FLEX-MPI: An MPI Extension for Supporting Dynamic Load Balancing on Heterogeneous Non-dedicated Systems

Gonzalo Martín[1], Maria-Cristina Marinescu[2], David E. Singh[1], and Jesús Carretero[1]

[1] Universidad Carlos III de Madrid, Leganés, 28911, Spain
gmcruz@arcos.inf.uc3m.es
[2] Barcelona Supercomputing Center, Barcelona 08034, Spain

Abstract. This paper introduces FLEX-MPI, a novel runtime approach for the dynamic load balancing of MPI-based SPMD applications running on heterogeneous platforms in the presence of dynamic external loads. To effectively balance the workload, FLEX-MPI monitors the actual performance of applications via hardware counters and the MPI profiling interface—with a negligible overhead and minimal code modifications. Our results show that by using this approach the execution time of an application may be significantly reduced.

Keywords: Dynamic load balancing, distributed computing, heterogeneous systems, hardware counters.

1 Introduction

The work described in this paper focuses on the efficient distribution of program workloads on heterogeneous platforms composed of processors with the same instruction set architecture (ISA) but with different performance. This work targets parallel applications based on the SPMD (*Single Program Multiple Data*) paradigm. A large proportion of these applications are iterative and alternate phases of computation and communication; linear system solvers such as Jacobi and Conjugate Gradient from NPB [1] are good representatives of this class of applications and are used as benchmarks in our evaluation. We also evaluated our approach on EpiGraph [2], a significantly more complex HPC application.

We introduce FLEX-MPI, an MPI extension which monitors the performance of an application and uses this information to make decisions with respect to the distribution of the workload and the data between processes. We focus on an adaptive strategy for balancing the workload of applications that run on non-dedicated systems, in which several applications run concurrently and share the computing resources, e.g. CPU, memory, and cache. Sharing resources means that applications have external loads which degrade their performance.

We consider both burst and long-term external loads. Burst loads correspond to short-duration external loads which do not significantly affect the application performance. Long-term external loads reduce the application's CPU usage affecting its performance. FLEX-MPI is able to discriminate between these two kinds of loads and flexibly apply different load balance policies depending on their magnitude. One of the advantages of this approach is that it does not require prior knowledge about the underlying architecture. We use hardware counters and the MPI profiling interface to directly measure performance metrics at runtime. The main contributions of this work are:

– A **precise**, **flexible** dynamic load balancing technique based on monitoring the actual performance of the applications via hardware counters and the MPI profiling interface.
– A **powerful**, **decentralized** approach that works for homogeneous and heterogeneous systems which can be either dedicated or non-dedicated.
– A **low overhead**, **generic** method for integrating dynamic load balancing into existing MPI-based SPMD applications.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 introduces FLEX-MPI. Before concluding in Section 5, we present an extensive performance evaluation in Section 4.

2 Related Work

Dynamic load balancing for heterogeneous systems is a topic of great interest since current distributed computing systems—mainly grid and cloud, but also cluster—are becoming predominantly heterogeneous [3]. An efficient approach, originally designed for homogeneous systems, consists in adapting the parallel code by software techniques which balance the load depending on the computational power of each computing unit. But to adapt parallel code to run on a heterogeneous system requires prior knowledge about the characteristics of the architecture [4], which is not always feasible.

The basic approach for dynamic load balancing in heterogeneous systems is based on a theoretical model of the system. Belikov *et al.* [5] present an architecture-aware cost modeling technique based on theoretical CPU speed, cache, RAM, and latency. However, theoretical values do not always match the performance achieved when running a real application and do not consider the external load introduced by other processes running on the same processor.

Several projects propose approaches based on collecting performance metrics at runtime. Galindo *et al.* [6] present a model based on the relative computing power, a metric which is obtained by measuring the execution time invested by a processor in performing a given computation. The computation is measured as the number of rows of a dense matrix, an inaccurate model when it comes to sparse data structures. Their approach only considers executions on dedicated systems. ALBIC [7] is a system based on [6] which measures the system load by collecting performance metrics at runtime. However, this technique is intrusive since to collect this data and feed it to the monitoring system they add a specific module in the Linux kernel. A similar approach is Dyn-MPI [8], a dynamic MPI implementation which targets parallel applications running on non-dedicated architectures. Dyn-MPI requires a daemon running in each computing node to extract performance metrics. It is highly code intrusive since many of the calls, including standard MPI functions, must be instrumented.

Bohn *et al.* [9] measure the performance of compute nodes by extracting information from files of the Linux OS and benchmarking both the processor and the memory, operations which are usually expensive. HeteroMPI [10] is an MPI extension which was specifically designed for programming on heterogeneous systems. It can measure processor performance by using a benchmarking function whose code must be provided by the programmer. HeteroMPI requires a significant intrusive instrumenting, even for simple parallel programs. AdaptiveMPI [11] is an adaptive implementation of MPI built on top of the CHARM++ runtime environment which supports dynamic load balancing through processor virtualization. It only offers full compatibility with the MPI-1.1 features and MPI standard programs need to be significantly modified.

Hardware counters have been demonstrated to be an effective way of measuring computer performance [12]. FLEX-MPI introduces a novel dynamic load balancing algorithm which flexibly adapts to external load in heterogeneous non-dedicated systems. Our approach is based on collecting precise system performance metrics at runtime via hardware counters and can be integrated in existing MPI-based SPMD applications with minimal code modifications.

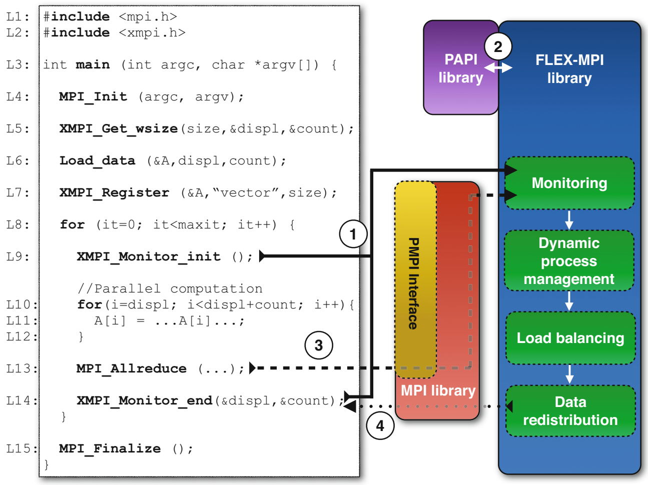
3 FLEX-MPI

FLEX-MPI is an MPI extension which integrates three functionalities: *monitoring*, *load balancing* (LB), and *data redistribution*. We implemented FLEX-MPI as a library on top of the MPICH-2 implementation. This makes it fully compatible with the MPI-2 features and allows it to easily link with any existing MPI-based application. FLEX-MPI's API is described in detail in [13].

3.1 Monitoring

The purpose of the monitoring functionality is to collect performance metrics for each process of the parallel application during its execution. The applications we target are iterative and alternate computation and communication phases. We monitor computation by means of hardware counters (via PAPI [14]) and communication by using the MPI profiling interface (PMPI), which allows to profile the communications without modifying the source code of the application.

FLEX-MPI targets SPMD applications using one-dimensional domain decomposition with distributed data, a parallelization method used by a large number of scientific parallel applications. In these applications the portion of the domain assigned to a process is usually expressed as a combination of a count—which represents the number of elements, rows, or columns assigned to the process—and a displacement. Fig. 1 illustrates an example of a SPMD application using the FLEX-MPI library, in which the data structure managed by the application



```
L1:  #include <mpi.h>
L2:  #include <xmpi.h>

L3:  int main (int argc, char *argv[]) {

L4:    MPI_Init (argc, argv);

L5:    XMPI_Get_wsize(size,&displ,&count);

L6:    Load_data (&A,displ,count);

L7:    XMPI_Register (&A,"vector",size);

L8:    for (it=0; it<maxit; it++) {

L9:      XMPI_Monitor_init ();

         //Parallel computation
L10:     for(i=displ; i<displ+count; i++){
L11:       A[i] = ...A[i]...;
L12:     }

L13:     MPI_Allreduce (...);

L14:     XMPI_Monitor_end(&displ,&count);
       }

L15:   MPI_Finalize ();
     }
```

Fig. 1. Structure and runtime calls of a parallel code linked with the FLEX-MPI library

(vector A) is distributed between the processes. Each process operates in parallel (L10-12) on a different subset of the data structure. The parallel code is instrumented with a set functions to get the initial partition of the domain assigned to every process (L5), register the data structure managed by the application (L7), enable monitoring (L9), and rebalance the load as needed (L14). The monitoring functionality dynamically collects performance metrics provided by PAPI (arrow labeled 2) and the MPI library through the PMPI interface (arrow labeled 3). When a program iteration finishes (at line L14) these metrics are fed to the LB functionality (arrow labeled 1), which computes the new distribution of the workload. In our work, we consider each of the computing cores of modern multiprocessors as an independent processing element (PE). After a new workload distribution has been decided, the data redistribution functionality moves the data to the processes that will need it (arrow labeled 4). A more detailed description of the instrumentation process can be seen in [13].

Our implementation uses low level PAPI interfaces to track the number of floating point operations $FLOP$, the real time $Treal$ (i.e. the wall-clock time), and the CPU time $Tcpu$ (i.e. the time during which the processor is running in user mode). These metrics are collected for each process of the parallel application, and they are preserved during context switching. The $FLOP$ is needed to effectively track and measure the performance at the granularity of each processing element, while the real time and CPU time allow us to identify if there exists external application load. In our model, we assume floating-point based applications which exhibit a linear correlation between the $FLOP$ and the workload size, which is reasonable for many parallel applications (e.g. linear system solvers). An initial calibration is required because in heterogeneous systems the events counted by hardware counters are processor specific. This calibration is carried out by performing a microbenchmark with a negligible overhead before starting the computation of the application.

3.2 Load Balancing

The load balancing functionality receives as input the per-process values for the performance metrics measured via monitoring. When load imbalance is detected, the algorithm decides the new distribution of workload based on the per-process performance metrics. Although monitoring can be performed at every iteration we trigger load balancing only every *sampling interval*—consisting of a fixed number of iterations—to reach a trade-off between the overhead of this operation and the performance gain as result of it.

To decide how much workload to re-assign to each process, the load balancing algorithm first computes the $MFLOPS$ that each process i executed during the previous sampling interval. $MFLOPS_i$ is defined in Equation 1 as the ratio between the number of floating point operations $FLOP_i$ and the real execution time $Treal_i$ during a given sampling interval. The fraction of the workload assigned to process i is computed in Equation 2 depending on the *relative computing power* (RCP_i) of a process i, which is computed as the $MFLOPS_i$ divided by the total $MFLOPS$ for all of the p processes. RCP is used to estimate workload distribution on parallel applications, since it provides a normalized value of the computational power of a process relative to the computational power of the whole system [4,7].

$$MFLOPS_i = \frac{FLOP_i}{Treal_i} \quad (1) \qquad RCP_i = \frac{MFLOPS_i}{\sum_{i=0}^{p} MFLOPS_i} \quad (2)$$

Algorithm 1 shows the pseudocode for the load balancing algorithm, which is evaluated at each sampling interval n. The first step (`line 1`) detects which of the processing elements involved in executing the application are dedicated and which are not. When the difference between the CPU time and the real time of a processing element is small we can safely assume that it executes only one process. When the real time is significantly higher than the CPU time then the processing element is being shared between multiple processes - either of the same, or of a different application. The real time is always a little higher than the CPU time because of OS interrupts; we use a threshold parameter TH_1 to account for this overhead and mark the difference between dedicated and non-dedicated processing elements. We consider that values of the real time that surpass the CPU time by 5% are reasonable for setting the tolerance threshold TH_1. Each process uses a boolean variable `dedicated` to record whether it uses the processing element in exclusive mode or not. By applying a reduce operation (`line 6`) over all processes we know whether there exists any non-dedicated processing element. The reduction result—stored in the variable `global_dedicated`—is false if at least one processing element is non-dedicated.

We evaluate whether we should redistribute the workload if either (1) the processing elements have been used in exclusive mode during the current sampling interval but the application is unbalanced or (2) long-term external load is detected on any of the processing elements. It is possible that the applications that share the resources with our application execute during short, isolated bursts which do not affect the overall performance of our application.

Algorithm 1. Pseudocode for the load balancing algorithm

```
1: if $((Treal_i - Tcpu_i)/Treal_i) < TH_1$ then
2:     $dedicated \leftarrow$ true
3: else
4:     $dedicated \leftarrow$ false
5: end if
6: $global\_dedicated \leftarrow Allreduce(dedicated, AND)$
7: $buf[n] \leftarrow global\_dedicated$
8: $external\_load \leftarrow evaluate\_external\_load(buf, k)$
9: if $(global\_dedicated == true)$ or $(external\_load == long\_term)$ then
10:     $\{FLOP, Treal\} = Allgather(FLOP_i, Treal_i)$
11:     $MFLOPS = compute\_MFLOPS(FLOP, Treal)$
12:     if $(max(Treal) - min(Treal)) > (TH_2 * max(Treal))$ then
13:         $RCP = compute\_RCP(MFLOPS)$
14:         $Data\_redistribution(RCP)$
15:     end if
16: end if
```

In contrast, long-term external loads consume a lot of computer resources during a continuous period of time and significantly degrade the overall performance of our application. In our algorithm each process stores the value of variable `global_dedicated` at each sampling interval (n) (`line 7`). When a processing element has been running in non-dedicated mode during k consecutive sampling intervals it is considered that long-term external load is present on that processing element and the workload should be considered to be redistributed. The function `evaluate_external_load` returns *long_term* when any of the processing elements have been running in non-dedicated mode during the past k sampling intervals (`line 8`). Section 4.2 discusses practical values of k.

If either all of the processing elements are dedicated during the current sampling interval, or long-term external load has been detected (`line 9`), then the algorithm analyzes the load balance of the application. Otherwise, when a bursty external load is detected, the algorithm tolerates it without performing load balancing for $(k-1)$ consecutive sampling intervals. In the k^{th} sampling interval one of two things will happen: (1) either there will be another burst, in which case it leads to the conclusion that rather than a series of bursts, a long-term load is present, or (2) the processing elements will run in dedicated mode, in which case it will also be a candidate for load balancing evaluation. When the application is evaluated for load balancing, the algorithm gathers and distributes the $FLOP$ and real time numbers of each process (`line 10`) to all the other processes. It then applies Equation 1 to compute $MFLOPS_i$ locally by each process i (`line 11`). If the difference between the maximum and minimum values of $Treal$ is larger than the threshold value TH_2 * *max(Treal)* (`line 12`) then the application is more unbalanced than what it can tolerate. In our experiments, we empirically set TH_2 to 15%. As a result, LB triggers the redistribution of workload based on Equation 2 and the RCP of each process (`line 13`), then uses the new workload distribution to perform the data remapping by invoking the data distribution functionality (`line 14`).

The algorithm implemented focuses on balancing computation workloads and requires precise computation time measurements which do not take into account the time spent on performing communication operations. Otherwise, the algorithm would lead to inaccurate workload distributions. For instance, an imbalanced parallel application where the fastest process spends most of the time waiting idle for other processes involved in communication operations will have a low $FLOP$ count and large $Treal$. By profiling MPI communications we can compute separately the time spent by each process in performing computation and communication, enabling a precise load balancing policy.

3.3 Data Redistribution

In SPMD applications the data is usually distributed—rather than replicated—between processes, which requires redistribution to move the data between processes each time a load balance operation is carried out. FLEX-MPI includes a data redistribution functionality which handles both one-dimensional (e.g. vectors) and two-dimensional (e.g. matrices) data structures, which may be either dense or sparse.

The user has to register each of the data structures which will need to be redistributed as result of load balance operations. The registering function (`XMPI_Register`) receives as input the pointer to the data structure and the size of the data structure. Depending on the domain decomposition type, the sizes of the data structures can be provided either as the number of elements, rows, or columns of the structures. FLEX-MPI can manage several data structures whenever they have been registered using the same type of domain decomposition.

Once the load balancing functionality has computed the RCP of each PE and the new workload distribution has been mapped to a data partition, the data redistribution functionality (1) computes the range of data associated to the new workload partition of every process, and (2) moves the data from the previous to the new owners. `XMPI_Monitor_end` returns—on behalf of the data redistribution functionality—the new count and displacement for the new data mapping used by each process. MPI standard messages are used to efficiently move data between MPI processes.

4 Performance Evaluation

We evaluate our approach using three iterative SPMD applications—Jacobi, Conjugate Gradient, and EpiGraph. Jacobi is an iterative method for solving a system of linear equations which operates on a symmetric dense matrix. Conjugate Gradient (CG) is a linear system solver suited for large sparse matrices. EpiGraph [2] is a distributed simulator for infectious diseases which iteratively operates on a sparse matrix which reflects a social interconnection graph.

For the experiments using Jacobi we generated random square matrices with different sizes: 5,000, 10,000 and 15,000 rows. For CG we used matrices from the University of Florida sparse matrix collection [15]. The matrices we selected are *nd24k* (size: 72,000, *number of nonzero elements* (*nnz*) 28,715,634), *ldoor* (size: 952,203, *nnz* 42,493,817), and *audikw_1* (size: 943,695, *nnz* 77,651,847). This subset is representative of data structures which exhibit regular and irregular data distribution patterns. We run Jacobi and CG for 10,000 iterations each. For the experiments using EpiGraph we simulated a population of 1,000,000 people (matrix size: 1,000,000, *nnz* 38,473,353) for a simulated time span of 20 days (2,880 iterations). The *sampling interval* is problem dependent and we experimentally set it to 100 iterations in our experiments.

Table 1 describes our target platform, a heterogeneous cluster consisting of 10 compute nodes of four different classes. All the compute nodes run under Linux Ubuntu Server 10.10 with 2.6.35-32 kernel and MPICH-2 (v.1.4.1p1), and are interconnected by a Gigabit Ethernet network.

4.1 Heterogeneous Dedicated System

We first evaluate FLEX-MPI by executing Jacobi, CG, and EpiGraph exclusively on heterogeneous configurations running 4, 8, 16, 32, and 64 processes. Table 2 describes the heterogeneous configurations of the cluster.

Table 3 shows the execution times for Jacobi and CG while Table 4 shows the execution times for EpiGraph. In our experiments we show the overall execution time (of the application and FLEX-MPI), including the computation and communication times of the application as well as the overhead of the monitoring, load balancing, and data redistribution of FLEX-MPI. The reference scenario (which execution time is T_{par}) employs an equal-size block distribution of the

Table 1. Heterogeneous cluster with number of nodes (N), sockets per node (S), and processing elements (PE) per socket for each class

Class	N	S	PE	Processor	Frequency	RAM
A	4	1	4	Intel Xeon E5405	2.00 GHz	4 GB
B	2	2	6	Intel Xeon E5645	2.40 GHz	24 GB
C	2	2	6	AMD Opteron 6168	800 MHz	64 GB
D	2	4	6	Intel Xeon E7-4807	1.87 GHz	128 GB

Table 2. Heterogeneous configurations, where $n(p)$ stands for the number of nodes (n) and the number of processes (p) running per node

Config.	Class A	Class B	Class C	Class D
HTC1-4	1 (1)	1 (1)	1 (1)	1 (1)
HTC2-8	1 (2)	1 (2)	1 (2)	1 (2)
HTC3-16	1 (4)	1 (4)	1 (4)	1 (4)
HTC4-32	2 (4)	1 (8)	1 (8)	1 (8)
HTC5-64	4 (4)	2 (8)	2 (8)	2 (8)

data without load balancing. Results show a significant improvement of up to 44% when executing the application with dynamic load balancing. Note that for 64 processes, the communication/computation ratio increases due to low workload per process. This produces performance degradation for the applications, and reduces the efficiency of the load balancing.

Fig. 2 illustrates a typical execution of Jacobi and EpiGraph when using FLEX-MPI. Jacobi is a regular application in which the amount of work done in each iteration is the same. This leads to very small variations over time in the execution time per iteration. In contrast, EpiGraph is an irregular application which exhibits a highly variable workload per iteration. When executing on a heterogeneous dedicated system, Jacobi requires a single data redistribution operation to balance the workload. It is triggered during the first sampling interval, in which the workload imbalance is larger than the imbalance tolerated by the algorithm. From that moment on the application is balanced and no further data redistribution operations are necessary. However, even on dedicated systems, irregular applications such as EpiGraph require several data redistribution operations to balance the workload.

Table 3. Heterogeneous dedicated system with: execution time of the application (T_{par}), execution time of the dynamic load balanced application - with FLEX-MPI (T_{FLX}), percentage of the time saved when executing with FLEX-MPI (T_{sav})

	Jacobi				CG			
Config.	Matrix	T_{par}(sec)	T_{FLX}(sec)	T_{sav}(%)	Matrix	T_{par}(sec)	T_{FLX}(sec)	T_{sav}(%)
HTC1-4	5,000	805	517	35.77	*nd24k*	1247	998	19.96
	10,000	3259	2071	36.45	*ldoor*	2634	2285	13.24
	15,000	7324	4683	36.05	*audikw_1*	4860	3537	27.22
HTC2-8	5,000	414	269	35.02	*nd24k*	682	538	21.11
	10,000	1665	1070	35.73	*ldoor*	1751	1676	4.28
	15,000	3707	2396	35.36	*audikw_1*	3562	2222	37.61
HTC3-16	5,000	208	151	27.40	*nd24k*	381	302	20.73
	10,000	843	698	17.20	*ldoor*	1387	1327	4.32
	15,000	1894	1384	26.92	*audikw_1*	2336	1789	23.41
HTC4-32	5,000	116	95	18.10	*nd24k*	220	188	14.54
	10,000	421	332	21.14	*ldoor*	1253	1234	1.51
	15,000	978	706	27.81	*audikw_1*	1844	1104	40.13
HTC5-64	5,000	108	100	7.40	*nd24k*	147	146	0.68
	10,000	580	446	23.10	*ldoor*	841	815	3.09
	15,000	880	756	16.40	*audikw_1*	1124	911	18.95

Table 4. Results of EpiGraph on heterogeneous dedicated system

	EpiGraph			
Config.	Matrix	T_{par}(sec)	T_{FLX}(sec)	T_{sav}(%)
HTC1-4	1,000,000	356	270	24.16
HTC2-8	1,000,000	222	156	29.73
HTC3-16	1,000,000	202	113	44.06
HTC4-32	1,000,000	161	102	36.65
HTC5-64	1,000,000	165	112	32.12

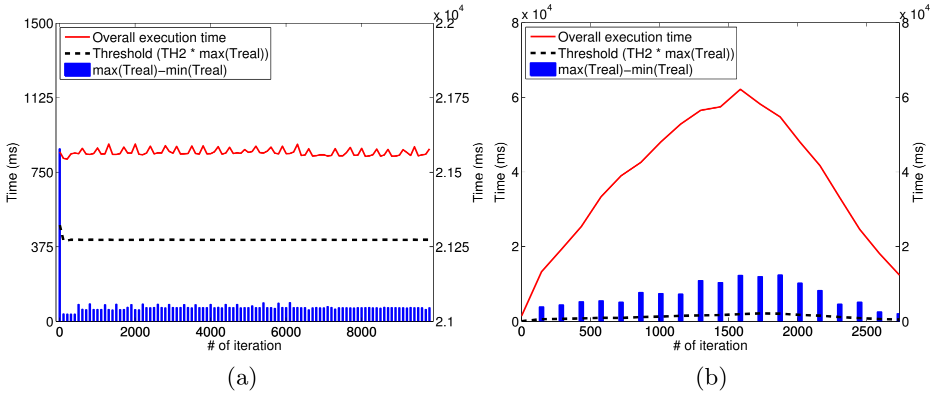


Fig. 2. Jacobi (a) and EpiGraph (b) on heterogeneous dedicated system. Right Y axis is the *Overall execution time* per iteration. Left Y axis shows both the difference between the maximum and minimum *Treal* (for all of the running processes for each sampling interval), and the *Threshold* value tolerated by the algorithm.

4.2 Heterogeneous Non-dedicated System

The following experiment evaluates how well the load balancing algorithm performs when external applications with workload that vary over time are sharing the underlying architecture for execution. We run Jacobi, CG, and EpiGraph for a heterogeneous configuration with 1 node Class A and 1 node Class B, each running 4 processes per node. We artificially introduce an external load which simulates an irregular computing pattern. This load consists of two processes which are simultaneously executed on the Class A node together with the application. The external load consists of a burst of short computing intervals followed by a single long computing interval which lasts until the end of the execution.

Table 5 shows the execution times for the benchmarks on the heterogeneous non-dedicated configuration we described above. We evaluated different values of k and their impact on the execution time. T_{par} stands for the execution time (in seconds) of the application when it runs without adapting to changes in performance due to the dynamic external load; $T_{k=n}$ stands for these execution times when the application does adapt to the external load, for different values of k. The execution time is reduced by up to 39.31% when the applications adapts to the external load. Results confirm our intuition to show that the most

Table 5. Jacobi, CG, and EpiGraph on heterogeneous non-dedicated system

Problem	Matrix	T_{par}(sec)	$T_{k=1}$(sec)	$T_{k=3}$(sec)	$T_{k=5}$(sec)
Jacobi	10,000	1652	1162	1162	1148
CG	*nd24k*	1076	688	665	653
EpiGraph	1,000,000	272	195	179	169

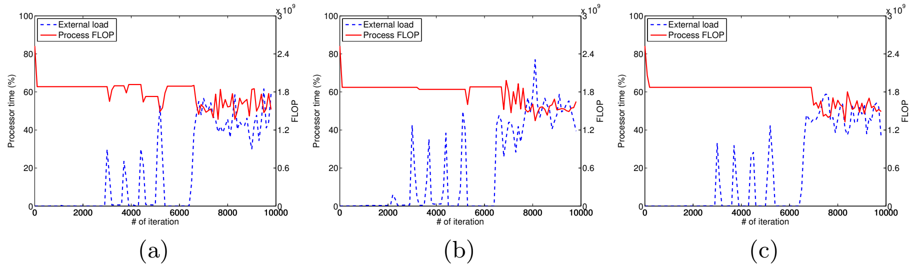


Fig. 3. Adaptive execution of Jacobi on a heterogeneous non-dedicated system for different values of k: (a) $k = 1$, (b) $k = 3$, and (c) $k = 5$. The external load (left Y axis) corresponds to the percentage of real time of the processing element consumed by the external load, while the process $FLOP$ (right Y axis) corresponds with the number of $FLOP$ performed by the process.

promising approach is to tolerate short external loads as to avoid the cost of re-balancing too eagerly.

Fig. 3 shows what happens on processing element $P0$ when we run Jacobi using FLEX-MPI and we introduce a dynamic external load on a subset of the processing elements. The workload redistribution triggered by the load balancing algorithm leads to a different number of $FLOP$ performed by the process (in red in the figure). The amount of data which needs to be redistributed depends on the magnitude of the external load (in blue) and the value of k. We can observe that for $k = 1$ the application adapts immediately to changes in the performance of the processing element, performing load balance for every external load burst. With $k = 3$ the first three smaller bursts are discarded, while larger values of k lead to discarding all the bursts but considering the long-term load.

5 Conclusions

We presented FLEX-MPI, an MPI extension for supporting dynamic load balancing of SPMD applications running on heterogeneous platforms in the presence of dynamic external workload. The extension we provide does not require prior knowledge about the underlying architecture, does not require dedicated resources, and it is based on precise runtime monitoring with negligible overhead. Our results show that by using FLEX-MPI the execution time of an application may be significantly reduced.

There are two main directions for future work that are of particular interest to us. The first extension we plan on developing is to improve FLEX-MPI by considering other types of hardware events, which are used to monitor other performance metrics. This is particularly useful for parallel applications which do not exhibit a linear correlation between the FLOP and the workload size, or applications based on integer operations. The second direction for future work is to improve FLEX-MPI by integrating the dynamic process management features of MPI-2 such that processes could be started or turned off on demand, based on a cost model and the performance goals of the application.

Acknowledgments. This work has been partially supported by the Spanish Ministry of Science under the grant IPT-430000-2010-14.

References

1. Bailey, D., et al.: The NAS parallel benchmarks summary and preliminary results. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, pp. 158–165. IEEE (1991)
2. Martín, G., Marinescu, M., Singh, D., Carretero, J.: Leveraging social networks for understanding the evolution of epidemics. BMC Syst. Biol. 5(suppl. 3) (2011)
3. Xu, C., Lau, F.: Load balancing in parallel computers: theory and practice. Kluwer Academic Publishers (1997)
4. Beltran, M., Guzman, A., Bosque, J.: Dealing with heterogeneity in load balancing algorithms. In: ISPDC 2006, pp. 123–132. IEEE (2006)
5. Belikov, E., Loidl, H., Michaelson, G., Trinder, P.: Architecture-aware cost modelling for parallel performance portability. In: Kolloquium Programmiersprachen und Grundlagen der Programmierung 5
6. Galindo, I., Almeida, F., Badía-Contelles, J.M.: Dynamic load balancing on dedicated heterogeneous systems. In: Lastovetsky, A., Kechadi, T., Dongarra, J. (eds.) EuroPVM/MPI 2008. LNCS, vol. 5205, pp. 64–74. Springer, Heidelberg (2008)
7. Martínez, J., Almeida, F., Garzón, E., Acosta, A., Blanco, V.: Adaptive load balancing of iterative computation on heterogeneous nondedicated systems. The Journal of Supercomputing 58(3), 385–393 (2011)
8. Weatherly, D., Lowenthal, D., Nakazawa, M., Lowenthal, F.: Dyn-MPI: Supporting mpi on medium-scale, non-dedicated clusters. Journal of Parallel and Distributed Computing 66(6), 822–838 (2006)
9. Bohn, C., Lamont, G.: Load balancing for heterogeneous clusters of PCs. Future Generation Computer Systems 18(3), 389–400 (2002)
10. Lastovetsky, A., Reddy, R.: HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. Journal of Parallel and Distributed Computing 66(2), 197–220 (2006)
11. Huang, C., Lawlor, O., Kale, L.: Adaptive MPI. In: Rauchwerger, L. (ed.) LCPC 2003. LNCS, vol. 2958, pp. 306–322. Springer, Heidelberg (2004)
12. Dongarra, J., Malony, A., Moore, S., Mucci, P., Shende, S.: Performance instrumentation and measurement for terascale systems. In: Sloot, P.M.A., Abramson, D., Bogdanov, A.V., Gorbachev, Y.E., Dongarra, J., Zomaya, A.Y. (eds.) ICCS 2003, Part IV. LNCS, vol. 2660, pp. 53–62. Springer, Heidelberg (2003)
13. Martín, G., Marinescu, M., Singh, D., Carretero, J.: FLEX-MPI - Technical Report. Technical report, Universidad Carlos III de Madrid (2012), `http://www.arcos.inf.uc3m.es/~desingh/publications.html`
14. Mucci, P., Browne, S., Deane, C., Ho, G.: PAPI: A portable interface to hardware performance counters. In: Proceedings of the Department of Defense HPCMP Users Group Conference, 7–10 (1999)
15. Davis, T.A., Hu, Y.: The University of Florida sparse matrix collection. ACM Trans. Math. Softw. 38(1), 1:1–1:25 (2011)