Lessons Learnt Porting Parallelisation Techniques for Irregular Codes to NUMA Systems

Juan A. Lorenzo*, Juan C. Pichel†, David LaFrance-Linden‡, Francisco F. Rivera* and David E. Singh §
*Computer Architecture Group, Electronics and Computer Science Dept., Univ. of Santiago de Compostela, Spain
Email: {juanangel.lorenzo,ff.rivera}@usc.es
†Galicia Supercomputing Centre, Santiago de Compostela, Spain
‡HP Labs, USA
§Dept. of Informatics, Carlos III University, Madrid, Spain

***Abstract*—This work presents a study undertaken to characterise the behaviour of some parallelisation techniques for irregular codes, previously developed for SMP architectures, on a several-node SMP NUMA system. The main objective is to determine the performance effect of bus contention and cache coherency in such a complex architecture. Results show that: (1) cores which share a socket can be considered as independent processors in this context; (2) for big data sizes, the effect of sharing a bus degrades the performance but masks the cache coherency effects and (3) the NUMA-ratio is a critical factor on irregular codes. These results allow us to study the effect in performance of the thread-to-core mappings and memory allocation policies.**
***Keywords*-Irregular Codes, Itanium2, Hardware Counters.**

I. Introduction

Irregular codes [1] are the core of many important scientific applications, so there exist several widespread techniques to parallelise them [1], [2], [3]. Some of the strategies to parallelise irregular codes were designed to work in SMP systems, where the memory access latency is the same regardless of the processor involved. However, state-of-the-art architectures involve many cache levels in *Non-Uniform Memory Access* (NUMA) configurations containing multi-core processors.

In this context, the memory allocation and the thread-to-core distribution may become very important in the performance of a generic code and, more noticeably, in strategies to parallelise irregular codes. These techniques reorder the data to take the maximum advantage of the memory hierarchy.

In a recent work, a framework for automatic detection and application of the best mapping among threads and cores in parallel applications on multi-core systems was presented [4]. Likewise, Williams et al. [5] propose several optimisation techniques for the sparse matrix-vector multiplication which are evaluated on different multi-core platforms. Authors examine a wide variety of techniques including the influence of the process and memory affinity. Regarding memory allocation, Norden et al. [6] study the co–location of threads and data motivated by the non-uniformity of memory in NUMA multi-processors.

The main objective of this work is to characterise the behaviour of some of the aforementioned techniques in a NUMA environment considering the influence of thread and memory allocations.

II. Parallelisation Techniques for Irregular Codes

The chosen techniques were *SPRT* (*Sorted Private Region Technique*) [7], *Array Expansion* [8] and *DWA-LIP* [3]. *SPRT* is a method based on the characterisation of the memory access pattern. The low-cost, compact characterisation of the subscript arrays can be used to perform an efficient parallelisation of the irregular code. *Array Expansion* is based on the privatisation of some variables in which writings are done. However, instead of making a private copy of the reduction array, this is expanded in an additional dimension with as many elements as processors. Finally, *DWA-LIP* aims to increase the data access locality. In this method, computations are based on grouping loop iterations into sets that are assigned to cooperating threads.

It must be pointed out that this work does not intend to compare these three techniques, study which was already carried out in [7]. The objective of this work is to study the influence of thread and memory allocations in a given test architecture: the HP Integrity rx7640.

III. Methodology

To test the efficiency of the previous techniques two well known irregular benchmarks (*Sparse matrix-Dense vector Product* and *Irregular Reduction*) were implemented in Fortran and parallelised using OpenMP. The pseudocodes are shown in Figures 1 and 2, where `N` is the matrix size and `NNZ` is the number of non-zeros. In both cases some selected sparse matrices stored in CSC format from the *Harwell-Boeing Sparse Matrix Collection* [9] were used as input data. The key features of the considered sparse matrices are shown in Table I.

Table I
Squared-Matrix input set used in our tests.

Name	N	NNZ	**Description**
s3dkq4m2	90451	4820892	FEM, cylindrical shell
3dtube	45330	3213618	3-D pressure tube
nasasrb	54872	2677324	Shuttle rocket booster
struct3	53570	1173694	Finite element
bcsstk29	13992	619488	Model of a 767 rear bulkhead
bcsstk17	10973	428650	Elevated pressure vessel

for $j = 1$ to N **do**
  for $k = col(j)$ to $col(j+1) - 1$ **do**
    $Z(row(k)) = Z(row(k)) + val(k) * X(j)$
  end for
end for

Figure 1. Sparse Matrix-Vector product pseudocode

for $j = 1$ to NNZ **do**
  $Z(row(j)) = Z(row(j)) + alpha$
end for

Figure 2. Irregular reduction pseudocode

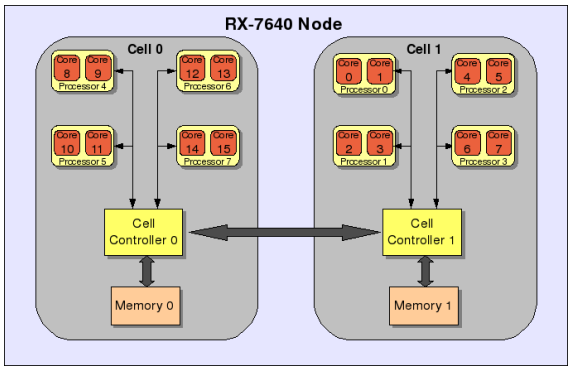


Figure 3. Block diagram of an HP Integrity rx7640 node.

A. The Test Platform

The target architecture where these techniques were tested is the *Finisterrae* supercomputer [10] installed at CESGA (*Galicia Supercomputing Centre*). *Finisterrae* is a SMP NUMA machine which comprises 142 HP Integrity rx7640 NUMA computation nodes. Each node consists of eight 1.6-Ghz-DualCore Intel Itanium2 Montvale (9140N) processors arranged in a two-SMP-cell NUMA configuration. Figure 3 shows the block diagram of a node as well as its core disposition. As seen, a cell is composed of two buses at 6.8 GB/s, each connecting two sockets (four cores) to a 64GB memory through a *sx2000 chipset* (Cell Controller). The Cell Controller maintains a cache-coherent memory system using a directory-based memory controller and connects both cells through a 27.3 GB/s crossbar. It yields a theoretical processor bandwidth of 13.6 GB/s and a memory bandwidth of 16 GB/s (four buses at 4 GB/s).

The main memory address range handled by the cell controller is split in two modes: three fourths of the address range map to the local memory, the remaining one fourth maps in an interleaved manner to addresses in both local and remote memory modules. This *interleaving zone* means that each address is spread, one by one, between the local and remote memories. Each processor comprises two cores and three cache levels per core. L1I and L1D (write-through, 16KB) are both a 4-way set-associative, 64-byte line cache. FP operations bypass the L1. L2I (1MB) and L2D (256KB) are 8-way set-associative, 128-byte line (both write-back). There exists a single L3 (write-back, 9MB) cache per core. The peak performance per core is 6.4 GFLOPS.

B. Tests Undertaken

The presented architecture shows a several-memory-layer platform where each NUMA node's cell must behave as a SMP. Our study is focused on quantifying the behaviour of such techniques on a *Finisterrae* node taking into account the data allocation, the memory latency and the thread-to-core assignment. To carry out the tests, an allocation scenario was set-up to allocate all data in a local cell memory, using only cores from the same cell with several thread-to-core assignments.

All outcomes were collected with *PAPI* [11]. The compiler used for the experiments was *Intel ifort 10.1*. The baseline compiler configuration used the `-O3` optimisation option and the interprocedural optimisation (`-ipo`). To allocate the data in a given memory module, the *libnuma* library and its command-line tool *numactl*, were used.

IV. Evaluation of Irregular Codes

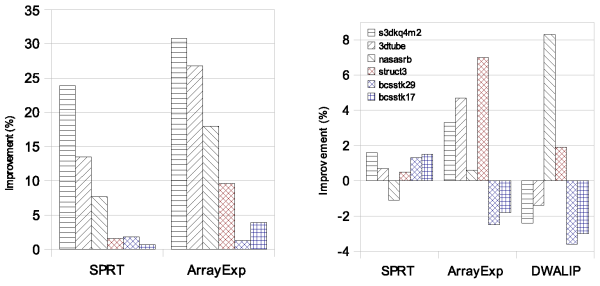
This section evaluates the parallelisation techniques for irregular codes taking into account the influence of the thread allocation in the bus contention and cache coherency. In this paper, only results for 4 cores are shown as representative case. Results for different numbers of cores are similar.

A. Influence of Thread Allocation in Bus Contention

In our tests, all data were allocated in Cell 0's memory, and only cores in that cell were used. Several scenarios were set up to study the effect of running the benchmarks with different distributions, in order to quantify the effect of the bus sharing. The distribution and observed results are as follows:

Cores that share the bus vs. cores connected to different busses (15-11-13-9 vs 15-14-13-12): In both benchmarks the sparse matrix is completely read, which is expected to cause some number of capacity and compulsory cache misses. A degradation in the performance is expected for the biggest matrices when sharing the bus. At the sight of the results (Figure 4), a higher improvement is obtained for the biggest matrices (*3dtube*, *nasasrb* and *s3dkq4m2*) when using two busses. Indeed, the bigger the matrix, the higher the traffic in the bus and the higher the improvement when there are only two cores per bus (15-11 and 13-9) instead of four threads in a single bus (15-14-13-12). It is noticeable that some important improvements occur only in the *SpMV* benchmark. This behaviour can be justified because *SpMV* executes a loop which goes through five different arrays, whereas *Irregular Reduction* goes only through two, so the cache reuse for the first benchmark is smaller and the generated traffic in the bus is higher.

Cores in different sockets sharing the bus vs. cores that share socket and bus (15-11-13-9 vs 15-14-11-10): In this case there were no noticeable differences for the SPRT between the case when the threads are allocated



(a) SpMV (b) Irregular Reduction

Figure 4. Improvement using different busses.

in different sockets and the case when two threads in the same bus share the socket. However, the *Irregular Reduction* benchmark showed, in the *Array Expansion* and *DWA-LIP* techniques, that some differences appear between the two considered cases. In particular, *Array Expansion* showed a bigger difference when the matrix size decreases, since a matrix not big enough does not generate enough traffic in the bus to mask the cache coherency effects. This suggest that some dependencies among threads could exist.

B. Influence of Cache Coherency

Taking into account that irregular codes strive for maximising the thread locality, we do not expect important performance decreases due to the cache coherency protocols when the input matrices are big enough, since they will be masked by the bus traffic. However, when all cores share the bus a improvement was noticed for some of the smallest matrices. If the matrices fit in each core's cache all misses, except the capacity ones, will be solved by the snooping protocol inside the bus, faster than asking the directory. This is an explanation to this behaviour.

V. Benchmarking the Test Platform

In order to confirm the hypothesis drawn when running irregular codes, some dense benchmarks were also executed in *Finisterrae* to analyse the effect of bus contention, memory allocation and cache coherency.

A. Influence of Memory Allocation

In this section an experiment was carried out to compare the theoretical memory latency given by the manufacturer [12] to our observations. We measured the memory access latency of a small program in Fortran which creates an array and allocates data on it. The measurements were carried out using *pfmon* and the Itanium2's *Event Address Registers* (EAR), which get accurately the memory position and access latency of a given sample.

In our tests, we allocated the data in the local core memory, in a remote memory and in the interleaving zone. In all cases, most of the occurrences appear under 50 cycles, corresponding with the accessed data which fit in cache memory. Furthermore, occurrences between 289 and 383 cycles were measured when accessing the cell local memory. As the frequency of our processors is 1.6 Ghz, it yields a latency from 181 to 240ns. The value given by the manufacturer is 185ns.

When accessing data in a remote memory we measured occurrences between 487 and 575 cycles, that is, from 305 to 360ns. The manufacturer does not give any values in this case. In the case of accessing data in the interleaving zone, the manufacturer's value is 249ns. Our measurements give two zones, depending on the local or remote memory the data are accessed. Indeed, the average access time in the interleaving zone is the average of combining accesses to the local or remote memory. Our outcomes gave an average value of 278ns.

We can conclude that, when working with codes mapped to cores in a same cell (especially for those with a high level of cache replacement), the data should be allocated in the same cell's memory. The access to a remote memory is very costly, so if cores in both cells must be used, the allocation of the data in the interleaving memory seems to be a good compromise.

B. Influence of Thread Allocation in Bus Contention

The second issue under study was the influence of the thread allocation upon the node buses. Considering that every four cores share a bus, it is reasonably foreseeable that any allocation which spreads out the threads as much as possible through the different buses would get a better performance than another one which maps several threads in cores of the same bus.

To quantify this effect, a benchmark called *memtest* [13] was used. This benchmark focuses on how multiple cores compete for the memory bandwidth. It allocates a given-sized, private block of memory per core filled with a randomly linked pointer trail and goes through it reading and writing the data, which creates traffic associated to the read data and, subsequently, written-back to memory. To quantify the effect of sharing a bus, several configurations comprising different thread allocations were used. All data were allocated in Cell 0. One thread was always mapped to Core 8 (see Figure 3). The other one was mapped either to Core 9 (same processor, same bus), Core 10 (different processor, same bus), Core 12 (different processor, different bus) or Core 14 (different processor, different bus). Additionally, tests between Core 8 and some cores in Cell 1 (cores 0, 2, 4 and 6) were also performed to quantify the effect of using two cores not sharing any resources in a cell. Finally, for comparison purposes, a test mapping just one thread to Core 8 was also carried out.

Table II quantifies the outcomes of those configurations for memory blocks of 10KB, 9MB, 1GB and 10GB in clock ticks per memory access. The results for 10KB show that, regardless of the pair of cores involved in Cell 0, the number of clocks to access the data is the same (3.5 ticks). As expected, for a data block small enough to fit into L1 there is almost no traffic in the bus and, therefore, no performance differences are observed. When the second core belongs to Cell 1 the time to access the data is also almost constant (3.9-4.0) and higher, as expected because all data are still allocated in Core 0.

Table II
MEDIAN MEMORY ACCESS LATENCY (IN TICKS) OF *memtest* BENCHMARK FOR DIFFERENT CONFIGURATIONS.

Cell	Processors	Memory allocated			
		10KB	64MB	1GB	10GB
Cell 1	8 - 0	4,0	338,6	349,8	532,7
	8 - 2	3,9	338,8	349,6	534,5
	8 - 4	4,0	338,4	349,6	532,6
	8 - 6	3,9	338,6	349,6	532,8
	8	3,5	329,0	340,6	525,7
Cell 0	8 - 9	3,5	352,9	366,4	546,2
	8 - 10	3,5	354,3	366,0	550,2
	8 - 12	3,5	342,3	353,7	539,3
	8 - 14	3,5	342,2	353,6	537,4

When the size of the block is higher than the L3 size (64MB and 1GB) three cases were detected. On Cell 0, the lowest average latency access occurs when Core 8 is alone in the bus. A second case with the highest latency access appears when Core 8 shares the bus with a core in the same processor (Core 9) or in a different socket but in the same bus (Core 10). Indeed, a data size large enough not to fit into cache can generate enough traffic in the bus to decrease the performance when both cores compete for it. The third case appears when two cores access memory from different buses (Cores 12 and 14). In this case, the performance decrease is not as important. Taking into account that the latency is not as low as the one-core case and that the throughput in the Cell Controller-to-memory bus is the same regardless of the pair of cores used, we conclude that the Cell Controller introduces a bottleneck when dealing with traffic from both buses. Besides, since there are no significative differences between allocating a thread in the same socket or in other socket sharing the bus we can also conclude that, regarding bus sharing, each core can be considered as an independent processor in this context. On Cell 1, the latency is approximately constant regardless of the core and bus involved ($\sim$338 cycles for 64MB and $\sim$349 for 1GB) and lower than the case in which two cores in the same cell are used. This upholds our conclusion about the Cell Controller introducing a bottleneck when dealing with two buses.

The last case studied (10GB) shows the same three latency regions. However, the latency increases noticeably whereas the 64MB and 1GB scenarios showed little difference between them. The randomly linked pointer in such a large block size was increasing the page eviction. We confirmed it by observing that, whereas the number of TLB misses is similar for the 64MB and 1GB cases, the 10GB case yields noticeable higher values.

In general, it seems advisable that the data, regardless of the size, be distributed in different buses.

C. Influence of Thread Allocation in Cache Coherency

The third issue studied was the influence of the thread allocation upon the memory coherency protocols. The rx7640 memory coherency is implemented in two levels. A standard snoopy bus coherence (MESI) protocol is used for the two sockets sharing a bus, with an in-memory directory (MSI Coherence) on top. Therefore, higher latencies are expected when the coherence has to be kept up between cores in different buses than for cores in the same bus.

To quantify the effect of sharing a variable between two cores, a *producer-consumer* benchmark was used. The producer allocates and accesses a whole data block filled with a randomly linked pointer trail, subsequently modifying the data after fetching it into cache. Once the producer has finished, the consumer just reads the whole data. We defined a configuration where Core 14 is always the producer and different cores play the role of the consumer. Table III shows the ticks per access to transfer the data from the producer to different consumers. 10KB, 128KB, 6MB and 1GB data sizes were used to make them fit in the L1, L2, L3 or in memory, respectively.

The results show that if the consumer is in the same bus as the producer the time to fetch a cache line is shorter than if both are in different buses, which is the case of Cores 12 and 15 for 10KB, 128KB and 6MB. It is also noticeable that the time is the same regardless of whether the consumer shares the socket with the producer or not. Remembering also that each core in an Itanium 2 Montvale processor does not share any cache level we can conclude that, regarding cache coherency, each core behaves as an independent processor in this context.

When the consumer is in a different bus than the producer, which is the case of Cores 0, 8 and 10, the directory must be read to check in which bus the requested data is, with the subsequent rise in the access time. Cores 8 and 10 are in the same cell, so their latencies are similar and lower than the latency from Core 0, which is in another cell. Since the data must be brought through another cell controller, this latter case is the most costly. Despite all data fitting in any cache in the previous cases (10KB, 128KB and 6MB), the time increases slightly as the data size also does.

An exception to the observed outcomes occurs for 1GB. In that case, the time to fetch a cache line is practically the same regardless of the cores in the same cell involved. This is due to the size of the allocated memory. For 10KB, 128KB and 6MB the data can reside in the L1, L2 or L3. However, for 1GB the producer must flush the data back to memory after modifying it, so the consumer must fetch the data from main memory in most cases instead of doing it from another cache.

We conclude that, in order to minimise the effect of cache coherency, any parallel application in which some of its threads work with a reduced amount of shared data

Table III
TICKS PER ACCESS OF THE *producer-consumer* BENCHMARK TO TRANSFER THE ALLOCATED DATA BETWEEN TWO CORES.

Prod-Cons	Memory allocated			
	10 KB	128 KB	6 MB	1 GB
14 - 0	317,6	353,2	353,2	296,8
14 - 8	220,0	258,4	261,2	194,4
14 - 10	225,2	258,4	261,2	194,4
14 - 12	79,2	79,2	87,2	192,0
14 - 15	79,2	79,2	87,2	192,0

should be mapped, as far as it is possible, to the available cores in the same bus, regardless of the socket in which they are. When this is not possible, the best choice is the adjacent bus in the same cell and, as last option, a core in a different cell. Nevertheless, this rule is not efficient for parallel applications which allocate a large amount of memory that could saturate the bus, as it was shown in Section V-B, with the subsequent decrease in the performance. In this case, mapping the threads to cores in different buses of the same cell is the best option since for a big amount of memory the latencies due to cache coherency become the same for all buses.

VI. Conclusions

This paper presents several tests carried out to study the suitability of applying the strategies *SPRT*, *Array Expansion* and *DWA-LIP*, initially developed to parallelise irregular codes on SMP systems, to the NUMA machine *Finisterrae*. A set of matrices was chosen and applied to a *sparse matrix-by-vector product* and an *Irregular Reduction* benchmarks. The main factors considered were the thread-to-core distribution regarding bus contention and cache sharing, as well as the memory allocation.

It was stated the importance of spreading the threads among cores in different buses when dealing with big data sizes. In particular, the SpMV benchmark, which generates an important traffic in the bus, showed noticeable improvements when combined with the *SPRT* and *Array Expansion* techniques. The *SPRT* technique proved to be more stable than the rest of techniques regardless the thread distribution. In general, the three techniques obtain better results with big matrices, where the coherency effects are masked by the bus sharing when increasing the data size. In addition, it was observed that each core behaved as an independent processor regardless of the socket they were placed in.

To confirm these issues, some dense benchmarks were executed. Firstly, a benchmark designed to test the performance of several cores when sharing a bus and allocating the data in local or remote memory. Secondly, another one to evaluate the influence of the cache coherency between two cores when sharing data. Finally, another test evaluated the memory access latency depending on the memory module allocated.

At the sight of the results we can claim that, especially for applications which use the bus intensively, the effect of sharing a bus between two or more cores degrades the performance regarding the memory access latency and should be avoided, spreading the threads among cores in different buses when possible. Regarding cache coherency, the effects in the performance are noticeable when dealing with small sizes of data which fit into cache. The more the memory used increases, the less significant the effect is. Therefore, for small data sizes would be advisable to map the threads in cores in the same bus. For bigger data sizes the effect of sharing a bus will be more important and it will mask the effect of cache coherency. Indeed, *SPRT* and *Array Expansion* proved to behave better using several busses for SpMV, which generates a large amount of traffic.

As a future work, we intend to develop strategies to guide applications in run-time in the frame of our project, so the conclusions presented here will help to define a thread-to-core mapping and memory allocation policies.

Acknowledgments

This work was funded by Hewlett-Packard and supported by the MCyT of Spain through the TIN2007-67537-C03-01 project. We also wish to thank the Distributed Systems Research Group of Charles University in Prague.

References

[1] L. Rauchwerger, N. M. Amato, and D. A. Padua, "Run-time methods for parallelizing partially parallel loops," in *Proc. of the Int. Conf. on Supercomputing*, ser. LNCS. ACM press, 1995, pp. 137–146.

[2] R. Eigenmann, J. Hoeflinger, and D. Padua, "On the automatic parallelization of the perfect benchmarks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 9, no. 1, pp. 5–23, 1998.

[3] E. Gutiérrez, O. Plata, and E. L. Zapata, "A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors," in *Proc. of the Int. Conf. on Supercomputing*, ser. LNCS, ACM SIGARCH. Springer-Verlag, May 2000, pp. 78–87.

[4] T. Klug, M. Ott, J. Weidendorfer, and C. Trinitis, "Autopin-automated optimization of thread-to-core pinning on multicore systems," in *Trans. on HiPEAC*, vol. 3, no. 4, 2008.

[5] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *SC '07: Proc. of the 2007 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–12.

[6] M. Nordén, H. Löf, J. Rantakokko, and S. Holmgren, "Dynamic data migration for structured amr solvers." *International Journal of Parallel Programming*, vol. 35, no. 5, pp. 477–491, 2007.

[7] D. E. Singh, M. J. Martin, and F. F. Rivera, "A run-time framework for parallelizing loops with irregular accesses," *Int. Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers (LCR)*.

[8] P. Feautrier, "Array expansion," in *ACM Int. Conf. on Supercomputing*, 1988, pp. 429–441.

[9] *http://math.nist.gov/MatrixMarket/collections/hb.html*, The Harwell-Boeing Sparse Matrix Collection.

[10] *http://www.cesga.es*, *Galicia Supercomputing Centre*.

[11] *http://icl.cs.utk.edu/papi/*, Performance Application Programming Interface (PAPI).

[12] *http://h18000.www1.hp.com/products/quickspecs/12470_div/12470_div.pdf*, HP Integrity rx7640 Server Quick Specs.

[13] L. Marek, "Parallel processing and software performance," Master's thesis, Dept. of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, 2008.