Runtime Address Space Computation for SDSM Systems

Jairo Balart, Marc Gonzàlez, Xavier Martorell, Eduard Ayguadé and Jesús Labarta

Barcelona Supercomputing Center (BSC),
Computer Architecture Department, Technical University of Catalunya (UPC),
Cr. Jordi Girona 1-3, Mòdul D6, 08034 – Barcelona, Spain
{jbalart,marc,xavim,eduard,jesus}@ac.upc.edu

Abstract. This paper explores the benefits and limitations of using a inspector/executor approach for Software Distributed Shared Memory (SDSM) systems. The role of the inspector is to obtain a description of the address space accessed during the execution of parallel loops. The information collected by the inspector will enable the runtime to optimize the movement of shared data that will happen during the executor phase. This paper addresses the main issues that have been considered to embed an inspector/executor model in a SDSM system: amount of data collected by the inspector, the accurateness of this data when the loop has data and/or control dependences, and the computational overhead introduced. The paper also includes a description of the SDSM system where the inspector/executor model has been embedded. The proposal is evaluated with four applications from the NAS benchmark suite. The evaluation shows that the accuracy of the inspection and the small overheads introduced by the approach allow its use in a SDSM system.

1 Introduction

Software Distributed Shared Memory (SDSM) systems has been one of the approaches proposed to provide a shared address space and overcome the programming difficulties of programming models based on message passing. Co-Array Fortran [19], Unified Parallel C (UPC) [3] or OpenMP [1] can simplify the programming of SDSM systems if the appropriate support is provided by the compiler and/or runtime system. In such systems both components are significantly stressed, and become responsible for the memory consistency and the data sharing, being these issues the most critical aspects in any SDSM system.

The inherent data movement overheads added to the overheads of this compiler/runtime support need to be minimized in order to take benefit of the potential performance of the parallel execution. On one hand, each memory access has to be monitored in order to check if it corresponds to a shared data. This memory monitoring can be performed in different ways. For instance, UPC implementations are based on the injection of runtime calls to intercept any memory access to shared data. In most SDSM implementations of OpenMP [6][8][10], the memory monitoring is done through the handling of the page fault exceptions. On the other hand, data and control communication are considered important sources of overhead. The impact of of data communication overheads can be reduced by overlapping communication and computation. Control communication is associated to the memory consistency protocol, and no matter the basis of the SDSM system implementation, it is always one of the main concerns for developers, and therefore the target of several optimization techniques [4][5][6][10].

The usual approach in most SDSM implementations is to perform both data and control communication on-demand during the parallel execution of the computation. At each page fault or memory access interception, the runtime is invoked in order to serve memory access requests and interchange the necessary control messages. Computation and communication alternate according the application requirements. The chances of the runtime system to foresee near-future data and control communication requirements are clearly limited by the amount of information available. The inspector/executor approach might play an interesting role by inspecting the set of memory addresses generated before the execution takes place and building an accurate description of them. From this information, the runtime can derive the strictly necessary data and control communication requirements and reduce the overhead associated to the memory consistency implementation. This information can be reused as long as the data access pattern has no significant changes.

This paper explores the possibility of using an inspector/executor approach in SDSM systems. The main objective is to show that applications can afford the overheads associated with building the data structures that record shared-memory memory access and computing the data distribution from the information collected in these data structures. The structure of this paper is as follows: section 2 outlines related work on the use of runtime approaches to optimize the performance of SDSM systems. Section 3 describes the main issues to consider while embedding the inspector/executor model within a SDSM system. Section 4 describes our prototype implementation that is evaluated in section 5. Finally, section 6 concludes the paper and outlines future work.

2 Related Work

This section comments some recent contributions related with data and control communication optimization in SDSM systems.

UPC implementations [2][3] perform address space monitoring through a deep coordination of the compiler and the runtime system. The compiler is in charge of detecting any suspicious memory access that might refer to shared data. Runtime calls are injected to intercept those memory accesses, and invoke the appropriate communication actions. Coalescing communication is an important source of optimization. Parallel loops are the target of the compiler, looking for statements where the set of memory references can be grouped and then served with a single communication action [4]. Beside that, the runtime tries to schedule the iterations in order to overlap the computation and the communication.

In SDSM-based OpenMP implementations [6][8][10], the address space monitoring is implemented through the pagination system. The page fault signal is intercepted to embed the communication protocol responsible for the memory consistency and data sharing. Each time a page fault takes place, the runtime system checks if the accessed page corresponds to shared data, and if necessary, takes the appropriate actions to maintain the memory consistency. Avoiding false sharing is one of the main concerns. The compiler can force particular memory alignments by inserting memory padding, which has been shown to be a reliable solution [5]. Some runtime techniques have been also proposed to modify the default assignment of work to threads in parallel loops. The runtime needs to be provided with the necessary services and structures to relate page faults (data movement) to the iterations where they occur [6]. With this information the runtime can redistribute the set of iterations in order to avoid false sharing, to minimize as much as possible the number of page movements, and to pre-send data and control messages in order to overlap computation and communication.

Regarding the data distribution, there have been some proposals that place the problem at the programming language level. For example, the ZPL [16] programming model includes several constructs and operators to specify data movements. Based on the *gather/scatter* operations, the language allows the programmer to control these operations through the content of variables, which are used as array indexes to specify the array elements to be selected within a *gather/scatter* operation.

The Co-Array Fortran [19] proposal follows the main guidelines of the traditional message-passing paradigm, but introduces considerable improvements on the data communication. Communication actions are hidden by a special treatment of the array-reference operator. This operator is overloaded and allows the specification of data distribution and remote memory accesses. Data distribution is accomplished by declaring a distributed object with extra array dimensions. The programmer controls the distribution by the shape the extra dimensions provide the object with. All memory accesses to shared and distributed data need to be expanded with particular values in the extra dimensions. The runtime derives the data location according to the defined distribution.

The introduction of the *inspector/executor* model for DSM environments was already proposed for HPF [18][19]. Our main contributions with respect those previous works are the parallel inspection process and the ability of recording the data produced by the inspector for reusing it along the different instances of the parallel code.

3 The Inspector/Executor model in SDSM systems

The aim of this section is to point out the main issues that have been considered to embed the *inspector/executor* model within a SDSM system. One of the main constraints of the *inspector/executor* model is its implicit computational overhead. Although the overhead of determining how shared data is accessed during the parallel execution may seem to be huge, we will show that for SDSM systems can be affordable. This is based on the following observations:

- It is generally accepted that in SDSM systems, unnecessary communication has much more incidence in performance than the overheads related to the execution of the runtime code. This could be summarized with something like “*better execute than communicate*”. The *inspector/executor* model follows this line.

- Most of the accesses performed in parallel codes allow the injection of a highly optimized inspector. For instance, loops represent the most common source of parallelism, and their execution usually defines a data distribution that is maintained along the whole application execution. Usually, shared data is organized as vectors or matrices, and the access pattern to those structures can be accurately described at compile time [7]. With reasonable compiler technology, it is possible to avoid the inspection of all the memory accesses at runtime, and still get an exact description of what data is referenced.
- Parallel loops are usually executed several times, giving the chance of reusing the information provided by the inspection mechanism. Therefore the execution of the inspector phase can be avoided if the data access patterns remain constant along the several instances of a parallelized loop. We are going to see that this is the most common case.
- It is possible to perform the execution of the inspector code in parallel. This is giving the runtime much space to perform the inspection without interfering with unacceptable overheads.
- One of the main limitations of the *inspector/executor* approach is the existence of control and data dependences that take part in the computation of memory addresses. This is the case when control flow statements and/or pointers appear within the body of a parallel loop. Typically, parallel loops affected with such dependences can not be treated with an optimized inspector. In the worst case, when dealing with parallel loops highly loaded with data and/or control dependences, the inspector will provide with an as much as possible accurate description of the address space used in each parallel flow. Beyond the inspector limits, the native SDSM mechanisms implementing the data sharing and memory consistency will apply. Depending on how accurate the description is, the more chances for optimizing the communications will appear, and hence, speeding up the parallel code execution.
- Finally, another important issue that needs to be considered with more detail is the amount of data that the inspector can produce, which may cause unacceptable overheads within the data distribution. This relation exists since the algorithm responsible for the data distribution totally depends on the data produced by the inspector.

All the issues comented before have conditioned the implementation of the *inspector/executor* approach that is going to be described in the next section.

4 Implementation

This section describes a specific SDSM system implementation where the *inspector/executor* model has been embedded. The implementation has been guided towards a main objective: evaluate the effectiveness of the *inspector/executor* model for SDSM systems as a source for optimization. Consequently, it has been reasonable to force the implementation to stress to the limit the inspector role, leading to a system that totally relies on the information provided by the inspection mechanisms. Therefore, the inspection process must provide the information from where to derive all the communications. For the purposes of this paper, it must be noted that all the code transformations and the generation of the inspector code have been done by hand. However, the compilation technology required by them is reasonable and should be available in any compiler.

In our implementation, computation and communication are decoupled. This forces the implementation to guarantee that shared data is available to the parallel flows prior to the execution of the parallel code. With that, we want to show that the inspector can provide with very accurate descriptions of the working sets used in each parallel flow. An immediate consequence of such approach is that three different phases can be differentiated along the parallel execution: inspection phase, communication phase and execution phase. No matter the phase, the current implementation works under a master/slave scheme, and the memory consistency protocol implements relaxed consistency.

During the inspection phase, the loop parameters (iteration space and scheduling) are broadcasted to all the slaves. Each slave computes the chunk of iterations that have been assigned to it, and the code inspection is executed. The result of the inspection consists of a list of pages that are read and/or written by each execution flow, and each slave sends this information to the master process. At this point, the communication phase starts, and the master computes the necessary page movements and which pages are written by two or more processes (conflicting pages). This computation gets as input the data produced by the inspector, and according to that, page queries are sent. Page distribution takes place, and then all processes start the parallel loop execution (execution phase). After execution, conflicting pages are treated with *diff* operations. The resulting differences are sent to the master thread. Although computation and communication could be overlapped, our current prototype implementation does not include this feature.

The current prototype is limited to loop-level parallelism. Parallel loops are specified using the OpenMP PARALLEL DO construct. Only STATIC schedules are supported with PRIVATE and SHARED data scoping clauses. REDUCTION operations have been implemented through variable expansion of the variable holding the reduction operation.

The following points describe the main aspects of the prototype implementation, according to the main issues that have been enumerated in the previous section. The code inspection process is the most critical part in the implementation so that we will try to reduce the computational overhead of the inspection process and to face the amount of data the inspection process is going to produce.

4.1 Basic inspector implementation

A simple but costly implementation can be easily achieved by intercepting any memory access in the parallel loop. For each statement in the loop body, memory accesses can be replaced by a runtime call that will record the address in internal runtime structures. It is obvious that only shared data must be monitored, so it is needed that the compiler can identify which objects are private and which are shared. This classification can be easily done by the compiler through the data scoping clauses in OpenMP. This strategy represents the simplest inspector implementation and the worst case in terms of overhead. Taking this basic approach as a baseline, several optimizations can be applied. Consider the parallel code shown in Figure 1.

```
#pragma omp parallel for
for (i=0; i<DIMX; i++) {
  for (j=0;j<DIMY;j++) {
    a[i][j] = a[i][j]*a[i][j];
    compute_row(a[i]);
  }
}
```

Figure 1: Simple parallel loop.

4.2 Amount of data produced by the inspector

A critical aspect to consider is the granularity level at which the inspector structures work. Trying to record each of the memory addresses can generate an amount of data impossible to deal with. So, it is better to work with a coarser memory unit. We propose to make the inspection at page level, being a page a continuous portion of the memory address space, similarly as in the pagination system. Even if the inspected code follows a fully predictable access pattern, the inspection mechanisms work at page level. Notice that nothing is forcing the implementation to define a uniform size for all the variables the application deals with. It might be interesting to work with smaller or bigger pages depending on the memory portion a page refers to. It is well known that particular data alignment can cause false sharing, stressing the SDSM implementation with a considerable source of control communication. Scalar variables involved in reduction operations or structured data structures (vectors, matrices) are well studied examples [5].

4.3 Parallelizing the inspector code

The inspector loop can be executed in parallel, scheduling the iterations with the same scheduled that wil be used for the loop execution. Computing the inspection of a chunk of iterations can be done applying the basic strategy described in section 4.1, but just over a subset of the whole iteration space.

Figure 2 shows the code skeleton, responsible for the inspection process. This code is executed by each parallel flow. The runtime call to *dsm_begin_for_sampling* allocates a Loop Descriptor. This subroutine forces all the threads to wait for a control message containing the loop parameters coming from the master process. The last parameter of the runtime call informs the runtime about if the information produced by the inspection can be reused in case the loop is executed several times (see section 4.6). For this example, nothing forbids to do so. The *while* statement makes the executing thread to be continuously asking for iterations to the runtime system until all the loop iterations have been executed. In the current implementation, only STATIC scheduling is supported, thus the call to *dsm_next_iters_sampling* runtime service updates the variables start and end only once, defining the chunk of iterations to execute.

```
int a[DIMX][DIMY];
int low,upper,step;
int start,end;
int i,j;
dsm_begin_for_sampling(&low,&upper,&step,1);
while (dsm_next_iters_sampling (&start,&end))
{
  for (i=start;i<=end;i+=1)
    for (j=;j<=DIMY;j+=1)
    {
      stmt_sample(&a[i][j],1,& a[i][j]);
      insp_compute_row(a[i]);
    }
}
dsm_end_for_sampling ();
```

Figure 2: Inspection code for parallelized loop.

4.4 Predictable access patterns

Even if the code inspection is done in parallel, it is necessary to look for more chances for optimization. Statements with invariant memory addresses can be omitted in the inspection process for all iterations, and treated just once. Predictable memory addresses, such as linear accesses to vectors or multidimensional matrices, can be managed with a single runtime service, summarizing the memory portion accessed by each execution flow. Figure 3 shows an optimized version of the inspecting code. Notice that interprocedural analysis phase is required to detect that the call to *compute_row* subroutine is invariant across the *j-loop* iterations. For similar cases where the inspection process can be optimized, the data produced by the inspector is organized at page level, as it has previously mentioned in section 4.2.

```
int a[DIMX][DIMY];
int low,upper,step;
int start,end;
int i,j;
dsm_begin_for_sampling(low,upper,step,1);
while (next_iters_sampling (&start,&end))
{
 for (i=start;i<=end;i+=1)
 {
  sample_region(&a[i][0],DIMY,1,&a[i][0],DIMY);
  insp_compute_row(a[i]);
 }
}
dsm_end_for_sampling ();
```

Figure 3: Optimized inspecting loop code.

4.5 Pointers and control dependences

Pointers and control dependences represent a considerable limitation to the *inspector/executor* model. Current implementation does not include any specific support for dealing with pointers. The case of index vectors is treated with the most conservative approach, which forces the inspector to assume that the variable accessed through an index vector will be totally referenced. In terms of communication, this is going to be translated to a broadcast operation of the variable. In case pointers appear to be invariant along the parallel loop execution, the inspector still can be executed with no limitation. Under any other circumstance, the inspection is inhibited.

Control dependences also limit the inspection process. When a control flow statement breaks the sequential execution, the inspector cannot always know which branch will be executed. If private data determines the branch, the *inspector* can include all the necessary operations to evaluate the control dependence. If not, a conservative approach is taken and the *inspector* inspects all the possible branches.

Although the current support to overcome the limitations related with pointers and control dependences is very small, this is not going to have a significant impact on the inspector functionality. It is quite common that parallel loops show a particular ratio between the amount of data and operations related to memory addresses computation and the total loop computation. Usually, parallel loops present a small percentage of data and operations related to memory addresses computations. Under such situation, the inspector code can still be applied, and the most conservative solutions that have been described are not going to suppose a significant loose of accurateness or an unacceptable increment of overhead.

4.6 Reuse of the inspector data

It is clear that having the possibility of reusing the *inspector* data becomes an important source of optimization. Detecting if this data can be reused along the different instances of a parallelized loop is not a simple task and the necessary compiler and runtime support to automate such issue is not available in the prototype. So, the current implementation is based on information provided by the programmer to specify if the inspector data can be reused. We have analyzed each parallelized loop and determined for each one, if data reuse was possible to be applied. In the evaluation section, the number of loops with reused inspector is discussed, as well as the impact of the reuse in performance.

5 Evaluation

The aim of this section is to describe and measure the limits on the *inspector/executor* model in SDSM systems. Hence, not the whole SDSM implementation is evaluated, just the effects of the inspection and data distribution mechanisms. Speedup and execution time numbers are the initial metrics for the evaluation process, but then broken down in different parts: communication associated to application itself, communication required by the runtime, computation time of the application code and computation time inside the runtime. The effects of the inspection process are mostly noticeable within two implementation mechanisms: the inspection execution and the algorithm responsible for deriving the data communication. Therefore, these two aspects are specifically measured. No comparison of the current prototype with other systems has been included. The main reason for that, is that the evaluation is centered around the effects of the inspection process and the accurateness of the data produced. In that direction, for all the tested applications, two versions of the inspector code have been considered: a non optimized and an optimized version. For each case, the optimizations are described.

The evaluation has been done using four applications from the NAS parallel benchmark suite: EP, IS, FT and CG, all of them in their C version [9][10]. The experiments have been performed in the *Marenostrum* [15] platform available at the Barcelona Supercomputing Center (BSC). The machine is composed by 2406 dual nodes based on PowerPC970FX, 2.2 GHz and Myrinet with a total amount of 9.6 TB of memory. A subset of 8 nodes was used for the evaluation.

5.1 EP

The *Embarrassingly Parallel* benchmark computes pairs of Gaussian random deviates, according to a specific scheme. The benchmark works mainly with private data and performs a reduction operation over two global variables. The whole computation is organized as a single loop executed just once. This benchmark allows for measuring the impact of the inspection process, conditioned by three issues. First, no reuse can be applied, as the computation takes place only once. Second, the inspection process has to deal with a considerable amount of private computation, needed to point out what private data has to be accessed in the reduction operations. Two versions of the inspection process can be studied, one including the private computations, the other not. Finally, negligible data communication is about to happen, since shared data is only composed by two objects, the global variables where the output of the reduction operations are stored.

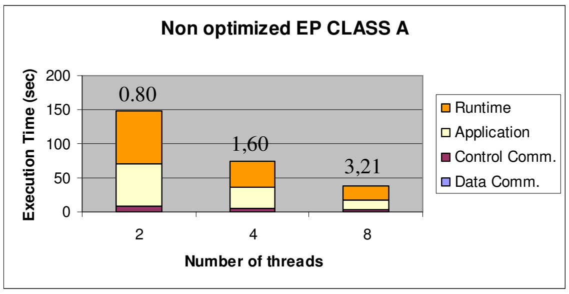

Figure 4: Non optimized EP CLASS A

Figure 4 shows the performance obtained in the execution of the EP (class A) benchmark, with 2, 4 and 8 threads and non optimized inspection. The numbers on top of the columns correspond to the speedup obtained in each experiment. The *Y* axis shows the execution time, which is broken down (top to bottom) in Runtime and Application code execution, and Data and Control communication. The serial time is 119,39 seconds and corresponds to the unmodified benchmark executed sequentially. The Runtime and Application code take near 93% of the execution time. The cost of the inspection process is included in the Runtime measurements and represents about 51% of total execution time. This behavior is maintained with 2, 4 and 8 threads, and suggests there is much space for optimization. The inspection process is too heavy and represents about having to execute twice the benchmark computation. The reason of such overhead is that all computations related to private data are inspected. Notwithstanding, some speedup is observed (3.21 with 8 threads).

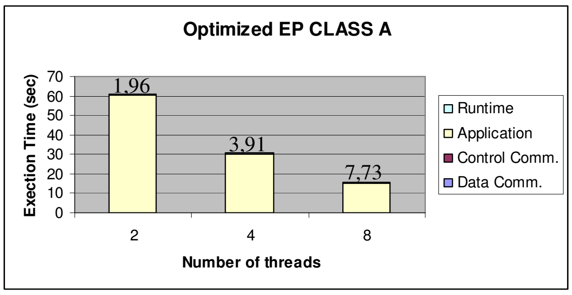

Figure 5: Optimized EP CLASS A.

Figure 5 shows the performance for the optimized inspection process. In this case, private computations have been taken out from the inspection code. This process could be easily done by means of the PRIVATE clause in the parallelism specification. Clearly, the benchmark performance is now improved, obtaining speedups of 1.96, 3.91, and 7.73. The Runtime execution time ranges from 0.17% (2 threads) to 1.16% (8 threads). The inspection process and the computation of the data distribution represent about 1.72% and 1.22% over the total execution time.

These results show that with a simple compiler optimization (avoiding the inspection of private data), the process can be implemented without noticeable overhead. In addition, the accuracy of the data produced by the *inspector* is enough to totally determine the data distribution in this simple benchmark.

5.2 IS

The Integer Sort benchmark works with a shared vector, uniformly distributed among all parallel processes. The computation is organized in a single parallel loop, executed several times. After each loop instance, a reduction operation is performed. That forces the parallel flows to flush some data back to the master process. The output of the inspection process can be reused along the benchmark execution, so it is only computed once. Two versions of the inspection process can be implemented: a non optimized inspection, which goes along the iteration space and records all memory accesses; and, an optimized version, where the inspection is done through a single runtime call, summarizing the access pattern to the shared vector.

Figure 6 shows the performance for the IS (class B) execution, with 2, 4 and 8 threads, and non optimized inspection. The serial time is 46.0 seconds. For the non optimized version each memory reference to a shared variable is intercepted. The execution of the application code scales with the number of threads, but not the execution of the runtime system. Data communication also increases with the number of threads. This is caused by an *all-to-one* communication pattern related to the reduction operation, previously mentioned. Notice that Control communication represents a very small percentage (0.01%, 0.02% and 0.03 with 2, 4, and 8 threads) of all the communication. This is caused because the *inspector* provides the runtime system with all the necessary information regarding the memory consistency (conflicting pages, written by more than one thread).

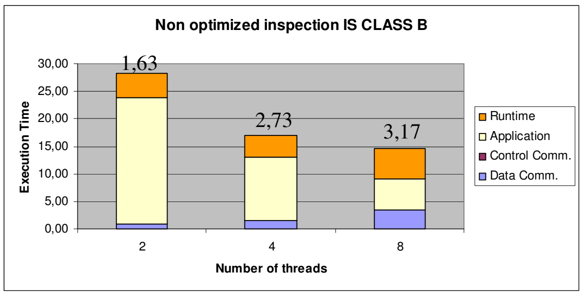

Figure 6: Non optimized IS CLASS B.

For the optimized version, predictable access patterns are assumed to be detected by the compiler. Linear memory accesses to shared vectors have been inspected through a single runtime call describing the access to the vector. The results in Figure 7 show the reduction of the execution time spent in the runtime system.

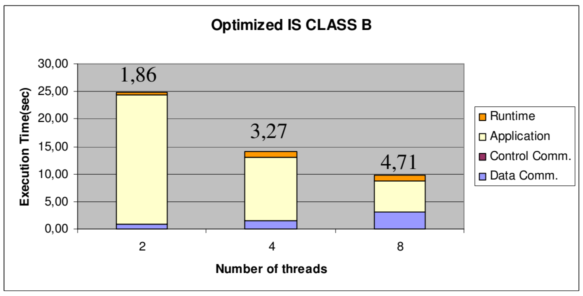

Figure 7: Optimized IS CLASS B.

5.3 FT

The *Fourier Transformation* benchmark computes a Fourier transformation over a three dimensional matrix. The computation is organized in four subroutines: *evolve,* *cffts1, cffts2* and *cffts3*. All execute one after the other and update the content of the main structure, the three dimensional matrix. This is repeated several times, depending on the input benchmark. Each subroutine implements the computation with three nested loops, one per dimension on the working set. While *evolve*, *cffts1* and *cffts2* distribute the data cross the same dimension, the computation in *cffts3* completely changes the data distribution. This causes this benchmark to be highly loaded with *Data* communication overhead. The output of the inspection process in each subroutine can be reused except for *evolve*, *cffts1* and *cffts3*, so for these subroutines, the code inspection is performed each time they are executed. Again, two versions of the inspection process have been tested.

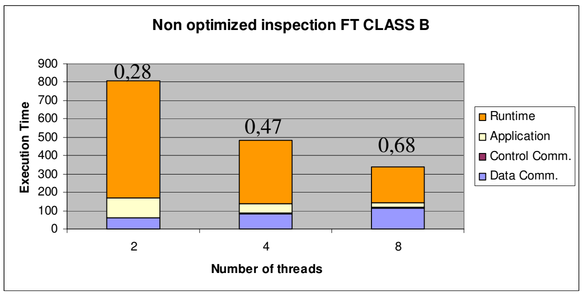

Figure 8: Non optimized FT CLASS B.

Figure 8 shows the performance of the non optimized version. The serial time is 232.33 seconds. Clearly, the unacceptable overhead produced by the inspection is preventing any chance for speeding up the execution. Although *Data* communication represents 7.64%, 17.09% and 33.72% of overhead, the weight for the inspection process (76.32%, 63.79% and 45.41%) is the main factor that degrades the performance. The inspection overhead comes out because of the structure of the inspected code: the nest of three loops. Running over the whole iteration space sinks any possibility of taking profit of the information gathered during the inspection process.

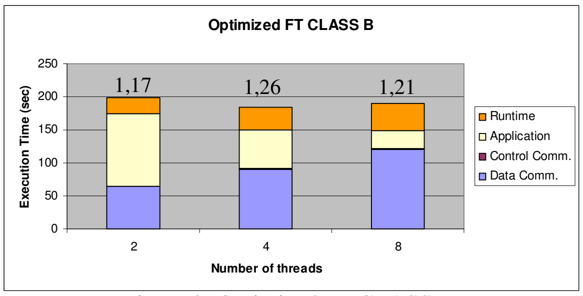

Figure 9: Optimized FT CLASS B.

Figure 9 shows the results for the optimized case. Although this version obtains very poor speedup, just 1.17, 1.26 and 1.22 for 2, 4 and 8 threads, now the time spent under the runtime execution is about 11.79, 18.33 and 21.82. If those percentages are broken down, we see that the inspection process is about 1.08%, 0.6% and 0.3% of the overall execution time. Therefore, the influence of the inspection process is not the point. Those overheads are related to *diff* operations needed for the memory consistency protocol. Anyway, Data communication becomes critical as it represents 32.26%, 49.22% and 63.20% of total execution time.

Notice that the data movement is totally determined by the application. The overhead contributions coming from the inspection process and the data distribution algorithm are negligible in front of Data communication times.

5.4 CG

The CG NAS parallel benchmark computes an approximation to the smallest eigenvalue of a large, sparse, symmetric, positive definite matrix using a conjugate gradient method. As in previous codes, two versions of the application have been evaluated. In the non-optimized version (in which each memory access to shared data is intercepted), the overheads related to the inspection process and the computation of the data distribution can be afforded by the application when running up to 4 threads (speedups of 1.87 and 2.89). With 8 threads the grain size assigned to each process becomes too small to be worth for parallel execution, compared to the amount of data that needs to be communicated. Figure 10 shows the results for this version.

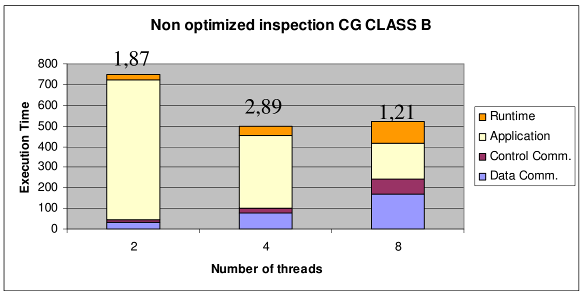

Figure 10: Non optimized CG CLASS B

For the optimized version, similar speedups are obtained. The overheads related to the inspection are reduced when running with 2 and 4 threads, but not with 8 threads. This is not translated to an increase of speedup because the accuracy in the data produced by the inspector is not very high. The performed optimizations are based on broadcast operations of shared data referenced through index vectors. The inspector assumes the whole data structure is used by all the threads. This causes an increment of the overhead related to the computation of the data distribution, as this mechanism depends on the output of the inspection process, in terms of the number of pages involved in the data distribution. Figure 11 shows the results for the optimized version.

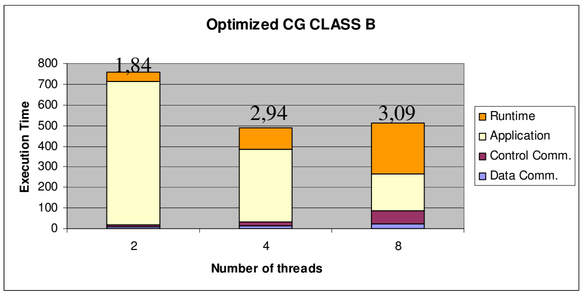

Figure 11: Optimized CG CLASS B

6 Conclusions

This paper shows benefits and limitations of the *inspector/executor* model within a SDSM system. The role of the *inspector* is to provide an accurate (as mush as possible) description of the references to shared data in each processor during the parallel execution. It has been proved that delivering this information to the runtime system creates many chances for optimizing the communication. The limits of the model are defined by the overheads, implicit to the basis model, but can be overcome by several optimization techniques, smoothing the impact of the inspection process on the overall execution time.

Our experiments with four benchmarks of the NAS parallel benchmark suite have demonstrated that it is possible to generate very accurate inspectors. It is possible to build on top of the *inspector/executor* model a SDMS implementation, and execute the parallel code performing the strictly necessary communication.

Acknowledgements

This work has been supported by the Ministry of Education of Spain under contract CICYT-TIN2004-07739-C02-01, and the Barcelona Supercomputing Center.

References

[1] OpenMP Application Program Interface, Version 2.5, May 2005, www.openmp.org.
[2] T. El-Ghazawi and F. Cantonet. “UPC Performance and Potential: a NPB experimental Study”. Proceedings of the 2002 ACM/IEEE International Conference on Supercomputing (ICS). 2002.
[3] T. El-Ghazawi, W. Carlson and J. Drapper. “UPC Language Specifications V1.1.1”. Oct 2003.
[4] W. Yu Chen, C. Iancu and K. Yelick. “Communication Optimizations for Fine-grained UPC Applications”, Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques, (PACT2005)
[5] H. Lu, A. L. Cox, S. D. R. Rajamony and W. Zwaenepoel. “Compiler and Software Distributed Shared Memory Support for Irregular Applications”. Proceedings of the Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), 1997.

[6] J.J Costa, T. Cortés, X. Martorell, E. Ayguadé and J. Labarta. “Running OpenMP applications efficiently on an everything-shared SDSM”. Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS). Santa Fe, New Mexico, USA. 2004.

[7] A. Basumallik, R. Eigemann. “Towards Automatic Translation of OpenMP to MPI”. Proceedings of the 19th Annual International Conference on Supercomputing (ICS). Cambridge, Massachusetts, USA. 2005.

[8] Y. Charlie Hu, Honghui Lu, Alan L. Cox, and Willy Zwaenepoel. “OpenMP for Networks of SMPs”. Journal of Parallel and Distributed Computing, vol. 60 (12), pp. 1512-1530, December 2000.

[9] M. Sato, S. Satoh, K. Kusano and Y. Tanaka. “Design of OpenMP Compiler for SMP Cluster”. Proceedings of the 1st European Workshop on OpenMP (EWOMP). 1999.

[10] M. Sato, H. Harada and Y. Ishikawa. “OpenMP Compiler for a Software Distributed Shared Memory System SCASH”. Proceedings of the 1st Workshop on OpenMP Applications and Tools (WOMPAT). 2000.

[11] NanosMercurium Compiler Infrastructure, www.cepba.upc.es/mercurium.

[12] S. D. Sharma, R. Ponnusamy B. Moon, Y. Hwang, R. Dasand J. Saltz. “Runtime and compile-time support for adaptive irregular problems”. Proceedings of Supercomputing’94, 1994.

[13] H. Jin M. Frumkin, and J. Yan. “The OpenMP Implementation of the NAS Parallel Benchmarks and its Performance”. Technical Report NAS-99-011, NASA Ames Research Center, October 1999.

[14] C. Koelbel and P. Mehrotra. “Compiling Global Name-Space Parallel Loops for Distributed Execution”. IEEE Transactions on Parallel and Distributed Systems, vol. 2(4), pp. 440–451, October 1991.

[15] Barcelona Supercomputing Center, www.bsc.es.

[16] S. J. Deitz, B. L. Chamberlain, S-Eun Choi, and L. Snyder. "The Design and Implementation of a Parallel Array Operator for the Arbitrary Remapping of Data",.Proceedings of the ACM SIGPLAN symposium on principles and practice of parallel programming (PPoPP 2003)

[17] C. Koelbel and P. Mehrotra. “Supporting shared data structures on distributed memory architectures”. Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP). 1990.

[18] R.V. Hanxleden, K. Kennedy, C. Koelbel, R. Das, and J. Saltz. “Compiler analysis for irregular problems in Fortran-D”. Proceedings of the 5th Workshop on Languages and Compilers for Parallel Computing (LCPC), 1992.

[19] Y. Dotsenko, C. Coarfa, J. Mellor-Crummey. “A Multiplatform CoArray Fortran Compiler”. Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT), 2004.