Parallel Research Kernels (PRK)

Tim Mattson, Rob Van der Wijngaart, Jeff Hammond, Srinivas Sridharan
Intel Labs

Introduction

Experienced parallel programmers know what they want from a parallel system. They know how certain operations should perform and how overheads should scale for commonly encountered parallel operations. In many cases, programmers can relate expected performance trends to one or more basic operations; allowing these operations to serve as proxies for full applications.

We have collected a set of common low level operations into a research tool we call the *Parallel Research Kernels* (PRK). The PRK are a test suite to communicate the expectations of parallel application programmers to system designers, which includes hardware, compilers, runtimes, and programming models. They do not define a standard problem set or generate consistent performance numbers to rank different systems. In particular, problem sizes, numbers of iterations, and certain other kernel properties are arbitrary and can be supplied by the user. In addition, a number of the kernel reference implementations contain different algorithms for solving the same problem that may trade memory use for absolute performance, different synchronization methods, etc. Hence, while some of the kernels are based on well-known benchmarks, the test suite itself is not a benchmark.

The PRK suite is defined as a paper-and-pencil set of operations largely independent of implementation. It is supplemented with reference implementations in C (for portability), some C++ where appropriate, and some Fortran, for execution as a serial baseline case, and in parallel for shared memory systems (e.g. multicore processors) and distributed memory systems (e.g. clusters) and combinations thereof. The reference implementations are an expanding set, as new parallel runtimes are being added to the suite.

An important design feature of the PRK is verifiability. All kernels are constructed such that the final answer is known analytically, and also aim to guarantee that any non-trivial error caused by incorrect implementation, faulty runtime, faulty compiler, or faulty hardware will lead to an observable error in the final answer. This requirement is among the most difficult to satisfy in the PRK, because of the fact that they are arbitrarily scalable. It leads to specific choices for initialization and other details of the kernel definition and implementation. For example, since the kernels are arbitrarily scalable, they must always be executable in repeated fashion to make sure the total time spent is not dominated by short-lived system noise. But an error in any of these repetitions must be exposed in the final verification test, which means that the result of one iteration must influence the result of the next.

It might be possible to verify intermediate results instead of only the final results, but that would violate another PRK design goal, namely the fact that the verification should not influence the system property being evaluated. Hence, verification should not happen until all relevant kernel operations have finished, and any accommodations made for

verification should be minimally intrusive in terms of *fundamental* overhead, though in certain runtimes the actual overhead may be substantial.

A third PRK design goal is that the verification should not be more costly in terms of memory and execution time than the actual kernel itself, and preferable much less costly. The reason is that the kernels are often scaled up and run in large batches, and hence consume substantial resources on expensive systems. Hence, all non-essential overheads should be minimized.

This also leads to a fourth design goal. Initialization, which is generally not timed[1], should be scalable (parallel formulations). This initialization, in turn, must be completely repeatable. That is, different executions, with potentially different system resources (numbers of cores/nodes) applied, should lead to identical initializations. The reason for this is that different initializations may influence observed performance, and hence add undesirable noise to the computational experiments.

Verifiability in the face of full parameterization also leads to two more requirements. First, the computations should never converge to an unchanging solution (fixed point), because it means that the computation could be stopped prematurely without affecting correctness. Second, only stable workloads should be chosen, i.e. solutions that grow at most polynomially in time, or the program could start experiencing arithmetic overflow.

A less concrete but no less important requirement for the PRK is that they do not require domain knowledge beyond high school physics or math. This is critical for their utility, because their behavior should be translatable to that of constructs in various disciplines far removed from the workloads that inspired them.

Finally, all kernels must be formulated such that they contain arbitrarily scalable amounts of exploitable concurrency—though that actual exploitation may be quite non-trivial, depending on the runtime or programming model used. The reason is that a fundamental load imbalance is not a system property, and hence does not contribute toward the PRK goals.

Reference implementations

The coding style we employ for the reference implementations is one of reasonable—and reasonably portable—efficiency, but without system-specific optimizations: no assembly language, no specialized math library functions (with the exception of DGEMM and intrinsic functions like sin, cos, sqrt, exp), no knowledge of the number of levels, sizes, or associativities of caches, no extreme, non-obvious data reorderings (Sparse uses traditional Compressed Row Storage, which is simple but known to be suboptimal), etc. This is sometimes referred to as non-ninja-style coding.

Timing is done by measuring the time it takes to complete all repetitions of the pattern under investigation and then dividing by the number of repetitions to arrive at the average time per execution of the pattern. We exclude the very first iteration to eliminate noise

[1] Initialization is not timed if it happens before the actual operations of interest to the specific PRK are executed. If an accelerator device with separate memory is used (e.g. OpenMP, OpenCL, CUDA with offload) that cannot hold the entire data set, initialization (and, more importantly, reloading) of data in multiple accesses, interspersed with actual operations, would be an inevitable part of the application and is thus included in the timing.

from initial paging, populating caches, training prefetchers, etc. In practical terms, a timer is started after the first iteration, and preceded by a semantic barrier to make sure all cores are synchronized before the measurements start. The timer is stopped after all repetitions have finished completely and the cores have reached a semantic barrier following the repetitions. Performance is computed by dividing work per iteration by the average time per iteration. The implication of this timing method is that barriers are not required between repetitions, but only at the start and end of time measurements. Hence, if the programming model or runtime used makes it difficult or impossible to avoid barriers between iterations, performance will typically suffer due to redundant synchronization.

PRK summary

The set of parallel research kernels includes:

1. Dense matrix transpose
2. Vector reduction; regular memory access on read/write.
3. Sparse matrix-vector multiplication; irregular memory access on read only
4. Random access update; irregular memory access on both read and write
5. Synchronization; global and point-to-point
6. Stencil; regular strided memory access
7. Atomic reference counting; shared and private.
8. Scaled vector addition (Stream* Triad)
9. Dense matrix-matrix multiplication (DGEMM)
10. Branching Bonanza
11. Particle In Cell (PIC)

Runtimes

The standard set of operational modes or runtimes used to implement the kernels is Serial, MPI with two-sided communications, and OpenMP. All kernels except atomic reference counting are also implemented in Fine-Grain MPI and Adaptive MPI. Of the most important kernels, Stencil, Synchronization/point-to-point, and Transpose we also provide implementations in MPI with OpenMP, MPI with MPI3 shared memory, MPI with one-sided communications, Grappa, Charm++, OpenSHMEM, UPC, and Coarray Fortran, while new implementations in HPX-3, HPX-5, Chapel, OCR and Legion are forthcoming.

Branching Bonanza is formulated only as a local (i.e. non-distributed, serial) or simply replicated task. No additional insight is expected from this kernel when implemented using parallel algorithms, except possibly with regard to shared caches.

Performance expectations

We provide high-level cost models for the performance of the PRK on a parallel computer. These models, which we call *performance expectations*, are one of the most important parts of this effort, since without them there is no way of knowing if observed performance reflects hardware issues, compiler or runtime defects, or artifacts of the implementations of the kernels.

A performance expectation codifies what an experienced parallel programmer expects from a good, real parallel system with finite resources. It ignores details of micro-architecture and of the particular software environment. Hence, the performance expectation reflects what performance of a kernel should be in the absence of low-level design errors or imperfections. The performance expectations assume either a single multi-core chip with attached off-chip memory, or a cluster of single-core processors, each with its own (private) memory and memory channel. In either case we leave the topology of the interconnection network unspecified. The algorithms we present are formulated such that they can perform reasonably well, in principle, even on a simple linear array or ring topology. For performance expectations we will not use any topological information, but only point-to-point and bi-section bandwidth.

Definitions and Assumptions:

We use the symbols α and β for latency and bandwidth, respectively. Aggregate (bi-section) bandwidth is $\underline{\beta}$. Subscripts "M" and "$" refer to memory and cache (line), respectively. Subscripts "N" and "C" refer to (cluster) Node(s) and Chip, respectively. "L" stands for Length with a subscript to define the specific context.

Symbol	Meaning	units
β_M	Memory bandwidth (one core or one cpu to memory)	Bytes/sec
α_M	Memory latency	sec
α_N	cluster message latency	sec
β_N	cluster point-to-point message bandwidth	Bytes/sec
$\underline{\beta}_N$	cluster aggregate (bi-section) message bandwidth	Bytes/sec
$\underline{\beta}_{CM}$	on-chip aggregate (bi-section) memory bandwidth	Bytes/sec
$\underline{\beta}_{C\$}$	on-chip cache-to-cache aggregate (bi-section) bandwidth	Bytes/sec
L_W	word size	Bytes
$L_\$$	cache line length	B Bytes
P	number of cores/cluster nodes	
FP	peak scalar/vector floating point performance (floating point multiply adds per sec) per core/node	1/ sec
IP	peak scalar/vector integer performance (instructions/s) per node/core	1/ sec

We note that β_N may be of the same order as β_M, depending on the system configuration and implementation. However, α_N will normally be much larger than α_M, even in the case of shared memory systems, due to software overheads. We assume that on-chip cache-to-cache transfers are much faster than accesses to off-chip memory.

We assume that the processor cores feature fused multiply-add floating point pipelines, so that maximum floating point speed is only reached for computations that have a balanced mix of multiplications and additions.

We ignore latency and bandwidth between the various levels of cache and registers within a specific node or core; these are considered irrelevant compared to traffic between memory and cache.

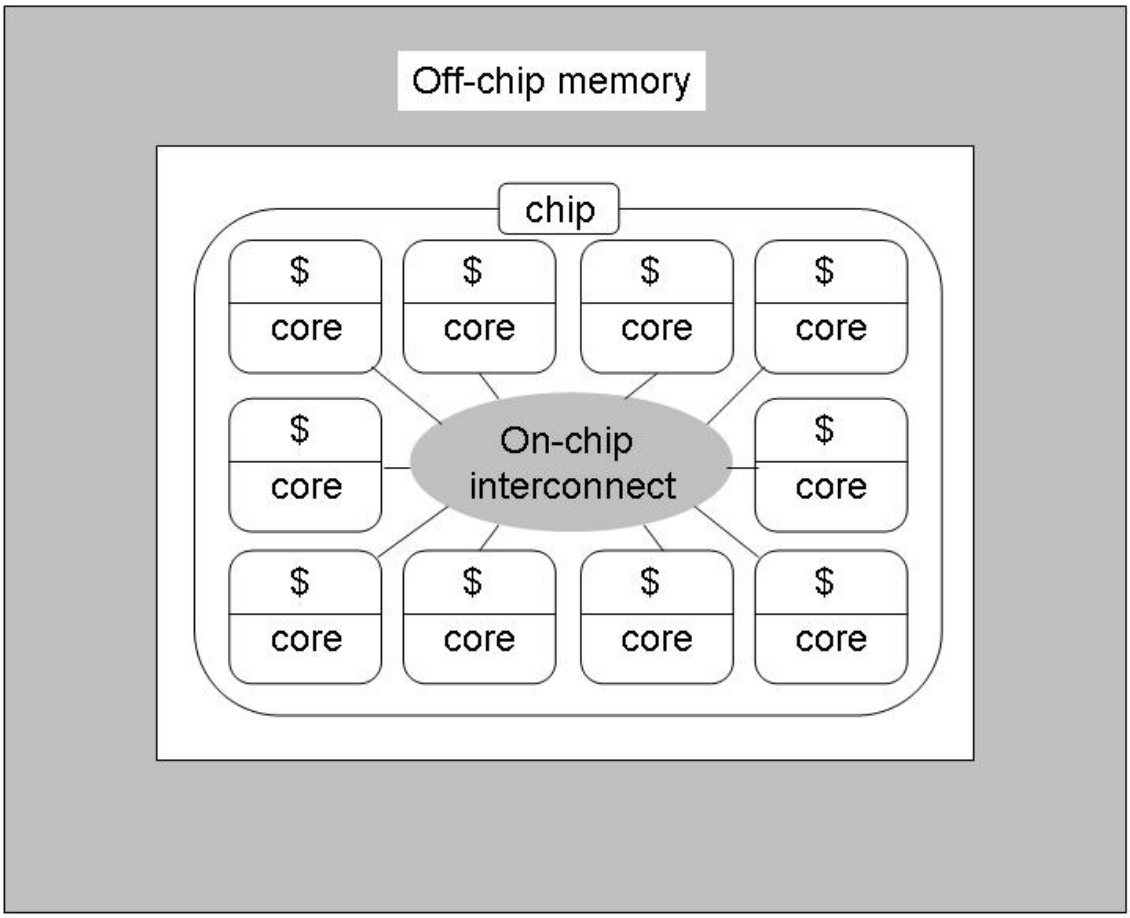

Figure 1. Schematic of multi-core chip. We only display one level of on-chip cache ($). It is shown as physically distributed, i.e. a tile architecture. The on-chip interconnect is left unspecified, as is the mechanism to access the off-chip memory, which maybe shared, private, or a mix of the two.

While the implementations of the parallel research kernels are all completely parameterized and thus allow us to investigate different stress points of the computer system, the performance expectations will only be provided for "extreme" cases, where one extreme is defined as so large that the problem exceeds reasonable on-chip resources (specifically, caches), and the other as sufficiently small that no off-chip resources are required (in that case effects such as latency usually become important). We use subscripts "L" and "S" for large and small cases, respectively. Where applicable, caches will always be assumed to be hot.

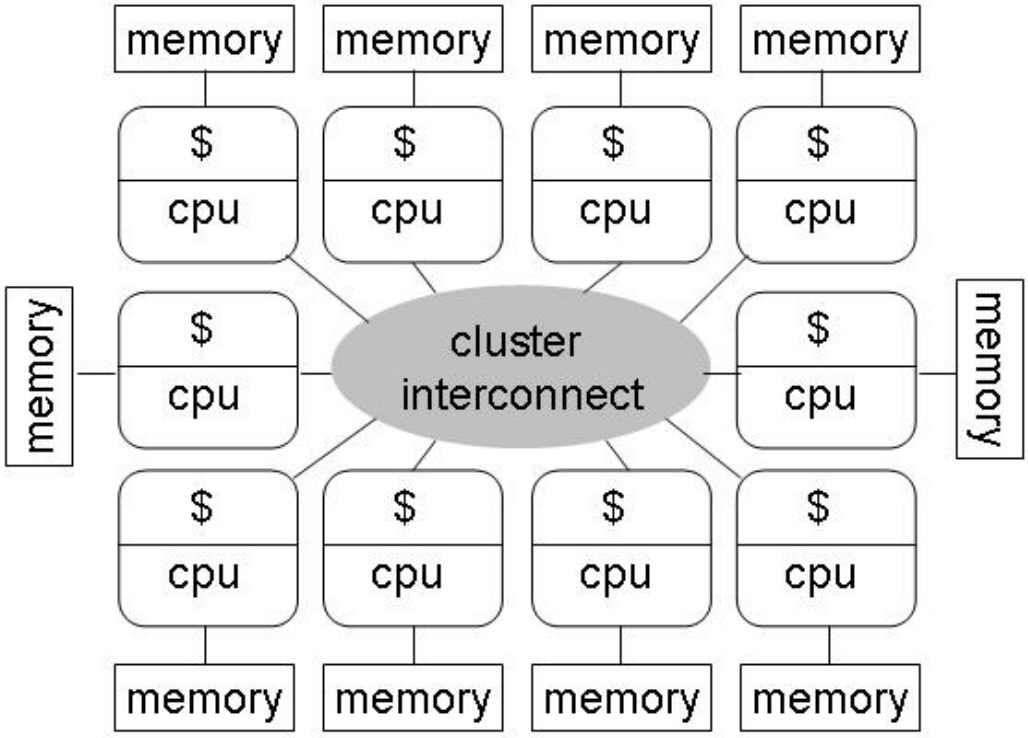

Figure 2. Schematic of cluster. We only display one level of on-chip cache ($). Each cpu has access to off-chip private memory. The cluster interconnect is unspecified.

We do not aim to cover parallel programming APIs (e.g. MPI, OpenMP) comprehensively. Instead, we focus on a set of commonly used operations whose efficiency significantly affects application performance. The set is chosen such that its performance depends comprehensively on the performance of the underlying hardware. In other words, if the set performs well, it can be assumed that all other significant operations will perform equally well, provided correct software implementation and correct, efficient compilers. For example, efficient implementations of OpenMP locks rely on the same hardware as do critical sections, so we only include one of the two in our reference implementations.

Whenever we refer to threads in this report, it will be understood that this may also refer to MPI *ranks*, Grappa *cores*, Charm++ *chares*, Legion *localities*, etc. We assume that a single thread will run on a logical core.

Although we attempt to include all first order effects on performance in the expectations, we exclude some that usually have an influence on current systems, but that are not fundamental:

- Memory consistency traffic. A write operation may invoke a prior read of the cache line involved to ensure that the entire cache line will be up to date when the write operation completes, which could potentially double the memory bandwidth requirement for the variables involved. This is not a strict necessity, especially not if the application (almost) always updates entire cache lines before writing them back, so we ignore this effect.

- Network congestion. This is incorporated, in principle, in the bi-section bandwidth, so we do not model it explicitly.

The Parallel Research Kernels: Specifications + performance expectations

Transpose

Name: TRANSPOSE

Description: A square matrix A(nxn), decomposed by columns among the threads, is written to another matrix B, also decomposed by columns among the threads, that is the transpose of the original. The problem size is parameterized, such that the smallest matrix fits entirely in the lowest level caches, up to a size that does not fit inside any reasonably sized last level cache. This scale-up will reveal the transition from performance dominated by the interconnect on a chip to performance dominated by memory bandwidth for the data exchange between threads. The transposition of data within each thread will also reveal the range of local memory bandwidths and latencies.

To satisfy the verifiability requirement we modify the original formulation slightly. The matrix A is changed trivially after each transpose operation, so that never the same matrix is transposed. In addition the transpose is added to matrix B, not stored in its stead. In pseudo-notation: ∀i,j: b(j,i) += a(i,j); a(i,j)++ This can be made to follow the rules of non-intrusiveness of verification support, since adding one to each input matrix element at the time it is referenced does not add noticeable memory system overhead.

Initialization: Matrix element A(i,j) is initialized with i+n*j. B is initialized to all zeroes.

Verification: At the end of K iterations matrix element B(i,j) has the value (n*i+j)*K + K*(K-1)/2.

Usage: This kernel is used in multi-channel digital signal processing algorithms, single and multi-dimensional FFT algorithms, and other applications that feature strong, directionally biased data dependencies that change in the course of the computation.

Performance expectation:

All reasonable implementations of dense matrix transposition will tile the transpose to improve cache usage and limit TLB misses, except if the matrix is so small that it fits in L1.

Assumptions:

- tiling is such that data within each cache line is used completely (memory bus payload fully utilized).
- Cluster: Computation and communication can be overlapped completely (asynchronous version). Tilings for local and remote transposes are independent. Each matrix element consumes inter-process bandwidth once. Messages can be pipelined, so we only experience one message latency. Filling the pipeline involves reading one input tile and storing it in a contiguous buffer, which is thus completely exposed.

- We ignore the fact that one tile does not need to be migrated (local to the calling core or cpu).
- For large matrices memory and message latencies are irrelevant.
- Each matrix element consumes memory bandwidth twice (read + write).
- Multi-core: for small matrices the transfer speed is limited by the on-chip core-to-core bi-section bandwidth. The assumption is that even in the case of logically shared caches, they will be physically distributed (tiled architecture)

Multi-core:

$T_L = 2*n^2*L_W/\underline{\beta}_{CM}$

$T_S = 2*n^2*L_W/\underline{\beta}_{C\$}$

Cluster:

Large matrices:

Pipeline fill time: 2*tile_size/memory_bandwidth = $2*(n/P)^2*L_W/\beta_M$.

Duration of pipeline stage: max(time to fill new message, time to send all messages across the network) = max($2*(n/P)^2*L_W/\beta_M$, $(n/P)^2*P*L_W/\underline{\beta}_N$

Number of pipeline stages: P-1

$T_L = L_W*(n/P)^2*[2/\beta_M + (P-1)*\max(2/\beta_M, P/\underline{\beta}_N)]$

Small matrices:

Pipeline fill time: message latency + time to issue load and store instructions to fill one message (cached data) = $\alpha_N + 2*(n/P)^2/IP$

Duration of pipeline stage: max(time to fill new message, time to send all messages across the network) = max($2*(n/P)^2/IP$, $(n/P)^2*P*L_W/\underline{\beta}_N$)

Number of pipeline stages: P-1

$T_S = \alpha_N + (n/P)^2*[2/IP + (P-1)*\max(2/IPS, P*L_W/\underline{\beta}_N)]$

Vector Reduction

Name: REDUCE

Description: The goal of this operation is to aggregate data owned by individual threads on a master thread. Each thread initializes a pair of different real-valued vectors, v_0, v_1, of length n. After the reduction operation, a single vector v_0 of length n results that is the element-wise sum of the collection of input vectors. In pseudo-code notation: $\forall$i,thread: $v_{0,thread}(i)$ += $v_{1,thread}(i)$; $v_{0,0}(i) = \Sigma_{thread}\, v_{0,thread}(i)$.

The reason we use two vectors per thread is to satisfy the PRK requirement that each kernel does real work, which should be independent of the resources applied to the problem. Single-vector reduction on a single thread would not involve any work.

Depending on the size of the vectors, the number of threads, level of sharing between caches and memories attached to the cores, performance will be governed by memory bandwidth, inter-core bandwidth, and inter-core synchronization.

At the time of development of this kernel, OpenMP/C lacked a vector reduction operation, so we provide it explicitly. Depending on architectural and problem parameters, different reduction algorithms will be optimal. We use three functionally different algorithms.

1. The master thread reads data from other threads and adds these values to its own vector elements. This algorithm leads to an unbalanced load for all non-trivial numbers of threads.
2. The threads employ a binary reduction tree. This algorithm leads to an unbalanced load for all non-trivial numbers of threads, though the unbalance is not as severe as for algorithm 1. We provide two variants of this option, using either pairwise or global synchronization among threads between stages of the binary reduction tree.
3. The threads use a modified scatter/gather algorithm [VANDEGEIJN] that is significantly better balanced for larger numbers of threads.

In MPI we currently use the library function MPI_Reduce. In a future version of the Parallel Research Kernels, this will be replaced with hand-coded versions that implement the corresponding algorithms in OpenMP.

Initialization: $\forall$i,thread: $v_{0,thread}$ (i) = $v_{1,thread}$ (i) = 1

Verification: At the end of K iterations, using P threads, we should have: $v_{0,0}$(i) = K+1 + K*(K+3)*(P-1)/2.

Usage: Vector reduction is used in scientific computing, statistics, data mining and other parallel applications. It is sufficiently common that MPI (C/Fortran) includes it.

Performance expectation:

Assumptions:

- For large vectors memory and message latency can be ignored, and algorithm 3 is optimal. For short vectors latency dominates, and algorithm 2 is optimal. Expectations are based on these algorithms.
- Because this is a streaming application with negligible reuse, the cost of floating point computations can be ignored for large vectors. For small vectors we can assume the data has been cached so we ignore memory costs, and the limiting factor becomes the floating point speed.
- Multi-core: Algorithm 3: The algorithm has three stages:
  1. local reduction; two local loads and one local store per vector element per core.
  2. scatter reduction: P-1 phases. In each phase each vector element is subject to one remote and one local load, and one local store.
  3. gather reduction: each element vector element is read locally and stored remotely.

  All phases are load balanced, meaning all cores execute the same number of loads and stores. Even if loads and stores are purely local, we assume they are limited in the aggregate by the on-chip bi-section memory bandwidth.
- Cluster: Algorithm 3: The algorithm has three stages:
  1. local reduction: two loads and one store per vector element per process.

2. scatter reduction: P-1 phases. In each phase each vector element is communicated once. In addition, two local reads and one local store are involved.
3. gather reduction: all processes send their (partial) result data to the master process, which receives a total of one whole vector length.

We assume all communications are bound by the bi-section bandwidth.
Because of dependencies, communication and computation cannot be overlapped.

Multi-core:

Large vectors:

(total # loads + total # stores) / memory bandwidth =
$(3*P*n + 3*(P-1)*n/P + 2*n)*L_W/\underline{\beta}_{CM}$

$T_L = n*(3*P^2+5*P-3)\ L_W/(P*\underline{\beta}_{CM})$

Short vectors:

1+log(P) stages. During each stage the active threads all busy themselves with doing 1 flop per vector element. The length of the vector does not change.

$T_S = (1+\log(P))\ *n/(½FP)$

Cluster:

Large vectors:

memory access time + communication time =
$(3*n+(P-1)*3*n)*L_W/\beta_M + ((P-1)*n*L_W + n*L_W)/\underline{\beta}_N$

$T_L = P*n*\ L_W\ *(3/\beta_M + 1/\underline{\beta}_N)$

Short vectors:

One local reduction stage, and log(P) communication + local reduction stages. At stage k of the reduction, the amount of data communicated is $n*2^{k-1}$words. Note that data transfer can never be faster than uncontended point-to-point bandwidth.

$T_S = n/(½FP)*(\log(P)+1) + \log(P)*\alpha_N +$

$\Sigma_{k=1}^{\log(P)} \max(n*2^{k-1}*L_W/\underline{\beta}_N, n*L_W/\beta_N)$

Sparse matrix vector multiplication: irregular memory access on read

Name: SPARSE

Description: Multiply a non-symmetric, matrix square M of a given sparsity (fraction of non-zero matrix elements) and order n, with a dense vector b. The result is vector a. Depending on the size of the vector and the sparsity, performance will be governed by inter-core bandwidth and memory latency.

This process is modified as follows to satisfy the verifiability requirement. After each iteration the input vector is changed slightly, and the result of the matrix-vector

multiplication is added to the result vector instead of replacing it. In pseudo-code notation: a += M b; b(j)+=j;

Initialization: The sparse matrix is built as follows. The standard star-shaped stencil with a user-specified radius is applied to a structured 2-dimensional square grid. Example of a star stencil operation with radius r=2:

$$a(p,q) = c_1*b(p-2,q) + c_2*b(p-1,q) + c_3*b(p,q) + c_4*b(p+1,q) + c_5*b(p+2,q) + c_6*b(p,q-2) + c_7*b(p,q-1) + c_8*b(p,q+1) + c_9*b(p,q+2)$$

Here the coefficients c_1 through c_9 are constants. Non-overlapping arrays a and b signify field variables defined on the grid, indexed by integral grid point coordinates. If we linearize a and b in the canonical fashion, we can write the stencil operation as: a += M b,. where M is the sparse matrix of interest.

A square grid with linear dimension 2^n has $2^{2n} = 4^n$ points. Hence, M has 4^n rows and 4^n columns, for a total of 16^n elements, but with only $(4r+1)*4^n$ or nonzeroes. The user specifies n. The stencil is applied in a periodic fashion, i.e. it wraps around the edges of the grid.

The columns of M are permuted in a pseudo-random way (we use bit reversal as the permutation), resulting in a general irregular sparse matrix, but with a known number of 4r+1 nonzeroes per row. We use Compressed Row Storage (CRS) to store only the nonzero matrix elements. Within a single (compressed) row matrix elements are stored in the order of increasing column index.

Initialization

$\forall i,j$ M(i,j) = 1/j (elements within the sparsity pattern only), a(j) = 0, b(j) = j. M does not correspond to any realistic discretization of a continuum problem, but makes verification easy.

Verification: After K iterations: $\forall j$: a(j) = (4r+1)*(K+1)*K/2

Usage: This kernel is used extensively in optimization problems, data mining and implicit PDE solvers.

Performance expectation:

This kernel is dominated by irregular vector reads and regular matrix reads, plus regular vector writes. There is usually little practical reuse of data.

Assumptions:

- requests for the irregular vector reads can be pipelined, so we can ignore memory latency, but only one word out of each cache line is used
- Long vectors: because there is negligible reuse, performance is dominated by memory traffic, and computational cost can be ignored, as long as the FPU units are properly pipelined. Although in practical situations the performance of this kernel may be dominated by TLB misses, this is an artifact of the software environment (page size), not of the underlying hardware, so we ignore this effect.
- CRS requires the storage and reading of indices to be used for indirect referencing. We take into account the bandwidth requirement of reading these indices, but assume that an index is as long as a floating point word, for convenience.

- Short vectors: All data is assumed to be already present in cache, and we ignore the cost of loading and storing data for both the vector initialization part and the matrix-vector multiplication part.
- Cluster: the method uses static domain decomposition (by rows) of the matrix and replicates the vector, so no communications are involved in doing the actual multiplication, but each process initializes its own chunk of the vector and broadcasts that to the other processes.
- Multi-core: each thread (re-)initializes its own chunk of the vector before each multiplication.

Multi-core:

Long vectors:

For each of the 4^n rows of the matrix a multiplication involves reading 4r+1 words randomly from the vector, requiring (4r+1) cache line loads. It also requires reading 4r+1 indices (stride one), 4r+1 matrix elements (stride one), one read of the result vector element (stride one), and one write of the result vector (stride 1). For the vector initialization we incur one write (stride one) per row of the matrix. Since all cores will be loading and storing simultaneously, we use the bi-section memory bandwidth to evaluate the cost of message traffic.

$T_L = 4^n[(4r+1)\ (L_\$+2L_W)+3L_W]/\ \beta_{CM}$

Cluster:

Long vectors:

Each process owns $4^n/P$ rows of the matrix and the same number of vector elements. Hence, we can compute the non-communication part of the execution time based on the multi-core evaluation. The communication involves broadcasting the locally generated vector segments to all processes. For long vectors the optimal algorithm communicates (P-1) complete vectors in the aggregate in a number of stages (see Reduction kernel). We use the cluster bi-section bandwidth to compute the cost of that data traffic.

$T_L = 4^n/P*[(4r+1)\ (L_\$+2L_W)+3L_W]/\ \beta_M + 4^n(P-1)*L_W\ /\ \beta_N$

Stencil: multiple regular strides on memory read access, unit stride on write

Name: STENCIL

Description: This kernel applies a scalar stencil operation to the interior of a two-dimensional discretization grid of size nxn. The stencil, which reflects a discrete divergence operator, has radius r. It is either star shaped or square, resulting in reuse factors of 4r+1 or $(2r+1)^2$, respectively, provided all the data belonging to a strip of the grid swept out by the stencil fits in cache. The distributed-memory version uses a two-dimensional domain decomposition to minimize the communication demand. The stencil computation is completely data parallel. Message passing is required to obtain boundary values from logically nearest neighbors. To ensure verifiability we write the stencil operation S with input field a and output field b in pseudo-code notations as: a += S b;

b++. In this case we cannot update an input field element the moment it is used in a stencil operation, because its "old" value is still required for other stencil evaluations. Hence, incrementing b takes place after the stencil operation is applied to all eligible elements of b. This implies one access to b in addition to any required for the stencil implementation.

Initialization

$\forall i,j\ b(i,j) = c_x*i+c_y*j$; $a(i,j) = 0$. The weights of the stencil are chosen such that they represent the discrete divergence operator on a grid with unit mesh spacing.

Verification

After K iterations we have: $\forall i,j\ a(i,j) = K*(c_x+c_y)$

Usage: Stencil operations form the core of almost all structured-grid computations. They are also used in local image filter operations.

Performance expectation: TBD

Random access update: irregular memory access on both read and write

Name: RANDOM

Description: This kernel generates within an array of length N a pseudo-random stream of n addresses whose contents need to be modified according to a bit mask. This tests the ability and efficiency of the system to read and write data elements with random stride, and to protect against data races for the same element. It is derived from the HPC Challenge Random Access benchmark. Since data is both read and updated, a major challenge is to pipeline memory accesses in the face of potential conflicts between successive writes to the same memory location by the same thread, as well as by different threads. If such pipelining cannot be done, performance of the kernel is governed by memory latency for large tables, and the sizes of the caches and memory covered by the TLB for small and intermediate size tables.

Threads can either all update elements of the same table of values simultaneously, in which data races can potentially arise, or of private, non-shared tables of values. Updates of the same element by different threads can be interchanged, as long as they happen atomically.

The algorithm to generate the stream of addresses has a look-ahead feature in which it is possible to jump to any sequence number within the stream without generating preceding addresses, which allows parallelization of the process.

The HPC Challenge specification of the algorithm was modified to use an initial seed for the random number generator that is not biased towards certain table elements. The user specifies the average number of times each table element is updated. Due to the nature of Random Access, this does not mean that each table element will be visited exactly that many times, since table indices are generated (pseudo-)randomly. However, we organize the table updates in two equal-size "rounds," and reinitialize the index computation before each round. This means that even though in each round not each table element is

updated the same number of times, in the two rounds combined each individual table element is updated an even number of times. Because the table element update method is an involution, this guarantees that, in the absence of errors, the two rounds should return the table elements to their initialization state.

Initialization: $\forall i$: Table(i) = i.

Verification: After an even number of rounds, we must have: $\forall i$: Table(i) = i. A user-specified percentage of errors is allowed.

Usage: This operation is used frequently in transaction processing and national security oriented applications.

Performance expectation:

This kernel is dominated by random reads of memory, and by writes to the same location.

Assumptions:

- Although there is a sequential dependency in the generation of addresses at which data needs to be read and updated, that generation itself is much faster than fetching the data from memory and writing it back. Hence, we can ignore the cost of the address generation.
- Because of the possibility of accessing the same table element multiple times in quick succession, we assume that memory accesses are fully serialized (not pipelined), so that for long vectors each access suffers a memory latency. We only suffer that latency upon the reading, writing back data is overlapped with reading new data. Memory bandwidth is immaterial.
- We ignore the effect of TLB misses, although on current systems that is often a dominant part of the cost.
- Multi-core: We ignore effects of false sharing.
- Cluster: We use the binning method for communicating update requests to "remote" processes. Bin size is K words, and we ignore any load imbalance. We choose K sufficiently small that any messages can stay in cache, which means we do not need to pay a memory access cost for those messages. As a first order approximation we will assume all n indices generated by the calling process are scattered to other processes.

Multi-core:

Long vectors:

execution time = number of table accesses per core * memory latency

$T_L = (n/P)* \alpha_M$

Cluster:

Long vectors:

execution time = local update + message passing cost = number of table accesses per process * memory latency + network bandwidth cost of sending all generated indices to other processes + number of messages per process * message latency

$T_L = (n/P)* \alpha_M + n*P* L_W /\underline{\beta}_N + (n/K)* \alpha_N$

Synchronization

Name: SYNCH

The two most common and most useful synchronization types in parallel programs are point-to-point (peer-to-peer), and global. These do not actually constitute work, but they are necessary to guarantee consistency of results. Synchronization is never a goal in its own right, it only makes sense if some information transfer between threads takes place. Hence, our synchronization tasks involve a certain amount of data transfer, either implicitly through shared memory consistency updates, or explicitly through messages. Synchronization implies *ordered* access to memory/data, as distinguished from *exclusive but unordered* access (see locks, atomicity).

Global synchronization: While this is typically fairly expensive and should be avoided if possible, it is a convenient and easy-to-use mechanism to produce correct code, and is sometimes required by a certain algorithms. Hence, it has to be executed with high efficiency on the target platform. The following task stresses that functionality. Each thread out of a total of P receives a (sub)string of a fixed number of n characters. The threads combine these into a single string S by concatenating their substrings in the order of their thread number. Using a deterministic algorithm, they then assemble another substring of the same length out of the concatenated string. Specifically, let substring S_t be assigned to thread t. The global string is then written as $S = \Sigma_t S_t$, where summation is understood to be in-order concatenation. The n characters of the new substring S_t' are obtained by: $S_t'(i) = S(t+i*P)$. However, the selection process should be agnostic to this particular filter function, meaning that each thread should have read access to the entire concatenated string. This process of concatenation and substring formation is repeated.

Initialization

The initial substring S_0 used consists of ASCII characters corresponding to digits: "276384726387462837427123112078 92." If the user specifies a longer substring than this, it is filled by repeating the same pattern of characters.

Verification

This kernel does not currently conform to the PRK design rules, since the checksum we compute by adding all the digits of the concatenated string is an invariant: $\Sigma(int)S(i) = P*\Sigma (int)S_0(i)$.

The global synchronizations can be implemented using a barrier in OpenMP and MPI_Allgather in MPI. There is no work to do that can be overlapped with the data exchange between cores. Depending on the size of the concatenation string and the number of threads, performance of the kernel will be governed by inter-core latency, inter-core bandwidth, or memory bandwidth.

Performance expectation:

Assumptions:

- The synchronization string is sufficiently long to avoid false sharing in the multi-core case. Hence, there will be no contention for write ownership of cache lines

- The barrier in the multi-core case is implemented using an O(1) memory latency algorithm [FastBarrier]. It needs to be called twice for each iteration. We use 2 memory latencies per barrier.
- The asymptotically optimal algorithms for long and short vectors for MPI_Allgather described by Van de Geijn et al. [VANDEGEIJN] are used to implement the synchronization on clusters.

Multi-core:

Long strings:

execution time = (time to read and write all strings). For reading the stride is P-1, which means each read of a cache line contains max($L_\$$/P,1) characters requested by the calling core. Hence, each core needs to read n* $L_\$$/max($L_\$$/P,1) bytes from memory. It writes n bytes (stride 1) to memory.

$T_L = n*(L_\$/\max(L_\$/P,1)+1)*P/\beta_{CM}$

Short strings:

execution time = (time to read and write all strings + 2 barriers). Reading is (mostly) remote, but writes are local and stride one, so they happen at the peak integer performance.

$T_S = n*P/\beta_{C\$} + n/IP + 2*2*\alpha_M$

Cluster:

Long strings:

each node needs to read data from the global string, stored in local memory, write back a substring in local memory, and do an allgather step. See multi-core for the number of bytes read. Writing is local and contiguous. For the allgather we can assume that the global string is copied across the network a total of (P-1)*2 times, once in each of the (P-1)*2 phases.

$T_L = n*(L_\$/\max(L_\$/P,1)+1)/\beta_M + n*(P-1)*2/\beta_N$

Short strings:

All reading and writing is between registers and cache, and goes at the peak integer performance. Since communication latencies are important, the optimal algorithm [VANDEGEIJN] tries to reduce the number of message/communication phases.

$T_S = 2*n/IP + \log(P)*\alpha_N + n*(P-1)*2/\beta_N$

Point-to-point synchronization: This is the most efficient way, in principle, of communicating between threads, and the programmer should be allowed to use it without fear of destroying performance of the code. The task in which we embed this operation is the one-dimensional software pipeline. A two-dimensional array A of size n x m is distributed among the threads in vertical strips. We apply the following difference stencil to the array values in such a way that only those elements are used that have been updated previously: A(i,j) = A(i-1,j) + A(i,j-1) – A(i-1,j-1). The pipeline algorithm to be used is as follows. The first thread computes one partial row (fixed j) of updated elements

of A. It then synchronizes with its right neighbor thread, which continues within the same row, while the first thread starts the second row. At the next synchronization point the third thread can commence its part of the first row, etc. After a sweep over the entire grid has been completed we copy the value at the top right corner of the grid to the bottom left corner after flipping its sign: A(0,0) = -A(n-1,m-1), in preparation for the next sweep over the grid (next iteration).

Initialization

This kernel only requires the values at the bottom and left boundary of the grid to be defined: $\forall i$: A(i,0)=i; $\forall j$: A(0,j)=j.

Verification

After K iterations we must have: A(n-1,m-1) = K*(n+m-2).

Peer-to-peer synchronizations can be implemented using shared flags plus the `flush` directive in OpenMP, and point-to-point messages in MPI, respectively. These techniques incur a forced write to and read from shared memory, and an inter-process communication latency, respectively. The latter may in turn require a forced write and read on a multi-core system with shared memory.

Performance expectation:

Assumptions:

- There is negligible data shared or communicated by the threads/processes in the course of the computation, so we can ignore the bandwidth involved in synchronization.
- The stencil is sufficiently compact that all data for it can stay in cache, even for large grids. Hence, each array element needs to be read from memory at most once. We also need to write one array element per stencil computation.
- For large n the synchronization cost (both MPI and OpenMP) can be ignored.
- The stencil operation has no multiplications, so peak floating point speed is limited to ½FP. We assume that memory loads and FPU activity can be overlapped in case of large (both n and m) grids, using prefetching. We ignore the effect of data dependencies that could inhibit pipelining the computations.
- We ignore load imbalance caused by pipeline fill and drain.
- Multi-core: memory bandwidth is shared, but each thread only carries out its own part of the computation.
- Cluster: memory bandwidth is not shared.

Multi-core:

Large grids:

> For each grid line segment (n/P points) each core needs to read n/P new words from memory, write back n/P word (both stride one), do 2 flops per point, and one synchronization (ignored).
>
> $T_L = \max(2*n*m* L_W /\underline{\beta}_{CM}, 2*(n/P)*m/½FP)$

Small grids:

All data can now be read from and written to cache so we assume that loads and stores are hidden by computation. We do need to take synchronization costs into account.

$T_S = 2*(n/P)*m/½FP + m*\alpha_M$

Cluster:

Large grids:

Similar to the multi-core case, except that memory bandwidth is not shared.

$T_L = \max(2*(n/P)*m*L_W/\beta_M, 2*(n/P)*m/½FP)$

Small grids:

Similar to multi-core case.

$T_S = 2*(n/P)*m/½FP + m*\alpha_N$

Usage: Parallel programs with data dependencies typically require inter-thread coordination and synchronization. In OpenMP a global synchronization is often implied at the end of work sharing constructs.

Atomic reference counting, shared and private

Name: REFCOUNT

Usage of shared variables is specific to the shared-memory programming model. Traditionally, locks and critical sections are used when groups of statements or statements with side effects must be executed atomically, i.e. by one thread at a time, to protect against competing for shared resources. Oftentimes they are used in the process of parallelizing a code to guarantee correctness. Locks may severely degrade performance, since they serialize access among threads that share the lock. Nonetheless, if used judiciously, they can help speed up the programming process without undue application slowdown. In properly constructed applications with significant use of locks, many locks will be uncontended, that is, the thread trying to acquire the lock will not have to wait. Optimizing for this situation, however, may lead to suboptimal treatment of the contended lock, causing potentially disproportionate slowdown of applications in which the locks are active. Hence, both contended and uncontended locks must be evaluated.

An alternative to locks is offered by Transactional Memory, which supports grouped atomic operations with light-weight synchronization.

Shared reference counters: All threads update a pair of shared counters, C_1, C_2, in tandem N times; this prohibits the use of the atomic directive in OpenMP, which only protects updates of a single memory location. The counters may not be stored in adjacent memory locations, which prevents an atomic update of a single large word. Performance of this kernel is governed by the system support for mutual exclusion.

We note that the performance of this kernel in its elementary form is almost certainly optimal if threads would access the counters in batches of maximum size, with minimal interleaving. This is because there is no opportunity for the threads to do work concurrently. Hence, no parallel speedup can be expected, which is obviously not true in

actual application. Therefore we allow the user to instruct the threads to do some work on a private data structure after each counter pair update. Forcing threads to wait their turn to do all their counter updates in one large batch completely serializes the process, whereas interleaving would allow concurrency. For this work we select the serial scaled vector addition kernel (NSTREAM), for which we already have a simple verification test. The length of the stream determines how much private work is available to the threads.

We define two variations of this kernel:

1. The updates to C_1 and C_2 are *independent*, i.e. (C_1++,C_2++), so could, in principle, be carried out with scalar atomic operations applied to the individual counters without the need to insist on a single transaction. Interleaved access to the counters by different threads within one counter pair update is not captured by this variation.
2. The updates to C_1 and C_2 are *dependent*, $(C_1,C_2)' = R(\theta)\ (C_1,C_2)$, where the 2x2 matrix $R(\theta)$ represents rotation through an angle of θ radians. In this case any interleaved thread access within one counter pair update will change the final value of the counter pair. We note that $R^K(\theta) = R(K\theta)$. For this kernel we set θ=1. Because a full circle equals π radians, we can be assured that this process will never return the counter pair to its original state.

Initialization

$(C_1,C_2) = (1,0)$.

Verification

After K updates we must have:

1. $(C_1,C_2) = (K+1,K)$
2. $(C_1,C_2) = (\cos(K),\sin(K))$

Performance expectation:

Assumptions:

- If the multi-core system supports atomic read-modify-write semantics, acquiring and releasing a shared lock each requires one memory latency, using, for example MCS-style list-based queueing locks . If not, then Lamport's Bakery Algorithm [Lamport] shows that 5 memory latencies are sufficient.
- We assume a fair lock, which means that after the threads queue their first request for the lock guarding the counters, they will be served in order, and after each lock acquisition and release the thread will enter the queue at the end. This means that a different thread will update the counters upon each lock acquisition.
- Updating the two reference counters can be overlapped with memory operations, but the counters themselves must be read from memory before we can update them.

Multi-core: $T_L = 4*\alpha_M*N*P$

Private reference counters: All threads update a pair of private counters, which are stored in a shared array whose rows are indexed by thread number. While any locks can be removed without affecting correctness, the compiler will not do so because the array is

shared. Performance of this kernel is governed by the effectiveness with which the system supports non-conflicting atomic transactions.

Performance expectation:

Assumptions:

- Reference counters as well as lock variables are kept in registers.
- Acquiring and releasing the private lock variables and updating the reference are integer operations. Each counts as a single, non-vectorizable integer operation.

Multi-core: $T_L = 4*N/IP$

Usage: Atomic constructs are often used to guarantee correctness of parallelized code whose correctness is not easily established by the programmer or parallelization tool.

Scaled vector addition (Stream triad)

Name: NSTREAM

Description: This task, derived from McCalpin's Stream benchmark, also often called DAXPY, entails addition of two vectors of length n, one of which is scaled by a constant, to form a third vector: a(i) += b(i) + q*c(i). It measures the speed with which the processor can maintain a stream of contiguous memory reads and writes, without any reuse. By varying the size of the loop, the different levels of the memory hierarchy are exercised. Prefetching is crucial to obtaining performance.

Usage: This test is a standard measure for the ability of a system to support applications with good spatial locality but limited or no data reuse.

Assumptions:

- Short vectors can be run out of cache.
- The computation fully utilizes the FPUs.
- There is no communication between the threads.

Performance expectation:

Multi-core: $TL = n*3\ L_W/\underline{\beta}_{CM}$

$TS = (n/P)/FP$

Cluster: $TL = (n/P)*3*L_W/\beta_M$

$TS = (n/P)/FP$

Dense Matrix multiplication

Name: DGEMM

Description: This is the most important of the BLAS3 routines. The full DGEMM is defined as

$$C = \sigma A*B + \rho C$$

where A, B and C are double precision n x n matrices, σ and ρ are scalars, and input parameters to the routine control whether the matrices are transposed or not. To simplify our analysis, we restrict ourselves to the case without any transposed matrices, and with σ=1, ρ=0.

While getting good performance for DGEMM (in excess of 40% of peak execution rate) is fairly straightforward on most systems because of its favorable temporal and spatial locality properties in case the matrices are blocked, obtaining a fraction of peak performance that is close to 100% usually requires substantial and non-obvious platform dependent tuning. Unlike most of the other kernels, performance is not governed by bandwidth between the different levels of the memory hierarchy, but by the proper blocking supporting data movements, and by scheduling work for the execution units.

Usage: DGEMM is the standard building block of dense linear algebra. It is used in some optimization problems, scientific computing (quantum chemistry, electrodynamics, etc), and a signal processing, etc. DGEMM is also the heart of the well-known Linpack benchmark.

Performance expectation:

Multi-core/Cluster: $T_L = T_S = n^3/(P*FP)$

Branching Bonanza

Name: BRANCH

Description: While synthetic work-load proxies may feature regular and irregular data access, they often are quite regular and predictable in terms of code paths, especially inside loops or loop nests in which most of the work is carried out. Real applications, however, often feature inner loop data-dependent branches. These can defeat speculative execution and prefetching, which might otherwise return sizeable performance improvements. Branch-intensive loops can be divided into several broad categories: a) executed statements differ depending on the branch(es) taken, but the memory references do not change; b) executed statements differ, depending on the branch taken, as well as the memory locations referenced; c) each branch choice causes a call to a different substantial function, causing potentially large numbers of instructions to be loaded into the instruction cache and discarded from it at high frequency; d) each branch choice corresponds to an alternative, with no or merely inlined calls to short functions. These categories are not all mutually exclusive.

In this kernel we focus on four cases. The first three all concern light-weight loops (length L), i.e. loops that have very few instructions associated with them.

1. branches inside vectorizable loops where the introduction of the branch does not necessarily inhibit vectorization.
2. branches inside vectorizable loops where the introduction of the branch does inhibit vectorization.
3. branches inside non-vectorizable loops.
4. branches inside loops in which each branch corresponds to a sizeable and different set of instructions.

Approach: For the light-weight loops we select the loop bodies as follows, where i is the loop index, expr1 is a very simple arithmetic expression, expr2 is an expression containing a single value, and expr3 is functionally the identity. All arrays and constants are of the integer data type.

```
aux = expr1(i);
if (expr2(i)>0)vector[i] -= 2*vector[expr3(i)];
else           vector[i] -= 2*aux;
```

Expr1(i) is the same for all three cases, and numerically equals vector[i] upon entry of the loop. Expr2(i) equals either aux (cases 1 and 3), or vector[index[i]] ≡ vector[i] (case 2). Expr3 in the first assignment of the loop body equals i (cases 1 and 2), or index[i] ≡ i (case 3). Vector has approximately the same number of positive and negative elements. Sign changes occur approximately every 3 to four loop iterations. Each version of the loop with a conditional branch has a counterpart that does not contain the branch, but which does the same amount of computation.. Specifically, the loop body gets replaced with:

```
aux = expr1(i);
vector[i] -= 2*(vector[expr3(i)]+aux);
```

The result of these choices is that the three different tasks all do exactly the same amount of computational work, so that the impact of the different types of branches can be compared. Moreover, the branches are "unpredictable," meaning that if the compiler guesses them to be always taken or to be always not taken, it will be wrong about 50% of the time. This ensures that the cost of branch misprediction will be measurable.

For case 4 we use a somewhat different approach. Expr1(i) is now not a simple expression, but is obtained as follows. A square matrix A of user-defined size is filled element by element, such that the resulting matrix is always the identity, but the actual instructions are different for different values of i modulo another prescribed value. Subsequently, the matrix B is computed as the arithmetic average of A and A^T. The returned value of expr1 is i if B equals the identity matrix, and zero otherwise. Finally, we compute the vector element in the same way as the non-branching version of cases 1 and 2.The non-branching version of the loop is essentially the same as the branching version, except that the construction of matrix A is now no longer dependent on the value of i.

The result of these choices is that an amount of program text proportional to N*N, where N is the matrix order chosen, is generated for each different branch in the loop.

Performance expectation:

Assumptions:

- integer operations are not pipelined, they complete in the integer units in 1 cycle.
- A comparison counts for one integer operation.
- Indexing into an array (address computation) counts for one integer operation. That does not include the computation of the index itself.
- Each iteration of 1), 2), 3) does 7.5, 5.5, and 7 integer operations, respectively, on average (depends on outcome of test). 2) vectorizes (with masking), but 1) and 3)

do not. Masking means that 2) will execute more instructions (7.5) than are actually used, but they are vectorized.

- The data set size for 1), 2) and 3) is constant and small, and the number of instructions involved is negligible, even for large problems (many iterations). Hence, for these problems instructions and data are all cached, so we do not need to take into account the cost of memory access.
- Each iteration of 4) does (20+4+2*N*N) integer operations
- Each function of 4) contains K*N*N bytes of instructions (K depends, among others, on the level of optimization), which have to be read from memory for large problems. We assume that the reading of these instructions can be overlapped with computations.

Performance expectations:

Multi-core:

1. $T_S = T_L = 7.5*L/IP$
2. $T_S = T_L = 7.5*L/IP$
3. $T_S = T_L = 7*L/IP$
4. $T_S = L*(24+2*N^2)/IP$
   $T_L = L*max((24+2*N^2)/IP, K*N^2*P/\beta_{CM})$

Cluster:

Items 1, 2, 3 and the value of T_S from item 4 are identical to the multi-core expectations. The only difference is for T_L for which the expectation is:

$$T_L = L*max((24+2*N^2)/IP, K*N^2/\beta_M)$$

Acknowledgements

This list of kernels was developed in close partnership with parallel application specialists in Intel's Parallel and Distributed Systems Division (Hugh Caffey, Thanh Phung and Hans-Joachim Plum). We obtained valuable inputs from Arch Robison, Bruce Leasure, Michael Greenfield, and Michael Frumkin (currently with NVIDIA Corp.).

References

[Bailey04] David H. Bailey and Allan Snavely, "Performance Modeling: Understanding the Present and Predicting the Future", www.dhbailey.com, Nov. 2004.

[Luszczek2005] Piotr Luszczek, Jack J. Dongarra, David Koester, Rolf Rabenseifner, Bob Lucas, Jeremy Kepner, John McCalpin, David Bailey, and Daisuke Takahashi, "Introduction to the HPC Challenge Benchmark Suite", 2005.

[OMP microbenchmark suite] Mark Bull of EPCC

[Stream] J.D. McCalpin. STREAM: Measuring sustainable memory bandwidth in high performance computers, www.cs.virginia.edu/stream, 1995.

[VANDEGEIJN] Ernie Chan, Marcel Heimlich, Avijt Purkaastha, and Robert van de Geijn, "Collective Communication Theory, Practice, and Experience", Concurrency and Computation: Practice and Experience, 00:1-38, 2006.

[McCalpin2001] John D. McCalpin, "An Industry Perspective on Performance Characterization: Applications vs Benchmarks", presented at the DOE/NSA Workshop on Performance characterization of algorithms, Oakland, CA, July 17, 2001.

[FastBarrier] Zhen Fang, Lixin Zhang, John B. Carter, Liqun Cheng, Michael Parker, "Fast Synchronization on Shared-Memory Multiprocessors: An Architectural Approach," J. Parallel Distributed Computing vol. 65, 1158 – 1170, 2005.