GALICIA SUPERCOMPUTING CENTER

Applications & Projects Department

FinisTerrae: Memory Hierarchy and Mapping

Technical Report CESGA-2010-001

Juan Carlos Pichel

Tuesday 12th January, 2010

Contents

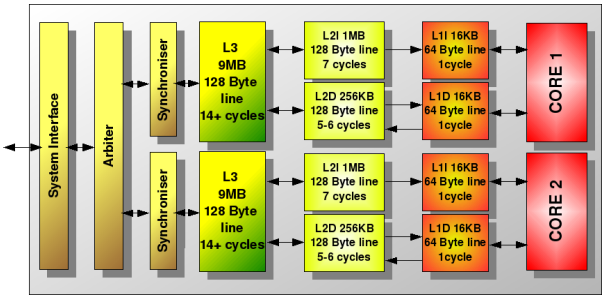


Figure 1: Block diagram of a Dual-Core Intel Itanium2 (Montvale) processor.

1 Introduction

In this technical report some topics about the FINISTERRAE architecture and its influence in the performance are covered. In Section 2 a brief description of the Itanium2 Montvale processors is provided. Additionally to this, a method for identifying the CPUs is explained. At the end of this section the memory coherency mechanism of a FINISTERRAE node is detailed. Section 3 deals with the tools provided by the Linux kernel to support NUMA architectures. An example of the benefits of using these tools in the FINISTERRAE for the sparse matrix-vector product (SpMV) kernel is shown.

2 Useful Topics about the FinisTerrae Architecture

FINISTERRAE supercomputer consists of 142 HP Integrity rx7640 nodes [1], with 8 1.6GHz Dual-Core Intel Itanium2 (Montvale) processors and 128 GB of memory per node. Additionally, there is a HP Integrity Superdome with 64 1.6GHz Dual-Core Intel Itanium2 (Montvale) processors and 1 TB of main memory (not considered in this report). The interconnect network is an Infiniband 4x DDR network, capable of up to 20 Gbps (16 Gbps effective bandwidth). It has, also, a high performance storage system, the HP SFS, made up of 20 HP Proliant DL380 G5 servers and 72 SFS 20 disk arrays. The SFS storage system is accessed through the Infiniband network. The operating system used in all the computing nodes is SuSE Linux Enterprise Server 10.

2.1 Itanium2 Montvale Processors

Each Itanium2 processor comprises two 64-bit cores and three cache levels per core (see Figure 1). This architecture has 128 General Purpose Registers and 128 FP registers. L1I and L1D (write-through) are both a 4-way set-associative, 64-byte line-sized cache. L2I and L2D (write-back) are a 8-way set-associative, 128-byte line-sized cache. Finally, there is an unified 12-way L3 cache per core, with line size of 128 bytes and write-back policy. Note that floating-point operations bypass the L1. Each Itanium2

cpu_id	*phys_id*	*hex_phys_id*	*core_id*	*thread_id*
0	65536	0x010000	0	0
1	65536	0x010000	1	0
2	65537	0x010001	0	0
3	65537	0x010001	1	0
4	65792	0x010100	0	0
5	65792	0x010100	1	0
6	65793	0x010101	0	0
7	65793	0x010101	1	0
8	0	0x000000	0	0
9	0	0x000000	1	0
10	1	0x000001	0	0
11	1	0x000001	1	0
12	256	0x000100	0	0
13	256	0x000100	1	0
14	257	0x000101	0	0
15	257	0x000101	1	0

Table 1: Example of core identification for the rx7640 node.

core can perform 4 FP operations per cycle. Therefore, the peak performance per core is 6.4 GFlop/s. HyperThreading is disabled in the FinisTerrae.

2.2 CPUs Identification

Next, we will show how to identify the cores of the two cell rx7640 node. With this purpose we must interpret the output of `/proc/cpuinfo`. An example is shown in Table 1. Note that *hex_phys_id* is not provided by the kernel.

The first two hex digits of the *hex_phys_id* are the cell number. The second two hex digits have the value of either 00 or 01, and each of these values appears twice with a 00 or 01 in the last 2 digits. The second field is the cell internal bus number, and the last field is the socket id on that bus. That is, the format of the hex representation of the *phys_id* field is 0xCCBBSS where C=cell, B=bus, and S=socket. *core_id* identifies the core in the socket. In this example, Hyperthreading is not enabled, so the *thread_id* is always 0.

According to the previous analysis a block scheme of a rx7640 node can now be provided (see Figure 2). Note that cores ranging from 0 to 7 belong to Cell 1, and cores 8 to 15 to Cell 0. In a more detailed way, a cell has two buses at 8.5 GB/s (6.8 GB/s sustained), each connecting two sockets (that is, four cores) to a 64GB memory module through a sx2000 chipset (Cell Controller). The Cell Controller maintains a cache-coherent memory system using a directory-based protocol and connects both cells through a 34.6 GB/s crossbar (27.3 GB/s sustained). It yields a theoretical processor-cell controller peak bandwidth of 17 GB/s and a cell controller-memory peak bandwidth of 17.2 GB/s (four buses at 4.3 GB/s).

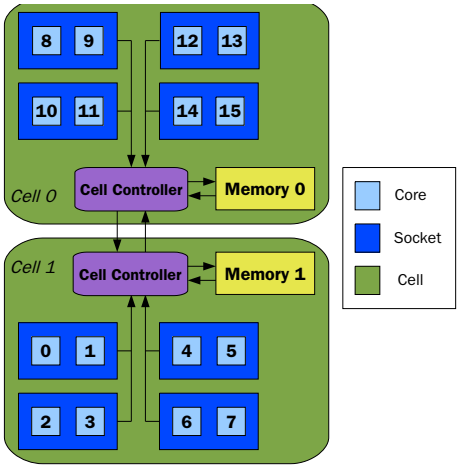


Figure 2: Block diagram of a rx7640 node.

2.3 Coherency Mechanism of a rx7640 node

The memory coherency is implemented on two levels. Standard snoopy bus coherence protocol (MESI) [2], for the two sockets on each bus, layered atop an in-memory directory (MSI Coherence). There are 4 buses on the rx7640. Each line in memory is tagged to indicate whether that line is held in the caches of any CPU on each bus. If a line of cache is shared, a bit for each bus indicates so. If the line is held exclusive or modified by a CPU, the bus the CPU resides upon is noted in the directory.

When a line is read, the directory entry for that line is also read, it is actually encoded in memory along with the line. If the access requires an update to the directory (such as to indicate a new sharer), the line is rewritten in the background.

If the request is for a line that is modified in the cache of a CPU, the following happens:

1. Requester sends a read to the home cell.
2. The home cell reads the line, and the directory.
3. The home cell notes the request is for a line modified in another CPU's cache.
4. The home cell forwards the request to the bus of the owner of the line.
5. A read invalidate is sent on the bus that owns the line.
6. The owning CPU responds with a HIT Modified, sends a copy of the line to requester, and invalidates it's copy of the line.
7. The tags are updated in memory to indicate the new owner of the line (in parallel with 4)

If a line is cast out of the L3 cache of a CPU, it is guaranteed to be not held in any other level of the CPU's cache. If the line is dirty, it is written back to memory. If the line is unmodified a clean castout

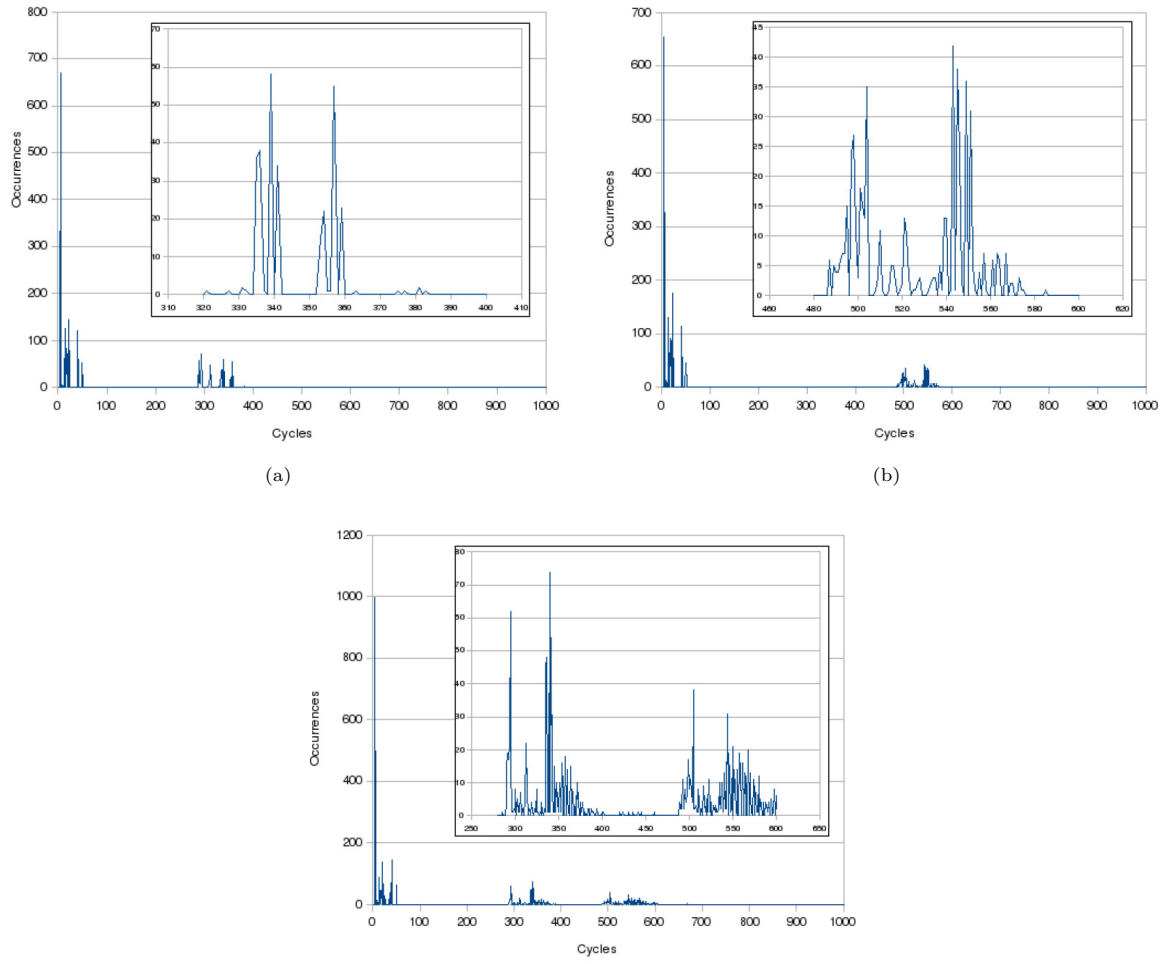


Figure 3: Latency of memory accesses when the data is allocated in memory local to the core (a), in memory on the other cell (b) or in the interleaving zone (c). The y-axis shows the number of occurrences of every access. The x-axis shows the latency in cycles per memory access. Regions of interest have been zoomed in.

transaction is generated to memory by the CPU indicating that the CPU no longer has a copy of the line. In either case the line ownership is set to uncached / unowned.

On Xeon, castout does not guarantee that a line is not still cached somewhere on the chip. Xeon does not issue clean castouts.

2.4 Memory Latency

An experiment was carried out to compare the theoretical memory latency given by the manufacturer with our observations [3]. We have 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` [4], samples the application at run-time using EARs, getting the memory position and access latency of a given sample accurately.

Figure 3 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). When interleaved memory is used, 50% of the addresses are to memory on the same cell as the requesting processor, and the other 50% of the addresses are to memory on the other cell (round-robin). The main goal of using interleave memory is to decrease the average access time when accessing data simultaneously from processors belonging to different cells. 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 3(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.

3 Linux NUMA support

NUMA scheduling and memory management became part of the mainstream Linux kernel as of version 2.6. Linux assigns NUMA policies in its scheduling and memory management subsystems. Memory management policies include *strict* allocation to a node, *round-robin* memory allocations, and *non-strict* preferred binding to a node (meaning that allocation is to be preferred on the specified node, but should fall back to a default policy if this proves to be impossible) [5]. The default NUMA policy is to map pages on to the physical node which faulted them in, which in many cases maximises data locality. A number of system calls are also available to implement different NUMA policies. These system calls modify scheduling and virtual memory related variables structures within the kernel.

Relevant system calls include `mbind()`, which sets the NUMA policy for a specific memory area, `set_mempolicy()`, which sets the NUMA policy for a specific process, and the `sched_setaffinity()`, which sets a process' CPU affinity. Several arguments for these system calls are supplied in the form of bit masks, and macros, which makes them relatively difficult to use.

3.1 The `libnuma` API

For the application programmer a more attractive alternative is provided by the `libnuma` API. Within `libnuma` useful functions include the `numa_run_on_node()` call to bind the calling process to a given node and `numa_alloc_onnode()` to allocate memory on a specific node. Similar calls are also available to allocate interleaved memory, or memory local to the caller's CPU. The `libnuma` API can also be used to obtain the current NUMA policy and CPU affinity. To identify NUMA related characteristics libnuma accesses entries in `/proc` and `/sys/devices`. This makes applications using `libnuma` more portable those that use the lower level system calls directly

Alternatively, `numactl` is a command line utility that allows the user to control the NUMA policy and CPU placement of a entire executable.

3.2 Using `numactl` in the FinisTerrae

As we have indicated before, `numactl` runs processes with a specific NUMA scheduling or memory placement policy. The policy is set for command and inherited by all of its children. In addition, it can set persistent policy for shared memory segments or files (not considered in this report).

Using the `show` flag the NUMA policy of the current process is displayed. This is an example of the output in a rx7640:

```
policy: default
preferred node: current
physcpubind: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
cpubind: 0 1
nodebind: 0 1
membind: 0 1 2
```

Memory placement policy options

Using the `hardware` option an inventory of avaliable nodes on the system is displayed. In the particular case of a rx7640 node, the output shows something like:

```
available: 3 nodes (0-2)
node 0 size: 57280 MB
node 0 free: 51784 MB
node 1 size: 57255 MB
node 1 free: 42300 MB
node 2 size: 16371 MB
```

```
node 2 free: 16137 MB

node distances:
 node   0   1   2
  0:   10  17  14
  1:   17  10  14
  2:   14  14  10
```

According to this a rx7640 has three memory nodes. Two correspond to the cells (a memory module per cell), and the third one refers to the *interleaving* memory (node 2). Information about the used and free memory per node is also provided. And finally, the NUMA factor between nodes is shown. This value shows the difference in latency for accessing data from a local memory location as opposed to a non-local one. Therefore, an access to a remote memory in a rx7640 is 1.7x slower than a local access, and 1.4x slower (on average) when the interleaving policy is considered. Experimentally (see the results in Section 2.4) we have observed that these values are slightly lower, in particular, 1.6x and 1.3x respectively.

`membind` option allows to only allocate memory from a particular set of nodes (0, 1 and 2 as we have shown previously). Allocation will fail when there is not enough memory available on these nodes. `preferred` option preferably allocate memory on node, but if memory cannot be allocated there fall back to other nodes. Finally, using the `interleave` option the interleave policy is set. When memory cannot be allocated on the current interleave target fall back to other nodes.

Processes placement options

The main options regarding the processes placement are `cpunodebind` and `physcpubind`. In the first case the process only executes on the CPUs belonging to the considered nodes. Note that nodes may consist of multiple CPUs. In particular, a rx7640 has two nodes (cells) with 8 cores each one (see above the `show` option output).

When using the `physcpubind` option the process can only be executed on a particular set of CPUs. This accepts physical CPU numbers as shown in the processor fields of `/proc/cpuinfo`. Each rx7640 has 16 cores, enumerated from 0 to 15 (see above the `show` option output).

3.3 Effects of using explicit data and thread placement

Next we will show an example using `numactl` to map data and threads explicitly. We will focus on evaluating its influence in the performance of the sparse matrix-vector product (SpMV). Tests have been performed on a rx7640 node of the FinisTerrae supercomputer. As matrix test set we have selected fifteen square sparse matrices from different real problems that represent a variety of nonzero patterns.

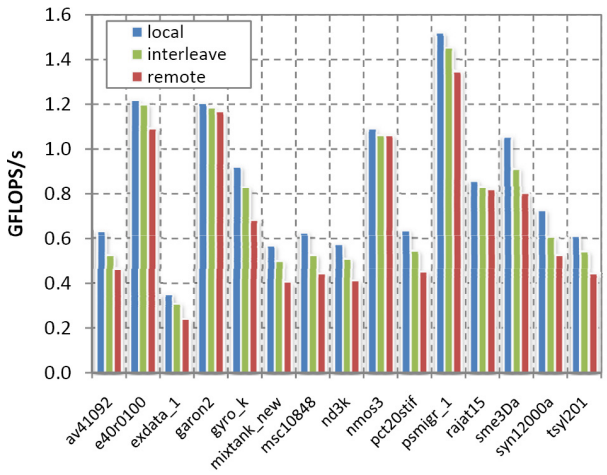


Figure 4: Influence of the data allocation on a rx7640 node.

These matrices are from the University of Florida Sparse Matrix Collection (UFL) [6].

First, the influence of the data allocation in the SpMV performance is studied. As example, we show in Figure 4 the behavior of SpMV code when using two threads mapped to cores 8 and 12 (see Figure 2). Data is allocated in the memory module of the cell of cores 8 and 12 (local), in the memory of the other cell (remote), and using the interleave policy. That is, using:

`numactl --membind=0 --physcpubind=8,12` (local)

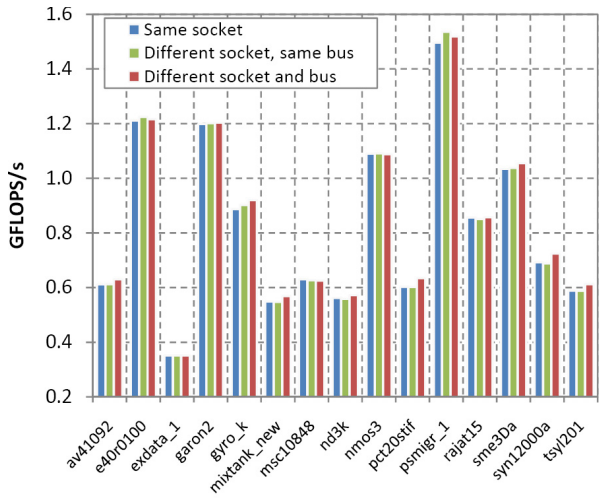
`numactl --membind=1 --physcpubind=8,12` (remote)

`numactl --membind=2 --physcpubind=8,12` (interleaving)

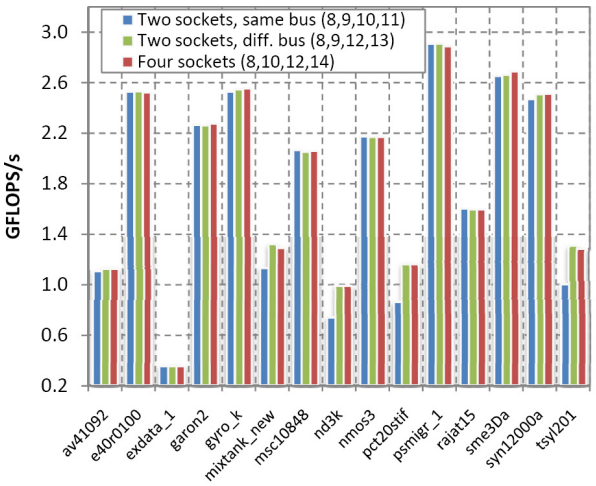
As is expected, an important degradation in the performance is observed when data is allocated on a remote cell with respect to local memory accesses. In particular, an average decrease of 20.7% is obtained, ranging from 3% in the best case (matrix `garon2`) to 32% (matrix `exdata_1`). Results point out that the worst behavior is achieved for big size matrices. When using interleave memory, this degradation is in average about 10%. Note that depending on the data distribution for a particular matrix, interleave policy offers performance close to local accesses (matrix `e40r0100`) or to remote accesses (matrices `nmos3` and `rajat15`). The overall behavior analyzed in this example is also observed when a different number of threads is considered.

Therefore, we conclude that data allocation has a great influence in the performance of SpMV on FinisTerrae. When threads are mapped to cores in the same cell, it is better to allocate the data in the memory module of the same cell. However, if cores in both cells must be used, the allocation of the data in the interleaving memory makes sense due to accesses to remote memory are very costly.

Now different aspects related to the influence of thread allocation in the SpMV performance are studied. First, we have focused on evaluating the influence of mapping threads to the same processor (socket). Figure 5(a) shows the performance achieved using two threads for several mapping configurations: same



(a)



(b)

Figure 5: Influence of the thread allocation on a rx7640 node: using two (a) and four threads (b).

socket (for example, cores 8 and 9), different socket and same bus (cores 8 and 10) and different socket and bus (cores 8 and 12). Note that data is allocated in the same cell where threads are mapped. Results point out that the influence of mapping the two threads to the same or to different socket sharing the bus is very low. In particular, the average difference in the performance is only about 0.2%. However, some improvements (up to 5%) are observed when mapping threads to cores that do not share the bus. This is particular true in the case of big size matrices (for example, matrices `mixtank_new` and `tsyl201`). Therefore, the contention of the bus to memory seems to have an impact on the SpMV performance.

In order to confirm the behavior observed previously, we have tested the SpMV performance using four threads mapped to: two sockets sharing the bus (cores 8,9,10,11), two sockets in different buses (cores 8,9,12,13) and four sockets (cores 8,10,12,14). Performance results obtained using these configurations

are shown in Figure 5(b). Several conclusions can be made.

First, the same trend is observed with respect to the results using two threads (Figure 5(a)). That is, better performance is achieved when threads are mapped to sockets that do not share the bus (configuration 8,9,12,13). In this case, improvements mean in average about 8% (higher than 30% for some matrices as `nd3k`, `pct20stif` and `tsyl201`). Therefore, the influence of the bus become more significant as the number of threads increases. Note that average improvement is about 2% when using two threads that do not share bus (see configuration "different socket and bus" in Figure 5(a))

Second, the impact of the bus for small matrices is minimal in such a way that the three considered mappings obtain similar results (for example, matrices `nmos3` and `rajat15`). And finally, there is no significant differences among using four sockets (8,10,12,14) and two sockets in different bus (8,9,12,13). An explanation to this behavior is that in both cases there are two threads mapped to cores that share the bus (see Figure 2).

Acknowledgements

We gratefully thank Hewlett-Packard (HP) for its valuable support in the development of this report.

References

[1] Hewllet-Packard Company. *HP Integrity rx7640 Server Quick Specs.*

[2] D. Culler and J. P. Singh. *Parallel Computer Architecture.* Morgan Kaufmann, 1998.

[3] J. A. Lorenzo, F. F. Rivera, P. Tuma, and J. C. Pichel. On the influence of thread allocation for irregular codes in NUMA systems. In *Proc. of the Int. Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2009.

[4] S. Eranian. *The Perfmon2 interface specification.* http://www.hpl.hp.com/techreports/2004/HPL-2004-200R1.html.

[5] J. Antony, P. P. James, and A. P. Rendell. Exploring thread and memory placement on NUMA architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. In *Proc. of the Int. Conf. on High Performance Computing (HiPC)*, pages 338–352, 2006.

[6] T. Davis. University of Florida Sparse Matrix Collection. *NA Digest*, 97(23), June 1997. http://www.cise.ufl.edu/research/sparse/matrices.