On the Influence of Thread Allocation for Irregular Codes in NUMA Systems

Juan A. Lorenzo, Francisco F. Rivera
Computer Architecture Group
Electronics and Computer Science Dept.
University of Santiago de Compostela
Spain
{*juanangel.lorenzo,ff.rivera*}*@usc.es*

Petr Tuma
Distributed Systems Research Group
Dept. of Software Engineering
Charles University, Prague
Czech Republic
petr.tuma@dsrg.mff.cuni.cz

Juan C. Pichel
Galicia Supercomputing Centre
Santiago de Compostela
Spain
jcpichel@cesga.es

***Abstract*—This work presents a study undertaken to characterise the FinisTerrae supercomputer, one of the biggest NUMA systems in Europe. The main objective was to determine the performance effect of bus contention and cache coherency as well as the suitability of porting strategies regarding irregular codes 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 final performance but masks the cache coherency effects; (3) the NUMA factor (remote to local memory latency ratio) is an important factor on irregular codes and (4) the default kernel allocation policy is not optimal in this system. These results allow us to understand the behaviour of thread-to-core mappings and memory allocation policies.**

***Keywords*-Itanium2, Hardware Counters, Irregular Codes, FinisTerrae.**

I. Introduction

Irregular codes [1] are the core of many important scientific applications, therefore several widespread techniques to parallelise them exist [1], [2], [3]. Many of these techniques were designed to work in SMP systems, where the memory access latency is the same for all processors. However, state-of-the-art architectures involve many cache levels in complex several-node NUMA configurations containing different number of multi-core processors. A good example is the new supercomputer FinisTerrae installed at the *Galicia Supercomputing Centre* (CESGA) in Spain [4]. FinisTerrae is an SMP-NUMA system with more than 2500 processors, 19 TB of RAM memory, 390 TB of disk storage and a 20-Gbps Infiniband network, orchestrated by a SuSE Linux distribution. Designed to undertake great technological and scientific computational challenges, it is one of the biggest shared-memory supercomputers in Europe.

In such a complex infrastructure, the observations we can make are not straightforward, because the interplay of cache contention, bus contention, cache coherency and other mechanisms is far from transparent and therefore requires experimental assessment. 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 which reorder the data to optimise the use of the cache hierarchy, as in iterative kernels such as the *sparse matrix-vector multiplication* (SpMV) and *Irregular Reduction*. Hence, different latencies, depending on the processor the data are assigned to, could significantly affect the performance.

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 [5]. In addition, Williams et al. [6] 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. [7] study the co–location of threads and data motivated by the non-uniformity of memory in NUMA multi-processors, although they do not analyse the behaviour of interleaved memory regions. To our knowledge, nobody has studied the behaviour of several cores sharing a socket and/or a bus on an Itanium2 platform and its influence on irregular codes before.

The main objective of this work is to characterise the behaviour of the FinisTerrae system and to study the suitability and impact of applying the mentioned strategies considering the influence of thread and memory allocations.

The article is organised as follows: Section II introduces the NUMA system under study. Section III explains the methodology followed to characterise its architecture, discussing the influence of the thread allocation. In Section IV the results of running some benchmarks in our architecture are presented and analysed. Afterwards, an actual parallelisation technique for irregular codes was ported to test its performance, which is explained in Section V. Finally, some concluding remarks and future work are given in Section VI.

II. The FinisTerrae Supercomputer

FinisTerrae is an SMP-NUMA machine which comprises 142 HP Integrity rx7640 computation nodes. Each node consists of eight 1.6-Ghz-DualCore Intel Itanium2 Montvale (9140N) processors arranged in a two-SMP-cell

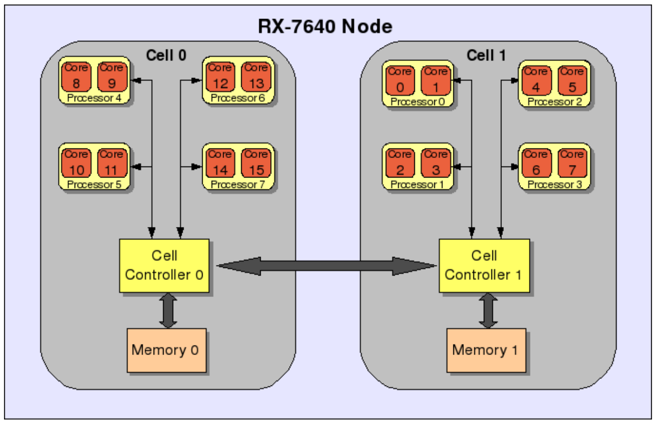


Figure 1. Block diagram of an HP Integrity rx7640 node. 4 dual-core Itanium2 Montvale are connected to their local memory in every cell through a Cell Controller, which uses a crossbar to communicate with an identical second cell.

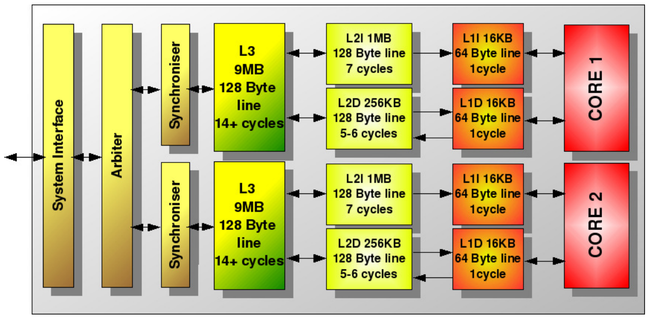

System Interface
Arbiter
Synchroniser
Synchroniser
L3 9MB 128 Byte line 14+ cycles
L3 9MB 128 Byte line 14+ cycles
L2I 1MB 128 Byte line 7 cycles
L2D 256KB 128 Byte line 5-6 cycles
L2I 1MB 128 Byte line 7 cycles
L2D 256KB 128 Byte line 5-6 cycles
L1I 16KB 64 Byte line 1cycle
L1D 16KB 64 Byte line 1cycle
L1I 16KB 64 Byte line 1cycle
L1D 16KB 64 Byte line 1cycle
CORE 1
CORE 2

Figure 2. 1.6Ghz Dual-Core Intel Itanium 2 Montvale (9140N) architecture. Note the separated L3 cache per core.

NUMA configuration[1]. Figure 1 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 addresses in the local memory, the remaining one fourth maps in an interleaved manner to addresses in both local and remote memory.

Focusing on the processors, Figure 2 shows a block diagram of an Itanium2 Montvale processor. Each processor comprises two 64-bit cores and three cache levels per core. This architecture has 128 General Purpose Registers and 128 FP registers. L1I and L1D (write-through) are both 4-way set-associative, 64-byte line-sized caches. L2I and L2D (write-back) are 8-way set-associative, 128-byte line-sized caches. Note that FP operations bypass L1.

III. Methodology

The previous section presented a several-memory-layer platform where each NUMA node's cell behaves as an SMP. Since a good performance of many shared-memory scientific applications depends on the correctness of this assumption, our study focuses on quantifying the behaviour of a FinisTerrae node depending on how the data allocation, the memory latency and the thread-to-core mapping can influence the code's final performance.

[1]There exists a 143th node composed of 128 Montvale cores and 1 TB memory, which is not considered in this work.

In order to characterise our test platform, an analysis in two stages was carried out:

1) *Benchmarking:* A set of benchmarks to measure the bus contention as well as the effect of the cache coherency overhead were executed.
2) *Actual scenario:* An actual strategy to parallelise irregular codes, designed to be executed on SMP systems, was ported to FinisTerrae to evaluate its behaviour and, if possible, to apply the knowledge acquired in the benchmarking stage to improve its performance.

Orthogonally to this, two policies were adopted:

1) *Intra-cell scenario:* In this scenario all data are allocated in local cell memory and only cores in the same cell, following several thread-to-core assignments, are involved.
2) *Inter-cell scenario:* Both data and threads can be allocated in any cell.

Outcomes were collected with *PAPI* [8] (which accesses the *Perfmon2* interface [9] underneath) and *pfmon* [10]. The benchmarks used in the benchmarking stage perform some warm-up iterations before starting to measure and, subsequently, they process the outcomes statistically to show the results in median values, which guarantees that the outliers are not taken into account. On the other hand, in the *actual scenario* stage, only values inside the outcomes' standard deviation were considered.

The next sections will develop the characterisation described above. Section IV will undertake the benchmarking stage to study issues such as the bus contention (Section IV-A), coherency overhead (Section IV-B) and the memory allocation (Section IV-C) in our architecture. Section V will study similar aspects (sections V-B and V-C) using an actual parallelisation technique for irregular codes.

IV. Benchmarking FinisTerrae

A. Influence of Thread Allocation in Bus Contention

The first issue under study was the influence of the thread allocation upon the node buses. Considering that every four cores share a bus, it was reasonably foreseeable that any

Table I

MEDIAN MEMORY ACCESS LATENCY (IN TICKS) OF *memtest* BENCHMARK FOR DIFFERENT CONFIGURATIONS. DEPENDING ON THE SECOND CORE INVOLVED, CORE 8 WILL SHARE WITH IT THE BUS, THE CELL OR NONE OF THEM. A MEASUREMENT WITH CORE 8 ALONE FOR EACH MEMORY SIZE IS ALSO SHOWN AS A REFERENCE.

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

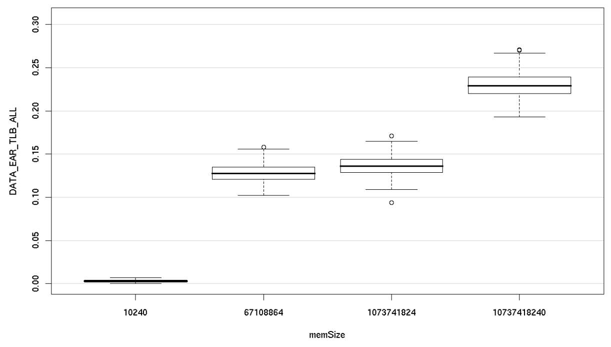


Figure 3. Average L1 DTLB misses per access for the *memspeed* benchmark. The y-axis represents the number of events. The x-axis, the memory size in bytes for the considered cases.

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 within the same bus, due to a possible bandwidth competition.

To quantify this effect, a benchmark called *memtest* was used. *Memtest* focuses on how multiple cores share the bandwidth to memory. 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 data read and, subsequently, written-back to memory. To quantify the effect of sharing a bus, several configurations comprising different thread allocations were used. To avoid the system allocating the data in different memory regions (local, remote or interleaved) during the tests, all data were allocated in Cell 0. One thread was always mapped to Core 8 (see Figure 1). The other one was mapped to a core in the same cell, 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 which do not share any resources inside a cell. To have a comparison value, a test mapping just one thread to Core 8 was also carried out. Table I quantifies the outcomes of those configurations for memory blocks of 10KB, 64MB, 1GB and 10GB in clock ticks per memory access (note that, since the RDTSC instruction was used to measure the memory access time, the final average calculation will include the access time but also some overhead). Depending on the two cores involved in each test and the size of the memory allocated, a decrease in the performance was expected as long as the traffic in the bus increases (because of bigger memory sizes) when both cores share the same bus. That is, a memory size of 10KB is not expected to consume too much bandwidth since the data fits in the L1 and, once loaded in cache, the bus will not be used until write-back. For block sizes bigger than 9MB (the L3 size) the traffic in the bus is expected to increase, not only because of write-back, but also because of the cache replacing. Therefore, a poorer performance is expected when two threads mapped to two cores in the same bus and a big amount of memory allocated are considered.

The outcomes in Table I for 10KB show that, regardless of the pair of cores involved in Cell 0, the number of required clocks to access the data is the same (3,5 ticks). As expected, for a data block small enough to fit into a 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 than the previous case, as can be expected because all data are allocated in Cell 0.

When the size of the block is increased over the L3 size (64MB and 1GB) three different cases can be identified. Focusing 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 as in the case when the same bus is shared. 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 should conclude that the Cell Controller introduces, somehow, a small bottleneck when dealing with traffic from both buses. Besides, since there are no significant 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. Focusing on Cell 1, the latency is approximately constant regardless of the core and bus involved ($\sim$338 cycles for 64MB case and $\sim$349 for 1GB) and lower than using two cores in the same cell. Although this upholds our conclusion about the Cell Controller introducing a bottleneck, it must be taken with caution given that the difference is quite small and some measurement errors are assumed to be introduced.

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. It seemed reasonable to think that the randomly linked pointer in such a large block size was increasing the page eviction. To confirm it, we studied the behaviour of the TLB. Figure 3 shows the L1 DTLB misses. Indeed, we can see that the number of cache misses is similar for both the 64MB and 1GB cases. However, the 10GB case yields a higher number of TLB misses. Even if some page requests can be satisfied by the L2 DTLB, the rest will produce page faults which will increase the access latency noticeably.

We can therefore conclude that it seems advisable that the data, regardless of the size (as long as it is larger than the cache size), be distributed in different buses.

B. *Influence of Thread Allocation in Cache Coherency*

The second 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 snooping bus coherence (MESI) protocol is used for the two sockets sharing a bus, having on top of it an in-memory directory (MSI) to keep inter-bus coherence. Therefore, higher latencies are expected when the coherence has to be kept up between two cores in different buses than for two cores in the same bus, since in the former case, the directory must be read.

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 II 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.

At the sight of the results we can observe 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. This is due to the behaviour of the MESI protocol implemented at bus level, which is faster than reading the directory. 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 cores in an Itanium 2 Montvale processor do not share any cache level, we can conclude that, regarding cache coherency, each core

Table II
DURATION (TICKS/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

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. In this latter case, since the data must be brought through another Cell Controller, it exhibits the highest latency.

Despite all data fitting in any cache in the previous cases (10KB, 128KB and 6MB), it can be noticed that the time increases slightly with the data size. This fact can be explained arguing that we are observing the effect of cache collisions due to the limited associativity of the caches.

An exception to the observed outcomes occurs for 1GB. In that case, the time to fetch a cache line is practically identical regardless of the cores involved, as long as they belong to the same cell. The cause for this behaviour lies in 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. Note also that the time to retrieve a data from a core in a remote cell is higher than from the local memory, as it is seen in the 14-0 case for 1GB, compared to the 6MB case.

We can conclude that, to minimise the effect of cache coherency, any parallel application working with a reduced amount of shared data –not much bigger than the cache size– should map its threads 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 a last option, a core in a different cell. On the contrary, a parallel application which allocates a large amount of memory might saturate the bus (as was shown in Section IV-A) with the subsequent decrease in performance. In this case, mapping the threads to cores in different buses of the same cell might be the best option since, as shown in Table II, for a big amount of memory the latencies due to cache coherency become the same for all buses. Therefore, the application should be first characterised to find out whether the restricting factor is the traffic in the bus due to the amount of allocated memory or due to the cache coherency, in order to take a proper decision.

C. *Influence of Memory Allocation*

As explained in Section II, about one fourth of a cell's memory in FINISTERRAE is configured in interleaving mode, which means that all data allocated in that part are distributed between local memory addresses and remote ones. This behaviour allows to decrease the average access time when accessing data simultaneously from cores belonging to different cells.

In this section, an experiment was carried out to compare the theoretical memory latency given by the manufacturer [11] with our observations. We measured the memory access latency of a small Fortran program, which creates an array and allocates data in it. The measurements were carried out using specific Itanium2's hardware counters, the *Event Address Registers* (EAR), using the *pfmon* tool. *Pfmon*'s underlying interface, *Perfmon2*, samples the application at run-time using EARs, getting the memory position and access latency of a given sample accurately.

Figure 4 depicts the results when allocating the data in the same cell as the used core (a), in the remote cell (b) and in the interleaving zone (c). In all of the cases, many accesses happen within 50 cycles, corresponding with the accessed data which fit in cache memory. There is a gap and, then, different values can be observed depending on the figure. Figure 4(a) shows occurrences between 289 and 383 cycles when accessing the cell local memory. The frequency of our processors is 1.6 Ghz, which yields a latency from 180,9 to 239,7 ns. Its average value is 210,3 ns, slightly higher than the 185 ns given by the manufacturer.

When accessing data in a cell remote memory we measured occurrences between 487 and 575 cycles, that is, from 304,8 to 359,8 ns, with an average value of 332,3 ns. The manufacturer does not provide any values in this case.

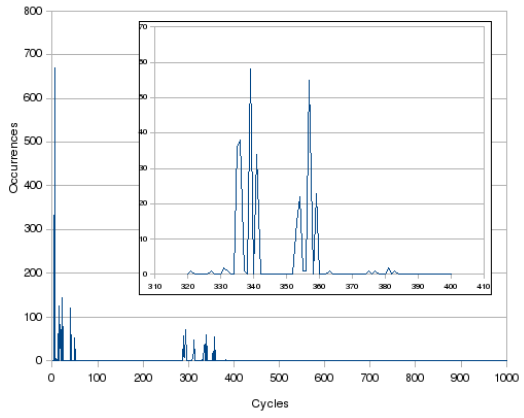
In the case of accessing data in the interleaving zone, the manufacturer value is 249 ns. Our measurements give two zones, depending on whether the local or remote memory are accessed. Indeed, the average access time in the interleaving zone is the average of combining accesses to the local or remote memory. Our outcomes gave an average value of 278,3 ns.

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

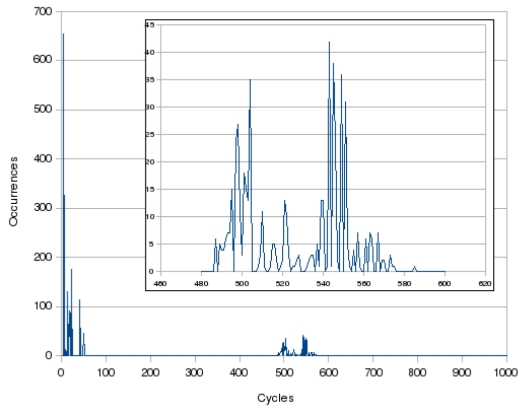
V. INFLUENCE OF THREAD ALLOCATION ON IRREGULAR CODES

A. *Ported Technique*

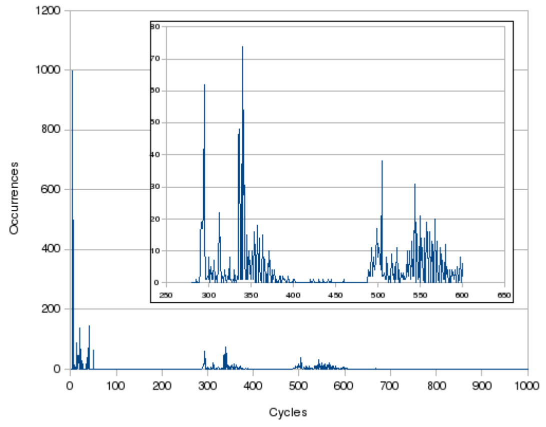
The next step consisted of porting a parallelisation technique for irregular codes to our target architecture, in order



(a)



(b)



(c)

Figure 4. Latency of memory accesses when the data is allocated in memory local to the core (a), in memory on the other cell (b) or in the interleaving zone (c). The y-axis shows the number of occurrences of every access. The x-axis shows the latency in cycles per memory access. Regions of interest have been zoomed in.

Table III
SPARSE MATRIX-VECTOR PRODUCT (LEFT) AND IRREGULAR REDUCTION (RIGHT) PSEUDOCODES

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

```
for $j = 1$ to $NNZ$ do
    $i = row(k)$
    $Z(i) = Z(i) + alpha$
end for
```

to check out the effect of thread allocation in an actual application. The chosen technique was *IARD* [12] (*Irregular Access Region Descriptor*). It is a method to efficiently characterise irregular codes at run-time which exploits the properties found in the access patterns, expressing them with a structure that allows a strong reduction in storage requirements without loss of relevant information. The low-cost, compact characterisation of the subscript arrays can be used to perform an efficient parallelisation of a wide set of irregular codes.

In order to test the efficiency of this technique, two well known benchmarks (*Sparse matrix-Dense vector Product* and *Irregular Reduction*) were implemented in Fortran and parallelised using OpenMP. The pseudocodes are shown in Table III. In both cases, some selected sparse matrices stored in CSC format from the *Harwell-Boeing Sparse Matrix Collection* [13], which had previously been reordered using this technique, were used as an input. The key features of these matrices are shown in Table IV.

The compiler used for the experiments was *Intel ifort 9.1.052*. The baseline compile configuration used in all our tests involved the `-O3` option and the interprocedural optimisation (`-ipo`).

B. Influence of Bus Contention

To study the influence of the bus contention several scenarios were tested. On the one hand, an intra-cell scenario where the data are allocated in a single cell's memory and the cores involved belong to the same cell. On the other hand, an inter-cell scenario where the data are fetched from the other cell's memory and there are cores involved from both cells. Before presenting the results, it is necessary to clarify the meaning of a so-called *thread distribution*. For example, a distribution 15-11-13-9 means that for a one-thread execution, the thread will be allocated in Core 15. For two threads, they will be allocated in Cores 15 and 11 and, for four threads, they will be in Cores 15, 11, 13 and 9. In the cases where more cores are used and the distribution is not pointed out, it means that the order in which the remaining available cores are assigned is not relevant.

Table IV
SQUARED-MATRIX INPUT SET USED IN OUR TESTS. N_a=MATRIX SIZE. N_z=NUMBER OF NON-ZEROS.

Name	N_a	N_z	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

Table V
PERFORMANCE IMPROVEMENT (% MFLOPS) OF THREAD DISTRIBUTION 15-11-13-9 OVER 15-14-13-12 FOR $N_c = 2$ AND $N_c = 4$ CORES. DATA ARE ALLOCATED IN CELL 0 MEMORY.

Matrix	N_c	Benchmark	
		IrregRed	**SpMV**
s3dkq4m2	2	0,0	**6,3**
	4	1,6	**23,9**
3dtube	2	0,8	**6,6**
	4	0,7	**13,5**
nasasrb	2	2,7	1,4
	4	-1,1	**7,7**
struct3	2	-1,1	0,7
	4	0,5	1,6
bcsstk29	2	0,9	1,7
	4	1,3	1,8
bcsstk17	2	1,0	0,2
	4	1,5	0,7

1) Intra-cell Scenarios: In this scenario, the *libnuma* library was used to allocate all data in the Cell 0 memory and only cores in this cell were used. Several scenarios were set up to study the effect of running the *Irregular Reduction* and *SpMV* benchmarks with different distributions, in order to quantify the effect of the bus sharing:

15-11-13-9 vs 15-14-13-12: This scenario is focused on 2 and 4 cores. In both benchmarks the sparse matrix must be completely read, which is expected to cause some number of capacity and compulsory cache misses. Therefore, the bigger the matrix, the higher the traffic in the bus, so a performance decrease is expected for the biggest matrices when sharing the bus. At the sight of the results (Table V), a better behaviour of the distribution 15-11-13-9 can be appreciated (differences over 6% are highlighted) for the biggest matrices (*3dtube*, *nasasrb* and *s3dkq4m2*) in the four-core case. Indeed, it was confirmed that the bigger the matrix, the higher the traffic in the bus and the higher the improvement when there are only two threads per bus (15-11 and 13-9) instead of four threads in a single bus (15-14-13-12). For the two-core case, although a slight improvement can be observed, the traffic is not high enough to be relevant. It is noticeable that the most important improvements occur only in the *SpMV* benchmark, especially for four processors. This behaviour happens because *SpMV* executes a loop which goes through five different arrays, whereas *Irregular Reduction* goes only through two. So in the former case the cache reuse is smaller than in the latter one. At any given moment, therefore, the *SpMV* benchmark generates more traffic in the bus than the *Irregular Reduction* one.

15-11-13-9 vs 15-14-11-10: This scenario is focused on 4 cores. The outcomes in Table VI show that there are no noticeable differences between the case when the threads are

Table VI
PERFORMANCE IMPROVEMENT (% MFLOPS) OF THREAD DISTRIBUTION 15-14-11-10 OVER 15-11-13-9 FOR $N_c = 4$ CORES. DATA ARE ALLOCATED IN CELL 0 MEMORY.

Matrix	N_c	Benchmark	
		IrregRed	SpMV
s3dkq4m2	4	4,1	2,4
3dtube	4	-0,5	1,6
nasasrb	4	0	0,8
struct3	4	0,6	0,3
bcsstk29	4	0,6	-0,2
bcsstk17	4	1,4	0,1

allocated in different sockets and the case when two threads in the same bus share the socket. We know that IARD has no dependencies among the cores involved. Since the outcomes show almost no differences between threads mapped to different cores in different sockets and threads sharing the same socket, we can confirm, then, the conclusion of Section IV-A, which stated that each core behaves as an independent processor regardless of the sockets they are placed in.

2) Inter-cell Scenarios: In this case, the influence of fetching data from a memory module in other cell was studied. Moreover, a comparison was carried out between the automatic thread assignment the system does and our manual assignments. To do that, *libnuma* was used through its command line tool `numactl`, in order to allocate all the data in the cell we were working in (memory 0), the remote memory (memory 1) or the interleaving zone (memory 2). We intended to infer from these experiments whether the default system behaviour is reliable or, on the contrary, another memory allocation policy should be used.

15-11-13-9 vs system-assigned: We used the *Perfmon2* library to check the system's default policy to assign threads to cores. This scenario compares a manual allocation with the default one done by the system. We found out that the system assigns the threads without a defined policy, changing it in every execution, although it strives to place the threads in separate cells as far as it is possible. Therefore, after discarding outliers, we present a range of results inside the standard deviation instead of a single average value. The outcomes in Table VII show a dramatic improvement with our distribution, especially for the biggest matrices (*3dtube*, *nasasrb* and *s3dkq4m2*). In our previous tests in the intra-cell scenario, there was no distribution that showed such a big difference in the performance, which indeed suggests that the system does not maximise locality.

Threads spread over both cells: Our second test intended to find out whether the system assignment spreads the threads over two cells, even if there are available cores in the same cell. So the automatic assignment was compared to a manual distribution where the number of involved threads is kept balanced between both cells. The threads were distributed to cores 15-7 for two threads, 15-7-11-3 for four and 15-7-11-3-13-5-9-1 for eight. Focusing on the

Table VII
PERFORMANCE IMPROVEMENT (% MFLOPS) OF THREAD DISTRIBUTION 15-11-13-9 (DATA ALLOCATED IN CELL 0 MEMORY) OVER AUTOMATIC ASSIGNMENT FOR $N_c = 2$ AND $N_c = 4$ CORES. AVERAGE, MAXIMUM AND MINIMUM IMPROVEMENTS ARE SHOWN.

Matrix	N_c	Benchmark					
		IrregRed			SpMV		
		Avg.	Max	Min	Avg.	Max	Min
s3dkq4m2	2	**23,8**	25,5	23,1	**32,5**	32,9	32,3
	4	**9,8**	14,3	7,5	**15,0**	17,5	13,9
3dtube	2	**17,8**	20,0	16,3	**34,0**	34,4	33,8
	4	**4,0**	7,7	1,5	**10,9**	13,5	9,1
nasasrb	2	**9,5**	13,5	7,9	**31,6**	33,0	31,2
	4	-0,8	4,3	-1,9	**14,6**	18,0	11,7
struct3	2	-0,3	2,5	-1,1	**13,9**	17,0	11,5
	4	-1,1	3,0	-1,7	**3,6**	8,4	1,6
bcsstk29	2	0,1	2,0	-0,3	1,3	5,2	0,6
	4	-1,5	3,8	-2,3	0	2,4	-0,7
bcsstk17	2	0,3	3,3	0	1,6	4,8	1,1
	4	-1,5	10,1	-3,3	0,2	3,2	-0,4

Table VIII
PERFORMANCE IMPROVEMENT (% MFLOPS) OF THREAD DISTRIBUTION 15-7-11-3 WITH DATA ALLOCATED IN THE INTERLEAVING ZONE VS. AUTOMATIC DISTRIBUTION FOR $N_c = 2$, $N_c = 4$ AND $N_c = 8$ CORES. AVERAGE, MAXIMUM AND MINIMUM IMPROVEMENTS ARE SHOWN.

Matrix	N_c	Benchmark					
		IrregRed			SpMV		
		Avg.	Max	Min	Avg.	Max	Min
s3dkq4m2	2	-6,9	-5,6	-7,4	4,2	4,4	4,0
	4	-4,5	-0,6	-6,5	0,8	3,0	-0,2
	8	-2,0	3,0	-3,0	2,0	6,3	-1,9
3dtube	2	-2,5	-0,8	-3,9	5,3	5,7	5,1
	4	4,4	8,1	1,9	-3,2	-0,9	-4,7
	8	2,2	18,3	-2,7	2,4	8,0	-1,7
nasasrb	2	-3,7	-0,2	-5,1	3,0	4,1	2,6
	4	-2,3	2,7	-3,3	-2,5	0,4	-5,0
	8	-0,1	12,1	-1,9	6,8	12,6	2,5
struct3	2	1,1	3,9	0,3	-3,6	-1,0	-5,6
	4	0,1	4,3	-0,4	-1,2	3,4	-3,1
	8	0,2	6,5	-1,5	3,0	7,3	0,6
bcsstk29	2	0,1	2,1	-0,3	-2,2	1,0	-2,9
	4	-0,1	5,2	-1,0	0,6	3,0	-0,1
	8	0,7	11,1	-2,7	-1,4	3,5	-2,6
bcsstk17	2	-0,1	2,9	-0,4	0,9	4,1	0,5
	4	0,5	12,6	-1,4	0,6	3,6	0
	8	1,4	20,4	-3,6	-0,6	5,0	-1,8

IARD, where there are no dependencies between threads, the outcomes in Table VIII show that all differences are below 6%, except for some cases in the biggest matrices: nasasrb (SpMV, 8 threads) and s3dkq4m2 (IrregRed, 2 threads). In general, especially for small matrices, our distribution performs better than the automatic one.

C. Influence of Cache Coherency

Taking into account that this kind of irregular codes strives to maximise the thread locality, we do not expect to notice important performance decreases due to the cache coherency protocols when the input matrices are big enough, since they will be masked by the traffic in the bus. 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. However, this will not usually be the case in real applications, because this type of codes usually operates with matrices far bigger than the cache size.

VI. Conclusions

This paper presented the results of several tests carried out to characterise FinisTerrae, an SMP-NUMA machine, in order to study the suitability of applying strategies to parallelise irregular codes initially developed for SMP systems.

The main factors considered were the thread-to-core distribution and the memory allocation. Firstly, a benchmark was executed to test the performance influence of several cores when sharing a bus and allocating the data in local or remote memory. Secondly, another benchmark was used to evaluate the influence of the cache coherency between two cores when sharing data. Furthermore, another test evaluated the memory access latency depending on the memory module allocated (local, remote or interleaving).

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 and should be avoided by spreading the threads among cores in different buses when possible. We must take into account, though, that the test to evaluate the local and remote memory access latencies yielded important differences between them. Therefore, in cases where the threads must be spread in two cells the best policy will be to analyse and split the data in both memory zones, maximising the locality or, when not possible, allocating the data in the interleaving zone.

Regarding the cache coherency, the effects in the performance are noticeable when dealing with small sizes of data which fit into caches. The more the memory size increases, the less significant the effect is. Therefore, for small data sizes, it would be advisable to map the threads on 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. A noteworthy fact is that every core in the same processor behaves as an independent processor, so we can consider two processors in a bus as four independent cores.

After the benchmarking stage, an actual strategy to parallelise irregular codes, successfully tested on SMP architectures, was ported to FinisTerrae. A set of matrices was chosen and reordered with this strategy, being applied subsequently to the *sparse matrix-vector product* and the *Irregular Reduction* benchmarks. Since this code works with sparse data, the effects regarding bus sharing were not as noticeable as in the previous stage. However, it was possible to confirm the importance of spreading the threads among cores in different buses when dealing with big data sizes, the behaviour of each core as an independent processor, and the fact that the coherency effects are masked by the bus sharing when increasing the data size.

As a future work, we intend to develop strategies to guide applications at 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.

Acknowledgements

This work was supported by the MCyT of Spain through the TIN2007-67537-C03-01 project and by the 2008/CE377 contract among HP, CESGA, UDC and USC.

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*. 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*, ACM SIGARCH. Springer-Verlag, May 2000, pp. 78–87.

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

[5] J. W. Tobias Klug, Michael Ott and C. Trinitis, “Autopin - automated optimization of thread-to-core pinning on multicore systems,” in *Transactions on HiPEAC*, vol. 3, no. 4, 2008.

[6] 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 Conf. on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–12.

[7] 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.

[8] *Performance Application Programming Interface*, http://icl.cs.utk.edu/papi/.

[9] S. Eranian, *The perfmon2 Interface Specification. Technical Report HPL-2004-200R1*. HP Labs, February 2005.

[10] *Perfmon2 monitoring interface and Pfmon monitoring tool*, http://perfmon2.sourceforge.net.

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

[12] D.E.Singh, M.J.Martin, and F.F.Rivera, “A run-time framework for parallelizing loops with irregular accesses,” in *Proc. Seventh Workshop Languages, Compilers, and Run-Time Systems for Scalable Computers*, Washington DC, USA, 2002.

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