Lock-Free Parallel Computing Using Theatre

Christian Nigro[1] and Libero Nigro[2(✉)]

[1] Independent Computing Professional, Rende, Italy
christian.nigro21@gmail.com
[2] DIMES, University of Calabria, Quattromiglia, Italy
l.nigro@unical.it

Abstract. Theatre is a control-based actor system currently developed in Java, whose design specifically addresses the development of predictable, time-constrained distributed systems. Theatre, though, can also be used for untimed concurrent applications. The control structure regulating message scheduling and dispatching can be customized by programming. This paper describes a novel implementation pTheatre (Parallel Theatre), whose control structure can exploit the potential of parallel computing offered by nowadays multi-core machines. With respect to the distributed implementation of Theatre, pTheatre is more lightweight because it avoids the use of Java serialization during actor migration, and when transmitting messages from a computing node (theatre/thread) to another one. In addition, no locking mechanism is used both in high-level actor programs and in the underlying runtime support. This way, common pitfalls related to classic multi-threaded programming are naturally avoided, and the possibility of enabling high-performance computing is opened. The paper demonstrates the potential of the achieved realization through a parallel matrix multiplication example.

Keywords: Actors · Reflective control structure · Theatre · Java · Lock-free data structures · Multi-core parallel computing · Dense matrix parallel multiplication

1 Introduction

Actors [1] represent a well-established formal computational model for the development of general concurrent distributed systems. The model is founded on the concept of an *actor*, which is a basic, modular, and isolated software entity that shares no data and communicates with its peers solely by *asynchronous message passing*. In its basic formulation, the actor model features one thread per actor and admits an input mailbox where all the incoming messages get stored. It is the responsibility of the thread that of extracting, if there any, one message at a time from the mailbox and processing it by updating the local data status and possibly creating new actors and sending messages to known actors (*acquaintances*) including itself. Message processing is *atomic* and follows the *macro-step semantics* [2]: a new message can only be processed when current message processing is finished. In the last years, the actor model emerged as a valid

X.-S. Yang et al. (eds.), *Proceedings of Fifth International Congress on Information and Communication Technology*, Advances in Intelligent Systems and Computing 1183, https://doi.org/10.1007/978-981-15-5856-6_65

alternative [3] to classical multi-threaded programming with data sharing and the use of *locks*, which are notoriously prone to subtle synchronization errors and deadlocks [4].

Actors are currently supported by such frameworks as Salsa, ActorFoundry, Scala/Akka, Erlang, CAF, Rebeca, etc. some of which have been experimented in the construction of large web-based software systems.

Classical actors, though, are more suited to untimed systems where non-determinism regulates message delivery. Extensions to actors have been defined to allow modelling, analysis and implementation of distributed time-constrained systems [5–8].

Theatre [6–8] distinguishes from similar actor-based frameworks by its control-centric character, which in turn favours time predictability and model transformation in development, thus facilitating a smooth transition from analysis down to design, prototyping and final, model compliant, implementation of a system. A general, distributed implementation of Theatre was achieved in pure Java and it is described in [9]. The possibility of supporting hybrid actors during modelling and analysis of cyber-physical systems has recently been shown in [10, 11].

The work described in this paper argues that Theatre can effectively also be used for parallel untimed applications. The paper specifically proposes Parallel Theatre (pTheatre), an original realization of Theatre which optimizes the implementation in [9] so as to enable high-performance computing on today's multi-core machines.

The rest of this paper is structured as follows. In Sect. 2, the basic concepts of Theatre are briefly presented. Then the design and lock-free implementation of pTheatre are discussed in Sect. 3. Section 4 proposes a scalable application of pTheatre to parallel matrix multiplication. Section 5 discusses the performance issues of the case study. Finally, Sect. 6 concludes the paper with an indication of future work.

2 The Theatre Actor System

2.1 Programming Concepts

Theatre actors have no internal thread and no local mailbox. Rather, actors execute on computing nodes said theatres. In a theatre, a *reflective control layer* (control machine) is used which, transparently, captures sent messages, buffers them and regulates message delivery according to a strategy that ultimately depends upon the application requirements. The control machine can reason on time (simulated time or real-time). A library of control machines was developed (see [9]). An Actor is at rest until a message arrives. Message processing is atomic and cannot be pre-empted nor suspended. In Java, a Theatre actor is programmed by a normal class that derives from the Actor base class. Selected methods (said *message servers -msgsrv-* as in [5]) specify how corresponding messages will be responded. For initialization purposes (see also Fig. 1), instead of relying on class constructors, an explicit message like init() can be used. Differently from normal Java methods, which have a synchronous invocation semantics, message servers can only be invoked by a non-blocking *send* operation, which can carry timing information [8, 9]. Therefore, message servers have an *asynchronous invocation semantics*. Message servers can have arguments but have no return type. To return a result, a message server must send back to its requestor an explicit message with the result as an argument. As a basic assumption, a message server is expected, normally, to have a negligible time

duration. The control machine of a theatre provides the execution context (stack frame) to local actors. Therefore, *concurrency*, within a theatre, is *cooperative* not pre-emptive, and naturally emerges from *message interleaving*. The control machine repeats a *basic control loop*. At each iteration, a pending message is selected and dispatched to its target actor by invoking a corresponding message server. At message server termination, the control loop is re-entered, any sent messages scheduled, then the next iteration is started. The control loop is actually continued until a final condition eventually holds: for example, in a concurrent application, message exchanges are exhausted; in a simulation application, the assigned simulation time limit is reached. A real-time application, instead, typically executes in a non-stop way. A time-sensitive control machine ensures timestamped sent messages are delivered in timestamp order. When messages have the same timestamp, a Lamport's logical clock is used (added meta-data in messages) to deliver messages according to their logical clock (*generation time*). A concurrent control machine ultimately delivers messages according to their sending order. Controlling the message delivery order contributes to the determinism of a Theatre application.

2.2 System-Level Concepts

A distributed Theatre system is a federation of theatres (logical processes, e.g. JVM instances or address spaces) currently implemented in Java [9]. Each theatre hosts a *transport layer*, a *control layer* (control machine) and a *collection of local* application *actors*. Theatre is based on *global time*. A *time server* component can be used (attached to a given theatre) to ensure global time is kept updated. Theatres (their control machines) coordinate each other with the time server, to preserve the notion of global time. The transport layer can be based, in a case, on the TCP transport, thus ensuring sent messages from a theatre and directed to the same destination theatre are received in first-in-first-out order. Whereas concurrency in the same theatre is cooperative, actors (message servers) belonging to distinct theatres, allocated, e.g. to distinct physical processors of a multicomputer, can be executed in truly *physical parallelism*. At initial configuration time, a *socket network* is established among theatres which, in a case, can be a complete mesh. Required socket connections are specified in an xml configuration file, which also declares the control machine type to instantiate on each theatre. A master theatre is elected which actually supervises the creation of the socket network and bootstraps system execution by creating and initializing first actors. In a Theatre system, actors are assumed to have *universal names*, that is unique system-level identifiers (strings). Actors are created in a theatre, then they can be migrated (*move*() operation) on a different theatre for load-balancing issues. Inter-theatre messages and migrated actors use the Java serialization mechanism. Intra-theatre messages, instead, are normal Java objects. When an actor is migrated, a *proxy* version of itself is kept in the originating theatre, which stores the last known address (theatre URL) of the migrated actor, and acts as a message *forwarder*. Dispatching a message to a proxy actor, causes an external message to be generated and routed to its destination through the transport layer. A *local actor table* (LAT) is maintained into each theatre to store information about proxy/actual actors. When a migrated actor comes back to a theatre where a proxy version of it exists, the proxy gets replaced by the actual actor along with its full data status. Messages exchanged by theatres can be *control messages* or *application messages*. Application

messages refer to the core business of the system. They are dispatched and processed by actors. Control messages, instead, are transparent to actors. They are intended to be received and processed by the control machine themselves. Control messages are used to start/stop a federation system, to interact (according to a protocol) with the time server to keep global time aligned, etc.

3 Implementing pTheatre in Java

pTheatre (Parallel Theatre) represents an optimization of basic distributed Theatre, aimed at enabling high-performance parallel computing on multi-core machines. The following are some main points about the design and implementation in Java of pTheatre.

1. Theatres are mapped onto Java threads (one thread per theatre).
2. A Theatre system coincides with one JVM instance. Therefore, all theatres of a parallel application share a common address space.
3. Actor universal naming reduces to the use of Java actor references. An actor's reference persists when the actor is moved from a theatre to another.
4. The message communication mechanism is lightweight and highly efficient because transferring a message (or migrating an actor) reduces to transferring a message object reference. No object serialization is needed.
5. No local actor table (LAT) per theatre is now required. Actors hold in an internal field the identity of the execution theatre. The move() operation just changes the running theatre reference in the moved actor.
6. The transport layer of a theatre no longer uses socket connections. It only contains a lock-free message queue (inputBuffer) managed by a couple of send/receive operations. The inputBuffer is used by external theatres for sending inter-theatre control or application messages. The inputBuffer is emptied, one message at a time, if there are any, at each iteration of the control machine loop. Control messages are directly managed by the control machine. Application messages get scheduled on the local message queue of the control machine.
7. Being untimed, a pTheatre application does not need a time server component. However, a "time server", implemented as a passive object, is still used as a global detector of the termination condition of a whole pTheatre application. When a control machine finds it has no messages in its local message queue, a stop control message is sent to the theatre hosting the time server. No further stop messages are sent, provided no new external messages are received. Would an external message arrive, a subsequent stop message will be re-issued when the message queue empties again. The stop message carries the control machine counters of sent/received external messages. The time server holds in two maps the identity of the stop requesting theatre and the values of its message counters. When all the theatres of an application have requested a stop and there are no in-transit messages among theatres (the total number of sent messages is found equal to the total number of received messages), the time server broadcasts a terminate control message to theatres which then break the control machine event loop.

8. Newly developed classes include Theatre, PConcurrent, PCTimeServer, PTransportLayer. Theatre extends Thread. Its default run() method just activates the control machine event loop (the controller() method). The run() method is redefined "on-the-fly" when the Theatre instance acts as the master theatre which configures and launches a parallel application (see Fig. 1). PConcurrent is a control machine which processes messages in the sending order. PCTimeServer implements the protocol for system termination.
9. Considering that actors have no built-in thread, a whole pTheatre system runs without any use of locks. All of this simplifies and contributes to safe concurrent programming, and it has the potential to deliver a high execution performance (see the example in the next section).

4 An Example Using Parallel Matrix Multiplication

The goal of the following case study is twofold: to illustrate the actual programming style of pTheatre in Java, and to show the achievable execution performance. The example is concerned with the Geoffrey Fox dense parallel matrix multiplication algorithm [12, 13], which is one of the most memory efficient algorithms [14]. Two squared matrixes $N \times N$ of doubles a and b are considered, and their product matrix c calculated. Toward this, matrixes are decomposed into squared sub-blocks of $\frac{N}{\sqrt{P}} * \frac{N}{\sqrt{P}}$ elements, where P is the number of available processors and $\sqrt{P}$ is assumed to be an integer number. Processors are organized into a $\sqrt{P} * \sqrt{P}$ toroidal grid. Each processor P_{ij} is initially assigned the sub-blocks a_{ij}, b_{ij} and it is responsible of computing the sub-block c_{ij} of the product matrix. In the practical experiment, $P = 4$ processors (cores) are assumed. The Java model is split into three classes: Configurer, Collector and Processor. In the main() method of the Configurer (see Fig. 1) four theatres are created. Theatre 0 is the master theatre (it redefines the run() method) and creates actors, initializes them and activates the parallel system. The Collector actor (not shown for brevity) receives from the processor actors the computed sub-blocks c_{ij} and composes them into the final result matrix c. The Processor actor class realizes the Fox algorithm. The configurer, as well as the time server, are allocated to theatre 0. Each separate processor is allocated to a distinct theatre (core) from 0 to 3.

The processor class is shown in Fig. 2. It is realized as a finite state machine. Each processor computes its sub-block c_{ij} by iterating p times, where $p = \sqrt{P}$, three basic phases expressed as internal states: *broadcast*, *computing* and *rolling-up*. During broadcast, only one processor in a row, starting from the diagonal position, broadcasts (by right piping and toroidally) its sub-block a_{ij} to all the remaining processors in the row. In the BROADCAST state, either a processor is sending its a_{ij} sub-block or it is waiting a sub-block from the left (toroidally) processor in the row. Let t be the received sub-block during a broadcast phase. In the computing phase, each processor updates its c_{ij} sub-block as: $c_{ij} = c_{ij} + t * b_{ij}$. In the subsequent rolling-up phase, each processor rolls up in the grid (upper in the column and always toroidally) its b_{ij} sub-block. As the iterations end, the processor sends its c_{ij} sub-block to the collector actor through a "report" message. Due to the asynchronous nature of the algorithm in Fig. 2, in the case

```
public class Configurer{
   static double[][] block( double[][] m, int d, int i, int j ){…}
   public static void main( String[] args ){
     new Theatre( 3, 4, new PTransportLayer(), new PConcurrent() );
     new Theatre( 2, 4, new PTransportLayer(), new PConcurrent() );
     new Theatre( 1, 4, new PTransportLayer(), new PConcurrent() );
     new Theatre( 0, 4, new PTransportLayer( new PCTimeServer() ), new PConcurrent() ){
       public void run() {
         final int N=…;
         double[][] a=new double[N][N];
         double[][] b=new double[N][N];
         //fill values to a and b
         int P=4; //nr of processors
         int dim=N/(int)Math.sqrt(P); //sub-block dimension
         long start=System.currentTimeMillis();
         Collector c=new Collector();
         c.send( "init",N,P ); c.move(0);
         int p=(int)Math.sqrt(P); //nr of processors in a row/column
         Processor[][] grid=new Processor[p][p];
         for( int i=0; i<p; ++i )
           for( int j=0; j<p; ++j )
             grid[i][j]=new Processor();
         for( int i=0; i<p; ++i )
           for( int j=0; j<p; ++j ){
             grid[i][j].send( "init",i,j,p,block(a,dim,i,j),block(b,dim,i,j),
               grid[i][(j+1)%p] /*right*/, grid[(i-1+p)%p][j] /*above*/, c );
             grid[i][j].move(i*p+j);
           }
         for( int t=0; t<P; ++t ) Theatre.getTheatre(t).activate();
         Theatre.getTheatre( Thread.currentThread().getName() ).
           getControlMachine().controller();
         System.out.println("Parallel WCT="+(System.currentTimeMillis()-start)+" msec");
       }
     };
   }//main
}//Configurer
```

Fig. 1. The configurer class for the pTheatre implementation of the Fox algorithm

a processor receives a message which is not expected in the current state, the message is later postponed by re-sending it to itself.

5 Experimental Analysis

The pTheatre implementation of Fox's algorithm for parallel matrix multiplication was executed on a Win 10, Intel Core i7-7700 CPU@3.60 GHz, 32 GB of memory, by varying the matrix size N from 100 to 3000. The emerged execution time (parallel wall clock time P_WCT) was then compared with that (sequential wall clock time S_WCT) of the classical sequential implementation of the matrix multiplication based on three nested for loops. Each experiment was repeated five times. The

```
public class Processor extends Actor{
  private int i, j, p, it=0, broadcastIteration;
  double[][] a, b, t, c;
  private Processor right, above;
  private Collector collector;
  private static byte CREATED=0,
    BROADCAST=1, COMPUTE=2,
    ROLLING_UP=3, STOP=4;
  private byte cs=CREATED; //current status

  @Msgsrv
  public void init( Integer i, Integer j,
            Integer p,
            double[][] a, double[][] b,
            Processor right, Processor above,
            Collector collector){
    this.i=i; this.j=j; this.p=p; this.a=a;
    this.b=b; this.right=right;
    this.above=above;
    this.collector=collector;
    this.c=new double[a.length][a.length];
    broadcastIteration=(i-j+p)%p;
    if( it==broadcastIteration )
      send( "broadcast", a, i, j );
    cs=BROADCAST;
  }//init

  @Msgsrv
  public void broadcast( double[][] a,
    Integer i, Integer j ){
    if( cs!=BROADCAST )
      send( "broadcast", a, i, j );
    else{
      this.t=a;
      if( j!=(this.j+1)%p )
        right.send( "broadcast", a, i, j );
      send( "compute" ); cs=COMPUTE;
    }
  }//broadcast

  @Msgsrv
  public void compute(){
    if( cs!=COMPUTE ) send( "compute" );
    else{
      update(); it++;
      above.send( "rollingUp", b );
      cs=ROLLING_UP;
    }
  }//compute

  @Msgsrv
  public void rollingUp( double[][] b ){
    if( cs!=ROLLING_UP )
      send( "rollingUp", b );
    else {
      this.b=b;
      if( it==p ){ send( "stop" ); cs=STOP; }
      else{
        if( it==broadcastIteration )
          send( "broadcast", a, i, j );
        cs=BROADCAST;
      }
    }
  }//rollingUp
  @Msgsrv
  public void stop(){
    collector.send( "report", c, i, j );
  }//stop
  private void update(){
    //c=c+t*b
    for( int row=0; row<t.length; ++row )
      for( int col=0; col<t.length; ++col ){
        double tmp=0.0D;
        for( int k=0; k<t.length; ++k )
          tmp+=t[row][k]*b[k][col];
        c[row][col]+=tmp;
      }
  }//update
}//Processor
```

Fig. 2. The processor actor implementing Fox's algorithm

speedup of the parallel implementation was then calculated, for each experiment, as best-case(S_WCT)/worst_case(P_WCT), to smooth out Operating System dependencies. Experimental results are collected in Table 1.

As one can see from Table 1, for low values of the matrix size, the speedup is low because the sequential implementation outperforms the more complex parallel implementation. Some super speedups emerge when a matrix size of 900–1500 is adopted. A speedup value close to the ideal value of 4 (according to the Amdahl law, when 4 cores are used) occurs for $N = 2000$. Obviously, such values are the result of complex combinations of hardware features (cache contents) with the efficient implementation of

Table 1. Speedup of parallel versus sequential program (4 cores)

Matrix size *N*	S_WCT (ms)	P_WCT (ms)	Speedup
100	3	33	0.09
200	10	40	0.25
300	36	63	0.57
400	88	81	1.1
500	170	98	1.73
600	296	152	1.95
700	506	213	2.38
800	858	252	3.4
900	1827	448	4.07
1000	5364	812	6.61
1500	30,445	6352	4.79
2000	81,972	20,907	3.9
2500	205,064	55,237	3.71
3000	350,438	106,753	3.28

pTheatre. On the other hand, similar super speedups were reported, for the Fox algorithm, also in [14].

6 Conclusions

This paper proposes pTheatre, an original, efficient, and totally lock-free Java implementation of the Theatre actor system [6–9]. pTheatre mission is that of supporting high-performance computing on modern multi-core machines. The potential of pTheatre was demonstrated in the paper by an actor-based, asynchronous implementation of the Geoffrey Fox parallel matrix multiplication algorithm [12, 13].

Prosecution of the research is directed to the following:

- Improving the implementation of pTheatre and applying it to complex parallel algorithms.
- Completing a specialization of pTheatre for high-performance agent-based modelling and simulations over multi-core machines, according to a conservative synchronization strategy.
- Porting pTheatre over the GPU, for supporting massive, data-parallel applications.

References

1. Agha, G.: Actors: a model of concurrent computation in distributed systems. PhD Thesis, MIT Artificial Intelligence Laboratory (1986)

2. Karmani, R.K., Agha, G.: Actors, pp. 1–11. Springer US, Boston, MA (2011)
3. Agha, G., Palmskog, K.: Transforming threads into actors: learning concurrency structure from execution traces. In: Principles of Modelling, pp. 16–37. Springer, Cham (2018)
4. Lee, E.A.: The problem with threads. Computer **39**, 33–42 (2006). https://doi.org/10.1109/MC.2006.180
5. Jafari, A., Khamespanah, E., Sirjani, M., Hermanns, H., Cimini, M.: PTRebeca: modelling and analysis of distributed and asynchronous systems. Sci. Comput. Program. **128**, 22–50 (2016)
6. Cicirelli, F., Nigro, L.: Control centric framework for model continuity in time-dependent multi-agent systems. In: Concurrency and Computation: Practice and Experience, vol. 28, no. 12, pp. 3333–3356. Wiley (2016)
7. Cicirelli, F., Nigro, L., Sciammarella, P.F.: Model continuity in cyber-physical systems: a control centered methodology based on agents. Simul. Model. Pract. Theory **83**, 93–107 (2018). https://doi.org/10.1016/j.simpat.2017.12.008
8. Nigro, L., Sciammarella, P.F.: Qualitative and quantitative model checking of distributed probabilistic timed actors. Simul. Model. Pract. Theory **87**, 343–368 (2018)
9. Cicirelli, F., Nigro, L., Sciammarella, P.F.: Seamless development in Java of distributed real-time systems using actors. Int. J. Simul. Process Model. **15**, Nos 1/2, 13–29 (2020)
10. Nigro, L., Sciammarella, P.F.: Verification of a smart power control systems using hybrid actors. In: IEEE WorldS4, London, 30–31 July 2019
11. Cicirelli, F., Nigro, L.: Home energy management using theatre with hybrid actors. In: IEEE/ACM 23rd International Symposium on Distributed Simulation and Real Time Application (DS-RT 2019), Cosenza, Italy, 7–9 Oct 2019
12. Fox, G.C.: Solving problems on concurrent processors. In: General Techniques and Regular Problems, vol. 1. Prentice-Hall (1988)
13. Fox, G.C., Otto, S.W., Hey, A.J.G.: Matrix algorithms on a hypercube I: matrix multiplication. Parallel Comput. **4**, 17–31 (1987)
14. Gergel, V.: Parallel methods for matrix multiplication. In: 2012 Summer School on Concurrency, pp. 1–50, St. Petersburg, Russia, 22–29 Aug 2012