J Supercomput manuscript No.
(will be inserted by the editor)

A cost-effective heuristic to schedule local and remote memory in cluster computers

Mónica Serrano · Julio Sahuquillo
Salvador Petit · Houcine Hassan · José Duato

the date of receipt and acceptance should be inserted later

Abstract Cluster computers represent a cost-effective alternative solution to supercomputers. In these systems, it is common to constrain the memory address space of a given processor to the local motherboard. Constraining the system in this way is much cheaper than using a full-fledged shared memory implementation among motherboards. However, memory usage among motherboards can be unfairly balanced.

On the other hand, remote memory access (RMA) hardware provides fast interconnects among the motherboards of a cluster. RMA devices can be used to access remote RAM memory from a local motherboard. This work focuses on this capability in order to achieve a better global use of the total RAM memory in the system. More precisely, the address space of local applications is extended to remote motherboards and is used to access remote RAM memory.

This paper presents an ideal memory scheduling algorithm and proposes a cost-effective heuristic to allocate local and remote memory among local applications. Compared to the devised ideal algorithm, the heuristic obtains the same (or closely resembling) results while largely reducing the computational cost. In addition, we analyze the impact on the performance of stand alone applications varying the memory distribution among regions (local, local to board, and remote). Then, this study is extended to any number of concurrent applications. Experimental results show that a QoS parameter is needed in order to avoid unacceptable performance degradation.

Keywords cluster computers · memory scheduling · remote memory assignment · quality of service · analysis of performance

Mónica Serrano · Julio Sahuquillo · Salvador Petit · Houcine Hassan · José Duato

Department of Computer Engineering (DISCA)
Universidad Politécnica de Valencia
Camino de Vera s/n, 46022 Valencia, Spain
Tel.: +34-677415807
Fax: +34-963877579
E-mail: spetit@disca.upv.es

1 Introduction

Cluster computers consist of a set of interconnected motherboards, each one hosting a given number of processor cores, which is expected to dramatically grow with future technologies. They are an alternative solution to high-end supercomputer systems, since they offer a good trade-off between cost and performance. For this reason, cluster computers currently represent an important segment of the market and they are used for a wide range of applications such as high-performance parallel computing, e-business or grid computing applications.

Nowadays, large computing intensive parallel applications run in cluster computers competing with heterogeneous workloads and services, which are often virtualized and executed on-demand. This can lead to situations where the usage of one or some computer resources becomes widely unbalanced among motherboards. For instance, processor or memory may become underused in some motherboards of the cluster, while there is a need of computational power or storage capacity in others. Regarding memory usage, this drawback is not found in shared memory systems, which allow applications to potentially use all the available memory in the whole system. However, supporting shared memory presents scalability constraints due to the overhead and costs incurred by the memory coherence protocol, which must act both intra and inter-motherboards.

In contrast, highly scalable systems like the BlueGene/P (which can be scaled up to 884,736-processors [1]) avoid memory coherence protocols by providing support to message passing mechanisms. In addition, to reduce the message communication latency, high-end systems like BlueGene/L [2], BlueGene/P [3], or Cray XT [4] implement Remote Memory Access (RMA) [5]. RMA allows a given board to directly access the memory of another board. Unlike data transfers performed by typical memory coherence protocols, these are explicitly managed by the applications. RMA is currently a feature of modern high-end systems, although it is expected to find commodity implementations for cluster computers in the near future. Message passing provides better scalability when the memory requirements of a given application exceed the available memory in the motherboard, although performance can be still hurt.

An alternative approach is the use of RMA to increase the available memory beyond the local motherboard, while still providing cache coherence support within the motherboard. To this end, the use of a fast interconnection network among motherboards is a critical design issue. This work focuses on a cluster prototype that uses the HyperTransport technology to interconnect motherboards. The HyperTransport consortium [6] has more than 60 members from leading industry (AMD, HP, Dell, IBM, etc.) and universities, and it is expected that systems like the studied in this paper will become widely used in the near future. In our system, the RMA device and the OS running in the motherboards support an inter-motherboard memory allocation system that assigns portions of remote memory to local applications. In such a system, memory assigned to applications can be located in three main regions, i) in the same node as the processor running the application (L), ii) in another node of the local motherboard (Lb), and iii) in a remote motherboard (R). As these regions have different latencies, performance of a given application strongly depends on how the assigned memory is distributed among regions. Therefore, like in any NUMA system, memory management is a critical performance concern.

In a previous work [7] we analyzed how the memory distribution (i.e., L, Lb and R) impacts on the performance of applications with different memory requirements, and presented an ideal memory allocation algorithm, referred to as SPP, that distributes the memory space among applications in order to improve the system performance. In addition, the quality of service (QoS) of each application was considered to guarantee reasonable performance. This paper extends that work in two main ways. First, we provide a much deeper study of the SPP algorithm by generalizing the analysis to any number of applications and by including more working examples. Then, a new efficient heuristic algorithm is proposed. Compared to the SPP algorithm, the heuristic obtains the same (or closely resembling) results while reducing the computational cost of allocating memory to n applications in a factor of $(n-1)!$

The remaining of this paper is organized as follows. Section 2 describes the system prototype and the model assumed in this work. Section 3 evaluates how the distribution of memory assignment impacts on the performance of single applications while Section 4 studies the impact on concurrent applications. Section 5 details the proposed memory allocation algorithms. Section 6 discusses previous research related to this work, and finally, Section 7 presents some concluding remarks.

2 System prototype and model

A cluster machine with the required hardware/software capabilities is being prototyped in conjunction with researchers from the University of Heidelberg [5], which have designed the RMA connection cards. The machine consists of 64 motherboards each one including 4 quad-core 2.0GHz Opteron processors in a 4-node NUMA system (1 processor per node), and a 16GB RAM memory per motherboard. The connection to remote motherboards is implemented by a regular HyperTransport [8] interface to the local motherboard and a High Node Count HyperTransport [9] interface to the remote boards. This interface is attached to the motherboard by means of HTX compatible cards [10].

When a processor issues a load or store instruction, the memory operation is forwarded to the memory controller of the node handling that memory address. The RMA connection cards include their own controller, which handles the accesses to remote memory. Unlike typical memory controllers, the RMA controller has no memory banks directly connected to it. Instead, it relies on the banks installed in remote nodes. This controller can be reconfigured so that memory accesses to a given memory address are forwarded to the selected remote motherboard.

Since the prototype is still under construction, in order to carry out the experiments and explore the performance of the proposed memory scheduler algorithms, the cluster machine has been modeled using the Simics 3.0.31 simulation framework [11] with GEMS 2.1 [12]. The devised algorithms have been implemented by modifying the GEMS Ruby submodule, which simulates the details of the memory system. In addition, the whole system has been scaled down to have reasonable simulation times.

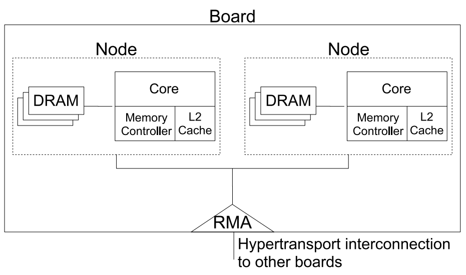


Fig. 1 Block diagram of the 2-node NUMA system model and RMA

Table 1 Memory subsystem characteristics

Characteristic	Description
# of processors	2 per motherboard
L1 cache: size, #ways, line size	64KB, 2, 64B
L1 cache latency	3
L2 cache: size, #ways, line size	1MB, 16, 64B
L2 cache latency	6
Memory address space	512MB, 256MB per motherboard
L Latency	100
Lb Latency	157
R Latency	1500

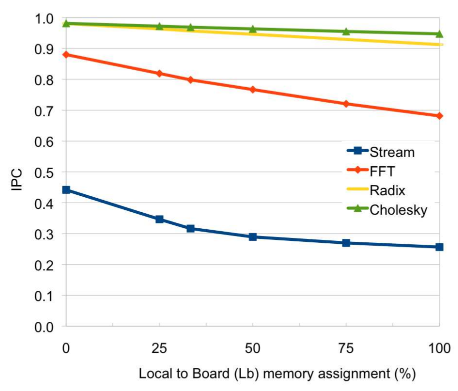
The system model consists of two motherboards, each one composed of a 2-node NUMA system as shown in Figure 1. Each node includes a processor with private caches, its memory controller and the associated RAM memory.

Table 1 shows the memory subsystem characteristics, where memory latencies and cache organizations resemble those of the real prototype. The RMA connection cards have been assumed with no internal storage capacity. Likewise, the AMD Hammer coherence protocol has been extended to model the RMA functionality.

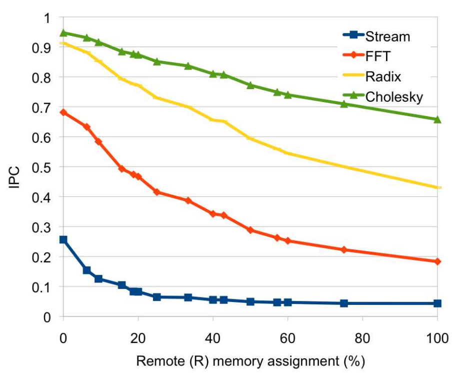
In this paper, we assume that the distribution of local and remote memory assigned to an application is set statically by interleaving the memory addresses at cache block size level (64B). Consequently, the memory controllers are configured when the system boots to support a given distribution.

3 Analysis and impact on performance of memory distribution

This section analyzes the impact on performance when varying the memory distribution across the three memory regions (L, Lb, R). Four benchmarks have been used to carry out the experiments: Stream [13] and three kernels (Radix, FFT and Cholesky) from the SPLASH-2 benchmark suite [14]. Stream is a benchmark designed to stress the memory hierarchy,



(a) Impact on IPC when assigning L and Lb



(b) Impact on IPC when assigning R

Fig. 2 Impact on performance (IPC) when assigning local and remote memory

Table 2 Impact on IPC for different memory distributions

Memory Distribution				IPC			
L(%)	**Lb(%)**	**(L+Lb)(%)**	**R(%)**	**Stream**	**FFT**	**Radix**	**Cholesky**
50	25	75	25	0,06	0,42	0,73	0,85
25	50			0,06	0,40	0,72	0,85
33,3	33,3	66,7	33,3	0,06	0,39	0,71	0,84
66,7	0			0,06	0,39	0,70	0,84
50	0	50	50	0,05	0,29	0,60	0,78
33,3	16,7			0,05	0,29	0,59	0,77
25	0	25	75	0,04	0,22	0,50	0,71
0	25			0,04	0,22	0,50	0,71

while the selected SPLASH-2 kernels have been chosen because they perform the highest number of memory accesses of the benchmark suite.

First, we study the case that only the RAM installed in the local board (i.e., L and Lb) is allocated for an application. The performance (i.e., IPC) has been analyzed varying the percentage of Lb with respect to the total assignment (i.e., $L+Lb$). Figure 2 (a) shows the results. As only L and Lb modules are assigned, it is enough to represent the Lb percentage in the X axis. For instance, a value of 75 in the X axis corresponds to an assignment of $Lb = 75\%$ and $L = 25\%$. Notice that the distribution of local memory assignment may have a strong impact on performance in some benchmarks while others are slightly affected. Stream is the most sensitive benchmark since its IPC degrades about 42% when all its memory is assigned to the Lb memory region, FFT performance degrades about 27%, and Radix and Cholesky performance hardly degrades.

In the second scenario, the impact of allocating remote memory is explored. Table 2 shows the performance results for eight different memory distributions. Notice that the performance slightly differs when varying the memory distribution of L and Lb while maintaining their accumulated value (dark column) constant. This means that the execution time is dominated by the much slower remote memory. For instance, for $L+Lb = 75\%$, Stream gets the same IPC (0.06) in both cases. This is because remote memory has a latency about one order of magnitude higher than local memory. From these results, we can conclude that, when assigning remote modules, the memory distribution within the board (L and Lb) has a negligible impact on performance. Consequently, to study the effect of remote memory, a single value is enough to represent both local memory regions (i.e., $L+Lb$).

Figure 2 (b) shows the adverse impact on performance as the percentage of assigned remote memory grows with respect to the total memory assignment (i.e., $L+Lb+R$). The initial points in the Y axis correspond to the lowest performance of each application in Fig. 2(a). Values in the X axis represent the percentage of remote memory assignment. For instance, $X = 25$ means that $R = 25\%$ and $L+Lb = 75\%$. Three different performance behaviors can be appreciated. Stream performance dramatically degrades with the assigned remote memory until approximately 25%, where it stabilizes. On the other hand, performance of both Radix and Cholesky constantly decreases in almost a linear way. Finally, FFT behavior falls in between these two trends. Its performance strongly drops until $R = 60\%$ and then it smoothly decreases.

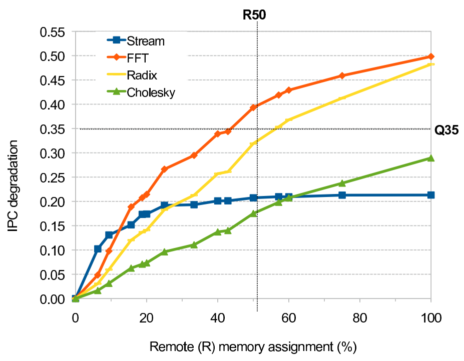


Fig. 3 IPC degradation as a function of remote memory

4 Concurrent execution of several applications

When running several applications, it may happen that the memory scheduler allocates too much remote memory to a given application, yielding to unacceptable performance for that application. This section first presents a Quality of Service (QoS) parameter to deal with such situations. This parameter specifies the maximum acceptable performance degradation for an application.

4.1 Quality of service definition

Figure 3 shows the IPC degradation calculated as $IPC(R=0\%)-IPC(R=X\%)$. The origin point means that no remote memory is assigned so there is no performance degradation. Notice that since performance always degrades as R increases, this figure can be used to define the maximum percentage of remote memory allocated for an application to guarantee a performance value. For instance, if the maximum IPC degradation permitted for FFT is 0.35, the scheduler will not assign more than 40% of remote memory to this benchmark (see label Q35). On the other hand, if the percentage of remote memory allocated for Cholesky is less than 50%, then its IPC degradation will be below 0.18 (see label R50). Therefore, there is a bijective relationship between performance degradation and percentage of assigned remote memory. Due to this equivalence, from now on, the maximum percentage of remote memory that is permitted to a given application will be referred to as its QoS, and this value will be used by the devised memory schedulers for controlling the system performance.

Table 3 Studied memory distributions

Application	Case A (L+Lb)	R	Case B (L+Lb)	R	Case C (L+Lb)	R	Case D (L+Lb)	R	Case E (L+Lb)	R
A1	100%	0%	75%	25%	50%	50%	25%	75%	0%	100%
A2	0%	100%	25%	75%	50%	50%	75%	25%	100%	0%

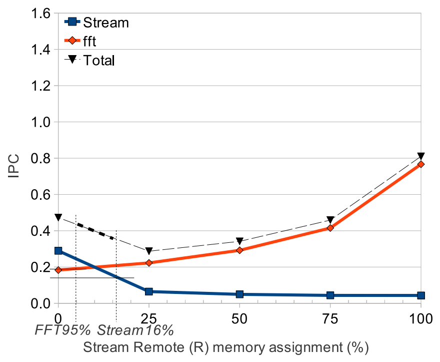
4.2 Two concurrent applications

Once the QoS has been defined, we analyze the impact of the memory distribution on the performance of several concurrent applications. In this section, the study focus on two applications. Two cases are analyzed: in the first one, the whole remote memory is allocated, and in the second one, only a fraction.

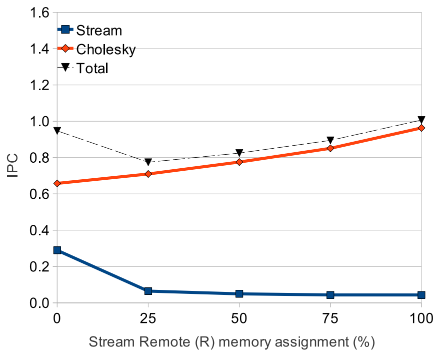
4.2.1 Complete usage of remote memory

This study assumes that all the memory installed in two motherboards (local and remote) is used by two applications and that there is enough memory to support their whole working set. Five memory distributions have been evaluated varying the percentage of remote memory assigned to each application from 0% to 100% in fractions of 25%, as shown in Table 3. For instance, case B means that the memory assignment for application A1 is R=25% and L+Lb=75% while the application A2 allocates R=75% and L+Lb=25%.

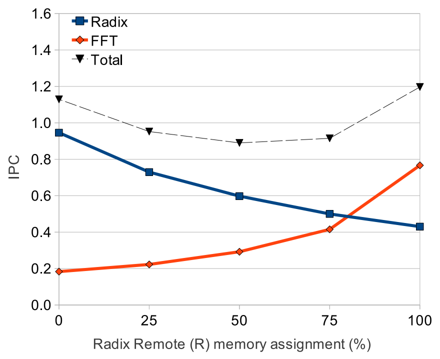
Using these distributions, the performance of each application as well as the combined performance are shown in Figure 4. Since remote memory is shared between both applications, if one of them uses $R = X\%$, the other one shall use $R = 100\% - X\%$. For instance, in



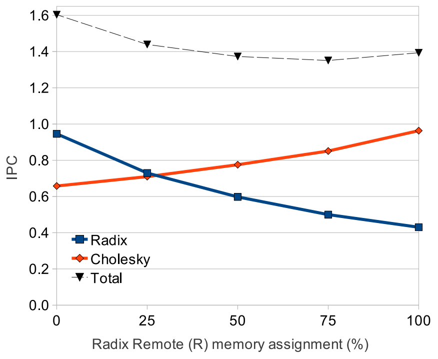
(a) Stream vs FFT



(b) Stream vs Cholesky



(c) Radix vs FFT



(d) Radix vs Cholesky

Fig. 4 Overall IPC for different couples of applications

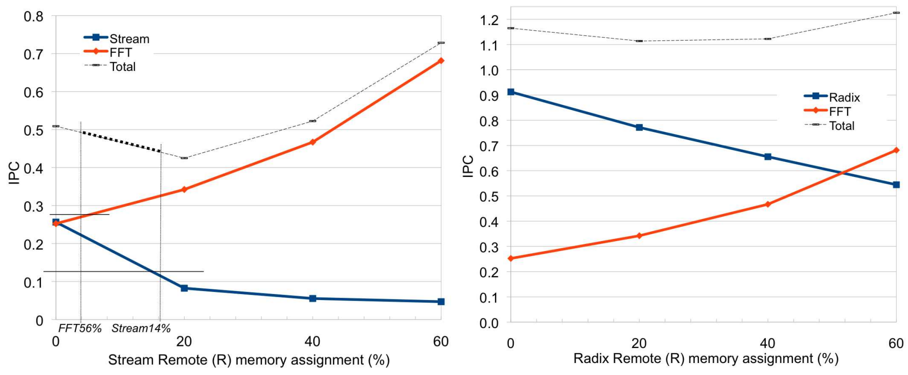
Figure 4 (a) if Stream consumes $R = 25\%$ then FFT will use the remaining $R = 75\%$. The dashed line stands for the total system performance. The highest point represents the maximum performance achieved by both workloads; however, reaching this point might imply poor and unacceptable performance for some applications. This situation can be controlled by defining a QoS for each application, as stated above. For instance, if Stream QoS is set to 16% and FFT is set to 95% (see 4 (a)), the scheduler will select the distribution corresponding to the aggregated $IPC = 0.45$, since it is the best performance falling in the interval [0.35, 0.45]. Consequently, the IPC of Stream will be twice as large as its worst case (0.10 versus 0.05).

Notice that as the distribution of R changes, the performance of one application increases while the performance of the other drops. Since this fact happens at different rates, the maximum performance is reached when the application that decreases performance at the highest rate (i.e., the most sensitive) only accesses local memory. This application can be easily identified by looking its IPC degradation value for $R = 100$ in Figure 3. For example, if the co-running applications are Stream and FFT, their IPC values will be 0.21 and 0.50, respectively. Thus, the maximum aggregated IPC is obtained when all the remote memory is assigned to Stream.

4.2.2 *Partial usage of remote memory*

In this case, the applications use all the local memory but only a fraction of the available memory in the remote motherboard. For illustrative purposes, we consider the scenario where 60% of remote memory is used for local applications. That is, if an application uses $X\%$ of remote memory, the other one will use $60\% - X\%$. Notice that since the amount of remote memory consumed is less than 100%, the overall performance will be better than in the previous study.

Figure 5 shows the results for two couples of benchmarks: Stream and FFT (Figure 5 (a)), and Radix and FFT (Figure 5 (b)). To plot the IPC, both applications have been profiled for different percentages of remote memory, the extreme points and two intermediate values (i.e., 0%, 20%, 40% and 60%). As in the previous case, the maximum system performance comes at the expense of unacceptable performance for a given application. For instance, in Figure 5 (a), the best system performance (about 0.73) is achieved when assigning the



(a) Stream vs FFT (b) Radix vs FFT

Fig. 5 Overall IPC for different memory assignments assuming a 60% of remote memory usage

remote memory (i.e., $R = 60\%$) only to Stream, thus clearly damaging its performance. Again, the QoS must be considered to avoid the performance degradation.

In summary, allocating a fraction of remote memory is analogous to allocating all the remote memory, since the only difference lies on the range of memory that is assigned, which is narrower, thus this is a particular case of the previous one. The same reasoning can be applied to the QoS parameters, where the maximum range of assigned remote memory is also limited in extent.

4.3 Extending the analysis to more applications

The analysis is now extended for a number of applications higher than two. Four different mixes have been evaluated and represented in Figure 6: mix1= {Stream, FFT, Cholesky}, mix2= {Stream, FFT, Radix}, mix3= {Stream, Radix, Cholesky}, and mix4= {FFT, Radix, Cholesky}. In these plots, values of X and Z axes refer to the percentage of remote memory assigned to two of the three applications, and $100\% - (X\% + Z\%)$ corresponds to the remote memory assigned to the third application. The Y-axis stands for the total system performance and its highest point shows the maximum performance achieved in the system. As in the previous analysis, the maximum is reached when all the remote memory is assigned to only one application. This application can be chosen as discussed in Section 4.2.1.

For example, when running mix1 (see Figure 6(a)), the maximum IPC (1.75) is achieved when the whole remote memory is assigned to Stream. For this mix, the maximum performance leads to very poor performance for this application. Again, this fact can be controlled by means of the QoS parameter. For example, by setting the QoS of Stream about 15%, its IPC does not drop below 0.1. In the graph, this is equivalent to remove the columns that do not fulfill the required QoS for Stream (i.e., from $Z = 25\%$ to $Z = 100\%$).

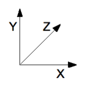
5 Proposed Memory Scheduler

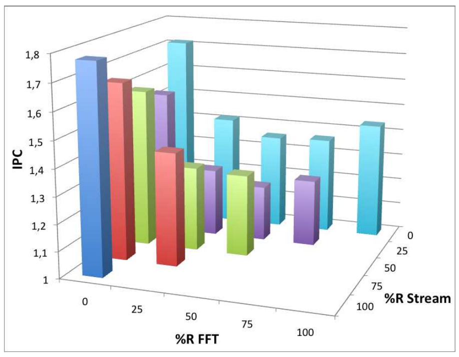
This section presents the memory allocation algorithm devised from the previous analysis. The aim of the algorithm is to distribute the available memory in the three memory regions among n applications running on the nodes of a given local board. This work assumes a static approach where the performance of each application has been profiled off-line varying R for a few points. This profile is provided as an input to the algorithm.

For the sake of clarity, the algorithm is split in two main parts: remote memory assignment and local memory assignment. Remote memory can be assigned by two different algorithms: ideal and heuristic. The former one, referred to as SPP (set of possible permutations) provides the best distribution but it requires a high computational cost, as it is based on an exhaustive search. SPP is useful as a reference to identify the maximum achievable IPC. The heuristic implements a cost-effective algorithm that reduces the computational cost of the SPP in a $(n-1)!$ factor while providing memory distributions close to or the same as SPP.

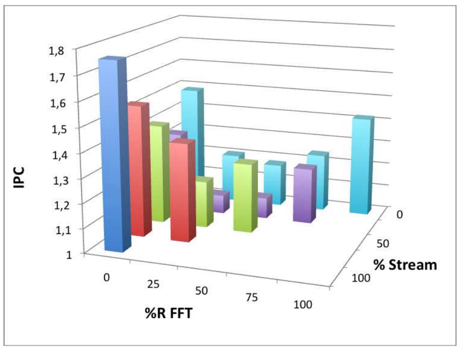
5.1 *SPP* remote memory scheduler

Figure 7 describes the *SPP* remote memory scheduler. The algorithm first checks if there is a need to use remote memory, that is, if the application requires more memory than the

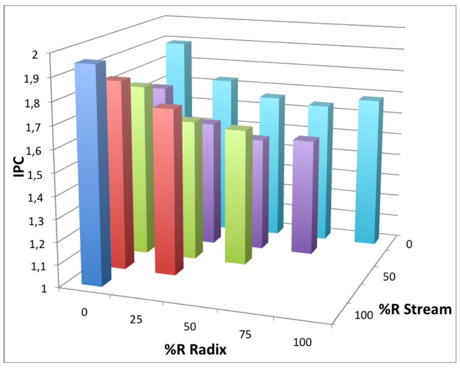




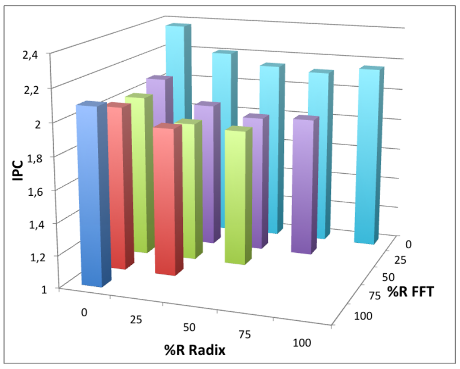
(b) Stream vs FFT vs Cholesky



(c) Stream vs FFT vs Radix



(d) Stream vs Radix vs Cholesky



(e) FFT vs Radix vs Cholesky

Fig. 6 Overall IPC for different memory assignments for three concurrent applications

available RAM in its motherboard (see LABEL 1). On such a case, it makes a search of the optimal remote memory distribution that maximizes the aggregated IPC of all the applications (see LABEL 2). After that, it allocates the remote memory for each application following this distribution (see LABEL 3).

A tuple RM composed of n values $(RM_0, RM_1, ..., RM_{n-1})$ is used to represent a given remote memory distribution among applications, where each value RM_i is the percentage of remote memory assigned to application i. Thus, the sum of the values of a given tuple is 100%. The algorithm has been designed with a QoS parameter to avoid unacceptable performance; that is, it must be fulfilled that the remote memory assigned to application i must be lower or equal than its QoS (i.e., $RM_i \leq QoS_i$ $\forall$ application i).

The experimental results discussed in Section 4 showed that the best performance is achieved when assigning the maximum allowed QoS_i to the application i with the least IPC contribution. The simplest case arises when the remote memory is only distributed between two concurrent applications, A0 and A1. In this case, there are only two choices: assigning as much remote memory as possible to A0 and the remaining to A1, and vice-versa.

```
Algorithm: SPP remote memory scheduler with QoS constraint

Data:
  $n$: number of running applications in the system
  $L$: Available local memory
  $R$: Available remote memory
  $M_i$: Remaining memory required by application $i$
  $QoS_i$: maximum allowed remote memory for application $i$
  $P$: Set of all the possible permutations of $n$ integers from 0 to $n-1$
  $IPCest_i(x)$: IPC estimation based on the profiled points of a given memory assignment $x$ to app. $i$
  $RM_i$: Percentage of remote memory assigned to application $i$. Initially, all its components are null

LABEL 1: CHECK IF THERE IS ENOUGH MEMORY IN THE MOTHERBOARD
if $\Sigma M_{i,\forall i=0..n-1} > L$ then

    LABEL 2: FIND THE OPTIMAL REMOTE MEMORY DISTRIBUTION
    $maxIPC \leftarrow 0$
    for all $p \in P$ do
        $permIPC \leftarrow 0$
        $AM \leftarrow 0\%$
        for $j = 0$ to $n-1$ do
            $RM_{p_j} \leftarrow MIN(QoS_{p_j}, 100\% - AM)$
            $permIPC \leftarrow permIPC + IPCest_{p_j}(RM_{p_j})$
            $AM \leftarrow AM + RM_{p_j}$
            if $AM = 100\%$ then
                exit
            end if
        end for

        if $permIPC > maxIPC$ then
            $max \leftarrow RM$
            $maxIPC \leftarrow permIPC$
        end if
    end for

    LABEL 3: ALLOCATE REMOTE MEMORY
    for $i = 0$ to $n-1$ do
        Allocate $RM_i \times R$ in remote motherboard to application $i$
        $M_i \leftarrow M_i - RM_i \times R$
    end for
end if
```

Fig. 7 SPP algorithm to distribute remote memory among n applications

For a higher number of applications, the algorithm uses a *priority vector*, where the position of a given application in the vector indicates the order in which remote memory is assigned. For instance, let us consider a system executing three concurrent applications with a priority vector $\mathbf{p} = (A0, A1, A2)$ and a quality of service vector $\mathbf{QoS} = (30, 40, 50)$. The priority vector states that the remote memory is first assigned to application A0, then to A1, and finally to application A2. Thus, the assigned remote memory is defined as $\mathbf{RM} = (30, 40, 30)$. In contrast, for a priority vector $\mathbf{p} = (A1, A2, A0)$, the algorithm provides $\mathbf{RM} = (10, 40, 50)$. That is, in the latter case, the remote memory is first assigned to A1, then to A2, and finally to A0.

The set P of possible combinations of the priority vector for n applications matches the set of all the possible permutations of n integers from 0 to $n-1$. For example, for $n = 3$ the set is composed of $P = \{(0,1,2), (0,2,1), (1,0,2), (1,2,0), (2,0,1), (2,1,0)\}$. This

```
Algorithm: Remote memory scheduling heuristic with QoS constraint

 $n$: number of running applications in the system
 $L$: Available local memory
 $R$: Available remote memory
 $M_i$: Remaining memory required by application $i$
 $QoS_i$: maximum allowed remote memory for application $i$
 $IPCest_i(x)$: IPC estimation based on the profiled points of a given memory assignment $x$ to app. $i$
 $RM_i$: Percentage of remote memory assigned to application $i$. Initially, all its components are null

LABEL 1: CHECK IF THERE IS ENOUGH MEMORY IN THE MOTHERBOARD
if $\Sigma M_{i,\forall i=0..n-1} > L$ then

    $AM \leftarrow 0\%$
    while $AM < 100\%$ do
        LABEL 2: FIND THE APPLICATION LEAST AFFECTED BY REMOTE MEMORY
        $minimpact \leftarrow \infty$
        for $i = 0$ to $n-1$ do
            if $RM_i = 0\%$ then
                $assig \leftarrow MIN(QoS_i, 100\% - AM)$
                $impact \leftarrow IPCest_i(0) - IPCest_i(assig)$
                if $impact < minimpact$ then
                    $minimpact \leftarrow impact$
                    $penalized \leftarrow i$
                end if
            end if
        end for
        LABEL 3: REMOTE MEMORY ASSIGNMENT TO THE LEAST AFFECTED APP.
        $RM_{penalized} \leftarrow MIN(QoS_{penalized}, 100\% - AM)$
        $AM \leftarrow AM + RM_{penalized}$
    end while

    LABEL 4: ALLOCATE REMOTE MEMORY
    for $i = 0$ to $n-1$ do
        Allocate $RM_i \times R$ in remote motherboard to process i
        $M_i \leftarrow M_i - RM_i \times R$
    end for
end if
```

Fig. 8 Heuristic to distribute remote memory among n applications

set is used by the SPP algorithm and can be obtained with the Steinhaus-Johnson-Trotter algorithm [15] whose computational cost is $n!$

Regarding the profile size, note that when plotting IPC as a function of the assigned remote memory, the curve looks sufficiently defined using five points. Thus, although the curve could be better defined with more points, its potential benefits would not be compensated with the cost of profiling the additional points. Therefore, the IPC profile of the proposed algorithms consists of the values obtained with five (0%, 25%, 50%, 75% and 100%) remote memory assignments for each application.

The algorithm also estimates the IPC of a given memory assignment when it falls in between two profiled values (e.g., 20%). This estimation can be done in two main ways: i) by using the value of the closest profiled point and ii) by using an approximation method. A linear approximation shows a good tradeoff between speed and accuracy since it can be quickly done (i.e., just a multiplication and a sum operation are required). Of course, more accurate results could be obtained with complex methods like quadratic approximation.

Once the optimal remote memory distribution has been found, the remote memory is allocated to applications, and the assigned memory M_i is updated. Finally, the pending memory is assigned to the local board as discussed in Section 5.3.

5.2 Remote memory scheduling heuristic

The heuristic presented in Figure 8 provides a remote memory distribution close to the optimal. As above, the heuristic relies on the profiled values and takes into account that the best memory performance is achieved when assigning the maximum allowed remote memory to the application with the least IPC contribution.

As the SPP algorithm, the heuristic is only applied if there is not enough local memory (see LABEL 1). In this case, it selects the application whose IPC is the least affected by the remote memory assignment, that is, the minimum value between its QoS and the remote memory still pending to be assigned (i.e., $100\% - AM$) in the system (see LABEL 2). Then, the heuristic calculates and assigns the percentage of remote memory ($RM_{penalized}$) that corresponds to the application selected above (see LABEL 3). The process of assigning remote memory to applications continues until the total remote memory required among the running applications has been assigned. Finally, once obtained the corresponding percentages, the remote memory is allocated to applications (LABEL 4).

5.2.1 Working examples

Let us discuss how the heuristic performs through two working examples: i) no application has QoS constraint, and ii) the applications have QoS requirements. Each working example analyzes four cases or mixes (see Section 4.3), each one composed of three benchmarks.

The simplest example arises when the algorithm works with no QoS constraint, that is, all the applications have their QoS parameter equal to 100%. Table 4 shows the mixes and how the heuristic solves each particular case (see column *Assigned R*). In this example, the overall IPC corresponds to the value of the highest bar of each graph illustrated in Figure 6.

When the applications have QoS requirements and the sum of these values is equal to 100%, the only thing the heuristic has to do is assigning a percentage of remote memory

Table 4 Working example without QoS constraints: **QoS** $= (100, 100, 100)$

Mix	Applications	Quality of Service(%)	Assigned R(%)	Overall IPC
1	Stream	100	100	1.773
	FFT	100	0	
	Cholesky	100	0	
2	Stream	100	100	1.756
	FFT	100	0	
	Radix	100	0	
3	Stream	100	100	1.952
	Radix	100	0	
	Cholesky	100	0	
4	FFT	100	0	2.371
	Radix	100	0	
	Cholesky	100	100	

equal to its QoS to each application. On the other hand, when the sum of the QoS values is greater than 100%, at least one application will receive an amount of remote memory less than its QoS.

The second working example focuses in the latter behavior. Table 5 shows the QoS of the mixes and how the heuristic solves each case. Notice that the QoS vector in the four cases is $\mathbf{QoS} = (20, 55, 70)$, so the sum of its components is not only greater than 100% but also these values do not correspond to any profiled point, so the algorithm must estimate them. To this end, it has been assumed that the algorithm approximates to the closest profiled point. For the first three mixes, the best remote memory distribution vector is $\mathbf{RM} = (20, 10, 70)$. The estimated overall IPC for these situations is approximated to the profiled remote memory percentages 25%, 0% and 75%, respectively. For mix 4, the heuristic provides a memory distribution of $\mathbf{RM} = (0, 30, 70)$, whose overall IPC is estimated by means of the values of the profiled points 0%, 25% and 75%, respectively.

5.2.2 Cost analysis

The SPP algorithm carries out a thorough search among the set of the possible remote memory assignments (i.e., $n!$) to find out the combination that optimizes the overall IPC. This set grows factorially with the number of applications running on the system so it leads to prohibitive computational costs for large sets. On the contrary, the devised heuristic algorithm finds an optimal or near-optimal remote memory distribution but largely reduces the computational cost by performing a reduced search.

Table 6 shows the computational costs of both schedulers. The SPP algorithm iterates through $n!$ different cases of n-cost each. As a result of the limitation in the number of possible remote memory assignments explored, the proposed heuristic reduces the number of analyzed cases from $n!$ to n, thus noticeably improving the total computational cost of the SPP algorithm.

To sum up, the proposed heuristic reduces the computational cost in a factor of $(n-1)!$ while providing reasonable performance since all the QoS of the applications are satisfied. On the contrary, the SPP algorithm might provide better performance for some mixes but at the expense of a much higher computational cost.

Table 5 Working example with QoS restrictions: $\mathbf{QoS} = (20, 55, 70)$

Mix	Applications	Quality of Service(%)	Assigned R(%)	Overall IPC
1	Stream	20	20	1.542
	FFT	55	10	
	Cholesky	70	70	
2	Stream	20	20	1.332
	FFT	55	10	
	Radix	70	70	
3	Stream	20	20	1.721
	Radix	55	10	
	Cholesky	70	70	
4	FFT	20	0	2.207
	Radix	55	30	
	Cholesky	70	70	

Table 6 Computational cost comparison

Algorithm	# Cases	Cost/case	Total cost
SPP	$n!$	n	$n \times n!$
Heuristic	n	n	$n \times n$

5.3 Local memory assignment

Once the remote memory has been scheduled to applications by any of the studied schedulers, the local memory must be assigned to complete the memory allocation. Figure 9 shows the local memory scheduler. For each application and following a circular order, the scheduler looks for free memory in each node of the local motherboard, beginning by the node where the application is running on. This process goes on until the required memory has been completely assigned to applications.

6 Related work

Different research papers dealing with remote memory allocation which mainly focus on memory swapping can be found in the literature.

Oleszkiewicz et al. propose a peer-to-peer solution called parallel network RAM [16]. This approach avoids the use of disk and better utilizes the available RAM resources in a cluster. It reduces the computational, communication and synchronization overhead typically involved in parallel applications.

Shuang et al. design a remote paging system for remote memory utilization in InfiniBand clusters [17]. These systems present the design and implementation of a high performance networking block device (HPBD) over InfiniBand fabric, which serves as a swap device of a virtual memory (VM) system for efficient page transfer to/from remote memory servers. They demonstrate that quicksort performs 1.45 times slower than local memory system, and up to 21 times faster than local disk.

```
Algorithm: Local memory scheduler

Data:
  $n$: number of running applications in the system
  $M_i$: Remaining memory required by application $i$
  $L_i$: Memory available in node $i$
for $i = 0$ to $n-1$ do
  if $M_i > 0\%$ then
    $j \leftarrow i$
    repeat
      $toAlloc \leftarrow MIN(M_i, L_j)$
      $M_i \leftarrow M_i - toAlloc$
      $L_j \leftarrow L_j - toAlloc$
      $j \leftarrow (j+1)MOD(n)$
    until $M_i = 0\%$
  end if
end for
```

Fig. 9 Algorithm to distribute local memory among n applications

Oguchi et al. investigate the feasibility of using available memory in remote nodes as a swap area when execution nodes need to swap out their real memory contents during the execution of parallel data mining on PC clusters [18]. In this work, nodes executing applications dynamically acquire extra memory from remote nodes through an ATM network. Experimental results on a PC cluster show that the proposed method is considerably better than using hard disks as a swapping device.

In [19], the use of remote memory for virtual memory swapping in a cluster computer is described. The design uses a lightweight kernel-to-kernel communications channel for fast and efficient data transfer. Performance tests are made to compare the proposed system to normal hard disk swapping. The tests show significantly improved performance when data access is random.

Midorikawa et al. propose the distributed large memory system (DLM), which is an user-level software-only solution that provides very large virtual memory by using remote memory distributed over the nodes in a cluster [20]. The performance of DLM programs that access remote memory is compared to ordinary programs that use local memory. The results of STREAM, NPB and Himeno benchmarks show that the DLM achieves better performance than other remote paging schemes using a block swap device to access remote memory. To obtain high performance, the DLM can tune its parameters independently from kernel swap parameters.

These papers use the remote memory for swapping over cluster nodes and present their system as an improvement of disk swapping. On the contrary, in this work the idle remote memory is allocated by a memory scheduler to the running applications in other motherboard as an extension of their local memory while guaranteeing their QoS requirements and without any kind of memory swapping.

7 Conclusions

Cluster computers represent an important segment of the current supercomputer market. They can use RMA hardware as an inexpensive and relatively fast way to extend the memory space of a single motherboard of the cluster. However, the accessible memory through RMA has latencies one order of magnitude higher than the local memory, which in turn can have different latencies depending on whether the accessed memory is on the processor running the application or not. Thus, memory allocation algorithms for such a cluster computer must be carefully designed to avoid potential performance penalties.

This work has presented an ideal algorithm and an heuristic to assign main memory, which can be located in three main regions (local to node, local to board, or remote), among applications running on a RMA-interconnected cluster. With this aim, three main steps have been followed to design these schedulers. For each one, different conclusions can be drawn.

First, since benchmarks have different memory requirements, the impact on performance of each application varying the memory distribution among regions (L, Lb, and R) has been studied. This study has shown that, i) the memory distribution between L and Lb can impact on performance when no R memory is allocated, ii) the previous distribution has a slight effect on the performance when R is allocated, and iii) performance degradation widely varies among applications when allocating R memory.

Second, the system performance, when several benchmarks running concurrently compete for memory, has been analyzed. This study has shown that the total performance is benefited when R memory is assigned (as much as possible) first to the application that least degrades its performance, then to the second one, and so on. However, this assignment strategy can lead to unacceptable performance degradation for some applications. Hence, a quality of service parameter has been defined for each application.

From these studies, a memory scheduling heuristic that approximates the best local and remote memory distribution among applications has been presented. The heuristic has been designed to guarantee a minimum QoS performance for each benchmark while optimizing the global system performance. Results have shown that the memory distribution provided by the heuristic is close or the same as the optimal distribution found by the ideal algorithm (SPP), whose computational cost is prohibitive for a high number of applications.

References

1. IBM journal of Research and Development staff. Overview of the IBM Blue Gene/P project. *IBM J. Res. Dev.*, 52(1/2):199–220, 2008.
2. M. Blocksome, C. Archer, T. Inglett, P. McCarthy, M. Mundy, J. Ratterman, A. Sidelnik, B. Smith, G. Almási, J. Castaños, D. Lieber, J. Moreira, S. Krishnamoorthy, V. Tipparaju, and J. Nieplocha. Design and implementation of a one-sided communication interface for the IBM eServer Blue Gene®supercomputer. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, pages 54–54, Tampa, FL, USA, November 2006.
3. Sameer Kumar, Gábor Dózsa, Gheorghe Almasi, Philip Heidelberger, Dong Chen, Mark Giampapa, Michael Blocksome, Ahmad Faraj, Jeff Parker, Joe Ratterman, Brian E. Smith, and Charles Archer. The deep computing messaging framework: generalized scalable message passing on the blue gene/P supercomputer. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, pages 94–103, Island of Kos, Greece, June 2008.
4. V. Tipparaju, A. Kot, J. Nieplocha, M.T. Bruggencate, and N. Chrisochoides. Evaluation of Remote Memory Access Communication on the Cray XT3. In *Proceedings of the 21th International Parallel and Distributed Processing Symposium*, pages 1–7, Long Beach, California, USA, March 2007.
5. Mondrian Nussle, Martin Scherer, and Ulrich Bruning. A Resource Optimized Remote-Memory-Access Architecture for Low-latency Communication. *International Conference on Parallel Processing*, pages 220–227, Sept. 2009.
6. http://www.hypertransport.org/.
7. Monica Serrano, Julio Sahuquillo, Houcine Hassan, Salvador Petit, and José Duato. A Scheduling Heuristic to Handle Local and Remote Memory in Cluster Computers. In *Proceedings of the 12th IEEE International Conference on High Performance Computing*, pages 35–42, Melbourne, Australia, Sept. 2010.
8. Chetana N. Keltcher, Kevin J. McGrath, Ardsher Ahmed, and Pat Conway. The AMD Opteron Processor for Multiprocessor Servers. *IEEE Micro*, 23(2):66–76, 2003.
9. J. Duato, F. Silla, and S. Yalamanchili. Extending HyperTransport Protocol for Improved Scalability. *First International Workshop on HyperTransport Research and Applications*, 2009.
10. H. Litz, H. Fröening, M. Nuessle, and U. Brüening. A HyperTransport Network Interface Controller for Ultra-low Latency Message Transfers. *HyperTransport Consortium White Paper*, 2007.
11. https://www.simics.net/.
12. http://www.cs.wisc.edu/gems/.
13. http://www.cs.virginia.edu.
14. Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995.
15. Anany Levitin. *Introduction to The Design and Analysis of Algorithms*. Addison Wesley, 2003.
16. John Oleszkiewicz, Li Xiao, and Yunhao Liu. Parallel Network RAM: Effectively Utilizing Global Cluster Memory for Large Data-Intensive Parallel Programs. In *Proceedings og 33rd International Conference on Parallel Processing*, pages 353–360, Montreal, Quebec, Canada, 2004.
17. Shuang Liang, Ranjit Noronha, and Dhabaleswar K. Panda. Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device. In *Proceedings of the 2005 IEEE International Conference on Cluster Computing*, pages 1–10, Boston, Massachusetts, USA, 2005.
18. Masato Oguchi and Masaru Kitsuregawa. Using Available Remote Memory Dynamically for Parallel Data Mining Application on ATM-Connected PC Cluster. In *Proceedings of the 14th International Parallel & Distributed Processing Symposium*, pages 411–420, Cancun, Mexico, 2000.

19. Paul Werstein, Xiangfei Jia, and Zhiyi Huang. A Remote Memory Swapping System for Cluster Computers. In *Proceedings of the Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 75–81, Adelaide, Australia, 2007.
20. H. Midorikawa, M. Kurokawa, R. Himeno, and M. Sato. DLM: A distributed Large Memory System using remote memory swapping over cluster nodes. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, pages 268–273, Tsukuba, Japan, October 2008.