Multicore Architecture-aware Scientific Applications

by

Avinash Srinivasa

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:

Masha Sosonkina, Major Professor

Zhao Zhang

James P. Vary

Iowa State University

Ames, Iowa

2011

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to take this opportunity to thank all those people who have helped make this research and hence this thesis a reality. First and foremost, I would like to thank my advisor Dr. Masha Sosonkina for her valuable suggestions and her constant support and encouragement throughout the course of my graduate studies. Second, I would like to thank my committee member Dr. James Vary for funding this research and for his valuable feedback and suggestions on this work. I would also like to thank my co-major advisor Dr. Zhao Zhang for guiding me through my research and writing of this thesis. Last but not the least, I would like to thank Dr. Pieter Maris and Dr. Fang Liu for helping me work through various technical issues and problems which cropped up during the course of my research work.

This work was supported in part by Iowa State University and Ames Laboratory under the contract DE-AC02-07CH11358 with the U.S. Department of Energy, by the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC02-05CH11231, and in part by the National Science Foundation grant NSF/OCI – 0749156, 0941434, 0904782, 1047772.

ABSTRACT

Modern high performance systems are becoming increasingly complex and powerful due to advancements in processor and memory architecture. In order to keep up with this increasing complexity, applications have to be augmented with certain capabilities to fully exploit such systems. These may be at the application level, such as static or dynamic adaptations or at the system level, like having strategies in place to override some of the default operating system polices, the main objective being to improve computational performance of the application. The current work proposes two such capabilites with respect to multi-threaded scientific applications, in particular a large scale physics application computing ab-initio nuclear structure. The first involves using a middleware tool to invoke dynamic adaptations in the application, so as to be able to adjust to the changing computational resource availability at run-time. The second involves a strategy for effective placement of data in main memory, to optimize memory access latencies and bandwidth. These capabilties when included were found to have a significant impact on the application performance, resulting in average speedups of as much as two to four times.

CHAPTER 1. Introduction

With the continuous advancements happening in high performance computer architecture, newer and more powerful architectures are being introduced resulting in increasingly complex systems. In such a scenario, it becomes crucial for high performance applications to be able to fully leverage these architectures, in order to maximize their computational performance. This involves introducing certain capabilities in these applications to make sure that they utilize the computational resources available to the best possible extent. These may be in the form of adaptations invoked within the application or system based strategies in place for ensuring efficient execution. The current work proposes two such capabilities for multi-threaded scientific applications which have been found to significantly improve application performance. The main focus of this work is on improving application performance for multicore shared memory architectures. The work described in Chapter 3 has been published in the proceedings of the PDSEC workshop, a part of the 2011 IEEE Parallel and Distributed Processing Symposium (IPDPS) Srinivasa et al. (2011). The work described in Chapter 4 has been filed as a technical report Srinivasa and Sosonkina (2011) with the Computer Science Department at Iowa State University.

First, a strategy for incorporating application level adaptations in considered to adapt to the changing computational resource availability during the course of execution of an application. This is especially true in modern multi-user cluster environments where users can run many high-performance applications simultaneously which share resources such as Processing Elements (PEs), I/O, main memory, network. In such a scenario, it would be greatly advantageous to have applications augmented with adaptive capabilities, particularly during run-time. This involves targeting a computationally intensive part of the application and invoking appropriate adaptations so as to be able to adjust to the dynamically changing system conditions, to prevent drastic performance loss. Here, the parallel application MFDn (Many Fermion Dynamics for nuclear structure) used for ab-initio nuclear physics calculations is integrated with a middleware tool for invoking such adaptations. In particular, the multi-threaded Lanczos diagonalization procedure in MFDn is targeted to observe the effect on performance of dynamically changing the number of threads during the iterative process. Performance gains between two to seven times were observed in the presence of competing applications by incorporating these adaptation strategies.

Second, a strategy for efficiently distributing data processed by an application is studied in order to optimize memory access. This is important because as the core counts on modern multiprocessor systems increase, so does the memory contention with all the processes/threads trying to access the main memory simultaneously. This is typical of UMA (Uniform Memory Access) architectures with a single physical memory bank leading to poor scalability in multi-threaded applications. To palliate this problem, modern systems are moving increasingly towards Non-Uniform Memory Access (NUMA) architectures, in which the physical memory is split into several (typically two or four) banks. Each memory bank is associated with a set of cores enabling threads to operate from their own physical memory banks while retaining the concept of a shared virtual address space. However, accessing shared data structures from the remote memory banks may become increasingly slow. This work proposes a way to determine and pin certain parts of the shared data to specific memory banks, thus minimizing remote accesses. To achieve this, the existing application code has be supplied with the proposed interface to set-up and distribute the shared data appropriately among memory banks. Experiments with NAS benchmark as well as with the realistic large-scale application MFDn have been performed. Speedups of up to 3.5 times were observed with the proposed approach compared with the default memory placement policy.

CHAPTER 2. Review of literature

Computational peformance is a very important aspect for applications in the High Performance Computing (HPC) domain. This is because of the large scale nature of these applications which places an increased demand on the CPU and memory resources and also the continuously evolving architectural features which are designed to meet these demands. In such a scenario, understanding the impact of the underlying architecture on application performance and formulating strategies for improving it constitute an active research topic in this domain. As a consequence, there have been a lot of studies which have been conducted with a view of analyzing performance of high performance applications on modern architectures and methodologies which have proposed for enhancing this performance, both at the application level as well as at the system level.

With computational resource availability often changing during application run-time on modern multi-user cluster environments, it becomes necessary for applications to adapt to these changing system conditions, to avoid loss of performance. A number of methods have been proposed for enabling run-time application adaptations, in a manner which enables the system related monitoring and decision making process to be delineated from the application execution. Hollingsworth and Keleher (1998) propose a resource management system to dynamically adapt ongoing computations to changing system conditions, the objective being to efficiently execute parallel applications in large-scale, dynamic environments. Andersen et al. (2000) present an easily accessible operating system module to enable internet applications to be notified of, and be able to adapt to, dynamically changing network conditions. A helper middleware tool has been presented by Sosonkina (2006) to make an application aware of system run-time system changes and to adapt it dynamically to the new conditions. Specifically, a packet probing module is implemented by the tool to detect contention on the nodes of a cluster. Ustemirov et al. (2006) use this middleware tool to switch the execution of electronic structure calculations between I/O and memory access based on I/O resource contention at run-time. Chang and Karamcheti (2000) propose an application independent framework for (1) exposing application adaptation choices in the form of alternate configurations and (2) emulating application execution in a virtual environment with changing resource availability, so as to gather information about the resulting behavior of the application. The resulting framework is used to adapt an image visualization application to changes in PE load and network bandwidth by controlling application related behavior such as the compression algorithm used or the image transmission sequence. To our knowledge though, the current work is the first one proposed for dynamically detecting and avoiding PE core oversubscription in a fine-grained manner by monitoring and varying the number of application threads during run-time.

In scientific high performance computing, efficient data placement and memory affinity becomes a crucial aspect due to the data intensive nature of applications. A lot of research has been done for managing memory affinity on multicore NUMA platforms, both from the kernel and user space, to optimize memory access for maximum performance. An API for implementing some basic memory affinity policies, overriding the default policy of the operating system has been described by Kleen (2005) for Linux NUMA platforms. Antony et al. (2006) provide a framework for performing thread and memory placement on Solaris and Linux. The framework uses a Placement Distribution Model (PDM) which describes performance as a function of bandwidth and latency and is used to analyse performance results. In Goglin and Furmento (2009), a *next-touch* memory affinity policy has been implemented in the Linux kernel. This policy causes data migration when a thread touches it for the next time, which in general gives a better picture of the frequency of usage of a particular piece of data by a thread, and allows for more local accesses. Löf and Holmgren (2005) use this *next-touch* policy for improving the performance of an industrial PDE solver on a Linux NUMA system. Terboven et al. (2008) explore data and thread affinity for OpenMP programs. A portable user-level interface named MAi (Memory Affinity Interface) has been presented by Ribeiro et al. (2009) to provide a set of memory affinity polices for fine grained data control in scientific applications on Linux NUMA platforms. However, all these solutions provide a set of very generic policies for dealing with memory affinity and data placement on NUMA machines, which leaves a lot of work for the user in determining how best to use these polices for a particular application. The current work is different in the sense that it proposes an easy to use interface designed for a certain class of computations which a user can directly integrate into their application without worrying about the internal details.

CHAPTER 3. Dynamic adaptations in ab-initio nuclear physics calculations

3.1 Background and significance

The direct solution of the quantum many-body problem transcends several areas of physics and chemistry. Nuclear physics faces the multiple hurdles of a very strong interaction, three-nucleon interactions, and complicated collective motion dynamics. The aim is to solve for the structure of light nuclei addressing all three hurdles simultaneously by direct diagonalization of the nuclear many-body Hamiltonian matrix in a harmonic oscillator basis.

A tool to study nuclear structure is the software package MFDn (Many Fermion Dynamics for nuclear structure) developed by Vary et al. Vary (1992); Vary and Zheng (1994); Sternberg et al. (2008); Sosonkina et al. (2008); Maris et al. (2010) at Iowa State University. In MFDn, the nuclear Hamiltonian is evaluated in a large harmonic oscillator basis and diagonalized by iterative techniques to obtain the low-lying eigenvalues and eigenvectors. The eigenvectors are then used to evaluate a suite of experimental quantities to test accuracy and convergence issues.

MFDn has been shown to have good scaling properties using the Message Passing Interface (MPI) Forum (1994) on existing supercomputing architectures due to the recent algorithmic improvements that significantly improved its overall performance. In Maris et al. (2010), the use of a hybrid MPI/OpenMP approach Rabenseifner et al. (2009) has been presented to take advantage of the current multi-core supercomputing platforms. Under this approach, MPI and OpenMP Dagnum and Menon (1998) are used to communicate among inter-node and intra-node cores, respectively. The number of OpenMP threads that are to be spawned per process is fixed statically at the start of the run and is the same for every MPI process in the execution. This makes sense when running on some of the larger supercomputers or leadership class facilities since here, applications typically get the full use of the node(s) on which they are running.

However, in the case of most interactive cluster environments or cloud computing testbeds, users can run multiple high performance applications simultaneously. As a result, computational resource availability can often change during the run-time of the application. To cope with this, it might be beneficial to have an adaptive algorithm to change the number of threads dynamically based on system information gathered at run-time. However, changing the source code of an application such as MFDn to insert these adaptations is not feasible since it will increase the complexity of the scientific code, which may adversely affect its accuracy and usability. In such a scenario, there is a need for some generic middleware which can monitor the system resources during the execution of the application and invoke appropriate application adaptations. In this work, the middleware tool NICAN Sosonkina (2006) is used to monitor the number of threads/processes in the system during the execution of MFDn. Based on this run-time information gathered, the number of threads spawned is changed at regular intervals during the Lanczos diagonalization procedure. (See, e.g., Maris et al. (2010) for a description of the Lanczos algorithm.) This particular section of the code is chosen for invoking adaptations due to its iterative and computationally intensive nature.

3.2 Overview of ab-initio nuclear sructure calculations in the MFDn package

The MFDn software is a parallel code for *ab initio* nuclear structure calculations written in Fortran90 and MPI, being actively developed for almost two decades. In the early development of the code Vary (1992) and Vary and Zheng (1994), the main focus has been efficient use of memory; significant improvements in its performance have been made over the last 3 years Sternberg et al. (2008); Sosonkina et al. (2008); Maris et al. (2010); Vary et al. (2009); Laghave et al. (2009) under the US Department of Energy Scientific Discovery through Advanced Computing (SciDAC) Program.

The MFDn code computes a few lowest converged solutions, that is, the eigenvalues (energy

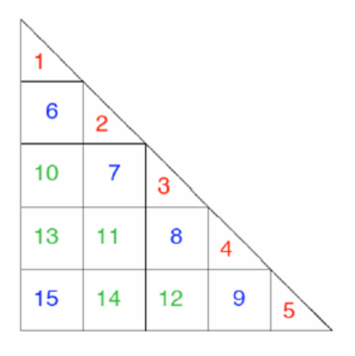


Figure 3.1 Two dimensional distribution of the lower triangle of the Hamiltonian matrix (Diagonal processors are numbered 1 – 5 and marked in red).

levels) and eigenvectors (wave functions), for the many-nucleon Schrödinger equation:

$$H\,|\phi\rangle = E\,|\phi\rangle\ . \tag{3.1}$$

One key feature of this calculation is the size of the very large sparse Hamiltonian matrix H it can produce. The dimension of the matrix characterizes the size of the many-body basis used to represent a nuclear wave function. In general, the larger the basis set and the total number of the oscillator quanta $N_{\max}$ above the lowest nuclei configuration, the higher the accuracy of the energy estimation Maris et al. (2009). MFDn constructs the many-body basis states, the Hamiltonian matrix, and solves for the lowest eigenvalues using the Lanczos algorithm. At the end of a run, it outputs the nuclear wave functions and evaluates selected physical observables, which can be compared to experimental data. The matrix is distributed in a 2-dimensional fashion over the processors (see Fig. 3.1), and only the lower triangle is stored and used, because the matrix is symmetric (and real-valued). The Lanczos vectors, needed for re-orthogonalization after every matrix-vector multiplication, are distributed over all the processors. Because of the 2-dimensional distribution of the matrix, MFDn runs on $n(n+1)/2$ processors, where n is the number of diagonal processors.

Since the Lanczos procedure is of particular interest in this work, an overview of the iterative process is provided (see Fig. 3.3). Each iteration consists of a matrix-vector multiplication, followed by an orthogonalization against all the previous Lanczos vectors (which are also all

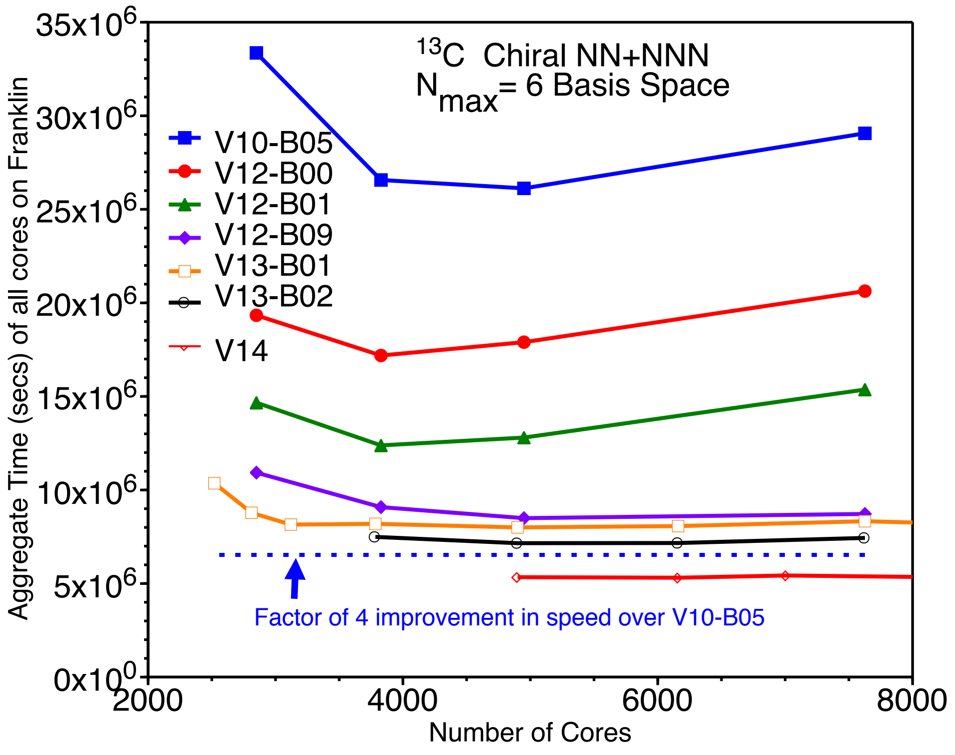


Figure 3.2 Performance improvement shown by the MFDn code over different versions.

stored in memory, distributed over all processors). After each matrix-vector multiplication, the resulting vector is accumulated on the n diagonal processors. Next, this vector is distributed to all the processors to do the orthogonalization, and finally the new input vector is distributed to all the processors. After a fixed number of Lanczos iterations (which is an input variable), the lowest eigenvalues and the corresponding wave functions are written to disk from the n diagonal processors.

Fig. 3.2 depicts the performance improvement obtained with the MFDn code over its different versions, starting from the very first version to the most recent. These numbers are from experiments conducted on the Franklin supercomputer[1].

3.2.1 MFDn using Hybrid MPI/OpenMP

Since modern processors are equipped with multiple cores, applications augmented with multi-threading capabilities can become considerably more efficient by making use of the mul-

[1]Franklin is a Cray XT4 at the National Energy Research Scientific Computing Center (NERSC) with 9,572 compute nodes. Each node is 2.3 GHz quad-core AMD Opteron processor (Budapest), with 2 GB per core.

tiple cores and overlapping memory access and communications with computations. To take advantage of the multiple cores, a hybrid MPI/OpenMP approach for MFDn has been presented in Maris et al. (2010). It employs multi-threading using OpenMP directives in the most computationally intensive parts of the code, which are the construction of the Hamiltonian matrix, the Lanczos iterations, and the evaluation of observables.

For the Lanczos iterations, the sparse matrix-vector multiplication is parallelized using an OpenMP *DO* directive to loop over the columns. Due to matrix symmetry, each matrix block is used twice in the multiplication. As a result, each thread has its own private output vectors for the result of the transpose matrix-vector multiplication, which are added inside an OpenMP *CRITICAL* region. The orthogonalization of the output vector against all the previous Lanczos vectors is also parallelized with an OpenMP *DO* directive. An experimental study of the speed-up with the increase in the number of threads has been carried out in Maris et al. (2010), during which the maximum speed-up of 2.5 was obtained with four threads on a single node of the Franklin supercomputer for ^{12}C nucleus with $N_{\max} = 4$. It has been observed that the scaling was hindered by the sequential computation fraction performed in the critical section and the MPI communication overhead becoming an increasing fraction of the overall Lanczos time.

By using OpenMP directives as described above, parallelism may be achieved by splitting the computation into multiple threads of execution. In the current implementation of the Hybrid MPI/OpenMP approach, the number of threads spawned for the OpenMP regions is determined statically at the start of the MFDn run. However, it might be necessary, during the run, to be able to adapt to the changing system conditions in terms of computational resource availability. In light of this, a strategy which involves changing the number of threads dynamically based on certain information about the state of the system resources at run-time might very well prove to be useful, especially in the presence of any competing applications. This work explores changing the number of OpenMP threads spawned at the beginning of every Lanczos iteration (as illustrated in Fig. 3.3) using system information gathered at run-time by a middleware engine integrated with MFDn.

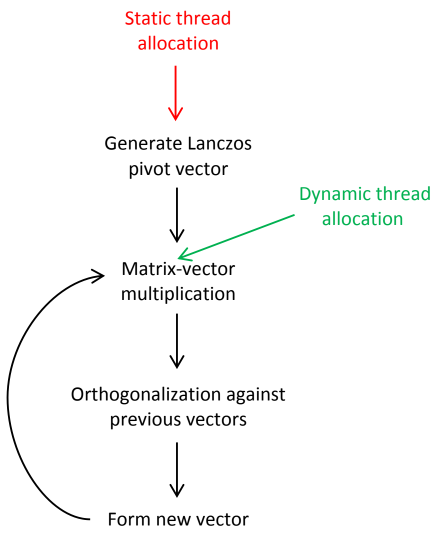

Static thread allocation
Generate Lanczos pivot vector
Dynamic thread allocation
Matrix-vector multiplication
Orthogonalization against previous vectors
Form new vector

Figure 3.3 Iterative model for the Lanczos procedure (The Lanczos pivot vector is the initial working vector required for the first MATVEC).

3.3 Using Middleware NICAN with applications

While running parallel and distributed applications, the assumption that the resources are dedicated to running only the current job may be too restrictive. This is especially true in the case of interactive cluster environments or cloud computing testbeds where users can simultaneously run different applications sharing resources, such as Processing Elements (PEs), I/O, main memory, and network. In such cases, system resource availability often changes during the course of execution of the application. This calls for certain run-time adaptations in these applications to be able to adjust to the dynamically changing system conditions. However, it is not desirable to insert these adaptations into the source code of an application, such as MFDn, since this will increase the complexity of the scientific code with adverse affects on its accuracy and usability. The latter is of particular concern since high performance applications are supposed to run on a computational platform by an application scientist who may not be an expert in computer architecture and performance tuning. Hence, there is a need for a generic middleware tool which can monitor the system resources and invoke appropriate run-time adaptations for a large class of applications, so that such a tool may be quickly geared towards a specific application. While leaving its main architecture and system monitoring capabilities intact, the middleware may be augmented by a specific application module Ustemirov et al. (2006), thus acting as an interface between the hardware and the application execution. In this work, the middleware tool NICAN Sosonkina (2006) developed at Iowa State University is used to serve as an interface with MFDn.

3.3.1 NICAN Overview

The main idea of integrating NICAN with an application is to decouple the system-related monitoring and decision making from the execution of the application, while timely invoking application adaptation functions (handlers). The NICAN engine is encapsulated into a separate thread, called Manager, which controls the functional modules and invokes application adaptations. Due to dynamically loadable modules, NICAN is versatile and may have a wide variety of interactions with the system or application. Each module is designated to perform a separable function, such as to determine a system runtime characteristics or to validate machine-dependent parameters. NICAN has a rather general and flexible interaction mechanism, which permits to "talk" to a variety of application codes. Enhanced with general-use modules, such as CPU monitoring or disk I/O checking, NICAN may not require customized integration with an application. However, to explore application-specific trigger conditions, specific-use NICAN modules may also be needed.

NICAN is mostly used with distributed applications running on many compute nodes of a cluster. The general architecture of integration (Fig. 3.4) involves a single instance of the Manager on one node, usually on the node on which the rank 0 (root) process is executing, and an instance of the daemon module on each of nodes executing the application. The root node shall henceforth be referred to as the *kickoff* node, with the remaining nodes being referred to as *remote* nodes. The main function of the daemon module is to act as an interface between the Manager and the distributed processes of the application. In some cases, it is also used to pick up system-related information on the remote nodes, which is to be relayed to the Manager to aid the decision making process.

An attractive feature of NICAN is that it does not require substantial coding modifications to the high-performance application with which it is interfaced. In the case of MFDn, only a few

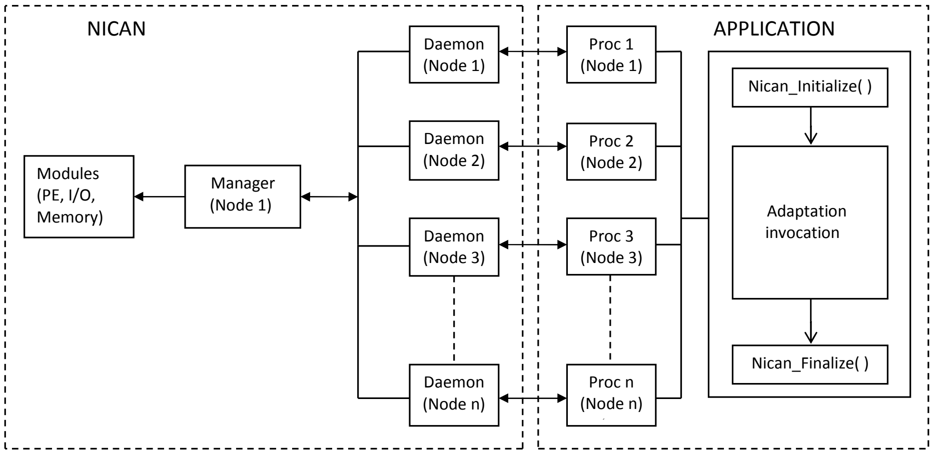


Figure 3.4 General architecture for integration of NICAN with a distributed application.

changes were made to the source code. Specifically, they include starting up NICAN, tearing it down, and the application specific adaptation handler, such as changing number of threads dynamically based on the information conveyed by NICAN. Resource monitoring, analysis and triggering of the adaptive mechanisms are implemented within the NICAN. Another salient feature of NICAN is to enable or disable its actions with ease and on-demand by the application. This fits very well with the idea of NICAN as a "black box" from an application scientist's perspective, abstracting away the details of its functioning.

3.4 Integration model and adaptation strategies

A major benefit of integrating NICAN with an application is to separate the system-based monitoring from the invocation of adaptations which are application related. The MFDn-NICAN integration may accomplish the goals described in this work by NICAN monitoring the workload on a core and deciding the number of threads to be spawned by MFDn at particular iterations of the Lanczos process. By collecting the information on the number of running threads/processes resident in the system, a decision may be made to change the thread count for the next iteration in order to avoid the oversubscription of a core. The core oversubscription occurs when more than one thread is running on the core processing element and has been shown to be detrimental for the application performance Apparao et al. (2008). It is clear that the oversubscription monitoring and avoidance is a dynamic process which may not be dealt with effectively using static tuning and configuration. Thus, dynamic resource monitoring and interfacing with applications as provided by NICAN is fully exploited in this work.

3.4.1 Oversubscription and context switching

Processors equipped with multiple cores have become ubiquitous in modern high-performance computers. The number of cores is growing higher in order to keep up with Moore's law, further aggravating the "Memory Wall" which is caused by the inability of memory access to keep up with the speed of processing. In such a scenario, to prevent cores from idling, the use of multi-threading can increase the efficiency of applications by masking memory accesses and communications with computations. Having multiple threads of execution in applications enables to extract thread-level as well as instruction-level parallelism on modern multi-core architectures. Some typical examples of multi-core processors are the Intel Core duo (2 cores), the AMD Phenom II X4 (4 cores) and the AMD FX-8150 (6 cores).

The concept of multi-threading brings into light the idea of context switching. A context switch means storing and restoring the processor state to resume execution from the point where the switch occurred. With regard to threads, it means switching the flow of execution among the different threads which execute on a single PE core sharing the same functional units and execution pipeline, in addition to resources such as caches and TLB (Translation Lookaside Buffer). The intervals between which context switching occurs are determined by the operating system scheduler which usually gives a time slice to each thread/process before preemption and control switching over to another thread/process. Context switches can be detrimental to the performance of a multi-threaded application due to the scheduler overhead of switching among the threads which execute on the same core. This process of switched execution of threads is known as *Simultaneous Multi-Threading (SMT)* in operating system parlance. Some modern processors however, such as the Intel Nehalem (Core i7) and the Itanium 9300, show good performance with SMT by providing an environment which gives the notion of having multiple virtual processors per physical core. This concept is called hyper-threading, but is beyond the scope of this work and is hence not addressed here.

Even in the absence of hyper-threading, most modern operating systems are well equipped to handle multi-threading depending on the number of cores available on a processor. The threads are usually distributed over the all the cores by the scheduler. This ensures that they truly execute in parallel, since each PE core has its own functional units and execution pipeline. Thus, to exploit the benefits of multi-threading, it is best to have the number of threads equal to the number of cores. To have more threads than the number of cores is commonly known as *oversubscription* because of the overhead incurred in context switching among the threads executing on the same core.

3.4.2 Architecture of integration

In MFDn, as per the hybrid MPI/OpenMP approach presented in Maris et al. (2010), MPI is used for distributed memory communication among the nodes with OpenMP being used to spawn multiple threads within a node. The aim is to increase efficiency by taking advantage of the multiple cores and shared memory structure on the node. Under the hybrid MPI/OpenMP approach on a majority of architectures, we would want to run MFDn with one MPI process per node and the number of threads per process equal to the number of cores on a node to prevent any cores from idling. However, in realistic situations, there is a possibility of applications, e.g., run by other users on the same node(s), competing for the PE resources. This causes context switching to occur which can be detrimental to the performance of MFDn. This is especially true when the competing threads/processes are compute-cycle intensive, as is commonly the case with large scale scientific applications. Hence, there is a need to develop strategies for detecting possible oversubscription during run-time and invoking appropriate adaptations in MFDn to be able to adjust to the changing system conditions. In this section, the architecture employed for the MFDn-NICAN integration is described along with the system monitoring and decision making strategies used for the invocation of adaptations in MFDn. These strategies are sufficiently general to be employed with other applications.

In the previous section, the general architecture for the integration of NICAN with an application was explained along with the main components of the NICAN engine. The same architecture is employed for the integration with MFDn with the modules used for system monitoring geared towards the need to change the number of threads spawned by MFDn at run-time. Fig. 3.5 depicts the architecture used for the integration at the Lanczos stage of MFDn.

The MFDn-NICAN integration model includes a PE load module which monitors the number of running threads/processes in the system during the run-time of MFDn. The main function of this module is to detect if there is oversubscription on any of the cores due to the presence of competing applications. This module is loaded by the NICAN Manager thread, which is started on the *kickoff* node and which interfaces with MFDn via the daemon module. A daemon instance is started on every node which executes the application. Besides being the Manager's contact point for all the NICAN-integrated applications running on the node, the daemon module also picks up information regarding the number of threads to be relayed to the Manager.

The Manager uses the thread information obtained from the PE and daemon modules to make a decision regarding the number of threads to be spawned by the Lanczos iterative algorithm at the beginning of an iteration. It then invokes the appropriate adaptations via the adaptation function. A simple algorithm is employed for the decision making process. Define (1) the total number of running threads in the system as Θ_s, (2) the number of MFDn threads spawned Θ_a, (3) the number of competing application threads Θ_c and (4) the number of PE cores per node K, the number of MFDn threads that should be spawned for the ith iteration for a particular node is calculated using Algorithm 1.

Algorithm 1 Number of MFDn threads to be spawned at the ith Lanczos iteration

```
$\Theta_c(i) = \Theta_s(i) - \Theta_a(i-1) - 1$
if $\Theta_c(i) \geq K$ then
    $\Theta_a(i) = 1$
else
    $\Theta_a(i) = K - \Theta_c(i)$
end if
```

Θ_s is found as a result of the monitoring process. K is a known constant for a node. In the first line of Algorithm 1, the current number of MFDn threads along with the thread which does the actual monitoring are subtracted from Θ_s. The NICAN thread is very lightweight and hence does not cause any performance penalty in MFDn due to context switching.

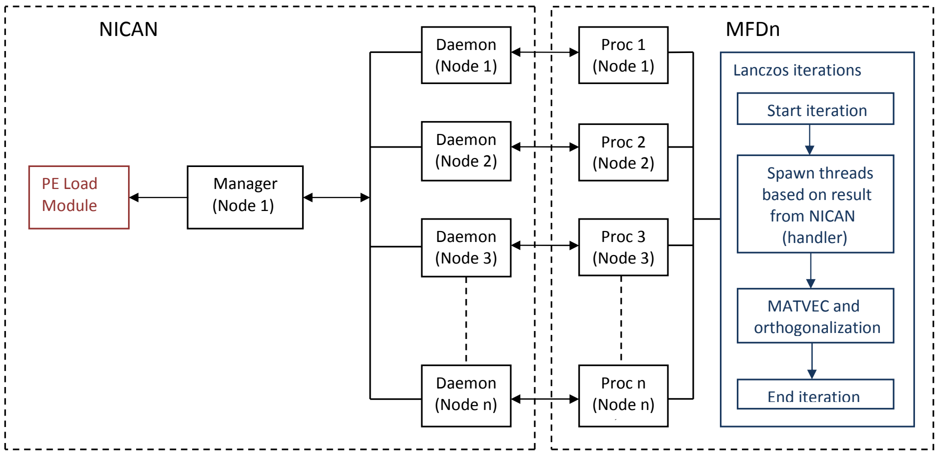


Figure 3.5 Architecture of the MFDn-NICAN integration.

With regard to implementation, information retrieved from the `/proc/loadavg` file is used by the NICAN modules to find the number of running threads/processes in the system during the run-time of MFDn. The number of threads as returned by NICAN after the decision making process is spawned for a particular iteration by MFDn by making use of the OpenMP function `omp_set_num_threads(numthreads)`. Here, `numthreads` refers to the number of threads to be spawned for a subsequent OpenMP region. Communication between NICAN and the distributed processes of MFDn is established using the TCP/IP socket library.

3.5 Experimental results and discussion

In this section, the experiments conducted with the MFDn-NICAN integrated model are presented along with some results that were observed in terms of improvement in performance with the inclusion of dynamic adaptations. In these experiments, the performance of the MFDn-NICAN code is tested in comparison with MFDn executing in a cluster environment. The aim is to observe the impact of changing the number of OpenMP threads spawned dynamically as opposed to running with a fixed number of threads. The NICAN middleware tool is used to gather information about the system, in particular, the number of running threads/processes sharing the PE resources on each node. Based on this information, a decision is taken by the tool regarding the number of threads to be spawned which is then relayed to MFDn at regular intervals to invoke the appropriate adaptations.

The testbed used for the experiments was the "Dynamo" cluster consisting of 34 SMP compute nodes, each having two quad-core 3.0 GHz Intel Xeon E5450 processor chips and 16 GB of RAM, i.e., equipped with a total of 8 cores per node and 272 cores overall. The nodes are connected with both Gigabit Ethernet and DDR Infiniband. In these experiments, both MFDn and MFDn-NICAN are run for ^{12}C nucleus using six MPI processes, one on each node. Furthermore, each process spawns eight OpenMP threads, which is specified at the start of run, thus ensuring that none of the cores on a node are left idle. For these experiments, multi-threaded regions are defined only during the Lanczos iterations with the rest of the code being single threaded. With MFDn-NICAN, the number of threads is subject to change during the course of the run depending on the PE resources available, while it is held constant throughout the run in the case of pure MFDn. The aims are (1) to consider the penalty incurred due to context switching in the presence of any application which competes for the same PE resources and (2) to show the usefulness of integrating NICAN with MFDn in coping with such a situation.

The tests were carried out for two problem sizes, $N_{\max} = 2$ and $N_{\max} = 4$ for the ^{12}C nucleus. The problem size is characterized by the dimension of the Hamiltonian matrix which is 17,725 for $N_{\max} = 2$ with 1,697,935 non-zero elements and 1,118,926 for $N_{\max} = 4$ with 279,405,126 non-zero elements. In general, as the $N_{\max}$ value increases, so does the size of the Hamiltonian matrix yielding more computationally-intensive and more accurate calculations. The bar graphs in Fig. 3.6 and 3.7 depict performance results that were obtained for both MFDn and MFDn-NICAN for these two problem sizes with varying degrees of competition. The competition is defined as the percentage of the total (8) cores per node occupied with other high-performance applications. For the purpose of competition, the quantum chemistry software GAMESS Baldridge et al. (1993) was used in the configuration that has been shown to be compute-cycle intensive. GAMESS processes were introduced on the nodes during the run-time of MFDn and their impact on the performance for both pure MFDn and MFDn-NICAN was observed.

From the two graphs (Fig. 3.6 and 3.7), it can be clearly seen that under increasing competition, the performance of MFDn reduces considerably due to context switches happening on

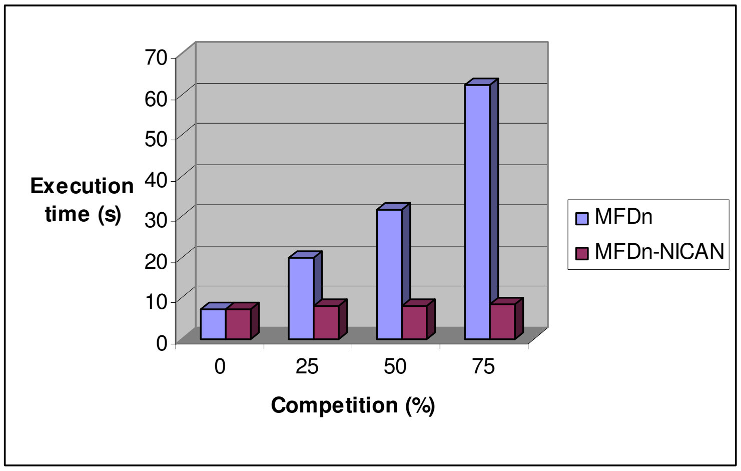


Figure 3.6 Comparison of execution times of MFDn and MFDn-NICAN for ^{12}C nucleus with $N_{\text{max}} = 2$.

more cores. On the other hand, with MFDn-NICAN, the performance is much better, being almost equal to that of MFDn with full resource availability (i.e., with no competition and the maximum number of threads) for the lower problem size $N_{\text{max}} = 2$. For $N_{\text{max}} = 4$, however, the performance of the adaptive algorithm does not reach the peak performance since with competition from other applications, it is forced to run on fewer threads than the maximum possible. This hinders the performance for larger problem sizes, as in the case of $N_{\text{max}} = 4$ here, that require full power of the node PE resources. Such a trade-off is acceptable, however, since the performance penalty incurred due to context switching between MFDn and competing applications leads to a much slower execution. As is evident from the graphs, the penalty increases with the increase in competition. Fig. 3.8 depicts the speed-up obtained for the multi-threaded Lanczos iterative procedure with the number of threads for the larger problem size $N_{\text{max}} = 4$. The graph confirms the observations presented in Maris et al. (2010): The scaling suffers going from two to eight threads since the multithreading fully parallizes only certain parts of the computation while the critical section and the MPI communication overhead are still present.

Fig. 3.9 and 3.10 illustrate the performance of various sections of the MFDn code, again with different degrees of competition for both the non-adaptive and adaptive algorithms. This serves to determine which section is incurring the most performance penalty due to the context

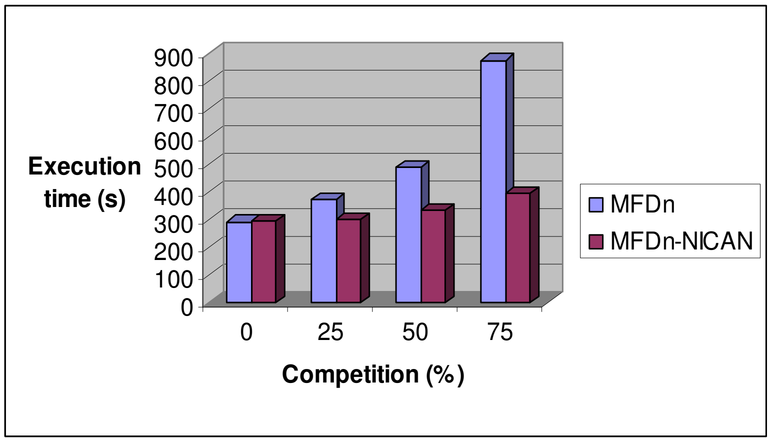


Figure 3.7 Comparison of execution times of MFDn and MFDn-NICAN for ^{12}C nucleus with $N_{\max} = 4$.

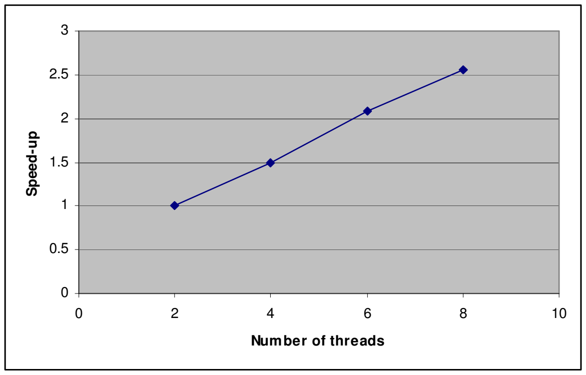
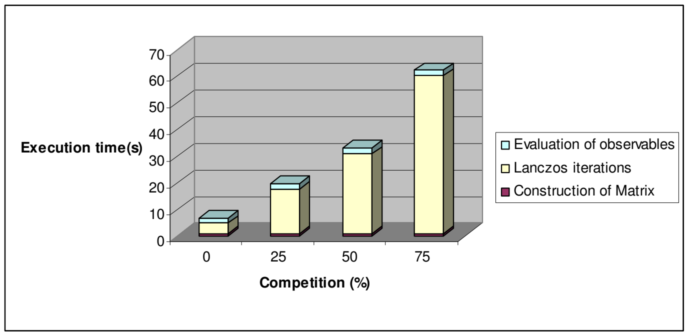


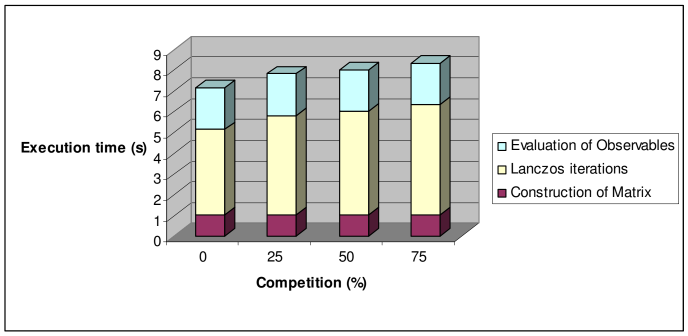
Figure 3.8 Scaling of the Lanczos iterative phase with the increase in the number of threads per MPI process.

switching. This is again shown for the same two problem sizes i.e., $N_{\max} = 2$ and $N_{\max} = 4$.

It can be seen that the multithreaded Lanczos iterative procedure incurs a higher penalty as the degree of competition increases. Thus, it bears the primary responsibility for the performance decrease whereas the other sections, namely, the construction of the Hamiltonian matrix and the evaluation of observables, which are single-threaded, retain the same performance even in the presence of competition. This is not surprising since the multiple threads in the Lanczos procedure are spread across all the cores and undergo context switching in the presence of competition on any of the cores while the calculations in the other two sections are undisturbed since they enjoy a dedicated core. (Note that the maximum competition is 75% meaning that MFDn has always a sole use of at least two cores, on which it performs the construction of



(a) MFDn

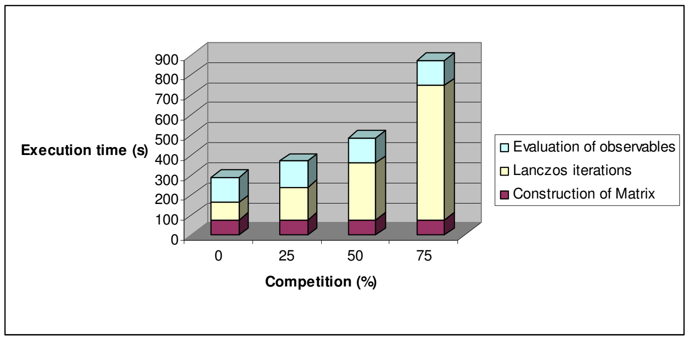


(b) MFDn-NICAN

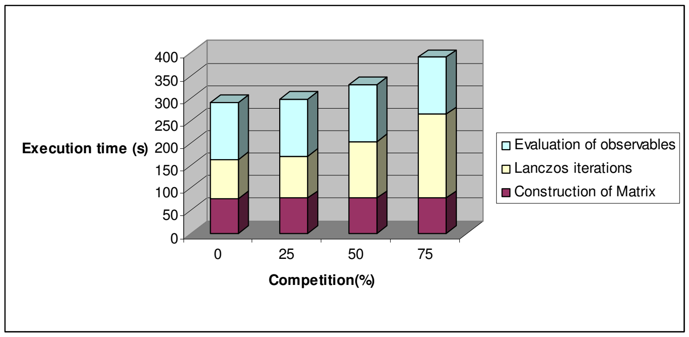
Figure 3.9 Execution times of different sections of the code with different degrees of competition for ^{12}C nucleus with $N_{\max} = 2$.

Hamiltonian and the observable calculation.)

These experiments indicate the usefulness of including adaptive capabilities in MFDn by dynamically changing the number of threads to deal with the problems of oversubscription and context switching. The advantage of using a middleware for this purpose, as explained in the earlier sections, is to decouple the system-related monitoring and decision making from the execution of the application without incurring much performance and interfacing overhead for adaptation. The performance gains obtained as a result validate this adaptive approach for MFDn. Since a generic middleware tool is employed to implement this approach, the strategies presented here may be extended to other multi-threaded distributed high performance applications.



(a) MFDn



(b) MFDn-NICAN

Figure 3.10 Execution times of different sections of the code with different degrees of competition for ^{12}C nucleus with $N_{\max} = 4$.

CHAPTER 4. Memory affinity strategy in multi-threaded sparse matrix computations

4.1 Background and significance

Transistor densities have been growing in accordance with Moore's law resulting in more and more cores being put on a single processor chip. With the increasing core counts on modern multi-processor systems, main memory bandwidth becomes an important consideration for high performance applications. The main memory sub-system can be of two types nowadays: Uniform Memory Access (UMA) or Non-Uniform Memory Access (NUMA). UMA machines consist of a single physical memory bank for the main memory, which may lead to the memory bandwidth contention when there are many application threads trying to access the main memory simultaneously. This problem of scalability may be alleviated by NUMA architectures wherein the main memory is physically split into several memory banks, with each bank associated to a set of cores, the combination of which is called a NUMA node. The cores associated with a particular NUMA node have a direct link to their own local memory bank, thus enabling fast memory access for their threads when accessing from this local bank. Thus, by having subsets of threads accessing data locally from individual memory banks, memory contention may be reduced among the threads.

However, accesses to remote memory banks as in the case of large shared arrays, for example, may become painstakingly slow since they have to take place over an interconnect. This may negatively affect the application scalability for higher thread counts Lameter (2006). Thus, it is imperative to carefully consider which parts of the shared data should be attributed to which physical memory bank based on the data access pattern or on other considerations. Such an attribution of data to physical main memory is often called *memory affinity* Bellosa and

Steckermeier (1996); Kleen (2005). This notion goes hand in hand with the CPU affinity, as noted in Grant and Afsahi (2007), such that the threads are being bound to specific cores for the application start and their context switches are disabled. Once threads are bound, the memory may be pinned too. On multi-core NUMA platforms, the ability to pin the memory in the application code becomes important since it is generally most beneficial for a data portion local to a thread to be placed on the memory bank local to the core it is executing on[1], so as to ensure the fastest access Antony et al. (2006).

Conversely, the default memory affinity policy — used in most Linux-type operating systems — is enforced system-wide for all the application. This policy, called *first-touch*, ensures that there is fast access to at least one memory bank regardless of the shared data access pattern within application threads Iyer et al. (2002). Specifically, the data is placed in the memory bank local to the thread writing to it first, which is typically done by the master thread. Thus, the downside of the first-touch policy is that all the threads accessing this shared data converge to this NUMA node, as shown in Fig. 4.1, causing bandwidth contention in the memory bank servicing the master thread. The problem may be exacerbated since the master thread typically initializes multiple shared data structures. Since the threads have to go out of their local NUMA node for accessing the data, the remote access latencies are also incurred, which causes the application performance overhead increase. Thus, the default first-touch memory placement policy calls for improvement to achieve better scalability, which may be obtained using already existing software libraries to work with NUMA nodes Kleen (2005).

The motivation for the present work was the need for improvements in sparse matrix-vector multiplications (SpMV), which constitute the bulk of computational load in large-scale applications modeling physical phenomena using structured or unstructured matrices Saad (2003). In particular, the nuclear physics application MFDn handles very large sparse unstructured matrices arising in the solution of the underlying Schrödinger equation, as discussed earlier.

[1] Here and throughout the chapter, it is assumed that only one thread is executing per core and there is no oversubscription of cores, as has been studied, e.g., in Srinivasa et al. (2011).

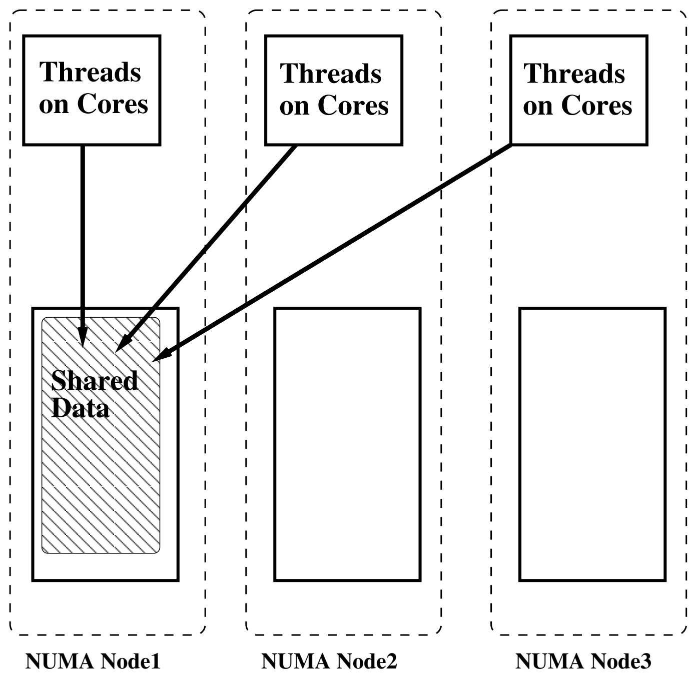


Figure 4.1 Shared data access pattern with the default *first-touch* policy on a NUMA architecture. A dashed curved-corner rectangle represents NUMA node.

4.2 Proposed memory placement strategy

The goal of the proposed memory placement strategy is to minimize the data transfer overhead between main memory and the application code when accessing shared data. Hence, the default (*first-touch*) placement has to be changed according to certain application and system considerations Goglin and Furmento (2009). In a nutshell, the following general steps need to be taken to study the application at hand to determine the memory placement for its shared data structures:

Step 1: Identify all the shared data structures in the application

Step 2: Classify them as having *deterministic* and *non-deterministic* access pattern by threads.

- For deterministic: Find a *chunk*-to-thread correspondence; Pin each chunk to the memory bank local to the corresponding thread.
- For non-deterministic: Spread the data across all the memory banks.

The classification step (Step 2) may be performed based on a definition of the *deterministic* and *non-deterministic* accesses to a data structure. In the former, portions of the structure is accessed by a thread exclusively, while several thread may access a portion in the latter case.

This definition is rather general and is featured, for example, in the case of multi-threaded loop parallelization, such that a block of loop iterations is dedicated to a thread. If the loop index corresponds to a data portion (called *chunk*), such as that of a shared array, then each thread accesses its own array chunk exclusively. Such an array may be classified as having deterministic access and then distributed among specific memory banks. Fig. 4.2 presents the obtained distribution to the local NUMA nodes, such that vertical arrows emphasize the local access patterns, that minimizes the access latency. Since, for the non-deterministically accessed data structures, their thread access pattern and timing may not be known in advance, they are spread out in a fine-grain fashion across all the memory banks, as sketched in Fig. 4.3, in an attempt to alleviate the memory bandwidth contention. Algorithm 2 specifies array chunk sizes attributed to each thread and, consequently, to each NUMA node by accepting the following inputs:

▷ Total array dimension *dim_total*;

▷ Total number of threads *nthreads*;

▷ Number of NUMA nodes *mnodes* (system parameter);

▷ Number of cores *lcores* per NUMA node *lcores* (system parameter).

and producing two outputs:

◁ Chunk size $dim\_per\_thread(i)$ attributed
to thread i, $(i = 1, \ldots, nthreads)$.

◁ Chunk size $dim\_per\_node(j)$ attributed
to NUMA node j, $(j = 1, \ldots, mnodes)$.

Note that each NUMA node is typically associated with several cores — thus, with a group of threads (one thread per core).

Algorithm 2 splits the data structure into chunks in accordance with the *exact assignment* thread access pattern, in which each thread is assigned an (almost) equal contiguous portion of the data structure. This pattern is common among multi-threaded programming models, such as OpenMP Dagnum and Menon (1998), with the default assignment size to ensure contiguous data in each chunk. Additionally, the thread scheduling (also called work-sharing) is assumed to be *static*, so that it is known before the loop execution. Thus, once the contiguous chunk

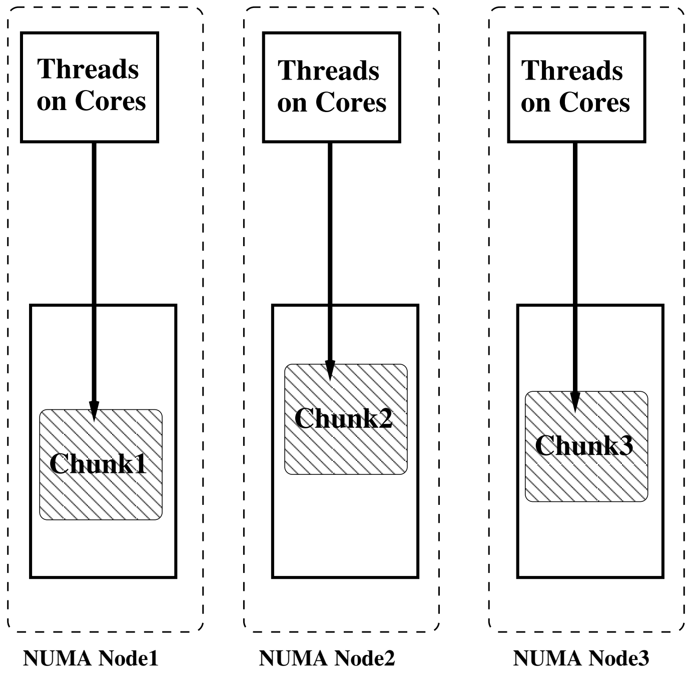


Figure 4.2 Proposed placement of the shared data accessed deterministically A dashed curved-corner rectangle represents NUMA node.

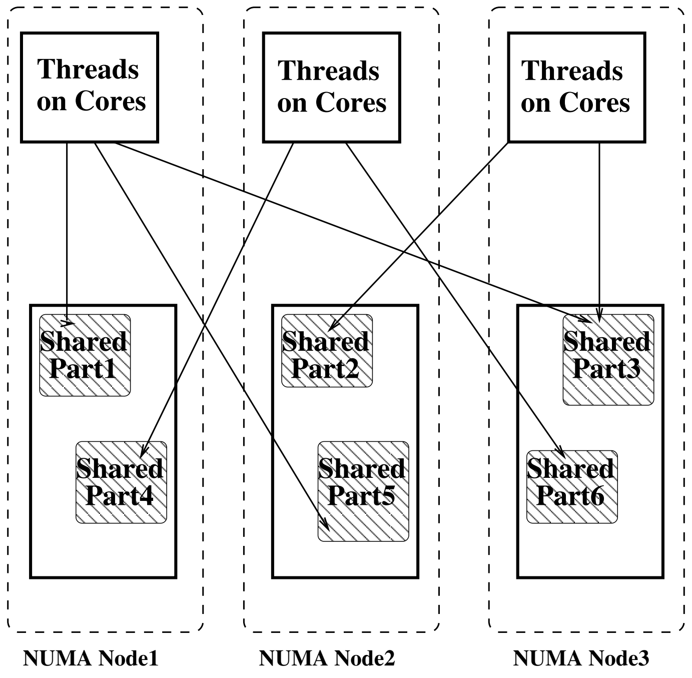


Figure 4.3 Interleaved placement of the shared data accessed non-deterministically. A dashed curved-corner rectangle represents NUMA node.

Algorithm 2 Determine chunk size per NUMA node.

```
for $j = 1$ to $mnodes$ do
    $dim\_per\_node(j) \leftarrow 0$
end for
$per\_thread\_dim \leftarrow ceiling(dim\_total/nthreads)$
$virtual\_dim \leftarrow per\_thread\_dim \times nthreads$
$offset \leftarrow virtual\_dim - dim\_total$
for $i = 1$ to $(nthreads - offset)$ do
    $dim\_per\_thread(i) \leftarrow per\_thread\_dim$
end for
for $i = (nthreads - offset + 1)$ to $nthreads$ do
    $dim\_per\_thread(i) \leftarrow per\_thread\_dim - 1$
end for
for $j = 1$ to $mnodes$ do
    for $i = lcores \times (j - 1) + 1$ to $lcores \times j$ do
        $dim\_per\_node(j) \leftarrow dim\_per\_node(j) + dim\_per\_thread(i)$
    end for
end for
```

sizes are determined by Algorithm 2, the actual chunk attribution is accomplished by providing a mapping of chunk number to NUMA node number, where array chunks and NUMA nodes are numbered consecutively, as in Fig. 4.2, for example.

4.3 Implementation details

The NUMA application programming interface (API) Kleen (2005) available for Linux is used in this work to control the data placement for shared arrays, overriding the default *first-touch* memory affinity policy employed by the operating system. This API offers two principal memory placement policies called *bind* and *interleave*. The former places (binds) memory of an application on a selected memory bank or set of banks whereas the latter spreads (interleaves) data on a page-by-page basis over the memory banks of a NUMA machine. If applied throughout the entire application, each policy may be too restrictive since it is often necessary to tailor the memory attribution to a particular access pattern of a data structure Ribeiro et al. (2009). For the fine-tuning purposes, the NUMA API provides a system call `mbind()` which may be used to apply these affinity polices selectively to certain regions of the memory. The `mbind()`

interface has been used in this work to implement the proposed shared data placement in which certain portions of shared arrays are to be assigned to memory in accordance with their access pattern within the multi-threaded application at hand. To benefit from the selective and intelligent data placement on the memory banks, the thread migration or their context switch have to be disabled. In other words, the CPU affinity must be observed, which may be accomplished with the `sched_setaffinity()` system call also available on Linux systems.

An important aspect to consider when using `mbind()` is that it is designed on work on large chunks of data which are aligned on a page boundary i.e., the starting address of the chunk should be an integral multiple of the system page size. So, once the shared array chunks have been determined, it becomes necessary to check whether each such chunk is page aligned before consigning it to a NUMA node. Otherwise, the nearest page-aligned address to the chunk is to be determined, starting from which the chunk can be pinned to the appropriate NUMA node. Such a pinning may cause an address mismatch between the page-aligned and the actual chunk boundaries as determined in Algorithm 2. To minimize the occurrence of the mismatches, all the shared arrays are allocated starting on a page boundary using the C function `valloc()`. With this set-up, the maximum difference between the mismatched addresses per NUMA node is estimated to be half of the system page size. Note that a typical system page is of the order of 10^3 bytes. Since high performance applications routinely work with the data in the order of gigabytes, this mismatch has no serious impact on the effectiveness of the strategies described in this work.

4.3.1 Shared arrays in sparse matrix-vector multiply

The sparse matrix-vector multiply (SpMV) forms an important computational core in many scientific applications. Hence, it is highly beneficial to employ its efficient implementation. Its naive implementations, however, may suffer from poor performance on multi-core NUMA architectures mainly due to the memory contention and latency problems as will be evident from the experiments described in Section 4.5. Therefore, the strategies described in Section 4.2 are being applied to sparse matrix data structures, such that the most common matrix storage formats are considered. Specifically, sparse matrices are characterized by a very large percentage

```
for $i = 1$ to $n$ do
    for $k = \mathtt{ptrA}(i)$ to $\mathtt{ptrA}(i+1) - 1$ do
        $y(i) \leftarrow y(i) + x(\mathtt{jA}(k)) \times \mathtt{A}(k)$
    end for
end for
```

```
for $i = 1$ to $m$ do
    for $k = \mathtt{ptrA}(i)$ to $\mathtt{ptrA}(i+1) - 1$ do
        $y(\mathtt{jA}(k)) \leftarrow y(\mathtt{jA}(k)) + x(i) \times \mathtt{A}(k)$
    end for
end for
```

Figure 4.4 Pseudo-code for sparse matrix-vector multiplication in the CSR (top) and CSC (bottom) format after the output vector initializations are performed.

— often as much as 95% — of zero entries, which are not stored for performance and space reasons. As a rule of thumb, a $n \times m$ sparse matrix is represented by three one-dimensional arrays:

▷ `A`, for all the non-zero values

▷ `jA` for their positions in the in each row or column.

▷ `ptrA` for the pointers to the beginning of each column or row.

Such a storage format is called Compressed Sparse Row (CSR) or Compressed Sparse Column (CSC) depending on whether column or row indices are being stored in `jA`, respectively. Then, a multiplication of the sparse matrix stored in CSR or CSC by a vector x of size m may be performed to obtain a vector y of size n, as shown in the top and bottom code segments, respectively, in Fig. 4.4.

Sparse matrices are shared among the threads involved in the SpMV computation and need to be bound to local memory banks to ensure minimal data transfer overhead. As is evident from the code segments in Fig. 4.4, the outer loop is over the rows (for CSR) or columns (for CSC) of the matrix. For a typical multi-threaded SpMV, this loop is parallelized such that each thread gets a certain number of rows/columns to work with. However, several threads may access the same vector components to read or write their values. Since the precise timing of the read/write operations may vary dynamically, for coherent results, the sequencing of the write accesses in the CSC format (line 3 of the bottom code segment in Fig. 4.4) must be

Table 4.1 Shared array access and pinning for CSR.

Array	Access		Policy
`A`	Deterministic	Read	Bind
`jA`	Deterministic	Read	Bind
`ptrA`	Deterministic	Read	Bind
x	Non-deterministic	Read	Interleave
y	Deterministic	Write	Bind

Table 4.2 Shared array access and pinning for CSC.

Array	Access		Policy
`A`	Deterministic	Read	Bind
`jA`	Deterministic	Read	Bind
`ptrA`	Deterministic	Read	Bind
x	Deterministic	Read	Bind
y	Non-deterministic	Write	Interleave

enforced in some way. On the contrary, in the CSR format, the SpMV has no shared arrays with simultaneous write accesses by threads, so no sequencing is needed.

The presence of shared arrays and parallelizable loops makes SpMV an ideal candidate for testing the proposed memory affinity policies. Once these shared arrays have been identified, they are assigned to the deterministic or non-deterministic category based on the thread access patterns. The bind and interleave strategies are then applied in accordance with the two-step strategy from Section 4.2. Tables 4.1 and 4.2 provide information regarding the shared array access patterns which are part of the CSR and CSC multiplications, respectively, along with the NUMA policy used for each.

It may be observed that sparse matrices are shared among the threads having exclusive access to their portions in the SpMV computation, and thus, need to be bound to local memory banks to ensure minimal data transfer overhead. On the other hand, the vectors x and y may be shared with either deterministic or non-deterministic access depending on the type of the storage format considered. Thus, to effectively distribute the shared arrays with the deterministic access pattern, it becomes necessary to select specific portions (chunks) of these arrays which are accessed by each thread. To accomplish this task, the output of Algorithm 2, i.e., the chunk size of each NUMA node, is used to determine the array staring and ending indices that delineate each chunk boundaries.

Application interface. To facilitate the usage of proposed memory placement strategies, a high-level interface set, termed MASA-SpMV (Memory Affinity for Shared Arrays-Sparse Matrix Vector multiply) has been developed for sparse matrix-vector multiply in CSC or CSR formats. This interface, encapsulating the implementation of Algorithm 2, determination of the contiguous chunk start and end positions within the arrays and the memory pinning function calls from Kleen (2005), is composed of the C function signatures as follows:

▷ `void masa_setcpuaff(int tid)`

- Pins the the worker threads to the corresponding cores based on the thread ID which is passed in as argument.
- Uses the Linux system call `sched_setaffinity()` for this purpose.

▷ `void masa_preprocess()`

- Determines the system constants *mnodes* and *lcores* required by Algorithm 2, maintained as global variables.
- Catches environment variables which deal with multi-threading, such as OMP_SCHEDULE in OpenMP.
- Disables the proposed strategies if the thread scheduling differs from static.

▷ `void masa_allocate(void **sharedarray)`

- Allocates the shared array as aligned to a page boundary.

▷ `void masa_compute_chunks(int dim_total,`
`int nthreads, void *ptrA)`

- Computes the chunk size per NUMA node (Algorithm 2) for each shared array used in the SpMV computation.
- Finds chunk-to-node mapping for these shared arrays and stores it using global data structures.

▷ `void masa_distribute(void *A, void *jA, void *ptrA, void *x, void *y)`

- Determines the nearest page-aligned address for each shared array chunk.
- Pins the shared arrays according to the mapping calculated in `masa_compute_chunks` by

calling `mbind()`.

The MASA-SpMV interface has been packaged, fully documented and is widely available for download from www.scl.ameslab.gov. The software is provided “as is” under the LGPL license. It is designed mainly for the Linux operating system running on multi-core NUMA platforms. To use the interface, one will need the NUMA API installed on their system (this can be done using either a numactl RPM or libnuma-dev) and a compiler with OpenMP enabled.

4.4 Test applications

Armed with the proposed strategies and their incarnation in the MASA API interface, the proposed strategy may be employed in realistic settings of scientific application codes. Two applications have been selected for their reliance on parallel SpMV: CG (Conjugate Gradient) NAS benchmark code and the *ab-initio* nuclear structure calculation code MFDn. Both codes exploit multi-threaded parallelism using OpenMP, in which the SpMV loop is parallelized as shown in Fig. 4.4(top).

4.4.1 CG: NAS parallel benchmark

NAS Parallel Benchmarks (NPBs) is a suite derived mainly from computational fluid dynamics (CFD) codes and is composed of both entire applications and computational kernels Jin et al. (1999). In particular, the CG kernel been selected for this work. It consists of an iterative solution of a linear system of equations with a sparse symmetric matrix and is performed as part of a “outer” eigenvalue computation. The most computationally intensive stage of the CG iterative method is sparse-matrix vector multiplication. As implemented in the CG of the NAS suite, this multiplication stores the full matrix — without regard for its symmetry — in the CSR format. Its main loop features multi-threaded parallelism with OpenMP[2].

4.4.2 MFDn: realistic application

The MFDn code has already been intoduced earlier and is used in this particular context because of its use of parallel SpMV. In MFDn, each Lanczos iteration spends most time in

[2]This implementation has been provided by the OMNI compiler group.

```
for $i = 1$ to $m$ do
    for $k = \texttt{ptrA}(i)$ to $\texttt{ptrA}(i+1) - 1$ do
        $y(\texttt{jA}(k)) \leftarrow y(\texttt{jA}(k)) + x(i) \times \texttt{A}(k)$
        $y^t(i) \leftarrow y^t(i) + x^t(\texttt{jA}(k)) \times \texttt{A}(k)$
    end for
end for
```

Figure 4.5 Pseudo-code for sparse matrix-vector multiplication in MFDn for the off-diagonal processors.

SpMV with the Hamiltonian matrix, only the lower half of which is stored (in the CSC format) to save memory.

Under the hybrid MPI/OpenMP approach for MFDn, the sparse matrix data is partitioned among the available compute nodes and is being exchanged by using the MPI distributed communication library. Then, the local portions of the data are being accessed also but, this time, by using multi-threaded programming tools such as OpenMP. Fig. 3.1 shows the MFDn sparse matrix distribution across the available MPI processes, which are organized in the 2×2 grid. The off-diagonal processors (numbered $6 - 15$ in Fig. 3.1) have more work to do during the SpMV phase since they have to work with the upper half of the matrix as well (for computing the transpose output vector) which is not stored in memory. The code segment in Fig. 4.5 describes the SpMV for MFDn on an off-diagonal MPI processor with x^t and y^t referring to the components of the input and output vectors, respectively, used with the transposed matrix (i.e., upper matrix half). In essence, this SpMV morphs the two loops shown in Fig. 4.4 into one, such that its multiplication operation in line 3 is the same as the one in Fig. 4.4(bottom). Therefore, during its multi-threaded execution, the write operation on y has to be also performed in sequence by the threads involved.

Using OpenMP, the serialization of the write operation has been implemented by way of a "critical section", which is entered one thread at a time and, thus, exhibits no parallelism hurting the code scaling at higher thread counts. Since the experiments presented here aim to test the proposed memory placement strategy for a large number of threads working in parallel, a workaround to circumvent the need for serialization has been developed for the testing purposes. Specifically, in MFDn, the SpMV has been augmented with the code to

```
1: for $i = 1$ to $m$ do
2:     for $k =$ ptrA$(i)$ to ptrA$(i+1) - 1$ do
3:         $y^t(i) \leftarrow y^t(i) + x^t(\texttt{jA}(k)) \times \texttt{A}(k)$
4:     end for
5: end for
6: for $j = 1$ to $n$ do
7:     for $k =$ ptrAr$(j)$ to ptrAr$(j+1) - 1$ do
8:         $y(j) \leftarrow y(j) + x(\texttt{jAr}(k)) \times \texttt{Ar}(k)$
9:     end for
10: end for
```

Figure 4.6 Pseudo-code for the modified sparse matrix-vector multiplication in MFDn for the off-diagonal processors.

perform the multiplication in the CSR as well as CSC matrix formats, such that CSC is used for the computation of y^t and CSR for y, respectively. Fig. 4.6 presents the modified SpMV for an off-diagonal processor. Note that this code requires additional storage for the CSR representation of the matrix, which is reprsented by the arrays `Ar`, `jAr`, and `ptrAr`. These stand to the value, position and pointer arrays in CSR.

4.5 Experimental results and discussion

Experiments were conducted for the two test applications with the following problem and execution parameters:

CG: Class C with matrix size 150,000; 75 iterations; single MPI process.

As described earlier, CG uses an iterative process to determine the lowest eigenvalue of a sparse symmetric matrix. The benchmark comes in 5 sizes (classes): A, B, C, W(orkstation) and S(ample). Class C is the largest problem size, in terms of the size of the matrix and the number of iterations required to achieve convergence. The current implementation employs parallelization only in the form of multi-threading using OpenMP[3]. It does not involve multiple processes or any form of distributed memory communication.

MFDn: Carbon-12 (^{12}C) nucleus with the quantum oscillation number $N_{\max} = 4$ resulting in the matrix of size 1,118,926; 400 Lanczos iterations; six MPI processes (one per compute node).

[3]This implementation has been provided by the OMNI compiler group.

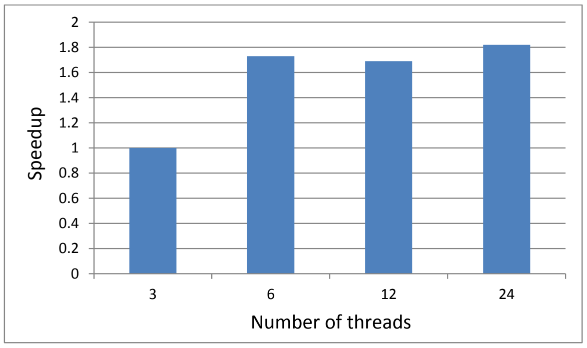


Figure 4.7 CG performance with *first touch* policy and increasing thread count.

The tests were performed on the Hopper supercomputer at NERSC. Hopper is a Cray XE6 with 6,384 compute nodes. Each compute node has a cache coherent Non-Uniform Memory Access (ccNUMA) architecture with two twelve-core AMD 'MagnyCours' 2.1 GHz processors and 32 GB of RAM. The RAM is split into 4 memory banks of 8 GB each with each group of 6 cores having a direct link to one memory bank. Thus, one NUMA node is associated with six cores. Hopper runs a SUSE Linux Enterprise Server 11 operating system and the default compiler is the Portland Group (PGI) compiler which is used in this work. All the results are reported by timing SpMV (wall-clock time) on a single compute node on Hopper. For MFDn, the maximum time is taken over all the compute nodes running off-diagonal MPI processes (Fig. 3.1) since they appear to be more compute-intensive with regard to SpMV.

First, a scaling study was conducted to observe the impact of the default *first touch* policy on the performance of the two applications when the number of threads is increased. Fig. 4.7 and 4.8 show the speed-ups obtained with varying thread numbers per node, normalized to the smallest considered number of threads, for CG and MFDn, respectively. Note that the case of one thread is skipped in exposition due to its triviality when investigating remote memory contention and latency in multi-threaded environments. A linear speed-up has been observed in SpMV when increasing the number of threads from one to three. From Fig. 4.7 and Fig. 4.8, it can be clearly observed that there is good scaling in moving from three to six threads. Beyond that, however, the scaling is erratic and poor, which may be be explained by remote access latencies and bandwidth contention.

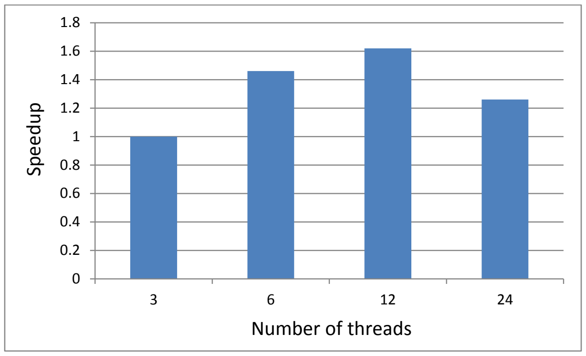


Figure 4.8 MFDn performance with *first touch* policy and increasing thread count.

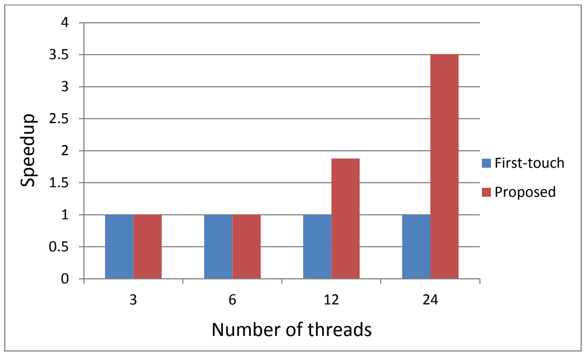


Figure 4.9 Performance gains of CG when the proposed strategy (red bars) is used. For each thread count, the result is normalized by the performance with the first-touch policy (blue bars).

Next, a performance comparison was conducted by applying the proposed memory placement strategy during the SpMV in the two applications. Fig. 4.9 and 4.10 illustrate the gains obtained with the proposed strategy compared with the default policy for CG and MFDn, respectively, while Fig. 4.11 and 4.12 present the "raw" speed-ups as calculated with respect to the lowest thread count used in the experiments.

From the comparison, it is clear that CG is benefiting from the proposed strategy to a much higher extent than MFDn. Additionally, the scaling for CG is almost ideal whereas it suffers for MFDn moving all the way up to 24 threads. The absence of parallelism in the critical section of SpMV hinders the performance of MFDn at higher thread counts. Hence, the SpMV as in Fig. 4.6 has been considered in the experiments. Fig. 4.13 and 4.14 present the obtained results for the performance gains with the proposed strategy and for the scaling, respectively. Near perfect scaling has been encountered for the modified SpMV, which proves

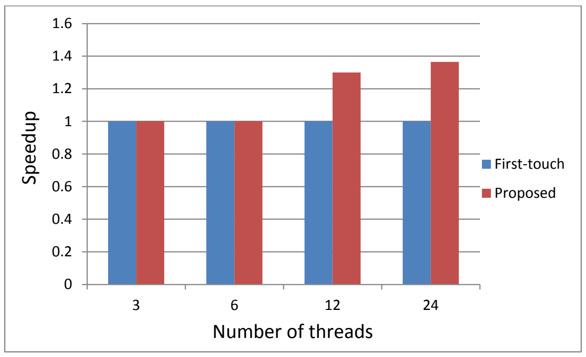


Figure 4.10 Performance gains of MFDn when the proposed strategy (red bars) is used. For each thread count, the result is normalized by the performance with the first-touch policy (blue bars).

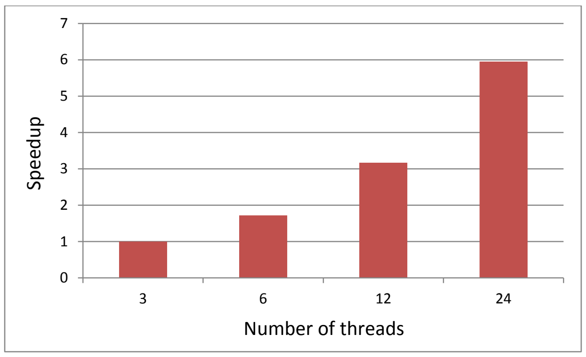


Figure 4.11 CG performance with proposed policy and increasing thread count.

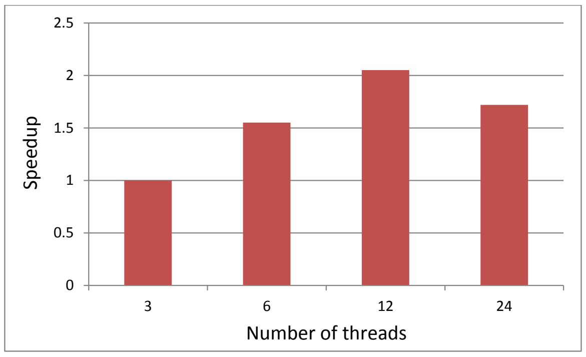


Figure 4.12 MFDn performance with proposed policy and increasing thread count.

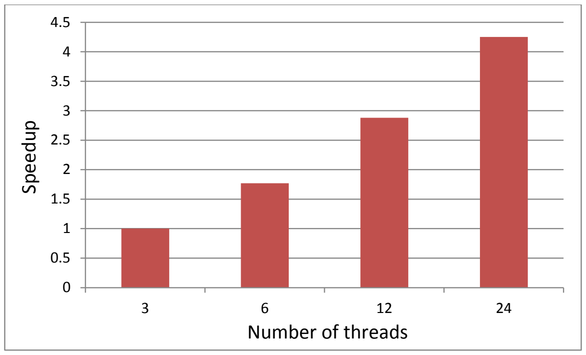


Figure 4.13 Performance gains of modified MFDn (SpMV) when the proposed strategy (red bars) is used. For each thread count, the result is normalized by the performance with the first-touch policy (blue bars).

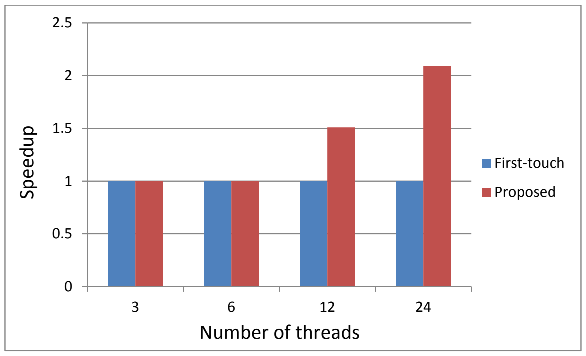


Figure 4.14 Modified MFDn (SpMV) performance with proposed policy and increasing thread count.

the effectiveness of the proposed strategies within multi-threaded applications with high degree of parallelism.

CHAPTER 5. Conclusions and future work

The main objective of this work is to make the execution of high performance scientific applications more architecture and system aware so that they are able to utilize the computational resources available to the best possible extent. To facilitate this, the applications have been augmented with certain capabilities which allow them to adapt, either statically or dynamically to the system on which they are executing, to achieve best performance. Two such capabilities are discussed in this work with respect to multi-threaded scientific computations, which have been found to have a significant impact on application performance, resulting in average speedups of two to four times.

First, the inclusion of dynamic adaptations is considered in MFDn, a large scale parallel code used for ab-initio nuclear physics calculations, by integrating it with the middleware tool NICAN. The tool monitors the system resources during the run-time of MFDn and makes a decision on the number of threads to be spawned by the multi-threaded Lanczos procedure during the iterative process. As a result, MFDn self-adapts to the dynamically changing system conditions, such as PE resource availability. Here, PE resource availability has been tested using competing applications that might execute simultaneously in the non-disjoint subsets of nodes in a cluster environment. The integration of MFDn with the middleware tool NICAN proved to be useful for facilitating MFDn adaptations in a non-intrusive manner. Adaptation decision-making strategies may be implemented in NICAN as application-specific or general-purpose modules. The proposed adaptation strategies brought about significant performance gains: more than two-fold improvement for very large problem sizes and surpassing a seven-fold improvement for the problem sizes that do not place excessive demands on the single-node PE resources. The adaptation of the number of threads available to the application to eliminate the context switches by the operating system is general. Thus, it may be used by a wide class of applications with a computationally-intensive iterative calculation.

Second, the impact of the memory affinity on multi-threaded applications is studied when executing on multi-core NUMA architectures with multiple physical memory banks. A strategy is proposed to place the shared data into specific memory banks based on the access pattern within the application. Specifically, the shared data is first categorized as being deterministically or non-deterministically accessed. Then, for the former, the chunk sizes are computed for the distribution to the memory banks local to the threads accessing the chunk. The data accessed non-deterministically, on the other hand, is to be interleaved across the memory banks on the uniform fine-grain (page) bases. A way of tailoring this general strategy to a computation has been also provided by considering sparse matrix-vector multiplication as a case study. For this purpose, two widely-used sparse matrix representations have been selected and an API proposed to employ the strategies within the multiplication code. By using this API, other multi-threaded computations that access shared data may be enhanced with the proposed strategy. The new strategy overcomes the shortcomings of the default operating system placement policy that may cause remote access latencies and bandwidth contention in NUMA architectures. Both the CG computational kernel from the NAS parallel benchmark suite and the *ab-initio* nuclear structure calculation MFDn benefited greatly from the proposed strategies. Improvements of up to 3.5 times were observed compared with the default memory placement. For any increase in the number of computational threads on a multi-core node, an almost perfect performance scaling has been achieved.

The system monitoring process employed by NICAN in the first part of the work is largely local in nature, in the sense that each node has its own PE which may be monitored independently. Hence, the proposed adaptation strategy suits well massively parallel architectures and may be employed in conjunction with global performance enhancing techniques thereby creating a multi-level adaptation. The exploration of hierarchical adaptations is left as future work. Also, with regard to the proposed memory affinity strategy, we envision that this may be expanded in the future to hierarchical NUMA architectures as they come on-board with the advent of exascale computing platforms.

BIBLIOGRAPHY

Andersen, D., Bansal, D., Curtis, D., Seshan, S., and Balakrishnan, H. (2000). System support for bandwidth management and content adaptation in internet applications. In *Proceedings of the 4th Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 213–226, Berkeley, CA, USA. USENIX Association.

Antony, J., Janes, P., and Rendell, A. (2006). Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. pages 338–352.

Apparao, P., Iyer, R., and Newell, D. (2008). Towards modeling and analysis of consolidated CMP servers. *SIGARCH Comput. Archit. News*, 36:38–45.

Baldridge, K. K., Boatz, J. A., Elbert, S. T., Gordon, M. S., Jensen, J. H., Koseki, S., Matsunaga, N., Nguyen, K. A., Su, S., Windus, T. L., Dupuis, M., Jr, J. A. M., and Schmidt, M. W. (1993). General atomic and molecular electronic structure system. *Journal of Computational Chemistry, November 1993*, pages 1347–1363.

Bellosa, F. and Steckermeier, M. (1996). The performance implications of locality information usage in shared-memory multiprocessors. *Jornal of Parallel and Distributed Computing*, 37:113–121.

Chang, F. and Karamcheti, V. (2000). Automatic configuration and run-time adaptation of distributed applications. In *High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on*, pages 11–20.

Dagnum, L. and Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55.

Forum, M. P. I. (1994). *MPI: A message-passing interface standard.*

Goglin, B. and Furmento, N. (2009). Enabling high-performance memory migration for multithreaded applications on linux. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–9.

Grant, R. E. and Afsahi, A. (2007). A comprehensive analysis of openmp applications on dual-core intel xeon smps. In *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pages 1–8.

Hollingsworth, J. and Keleher, P. (1998). Prediction and adaptation in active harmony. In *High Performance Distributed Computing, 1998. Proceedings. The Seventh International Symposium on*, pages 180–188.

Iyer, R., Wang, H., and Bhuyan, L. N. (2002). Design and analysis of static memory management policies for cc-numa multiprocessors. *Journal of Systems Architecture*, 48:59–80.

Jin, H., Jin, H., Frumkin, M., Frumkin, M., Yan, J., and Yan, J. (1999). The openmp implementation of nas parallel benchmarks and its performance. Technical report.

Kleen, A. (2005). A NUMA API for LINUX. Technical report.

Laghave, N., Sosonkina, M., Maris, P., and Vary, J. P. (2009). Benefits of parallel I/O in ab initio nuclear physics calculations. In Allen, G., Nabrzyski, J., Seidel, E., van Albada, G. D., Dongarra, J., and Sloot, P. M. A., editors, *Computational Science - ICCS 2009, 9th International Conference, Baton Rouge, LA, USA, May 25-27, 2009, Proceedings, Part I*, volume 5544 of *Lecture Notes in Computer Science*, pages 84–93. Springer.

Lameter, C. (2006). Local and Remote Memory: Memory in a Linux/NUMA System. Technical report.

Löf, H. and Holmgren, S. (2005). affinity-on-next-touch: increasing the performance of an industrial pde solver on a cc-numa system. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 387–392, New York, NY, USA. ACM.

Maris, P., Sosonkina, M., Vary, J. P., Ng, E. G., and Yang, C. (2010, ICCS 2010). Scaling of ab-initio nuclear physics calculations on multicore computer architectures. *Procedia Computer Science*, 1(1):97–106.

Maris, P., Vary, J. P., and Shirokov, A. M. (2009). Ab initio no-core full configuration calculations of light nuclei. *prc*, 79(1).

Rabenseifner, R., Hager, G., and Jost, G. (2009). Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *17th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 427–436, Los Alamitos, CA, USA. IEEE Computer Society.

Ribeiro, C., Mehaut, J.-F., Carissimi, A., Castro, M., and Fernandes, L. (2009). Memory affinity for hierarchical shared memory multiprocessors. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, pages 59–66.

Saad, Y. (2003). *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition.

Sosonkina, M. (2006). Adapting distributed scientific applications to run-time network conditions. In Dongarra, J., Madsen, K., and Wasniewski, J., editors, *Applied Parallel Computing, PARA 2004, 7th International Workshop, Lyngby, Denmark, Revised Selected Papers*, volume 3732 of *Lecture Notes in Computer Science*, pages 747–755. Springer.

Sosonkina, M., Sharda, A., Negoita, A., and Vary, J. P. (2008). Integration of ab initio nuclear physics calculations with optimization techniques. In Bubak, M., van Albada, G. D., Dongarra, J., and Sloot, P. M. A., editors, *Computational Science - ICCS 2008, 8th International Conference, Kraków, Poland, June 23-25, 2008, Proceedings, Part I*, volume 5101 of *Lecture Notes in Computer Science*, pages 833–842. Springer.

Srinivasa, A. and Sosonkina, M. (2011). Non-uniform Memory Affinity Strategy in Multi-Threaded Sparse Matrix Computations, TR no. 11-07. Technical report, Computer Science Department, Iowa State University.

Srinivasa, A., Sosonkina, M., Maris, P., and Vary, J. P. (2011). Dynamic adaptations in ab-initio nuclear physics calculations on multicore computer architectures. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1332–1339.

Sternberg, P., Ng, E. G., Yang, C., Maris, P., Vary, J. P., Sosonkina, M., and Le, H. V. (2008). Accelerating full configuration interaction calculations for nuclear structure. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA*, pages 1–12. IEEE/ACM.

Terboven, C., an Mey, D., Schmidl, D., Jin, H., and Reichstein, T. (2008). Data and thread affinity in openmp programs. In *Proceedings of the 2008 workshop on Memory access on future processors: a solved problem?*, MAW '08, pages 377–384, New York, NY, USA. ACM.

Ustemirov, N., Sosonkina, M., Gordon, M. S., and Schmidt, M. W. (2006). Dynamic algorithm selection in parallel GAMESS calculations. In *Proc. 2006 International Conference on Parallel Processing Workshops, Columbus, OH*, pages 489–496. IEEE Computer Society.

Vary, J. P. (unpublished, 1992). The many-fermion dynamics shell-model code.

Vary, J. P., Maris, P., Ng, E., Yang, C., and Sosonkina, M. (2009). Ab initio nuclear structure – the large sparse matrix eigenvalue problem. *Journal of Physics : Conference Series*, page 10.

Vary, J. P. and Zheng, D. C. (unpublished, 1994). The many-fermion dynamics shell-model code.