging from scalar and vector registers, several levels of cache and to possibly remote memory or disk as well as other advanced architectural features all increase the complexity of software performance tuning, see for example[10]. The second line of attack is often to replace the underlying algorithm, a costly process. Performance tuning can thus be a major operational cost for software that spans several generations of computers.

The ideal *agile* software library would auto-calibrate its internal algorithms and parameters to maximize the performance of any given application using it in any given runtime environment. Agile software libraries that implement the sequential Basic Linear Algebra Subroutines (BLAS) interface[4] such as ATLAS[37] and PHiPAC[3] calibrate some of their internal parameters and algorithms during an initial training phase at installation time. The success of these libraries is in good part based on a precise knowledge of which software parameters matter for dense linear algebra on most computer architecture. The main three components of the PHiPAC approach are: an automatic code generator that produces the trial source code, support for robust benchmarking of the resulting code, and a search algorithm that explores the space of feasible codes. The scope of PHiPAC extends beyond linear algebra but its main impact so far has been in generating optimized matrix-matrix multiplication software. The most common algorithms for matrix-matrix multiplication have the property that they can be expressed in terms of either matrix blocks or matrix elements without altering their overall control structure. The matrix-matrix multiplication algorithms generated by PHiPAC are composed of nested matrix multiplication algorithms that achieve an optimal trade-off between hardware features for a given matrix size. At the *register* level, all loops are unrolled. The optimal amount of loop unrolling, or *register block size*, is determined by the relative speed of integer arithmetic compared to that of floating point arithmetic. At the next higher level, all matrix blocks must fit entirely in the level-1 data cache. The optimal *level-1 cache block size* is determined by the relative speed of level-1 cache accesses compared to floating-point arithmetic. The next higher level is tuned with respect to the level-2 data cache, etc. The ATLAS Project on the other hand, focuses

[**] Computational Science and Applied Mathematics, Pacific Northwest National Laboratory, Battelle Boulevard, PO Box 999,Richland, WA 99352
[§] Email: jm.malard@pnl.gov
[†]The Pacific Northwest National Laboratory is operated by Battelle Memorial Institute for the U.S. Department of Energy under Contract DE-AC06-76RLO 1830.

on linear algebraic computations but considers a search space that extends beyond register and data cache block sizes; it also covers matrix-vector operations that are significantly more data intensive than the matrix-matrix operations. The *automated empirical optimization software* (AEOS) approach of ATLAS and PHiPAC can be very effective and it is being actively applied to other domains such as signal processing algorithms[13, 32]. The Fast Fourier Transform (FFT) and related transforms can be seen as specialized matrix multiplications and in that sense they are closely related to PHiPAC and ATLAS. They do have some additional levels of complexity. First, the prime number decomposition of the size of their input, which was first exploited in adaptive software by Frigo[13], determines the required number of floating point operations. Second, those computations are data intensive. This last point raises what may be the main obstacle to extending the AEOS approach to distributed memory computing platforms, or multi-computers, namely the need to tune communication operations for optimal performance.

Extending the two-stage approach of ATLAS and PHiPAC to larger libraries, such as LAPACK[2] or SCALAPACK[7], is an active area of research. One hurdle is the size of the datasets that results when all software parameters are taken into account. For example, a parallel algorithm for solving a dense distributed system of linear equations AX=B, where A is a mxn and B is mxq has about 8 parameters: 3 parameters govern the size of the problem, 2 parameters govern the shape of the process mesh, 2 parameters govern the data distribution of A, one arithmetic block size governs the size of Level 3 BLAS arguments[14, 20]. The problem sizes and the arithmetic block size affect the amount of level-3 BLAS operations that can be performed, which in turns has an impact on which algorithm is most efficient. On the other hand, the parameters governing the data distribution impact the number of communications and their volume. The relationship between such software parameters and hardware features that drive the wall-clock time is complex. First, one needs to determine which hardware features drive the wall-clock time for a given class of library applications and then one needs to relate those hardware features to software parameters that are under the control of the library or the application. A third aspect becomes important in the context of automatically tuned, *agile*, libraries: due to the complexity of the algorithms and the generality of the library API, different hardware events drive the wall-clock time performance for different applications of the library. For instance, the wall clock time for solving AX=B with a dense matrix A may be related most closely to the number of branches and the amount of integer arithmetic for a small matrix, to the number memory load and store operations for a medium sized matrix, and to the amount of floating point operations for very large matrices. Advances in computer hardware and architecture have enabled the development of on-chip registers, so called *hardware event counters*, that can tally such things as data and instruction cache misses, branch misprediction, etc. These hardware event counters are now widely available through Application Programming Interfaces (API) such as PAPI[5], RS2HPM[23], Rabbit[19], Photon MPI[36], WatchTower[21]. It is thus possible to gather datasets that combine software parameters and hardware event counts including wall-clock time. Such datasets enable the systematic search for *performance regimes*, which are regions of the input space where some specific combinations of hardware event counts are highly correlated to the wall clock time. Agile software libraries should be able to detect and adapt to changes in performance regimes.

Floating point arithmetic dominates the performance of most linear system solvers for very large dense linear systems, but communication is dominant for medium sized distributed linear systems. Automatic tuning of communication is therefore at the center of the problem. There is a large body of research aimed at the optimization of communication within specific parallel applications such as linear algebra, FFT's etc. There is also a growing body of research done on optimization of generic communications[9, 27, 29]. Whereas the execution time of any basic floating-point operation can be assumed constant, the execution time of point-to-point communication, such as send and receive, depends on the length of its message. This complicates matters, because the optimization can either reduce the number of communications or the volume of data moved across process boundaries. Collective operations can be very effective for dense linear algebra computations[26] but their runtime depends also on the number of processors executing the operation. The automatic tuning of collective communication has recently received some attention[34]. Collective communication operations defined in the Message Passing Interface Standard (MPI)[15] are blocking. Each operation corresponds to a single procedure call that should be issued by all active processes before the operation can complete. Such operations have a well-defined runtime and are good candidates for automatic tuning. Asynchronous communication operations, implemented on top of multiple threads[30], one-sided communications[15, 33] or active messages[22, 6] may be key to scalable statistical estimation algorithms[24, 25], but pose other challenges if only because their runtime is not so well defined. They have an initiation time, a completion time in addition to a barrier time that each correspond to a distinct phase of communication. The *initiation* time is the time required for the origin process to issue a communication request to a target process. The *completion* time is the time needed for the communication operation to complete at the origin once it has been initialized. The *barrier* or global time is the minimum time between two barriers that enclose this communication operation.

Communication optimization at the application level is often most profitable but harder to extend to other applications; it is akin to replacing a dense matrix computation by a faster sparse matrix computation. Optimization at the middleware level could be for instance to switch between two possible broadcast algorithms based on the length of the message and the number of processors. Such optimization is akin to replacing a loop-based matrix-matrix multiplication algorithm by the theoretically faster recursive Strassen Algorithm. If one follows the analogy between floating point and communication operations, a natural question to ask is what are the block sizes that characterize communication operations and which can be profitably incorporated into an Application Programming Interface (API)? That problem is hard: the level of noise introduced by the underlying software and hardware can be very high, the performance data has more dimensions than for floating point computations, the data itself has some statistical properties that makes it unique. Software performance data is typically unbalanced, possibly longitudinal, meaning that the same events are measured many times while others may be measured only a few times. Missing values do occur because of overflows in hardware event counters but they are rarely missing at random. Observations may be statistically dependant due to contentions for network or CPU resources.

The problem addressed in this paper is that of finding related hardware events in a large performance dataset. Commonly several possibly conflicting "measures" of relatedness are agglomerated together into a single figure of merit from which experiments can be designed and conclusions can be drawn. However, the fine details of the associations between hardware events can be lost that way. It is shown experimentally how concepts from multi-objective (or vector) optimization[11] may be enable the extraction of fine structure in the data that may not be attainable by classical data mining techniques. This is a small but important step toward building agile communication libraries. The relevant concepts from data mining and mathematical optimization are reviewed in Section 2. That section concludes with an application of these concepts to the LAPACK library. Although the LAPACK library is well known and sequential, that first example provides some validation of the subsequent application to a novel communication middleware. The latter middleware was the main reason why this work on data mining was undertaken in the first place. Section 3 is therefore the main section. Conclusions and directions of further research appear in Section 4.

2. Performance Data

Multivariate statistics have been used for some time in the context of performance modeling of parallel software. Recently Clement and Quinn[8] showed how to compute confidence intervals on the expected runtime of programs using statistical linear models. A (fixed effects) linear model consists of a system of linear equations, say $r = Ax+e$ together with some assumption on the statistical distribution of the components of the random error vector e. The matrix A corresponds to known parameters and the vector x is a vector of unknown but fixed parameters. The purpose of the computation is to estimate x and e in such a way that the estimated error vector matches some given statistical assumptions as closely as possible. An important case is when the components of the vector e are assumed independent and normally distributed with zero mean and unit variance; the computed estimate for x is then exactly the least-squares estimate. The two important aspects of a linear model is that first, one can incorporate a-priori knowledge of the structure of the error; and second, one can compute confidence intervals for e and hence for r. However, the model is linear and may be accurate on only part of the data. Figure 1 illustrates this point by showing the base 10 logarithm of the wall-clock time for a straightforward broadcast of a length h vector of double precision numbers across a network of $p/2$ Symmetric Memory Multiprocessors (SMPs) with 2 processors per node. The actual platform is a cluster of HP workstations with 1.2 GHz Itanium II dual processor nodes connected by a Quadrics Elan3 Interconnect. The parameter space has two dimensions: the number p of processors and the length h of the vector; it was sampled along a logarithmic grid in both dimensions from 1 to 128 processors and from 1 to 2^{24}=16777216 64-bit numbers. All the measurements, including repetitions, took about 128 CPU hours. For actual numerical kernels such as parallel matrix-matrix multiplications, CPU resource allocations and budgets can make random sampling of the parameter space a necessity. The main point of Figure 1 is that four, possibly five, regions can be seen, each corresponding potentially to a different linear model. The first region, denoted A, contains all measurements for a single processor, which involve no communication. The second region, denoted B, consists of all two-processors measurements, which involve a single send operation although not necessarily between two processors on the same node. The third region (C) in Figure 1 consists of all measurements for more than two processors but with vector lengths vector lengths not exceeding 65563. The fourth region (D) includes all other measurements. The upper left corner of the latter region corresponds to a wall clock time of 14.3 seconds with its nearest neighbors on the grid at 5.37 seconds for 128 processors, and 3.56 and 1.34 seconds for 64 processors. The legend for this plot is misleading because no measurement exceeds 14.3 seconds.

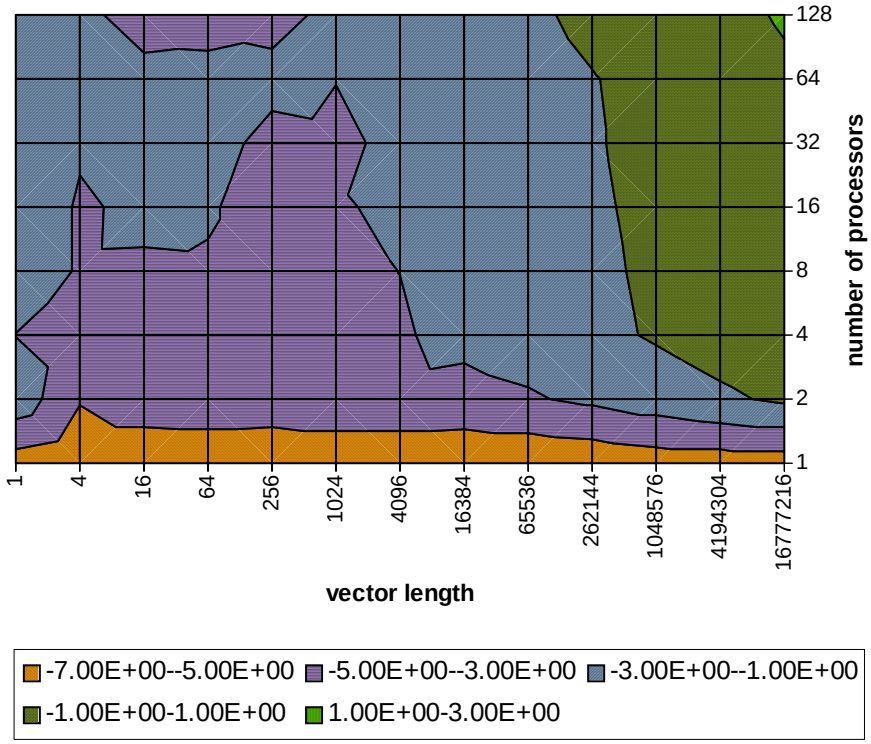

Figure 1 Base 10 logarithm of the runtime of MPI_Bcast on a cluster of 2-way SMP nodes, see text for details.

Table 1 shows clearly that different regions of the input space correspond to different least squares estimates. The runtime r of a broadcast on 1 and 2 processors, regions A and B respectively, is modeled as $r=c+\tau h$, where λ is the latency and $\beta=64*10^6/\tau$ is the bandwidth in MB/s and h is the message length in bytes. Over the other regions, the two most common runtime models of broadcast are $r=(\lambda+\tau h)log(p)$ for algorithms based on a single process tree and $r=\lambda log(p)+\tau h$ for scatter-gather broadcasts based on recursive-doubling; only the latter model yields realistic bandwidth estimates. Table 1 shows a fifth region (E) that encloses tightly the right-hand side upper corner of Figure 1. Latency estimates λ* vary widely. On this particular system the peak bandwidth for point-to-point communication across nodes is 300MB/s latency: the bandwidth estimates β* on the bottom four rows are realistic. The larger than real-life β* computed for the 2-processor case reflects that some processor pairs reside on the same node. There does not appear to be a single estimate of either latency or bandwidth that is representative of the whole sampled input space. A natural question then is how to find a minimum number of convex regions that cover the sample space and within which least squares estimates are accurate. That problem is closely related to the problem of identifying performance regimes.

Table 1 Least squares estimates of latency and bandwidth depend on which subset of the data is used.

Region	*log(p)*	*log(h)*	λ*	β*
A	0	0..24	.334μs	*875TB/s*
B	1	0..24	2.46ms	*5.29GB/s*
C	2 .. 7	0 .. 16	966μs	215MB/s
D	2 .. 7	18 .. 24	125ms	220MB/s
B+C+D	1 .. 7	0 .. 24	28.4ms	198MB/s
E	5 .. 7	22 .. 24	81.1ms	143MB/s

Vetter *et al*[1, 35, 36] advocate the use of several traditional multidimensional statistics in the context of high-performance computing, mostly based on data clustering. For instance, they were able, by clustering trace data, to differentiate between slave and master processes as well as between processes communicating through OpenMP from those

communicating through message passing. Ahn and Vetter also recommend using Factor Analysis and Principal Component Analysis (PCA) along with data clustering for finding small sets of representative hardware event counters that capture the runtime behavior of complex applications on Teraflop platforms. Factor analysis, like linear model estimation, regroups many procedures and algorithms to classify features (here hardware event counts) based on their pair-wise correlations; the correlation between for two events x and y with sample means <x> and <y> is ratio of the sample variance of the product xy by the product of the sample variances of x and y. Principal components are simply the left and right factors (U and V) of the singular value decomposition of the data matrix A=UDV' where U and V are orthogonal and D is a diagonal matrix. One difficulty with the SVD is that it is not defined when A has missing entries.

2.1 Data Clustering

The purpose of (data) clustering is to partition a set of observations into disjoint subsets (or *clusters*) so that observations within each cluster are in some sense more similar to one another than to observations belonging to other clusters, see for example the book by A. Gordon[17] and its bibliography. The notion of similarity can be implemented in many ways but very often, when the data is numeric it is defined as proximity in the Euclidean sense. The numerical data can also be transformed in many varied ways to increase the signal-to-noise ratio before a clustering algorithm is applied. Numeric data is often represented as a table with a fixed number of columns and one row for each observation. Missing values can be estimated (imputed), or can be left intact and treated as such by the clustering algorithm. It is customary to remove rows and columns that either contain too many missing values or whose values are all nearly equal. Commonly, the 'cleaned' data is rescaled so that either the rows or the columns have a mean zero and a unit standard deviation or so that their range lies between 0 and 1. The application of scaling and more elaborate transformations to the original data to expose the underlying structure is an art and several attempts may be necessary before some structure is observed which can then be analyzed statistically. This is easy to see because scaling the rows of a data table does not preserve their Euclidean distance.

There are two main strains of clustering algorithms. The iterative ones, such as *k-means*, start with some putative cluster definitions and repeatedly improve these definitions until no further progress occurs. In the case of k-means, each cluster is defined by its center of mass, or *centroid*. The choice of the initial centroids has an impact on the computed clusters. The second strain of clustering algorithm, *hierarchical clustering* algorithms produce a nested sequence of partition ranging from the partition that groups each observation into a separate cluster to the partition that regroups all observations into one cluster. The *agglomerative* hierarchical algorithms start from the former partition; *divisive* algorithms start from the latter. *Single-linkage* clustering can be implemented by removing from a minimum spanning tree of the data all those arcs that exceed a given threshold. The computed clusters are the remaining connected components. Single-linkage tends to produce elongated clusters and is often complemented by a few steps of another clustering algorithm such as k-means. The clustering algorithm used here is a slight variant of (agglomerative) *complete-linkage*. At each step of complete-linkage, the two clusters merged are those that result in a cluster of smallest possible diameter, those are the two nearest clusters.

The *diameter* of a cluster is the largest distance separating any two of its members. Commonly, the diameter is computed relative to the Euclidean distance and this is most efficiently done when all distances between pairs of clusters fit in an array in level-1 cache. The diameter computed from the L_1 distance is simply the longest side of the smallest box enclosing both clusters and whose sides are parallel to the coordinate axes. Using the L_1-norm eliminates the need to keep in main memory the matrix of pairwise distances. Second, every cluster that may still be merged into a bigger one is associated with a pointer to a *near* neighbor cluster. Traditionally those pointers are to a *nearest* neighbor cluster. When two clusters, say C_i and C_j are merged and form a new cluster C_k, all the nearest neighbor pointers directed at either C_i or C_j need to be updated. Say there is a pointer from C_h to C_i. In the absence of a distance matrix, one is forced to compute anew the distance between C_h and every other active cluster including C_k. Thus, for a dataset with n observations and in the absence of a distance matrix, the standard implementation of complete linkage may require up to $O(n^3)$ distance computation to terminate. Here, the near neighbor pointer of C_h is reset to a randomly selected cluster among the nearest ones that are not already pointing toward a nearer neighbor than C_h. The near neighbor links are set initially using a minimum spanning tree of the data. It has been observed in practice, that with this heuristic each merge requires at most three passes of the data on average in which case the overall runtime is quadratic, the worst case is still $O(n^3)$. The book by Gordon contains the details of other fast serial clustering algorithms. In terms of clustering, the main impact of these modifications is on the runtime but the substitution of nearest neighbor links by randomly selected near neighbor links could affect the computed clusters.

Finally, missing values are kept in the data and are accounted for during distance computations in a most standard way. Consider, for example, two vectors u and v, say of length k, with u_0 unknown and $u_{1..k-1}$ and $v_{0..k-1}$known; the distance between u and v is the distance between $u_{1..k-1}$ and $v_{1..k-1}$ multiplied by a factor $1-1/k$, recall that the L_1 metric is used here. Each code fragment for which counter increments was collected was manually enclosed in a loop that ranged over all measurable hardware events. The resulting readings were then inserted into a single data record. The reason for doing so is that the number of counters that can be read simultaneously is limited and that often many pairs of hardware events cannot be monitoring concurrently without some additional software support. In so doing, it is assumed that the hardware event counters would be incremented by (nearly) the same value at every step of the enclosing loop. To be exact, several counters could have been read simultaneously and inserted into a single data record with all other entries missing. The resulting dataset could have had in excess of 80% missing values, many more than can be accommodated by most existing statistical analysis algorithms.

The following example illustrates hierarchical clustering and will serve later to introduce concepts from vector optimization. The purpose is to see how six hardware events are related to one another during the completion phase of a distributed hash table[24]. There are 14268 samples from randomized micro benchmarks described in Section 3. The events are labeled: BRU_IDL_B (number of cycles the branch unit was idle), L2_STM_B (number of level 2 cache store misses), BTAC_M_B (number of branch target cache misses), PRF_DM_B (number of data prefetch cache misses), PR_PRC_B (number of correctly predicted conditional branches) and SYC_INS_B (number of completed synchronization operations). Those events correspond to a completion phase and the suffix _B stands for blocking. The number of events was chosen so that the example remains manageable for a printed document. Here the vector of values associated with any particular hardware event is called a *profile*. Each profile has been scaled so that its entries range between zero and a unit. Hardware events that are tightly coupled are likely to correspond to similar profiles, and one may further expect that the Euclidean distance between these profiles once scaled will be small. That is profiles that form tight clusters may also measure closely related things. Table 2 shows the names and distances between every pair of profile vectors. The first cluster defined by complete-linkage is C_7={L2_STM_B, BTAC_M_B} whose diameter is 1.63. At this point {L2_STM_B} and {BTAC_M_B} are removed from consideration and nearest neighbor relationships are recomputed using the newly formed cluster. For small data sets, the rows in Table 2 that correspond to L2_STM_B and BTAC_M_B could be merged into a single row for the new cluster, and similarly for the columns. The operation is simple; the entries in the new row for C_7 are the maximum values for the corresponding entries of the rows for L2_STM_B and BTAC_M_B, similarly for the new column.

Table 2 All Euclidean distances among six profiles vectors: the minimum distance is shown in bold face.

	BRU_IDL_B	L2_STM_B	BTAC_M_B	PRF_DM_B	BR_PRC_B	SYC_INS_B
BRU_IDL_B	0	3.08	3.86	2.74	1.86	2.61
L2_STM_B	3.08	0	**1.63**	2.17	3.52	4.59
BTAC_M_B	3.86	**1.63**	0	1.99	4.37	5.43
PRF_DM_B	2.74	2.17	1.99	0	3.20	4.20
BR_PRC_B	1.86	3.52	4.37	3.20	0	1.79
SYC_INS_B	2.61	4.59	5.43	4.20	1.79	0

The next clusters are in order C_8={BR_PRC_B, SYC_INS_B} with radius 1.79; C_9={C_7, PRF_DM_B} with radius 2.17; C_{10}={C_8, BRU_IDL_B} with radius 2.61; and C_{11}={C_9, C_{10}} with radius 5.43. The whole process is captured in a tree, known as a *dendogram*, where the profiles are represented by leaves and computed clusters correspond to internal nodes. The dendogram for the data shown in Table 2 appears in Figure 2. The number of clusters in the final partition is determined by stopping the agglomeration at an appropriate step, and this flexibility compared to say k-means, is another distinguishing feature of hierarchical clustering algorithms. Once a dendogram has been computed the question remains as to which clusters are real and which are numerical artifacts.

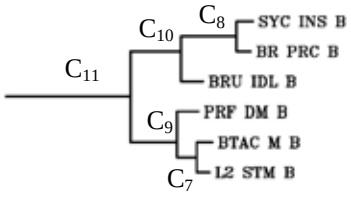

Figure 2 Six hardware event profiles and the corresponding dendogram computed by complete linkage.

2.2 Pareto Ranking

The estimation of the number of clusters present in a dataset involves several conflicting goals. On one hand, the computed clusters would ideally be as homogeneous as possible; *homogeneity* can be defined in many ways, for instance as the diameter of the given cluster, as the maximum distance from its centroid, or even as the likelihood of its members obeying a Gaussian distribution. On the other hand, the computed clusters would ideally be as far apart as possible; *separation* between two clusters can also be defined in many ways, such as the diameter of the union of the two clusters, or the distance between their centroids. Although quantitative measures of separation and homogeneity (or its opposite *heterogeneity*) need not be commensurate, they are routinely combined into a single figure of merit that is included into a threshold rule. One such rule found in practice is that the ratio R of the diameter of the largest cluster to that of the entire dataset should be as large as possible without exceeding a given threshold (say 75%). Milligan and Cooper[28] review 30 such rules and do not find one that is systematically better. In addition to homogeneity and separation, a priori knowledge might constrain cluster membership, or one might be interested in separation and homogeneity with respect to similarity measures that did not drive the classification. For the illustration, consider three *"measures"* of partition quality based on using the L_2 distance: H_1 is the average diameter of active clusters, S_1 is the average minimal distance between nearest neighbor clusters, and $R_{.75}=|R-0.75|$ the absolute value of the difference between R and its threshold 0.75.

The values of S_1, H_1 and R for each step of complete linkage applied to the data in Table 2 are shown in Table 3. For clarity sake, denote by $\mathbb{P}_k$ the partition of the data into k clusters. The best partition according to the 75% rule is $\mathbb{P}_2$. From a different perspective, none of $\mathbb{P}_6$ through $\mathbb{P}_3$ is better than the others with respect to both maximizing S_1 and minimizing H_1. The technical term is that $\mathbb{P}_6$ through $\mathbb{P}_3$ are *non-dominated* (in the *Pareto* sense and relative to the set of computed partitions). The next before last partition, $\mathbb{P}_2$, is dominated by $\mathbb{P}_3$ but not any of $\mathbb{P}_4$, $\mathbb{P}_5$ or $\mathbb{P}_6$. Along this line of reasoning $\mathbb{P}_3$ would be the natural partition to choose. The important point here is that either ways a decision is made based on what are the important characteristics of good partitions for the problem at hand. Pareto ranking provides a means to explore such (often implicit) assumptions.

Table 3: Measures of separation and homogeneity for each step of complete linkage for the data in Table 2.

Partition Name	Last Cluster Formed	S_1	H_1	$R_{.75}$
$\mathbb{P}_6$	Initially	1.78	0.00	0.75
$\mathbb{P}_5$	C_7	1.96	0.33	0.45
$\mathbb{P}_4$	C_8	2.39	0.86	0.42
$\mathbb{P}_3$	C_9	3.03	1.92	0.35
$\mathbb{P}_2$	C_{10}	2.82	2.71	0.04
$\mathbb{P}_1$	C_{11}	N/A	5.43	0.25

Let there be some number of independent measures of partition quality that one particularly cares to maximize, say $\mu_1 = S_1$, $\mu_2 = -H_1$, and $\mu_3 = -R_{.75}$. The best partitions among all those computed are those that cannot be improved on all accounts. Formally, a partition $\mathbb{P}_k$ is *non-dominated* in the Pareto sense (relative to the set of computed partitions) if there does not exist another partition $\mathbb{P}_h$ such that: $\mu_i(\mathbb{P}_k) \le \mu_i(\mathbb{P}_h)$ for every $1 \le i \le 3$ and at least one of those inequality is strict. In this paper, a non-dominated partition will be called *efficient* although strictly speaking the partition may be dominated with respect to the set of all possible partitions of the data. In what follows, a *strong cluster* is one whose formation resulted in an efficient partition. Pareto efficiency can be used as above to estimate the number of clusters in a dataset; but the actual usefulness of Pareto dominance actually comes from the light it sheds on the transitions from "strong" to "weaker" partitions and vice-versa within a single dendogram as well as between related dendograms. The

notion of strength is translated here in terms of *Pareto ranks*. Efficient partitions are assigned a rank of 0. The Goldberg ranking algorithm[16] assigns rank $k+1$ to all those partitions that would become efficient if all partitions of rank at most k were removed from consideration. In essence, the Goldberg rank measures the distance between a datum and the set of non-dominated vectors; its worst-case computational complexity for n partitions relative to h metrics is $O(n^3h)$. The Fonseca-Fleming rank[12] of a partition is the number of partitions that dominate it; its worst-case complexity is $O(n^2h)$. The *Pareto rank of a cluster* computed by a hierarchical algorithm is the rank of the corresponding partition of the data. Neither ranking scheme "dominates " the other. The Goldberg rank is typically more uniform but also harder to compute than the Fonseca-Fleming rank because it has as many steps as there are levels in the dataset. The Fonseca-Fleming scheme penalizes partitions in densely sampled regions of the Pareto front, which is the set of globally dominant vectors, and although parallel ranking algorithms are not considered here their worst case communication requirement is $O(n^2h)$. The Goldberg rank is illustrated in Table 4; it is used throughout the paper. The second through seventh columns of Table 4 show the relative strengths of the corresponding partitions relative to μ_1, μ_2 and μ_3 in that order. For example, the 101 entry at the intersection of the row for $\mathbb{P}_5$ with the column for $\mathbb{P}_6$ means that $\mu_1(\mathbb{P}_5)>\mu_1(\mathbb{P}_6)$, $\mu_2(\mathbb{P}_5)\leq\mu_2(\mathbb{P}_6)$, and $\mu_3(\mathbb{P}_5)>\mu_3(\mathbb{P}_6)$ or equivalently that $S_1(\mathbb{P}_5)>S_1(\mathbb{P}_6)$, $H_1(\mathbb{P}_5)\geq H_1(\mathbb{P}_6)$, and $R(\mathbb{P}_5)<R(\mathbb{P}_6)$. The symbol x is used to denote a missing value. The rightmost four columns show the Goldberg rank computed for various combinations of quality measures, the ± superscripts denote maximization or minimization of the corresponding measure. In the text that follows, unless otherwise specified, the Goldberg rank is computed relative to S_1^+, H_1^- and maximizing the minimum cluster density, which is the ratio of the number of data records in the cluster divided by the volume of the smallest box enclosing that cluster.

Table 4 Goldberg ranks for the data in Table 3 using different combinations of cluster quality measures.

Partition Name	$\mathbb{P}_6$	$\mathbb{P}_5$	$\mathbb{P}_4$	$\mathbb{P}_3$	$\mathbb{P}_2$	$\mathbb{P}_1$	$S_1^+,H_1^-,R_{.75}^-$	S_1^+,H_1^-	$S_1^+,R_{.75}^-$	$H_1^-,R_{.75}^-$
$\mathbb{P}_6$	xxx	010	010	010	010	x10	0	0	3	0
$\mathbb{P}_5$	101	xxx	010	010	010	x10	0	0	2	0
$\mathbb{P}_4$	101	101	xxx	010	010	x10	0	0	1	0
$\mathbb{P}_3$	101	101	101	xxx	110	x10	0	0	0	0
$\mathbb{P}_2$	101	101	101	001	xxx	x11	0	1	0	0
$\mathbb{P}_1$	x01	x01	x01	x01	x00	xxx	N/A	N/A	N/A	1

Note that in general, strong clusters are only symptomatic of structure within the data. Once they have been identified, their memberships and relevance remain to be explored, which requires a good understanding of the application domain. One reason is that computed clusters tend to conform to the underlying assumptions of the clustering algorithm, e.g. spherical clusters for k-means and elongated clusters for single linkage.

2.3 A First Example: LAPACK

An example based on the LAPACK Library illustrates the use of Pareto ranking in extracting structure from dendograms. The LAPACK library is well known and studied, so this example also helps explain the use of Pareto ranking for mining performance data. Performance data was generated running the timing program *xlintimd* on a single processor of a dedicated SGI Octane with two 175 MHz R10K (IP30) CPUs. The source code for the program *xlintimd* can be found in the file named *dtimaa.f* inside the public domain distribution of LAPACK; it times FORTRAN77 subroutines for the BLAS, for orthogonal factorizations to triangular and Hessenberg form, for solving linear systems of equations $AX=B$ where A is a m by n matrix and B is a m by k matrix and where A can be a general or a positive definite matrix that is either dense, banded, triangular or symmetric. Packed storage schemes that avoid storing zero entries are available for specific types of matrices. All input options of *xlintimd* were set with matrix sizes ranging from 50 to 500 for m and n and from 1 to 100 for k, level-1 cache block sizes were 1 and multiple of 16 up to 64. Fortran stores matrices in column major order, that is consecutive elements in one column are stored contiguously in memory. The columns need not be stored contiguously however and the number of entries that could in principle be stored between any two contiguous entries in the same row (the *leading matrix dimension*) can be larger than the number of rows in the matrix. This leading dimension was set to 513 for all matrix arguments. This setup resulted in 3391 instrumented library calls for various sizes of arguments.

The R10K processor has two special *performance registers*, or *counters*, for keeping counts of the occurrence of anyone of 32 different hardware events. Each register has a designated set of 16 events that it can monitor. The PAPI interface maps those 32 events into 24 (virtual) hardware events. Each of calls in *xlintimd* to the LAPACK library was augmented by hand to measure consecutively the corresponding increments in the 24 hardware events counters

available through PAPI. The resulting data table was written to file just before *xlintimd* exited and it was transposed into a table of 24 profiles of length 3391. Each column of the table contains the counter increments for a distinct call site. Profiles were rescaled to have all their entries between 0 and 1. The data is slightly incomplete with eight values missing, due to counter overflows. The resulting dendogram is shown in Figure 3 where six strong clusters can be seen, labeled *C*, *C1*, *C2*, *O*, *O1* and *O1a*. Their labels reflect set inclusion. It is not possible to choose any one of them as the largest cluster in the final partition without making a judgment call on the relative importance of heterogeneity versus separation. Clusters *C* and *O* enclose all the other strong clusters. Cluster *C* is formed of profiles for various types of level 1 and 2 cache misses. Cluster *C1* consists of the profiles of the total number of level-2 data cache misses (L2_DCM) and the total number of level-2 cache data and instruction cache misses (L2_TCM), meaning that the number of instruction cache misses is either very low or proportional to that of data cache misses. The former explanation is more probable and it is reinforced in that the level1 and level-2 instruction data cache misses for a small cluster. The fifth element of cluster *C* is the total number of requests for exclusive access to a shared cache line (CA_SHR).

The total elapsed CPU cycles (TOT_CYC) belongs to the second largest strong cluster *O*. Conditional store instructions are the output equivalent of prefetching. When the instruction is issued, a reservation is placed on its destination address. The conditional store operation fails if the reservation is invalidated by another instruction. The fact that the profiles for the total number of conditional stores issued (CSR_TOT) and failed (CSR_FAL) form a strong cluster indicates that a fraction of all issued conditional store operations do fail. The exact fraction could be 10% or 90%; it is not possible to know this fraction by simply looking at *O1a* because profiles are rescaled before clustering. The other four profiles in *O1* are the number of requests for exclusive access to clean cache line (CA_CLN), the number of successful store conditional instructions (CSR_SUC), the average number of floating-point operations per second (FLOPS) and the average number of instructions executed per second (IPS). The latter average does not account for operations that were issued proactively but failed to execute, such as the failed conditional stores in CSR_FAL. Since the total number of conditional stores issued (CSR_TOT) is less closely related to the number of executed conditional stores (CSR_SUC) than to the number of failed conditional stores (CSR_FAL) it appears that the latter is not negligible. The remaining profiles in cluster O are the number of requests for cache line intervention (CA_ITV), the number of translation look-aside buffer misses (TLB_TL), the number of conditional branch instructions mispredicted (BR_MSP), the total number of instruction issued (TOT_IIS), the total number of branches tested (BR_INS) and the number of elapsed CPU cycles (CPU_CYC). It appears that the level-1 data cache is used effectively because it does not appear closely related to the number of elapsed CPU cycles (CPU_CYC). In that case, the total number of store operations for an n by n argument will be roughly $O(n^2)$. On the other hand, the number floating-point operations computed by many of the subroutines timed by *xlintimd* for the same argument is roughly $O(n^3)$. The closer association between CSR_SUC and FLOPS in *O1* than with CPU_CYC is in agreement with the relatively small matrix sizes used in the benchmark.

Four event profiles in Figure 3 are not included in any strong cluster. They are the total number of executed store instructions (SR_INS), the total number of floating point instructions executed (FP_INS), the total number of load instructions executed (LD_INS) and the total number of instructions executed (TOT_INS). None of these achieve a constant execution rate over all call sites for otherwise its profile would be closely related to that of the total number of elapsed CPU cycles. This fact would be hard to detect from simple inspection of the dendogram but it can be justified by the wide variety of algorithms included in the benchmark.

Figure 3 illustrates another advantage of looking at dendograms from the perspective of multi-objective optimization. The internal nodes of the dendogram in Figure 3 can be linearly ordered according to their distance from the left margin; that distance is indicative of the order in which the clusters were formed. A threshold rule, such as minimizing R in Subsection 2.2 is typically implemented by marching through the list of internal nodes say from the left, until the threshold condition is met. All visited internal that failed the test are then removed from the tree and the remaining sub-tree define the final partition. In particular, no such threshold rule would catch both of the largest strong clusters.

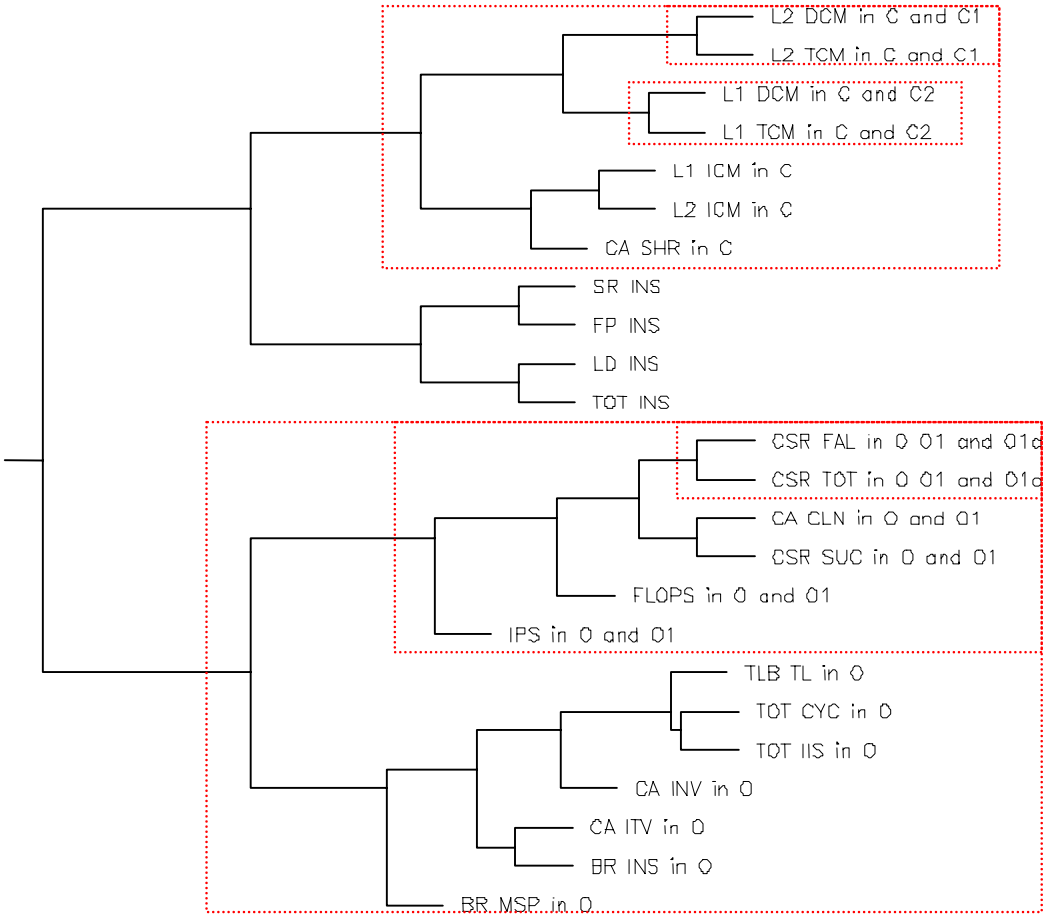

Figure 3 Annotated dendogram shows two clear clusters of hardware events as well as sub-cluster structure.

3. A Close Look at a Distributed Hash Table Library

Algorithms for indexing and sorting have been employed in numerical and data intensive applications since the very early days of computers. The best-known example of sequences of lists may be in numerical computing where sparse matrices are represented either as sequences of sparse vectors or as vertices and edges of adjacency or incidence graphs. There are several examples of hash tables used in high-performance computing[18, 31]. The Multipol library of distributed data structures from Katherine Yelick et al.[38] contains a general-purpose non-blocking distributed hash table. Several design criteria came into play for the hash table library described next. First, it was required that any application programmer wishing to use this library should be able to do so without adopting a specific memory or execution model. Second, it had to be lightweight enough to support fast implementations of distributed matrix-vector operations on sparse vectors that are needed for a fully-adjoint computation of some statistical estimation application.

The main functions of this communication library are HASH_Insert, HASH_Delete, HASH_Find and HASH_Wait. The first three functions initiate an appropriate active message and return before completion of the corresponding transaction. The same C Language calling sequence for HASH_Insert, HASH_Delete and HASH_Find, looks like this

ierr = HASH_Insert (*hash*, *tid*, *key*, *nval*, *arrayofvalues*, &*request*).

Here, *hash* is a handle to a hash table, and *tid* designates the target process. The third argument is the value of the key, and *nval* is the number of values transferred into the hash table on the target process. HASH_Delete and HASH_Find return up to the specified number *nval* of values to the origin process. These operations are complete when a matching call to HASH_Wait returns. The values transferred between processes are stored contiguously in the one-dimensional array *arrayofvalues*. HASH_Wait operates on an array of request handles and returns only after all the corresponding

operations are completed. In the present context, the *initiation time* is the time for a call to HASH_Insert, HASH_Find or HASH_Delete to return. The *completion time* includes the time spent in a subsequent matching call to HASH_Wait in addition to the initiation time. The *barrier time* includes also the time needed for processes to synchronize at a barrier immediately after the call to HASH_Wait returns. Different strategies are actually required for reducing each one of these three runtimes. The LAPI execution model for active messages consists of three types of threads for each process. A user thread carries the application and calls the LAPI library. A unique header thread assembles the content of the active message in a buffer allocated by a user-defined header handler. Finally, a single completion thread calls a user defined completion handler that processes the assembled message. Under this specific LAPI implementation, there can be only one active header handler at any time, and similarly with completion handlers. The broad-brush description of how remote accesses are implemented is that the source process initiates an active message toward the appropriate target process. When the appropriate completion handler is called, it locks the appropriate local hash table, performs that requested operations and returns the result by a one-sided put operation. There are effectively three times when a remote access can stall at the target process: waiting for the message body to arrive in the header thread, waiting to lock the local hash table in the completion thread and waiting for completion of the put operation before releasing unneeded storage. The third cause is a major one, so it is handled in the user thread using a simple queue. The active completion handler is forced to wait when that queue is full, but it returns early otherwise.

The second datasets considered in this paper consists of hardware event counter measurements for two hundred randomized micro-benchmarks on an IBM SP with 4-Way 375MHz Power III nodes. Keys and values were long integers. A global unsorted sequence of one million unique keys between 0 and m=7,000,000 was generated. PAPI measurements were done in batch (and dedicated) mode. The same sequence of table accesses was repeated for each of the 44 hardware events measurable through PAPI on this IBM SP. One column of data was produced for each type of table accesses by each process and for each run. Each of the three wall-clock times corresponds to a separate PAPI measurement; the final dataset is a 3x44=132 by 14268 table, excluding record and feature tags. The number of keys per process was fixed at 10,000. For each run, a single array length *nval* was randomly selected between 1 and 8 inclusively for all calls to HASH_Insert, HASH_Find and HASH_Delete, see the calling sequence for HASH_Insert on page 10. Within each run, all but the last call to HASH_Wait received the same number of pending requests. That number was a random multiple of ten ranging between 10 and 200. The number of computer nodes varied from 1 to 11 with each of the four processors one a node running one process. The number p of processors is four times the number of nodes. Each micro-benchmark implements a randomized all-to-all scenario where the i^{th} process is both the target for all operations involving keys whose residue modulo p is i and the origin for all table accesses for the keys $m/p*i$ through $m/p*i+i-1$ in the sequence. Runtime averages are taken over the number of values inserted by each *origin* process. The mapping from hash key to processor, which is not defined by the API, is that all values for the key k are mapped to process number $k\%p$.

Each hardware event profile, one row in the data table, was scaled before clustering so that its entries range between 0 and 1. Again, the object is to identify controllable hardware events that are closely related to measures of elapsed CPU cycles. Figure 4 shows two separate dendograms from clusterings separately all profiles from the initiation phase (_I suffix for immediate) and from the completion phase (_B suffix for blocking) respectively. Strong clusters are outlined in the dendograms. An important point can be made from Figure 4: the two computational phases correspond to different fine scale structures. Precisely, during the initiation phase of hash table accesses, the profile of the elapsed CPU cycles (TOT_CYC_I in Figure 4) is most similar to that of the number of load-store instructions completed (LST_INS_I) and to that of the number of synchronization operations completed (SYC_INS_I). The middle cluster in the top dendogram shows that the profile of instruction-cache misses (L1_ICM_I) is mostly related to the profile of requests for exclusive access to shared cache lines (CA_SHR_I). The implication is that the initiation time is dominated by requests to process active messages. The dendogram in the lower part of Figure 4 shows a different picture for the completion phase, with the profile of the total CPU cycles mostly related to the counts of issued and completed instructions. The impact of incoming active messages can still be seen in the lower dendogram.

Data clustering can only point at interesting connections between hardware events or between library calls. Once an observation has been made, it remains to be explained from the applications perspective and it should be validated. The observation that different phases correspond to different performance regimes is confirmed by looking at the strong clusters of another dataset, which includes all possible 132 profiles. The resulting dendogram is partitioned into three strong clusters but is not shown here for reasons of space. A first cluster contains all and only all profiles for the initiation phase. A second cluster contains all but six profiles from the completion phase and no barrier profiles. A third strong cluster contains all profiles for the barrier phase plus the latter six completion profiles. Those profiles were

encountered earlier during the presentation of hierarchical clustering; they do not form any strong sub-cluster on their own within the larger context.

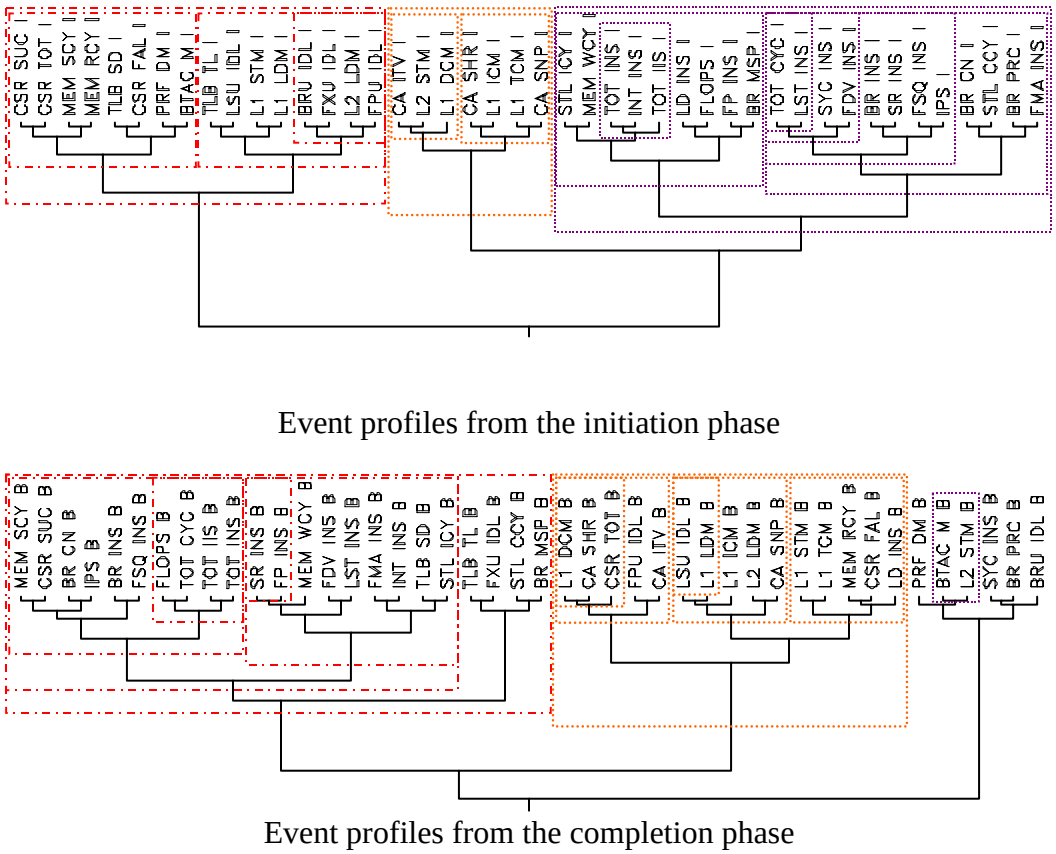

Figure 4 Associations among hardware event readings differ between the initiation and completion phases.

The suggestion that incoming messages impact the initiation time is verified indirectly. The data from the 200 randomized micro-benchmarks is binned according to one software parameters that influences the number of active messages issued, that is the block size called *nval* in the function prototype above. A Krukal-Wallis test is able to show with a high degree of confidence that the variation across processes for the ratio of the initiation (and completion) time to the number of values processed has different statistical distributions across these bins. The variation is defined here as the ratio of the maximum value (across all processors) to the mean value (across all processors), see [24] for details. Better non-parametric tests would be needed to demonstrate the existence of a trend in this variation as a function of *nval*. Validation from the application is given by the observation that the "optimal" block size depends on the length of the queue of pending put operations. In a sense, this last queue operates as an anti-cache; it would be interesting to see how well it might work if implemented in hardware.

A third way to assess the hypothesis that incoming active messages impact the initiation time is to look at the data from the other end. Pareto ranking can be applied to call sites instead of profiles. The Goldberg ranks of all 14268 data columns (library calls) were computed with respect to the two events SYC_INS_I and CA_ITV_I (requests for cache line intervention). These two profiles were chosen because they appear closely related to elapsed CPU cycles during the initiation phase. The computed ranks ranged from 0 to 228. This represents a coarse quantization of the data since the actual values for these two events are different for almost every data record. It was achieved without setting an arbitrary threshold value. It can be seen in Figure 5 that library calls that have a low Goldberg rank, with respect to SYC_INC_I and CA_ITV_I, have relatively uniform initiation and completion wall-clock times. Spikes in either of those wall-clock

times appear associated with higher Goldberg ranks. Figure 5 also illustrates the performance of this hash table library. For comparison, repeated measures of the one-sided point-to-point latency for the IBM's proprietary MPI library on the same IBM SP ranged between 25μs and 34 μs.

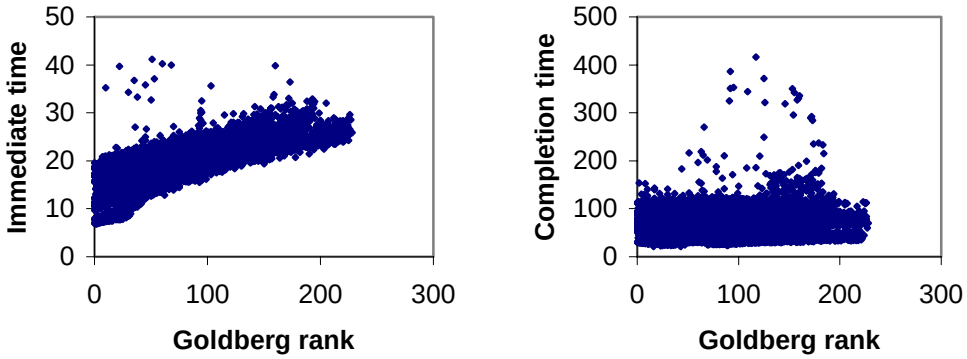

Figure 5 Scatter plots of wall clock times for the initiation and completion phases for 200 randomized micro-benchmarks versus Goldberg ranks based on the SYC_INC_I and CA_ITV_I event profiles.

Call sites that involve larger numbers of processors also correspond to somewhat larger Pareto ranks, so the possibility of sampling bias (over and under sampling) is real. Figure 6 that shows scatter plots of the completion and initiation times versus the Goldberg rank for four groups of library calls arranged by numbers of processors. In all four plots, the initiation wall clock time can be seen as a line at the bottom of the plot that increases slowly with the rank. These graphs suggest that hardware that would improve the rate of completion of synchronization operations and speedup requests for cache line intervention is also likely to result in a more uniform initiation and completion time for this communication library.

4. Conclusions and Further Work

Some of the clustering and data mining techniques discussed here are computationally expensive, e.g. hierarchical clustering and Goldberg ranking. To give an order of magnitude of the work involved, each of the dendograms in Figure 4 is computed serially by a java (non-native) code in about 26 seconds on a 3GHz Xeon processor: 1.7 seconds for the initial minimum spanning (MST) tree needed to start the hierarchical clustering, 18 more seconds for the complete linkage variant and the rest for computing Goldberg ranks and writing the results to an ASCII file. On the other hand, the transpose table takes 14 minutes for the MST, 72 minutes for hierarchical clustering and 78 minutes for Goldberg ranking. Parallel Pareto ranking algorithms are easy to write in MPI at least and experience suggests that they scale well in practice. Hierarchical clustering of very large data sets is a key problem in other areas such as bio-informatics and atmospheric sciences. This paper focus on complete-linkage clustering for conciseness sake but the multi-objective concepts discussed are orthogonal to the way clusters are computed; the successive partitions generated by k-means could be used instead. Another possible use for Pareto ranking is in data compression; when 14268 call sites of the second data set are ranked according to all 132 pairs of hardware events and computation phases, only 799 of them turn out to be non-dominated.

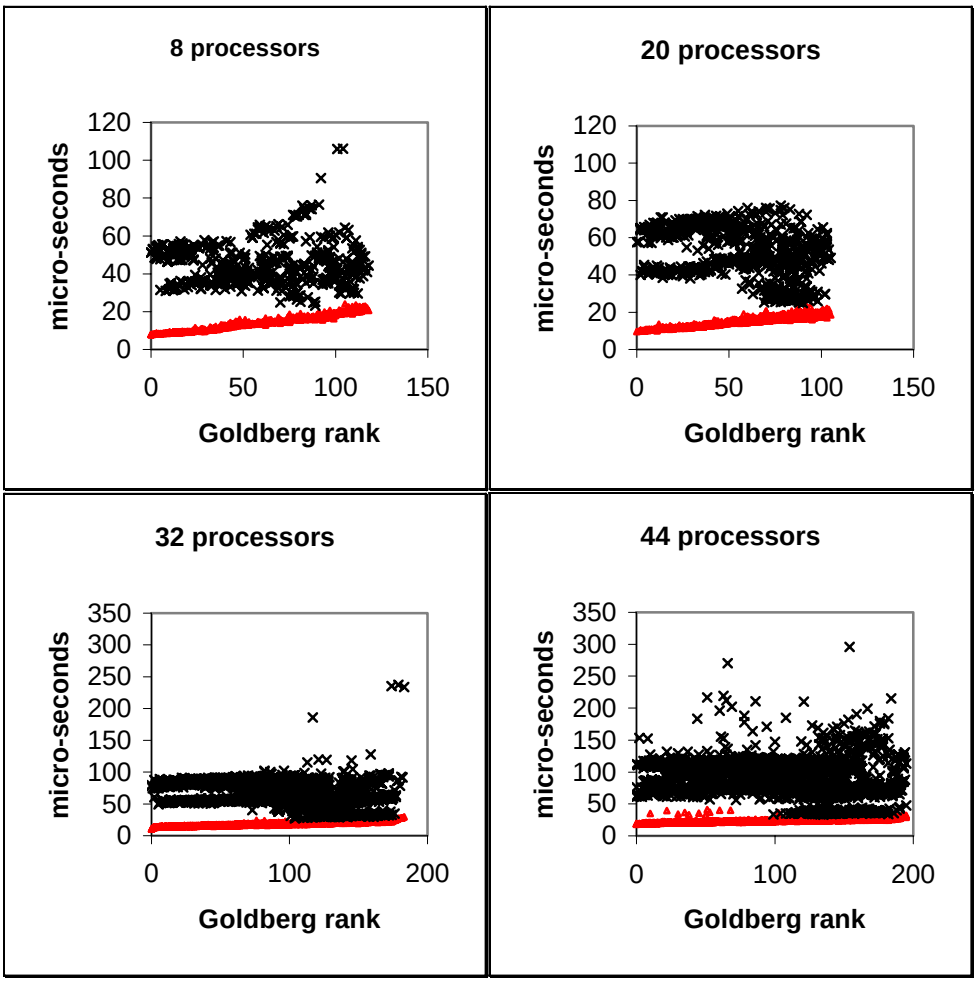

Figure 6 Scatter plot of the initiation (Δ) and completion (+) times of library calls versus Goldberg ranks for various numbers of processors.

The project that led to the work described here was tightly focus on the development of efficient middleware for managing distributed dynamic data structures. Data mining efforts were aimed at answering specific questions relevant to this middleware rather than on software tool development. With this context in mind, data collections and summary tools such as Photon MPI and WatchTower would have been very useful to have then. Data collection code was inserted by hand and summarization was done at the end of each benchmark by each process. Randomized micro-benchmarks ran in batch mode and most were submitted in succession to avoid disabling the batch scheduler. Batch requests were generated by a small C program. Data tables were assembled using shell scripts whenever possible, although a C program was written for computing the transpose of the larger table because some utilities silently truncated all columns to the right of some internal limit. Surprisingly, one of the most troublesome aspects of this work was related to editing drafts on an off-the-shelf laptop. The sizes of those drafts varied between 15 and 25 Mbytes. Some serious editing sessions required that swap space be defragmented and the hard disk cleaned up. Another observation may be helpful to the reader. Non-parametric statistical tests such as Kruskal-Wallis are based on some ranking of the data with appropriate tie-braking rules. On the other hand, performance data from hardware event counters, or instruction profilers, is read out of 32-bit or 64-bit registers. The low order bits can *"confuse"* a non-parametric test by preventing ties to occur where they could or should. It helps in this respect to repeat the tests for data rounded or truncated to the same units, for example with 1.9876 and 10.777 first rounded to 2.0 and 10.7 and then to 1.99 and 10.78.

The use of data caches and arithmetic blocking factors in dense matrix algebra pushes the envelope of achievable computations along *one* dimension of computer architecture, raw arithmetic speed. The performance of data intensive

applications is driven by other factors such as the speed of branching, latency of remote memory accesses, and the need for very high numerical accuracy. It is likely that the hardware advances that will enable adequate sustainable performance of data intensive applications on Teraflop and Petaflop architectures will be the result from systematic studies based on real performance data. This work on communication middleware using micro benchmarks suggests that novel statistical tools and techniques will be needed to carry such studies.

Software performance data is typically longitudinal. Missing values do occur but they are rarely missing at random. Data records are typically dependant due to contentions for network or CPU resources. Modestly complex algorithms such as parallel dense matrix factorization algorithms can result in huge datasets. Because software performance datasets are challenging from a statistical perspective, novel techniques are needed to sort out which software parameters can most effectively drive the performance of communication software libraries. Pareto ranking of dendogram nodes and their associated clusters is shown to be an effective technique for detecting sub-cluster structures. Pareto ranking of data records is also shown to be an effective technique for identifying subset of the data with good runtime properties.

Acknowledgments

This research was performed in part using the Molecular Science Computing Facility (MSCF) in the William R. Wiley Environmental Molecular Sciences Laboratory, a national scientific user facility sponsored by the U.S. Department of Energy's Office of Biological and Environmental Research and located at the Pacific Northwest National Laboratory. Pacific Northwest is operated for the Department of Energy by Battelle.

This manuscript has been authored by Battelle Memorial Institute, Pacific Northwest Division, under contracts No. DE-AC06-76RLO 1830 with the US Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or to allow others to do so, for United States Government purposes.

Bibliography

[1] Ahn D.H. and J.S. Vetter, “Scalable Analysis Techniques for Microprocessor Performance Counter Metrics,” Proc. SC 2002, 2002.

[2] Anderson E., Bai Z., Bischof C., Demmel J., Dongarra J.J., Du Croz J., Greenbaum A., Hammarling S., McKenney A., Ostrouchov S. and Sorensen D., "LAPACK Users' Guide", SIAM Press, Philadelphia, 1994

[3] Bilmes J., Asanonic K., Chin C-W. and Demmel J., "The PHiPAC v1.0 Matrix-Multiply Distribution", University of California at Berkeley, Computer Science Division, Technical Report UCB/CSD 98-1020, 1998

[4] Blackford L. S., Demmel J., Dongarra J.J., Duff I., Hammarling S., Henry G., Heroux M., Kaufman L., Lumsdaine A., Petitet A., Pozo R., Remington K., Whaley R.C., "An Updated Set of Basic Linear Algebra Subprograms (BLAS)", submitted to ACM Transactions on Mathematical Software, February, 2001

[5] Browne S., Dongarra J.J., Garner N., Ho G. and Mucci P., "A Portable Programming Interface for Performance Evaluation on Modern Processors", The International Journal of High Performance Computing Applications, 14(3), pp 189-204, 2000

[6] Carissimi A. and Pasin M., “Athapascan: An experience on Mixing MPI Communications and Threads", Proceedings of 5th European PVM/MPI Users' Group Meeting, Alexandreov, V. and Dongarra, J.J (eds), Liverpool UK, Springer Verlag Lecture Notes in Computer Science 1497, pp 137-44, 1998

[7] Choi J., Dongarra J.J., Pozo R. and Walker D.W., “LAPACK Working Note 55: ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers", U. of Tennessee, Knoxville TN, CS-92-181, 1992

[8] Clement M.J. and Quinn M.J., "Automated performance prediction for scalable parallel computing", Parallel Computing, 23(10), 1405-20, 1997

[9] Cociorva D., Baumgartner G., Lam C., Sadayappan P., Ramanujam J., “*Memory-Constrained Communication Minimization for a Class of Array Computations*”, in Proceedings of the 15th International Workshop on Languages and Compilers for Parallel Computing (LCPC '02), College Park, Maryland, July 2002.

[10] Culler D., Singh J.P. and Gupta A., "Parallel Computer Architecture A Hardware/Software Approach", Morgan Kaufmann, 1997.

[11] Deb K., "Multi-Objective Optimization using Evolutionary Algorithms", John Wesley & Sons, Systems and Optimization Series, Chichester UK, 2001

[12] Fonseca C.M. and Fleming P.J.: “Genetic algorithms for multi-objective optimization: Formulation, discussion and generalization", in Genetic Algorithms: Proceedings of the Fifth International Conference, Forrest S. (ed.), Morgan Kaufmann, San Mateo, CA, pp416-23, 1993.

[13] Frigo M. and Johnson S.G., "FFTW: An Adaptive Software Architecture for the FFT", in proceedings of the 1998 ICASSP Conference, vol. 3, 1381-1384, 1998.

[14] Gallivan K., Jalby W. and Meier U., "The use of BLAS3 in linear algebra on parallel processor with a hierarchical memory", SIAM Journal on Scientific and Statistical Computing, 8(6), pp 1079-84, November, 1987

[15] Geist A., Gropp W., Huss-Lederman S., Lumsdaine A., Lusk E., Saphir W., Skjellum T., Snir M., MPI-2: Extending the message-passing interface, Argonne National Laboratory, Technical Report MCS-P568-0296, 1996

[16] Goldberg, D.E.: “Genetic Algorithms” in Search, Optimization and Machine Learning, Addison-Wesley, Reading, MA, 1989.

[17] Gordon A.D., “Classification”, Monographs on Statistics and Applied Probability 82, 2nd edition, Chapman & Hall/CRC, Boca Raton, 1999

[18] Griebel M. and Zumbusch G., “Hash-Storage Techniques for Adaptive Multilevel Solvers and Their Domain Decomposition Parallelization” Contemporary Mathematics, vol. 218, pp271-8, 1998.

[19] Heller D., "Rabbit: A performance counters library for Intel/AMD processors and Linux", URL: http://www.scl.ameslab.gov/Projects/Rabbit/

[20] Hendrickson B. A. and Womble D. E., "The Torus-Wrap Mapping for Dense Matrix Calculations on Massively Parallel Computers", SIAM Journal on Scientific and Statistical Computing, Sept., 1994

[21] Knop M.W., Schopf J.M., and Dinda P.A., "Windows Performance Monitoring and Data Reduction Using WatchTower", Proceedings of workshop on Self-Healing, Adaptive and Self-Managed Systems, 2002

[22] Lumetta S.S., Mainwaring A.M. and Culler D.E, "Multi-protocol Active messages on a cluster of SMP's", Supercomputing 1997, November, 1997

[23] Maki J.,"POWER2 Hardware Performance Monitor Tools", URL: http://www.csc.fi/~jmaki/rs2hpm-paper, November 1995

[24] Malard J.M. and Stewart R.D., “Distributed dynamic hash tables using IBM LAPI”, submitted to Supercomputing 2002

[25] Malard J.M., “Parallel Restricted Maximum Likelihood for Linear Models with a Dense Exogenous Matrix”, Parallel Computing, 28, pp343-53, 2002

[26] Malard J.M. and Paige C.C., “Data Replication in Dense Matrix Factorization", Parallel Processing Letters, 3(4), 1994

[27] Massari L. and Mahéo Y., "Performance Analysis of Automatically Generated Data-Parallel Programs", *in 4th Euromicro Workshop on Parallel and Distributed Processing*, Braga, Portugal, 1996.

[28] Milligan G.W. and Cooper M.C., “An examination of procedures for determining the number of clusters in a data set”, Psychometrika, 50, pp 159-79, 1985

[29] Navarro A.G., Paek Y., Zapata E.L., Padua D., “*Compiler Techniques for Effective Communications on Distributed Memory Multiprocessors”,* in International Conference on Parallel Processing, August 1997.

[30] Nichols B., Buttlar D., Proulx-Farrell J. and Farrell J., “Pthreads Programming: A POSIX Standard for Better Multiprocessing”, O'Reilly Nutshell Series, 1996

[31] M. Parashar and J. C. Browne, “System Engineering for High Performance Computing Software: The HDDA/DAGH Infrastructure for Implementation of Parallel Structured Adaptive Mesh Refinement,” IMA Volume 117: Structured Adaptive Mesh Refinement (SAMR) Grid Methods, Editors: S. B. Baden, N. P. Chrisochoides, D. B. Gannon, and M. L. Norman, Springer-Verlag, pp. 1 – 18, January 2000.

[32] Püschel M., Singer B., Xiong J., Moura J.M.F., Johnson J., Padua D., Veloso M.M., and Johnson R.W., "SPIRAL: A Generator for Platform-Adapted Libraries of Signal Processing Algorithms", Journal of High Performance Computing and Applications, *accepted for publication.*

[33] Shah G., Nieplocha J., Mirza C., Harrison R., Govindaraju R.K., Gildea K., DiNicola P. and Bender, C., Performance and experience with LAPI: a new high-performance communication library for the {IBM} RS/6000 SP", Proceedings of the International Parallel Processing Symposium, pp 260-66, 1998.

[34] Vadhiyar S.S., Fagg G.E. and Dongarra J.J., "Performance Modeling for Self Adapting Collective Communications for MPI", LACSI Symposium 2001, Santa Fe NM, October 2001

[35] Vetter J.S., *Dynamic Statistical Profiling of Communication Activity in Distributed Applications*. Proc. SIGMETRICS: International Conf. Measurement and Modeling of Computer Systems, 2002

[36] Vetter J.S. and Mueller F., “Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures,” Proc. International Parallel and Distributed Processing Symposium (IPDPS), 2002.

[37] Whaley W., Petitet A. and Dongarra, J.J., "Automated Empirical Optimization of Software and the ATLAS Project", University of Tennessee, Knoxville, UT-CS-00-449, 2000

[38] Yelick K., Chakrabarti S., Deprit E., Jones J., Krishnamurthy A., and Wen C. “Parallel Data Structures for Symbolic Computation”, Workshop on Parallel Symbolic Languages and Systems, October 1995.