Performance Evaluation of Scientific Applications on POWER8

Andrew V. Adinetz[1], Paul F. Baumeister[1], Hans Böttiger[3], Thorsten Hater[1], Thilo Maurer[3], Dirk Pleiter[1], Wolfram Schenck[4], and Sebastiano Fabio Schifano[2]

[1] Jülich Supercomputing Centre, Forschungszentrum Jülich, 52425 Jülich (Germany)
[2] Dip. di Matematica e Informatica, Università di Ferrara and INFN, Ferrara (Italy)
[3] IBM Deutschland Research & Development GmbH, 71032 Böblingen (Germany)
[4] SimLab Neuroscience, Institute for Advanced Simulation and JARA, Forschungszentrum Jülich, 52425 Jülich (Germany)

Abstract. With POWER8 a new generation of POWER processors became available. This architecture features a moderate number of cores, each of which expose a high amount of instruction-level as well as thread-level parallelism. The high-performance processing capabilities are integrated with a rich memory hierarchy providing high bandwidth through a large set of memory chips. For a set of applications with significantly different performance signatures we explore efficient use of this processor architecture.

1 Introduction

With power consumption limiting the performance of scalar processors there is a growing trend in high-performance computing (HPC) towards low clock frequencies but extremely parallel computing devices to achieve high floating-point compute performance. A remarkable increase in the number of systems exploiting accelerators like GPGPUs and Xeon Phi for leading Top500 systems can be observed. The POWER server processors, while providing increasing on-chip parallelism, continue to be optimized for high single-thread performance. In June 2014 the most recent generation of POWER processors, namely POWER8, became available in a pre-release program. In this paper we investigate the performance of this processor for a set of micro-benchmarks as well as mini-applications based on real-life scientific HPC applications.

A description of the processor architecture with up to 12 cores be found in [1] and [2]. POWER8 complies with version 2.07 of the Power ISA like its predecessor, but features changes to the underlying micro-architecture. For an early evaluation of the architecture we used a single SMP server with two dual-chip-modules (DCM) and a total of twenty cores. Each core can be clocked at up to 4.2 GHz and is capable of running up to eight hardware threads per core in simultaneous multi-threading (SMT) mode. Per core there are two floating point pipelines capable of executing single and double precision scalar instruction or vector instructions on 128 bit registers. Alternatively, this VSX unit can operate on fixed point vectors. Further, two fixed point pipelines are present. All arithmetic functional units can execute fused-multiply-add instructions and variants thereof. The interface to the memory system consists of two load/store units and two dedicated load units. All of these may execute simple fixed point computations. The dispatch unit is capable of out-of-order execution.

The cache hierarchy consists of three levels. L1 and L2 are core private and inclusive. L1 is split between data and instructions with a capacity of 64 KiB and 32 KiB, respectively. The 512 KiB L2 cache is unified. The L2 caches are connected via a cache coherency protocol and can move data between caches. The store engine is located in L2, with L1 being write-through. L3 consists of 8 MiB of embedded DRAM (eDRAM) per core and functions as a victim cache for the local L2 and remote L3 caches. The pre-fetch engine pulls data into L3 directly and into L1 over the normal demand load path.

One of the differentiating features of the POWER8 architecture is the inclusion of external memory interface chips with an integrated cache, the Centaur chip. Its additional cache level of 16 MiB eDRAM is some times referred to as the fourth level cache (L4). Each link connecting processor and memory buffer offers an 8 GB/s to 9.6 GB/s write and 16 GB/s to 19.2 GB/s read bandwidth. With up to 8 links the aggregate peak bi-section bandwidth per socket is 192 to 230.4 GB/s. The dual-socket system evaluated in this paper featured an aggregated read bandwidth of 256 GB/s and 128 GB/s for write access.

We used a pre-release version of Red Hat Enterprise Linux 7.0 which features support for the POWER8 architecture. In this paper we only report on results obtained using the GCC compiler version 4.8.2 which includes POWER8 support and offers access to vector intrinsics.

As SMP domains grow in size and heterogeneity, the placement of memory allocations becomes more important. The test system comprises four NUMA domains, as each socket consists of a dual-chip-module. The standard tool `numactl` was used for pinning allocations.

With this paper we make the following contributions:

– Performance evaluation of different aspects of the POWER8 through micro-benchmarks.
– Performance characterization on POWER8 for different scientific applications.
– Identification of a set of events relevant for analyzing performance of such applications.

2 Related Work

Recently various papers have been published exploring the POWER7 architecture and investigating its performance. Very few papers have been published about the new POWER8 processor.

The approach taken in [3] is close to our's in the sense that the performance of relevant scientific applications was analysed on a POWER7-IH system. In this paper a system comprising 8 nodes and a total of 256 POWER7 cores was used. Focussing on scale-up capabilities of POWER7, analysis of data transport performance, like memory-to-processor or processor-to-processor, was given more attention than evaluation of the micro-architecture which is the focus of this paper.

The performance evaluation presented in [4] takes a more architectural approach by analysing the performance benefits of specific features of the POWER7 processor like different SMT modes, support of different clock speeds and the use of (at that time new) VSX instructions. For this purpose synthetic benchmarks are used. Detailed information on the POWER7 performance measurement capabilities is given.

First papers on POWER8 [2, 1] mainly focus on chip design, applied technologies and I/O capabilities.

3 Methodology

3.1 Hardware counters

The POWER8 processor allows for monitoring of up to six hardware events in a single set. These events can be chosen out of more than a thousand defined counters. We identify those that map to the functional units at the disposal of the core:

Unit	**Counter**
Vector Scalar Units	`VSU{0,1}_FIN`
Fixed Point Units	`FXU{0,1}_FIN`
Branch Unit	`BRU_FIN`

The common prefix `PM_` has been suppressed for brevity in all counter names. We use PAPI version 5.3.2 to access the counter values via the interface to platform specific hardware counters [5]. In Fig. 1 we summarize our analysis of the memory architecture for the propagation of load requests. As to our knowledge, all counters are specific to the core executing the read request. However, some information is missing, like how to compute data movements from L2 to L3 and from L1 to L2.

3.2 Performance metrics

In [6] a set of performance metrics was defined in order to characterize application behavior on the BG/Q architecture. We use these metrics as a basis for our work on the POWER8 architecture. However, we focus on those relevant to the core micro-architecture, mainly instruction counts and their interplay with the available functional units. Further we address the data movement between the CPU and memory, as well as chip-internal traffic. We give a summary in table 1.

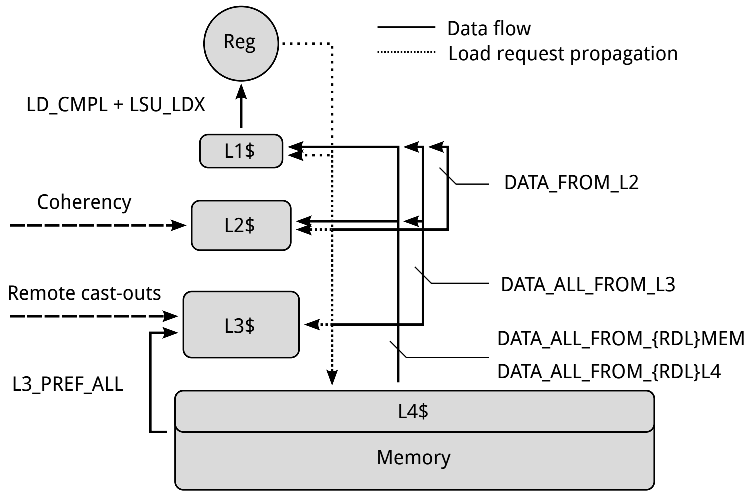


Fig. 1: Memory hierarchy for load request propagation and prefetch resolution. All values have to be scaled by the width of a cache line of 128 B, except the traffic between register file and L1 where the factor is the register width of 8 B.

3.3 Porting and tuning

All applications and micro-benchmarks were ported to the POWER8 architecture. We give results for the optimal performance we were able to attain. Details for tuning applications can be found in the relevant sections, but we give some general methods here.

OpenMP Thread placement – the mapping of threads to CPU cores – is a critical factor for the performance of concurrent applications beyond modest numbers of threads. We use the GNU OpenMP runtime control variables to control the layout. The best results are achieved by using a round-robin allocation with stride $s = min(8, \frac{160}{T})$ for T threads.

NUMA The Linux tool `numactl` was used to tune memory allocation, where the interleaving of the four NUMA domains shows the best results.

4 Micro-Benchmark Results

We investigated the baseline of available performance in terms of instruction throughput, memory bandwidth and multi-threading overhead by a series of micro-benchmarks.

4.1 Instruction throughput and latency

We use an in-house tool to measure the latency and saturated throughput of various assembly instructions. The basic approach is to time a tight loop of

Name	**Description**	**Formula**
t_{wc}	Wallclock time	CYC*
N_x	Instructions	INST_CMPL
N_{FX}	Fixed point instructions	FXU{01}_FIN + LSU_FX_FIN
N_{FP}	Floating point instructions	$\sum_n$ VSU{01}_nFLOP
N_{LS}	Load/Store instructions	LD_CMPL + ST_FIN
N_{BR}	Branch instructions	BRU_FIN
N_{fp-op}	FLOPs	$\sum_n$ VSU{01}nFLOP
$Reg \leftarrow L1\$$	Data read from L1	$8 \cdot ($LD_CMPL + LSU_LDX$)B$
$Reg \rightarrow L2\$$	Data written L2†	$8 \cdot ($ST_CMPL + VSU{01}_SQ$)B$
$L1\$ \leftarrow L2\$$	Data from L2 into L1$^+$	$128 \cdot$ DATA_FROM_L2B
$L1\$ \leftarrow L3\$$	Data from L3 into L1	$128 \cdot$ DATA_ALL_FROM_L3B
$L1\$ \leftarrow Mem$	Data from memory into L1	$128 \cdot ($DATA_ALL_FROM_{LDR}MEM$+$ DATA_ALL_FROM_{LDR}L4$)B$
$L3\$ \leftarrow Mem$	Data from memory into L3	$128 \cdot ($L3_PREF_ALL$)B$
$L3\$ \rightarrow Mem$	Data into memory from L3	$128 \cdot ($L3_CO_ALL$)B$
N_{mem}	Total data from/to memory	$L1\$ \leftarrow Mem + L3\$ \leftarrow Mem + L3\$ \rightarrow Mem$

Table 1: Performance metrics for characterizing applications on POWER8.
* Only incremented while thread is active.
† L1 is store-through.
$^+$ L1 and L2 have the same prefetch states, so no prefetch is excluded.

assembly instructions, which is then repeatedly executed to achieve stable results. Using independent instructions allows for estimating the maximum throughput, while the introduction of dependencies will yield the minimal latency between instructions. Results for a selection of assembly instructions are given in table 2.

4.2 Memory sub-system (STREAM)

We first investigated the behavior of the memory sub-system under an artificial load designed to exercise the memory bandwidth. We used version 5.9 of the STREAM benchmark [7], which we tuned for the POWER8 architecture. STREAM consists of four micro-benchmarks on the vectors **a**, **b**, **c** and a scalar

$$\begin{array}{ll} \text{copy } \mathbf{c} \leftarrow \mathbf{a} & \text{scale } \mathbf{b} \leftarrow s \cdot \mathbf{c} \\ \text{sum } \mathbf{a} \leftarrow \mathbf{b} + \mathbf{c} & \text{triad } \mathbf{a} \leftarrow s \cdot \mathbf{b} + \mathbf{c} \end{array}$$

The GCC compiler fails to recognize the opportunity to vectorize the *copy* benchmark. The necessary vectorization was done by hand using VSX intrinsics. To achieve better parallel performance, *core binding* and *NUMA placement* were investigated, see section 3.

First, we turn to the raw bandwidth between CPU and main memory. The working set size was chosen to be 512 MiB per array in order to avoid cache effects. As the STREAM benchmarks are highly regular, the efficiency of the

Instruction	**Type**	**Latency**	**Throughput**
`add`	Fixed	8	1
`ld`	Memory	–	1
`st`	Memory	–	1
`ld+st`	Memory	–	$1/7$
`xsmuldp`	64b Floating	6	1
`xsdivdp`	64b Floating	33	$1/29$

Table 2: Latency and maximum throughput for examples of fixed point, simple and complex floating point and memory access instructions.

pre-fetching mechanism has a large impact on the results. To obtain statistically sound results, we repeated the measurements 1000 times. We give the optimal results as the median values for all four benchmarks in Fig. 2 as a function of the number of threads. We find sustainable bandwidths for *triad* of just over 320 GB/s, corresponding to roughly 84.6 % of the maximum sustained bandwidth. the achievable bandwidth for *copy* and *scale* is lower than for *sum* and *triad*. The later use two load and one store streams which fits the balance of the memory links exactly. The peak performance is achieved with 40 threads, at which point every LSU is busy. For this case, the inset in Fig. 2 shows the distribution of the results over 1000 runs of the benchmark. We notice a clearly peaked distribution at the median and a quite long tail towards smaller values.

Next, we investigate the impact of the different cache levels. Due to the prefetch mechanism, we expect only the first and third level to have impact on the STREAM benchmarks. Cache lines recognized as part of a prefetch stream are fetched into L1 and L2 up to six cache lines ahead of the stream. These requests traverse the cache hierarchy like demand loads. The last level cache L3 is populated by the prefetcher directly from the memory up to 16 lines ahead of the stream. STREAM is perfectly regular, so we expect no significant impact of the L2 on the memory bandwidths. In the steady state of the prefetch engine, every load request must hit in L1 as it is large enough to hold three streams for eight threads per core. Prefetch requests themselves miss L2, as it has the same data prefetched, and hit L3 as it is ahead of the L1 prefetch. Every line is only traversed once. We monitor hardware counters to understand the impact of the prefetcher and cache hierarchy, and recorded the counter values for different array lengths. The data for the *copy* benchmark is presented in Fig. 3. Despite the effort of rotating the arrays to avoid such behavior, for small n, when the majority of the working set fits in L2, it supplies the full data to the core. For larger n, the traffic from L2 into L1 drops to a constant, due to remnants of the working set in L2. The last level cache satisfies almost all requests, including prefetches, beyond the size of L2. A constant amount of data is fetched directly from memory into L1, most likely before prefetch streams are established.

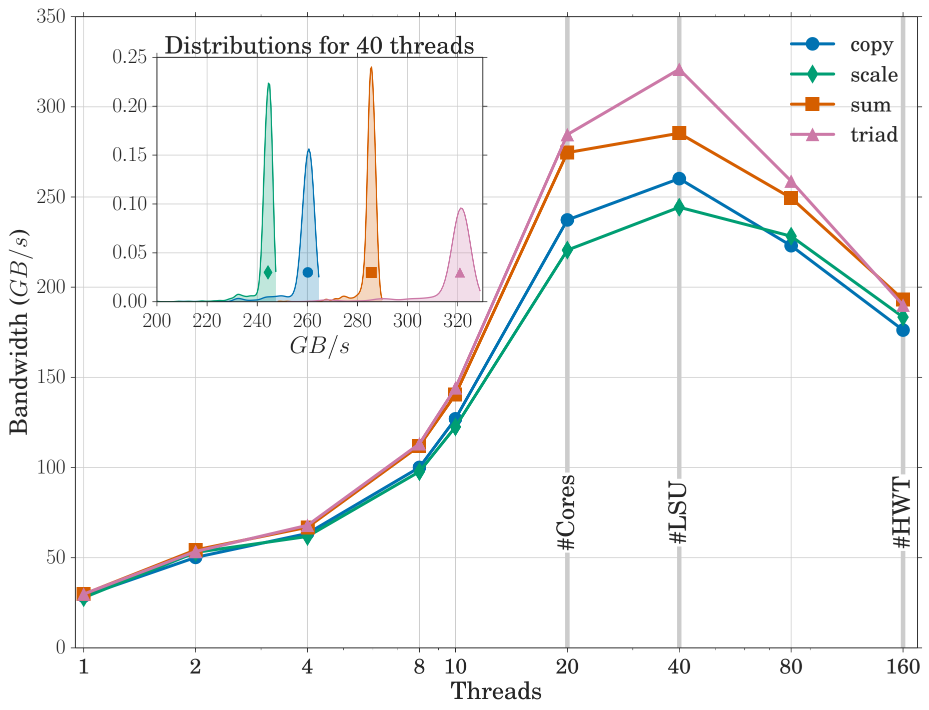


Fig. 2: Median bandwidths for the STREAM benchmarks over number of threads. We mark the thread counts where every core is occupied by single thread (#Core), by two threads (#LSU) and the SMT capacity is fully used (#HWT). Beyond using every LSU on all cores, the achievable bandwidth drops off sharply. **Inset:** Probability density estimates for the bandwidths at 40 threads over 1000 repetitions of the experiment. Note the clear peak at the median value and the relatively long tail towards smaller values, most probably indicating other system activity at the time of the iteration.

The traffic volumes between the register file and the L1 cache fit the prediction of $8{\cdot}nB$ perfectly, as does the store volume (not shown). The accumulated transfers into L1, from L2, L3 and memory, sum up to the same values within the margin of error. We find a clear effect of cache sizes as the data set grows too large for each level. The impact of the second level cache at small sizes is explained by the fact that at this point the full working set fit into L2. Although the design of the benchmark tries to avoid caching effects by rotating the assignments, this does not fully work for small sizes n.

4.3 OpenMP overheads

The POWER8 system is relying on thread-level parallelism for optimal performance. A total of 20 threads are needed to occupy all cores with a single

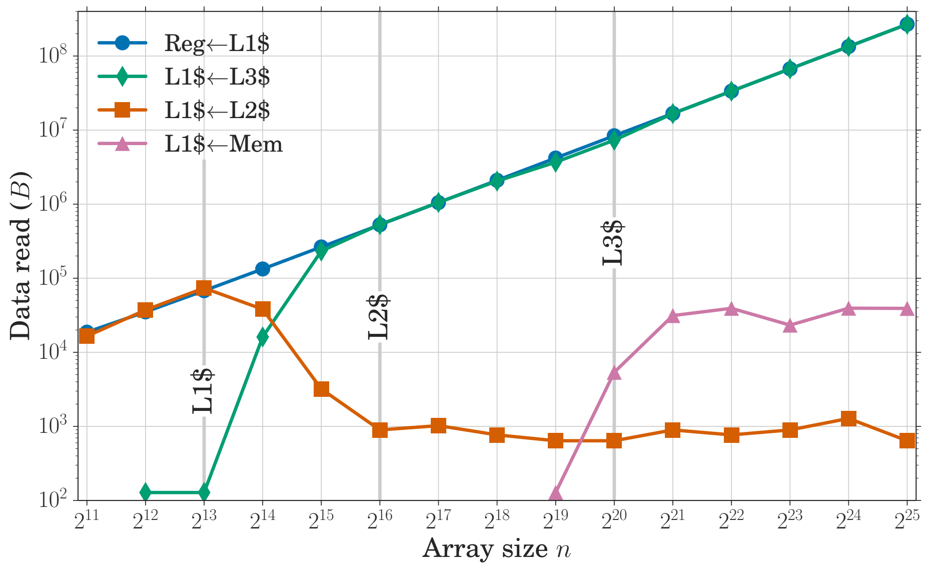


Fig. 3: Memory traffic as derived by monitoring hardware counters over the number of double precision array elements for STREAM *copy*. Counters were summarized into traffic volumes according to section 3.1 and scaled into units of bytes. The predicted value of $8 \cdot nB$ perfectly matches the line for *Reg←L1$*.

thread (ST mode). Further gains may be achieved by using multiple threads on a single core in simultaneous multi-threading mode (SMT). This has the benefit of issuing more instructions per cycle, thus utilizing voids in the execution pipelines. However, as more threads are executing on the same hardware, the overheads for managing these threads, synchronization and bookkeeping grow. Since almost all applications and micro-benchmarks in this study are parallelized using OpenMP, we can estimate an upper bound for the number of threads to be used productively.

We use the OpenMP micro-benchmark suite (version 3.X) from EPCC to quantify these overheads [8]. The *overhead* $\tau(n)$ at n threads is here defined as the difference in execution time between expected and measured timings. We execute independent workloads, essentially empty loops, for a given number of iterations and time the execution $t(n)$ with n threads

$$\tau(n) = t(n) - \frac{t_0}{n}$$

where t_0 is the timing of serial execution of the same workload. The whole measurement is repeated to achieve a representative result.

The central component is the GNU OpenMP runtime shipped with GCC 4.8.2 and its interaction with the test-system. Relevant environment variables for distributing threads over cores and tuning thread migration and waiting policy are tuned for performance as described in section 3.3.

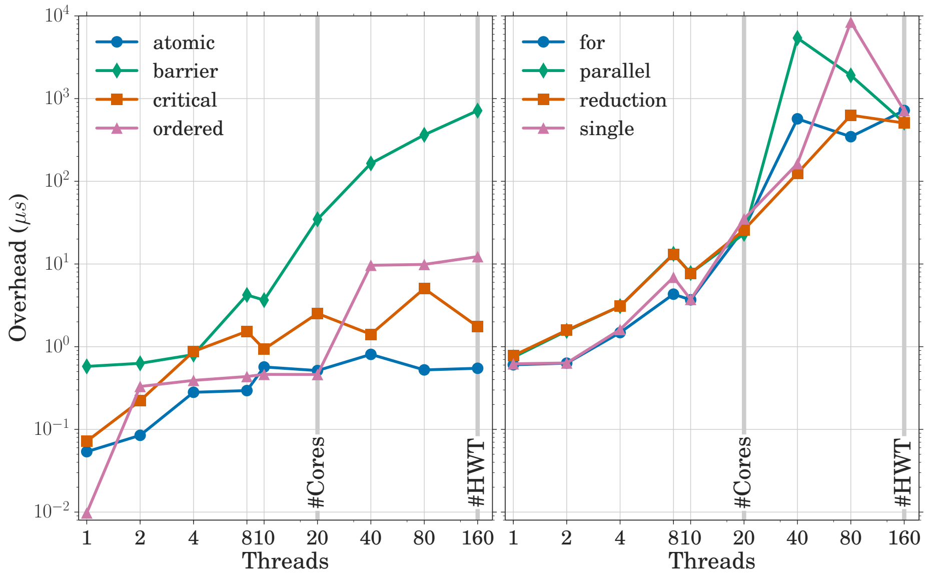


Fig. 4: Overheads for OpenMP threading as measured by the EPCC suite v3.X. The baseline cost setting up worksharing constructs and synchronization is well below one micro-second.
Left: Explicit synchronization constructs. The most expensive statement to use is `barrier`, consuming up to $0.75ms$ on 160 threads.
Right: Implicit synchronization by closing a parallel region and the overhead of various worksharing constructs. Apart from a few outliers, we observe overheads of $0.6\mu s$ to $0.7ms$.

Fig 4 summarizes our findings for the impact of various explicit and implicit synchronization constructs. These are the relevant sources of overhead for the further workloads in this report. The large overhead at the maximum number of 160 threads of around a millisecond suggests that using this level of concurrency will generally not be beneficial for worksharing. Regarding explicit synchronization, using `atomic` sections is to be preferred over the alternatives.

5 Application Performance Results

We present results on the analysis of three scientific applications on the POWER8 architecture: Lattice Boltzmann, MAFIA and NEST. The applications cover a wide scientific field: fluid dynamics (LB), data analysis (MAFIA) and neuronal networks (NEST). Furthermore, their performance profiles are diverse and gives good coverage of the architectural features.

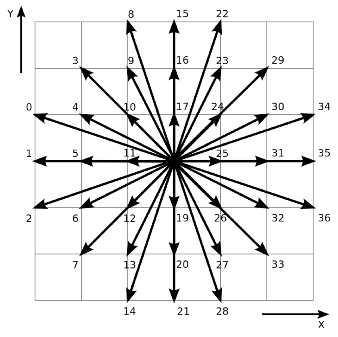


Fig. 5: The 37 element stencil for the `propagate` function.

5.1 Lattice Boltzmann Performance Results

The Lattice Boltzmann (LB) method is widely used in computational fluid dynamics, to numerically solve the equation of motion of flows in two and three dimensions. While conceptually less efficient than spectral methods, LB approaches are able to handle complex and irregular geometries as well as complex and multi-phase flows. From a computational point of view, LB methods are "easy" to implement and a large degree of parallelism is exposed.

LB methods (see, e.g., [9] for an introduction) are discrete in position and momentum spaces; they are based on the synthetic dynamics of *populations* located at the sites of a discrete lattice. At each time step, populations are *propagated* from lattice-site to lattice-site and then incoming populations *collide* among one another, that is, they mix and their values change accordingly.

LB models in x dimensions with y populations are labeled as $DxQy$. Here, we consider the $D2Q37$ a state-of-the-art bi-dimensional model with 37 populations per site, see fig 5, that correctly reproduces the thermo-hydrodynamical equations of motion of a fluid in two dimensions and automatically enforces the equation of state for an ideal gas ($p = \rho T$) [10, 11].

From a computational point of view the most relevant steps performed by a LB simulations are the computation of the `propagate` and `collide` functions:

1. `propagate` moves populations across lattice sites according to a stencil extent pattern of 7×7 excluding corners; it collects at each site all populations that will interact at the next phase: `collide`. Implementation-wise, `propagate` moves blocks of memory locations allocated at sparse memory addresses, corresponding to populations of neighbor cells.
2. `collide` performs all the mathematical steps associated to the computation of the collisional function, and computes the population values at each lattice site at the new time step. Input data for this phase are the populations gathered by the previous `propagate` phase.

We stress again that the D2Q37 LB method correctly and consistently describes the thermo-hydrodynamical equations of motion as well as the equation of state

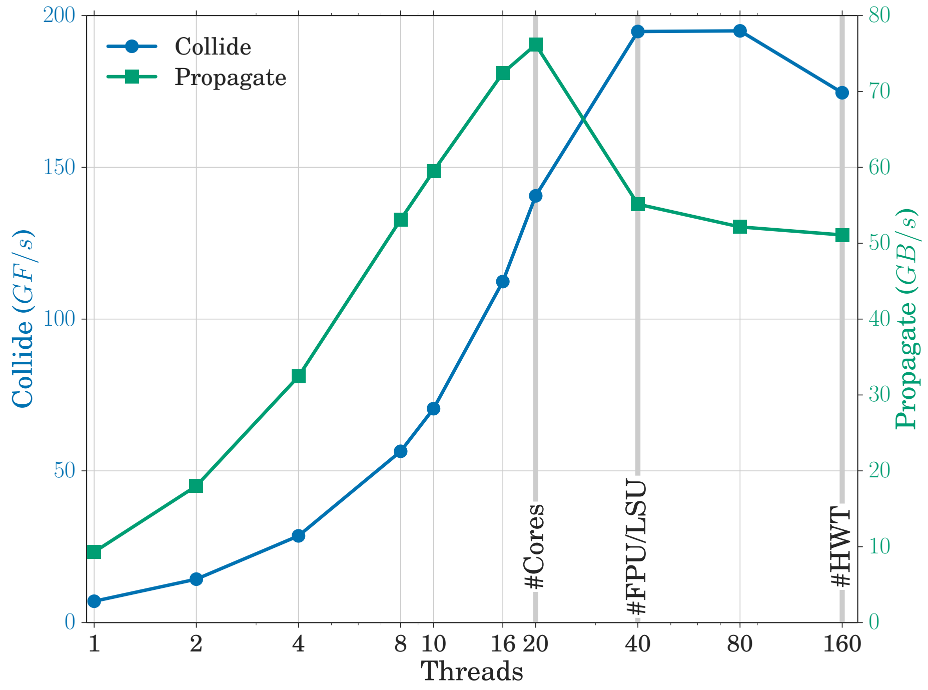


Fig. 6: Performance of the *D2Q37* kernels over the number of threads used. The efficiency of `collide` is given in GF/s and `propagate` measured in GB/s.

of a perfect gas; the price to pay is that, from a computational point of view, its implementation is more complex than simpler LB models. This translates into severe requirements in terms of memory bandwidth and floating-point throughput. Indeed, `propagate` implies accessing 37 neighbor cells to gather all populations; this step is mainly memory-bound and takes approximately 10% of the total run-time. The `collide` requires approximately 7600 double-precision floating point operations per lattice point, some of which can be optimized away by the compiler, see later sub-section. `collide` exhibits a significant arithmetic intensity and is the dominating part of the overall computation, taking roughly 90% of the total run-time. All tests have been performed on a lattice of 3200×2000 sites, corresponding to $1.76 GiB$ of data per lattice. Input/output, diagnostics and computation of boundary conditions are not accounted for in this benchmark.

The D2Q37 model is highly adaptable and has been implemented on a wide range of parallel machines like BG/Q [12] as well as on a cluster of nodes based on commodity CPUs [13], GPGPUs [14] and Xeon-Phi [15]. It has been extensively used for large scale simulations of convective turbulence (see e.g., [16, 17]). These implementations have been extensively tuned for the hardware in question, which is beyond scope of this study.

The `collide`-operation consists of three phases, first computing the moments of the distribution function, then resolving the collision effects in terms of these moments and those of the equilibrium distribution and finally transforming the result back. All transformations are linear. The GCC compiler generates optimized code with 4550 instructions. The main optimization is unrolling of each of the three loops over 37 into one single iteration and 18 iterations with vectorized load and FP instructions. All results for the hardware counter analysis are given per lattice site, corresponding to 37 elements of $64b$ floating point data. For a break-down of the instruction mix and pipeline filling refer to table 3. We find that the required 6200 floating point operations are performed in 2100 instructions, which are mostly vectorized fused-multiply-add instructions. Address calculation and loop variables contribute roughly 860 fixed-point instructions. There are just above 1500 load instructions plus close to 100 store instructions, in addition to the input data of $37 \cdot 8B$ we have to read some constants, but the bulk of this overhead stems from spilling the working set to L1. The actual amount of data read from memory is $459B$, roughly 50% more than the 37 populations. The additional traffic may be explained by the coefficients for the polynomial expansions and the data that is prefetched but cast-out of L3 before it is used and re-read later. This is supported by the fact that almost all incoming memory traffic is due to pre-fetches ($457B$). The function stores $37 \cdot 8B = 296B$, which is the updated site data, into memory. The full operation take 3000 cycles per site.

A thread scaling analysis of `collide` in Fig. 6 shows that the peak performance of 194 GF/s is reached with 80 threads on 20 cores, i.e. 9.7 GF/s per core in `SMT4` mode, closely followed by 9.65 GF/s in `SMT2` mode. It is interesting to see that further oversubscription of the core (160 threads, `SMT8`) reduces the performance by 11 % compared to the maximum. The performance of a single thread per core is reported as 7 GF/s. The POWER8 core architecture is optimized for both single threaded execution (ST) as well as SMT threading; it adapts the way instruction dispatch works accordingly. This explains the subtle differences in the interplay of functional units that can be observed in the two modes. The peak performance is about 74 % higher than running on a single thread per core (ST mode). This gain stems from better filling of the instruction pipelines.

`Propagate` performs a swap on 37 memory locations per lattice site and is, therefore, limited by the effective random access memory bandwidth. Benchmarking with thread numbers between 1 and 160 shows that the shortest runtime of `propagate` is reached at 20 threads, i.e. one thread per core (ST mode).

The structure of memory access leads to a factor 3 to 4 lower bandwidth compared to values for the sustained bandwidth obtained with the STREAM benchmark 4.2. Here, we can see that the throughput of instructions is highest in ST mode and, similar to the performance of the `collide`-kernel, degrades the more threads we use per core. The lower part of the scaling analysis – 1 to 20 threads – shows the effect of shared resources. The original version of the code exhibited less than optimal performance due to misuse of the cache hierarchy. The loop over the lattice was optimized by using cache blocking, giving a gain of 20% in bandwidth. Again, table 3 shows a detailed breakdown of the instruction mix

			Throughput			
Function	**Unit**	**Fraction**	ST	SMT2	SMT4	SMT8
`collide`	`LSU`	0.35	0.28	0.42	0.43	0.38
	`VSU`	0.50	0.41	0.57	0.57	0.51
	`FXU`	0.14	0.11	0.13	0.12	0.11
`propagate`	`LSU`	0.45	0.04	0.04	0.05	0.04
	`VSU`	0.41	0.04	0.06	0.12	0.05
	`FXU`	0.11	0.01	0.02	0.05	0.03

Metric	`collide`	`propagate`
t_{wc}	3007	214
N_x	4933	87
N_{FX}	857	12
N_{FP}	2122	–
N_{BR}	80	2
N_{LS}	1641	74
N_{mem}	755	2141
N_{fp-op}	6197	–

Table 3: **Left:** Characteristics of LBM seen by the instruction pipelines; measured on a single thread. Given are the relative fractions dispatched to the pipelines and the throughput relative to the maximum for different numbers of threads per core.
Right: Instruction counts and general metrics for the LBM application.

and pipeline filling. No floating point operations are performed, although the `VSU` pipelines report significant filling, since store instructions are executed in both `LSU` and `VSU`. We find exactly 37 load and store instructions, one per population. These result in $296B$ of write traffic and $1837B$ are read from memory. This is roughly six times more than we would naively expect. Again, almost every incoming byte is due to pre-fetching ($1537B$), indicating that streams may be established, but never fully utilized. Further, due to the nature of the stencil the read accesses are not continuous, potentially resulting partially consumed cache lines. Addressing and loop computations result in 12 fixed point computations per site. Processing a single site requires 214 cycles.

In summary, LBM is split into two parts, both of which have completely different performance requirements. The computationally expensive `collide` operation, which is largely vectorized by the compiler, reaches about 29% of the peak floating point performance. It further benefits from the higher pipeline filling by using up to four threads per core. On the other hand `propagate` which is purely memory bound can capitalize roughly 20% of the aggregated read/write bandwidth. However, the maximum *achievable bandwidth* is $256GB/s$ as the requirements of `propagate` are symmetric in read and write. Of this figure, we can exploit close to 30%, the remaining gap is mainly a result of the non-continuous access pattern.

The overall performance is summarized in fig. 6, the maximum achieved for 40 threads or two per core, which is due to the unequal shares of both phases on the total runtime. We find close to ideal scaling up to twenty threads and significant gains from using two threads per core, beyond that, performance stagnates (SMT4) and finally degrades (SMT8). This is expected as the application utilizes the available pipelines efficiently at SMT2. Gains from filling potential voids in the pipelines are offset by threading overheads as described in section 4.3. We are investigating the reduced bandwidth at more than one thread per core.

5.2 MAFIA Performance Results

MAFIA is a subspace clustering application [18] which implements the algorithm of the same name [19]. For the purpose of this report, we concentrate on the CPU version implemented using OpenMP. MAFIA algorithm builds *dense units (DUs)* starting from lower dimensionalities and progressing to higher ones, until no new DUs can be built. For each dimensionality, it generates *candidate DUs (CDUs)* from DUs of lower dimensionality. The CDU is accepted as a DU if the number of contained points lies above a certain threshold. The cardinality computation, **pcount**, is the most computationally intensive kernel of the algorithm. The generated DUs are then merged into clusters, which are the final output.

In MAFIA, each CDU is represented as a cartesian product of *windows*, one window per dimension of the CDU. Each window, in turn, is represented by a set of contained points, implemented as a bit array. Thus, the number of points inside a CDU can be computed as the number of bits set in intersection (bitwise AND) of all windows. The loop over words of the bit array was strip-mined, so that auto-vectorization by the compiler is possible.

[18] presents performance estimates as well as an empirical analysis of MAFIA. Assume that the algorithm runs on n points in d dimensions, and the dataset contains a single hidden cluster of dimensionality k. Then the total number of logical bit AND operations in **pcount** kernel for the entire program run is given by the equation

$$N_{bitops} = n \cdot k \cdot 2^{k-1} . \tag{1}$$

As the number of windows is several orders of magnitude smaller than the number of points ($\mathcal{O}(10)$ versus $\mathcal{O}(10^6)$), it can be assumed that the array of window indices is cached, and only bit arrays need to be transferred from memory. Thus, equation 1 also gives the number of bits transferred from memory by the **pcount** kernel.

We started by analyzing the scaling behavior of MAFIA with different OpenMP thread placements, by altering the stride s with which the threads are spread out across cores. The **pcount** kernel was parallelized across CDUs, with only a single thread executing the point count loop for each CDU. Scalability results for the **pcount** kernel for a dataset with $n = 10^7$ points of dimensionality $d = 20$ and a cluster of dimensionality $k = 14$ are presented in Fig. 7. The scalability is quite good, with a speedup of up to 25 achieved with 80 threads and threads allocated round-robin to every second core, see section 3.

We then proceeded to analyzing counter values. The MAFIA application was run with 20, 40, 80 and 160 threads with the same point and cluster dimensionalities as above ($k = 14$, $d = 20$). The number of points, n, varied on a logarithmic scale between $1 \cdot 10^6$ to $64 \cdot 10^6$. Counter values are given as averages across three runs.

Next, we analyze vector instruction throughput. As MAFIA **pcount** contains only integer vector instructions, counters for floating-point vector instructions are of no interest here. For each of the three counters, we assume that its value

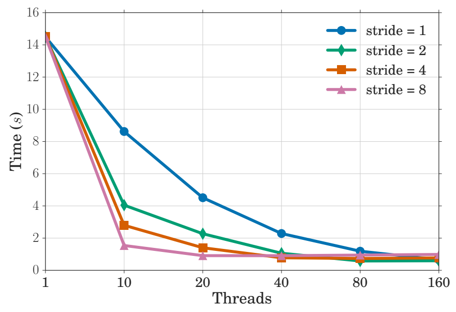


Fig. 7: MAFIA OpenMP scaling

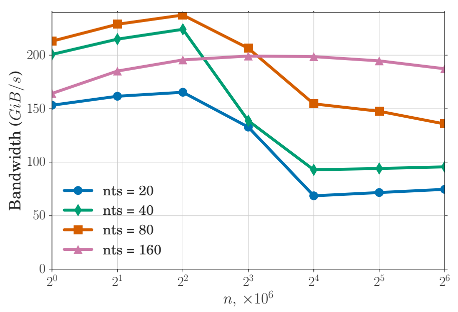


Fig. 8: Theoretical memory bandwidth for MAFIA

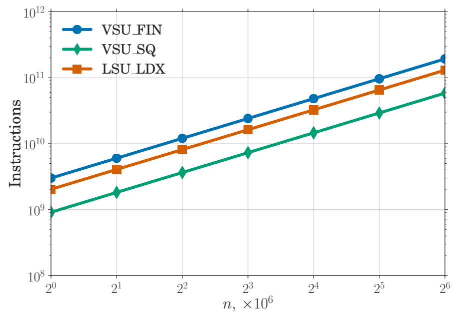


Fig. 9: Predicted and measured values for vector instruction counters for 20 threads

	Assembly		**Expected**	
Counter	c_0	c_1	c_0	c_1
`VSU_SQ`	0.125	1.0	0	0
`VSU_FIN`	2.5	3.0	1	1
`LSU_LDX`	1	2.125	0	1

Table 4: Coefficients for vector counters, both expected based on the code and actual extracted from the assembly

can be modeled by the equation

$$c(n) = (c_0 2^k + c_1 k 2^{k-1}) \cdot \frac{n}{128} \tag{2}$$

where the coefficients c_0 and c_1 are both in terms of operations performed on a single vector. The term with c_1 is derived from equation 1, and corresponds to the loop over windows, where the number of iterations varies with CDU dimensionality. The term with c_0 corresponds to the rest of the iteration of the loop over words, where the number of instructions executed does not depend on CDU dimensionality.

For prediction purposes, we derived values for c_0 and c_1 from the assembly code generated by the compiler. Their values for different counters are listed in table 4. The innermost loop was unrolled by the compiler, therefore some coefficients have a fractional part. Fig. 9 compares predictions with the actual measured values. The predictions are almost perfect, with less than 0.001% difference. Numbers for other thread counts are very similar and are omitted for brevity.

It is also worth comparing coefficients extracted from assembly to the minimum values expected by looking at the original code; both are listed in Table 4.

- One store instruction is executed per vector instead of none expected. This indicates that the storage for words resulting from logical AND operation is in L1 rather than registers.
- Similarly, there are 2.125 load instructions instead of one expected. One of those is needed to load the array holding result of logical AND to registers (from cache), and 0.125 is due to imperfect alignment of bit arrays in main memory.
- Three vector instructions are generated instead of one expected. One is due to vector store counted as a vector instruction, and the second is a permutation instruction, again to compensate for mis-alignment of the bit arrays in memory.

The values obtained from the assembly differ from minimum values expected from the original code, which indicates optimization potential. Compiler optimization is one of the way to address that, and we are planning to look into that.

We then proceeded with analyzing the memory traffic. Fig. 8 plots a semi-empirical memory bandwidth, i.e. the estimate of memory traffic divided by measured running time. For a given number of points, more bandwidth actually indicates *lower running times*, as the theoretical memory traffic does not depend on the number of threads. Fig. 10 plots the ratio of traffic between main memory and various levels of caches to the theoretical value, derived from equation 1.

With 20 threads, L1/L2 caches and L3 cache partition of a single core are used by a single thread only, which gives the most predictable plot. Indeed, the amount of data flowing into L1 cache is very close to the theoretical prediction. L1 is mostly filled from L3, and data flow from L2 is almost non-existent. Up to and including $4 \cdot 10^6$ points, the aggregated size of the bit arrays fits into L3; only after that is data fetched from main memory. Even then, it is mostly prefetched into L3, from where it goes further up. Because of that, there is almost no need to fetch the data from main memory directly into L1.

The plots for 40 and 80 threads show the same qualitative behavior, although effects of cache sharing play a role. On the positive side, the same cache lines can be used by multiple threads; as a result, the amount of data loaded into L1 is actually *less* than the theoretical prediction, down to 50% for 80 threads. On the negative side, as the amount of cache of all levels per thread is lower, there is less space to store data on-chip. As a result, for 80 threads, data should be fetched from memory for all dataset sizes. However, this does not seem to affect performance, as the 80-thread version is actually the fastest for the cases when the data size fits into L3 cache. It may be that though the L3 prefetcher kicks in, it does not provide the data further referenced by the algorithm. For 20 to 80 threads, there is also a small but not insignificant amount of data retrieved from cache partitions of other cores.

The plot for 160 threads differs qualitatively from the others. First of all, there is significant over-prefetching of data from the main memory. We assume that

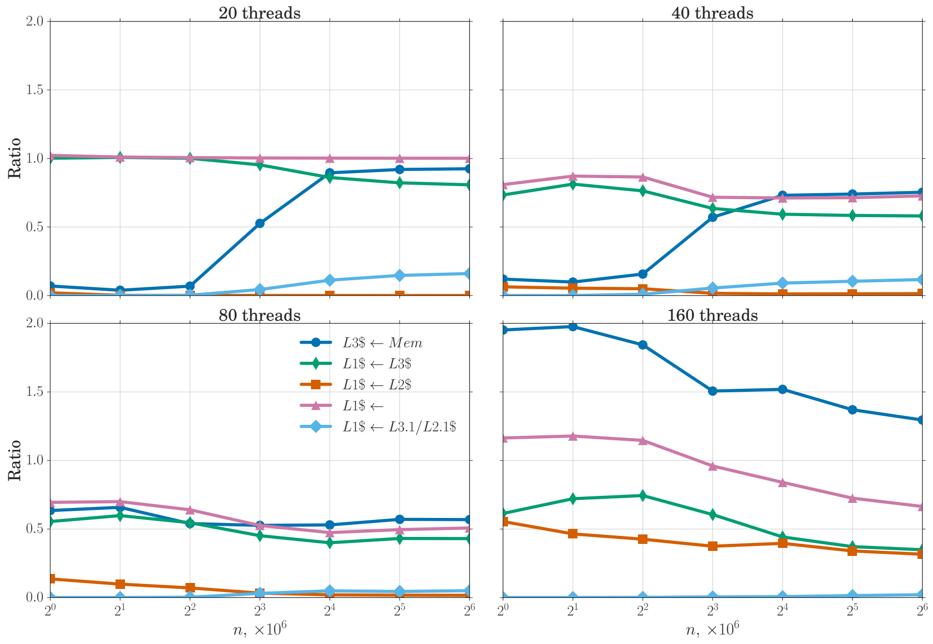


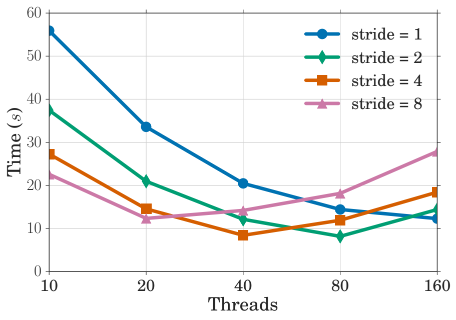
Fig. 10: Memory traffic ratio for various thread counts. Ratio for $L1\$ \leftarrow Mem$ is very close to zero and therefore not shown.

due to too many threads contending for prefetcher resources, prefetch streams get tried but do not reach steady state. Also, a much larger fraction of data is sourced from L2. Again, we assume that due to over-subscription of over-prefetching into L1, many of the prefetched L1 cache lines get cast out into L2 even before they get accessed. This agrees with other counters, which indicate that lines from L2 and L3 come due to explicit accesses and not due to prefeches. Nevertheless, overall use of hardware with 160 threads is relatively good, as for more than $8 \cdot 10^6$ points this is where the maximum performance is achieved.

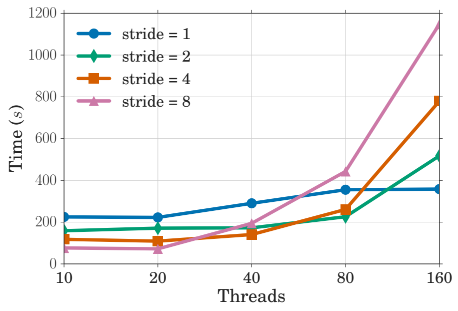
To summarize, MAFIA's **pcount** loop is rather regular, and we can get a good understanding of it. Vector instruction counters are perfectly understood in terms of algorithmic properties and instructions in the assembler code. Memory behavior is also understandable, particularly for lower number of threads $nts \leq 40$, where effects of L3 cache size are clearly visible. With larger number of threads, however, our understanding is limited, and it is here where the highest performance is achieved. We thus assume that the application is latency-limited, as neither instruction throughput nor memory bandwidth limit its performance.

5.3 NEST Performance Results

NEST (NEural Simulation Tool) is an application from the field of computational neurobiology [20]. It models brain tissue as a graph of neurons interconnected by



(a) 512 processes



(b) 16384 processes

Fig. 11: NEST scaling behavior with OpenMP

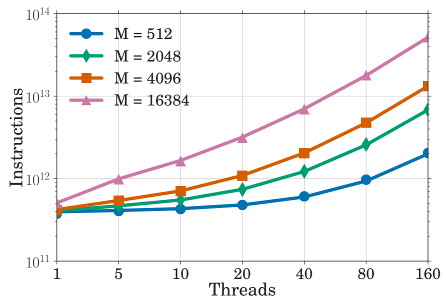
synapses. Neurons exchange spikes along the synapse connections. On an abstract level, NEST can be understood as a discrete event simulator on a distributed sparse graph. It is built as an interpreter of a domain specific modeling language on top of a C++ simulation core. Most simulations include stochastic connections and sources of spikes, which makes static analysis and load balancing unfeasible.

The performance profile of NEST leans towards fixed point operations due to the necessary graph operations and dynamic dispatch of events. Memory capacity is a major bottleneck for large-scale simulations with NEST, so optimizations tend to favor size over speed. Despite the obvious need for good fixed point performance, a small but non-negligible fraction of floating point operations is needed to update the neuron models.

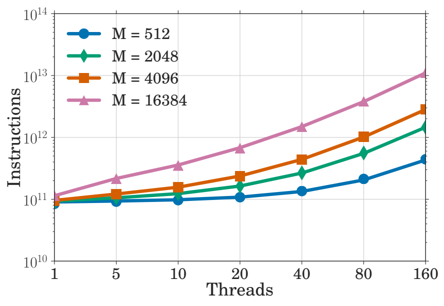
For our experiments, we used *dry run* mode of NEST. This enables simulating performance characteristics of a NEST run on many thousands of nodes by running on a single system. The parameters of a run are the simulated number of MPI processes, M, and the number of threads running on a single node, T. Typically, NEST run parameters also include n, the number of neurons owned by a single process, which is fixed at $n = 9375$, and therefore omitted, in our experiments. The total number of neurons is proportional to M. Each active thread is called a *virtual process (VP)* and the total number of VPs is given by $M \cdot T$. For our experiments, we simulate random balanced networks with nM neurons and both static and adaptive synapses [21].

We started with analyzing performance of NEST simulation loop with different OpenMP settings. We performed experiments with 10, 20, 40, 80 and 160 threads strided by 1, 2, 4 and 8 over the cores under 3 values of M. Results for $M = 512$ and $M = 16384$ processes are depicted in Fig. 11; results for $M = 4096$ (not shown) exhibit the same behavior.

NEST exhibits non-trivial scaling behavior. Some parts, such as *neuron update* or *synapse processing*, scale well, while others, such as *spike buffer processing*, do not scale, as all threads should go through the entire spike buffer. Moreover, with larger number of processes, and as a consequence, of neurons, the relative weight



(a) Instructions executed (N_x)



(b) Loads completed (`LD_CMPL`)

Fig. 12: Modeling values of NEST “work-related” counters

of spike buffer processing increases. Therefore, while with $M = 512$ having more threads per core improves performance to some extent, with $M = 16384$, more threads always means worse performance. For each T, the optimal stride is given by $min(160/T, 8)$, which we use for further experiments.

We then proceeded to analyzing hardware performance counters for NEST. Understanding how resource contention affects running times of various parts of NEST is still a work in progress. We therefore restrict ourselves to counters which can be characterized as *amount of work performed*, such as the number of instructions or loads executed. We analyzed only the *spike delivery* phase, as for a large number of processors, it takes more than 90% of simulation time. We performed experiments with $T = 1, 5, 10, 20, 40, 80, 160$ and $M = 512, 2048, 4096, 16384$. The work done in spike delivery phase can be broken into contrbutions from processing the following items:

– **synapses**, which is constant for fixed n;
– **spikes in the buffer**, proportional to $M \cdot T$, as the number of spikes is proportional to M, and this work has to be done by each thread;
– **markers in the buffer**, proportional to $M \cdot T^2$, as the number of markers is equal to the number of virtual processes and the work is done in each thread;

Note that the code for all components is intermixed, so it is impossible to accurately measure each of them without introducing significant measurement bias. The total amount of work done can be written as the sum of all components

$$C = c_0 + c_1 \cdot MT + c_2 \cdot MT^2 \, . \tag{3}$$

The coefficients of the equation 3 has been derived by fitting it into experimental data using least squares method. Fig. 12 plots values for both total instructions (N_x) and loads executed. The points represent the measured values, and the lines represent the fitted values. The fits are very close, with the deviation being less than 5.5% (mostly less than 3.3%) for instructions and less than 7.5% (most less than 3.7%) for loads. Note that equation 3 also holds for other work-related counters, which include: floating point loads and stores, vector instructions, both the total and actual arithmetic operations.

To summarize, though our understanding of NEST performance characteristics is far from complete, some points are clear. Specifically, spike delivery takes most of the time for simulations with large number of MPI processes M. We also understand the number of instructions executed by spike delivery, and it is clear that it contains parts that do not scale with either the number of processes M or threads T. And while scalability with T could be improved by parallelizing the loop processing spikes in the buffer, improving scalability with M requires more fundamental changes in NEST architecture, specifically the way spikes are exchanged between processes.

6 Summary and Conclusions

We presented the characterization of three different scientific codes on a new server-class processor, the POWER8. Further, results of micro-benchmarks were collected as a first impression of the performance characteristics.

The LBM and MAFIA applications benefit from the available instruction-level parallelism and vectorization capabilities. Although parts of LBM depend strongly on the memory bandwidth, the available capacity can only be exploited to a fraction, due to the access pattern. NEST is an irregular application limited by memory accesses, and could, in theory, benefit from SMT. However, in order to achieve this, its scalability should be improved first.

On the basis of the performance we were able to achieve in our tests, POWER8 is a candidate for the host CPU in GPU-accelerated systems. The focus on integer performance, out-of-order execution and memory bandwidth complement the floating-point optimized profile of the accelerator. Exploring this direction is planned for the near future.

With up to 160 threads in total or eight per core, overheads from thread management, especially by the OpenMP runtime, become an important factor. This is even more critical, as the SMT facilities are means to optimize pipeline filling and therefore require lightweight threading. However, the gains from these large numbers of threads per core are expected to be significant only if the pipelines are not sufficiently saturated to begin with. The applications we tested did not suffer from this problem, so speed-ups were not expected. We were not able to obtain results with an OpenMP runtime optimized for POWER8 in the time frame of the preview. This too, is planned for the near future.

References

1. Friedrich, J., Le, H., Starke, W., Stuechli, J., Sinharoy, B., Fluhr, E., Dreps, D., Zyuban, V., Still, G., Gonzalez, C., Hogenmiller, D., Malgioglio, F., Nett, R., Puri, R., Restle, P., Shan, D., Deniz, Z., Wendel, D., Ziegler, M., Victor, D.: The POWER8 processor: Designed for big data, analytics, and cloud environments. In: IEEE International Conference on IC Design Technology (ICICDT). (2014)

2. Fluhr, E., Friedrich, J., Dreps, D., Zyuban, V., Still, G., Gonzalez, C., Hall, A., Hogenmiller, D., Malgioglio, F., Nett, R., Paredes, J., Pille, J., Plass, D., Puri, R., Restle, P., Shan, D., Stawiasz, K., Deniz, Z., Wendel, D., Ziegler, M.: POWER8: A 12-core server-class processor in 22nm SOI with 7.6Tb/s off-chip bandwidth. In: Solid-State Circuits Conference Digest of Technical Papers (ISSCC), IEEE International. (2014)
3. Barker, K.J., Hoisie, A., Kerbyson, D.J.: An early performance analysis of POWER7-IH HPC systems. In: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis. SC '11, New York, NY, USA, ACM (2011)
4. Srinivas, M., Sinharoy, B., Eickemeyer, R., Raghavan, R., Kunkel, S., Chen, T., Maron, W., Flemming, D., Blanchard, A., Seshadri, P., Kellington, J., Mericas, A., Petruski, A.E., Indukuru, V.R., Reyes, S.: IBM POWER7 performance modeling, verification, and evaluation. IBM Journal of Research and Development **55**(3) (2011)
5. Browne, S., Dongarra, J., Garner, N., London, K., Mucci, P.: A Scalable Cross-platform Infrastructure for Application Performance Tuning Using Hardware Counters. (2000)
6. Baumeister, P.F., Boettiger, H., Hater, T., Knobloch, M., Maurer, T., Nobile, A., Pleiter, D., Vandenbergen, N.: Characterizing performance of applications on Blue Gene/Q. In Bader, M., Bode, A., Bungartz, H.J., Gerndt, M., Joubert, G.R., Peters, F.J., eds.: PARCO. Volume 25 of Advances in Parallel Computing., IOS Press (2013)
7. McCalpin, J.D.: STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia (1991-2007)
8. Bull, J.M., O'Neill, D.: A Microbenchmark Suite for OpenMP 2.0. SIGARCH Comput. Archit. News **29**(5) (December 2001) 41–48
9. Succi, S.: The Lattice-Boltzmann Equation. Oxford university press, Oxford (2001)
10. Sbragaglia, M., Benzi, R., Biferale, L., Chen, H., Shan, X., Succi, S.: Lattice Boltzmann method with self-consistent thermo-hydrodynamic equilibria. Journal of Fluid Mechanics **628** (2009) 299–309
11. Scagliarini, A., Biferale, L., Sbragaglia, M., Sugiyama, K., Toschi, F.: Lattice Boltzmann methods for thermal flows: Continuum limit and applications to compressible Rayleigh–Taylor systems. Physics of Fluids (1994-present) **22**(5) (2010) 055101
12. Pivanti, M., Mantovani, F., Schifano, S., Tripiccione, R., Zenesini, L.: An Optimized Lattice Boltzmann Code for BlueGene/Q. In Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J., eds.: Parallel Processing and Applied Mathematics. Lecture Notes in Computer Science. Springer Berlin Heidelberg (2014)
13. Biferale, L., Mantovani, F., Pivanti, M., Pozzati, F., Sbragaglia, M., Scagliarini, A., Schifano, S.F., Toschi, F., Tripiccione, R.: Optimization of Multi-Phase Compressible Lattice Boltzmann Codes on Massively Parallel Multi-Core Systems. Procedia Computer Science **4**(0) (2011) Proceedings of the International Conference on Computational Science, ICCS 2011.
14. Kraus, J., Pivanti, M., Schifano, S.F., Tripiccione, R., Zanella, M.: Benchmarking GPUs with a parallel Lattice-Boltzmann code. In: Computer Architecture and High Performance Computing (SBAC-PAD), 2013 25th International Symposium on, IEEE (2013)
15. Crimi, G., Mantovani, F., Pivanti, M., Schifano, S.F., Tripiccione, R.: Early Experience on Porting and Running a Lattice Boltzmann Code on the Xeon-phi Co-Processor. Procedia Computer Science **18** (2013)

16. Biferale, L., Mantovani, F., Sbragaglia, M., Scagliarini, A., Toschi, F., Tripiccione, R.: Second-order closure in stratified turbulence: Simulations and modeling of bulk and entrainment regions. Physical Review E **84**(1) (2011) 016305
17. Biferale, L., Mantovani, F., Sbragaglia, M., Scagliarini, A., Toschi, F., Tripiccione, R.: Reactive Rayleigh-Taylor systems: Front propagation and non-stationarity. EPL (Europhysics Letters) **94**(5) (2011)
18. Adinetz, A., Kraus, J., Meinke, J., Pleiter, D.: GPUMAFIA: Efficient subspace clustering with MAFIA on GPUs. In Wolf, F., Mohr, B., an Mey, D., eds.: Euro-Par 2013 Parallel Processing. Volume 8097 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2013)
19. Nagesh, H., Goil, S., Choudhary, A.: Parallel Algorithms For Clustering High-Dimensional Large-Scale Datasets. Kluwer (2001)
20. Gewaltig, M.O., Diesmann, M.: NEST (NEural Simulation Tool). Scholarpedia **2**(4) (2007)
21. Morrison, A., Aertsen, A., Diesmann, M.: Spike-timing-dependent plasticity in balanced random networks. Neural computation **19**(6) (2007) 1437–1467