Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

NUMA-CIC: Issues and Challenges for Scaling Scientific Applications on a Large Scale ccNuma Prototype

Dimitris Siakavaras[1,*], Konstantinos Nikas[1], Nikos Anastopoulos[1], and Georgios Goumas[1]

[1]Greek Research & Technology Network (GRNET), Greece

Abstract

This whitepaper studies the various aspects and challenges of performance scaling on large scale shared memory systems. Our experiments are performed on a large ccNUMA machine that consists of 72 IBM 3755 nodes connected with *NumaConnect* and provides shared memory over a total of 1728 cores, a number that is far beyond conventional server platforms. As benchmarks, three data-intensive and memory-bound applications with different communication patterns are selected, namely Jacobi, CSR SpM×V and Floyd-Warshall. Our results illustrate the need for numa-aware design and implementation of shared-memory parallel algorithms in order to achieve scaling to high core counts. At the same time, we observed that, depending on its communication pattern, an application could benefit more from explicit communication using message passing.

1. Introduction

The *NumaConnect* technology [1] enables the realization of large scale SMPs from commodity servers. This is achieved through the use of a novel node board connected to the HyperTransport processor bus of standard AMD based servers. The resulting system is a cache coherent Non-Uniform Memory Access (ccNUMA) machine.

Shared memory is attractive to developers, as any processor can access any data in the system through direct load and store operations, thus making the programming of the system easier and the code less error-prone. On the other hand, performance scaling on large scale systems is generally not straightforward. In NUMA systems, however, it can be even more difficult and complicated as the allocation of shared data can have a significant impact on the performance.

In this whitepaper we attempt to gain a deeper understanding of the characteristics of a large scale ccNUMA machine and the challenges the developers are facing. The rest of the paper is organized as follows: Section 2. offers a short description of the platform and the benchmarks used in our experiments; Section 3. discusses our findings and Section 4. concludes and highlights possible directions for future work.

2. Experimental Methodology

2.1. Hardware Platform

Our experiments were performed on a PRACE prototype, the large shared memory *NumaConnect* cluster at the University of Oslo in Norway. The system consists of 72 IBM 3755 nodes [2] connected in a 3D torus with *NumaConnect*. Each node contains two AMD 12-core Opteron 6174 Magny-Cour processors, four memory channels and 64GiB of memory. The details of the system are summarized in Table 1.

	Node	System
Cores	24	1728
Mem. Controllers	4	288
RAM	64GiB	4.6TiB

Table 1: System characteristics (72 nodes)

*To whom correspondence should be addressed. Email: jimsiak@cslab.ece.ntua.gr

2.2. Benchmarks

For our experiments we have selected three kernels that are widely used in scientific and engineering applications. In general, all three applications are data intensive and memory-bound (at least in their original implementations) and exhibit different communication patterns. The selected benchmarks are in more detail:

1. **Jacobi**: The Jacobi kernel is a common stencil computation. In our case, we use the Jacobi computational method to solve the 3-dimensional heat equation. Our baseline shared-memory version is implemented in OpenMP with straightforward loop parallelization. We also use a message-passing implementation with MPI which employs a nearest-neighbor communication pattern.
2. **SpM×V**: Sparse Matrix multiplication with Vector (SpM×V) is an important and ubiquitous computational kernel in sparse numerical algebra [3]. Matrices are stored in the CSR format [4]. We use a shared-memory version implemented in Pthreads and a message-passing version in MPI. The algorithm exhibits an irregular communication pattern, in which each process communicates with varying numbers of processes and message sizes dependent on the sparse matrix structure and the total number of participating processes in the computation. We used 5 different matrices collected from the University of Florida sparse matrix collection [5] with various memory footprints.
3. **Floyd-Warshall**: The Floyd-Warshall algorithm is a graph analysis algorithm used for finding shortest paths in weighted graphs. Our shared-memory version is implemented with OpenMP and the message-passing version is implemented in MPI. Communication occurs in a collective way, where one process in each time step broadcasts a matrix row to all other processes.

The memory footprints of the inputs that we used in our experiments are presented in Table 2.

Jacobi		SpM×V		Floyd-Warshall	
Input	Size (GiB)	Name	Size (GiB)	Input	Size (GiB)
512^3	2	msdoor	0.118	4096	0.0625
1024^3	16	rajat31	0.262	16384	1
$1024^2 \times 2048$	32	ldoor	0.271	32768	4
$1024^2 \times 4096$	64	boneS10	0.626	65536	16
$1024^2 \times 8192$	128	HV15R	3.18		

Table 2: Memory footprint of benchmarks' inputs

In order to take advantage of the characteristics of the underlying platform, numa-aware versions of all the benchmarks were developed. In these versions, each thread allocates its data in the memory of the node where it executes, in an attempt to reduce the amount of off-node memory accesses.

2.3. Instrumentation

The Linux kernel employed by the experimental platform is NUMA aware, i.e. it provides the appropriate mechanisms to schedule multiple threads efficiently; specifically, we use the `numactl` utility to instrument the execution of our kernels. Moreover, we bind each thread to a specific processor, following what we call *packed placement policy*, i.e. we employ all the physical cores available in one physical package/node before assigning a thread to another physical package/node. So, the first 12 threads are executed on the same die, the first 24 threads are executed on the 24 cores of the first node of the system, the next 24 threads on another node and so on.

3. Results and discussion

The *NumaConnect* prototype provides an exceptional platform supporting shared memory in a number of cores that is far beyond conventional server platforms and HPC compute nodes. The efforts of this work are primarily focused on shedding light on the potential of this platform to deliver high performance to parallel application while taking also into consideration the programming effort. In our first set of experiments, we compare straightforward and numa-aware shared-memory implementations. In our second set of experiments our goal is to evaluate the prototype as a shared-memory execution platform for parallel applications in terms of scalability. Finally, in the third set of experiments we compare shared-memory and message-passing in order to understand the implications of the implicit and explicit communication mechanisms thar are triggered in each case and their impact on performance.

3.1. Non-numa vs numa-aware implementations

Fig. 1 presents the execution time of all the benchmarks, comparing the non-numa and numa-aware versions for different number of threads. Note that, due to the scale of the graph, the three curves of the numa-aware implementations overlay each other. As long as the threads remain inside the same node (up to 24 threads), the performance of the non-numa and numa-aware versions are similar. However, when more than one node is employed, the performance of the non-numa code degrades severely for all the benchmarks. This is attributed

to the high cost of the off-node memory accesses. On the other hand, for the numa-aware implementations, the amount of off-node memory accesses is reduced and the performance degradation is avoided. Thus, numa-awareness is a definite prerequisite for shared-memory parallel applications.

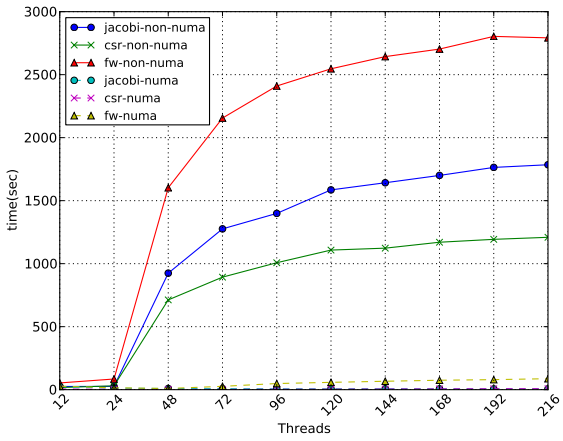


Fig. 1: Execution time of non-numa vs numa-aware implementations

3.2. Scalability

In this section we evaluate the scalability of the numa-aware versions of the selected benchmarks, using different shared-memory programming models. More specifically we employ OpenMP for Jacobi and Floyd-Warshall and Pthreads for SpM×V.

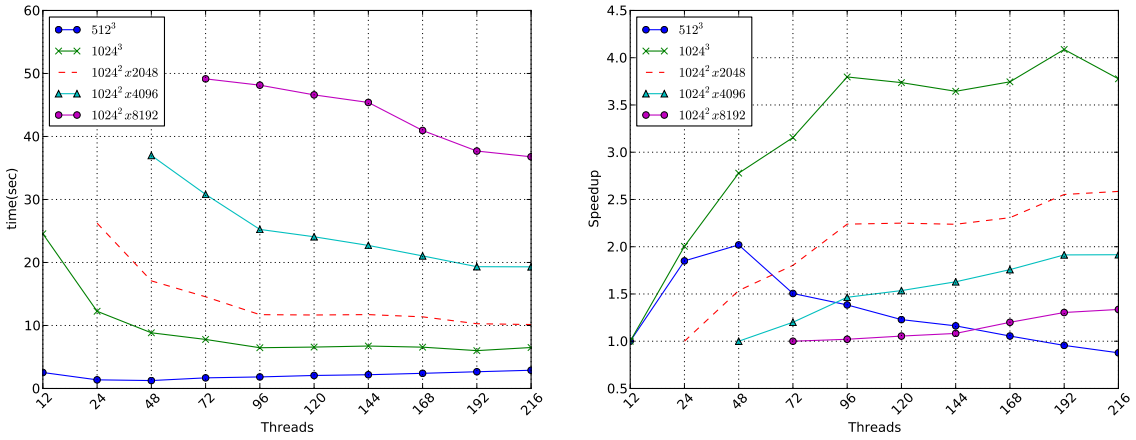


Fig. 2: Jacobi scalability

Fig. 2 presents the execution time, as well as the speedup of the Jacobi kernel for different number of threads and input sizes. For input sizes $1024 \times 1024 \times 4096$ and $1024 \times 1024 \times 8192$, 64GiB and 128GiB of memory are required. As each node has a total memory of 64GiB, a part of which is occupied by the OS, it is impossible to utilize a numa-aware data allocation, and avoid remote memory accesses, when running with less than 2 nodes for the 64GiB input and less than 3 nodes for 128GiB. Therefore, these executions were omitted from our experiments.

It is evident that the performance scales to a high core count, especially for large input sizes. Typically, there exists a point after which scalability breaks due to communication overheads. The same conclusion can be drawn for the other two benchmarks as well. Fig. 3 and Fig. 4 illustrate the execution time and speedup for SpM×V and Floyd-Warshall respectively, for different number of threads and input sizes. Similar to the Jacobi kernel, as the input size increases, the performance scales to higher core counts up to the point where communication overheads overcome the time consumed by each thread on useful computations.

3.3. Shared memory programming vs MPI

In this section we compare the developed shared-memory, numa-aware implementations against MPI implementations. NumaScale provides an optimized OpenMPI Byte Transfer Layer (BTL), and therefore we use in our

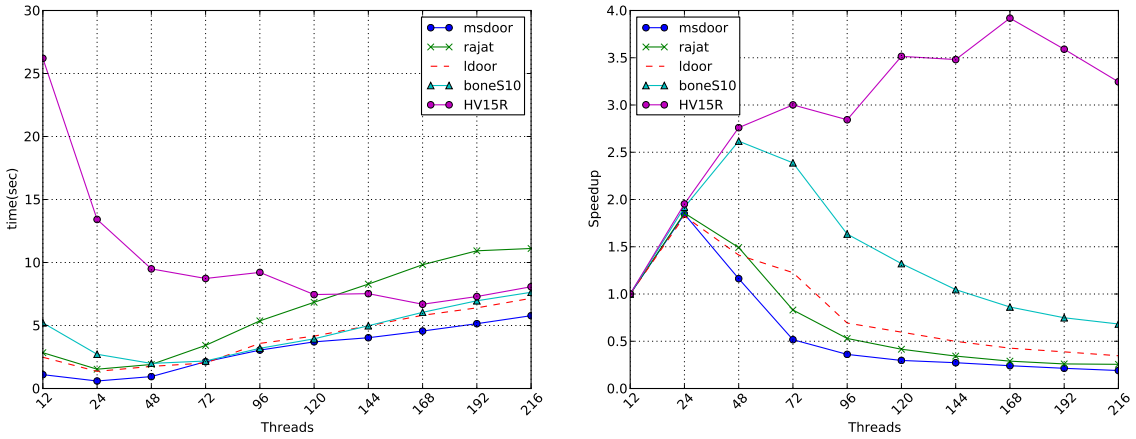


Fig. 3: SpM×V scalability

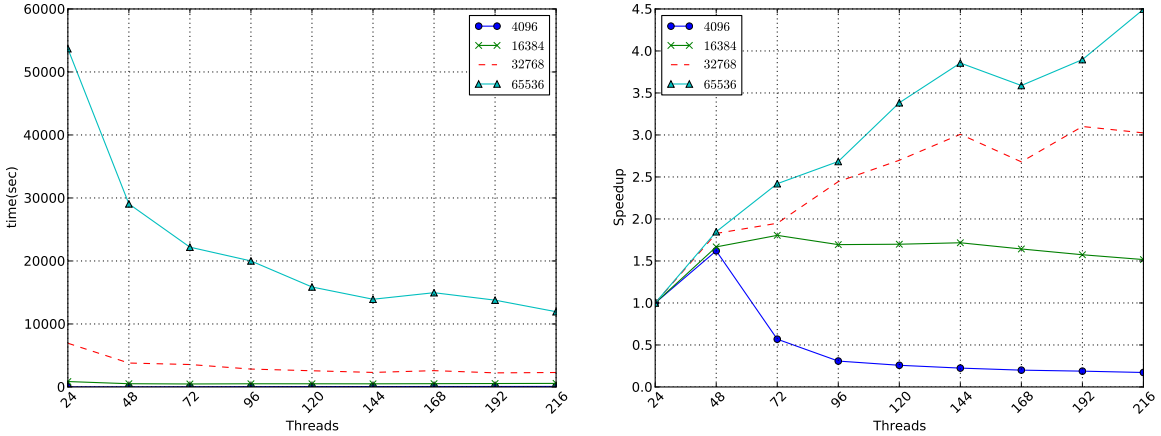


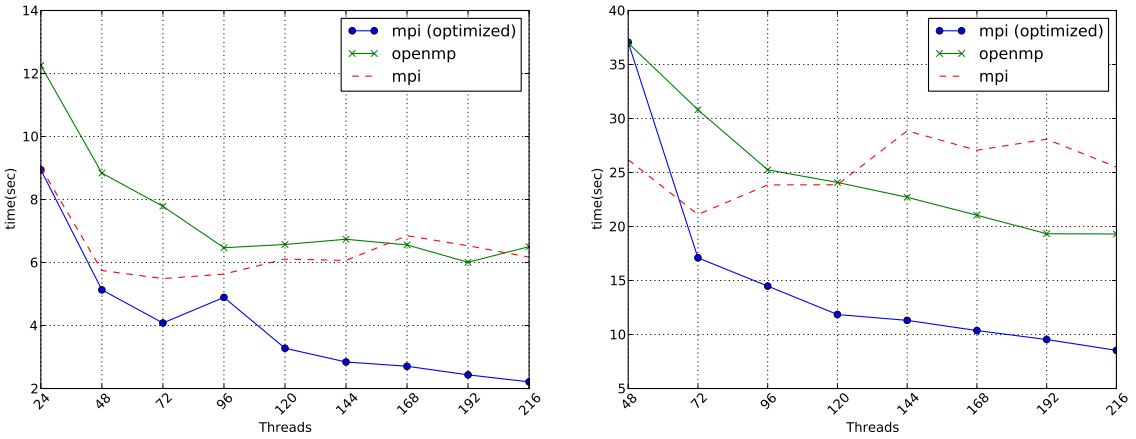
Fig. 4: Floyd-Warshall scalability

evaluation two MPI versions: one employing the provided NC-BTL optimised OpenMPI library and another employing the standard OpenMPI library.

Fig. 5 presents the execution time of the OpenMP Jacobi kernel together with the time of the two MPI implementations for different number of threads and input size. The communication optimization offered by the BTL mechanism is evident at the application level. Regarding the comparison between OpenMP and MPI (with BTL), we may observe that the explicit message-passing implementation in MPI outperforms OpenMP even inside a a single node (24 threads) where the data transfers serviced by hardware mechanisms have the minimum cost. The performance gap between the two implementations widens with an increasing number of cores. This is due to the nature of the algorithm, since communication data are large 2D surfaces which in the case of MPI are explicitely packed and sent in bulk with point-to-point operations. This bulk data transfer is able to diminish the message latency of MPI messages. On the other hand, in the OpenMP implementation, each element in the communication 2D surface is touched with read/write operations by two neighboring threads, a pattern that is too fine-grain and suffers from high traffic in the memory subsystem due, also, to the cache-coherence protocol.

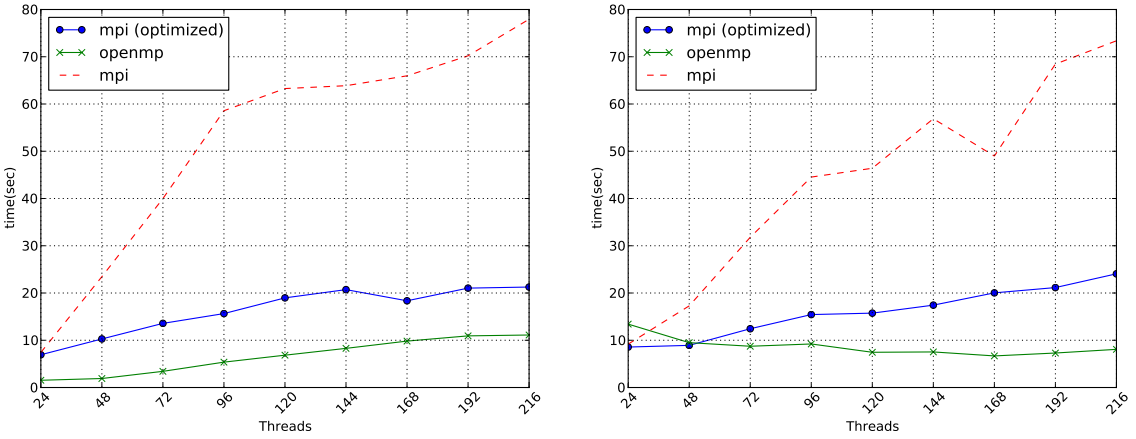
The landscape in SpM×V however, is completely different. As shown in Fig. 6, OpenMP outperforms both MPI implementations. In this case the explicit MPI implementation suffers from higher communication overheads, since a large number of small messages need to be exchanged by the MPI processes. The situation becomes worse (number of messages increases, message size decreases) as the number of cores (MPI processes) increases.

Finally, in the case of Floyd-Warshall the situation is more balanced (especially in the large data set). Recall that in this case the communication is due to the fact that a single matrix row (different per algorithmic iteration) needs to be broadcasted by the holding process. In the shared-memory version this is accomplished by read operations to the "critical" matrix row, which for the majority of the threads can be remote, imposing a signficicant remote-read overhead. On the other hand, shared caches can assist in reducing the overhead of this implicit broadcasting, as data may be found closer to their initial source, brought there by another thread sharing the same cache. In the MPI case, the communication overhead comes from the explicit broadcast of a



(a) $1024^2 \times 1024$ (b) $1024^2 \times 4096$

Fig. 5: Jacobi numa-aware vs mpi



(a) rajat (b) HV15R

Fig. 6: CSR SpM×V numa-aware vs mpi

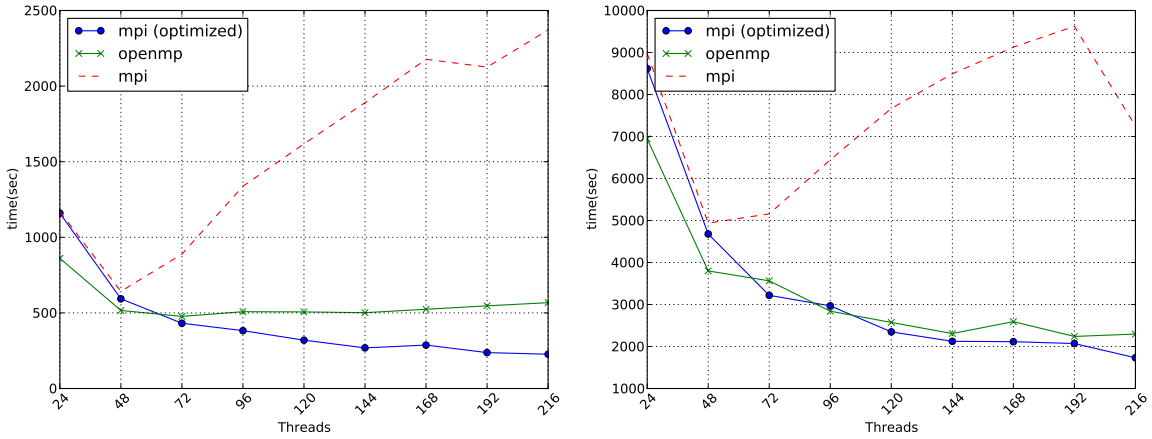
matrix row (a few Kbytes of data) to the participating processes. Overall, it seems that the two communication mechanisms in shared-memory and message-passing happen to have a similar overhead.

4. Conclusions

Scaling applications on large scale shared memory systems is not easy and straightforward. Scaling beyond a single node, requires the developer to acquire a deep understanding of the NUMA effects of the system. This knowledge is essential in order to optimise thread binding, data allocation and workload distribution as they can have a significant impact on the performance.

More specifically, thread binding and data allocation can help minimise the amount of the expensive off-node memory accesses. Similarly, workload distribution is important for scaling to high number of cores, because when the amount of work performed by each thread becomes negligible the execution is dominated by the communication overheads.

Finally, the selection of the programming model is not straightforward. Whether the application will benefit from explicit communication using a message passing programming model or not, depends on its communication pattern. Our results reveal that for some applications the employment of MPI offers the best performance (Jacobi), others benefit more from a shared-memory programming model (SpM×V), while for others (Floyd-Warshall) it makes little, if any, difference. On the other hand, the implementation on each programming model requires different effort from the programmer. In the case of the shared memory model, normal read and write instructions can be used for accessing every memory location, making it relatively easy and straightforward to produce parallel code. On the contrary, the message passing programming model requires extra effort by the programmer to orchestrate the messages that need to be exchanged between the parallel threads of the



(a) $16K$ (b) $32K$

Fig. 7: Floyd-Warshall numa-aware vs mpi.

application. When this effort is taken into account, we can conclude that OpenMP is probably the best option for Floyd-Warshall.

In general, for our expirements in this whitepaper we used inputs with a tiny memory footprint compared to the total of 4.6TiB provided by the system. That decision was made due to time limitations since benchmarks with larger footprints could take even weeks to execute. For future work we plan to extend our evaluation with higher core counts and inputs with larger memory footprints, as well as more benchmarks exhibiting a variety of communication patterns. Moreover, we aim to perform a more exhaustive analysis of the communication overheads induced by the hardware coherence protocol, in order to gain a deeper understanding of the parameters affecting the scalability of parallel applications on large scale ccNUMA systems. Finally, we intend to investigate the impact of threads' distribution on the scalability, caused by the distance between the nodes on which the threads are executing.

Acknowledgments

We would like to thank the Research Infrastructure Services Group of the Department for Research Computing of the University of Oslo for providing and supporting our access to the NUMA-CIC prototype, as well as our reviewers for their useful comments and observations. This work was financially supported by the PRACE-1IP project funded in part by the EU's 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-261557.

References

1. `http://www.numascale.com/numa_technology.html`.
2. `http://www-03.ibm.com/systems/x/hardware/rack/x3755m3/`.
3. G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Performance evaluation of the sparse matrix-vector multiplication on modern architectures," *The Journal of Supercomputing*, 2008.
4. R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Philadelphia: SIAM, 1994.
5. T. Davis, "University of Florida sparse matrix collection," *NA Digest*, vol. 97, no. 23, p. 7, 1997.