Memory Access Characterization of OpenMP Workloads on a Multi-core NUMA Machine

Christiane Pousa Ribeiro, Alexandre Carissimi, Jean-François Méhaut

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Memory Access Characterization of OpenMP Workloads on a Multi-core NUMA Machine

Christiane Pousa Ribeiro — Alexandre Carissimi — Jean-François Méhaut

N° 7330

June 2010

Thème NUM

ISSN 0249-6399 ISRN INRIA/RR--7330--FR+ENG

INSTITUT NATIONAL
DE RECHERCHE
EN INFORMATIQUE
ET EN AUTOMATIQUE

INRIA

centre de recherche
GRENOBLE - RHÔNE-ALPES

Memory Access Characterization of OpenMP Workloads on a Multi-core NUMA Machine

Christiane Pousa Ribeiro, Alexandre Carissimi , Jean-François Méhaut

Thème NUM — Systèmes numériques
Équipe-Projet MESCAL

Rapport de recherche n° 7330 — June 2010 — 26 pages

Abstract: Nowadays, on hierarchical shared memory multiprocessors with Non-Uniform Memory Access (NUMA), the number of cores accessing memory banks is considerably high. Such accesses produce more stress on the memory banks, generating load-balancing issues, memory contention and remote accesses. In this context, it is important to have a good understanding of memory access patterns and what are the influences of data placement on such patterns. In this document, we have investigated memory accesses behavior of microbenchmarks and benchmarks over a ccNUMA platform with multi-core processors. Additionally, we have evaluated a set of memory policies that were used to place data among the machine memory banks. Our results have shown that an appropriate selection of data placement, considering the memory accesses, can generated great improvement gains.

Key-words: multi-core processors, NUMA architecture, memory affinity, numerical application, performance evaluation, characterization

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe, 38334 Montbonnot Saint Ismier
Téléphone : +33 4 76 61 52 00 — Télécopie +33 4 76 61 52 52

Memory Access Characterization of OpenMP Workloads on a Multi-core NUMA Machine

Résumé : Sur les nouvelles machine hiérarchise multiprocesseurs à mémoire partagée avec ses accès mémoire non-uniforme (NUMA), le nombre de coeurs que font des accès aux banques mémoire est considérablement grand. Ces accès produisent des problèmes d'équilibrage de charge, contention de mémoire et les accès distants coûteux. Dans ce contexte, il est important d'avoir une bonne compréhension des ces accès de mémoire et quelles sont les influences de placement des données sur de tels modèles. Dans ce document, nous avons étudié le comportement d'accès mémoire utilisant benchmarks sur une plate-forme cc-NUMA avec processeurs multi-core. Nous avons aussi évalué un ensemble de politiques de la mémoire qui ont été utilisés pour placer des données sur les banques mémoire de la machine. Nos résultats ont montré qu'une sélection appropriée de placement des données, en considérant les accès mémoire, peut générer des grands améliorations de performance.

Mots-clés : architectures NUMA, multi-core processeur, affinité mémoire, application numérique, étude de performances,catégorisation

1 Introduction

The concept of ccNUMA machines was first proposed on 80's/90's to overcome scalability problems on classical symmetric multiprocessors [1]. A ccNUMA platform is a large scale multi-processed system in which the processing elements are served by a shared memory that is physically distributed into several memory banks interconnected by a network. NUMA architectures combine the efficiency and scalability of MPP (Massively Parallel Processing) with the programming facility of SMP machines [2]. Because of the network interconnection, memory access costs may vary, depending on the distance between processing units and memory banks. Thus, time spent to access data is conditioned by the distance between the processor and memory bank where data was placed. The memory access by a given processor can be local (data is close) or remote (it has to use the interconnection network to access the data) [2, 3].

The increasing number of cores per processor and the efforts to overcome the memory wall problem remain a problem in High Performance Computing (HPC). Due to this, cache-coherent Non-Uniform Memory Access (ccNUMA) platforms are coming back as computing resources for numerical scientific HPC. Besides the non-uniformity on memory access that were already present on 80's/90's ccNUMAs, on nowadays ccNUMAs (e.g., machines based on AMD Opteron and Intel Nehalem processors), the number of cores accessing memory banks is considerably larger than in the older ccNUMA machines (e.g., DASH and SGI). These accesses produce more stress on the memory banks, generating load-balancing issues, memory contention and remote accesses. As these machines are extensively used in HPC, it is important to reduce memory access costs. To do this, we have to understand memory access patterns and what are the influences of data placement on such patterns [4].

In this work, we have investigated memory accesses behavior of microbenchmarks and benchmarks over a AMD Opteron ccNUMA platform with multi-core processors (dual core) [5]. We have focus our evaluation on numerical scientific parallel benchmarks that have as main characteristic high memory consumption (Stream benchmark [6] and NAS Parallel benchmark [7]). The memory accesses behavior investigation has been based on three types memory operations (read access, write access and read/write access), how data are accessed (regular, irregular and random accesses) and how work were distributed to threads. Our results have shown that memory accesses behavior is related to data and threads placement on the machine nodes. To confirm such results, we have used a set of memory policies (MAi interface [8]) on NAS Parallel Benchmarks to better distribute data and improve performance.

The report is structured as follows. In Section 2 we introduce benchmarks we have used for the characterization. Section 3 describe the ccNUMA platform that has been used on the workload characterization . We present in Section 4 the workload characterization and discuss the obtained results. Finally, in the last section we present our conclusions and future work.

2 Microbenchmarks and Benchmarks

In this section, we present the microbenchmark and the benchmark we have used in this work to characterize memory access patterns. We first present two

microbenchmarks, Stream [6] and Bandwidth [9]. After that, we present the BenchIt Benchmark [10, 11] and NAS Parallel Benchmark [12, 7].

2.1 Stream

Stream is a simple benchmark that is largely used to memory performance evaluation [6]. It is a synthetic benchmark application that measures memory bandwidth and the computation rate vector for complex memory access patterns. To compute such metrics, Stream uses three vector and four operations (copy, scale, add, triad). Additionally, in order to avoid any cache influence on the results, each vector has a large number of elements.

Table 1 shows Stream operations and their specification. As we can observe, all operations are performed with double vectors. Copy operation allows user to measure transfer rates between processing unit and memory bank. The operation scale adds a multiplication by a scalar to the copy operation. Sum allows users to verify memory system performance when multiple loads/stores are performed. The operation triad is a merge of all operations (copy, scale and sum).

Table 1: Stream operations

Operation Name	Operation	Data type
Copy	a[i] = b[i]	double
Scale	a[i] = q*b[i]	double
Sum	a[i] = c[i]+b[i]	double
Triad	a[i] = c[i]+q*b[i]	double

In this work, we have used the C implementation of Stream with OpenMP for code parallelization. For the parallel version of Stream, threads share all the three vectors. However, each thread computes a chunk of the workload. The chunk size is equal for all threads, except for the last thread that can has a larger chunk size if the number of elements of vectors are not divisible by the number of threads. In our experiments, we have used 2 millions of elements for each vector of Stream (larger than the cache size of the ccNUMA platform).

2.2 Bandwidth

Bandwidth is a microbenchmark that has as main goal to measure memory bandwidth of platforms. It helps users to have a better idea of real bandwidth of memory subsystem [9].

The benchmark is composed by a set of functions that performs sequential read and write on main memory and cache L2. Buffers large than cache sizes for main memory tests are allocated with malloc and then pointers are used to access data on read and on write. The same approach is performed for cache tests. At the end of the execution the benchmark generates as output the bandwidth in MB/s.

2.3 BenchIT

BenchIT is a framework that allows users to perform measurements and analysis of high performance systems [10]. The framework only support systems that are UNIX based (POSIX compliant shell and C-compiler). It is composed by several applications (e.g. reflection) and kernels (e.g. BLAS, FFT, Jacobi) that provides different levels of perform measurements and analysis.

This framework has been proposed as a project and it aims at providing the computational tools to evaluate a machine in different contexts. Additionally, BenchIT provides an graphical interface that helps users in plotting and visualizing their results.

Since we are interested in memory characteristics, in this work, we will work with a subset of benchmarks from BenchIT.The chosen benchmark is named x86 memory benchmark and it allows us to measure memory bandwidth (parallel and sequential way) and memory latecy [11].

2.4 NAS Parallel Benchmarks

NAS Parallel Benchmarks (NPB's) is a benchmark derived from computational fluid dynamics (CFD) codes and it is composed by a set of applications and kernels [7]. NPB's applications and kernels perform representative computation and data communication of CFD codes. Such benchmark has been implemented on different languages and using different strategies for code parallelization.

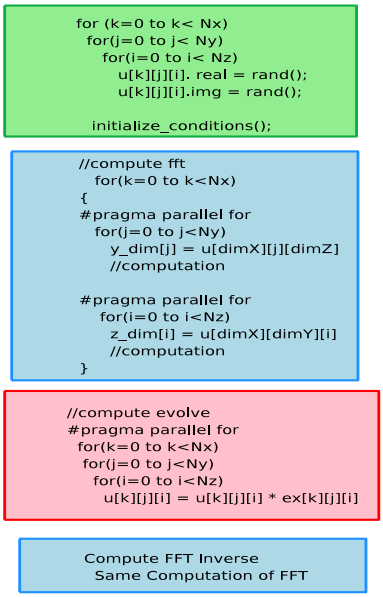


Figure 1: Fast Fourier Kernel

From NPB's, we selected five kernels/applications: fast Fourier Transform (FFT), Multigrid (MG), lower and upper triangular system solution (LU), Conjugate Gradient method (CG), solution of pentadiagonal equations (SP) and block tridiagonal equations solution (BT). These kernels were chosen due to

their memory access patterns (both have irregular data access patterns) and different data structures types. Additionally, they represent important classes of algorithms and computations.

FFT is a kernel that computes the fast transform of Fourier for three dimensional systems. The application works with complex numbers that are represented with structures. The computation is done in one direction by step and each thread computes Z imaginary planes. There are three main steps in the FFT computation and data are shared just in the second step. In our experiments, we used a $512x256x256$ matrix. Such kernel was implemented in C using OpenMP to code parallelization. Figure 1 shows a schema of the application.

MG is a kernel that uses a V cycle MultiGrid method to calculate the solution of the 3D scalar Poisson equation. The main characteristic of this kernel is that it tests both short and long distance data movement. In our experiments, we have used $102x102x102$ elements for matrices. Such kernel was implemented in C using OpenMP for code parallelization.

LU is a well know application that solves a 3D seven-block-diagonal system using lower-upper triangular systems solution. This application works with regular sparse matrices and it uses symmetric successive over relaxation(SSOR) operations. In our experiments, we have used $102x102x102$ elements for matrices. LU was implemented in C using OpenMP for code parallelization.

```
#pragma omp parallel for
    for (j=0 to j<elements)
    //computation

#pragma omp parallel for reduction()
   for (j=0 to j<columns)
    //computation


#pragma omp parallel for
  for (i=0 to i<rows)
     mat_stat[i] = mat[-+i] .....


 #pragma omp parallel for
for (i=0 to i<elements)
  for (j=0 to j<max)
    mat[i] = mat_stat[index[j]] ......
```

Figure 2: Conjugate Gradient Kernel

CG is also a kernel that uses a conjugate gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. This kernel tests unstructured vector computations and communications. It uses a matrix with randomly generated locations of entries which gives a large amount of cache misses. The input parameter of this kernel is the size of the array that will be used for computation. In this case, we used an array of size 75000. Such kernel was implemented in C using OpenMP for code parallelization. Figure 2 presents a schema of the application.

SP is an application that computes the solution for a scalar pentadiagonal systems. The computation is donne with three dimensions matrices but, in just one direction by step. This application has one particularity, the number of process/threads used in the computation must be square. Furthermore, this

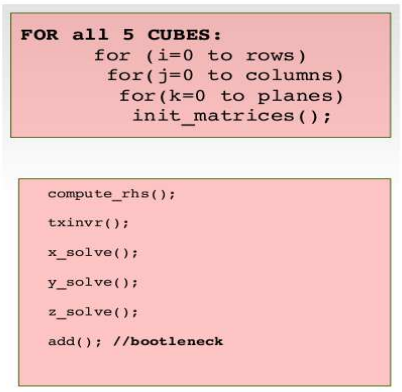


Figure 3: Scalar Pentadiagonal Application

application has a bottleneck function that minimizes the scalability of the code in some architectures, as describe in [13]. In our experiments, we have used $102x102x102$ elements for matrices. Such kernel was implemented in C using OpenMP for code parallelization. Figure 3 presents a schema of the application.

```
   c----BT:
   call compute_rhs

   call x_solve

   call y_solve

   call z_solve

   call add


c-- ADD function
!$omp parallel do default(shared) private(i,j,k,m)
      do     k = 1, grid_points(3)-2
         do     j = 1, grid_points(2)-2
            do     i = 1, grid_points(1)-2
               do    m = 1, 5
                  u(m,i,j,k) = u(m,i,j,k) + rhs(m,i,j,k)
               enddo
            enddo
         enddo
      enddo
```

Figure 4: Block-Tridiagonal Application

BT is an application that computes a solution for multiple and independent systems of non diagonally dominant. As in SP, the computation is donne with three dimensions matrices but, in just one direction by step. The steps on each dimension are represented by the functions: solve_x, solve_y and sole_z. In our experiments, we have used $102x102x102$ elements for matrices. Such kernel was implemented in C using OpenMP for code parallelization. Figure 4 shows a schema of the application.

3 ccNUMA Platform

The ccNUMA platform used on this work is an eight dual core AMD Opteron Processor 875 2.2 GHz (Figure 5). It is organized in eight nodes of two cores with 1 MB of cache L2 for each core. It has a total of 32 GB of main memory (4 GB of local memory). Each node has three connections (HyperTransport [5]) which are used to link with other nodes. However, nodes zero and one have just two connections to other nodes. On such nodes, the third connection is used to connect input and output devices. The connections give different memory latencies for remote access by nodes of the platform.

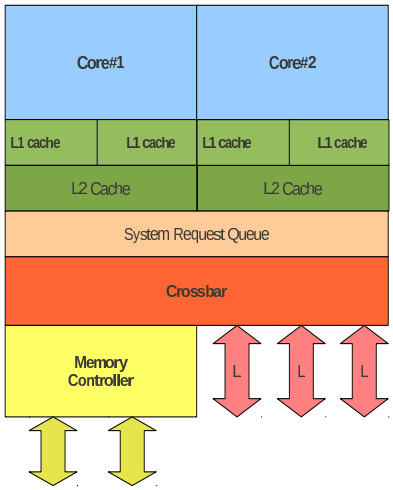


Figure 5: AMD Opteron Processor 875

The compiler that has been used for the OpenMP code compilation was the GCC (GNU C Compiler) version 4.3. The operating system that has been used in this machine is Linux version 2.6.23-1-amd64 and the distribution is Debian with support for NUMA architecture (system calls, user API and numactl). A schematic representation of this machine is given in Figure 6.

3.1 NUMA Impact

On the platform described on above section, there are NUMA penalties caused by different memory access costs. To better understand the selected platform, we have been computed NUMA penalties by performing some experiments with Bandwidth and Stream microbenchmarks and numactl tool [14]. The numactl tool allows user to select where to place data and threads among the machine nodes. It binds data and threads to memory banks and cores.

We have run the Bandwidth benchmark on the Opteron machine to measure the bandwidth inside a node. Table 2 shows the bandwidth that has been obtained for cache L2 and main memory of a node. Since nodes 0 and 1 have connections to input and output, we have used numactl to run the application on node 2 to avoid any I/O interference.

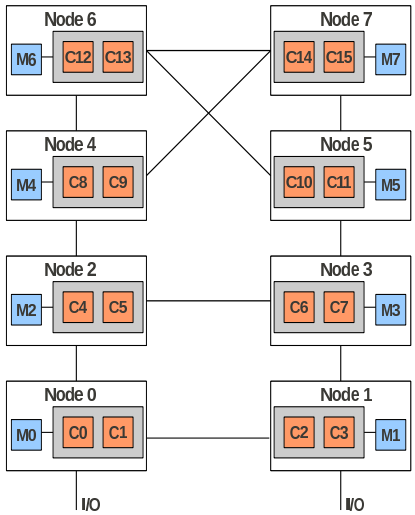


Figure 6: ccNUMA Platforms

Table 2: Bandwidth of Cache and Main Memory for a Node

	L2 cache read	L2 cache write	Main Memory read	Main Memory write
Bandwidth	3532.05 MB/sec	2508.74 MB/sec	1383.69 MB/sec	1157.05 MB/sec

In Table 2, we can observe that bandwidth for read operations are larger than for write operations. In order to perform a write operation, the machine has to read the target data from memory into cache before performing the write. Due to this, more data has to be transfered to perform the operation.

Bandwidth benchmark is useful to measure memory performance of one node of the NUMA machine. Additionally, it has very simple access patterns (read and write). In order to evaluate memory performance of more complex access patterns and of concurrent accesses on memory we have performed some experiments with Stream benchmark.

Stream have four memory operations and for thiw work we have selected the most complex and significant operation for our experiments. The selected operation was the triad operation, in which three arrays and one scalar are used (as described on previous section).

Since we are interested on evaluate ccNUMA platform memory access costs and how multi-cores can impact on such costs, we have select as metrics: average execution time, bandwidth and NUMA factor [1]. Regarding to the average time to access some data from node i to node j, we have used numactl to place data and threads over the machine. We have done this computation for all nodes of the machine, for read and write operations. Considering bandwidth, we have

[1]NUMA factor is the ration between the remote latency and local latency to access some data

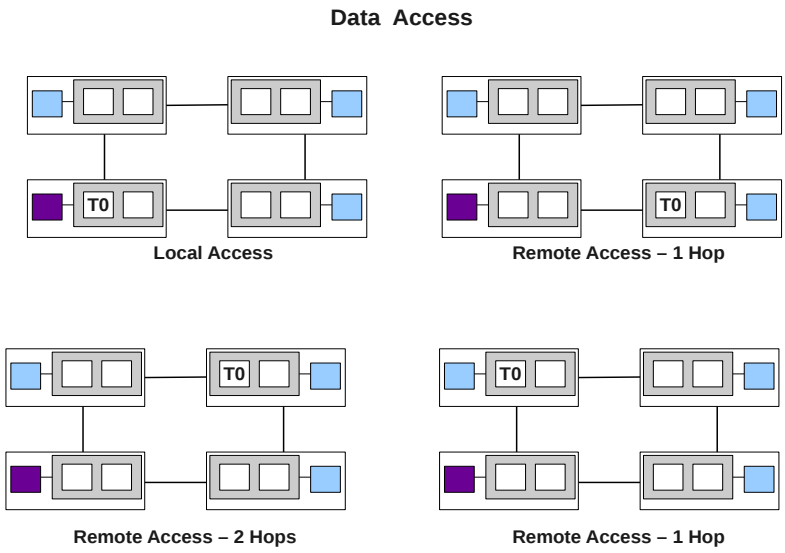


Figure 7: Local and Remote Data Access Performed by Thread T0.

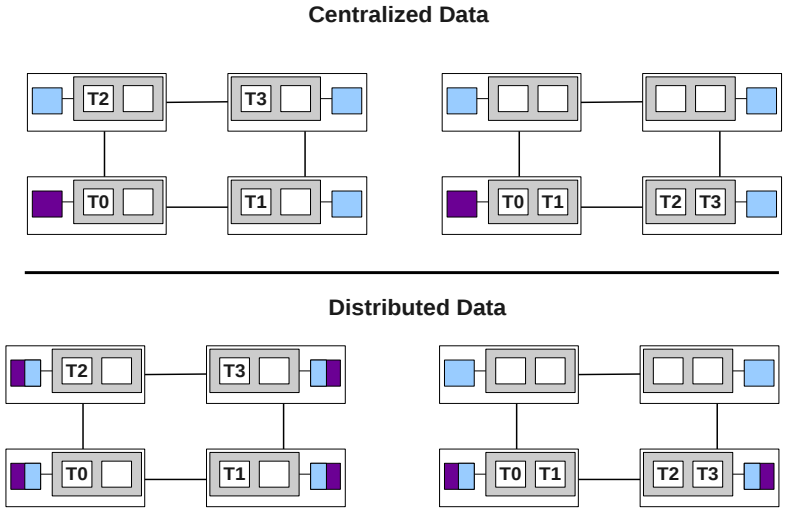


Figure 8: Centralized and Distributed Data and Threads Placement.

computed the amount of data transfered per second intra and inter nodes, using different number of cores (we used Stream for this). For NUMA factor, we have

calculated it for all machine nodes. We have placed data on one node and then, we have accessed it by a different node using numactl.

The results have been obtained through the average of several executions varying the number of threads (from 1 to twice maximum number of cpus/cores of the platform) and data placement strategies. Memory accesses costs have been computed by placing data and threads on different nodes of the machine. Considering to data access, we have placed data on different nodes (local or remote to thread) of the machine in order to compute the impact on memory access costs (Figure 7). Different data and thread placements have been also used (8). In this case, we have placed data in two fashions: a centralized way (allocate on only one memory bank) or a distribute way (allocate some memory banks). Considering threads, we have binded them to cores, using all the cores of a processor or using just one core per processor. Results have presented a low standard deviation (1.22), since all experiments have been done with exclusive access to the ccNUMA machine. Our results are organized by metrics, we first present the results for latency. After that, we present bandwidth and NUMA factor results.

Table 3: Average Time in seconds for Local and Remote Triad Operation.

Operation Name	Local	Adjacent Node	2 Hops Node	3 Hops Node
Triad	0.279	0.319 (x1.14)	0.325 (x1.16)	0.386 (x1.38)

Table 3 presents average time for sequential Triad operations for local and remote data (data size of 30.5MB). The results show that remote operation costs are more expensive than local ones. The protocol used on Opteron machines to assure cache coherence is the MOESI protocol [15]. On such protocol, write operations are more expensive than read operations. On write operation more traffic is generated over the network, because the protocol have to invalidate/update data copies of other nodes.

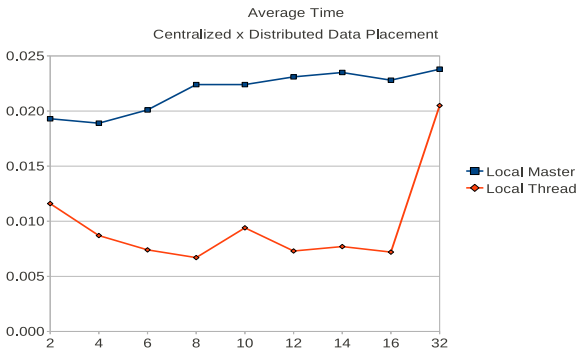


Figure 9: Average time (s): Centralized x Distributed Data Placement

Figure 9 shows the average time obtained with Triad operation from Stream benchmark when data is allocated in just one memory bank (Local Master

curve) and when data is distributed among the machine memory banks. We can observe, that data placement on just one memory bank have generated worst latencies. In this case, all threads access concurrently the same memory bank, generating remote accesses and memory contention. Remote accesses are generated because some threads are distant from the memory bank where data was placed and must pass through other nodes to access data. Considering memory contention, several threads uses the same interconnection links to access the required data. Someone may think that this is an idiot way of placing data but several parallel applications have been implemented in this fashion. On such applications, the master thread is responsible for allocate and initialize data. Thus, this thread touches data firstly and in some operating systems (e.g., Linux), data is placed on the node that first touched it.

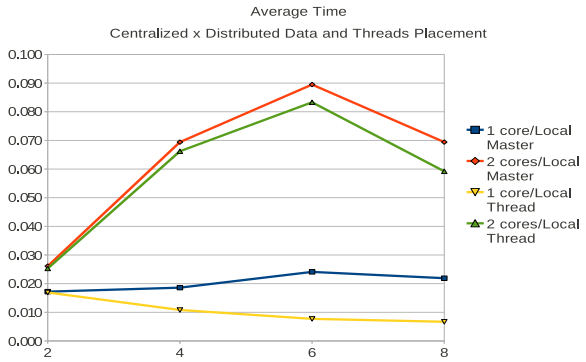


Figure 10: Average Time (s): Centralized x Distributed Data and Threads Placement

In Figure 10, we present average time obtained with data and thread placement presented in Figure 8. The best times have been obtained with local thread data placement. This is mean that when threads have their data on the same node the number of remote access is minimized. An important result that must be observed in this figure is latencies with two cores. The usage of two cores has resulted in worse performance when compared to the usage of only one core per processor (curves yellow and green). In this case, results have been influenced by memory contention inside the nodes. Thus, we must avoid to use all the processor cores when the application number of threads is smaller than the machine number of cores.

In Table 4, we present the NUMA factor for all nodes of this machine. We have used a tuned version of Stream benchmark to compute the NUMA factor. Data size have been larger than cache size. Considering our experiments, the NUMA factor on this platform varies from **1.024** to **1.55**, which means that on ccNUMAs based on Opteron processors, remote access are not expensive. Some variations on NUMA factor values may be expected, since it depends on the memory access patterns of the application and the cache usage.

Figure 11 shows bandwidth results that have been obtained with Stream Benchmark. From one to eight cores the bandwidth increases almost linearly.

Table 4: NUMA Factor for Opteron Platform Nodes

	Node 1	Node 2	Node 3	Node 4	Node 5	Node 6	Node 7	Node 8
Node 1	1.0000	1.55	1.2122	1.1974	1.2374	1.4388	1.3093	1.3057
Node 2	1.55	1.0000	1.1872	1.2959	1.4606	1.2284	1.3071	1.2589
Node 3	1.2122	1.1872	1.0000	1.1808	1.1632	1.2234	1.1795	1.2305
Node 4	1.1978	1.2959	1.1808	1.0000	1.3860	1.2576	1.0337	1.1648
Node 5	1.2374	1.4606	1.1632	1.3860	1.0000	1.1020	1.3877	1.2820
Node 6	1.4388	1.2284	1.2234	1.2576	1.1020	1.0000	1.1242	1.1318
Node 7	1.3093	1.3071	1.1795	1.0337	1.3877	1.1242	1.0000	1.024
Node 8	1.3057	1.2589	1.2305	1.1648	1.2820	1.1318	1.024	1.0000

However, if the system uses more than eight cores the overall performance decreases. In this case, interconnection links start to be saturated.

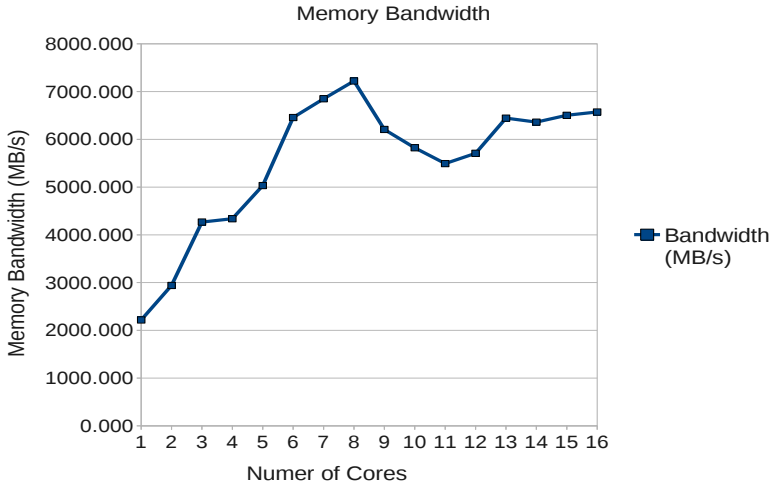


Figure 11: Memory Bandwidth

In order to complete our analysis of NUMA penalties on the selected platform, we have also performed some experiments with BenchIT benchmark. BenchIT allows us to have a deep understanding of memory bandwidth and read latency of the machine. To measure such metrics the benchmark has two kernels in which, several memory accesses are performed on arrays of different sizes. Additionally, this benchmark express different memory access patterns (e.g. multiple reader, multiple writer and single reader).

In Figure 12, we present memory bandwidth for different memory accesses patterns and number of threads per cpu. We can notice that aggregate bandwidths for multiple parallel threads are similar for read, write and read/write accesses. In these experiments, each thread allocates data locally, but after they

accesses the others threads data. Due to this, memory bandwidth is saturated on the machine (bandwidth peak 3500 MB/s). One important result in this figure is the similarity of results obtained for one thread per cpu and two threads per cpu.

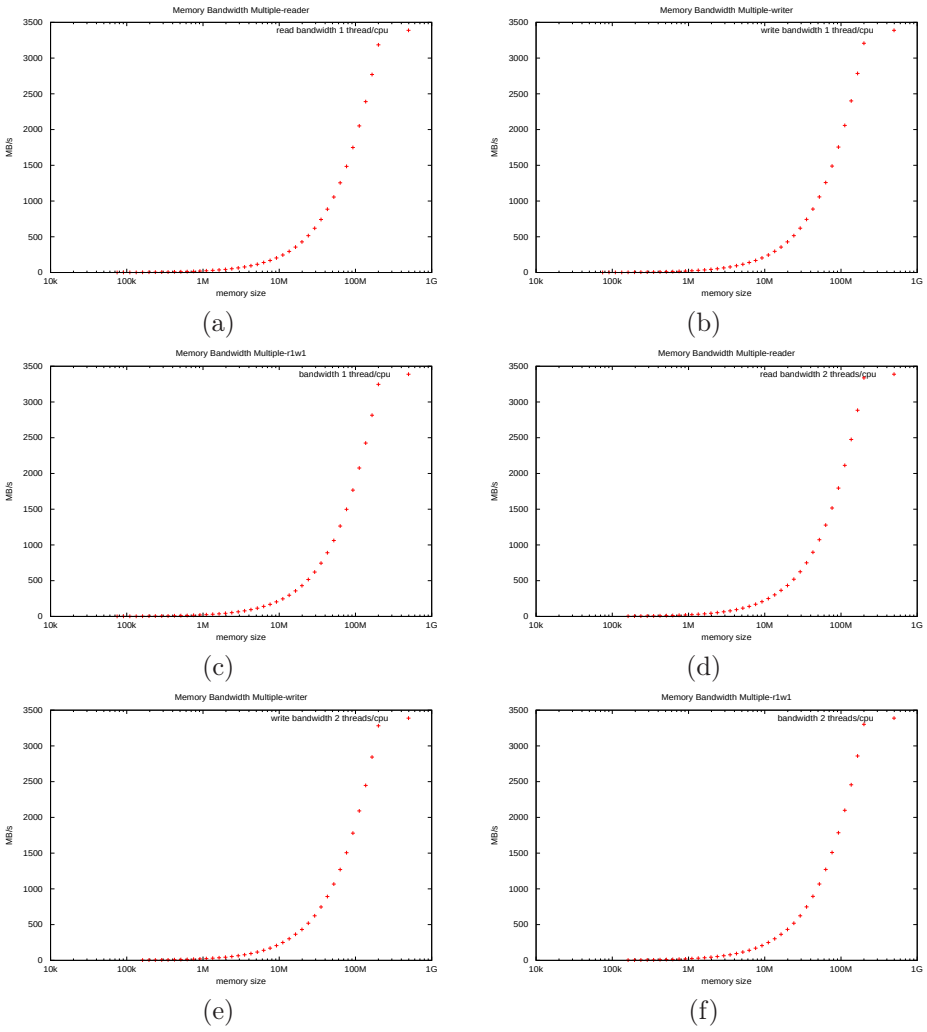
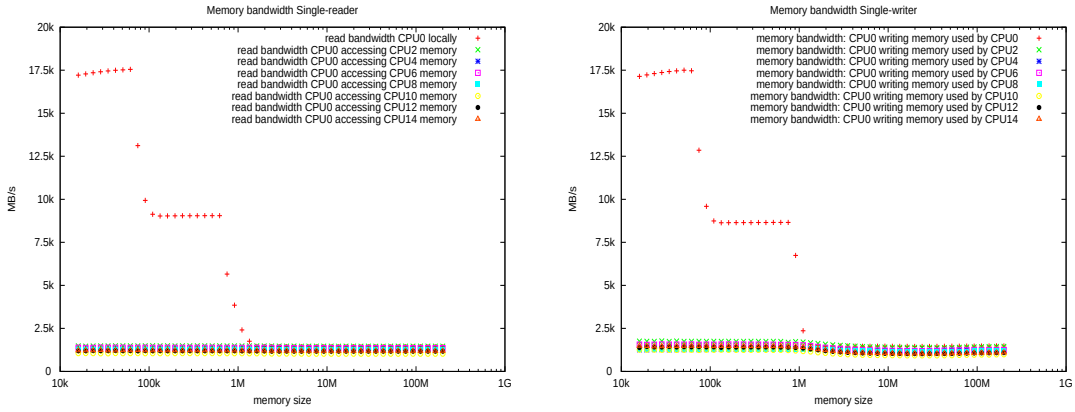


Figure 12: Memory Bandwidth: (a) Multiple Reader 1 thread/cpu (b) Multiple Writer 1 thread/cpu (c) Multiple Reader/Writer 1 thread/cpu (c) Multiple Reader 2 threads/cpu (e) Multiple Writer 2 threads/cpu (f) Multiple Reader/Writer 2 threads/cpu.

Figure 14 shows memory bandwidth for single accesses on data from a cpu i to cpu j for read and write operations. We can observe that read bandwidth for local accesses are much larger than for remote accesses. These results let us to conclude that it is important to manage data distribution in order to avoid the usage of the network interconnection between the NUMA nodes. Additionally,

one can noticed that for all data sizes read bandwidth is constant for shorter (CPU 0 accessing memory of CPU2) and longer accesses (CPU 0 accessing memory of CPU14). Regarding to write bandwidth, we observe a slightly difference when compared to read bandwidth. In this case, more data is transfered between NUMA nodes, since for written operations data must be firstly read and then written.



(a) (b)

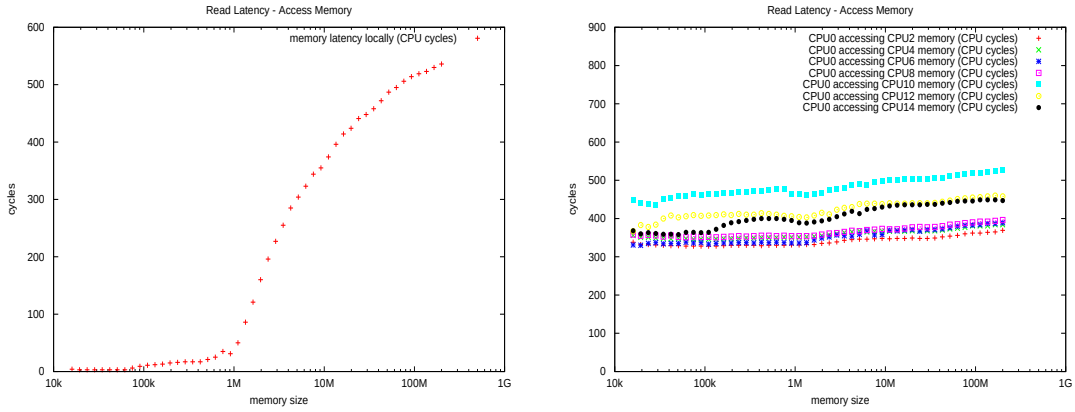
Figure 13: Memory Bandwidth: (a) Single Reader (b) Single Writer.

Read latencies for local and remote data accesses are presented in Figure **??**. We can observe that latencies for local accesses (Figure **??** (a)) are negligible until the size of the cache L2. When data size is larger than cache L2 the number of cycles needed to access data becomes higher because more cache misses and main memory accesses are generated. Contrary to local access costs, remote accesses are expensive even to small data sets (Figure **??** (b)). This is related to the communication costs, on such accesses the network is used to get data from remote memory banks.

The experimental results presented in this section have led us to conclude that on this platform, it is important to spread data among the machine nodes (bandwidth optimization) trying to place it close to threads (latency minimization). In this case, we can think about memory policies that spread data in a round-robin way.

4 Workload Characterization

In this section, we present the workload characterization for the Opteron cc-NUMA platform. We first present the characterization based on memory access pattern. After that, we present some experiments we have performed in order to use our characterization to improve performance of numerical scientifical benchmarks.



(a) (b)

Figure 14: Memory Latency: (a) Local access (b) Remote access.

4.1 Memory Access Characterization

In this work, the characterization of memory access patterns have been done using a tuned version of Stream benchmark. In this section, we aim at providing a good understand of the NUMA impact on memory accesses. We have considered three types of memory operations: read only, write only and read/write. We have chosen these operations because they are the basic operation of most part of computations. The operations, read only, write only and read/write have been added on Stream by us. Additionally, we have also considered different access on the vector (regular, irregular and random) and different strategies to distribute workload for each thread (static and dynamic).

Regular

```
//initialization
#pragma omp parallel for
for (j=0; j<N; j++)
      c[j] = a[j] = 1.0;

  .......

//copy
#pragma omp parallel for
      for (j=0; j<N; j++)
         c[j] = a[j];

//add
#pragma omp parallel for
      for (j=0; j<N; j++)
         c[j] = a[j]+b[j];
```

Irregular

```
//initialization
#pragma omp parallel for
for (j=0; j<N; j++)
      c[j] = a[j] = 1.0;

  .......

//copy
#pragma omp parallel for
      for (j=N; j>=1; j--)
         c[j] = a[j];

//add
#pragma omp parallel for
      for (j=0; j<N; j++)
         c[j] = a[j]+b[j];
```

Random

```
//initialization
#pragma omp parallel for
for (j=0; j<N; j++)
      c[j] = a[j] = 1.0;

  .......

//copy
#pragma omp parallel for
  for (j=0; j<N; j++)
      c[page[j]] = a[page[j]];

//add
#pragma omp parallel for
  for (j=0; j<N; j++)
      c[page[j]] = a[page[j]]
                  +b[page[j]];
```

Figure 15: Vector Access

By regular access on vector, we mean that threads access the same set of data in different phases of the application whereas in irregular access, data set may be different. The regular and irregular access have led us to simulate the impact of local and remote accesses on application performance. The random access assures that threads will touch different memory pages on every memory access and different threads may access the same page (simulates memory contention). Considering workload distribution, in static distribution, loop iterations are divided into blocks and then assigned to threads in a static fashion. In dynamic distribution, the difference to the static one is that blocks are assigned to threads during runtime. In such distribution, if a thread finishes its work, it can steal blocks from other threads.

In order to have different access on vectors and workload distribution, we have made some modifications on Stream Benchmark source code. Different access on vectors have been implemented by using different ways to index vectors (Figure 15). Considering workload distribution, we have included on Stream source code the OpenMP clauses **schedule(dynamic)**. Besides vector access and workload distribution, we have also done some changes on Stream source code using MAi interface [8] to apply memory policies on its data and better control data placement. The memory policies that have been used in these work are described in Table 5.

Table 5: MAi Memory Policies Description

Memory Policy	Description
Bind_block	data is divided into blocks depending on the number of threads and placed close to the thread that will use it
Cyclic	data is placed in the memory blocks in a linear round-robin way
Skew_mapp	a page i is allocated in the node $(i + \lfloor i/M \rfloor + 1)\ mod\ M$, where M is the number of memory blocks
Prime_mapp	a two-phase strategy. In the first phase, the policy places data using *cyclic* policy in (P) virtual memory banks, where P is a prime greater or equal to M (real number of memory banks). In the second phase, memory pages previously placed in the virtual memory blocks are reordered and placed into the real memory banks also using the *cyclic* policy
Random	memory pages are placed randomly in the NUMA nodes, using a random uniform distribution

Based on these parameters, we have performed some experiments on the Opteron ccNUMA platform. The results have been obtained through the average of several executions varying the number of threads of 2, 8 and the maximum number of cpus/cores of the platform (one thread per core). These results have presented a low standard deviation, since all experiments have been done with exclusive access to the ccNUMA machine. Our results are organized by memory access operations and for each operation, we show results for the three vector access types and workload distribution.

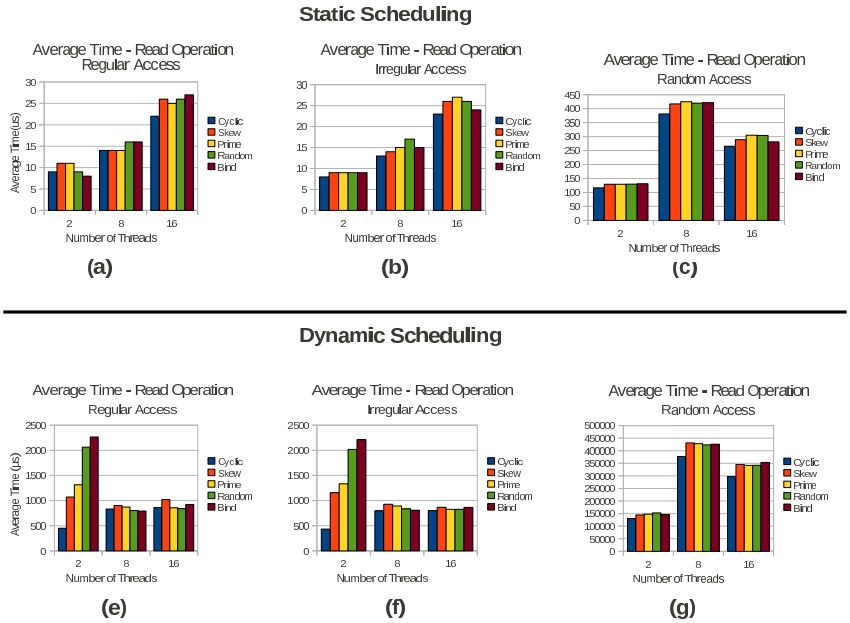


Figure 16: Average Time for Read Operation

In Figure 16, we can observe the average time for read operation with different memory policies, vector accesses and workload distribution. In this figure, we show the results for two threads (only two nodes of the machine have been used), eight threads (all the machine nodes have been used, one thread per core) and sixteen threads (all nodes and cores have been used).

As briefly discussed on section NUMA Impact, the read operation on Opteron machines is less expensive than the write operation. Due to this, its impact on the performance of the application is less important than write operations. However, we have also shown that on NUMA architectures even read operations can have an important impact on the application performance if they are executed on remote data. For both scheduling strategies, on general read operation has presented better performance when data was distributed using *cyclic* memory policy. On read operations, the cache coherence protocol has no influence in the performance, because it does not have to update or invalidate any copy. Furthermore, once data have been touched for read, it will still in the cache of the core. Thus, only the first access on data will impact on the application performance.

Figure 17 shows the average time for write operation with different memory policies, vector accesses and workload distribution. In this figure, we present results for two threads, eight threads and sixteen threads. On general, for write operation best performance have been obtained with *bind_ block* and *cyclic* memory policies.

Considering static scheduling (Figure 17 (a), (b) and (c)), we have observed that the best memory policy to place data among the ccNUMA platform depends on the number of threads. *Cyclic* memory policy have presented better

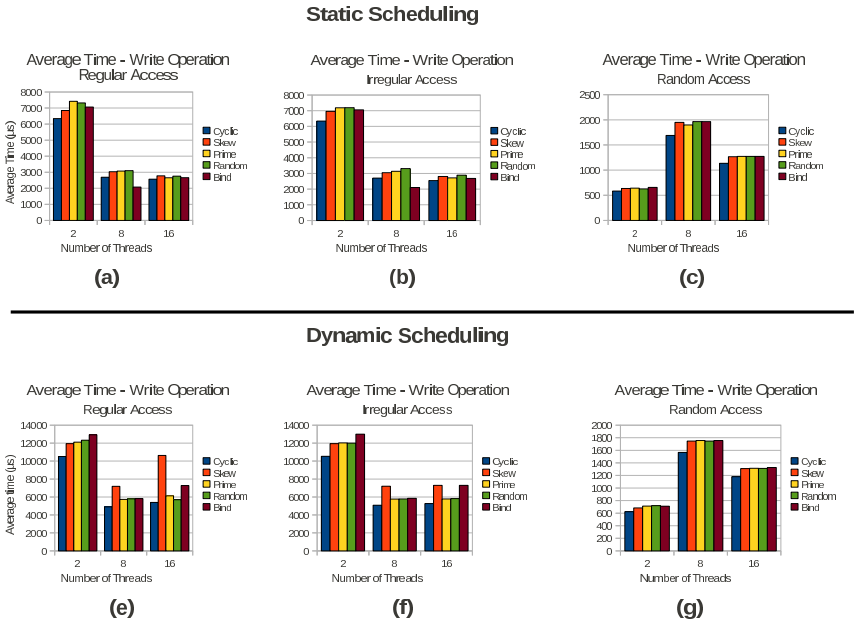


Figure 17: Average Time for Write Operation

results for a small number of threads (two threads). Such memory policy have used two cores in different nodes to bind threads and two memory banks to spread data. Due to this, the memory policy minimizes memory contention, because it divides accesses into two memory banks. For a high number of threads (8 and 16 threads), the best memory policy have been *bind_ block*. Besides memory contention minimization, this memory policy also minimizes the latency costs for write operations.

The average time that has been obtained for dynamic scheduling were different from ones obtained with static scheduling. In figure 17 (d), (f) and (g), we can observe that *skew* memory policy has presented the worst time for this operation. This memory policy spreads data among the machine nodes by doing a non linear round-robin distribution. Due to this, neighbor memory pages may be placed in distant memory banks, losing data locality. *Cyclic* memory policy has presented lowest times for write operation. In dynamic scheduling, threads can steal work and due to this, spread data among the machine nodes is the best solution. Additionally, *cyclic* preserve data locality, since it places neighbor pages on adjacent memory banks.

In Figure 18, we present the average time for read/write operation with different memory policies, vector accesses and workload distribution. This figure shows results for two threads, eight threads and sixteen threads. On general, for read/write operation best performance have been obtained with *bind_ block* for static scheduling and *cyclic* for dynamic scheduling.

In static scheduling, we have observed that *bind_ block* have presented better results for this operation. Since *bind_ block* considers regular access on data in its blocks placement, it is a suitable memory policy for this operation and

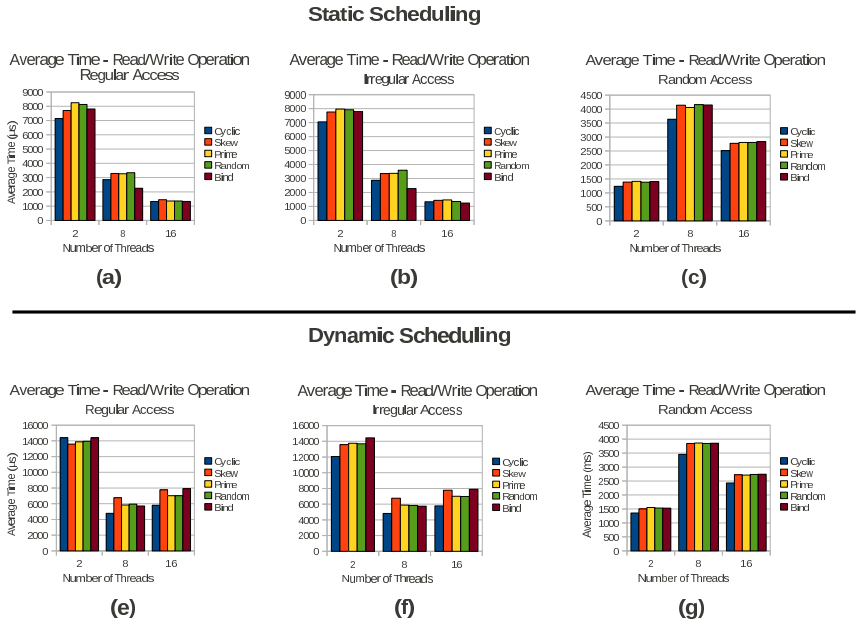


Figure 18: Average Time for Read/Write Operation

scheduling. However, for two threads the best memory policy for this operation have been *cyclic*. In this case, the memory policy divides accesses into two memory banks, minimizing memory contention.

Considering dynamic scheduling, for two threads and regular access, *skew* memory policy has presented the best results. *Skew* policy has preserved data locality better than *cyclic*, generating better performance for the operation. For a high number of threads, the best memory policy has been *cyclic*. Since threads can steal work from other threads, *cyclic* policy guarantees that data will be spread among all nodes of the machine, minimizing memory contention. In Figures 16, 17 and 18, it is important to notice that for a high number of threads (8 and 16 threads) there is not a high difference between the obtained latencies. On Opteron ccNUMA platform, the main characteristics are low NUMA factor and bandwidth problem. This is mean that remote access are not expensive but bandwidth may be a big problem. Due of this, we have done experiments with memory policies that distribute data among the machine nodes. Consequently, average times to perform the operation have been similar for different operations. However, it is important to remember that in this characterization we have used a microbenchmark with only 2 millions of access on vectors. Thus, even a small difference can be a high difference on real applications.

We can conclude that on the Opteron ccNUMA platform, for a high number of threads best memory policies are *cyclic* and *bind_ block*. On one hand, read operations that are less expensive than write and read/write operations have presented better results with *cyclic*. On the other hand, write and read/write operations have obtained better performance with *bind_ block*. For small number

of threads, the best memory policy depends on the access type and workload distribution.

4.2 Memory Affinity

In this section we present the improvement gains we have obtained on Stream Benchmark and NPB's benchmarks by using the memory access characterization presented on above section. We have used the memory access characterization to assure memory affinity on Stream and NPB's applications and kernels. Memory affinity is assured when a compromise between threads and data is achieved by reducing either the number of remote accesses (latency optimization) or the memory contention (bandwidth optimization).

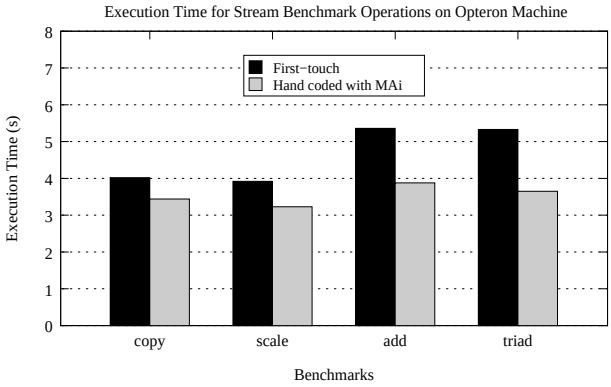


Figure 19: Performance of Stream Operations on Opteron

In order to guarantee memory affinity for such benchmarks, we have changed source codes using an interface named MAi [8]. This interface allows us to better place data and threads over the ccNUMA platform by using some memory policies. To apply MAi memory policies in source codes, we have just to allocate data using MAi allocators and then apply a memory policy to such data. Considering Stream and NPB's characteristics, we have selected four different memory policies from MAi (*cyclic*, *prime_ mapp*, *skew_ mapp* and *bind_ block*). The first three memory policies are ideal for irregular applications, since they spread data among nodes. The latter memory policy is suitable for regular applications where threads always access the same data set. More detail about source code modifications are described during the results presentation.

We have performed some experiments with the changed version of each benchmark and compared to the results obtained with their original version. Such results have been obtained through the average of several executions varying the number of threads from 2 to the maximum number of cpus/cores of the platform. These results have presented a low standard deviation, since all experiments have been done with exclusive access to the ccNUMA machine. Our results are organized by application (Stream, FFT, MG, CG, LU, SP and BT).

In Figure 19, we present the average time obtained for each Stream operation on the ccNUMA platform using the original version of the code and modified version with MAi. In the original version of the code, the memory policy used to

place data have been *first_ touch*, the default memory policy of Linux operating system. As we can observe, MAi has outperformed *first_ touch* results for all operations. Considering the characteristics of the application (regular accesses with static workload distribution), we have used *bind_ block*, *skew_ mapp* and *cyclic* memory policies on MAi version of Stream. *Bind_ block* policy have been used for a high number of threads (greater than 8) for all memory operations (read, write and read/write). This memory policy has been chosen because it places threads and data closer, minimizing latency costs and memory contention. *Skew_ mapp* and *cyclic* policies have been used for a small number of threads. In this case, spread data among the machine nodes have presented better results because it increases memory bandwidth.

Figure 20 shows the speedup for NAS Benchmarks on Opteron platform for the original version and the version with memory affinity optimization using MAi interface. As one can observed, MAi version has outperformed most of the original implementations of NAS benchmarks.

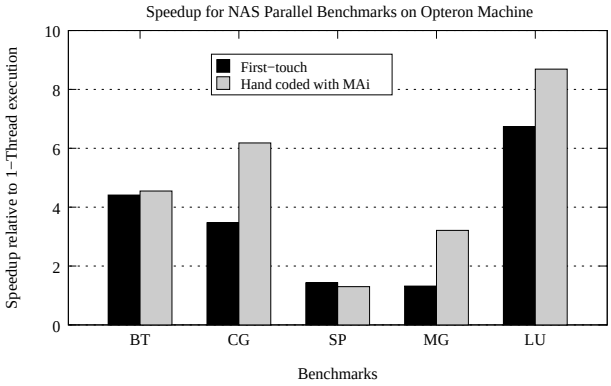


Figure 20: Performance of NAS Parallel Benchmarks on Opteron

Figure 20 shows the speedups for FFT on Opteron platform for the original version and the MAi version. As it can be observed, Minas has outperformed all other memory affinity solutions. As we can observe in Figure 20, MAi have obtained better results than the original version of the application. Considering the characteristics of Opteron platform, we have chosen *prime_ mapp* as memory policy to be applied in the most important arrays (memory access and consumption) of FFT. Such policy aims at providing a non-uniform distribution of memory pages among the ccNUMA nodes. Due to this fact, it spreads memory pages in a better way, since it avoids any patterns during data distribution. The used ccNUMA has a small NUMA factor and bandwidth optimizations are important. Additionally, FFT is an irregular application in which three dimensional arrays are accessed in a non linear way. On general, the original version have not presented good results. The original version uses the operating system memory affinity management, that for Linux is *first-touch*. This memory policy optimizes latency and considering this platform and application *first-touch* is not a efficient choice. We can also observe that the results with MAi and *first-touch* have been similar for two, four and sixteen threads. When a small number of threads is used memory contention is not high, thus different mem-

ory policies may have the similar performance on platforms with small NUMA factor (remote access costs are not high).

In Figure 20, we present the speedups obtained with CG on the ccNUMA platform for the original version and the MAi version. As we can observe, MAi has performed well on the platform. On general, MAi has been 17% better than the original version with *first-touch*. *First-touch* policy is not suited to irregular applications since it optimizes latency instead of reducing memory contention. This optimization results in several memory accesses on the same memory banks. In this case, considering the platforms network interconnections, we have selected *cyclic* and *bind_ block* memory policies for the platform. Since Opteron has a low NUMA factor and a simple interconnection network, we have applied *cyclic* for arrays that are accessed irregularly, whereas *bind_ block* has been applied for those accessed regularly. Thus, we can both optimize bandwidth and reduce memory contention.

Considering the results for MG application, we can observe that without memory affinity optimizations the OpenMP solution does not present performance gains when the number of threads is increased (Figure 20). In MAi version of MG application, *cyclic* policy has been used to optimize bandwidth. As MG has irregular read memory access as its main characteristic and considering our memory access characterization, *cyclic* is the most suited policy.

Generally, the LU version using MAi has been more efficient than the original version with *first-touch* (Figure 20). Since threads will use memory pages in its computation *first-touch* can not be used to place memory pages because LU computations are not regular. Thus, several memory accesses on remote memory banks are performed to access matrices elements. In this case, considering the platform network interconnections, we have selected *cyclic* and *bind_ block* memory policies for LU. *Cyclic* memory policy has been chosen to improve the bandwidth usage of the machine when the machine was not fully used (2 to 8 threads). *Bind_ block* memory policy has been used with 16 threads to optimize both latency and bandwidth.

In the case of BT benchmark, MAi optimizations on the source code have led to some small performance gains when compared to the original version. The benchmark BT has several parallel sections and all of them uses static scheduling strategy. Due to this, workload is split into several chunks of the same size for all threads. However, the parallel sections are parallelized in different directions. Because of this threads computes different data sets on the different sections of the benchmark. In the case of the MAi, it would be more effective to have memory policies for each different parallel section. However, in this version of the benchmark, we have used the same memory policy per variable for all parallel sections of the applications.

5 Related Work

The advent of large scale hierarchical shared memory multiprocessors with NUMA characteristics has demanded better understanding of memory access patterns and the influences of data placement on such patterns. Because of this research groups have investigated the performance and behavior of several workloads over multi-core machines and ccNUMA platforms [16, 17, 15, 18].

Scientific workload characterization over ccNUMA platforms with multi-core processors has already been studied in [16]. This work is similar to ours and it has investigated the impact of multi-cores and processor affinity on hybrid scientific workloads (based on Message Passing Interface (MPI) and OpenMP). However, the main focus of [16] was to understand and analyze the impact of multi-cores on workloads. They do not really address the impact of NUMA characteristics on numerical scientific workloads. Since ccNUMA platforms have become more common on HPC (machines based on AMD Opteron processors [5] and Intel Nehalem processors [19]), it is important to investigate the impact of the non-uniformity on memory accesses on such platforms.

In [17], researchers have studied the impact of clusters with multi-core processors, multi-processor nodes and multi-core, multi-processor nodes on a subset of NAS parallel benchmarks implemented using MPI. They have analyzed inter-communication efficiency, cache effects and initial process distribution. Based on such analysis, they have proposed some guidelines for optimizing MPI applications on such type of clusters. Thus, they are interested on effects of multi-cores may cause on MPI applications. In our work, our focus is memory access patterns on ccNUMAs with multi-cores, and not the effects of multi-core on some particular parallel programming interface.

The 2312 Opteron cores system based on Sun Fire servers was considered as case study, in the work [15]. They have characterized the performance behavior of the cluster and its nodes. Their main object was to investigate performance bottlenecks and provide some solutions to improve the system utilization. They have used well know benchmarks and some synthetic micro-benchmarks. The results showed that performance loss are caused by the interconnection between nodes, cache hierarchy and memory affinity management. This work may be similar to ours but they do not investigate the effects of memory affinity on memory operations, they do not consider data and thread affinity on their experiments, and they do not consider a large set of numerical scientific workloads.

In [18], authors have evaluated the performance of multi-core platforms (quad-core AMD Barcelona and dual-core Intel Woodcrest) with scientific applications. They have focus their analyzes on performance of the memory and communication sub-systems. The obtained results have led authors to identify some configurations to get optimal performance on such systems. In this work, authors have considered just multi-core aware on the study.

In our work, we have investigated memory accesses behavior of microbenchmarks and benchmarks over a ccNUMA platform with multi-core processors. Additionally, we have evaluated a set of memory policies that were used to place data among the machine memory banks. Our results have shown that an appropriate selection of data placement, considering the memory accesses, can generate up to 55% of improvement gains.

6 Conclusions and Future Work

In this work, we have focused our work on characterization of numerical scientific workloads on a ccNUMA platform with multi-core processors. We have also presented some performance evaluation of the ccNUMA platform and NAS Parallel Benchmarks. In order to do the characterization and the performance

evaluation we have performed some experiments using micro-benchmarks and Benchmarks.

Our experiments has shown that on parallel numerical scientific workloads, different memory accesses (operations, data access and data distribution) have different behaviors. Ours results have also shown that such different access need different strategies to place data and threads in order to obtain an optimal performance. The experiments with NPB's have confirmed that on ccNUMAs with multi-cores processors, it is necessary to assure memory affinity by placing optimizing latency and bandwidth.

Our future work includes providing an mechanism to chose the best memory policy for each type of memory access. Additionally, we want to extend this study on larger ccNUMA systems.

Acknowledgment

This research was supported by the French ANR under grant NUMASIS ANR-05-CIGC and CAPES (Brazil) under grant 4874-06-4.

References

[1] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy, "The dash prototype: Logic overhead and performance," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 1, pp. 41–61, 1993.

[2] T. Mu, J. Tao, M. Schulz, and S. A. McKee, "Interactive Locality Optimization on NUMA Architectures," in *SoftVis '03: Proceedings of the 2003 ACM Symposium on Software Visualization*. New York, NY, USA: ACM, 2003, pp. 133–ff.

[3] J. Marathe and F. Mueller, "Hardware Profile-Guided Automatic Page Placement for ccNUMA Systems," in *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2006, pp. 90–99. [Online]. Available: http://portal.acm.org/citation.cfm?id=1122987

[4] A. Joseph, J. Pete, and R. Alistair, "Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport," 2006, pp. 338–352.

[5] AMD, "Advanced Micro Devices - AMD Opteron," 2009. [Online]. Available: http://www.amd.com

[6] J. D. Mccalpin, "STREAM: Sustainable memory bandwidth in high performance computers," 1995. [Online]. Available: http://www.cs.virginia.edu/stream/

[7] J. Y. Haoqiang Jin, Michael Frumkin, "The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance," NAS System Division - NASA Ames Research Center, Tech. Rep. 99-011/1999, 1999. [Online]. Available: https://www.nas.nasa.gov/Research/Reports/Techreports/1999/PDF/nas-99-011.pdf

[8] C. P. Ribeiro, M. Castro, L. G. Fernandes, A. Carissimi, and J.-F. Méhaut, "Memory Affinity for Hierarchical Shared Memory Multiprocessors," in *21st International Symposium on Computer Architecture and High Performance Computing - SBAC-PAD*. São Paulo, Brazil: IEEE, 2009.

[9] Z. Smith, "Bandwidth: a memory bandwidth benchmark for x86 x86_64 ARM based Linux and ARM Windows MobileCE," 2010. [Online]. Available: http://home.comcast.net/~fbui/bandwidth.html

[10] The BenchIT Project, "Performance Measurement for Scientific Applications," 2010. [Online]. Available: http://www.benchit.org/

[11] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory performance and cache coherency effects on an intel nehalem multiprocessor system," in *PACT '09: Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 261–270.

[12] D. H. Bailey, E. Barzcz, L. Dagum, and H. D. Simon, "Nas parallel benchmark results," *IEEE Concurrency*, vol. 1, no. 1, pp. 43–51, 1993.

[13] M. F. H. Jin and J. Yan., "The OpenMP Implementation of NAS Parallel Benchmarks and its Performance," Tech. Rep. NAS-99-011, October 1999. [Online]. Available: www.nas.nasa.gov/News/Techreports/1999/PDF/nas-99-011.pdf

[14] A. Kleen, "A NUMA API for Linux," Tech. Rep. Novell-4621437, April 2005. [Online]. Available: http://whitepapers.zdnet.co.uk/0,1000000651,260150330p,00.htm

[15] A. Kayi, E. Kornkven, T. El-Ghazawi, S. Al-Bahra, and G. B. Newby, "Performance evaluation of clusters with ccnuma nodes - a case study," *High Performance Computing and Communications, 10th IEEE International Conference on*, vol. 0, pp. 320–327, 2008.

[16] S. R. Alam, R. F. Barrett, J. A. Kuehn, P. C. Roth, and J. S. Vetter, "Characterization of scientific workloads on systems with multi-core processors," in *IISWC*, 2006, pp. 225–236.

[17] H. Pourreza and P. Graham, "On the programming impact ofmulti-core,multi-processor nodes in mpi clusters," *High Performance Computing Systems and Applications, Annual International Symposium on*, vol. 0, p. 1, 2007.

[18] A. M. DeFlumere and S. R. Alam, "Exploring multi-core limitations through comparison of contemporary systems," in *TAPIA '09: The Fifth Richard Tapia Celebration of Diversity in Computing Conference*. New York, NY, USA: ACM, 2009, pp. 75–80.

[19] Intel, "Laptop, Notebook, Desktop, Server and Embedded Processor Technology - Intel," 2009. [Online]. Available: http://www.intel.com

Centre de recherche INRIA Grenoble – Rhône-Alpes
655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Lille – Nord Europe : Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399