Basic Schemes to Exploit Loop-level Parallelism on the Cell Broadband Engine*

José L. Abellán, Juan Fernández, and Manuel E. Acacio

Dept. de Ingeniería y Tecnología de Computadores
Facultad de Informática - Universidad de Murcia
30100 Murcia, Spain
{jl.abellan,juanf,meacacio}@ditec.um.es

Abstract. CMP architectures provide very high performance and are more energy-efficient than their contemporary single-core. However, programmers must be aware of a wide variety of communication and synchronization mechanisms in order to efficiently exploit thread-level parallelism (TLP). In this work, we present a workload scheduler for CMP architectures to achieve static and dynamic loop-level parallelism by load-balancing work among all available cores, in order to minimizes idle time. As present and future work, we are extending our proposal to also tackle task-level parallelism. In particular, the CMP architecture under consideration in this work is a dual Cell-based Blade. This platform contains two Cell BEs providing a number of communication and synchronization mechanisms. In addition, it enables inter-Cell communications which further increases the complexity of making efficient load-balancing.

1 Introduction

CMPs provide applications with an opportunity to achieve much higher performance than contemporary uniprocessor systems. However, in order to harness the additional compute resources of a CMP, applications must expose their thread-level parallelism to the underlaying hardware [1].

In this paper, we present a workload scheduler for CMP architectures, that focuses on applications exhibiting loop-level parallelism. We evaluate both static and dynamic distributions of the iterations that conform the loop. We can also select two execution models for all iterations of the loop: computation model, and computation-communication model which also executes transfers of data. As present and future work, we are extending the scheduler to consider more common execution models and also tackle task-level parallelism.

In particular our target machine is a dual Cell-based blade, which is composed of two separate Cell BEs linked together through the EIB. The main components of this hardware configuration are shown in Figure 1. In this architecture the

* This work has been jointly supported by the Spanish MEC and European Comission FEDER funds under grants "Consolider Ingenio-2010 CSD2006-00046" and "TIN2006-15516-C04-03".

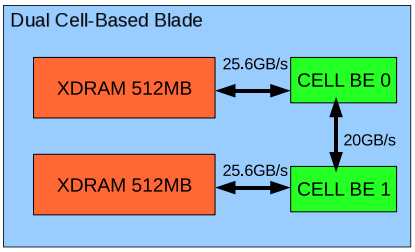


Fig. 1. Block Diagram of a Dual Cell-Based Blade.

two Cell BEs operate in SMP mode with full cache and memory coherency. In [2] we show the communication asymmetries exhibited by this platform, that arise when go through the Cell-to-Cell interface. This fact further increases the complexity of achieving efficient load-balancing.

2 Workload Scheduler

The workload scheduler supports static and dynamic loop-level parallelism, and two different SPE execution models. Moreover, it is possible to define the number of SPEs that will intervene, the total number of iterations and some other parameters for each iteration of the loop, that define the particular execution model being considered. However, the process to launch, execute and synchronize the threads is the same for all characterizations. First, the user sets an specific scheduler configuration by assigning values to an input configuration file. After that, the PPE reads the configuration file and marshalls a structure, which is allocated in main memory, called `InfoTask`. Next, the PPE creates as many threads as specified by the user. After that, each SPE reads its particular configuration from `InfoTask`, and synchronizes with the PPE in order to start the execution. Next, the PPE starts a function to measure the elapsed execution time of each SPE. Then, each SPE executes its iterations and meanwhile, the PPE waits for completion through event management notifications provided [3]. At the end of the loop execution, each SPE writes in its outgoing interrupt mailbox to generate an interrupt which is received by the PPE through a new event. Finally, the PPE gathers all execution times for each SPE, which are calculated from the times of incoming events from SPEs' outgoing interrupt mailbox writing.

2.1 Loop-level parallelism

Loop-level parallelism refers to situations where the iterations of a loop can be executed in parallel on as many threads as available cores. This parallelism is easy to expose because often requires little more than identifying the loops whose iterations are meant to be executed in parallel. In this way, iterations can be statically or dynamically distributed across SPEs.

Static scheduling In this type of scheduling, the PPE divides the total number of iterations by the number of SPEs, so that all SPEs execute the same number of iterations.

Dynamic scheduling In this scheme, each SPE asks the scheduler for a subset of the total number of iterations. To do that, a shared variable in main memory is updated through atomic operations. At the beginning, the PPE stores the total number of iterations, the number of iterations to fetch (`ITERATION_FETCH`) and the memory address of the shared variable (`sh_var`) in `InfoTask`. At SPE execution time, SPEs execute `fetch&add(sh_var, ITERATION_FETCH)`. This operation performs an atomic operation on `sh_var` by atomically adding it the number of iterations specified by `ITERATION_FETCH`. This process is repeated until the atomic operation returns a number of iterations greater than or equal to the total number of iterations.

2.2 Execution models

In the current version of the scheduler, there are two types of loop-based execution models: Computation Model and Computation-Comunication Model.

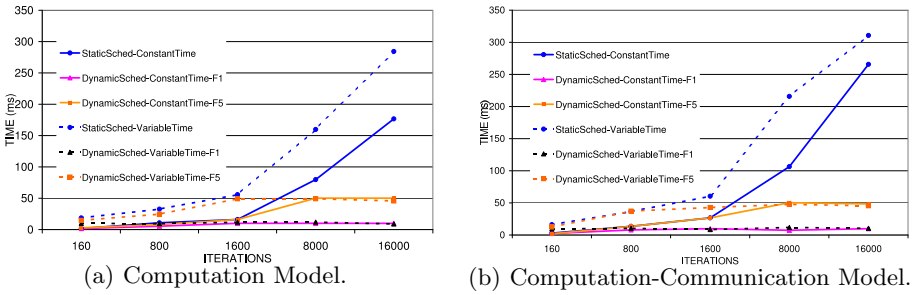
Computation Model In this model, each SPE only performs computation in each iteration. Therefore, the user can specify in the input file, the compute time in milliseconds, and also the variability of compute time in terms of random values between the maximum time (`MAX_COMPUTE_TIME`) and the minimum time (`MIN_COMPUTE_TIME`).

Computation-Communication Model In this model, each SPE not only performs computation, but also transfers data from and to main memory. Communication and computation are overlapped through the double-buffering technique implemented by SPEs. The user can specify the same parameters than in the Computation Model and the size of the buffers (`BUFFER_SIZE`) used by the double-buffering.

3 Evaluation and Conclusions

In this section, we carry out a performance analysis of the execution models mentioned in Section 2.2. To obtain the experimental results, we installed the IBM SDK v2.1 atop Fedora Core 6 on a dual Cell-based IBM BladeCenter QS20.

In Figures 2(a) and 2(b), we plot the difference (in milliseconds) between the time taken by the first SPE to complete its task and the last SPE completion time for the two execution models previously discussed. In both cases we assume that the 16 SPEs participate in the computations, and for the Computation-Communication Model, that transfers of 4KB are performed. We present results



(a) Computation Model. (b) Computation-Communication Model.

Fig. 2. Imbalance between first and last SPEs' times.

for static and dynamic scheduling algorithms (*StaticSched* or *DynamicSched* respectively), and for both 10 milliseconds constant computation time per iteration (*ConstantTime*) and variable computation time ranging between 8 and 12 milliseconds (*VariableTime*). Furthermore, in case of Dynamic Scheduling we plot two scenarios: one in which each SPE fetches 1 iteration, and other in which 5 iterations are assigned (F1 and F5 respectively). As we can see, the behavior is almost identical in both figures because, according to [2] the elapsed time to perform `dmaget` and `dmaput` operations among all SPEs to different memory locations is close to 4us, but computation time take 8ms or more in case of variable times. Therefore, the communications introduce insignificant overhead to the time of computation. Notice that, static scheduling reports higher times than dynamic scheduling. Because, in the first case each SPE executes a fixed number of iterations distributed by the PPE, finishes the execution and waits for the rest of SPEs to complete their assigned iterations. Nevertheless in case of dynamic scheduling, each SPE has not a fixed number of iterations to do. Then, SPEs only waits when the atomic operation returns a number greater than or equal to the total number of iterations, thus minimizing idle time among SPEs' executions.

As a result of this paper, it is shown that static scheduling introduces higher overhead than its dynamic counterpart. This is due to increasing the load-imbalance because of incrementing the granularity with respect to dynamic scheduling. But in general, choosing a particular scheduling model depends on the specific configuration of the model executions aforementioned.

References

1. Kumar, S., Hughes, C.J., Nguyen, A.: Carbon: Architectural Support for Fine-Grained Parallelism on Chip Multiprocessors. In: Proceedings of 34^{th} International Symposium on Computer Architecture, San Diego, California, USA (2007)
2. Abellán, J.L., Fernández, J., Acacio, M.E.: Characterizing the Basic Synchronization and Communication Operations in Dual Cell-Based Blades. In: Proceedings of 8^{th} International Conference on Computational Science, Kraków, Poland (2008)
3. IBM Systems and Technology Group: SPE Runtime Management Library Version 2.1. (2007)