High Performance Computing Applications using Parallel Data Processing Units

Keyvan Azadbakht, Vlad Serbanescu, and Frank de Boer

Centrum Wiskunde & Informatica, Amsterdam, Netherlands
{k.azadbakht,vlad.serbanescu,f.s.de.boer}@cwi.nl

Abstract. Multicore processors are growing with respect to the number of cores on a chip. In a parallel computation context, multicore platforms have several important features such as exploiting multiple parallel processes, having access to a shared memory with noticeably lower cost than the distributed alternative and optimizing different levels of parallelism. In this paper, we introduce the Parallel Data Processing Unit (PDPU) which is a group of objects that benefits from the shared memory of the multicore configuration and that consists of two parts: a shared memory for maintaining data consistent, and a set of objects that are processing the data, then producing and aggregating the results concurrently. We then implement two examples in Java that illustrate PDPU behavior, and compare them with their actor based counterparts and show significant performance improvements. We also put forward the idea of integrating PDPU with the actor model which will result in an optimization for a specific spectrum of problems in actor based development.

Keywords: Multicore Processors, High Performance Computing, Actor Based Implementation, Shared Memory, Programming Construct, Data Management

1 Introduction

In computer science research and industry, hardware development has always been progressing at a very fast rate in terms of performance and costs compared to the software adapted to run on it. Ever since the notion of parallel programming was first introduced, the demand for algorithms and models to support this paradigm has drastically increased. Very important issues like synchronization, concurrency and fine-grained task parallelism have been raised in a wide spectrum of domains requiring significant computing power and speed-up. Currently, chip manufacturers are moving from single-processor chips to new architectures that utilize the same silicon real estate for a conglomerate of multiple independent processors known as multicores, which is also the focus of our ongoing research in the UPSCALE European Project [26].

Throughout all of the mainstream languages, several libraries have been proposed with the objective to efficiently and reliably map tasks to these cores providing a high degree of parallelism to applications while avoiding race conditions and data inconsistency. At a lower level, compilers have also been adapted

to ensure instruction-level parallelism on operations that do not depend on each other and these optimizations are completely transparent to the user. Our ongoing research in this project focuses on how to "lift" this transparency to a higher level, offering an abstraction of task-level parallelism that allows the user to specify how and which tasks are executed in parallel without the complexity of monitoring data dependencies. We present this approach in one of the main programming languages, namely the Java language, while avoiding the need to learn specific instructions of particular Java libraries and packages or forcing the programmer to adopt a certain "pattern" to developing highly concurrent applications.

In a parallel computation context, multicore platforms provide some features like exploiting several parallel processes and having access to a shared memory that enables us to propose new higher level software abstractions containing both parallel processes and the memory which is shared among them, and encapsulating the before-mentioned low level coordination issues as one solid entity. In this direction, we introduce the Parallel Data Processing Unit (PDPU) as the elementary effort towards the elaboration of this category of software abstractions. In a nutshell, we may have multiple PDPUs in a software, each of which has its own memory which is shared among constituent processes running in parallel. In addition, the synchronization considerations caused by concurrent access to the shared memory are managed as internal features and are hidden from the programmers.

Through this solution we offer designers a reliable and efficient framework for avoiding race conditions, deadlocks and managing critical sections in their programs. We also allow them to analyze their code and identify the exact degree of parallelism and cost of their parallel sections, while making a clear separation between sequential and concurrent parts of their programs. Finally our solution focuses on how to optimize memory accesses by separate processes in a MIMD architecture [24, 25]. This is a crucial research question in the field of Computer Science as more and more computation intensive applications are moving to GRID environments or even further to CLOUD storage and resources. Therefore we formulate our main objective in this paper as follows: to introduce a new model for programming parallel data processing applications which encapsulates the multithreaded java programming model and its synchronization features. The model exposes an interface that is easy to use and transparent, while adding optimizations for efficient memory management and data consistency. In the rest of this paper, we first survey the related work in section 2. We then introduce the definition of PDPU in section 3. The implementation efforts and evaluation of PDPU will be addressed in section 4. In section 5, we put forward the idea of integrating PDPU with the actor model in order to take advantage of simplicity of higher level abstractions and better performance. Finally, we conclude the paper and present future works in section 6.

2 Related Work

In this section we look at several solutions proposed and developed in mainstream programming languages for adapting programs to run on multicores. We start from some of the basic concepts and examples that have been validated and used in research and industry for multicore programming. For each example we will look at what aspects are drawn into our solution, mainly the ease of use and readability of these solutions, as well as the drawbacks that we want to avoid in our approach. Furthermore, we look at some complex directions of research that are oriented towards memory management and mapping user-level threads to kernel-level threads and propose their integration into our model.

We first look into the kernel-level threads [23] which is a POSIX standard for programming in C, C++ and Fortran. The advantages of this standard are that, when implemented correctly, it is extremely efficient and fast, ensuring a high degree of parallelism that is specified by the user explicitly. The advantage of these threads is that they can be directly mapped to the kernel threads of an operating system making it very easy for the user to observe the load of each task and appropriately balance the computation amongst cores by correctly defining each thread's functionality and adjusting it according to its profiling results. It has been validated in numerous applications and has yielded the best scaling results among parallel programming solutions [18, 22]. The drawbacks of this approach are centered around the fact that the user is responsible for synchronization, avoiding race conditions or deadlocks and managing critical sections and variables. The POSIX Library offers no warnings, compiler errors or exceptions when these issues occur. In our solution we use the thread mechanism due to its excellent performance and offer a certain degree of control to the user, however some of the basic synchronization issues are handled implicitly and due to our solution being specific to the Java language, it offers the user exceptions on these issues if they are violated.

Another contribution to our proposal is related to the OpenMP standard proposed in [21]. This solution is also specialized in shared-memory programming and parallelism is fully implicit. It comprises of a set of directives used to control repetitive instructions in particular and allow them to be scheduled on the available cores such that they can be executed in parallel. The directives offer limited control over scheduling options, the degree of parallelism and critical sections. What is the most important aspect that we draw from OpenMP is the transparency of the parallelism, as it does not have to be explicitly specified [17]. Basic instruction-level parallelism and, starting with the recently passed OpenMP 3.0 standard, task-level parallelism are achieved by adding the appropriate directive before a repetitive instruction or a code-block.

A significant research topic related to OpenMP is how to use this standard together with the well-known Message Passing Interface (MPI) standard for distributed programming [16]. This solution allows the user to explicitly model parallel processes at a much higher programming level than POSIX Threads while at the same time handling remote or local communication via messages. The communication is completely transparent to the user and avoids the im-

plementation of sockets or remote method invocation. Essentially, every node is considered a separate entity with its own address space, a model which disregards shared memory. From a software engineering standpoint this standard is the easiest to use in the FORTRAN, C and C++ languages as it does not present any difficult language constructs and offers high level methods to handle, spawn, finalize and synchronize processes either on the same machine or on several computing entities. Communication transparency is the key feature that we introduce in our model from the MPI standard, but without affecting the shared memory model that an application may or may not be running on. A study in [20] has shown that a hybrid approach between OpenMP and MPI depending on the programming model can yield the best performance results and our solution is based on this hybrid approach.

Nobakht et al. [3] proposed a modeling language for leveraging performance and scheduling concepts to application level. The proposal introduces the notion of concurrent object groups (COGs) that isolates multiple objects into separate entities and exposes a user-friendly solution to set scheduling policies at a higher-programming level. As stated in our main objective, we model our solution for the Java language, therefore we needed to carefully study the parallel programming concepts introduced by the Java Platform[15]. The solutions for this programming language are similar to POSIX threads in the sense that the user is responsible for every step in the concurrent application's design. Although Java provides an abstraction for both Threads [14] and synchronization mechanisms [13] a programmer still has the difficult task of learning and using these new constructs being responsible for handling deadlocks, race conditions and data consistency. Our goal is to combine these solutions and their advantages to present a novel approach in multicore programming with a shared memory model. Our proposed setting is more general than coordination languages [32] in the sense that the data structures used for memory management can be customized and are not restricted to a specific tuple-space, with our aim being towards a more general data component.

3 The Definition of Parallel Data Processing Unit

Parallel Data Processing Unit (PDPU) is an abstract object (or unit) that puts together a group of objects so that they process the data in the shared memory concurrently. PDPU has two main constituents (Figure 1):

A group of processes: a frame that aggregates objects so that one can look at the group of objects as one solid entity. The group members and their corresponding details are abstracted away from the rest of the application (like a black box). Instead, PDPU provides one interface just like one coarse-grained object. There should be some reason that makes this frame meaningful e.g. conceptual coherence among active object classes. At least, all of them have one feature in common; they need a specific kind of data to process. From now on, we refer to the group members as processes since they process the data in shared memory,

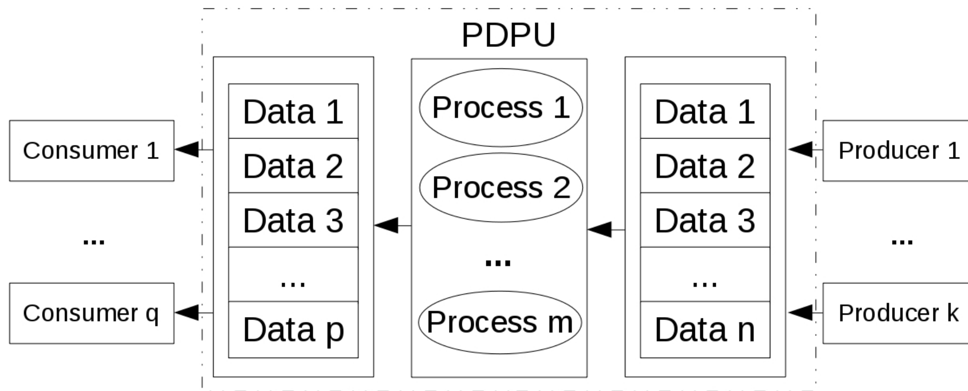

Fig. 1. General Perspective of PDPU

though they can also behave the same as producers by pushing data in the input shared memory through the interface.

Input and output shared memory: PDPU, as an individual object, has one input shared memory for storing data received from outside. This memory is shared within the processes, and the group processes the data from it. There is also an output shared memory which is filled with the data produced by processes. The processes do not share any memory except the input for reading and the output for writing. Furthermore, the data elements are just added or removed and they are immutable. We refer to those objects which are responsible to fill in the input shared memory as data producers. Producers are not a part of the PDPU, instead they use its interface in order to put the data. The output shared memory is also accessed by the objects called consumers through PDPU interface. Inside the PDPU, the shared memory is responsible for thread safety and data consistency when processes work with data concurrently.

3.1 PDPU Interface Description

As shown in Figure 1, there are a group of processes inside the PDPU. Each process must follow the following template:

```
Start Process ()
   Do
      data = retrieve()
      result = process(data)
      write(result)
   Until (data meets ending condition)
End Process
```

This abstract template shows how proactively processes obtain the data from the input shared memory and then process it based on their own logic. They

Table 1. PDPU interface

Method Name	Output Type	Method Description
`PDPU<InputType, OutputType>(Boolean, Runnable, int)`	Object	PDPU Consructor which generates PDPU with m individual processes from reproducible `Runnable` process and specified data retrieving policy (`isAll`). `InputType` is the input data type and `OutputType` is the output data type.
`PDPU<InputType, OutputType>(Boolean, List)`	Object	PDPU Consructor which generates PDPU with individual `Runnable` processes from `List` and specified data retrieving policy (`isAll`). `InputType` is the input data type and `OutputType` is the output data type.
`retrieve()`	InputType	Retrieve the data for process usage based on retrieving policy
`add(InputType)`	Boolean	Add the data to the shared memory and return True if it is successful
`write(OutputType)`	Boolean	Write the data to the output shared memory and return True if it is successful
`read()`	OutputType	Return the data from output shared memory

may, if necessary, generate a result and put it in the output shared memory. In the above mentioned code, there are two functions which are provided by PDPU interface: *retrieve* and *write*. A brief description of the PDPU interface is given in Table 1. The "retrieve()" method provides the process with the next data element from input shared memory. It encapsulates which is the next data element and how the synchronization issues are handled. The process may generate some explicit result for processing each data element. In this case it uses "write(data)" to record them in the output shared memory. This function also encapsulates the synchronization issues for writing in the output shared memory as well. The process's result, however, may be produced implicitly through the "process()" method, as you will see in section 4. On the other hand, there has to be a data producer (or producers) which fills the input shared memory and consequently provides the processes with the data to be retrieved and processed. To this aim, the producer uses the interface method "add(data)" which adds the data to the shared memory. This function is propagated by PDPU interface.

In the definition of PDPU, there are two factors that impact the internal design of it and both should be specified based on user's problem requirements.

- *Initialization phase:* initial state and logic of the processes.
- *Memory access:* the way that the data in shared memory will be retrieved.

The processes may be instances of the same class and their internal state may be the same after initialization. We refer to this type of processes as Reproducible (R) and otherwise as Non-Reproducible (NR). If the process is reproducible, it is enough to initiate one process and to send it to the PDPU object along with its number of replicas. Otherwise, the programmer is supposed to make a list

of processes with different initial states, and then to send the list to the PDPU object.

```
(1)   PDPU(isAll, process, m) //for Reproducible processes
(2)   PDPU(isAll, processList) // For Non-Reproducible processes
```

Furthermore, the data elements in the shared memory can have two different ways of being retrieved. The shared memory's data is targeted either to **all** processes or to **any** of them. In the first instance, it means all the m processes will retrieve one specific data, and in the second case it means that it is enough for the data to be processed just by one of the processes. An example of how PDPU is used is given in Listing 1.1. The processes inside PDPU retrieve the next data element as soon as they become idle. The start-up sequence of the processes is also based on which one retrieves data from the input shared memory earlier.

Listing 1.1. PDPU User Example

```
public class PDPUUser {

  Process[] process;
  PDPU<InputType, OuputType> pdpu;
  Input dataFlux = new input();
  //this can be a socketHandler
  //or anything which continuously receives Data

  public void Init(int m) {
    process = new process[m];
    pdpu = new PDPU<InputType, OuputType>("all", process);
    //the first parameter corresponds to how data is processed

    for (int i = 0 ; i < m; i++){
      process[i] = new Process();
      process[i].setPDPU(pdpu);
    }

    while(dataFlux.hasData())
      pdpu.add(dataFlux.getNewData());
      //The producer rule in the model

    for (int i = 1 ; i <= m; i++)
      pdpu.add(dataFlux.endData());
      // ending condition
  }
}

public class Process implements Runnable{

  PDPU<InputType, OutputType> pdpu;

  public void setPDPU(PDPU<InputType, OutputType> pdpu){
```

```
    this.pdpu = pdpu;
  }

  public void run() {
    InputType x;
    OuputType y;

    while (x != END_DATA){
      x = pdpu.retrieve(i);
      y = process(x);
      pdpu.write(y);
    }

    OutputType process(InputType x){} // process related code
}
```

3.2 PDPU scientific impact

PDPUs bring about some advantages with respect to software engineering qualities, as they provide:

- *Ease of use*: some lower level implementation details are handled not by programmers, but by the PDPU; like allocating computation resources to the processes, allocating shared memory, and the thread-safety issues concerning access to the shared memory.
- *Understandability of system design and code*: PDPU as a coarse granular cohesive design component (or module) makes the design models of the system simpler and easier to understand.
- *Loose coupling*: the producer interacts only with PDPU interface instead of all processes.

Furthermore with respect to concurrent computation, PDPUs provide new abstract constructs for parallel programming languages and, more generally, they put forward a new paradigm of designing coarse grained objects which encapsulates both memory and multiple computation resources.

4 PDPU Implementation and Evaluation

In this section we present a technical explanation of the shared memory management. We illustrate the operation of PDPU through a case study with two examples in Java.

4.1 Memory management

As explained in section 3 the PDPU hides all of the elements concerning synchronization, thread-safety and data consistency from the user. The memory can

be customized to support retrieval of each data item by either **a single** process or **all** processes. In the first case, we implement the memory as a LinkedBlockingQueue[13] which translates to a classic producer-consumer [12] problem with the synchronization hidden from the user. The real issue appears in the second instance where the memory must take into account blocking all processes when new data is not available, releasing them for work when new data is added and cleaning up the input data when all processes have read it. This case maps to the classic readers/writers problem [11] with a garbage collection issue. We use a ConcurrentHashMap with an index for a key and a counter/data pair for a value. A CyclicBarrier is assigned to a separate counter (the current number of items that were added up to a moment) which blocks all processes when no new data is available. After each process reads an item from the hash map it increments the item's corresponding counter. The last process to read an item is responsible for eliminating the item based on the counter reaching the fixed number of processes. It is worth mentioning that because according to the definition of PDPU the data is immutable both in input and output shared memories, we use object cloning in order to have a copy of the data instead of reference to guarantee the immutability of the data.

4.2 Example 1: The Behavior of PDPU with Respect to Transferring Data

We first compare a Java program in the domain of actor based applications with its PDPU-based counterpart. In this example, there is no computation and the only important factor is delivering data elements to the processes. Figure 2 presents the PDPU in Object Oriented model. For implementing above mentioned actor based Java programs, we use ABS-API [7], an actor-model library implemented in JAVA 8 using the newly introduced feature of lambda expressions. We also run all the programs on SaraSURF cluster on a 16 core processor 2.70 GHz (Intel Xeon CPU E5-4650 0) with 128GB of memory[8] to have the same framework for comparison.

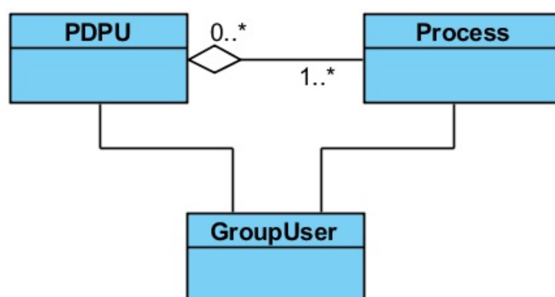

Fig. 2. The Object Oriented model containing PDPU

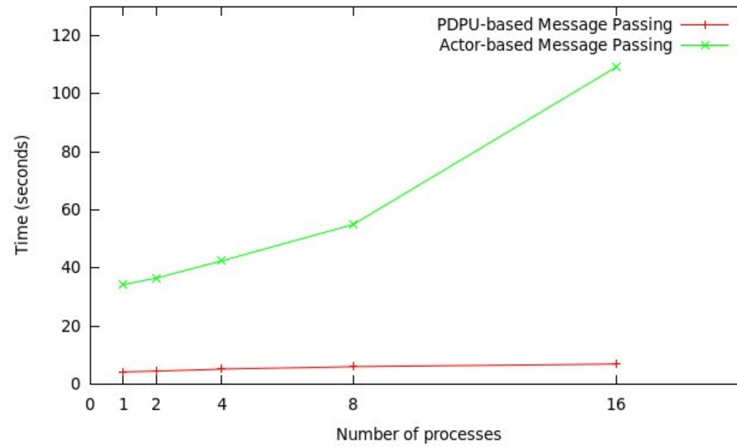

Fig. 3. The comparison between the behavior of Actor based and PDPU based Java programs. 10^6 data items are sent to each process.

Let us assume a common actor based configuration in which there is a producer of data that provides multiple actors with a stream of data objects through message passing. The producer generates the data and sends it through asynchronous method invocation. Each data element will be processed by all of the actors. Therefore the producer composes and sends m messages for each data, where m is the number of actors receiving the message. Thus, for n data elements, the producer sends $m * n$ messages. However, instead of broadcasting each message to all actors, the PDPU alternative for this implementation involves having a PDPU which contains a fixed number of m processes. The producer is supposed to add each data element to the shared memory of the PDPU instance just once through calling "add(data)" and processes retrieve and process the data. Therefore the producer adds the n data elements to the PDPU which starts m processes. It is clear that, in terms of transferring data to the processes, the computation complexities of producers are $O(nm)$ and $O(n+m)$ for the actor based and PDPU based implementations respectively. The difference is more clear, when we consider large m because of the future of multicore platforms, namely manycores, with thousands or even millions of cores on a chip. Figure 3 illustrates the advantage of PDPU in terms of performance. The important point of this plot is the behavior of these two approaches, disregarding the elapsed times. As you can see, the line corresponding to actor based implementation grows exponentially, when m is increased. The PDPU based implementation, however, grows at a linear rate. There are multiple factors other than computation complexity that impact on the elapsed time in the actor based configuration, namely, the resource consumption due to among others enqueueing and dequeueing the messages and generating the call stacks.

4.3 Example 2: The concurrent version of sieve of Eratosthenes

In mathematics, the sieve of Eratosthenes, one of a number of prime number sieves, is a simple algorithm for finding all prime numbers up to any given limit. It does so by iteratively marking as composite (i.e. not prime) the multiples of each prime, starting with the multiples of 2 [6]. To model the algorithm in two different versions using actors [5] and PDPU, we use the well-known parallel algorithm which partitions the sequence of candidate numbers [2, 1]. In this algorithm, the numbers are partitioned into smaller sequences of numbers with almost the same size. The size of each partition must be equal or greater than $\lfloor\sqrt{n}\rfloor$, and the number of partitions must be equal or less than $\lceil n/\lfloor\sqrt{n}\rfloor\rceil$, where n is the target number such that the first partition contains all of the prime numbers that sieve composites throughout all partitions. Therefore the first actor in the model, namely producer, will send asynchronous messages to the others that will invoke the sieving process. To this aim, each prime number must be sent m times to the m actors. This is where the PDPU based model affects the performance of the program. If prime numbers are processed on the same machine they can exploit PDPU abstraction which reads and writes the numbers in a shared data structure, with message passing being required only between the producer and remote partitions. So each prime number is produced and written in shared memory just once. The comparison of actor based and PDPU based implementations of prime sieve is shown in Figure 4. While both algorithms scale, PDPU based implementation outperforms the actor based one. However it is not because of theoretical analysis we have mentioned in Section 4.2, but because of practical overheads in ABS-API for receiving the messages containing data. In other words, when the producer is not the bottleneck, in opposite to example 1, then both programs performances are limited to the computations done in actors and processes. In this example, they behave the same, though there is some constant difference in performance. In contrast, if we consider just producers overheads and disregard other parts of programs, the PDPU based implementation significantly outperforms since the same reason mentioned in Section 4.2.

5 Discussion: Integrating PDPU with Actor Model

In previous sections, we introduced PDPU as a high abstraction level object that is orthogonal to both Java Threads and the actor model. PDPU enhances the Java language to obtain parallelism in processing data via encapsulating synchronization. In other words, it is simpler to implement a parallel data processing algorithm using PDPU than using Java Threads, and the programmer does not need to face synchronization issues, race conditions and so forth. We have also shown how PDPU outperforms actor based implementations and what is the reason behind it through the examples in section 4. In this section, we explain how PDPU concept is integrated with actor model in such a way that the extended actor model exploits the strength points of both concepts.

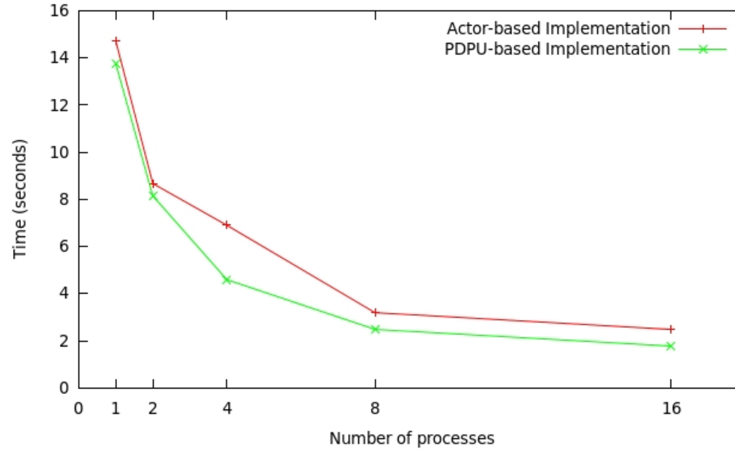

Fig. 4. The comparison between the behavior of Actor based and PDPU based implementations of Sieve of Eratosthenes. The target number for sieving is 10^9

Although it is achieved novel advantages in the area of actor model and asynchronous message passing, there are some downsides in practice e.g. the overhead of composing, sending and receiving messages and dealing with obtaining future results when their number is large. Sometimes the overhead is because of the nature of broadcasting mechanism — that is — broadcasting the same message to several actors which has both computation overhead, because of redundant repetitive actions for broadcasting message, and memory overhead, because of redundant queuing the same message by several actors. On the other hand, the actor model [7] provides the *actor* notion which is an entity with high abstraction level that leads to eliminating design and implementation complexities caused by the nature of parallel computation and programming. However, there can be higher abstract constructs or design patterns that still follow the actor concept and eliminate or lessen these cumbersome and confusing details in some specific circumstances.

This section puts forward the idea of a new abstract programming language construct, which is called **Active Group** in order to benefit from both actor based implementation and PDPU features. The general idea is that we have one actor-like component in a higher level abstraction. This component consists of one queue of runnable messages and one processing part which processes the messages. At lower abstraction level, similar to PDPU, the processing part contains multiple processes, i.e. actors, receiving messages. Instead of data elements in PDPU, here we will have runnable messages in the shared memory. You can see a simple scheme of Active Group in Figure 5. However we ignore some of its details, e.g. how actors have access to the shared queue. Here we briefly address some of these details as the main features of Active Groups. We use the same terminology as the definition of PDPU:

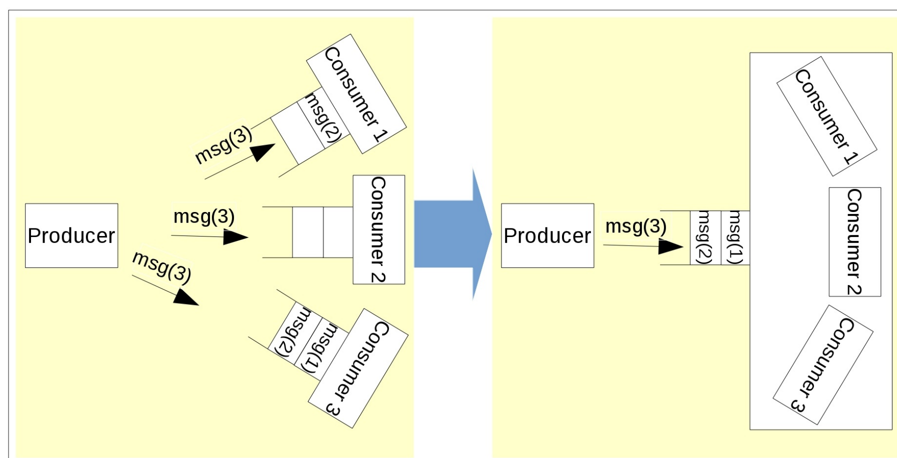

Fig. 5. Active Group (right side) and its extended actor based counterpart (left side)

Proactive or reactive actors: The actor can proactively fetch one message from shared memory or wait for the next element to be prepared. In contrast, it may passively receive the message through an independent scheduler in the group, and process it. Thus the values for this feature are **Proactive** and **Reactive**. In the former case, there is no need for each individual actor to have a message queue.

Different scheduling policies: One of the advantages of the active group is to apply different scheduling policies. Data is usually received in a particular order, but if the policy is not FIFO, the group may process them in other orders. To this aim, data can be reordered based on, say, **Priority** or **Content**. Furthermore, if actors are not proactive, they are supposed to be managed as computation resources by schedulers. They may be selected as the target actor based on different policies like **Round Robin**.

Different ways of using the shared memory: In some applications of active groups, each message in the shared memory will be fetched by **one** actor and processed by it. So, in that case, the message is removed from shared memory (e.g. a queue of HTTP requests, each of which will be enough to be processed by one server). This type of shared memory is technically a queue. However, in some other applications, the message is used by **all** actors. In that case, the data is read from the shared memory but it is not removed since it will be used by others (e.g. in section 4.3, the messages containing prime numbers that are used by actors in concurrent version of sieve of Eratosthenes).

There can be several distinct versions of active group, each of which have different feature values. If we refer to the above-mentioned issues as customizable features of active group, then the ideal active group definition provides the user with all these features to be customizable with existing values. To reach this aim, there can be different ways:

1. Parameterizing the active group construct so that the user can initialize appropriate feature values.
2. Using polymorphism to refine the abstract active group.
3. Having distinct types of active groups so that their definition illustrates their features values.

6 Conclusion and Future Works

In this paper we proposed a coarser granular object, i.e. Parallel Data Processing Unit, which contains both computation and memory, and encapsulates efforts for synchronization issues to make parallel programming easier for programmers. Through case studies and complexity analysis, we have shown how it overcomes one of the drawbacks of actor model and significantly improves performance.

Reasoning about multi-threaded Java programs is notoriously hard (see [9]) because of its fine-grained interleaving. In contrast PDPUs allow for a compositional proof method along the lines of the proof method for monitors as introduced in [10]. Given an appropriate assertion language for describing the internal data structures of a PDPU such a proof method is based on the specification of these data structures by means of an *invariant*. The external proof obligations for the invariant are specified in terms of the implementations of the "add" and the "read" operation, given a precondition of the caller specifying the input parameter in case of an "add" operation. The internal proof obligations of the invariant are specified in terms of the implementations of the processes which involves the implementations of the "retrieve" and "write" operations.

We then put forward the idea of integrating this novel approach with the actor model by bridging the gap between their conceptual differences. To this aim, we have generalized PDPU as a new concept, called Active Group, which is based on actor model. It will make it possible for a broader types of problems to be implemented in Active Groups. As future work, we aim to extend the syntax and the operational semantics of ABS language [4] to have the new construct, namely Active Group, and also extend the ABS-API so that it contains support for defining and using Active Groups.

7 Acknowledgements

Partly funded by the EU project FP7-610582 ENVISAGE: Engineering Virtualized Services (`http://www.envisage-project.eu`). Partly funded by the EU project FP7-612985 UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations (`http://www.upscale-project.eu`). This work was carried out on the Dutch national e-infrastructure with the support of SURF Foundation.

References

1. Pop, Florin, and Maria Potop-Butucaru. "Adaptive Resource Management and Scheduling for Cloud Computing."

2. Serbanescu, Vlad, et al. "Towards Type-Based Optimizations in Distributed Applications Using ABS and JAVA 8." Adaptive Resource Management and Scheduling for Cloud Computing. Springer International Publishing, 2014. 103-112.
3. Nobakht, Behrooz, et al. "Programming and deployment of active objects with application-level scheduling." Proceedings of the 27th Annual ACM Symposium on Applied Computing. ACM, 2012.
4. Johnsen, Einar Broch, et al. "ABS: A core language for abstract behavioral specification." Formal Methods for Components and Objects. Springer Berlin Heidelberg, 2012.
5. Nobakht, Behrooz, and Frank S. de Boer. "Programming with actors in Java 8." Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications. Springer Berlin Heidelberg, 2014. 37-53.
6. Bokhari, Shahid H. "Multiprocessing the sieve of Eratosthenes." Computer 20.4 (1987): 50-58.
7. Nobakht, Behrooz, and Frank S. de Boer. "Programming with actors in Java 8." Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications. Springer Berlin Heidelberg, 2014. 37-53.
8. SurfSara https://surfsara.nl/
9. brahm, Erika, et al. "An assertion-based proof system for multithreaded Java." Theoretical Computer Science 331.2 (2005): 251-290.
10. Hoare, Charles Antony Richard. "Monitors: An operating system structuring concept." Communications of the ACM 17.10 (1974): 549-557.
11. Andrews, Gregory R. Concurrent programming: principles and practice. Benjamin/Cummings Publishing Company, 1991.
12. Li, S., et al. "Analysis of the producer-consumer problem." Journal of Large-Scale Archetypes, vol. 0 (2002): 72-92.
13. Lea, Doug. "The java. util. concurrent synchronizer framework." Science of Computer Programming 58.3 (2005): 293-309.
14. Oaks, Scott, and Henry Wong. Java threads. " O'Reilly Media, Inc.", 1999.
15. http://docs.oracle.com/javase/7/docs/api/overview-summary.html
16. Gropp, William, Ewing Lusk, and Anthony Skjellum. Using MPI: portable parallel programming with the message-passing interface. Vol. 1. MIT press, 1999.
17. Pop, Antoniu, and Albert Cohen. "OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs." ACM Transactions on Architecture and Code Optimization (TACO) 9.4 (2013): 53.
18. Briggs, Emil, et al. "DFT-Based Electronic Structure Calculations on Hybrid and Massively Parallel Computer Architectures." Bulletin of the American Physical Society (2014).
19. Asai, Ryo, and Andrey Vladimirov. "Intel Cilk Plus for Complex Parallel Algorithms:" Enormous Fast Fourier Transform"(EFFT) Library." arXiv preprint arXiv:1409.5757 (2014).
20. Rabenseifner, Rolf, Georg Hager, and Gabriele Jost. "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes." Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on. IEEE, 2009.
21. Dagum, Leonardo, and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." Computational Science & Engineering, IEEE 5.1 (1998): 46-55.
22. Gravvanis, G. A., et al. "A note on parallel finite difference approximate inverse preconditioning on multicore systems using POSIX threads." International Journal of Computational Methods 10.05 (2013).

23. Mueller, Frank. "A Library Implementation of POSIX Threads under UNIX." USENIX Winter. 1993.
24. Snyder, Lawrence. A taxonomy of synchronous parallel machines. WASHINGTON UNIV SEATTLE DEPT OF COMPUTER SCIENCE, 1988.
25. Johnson, Eric E. "Completing an MIMD multiprocessor taxonomy." ACM SIGARCH Computer Architecture News 16.3 (1988): 44-47.
26. UPSCALE European Project, http://www.upscale-project.eu/
27. Smith, T.F., Waterman, M.S.: Identification of Common Molecular Subsequences. J. Mol. Biol. 147, 195–197 (1981)
28. May, P., Ehrlich, H.C., Steinke, T.: ZIB Structure Prediction Pipeline: Composing a Complex Biological Workflow through Web Services. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 1148–1158. Springer, Heidelberg (2006)
29. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (1999)
30. Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid Information Services for Distributed Resource Sharing. In: 10th IEEE International Symposium on High Performance Distributed Computing, pp. 181–184. IEEE Press, New York (2001)
31. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: an Open Grid Services Architecture for Distributed Systems Integration. Technical report, Global Grid Forum (2002)
32. Gelernter, David, and Nicholas Carriero. "Coordination languages and their significance." Communications of the ACM 35.2 (1992): 96.
33. National Center for Biotechnology Information, http://www.ncbi.nlm.nih.gov