PPF – A Parallel Particle Filtering Library

Ömer Demirel*, Ihor Smal†, Wiro J. Niessen†, Erik Meijering† and Ivo F. Sbalzarini*

*MOSAIC Group, Center of Systems Biology Dresden (CSBD), Max Planck Institute of Molecular Cell Biology and Genetics, Pfotenhauerstr. 108, 01307 Dresden, Germany. email: {demirel,ivos}@mpi-cbg.de

†Biomedical Imaging Group Rotterdam, Departments of Medical Informatics and Radiology, Erasmus MC – University Medical Center Rotterdam, Rotterdam, The Netherlands. email: {i.smal,w.niessen}@erasmusmc.nl, meijering@imagescience.org

Abstract. We present the parallel particle filtering (PPF) software library, which enables hybrid shared-memory/distributed-memory parallelization of particle filtering (PF) algorithms combining the Message Passing Interface (MPI) with multithreading for multi-level parallelism. The library is implemented in Java and relies on OpenMPI's Java bindings for inter-process communication. It includes dynamic load balancing, multi-thread balancing, and several algorithmic improvements for PF, such as input-space domain decomposition. The PPF library hides the difficulties of efficient parallel programming of PF algorithms and provides application developers with a tool for parallel implementation of PF methods. We demonstrate the capabilities of the PPF library using two distributed PF algorithms in two scenarios with different numbers of particles. The PPF library runs a 38 million particle problem, corresponding to more than 1.86 TB of particle data, on 192 cores with 67% parallel efficiency.

1. Introduction

Particle filters (PFs) are popular algorithms for tracking (multiple) targets under non-linear, non-Gaussian dynamics and observation models. From econometrics [1] to robotics [2] and sports [3], PFs have been applied to a wide spectrum of signal-processing applications. Despite their advantages, the inherently high computational cost of PF limits their practical application, especially in real-time problems. This challenge has been addressed by algorithmic improvements [4], efficient shared-memory [5] and many-core [6–8] implementations, and scalable distributed-memory solutions [9, 10].

Here, we present a parallel software library for particle filtering, called the PPF library, which aims at providing application developers with an easy-to-use and scalable platform to develop PF-based parallel solutions for their applications. The main contributions of the PPF library are a highly optimized implementation and the extension of distributed resampling algorithms [11] to hybrid shared-/distributed-memory systems.

The library is written in Java and has an object-oriented architecture. It exploits a hybrid model of parallelism where the Message Passing Interface (MPI) [12] and Java threads are combined. The framework relies on the recent Java bindings of OpenMPI [13, 14] for inter-process communication, and on Java threads for intra-process parallelism. Java is an emerging language in high-performance computing (HPC) offering new research opportunities.

The PPF library includes implementations of different strategies for dynamic load balancing (DLB) across processes, and a checkerboard-like thread balancing scheme within processes. Inter-process DLB is used in the resampling phase of a PF, whereas thread balancing (i.e., intra-process balancing) is used throughout the entire library. Non-blocking point-to-point MPI operations are exploited wherever the chosen DLB strategy allows for them. Furthermore, the framework has interfaces for ImageJ [15], Fiji [16], and imagescience [17], allowing these popular image-processing applications to directly access PPF's application programming interface (API). The proposed framework can be used to facilitate implementation of other computationally demanding PF based techniques such as pMCMC [18], SMC^2 [19], and SMS-(C)PHD [20].

The manuscript is structured as follows: The generic PF algorithm is described in Section 2. In Section 3, distributed resampling algorithms are briefly discussed, followed by DLB schedules (Section 4) and effective particle routing (Section 5). In Section 6, we provide implementation details of the PPF library. Section 7 shows an application from biological imaging processing, where a single target is tracked through a sequence of 2D images. Two different distributed PF algorithms are compared. We summarize the contributions of the PPF library and discuss future work in Section 8.

2. Particle Filters

A generic PF algorithm consists of two parts: (i) sequential importance sampling (SIS) and (ii) resampling [21]. A popular combined implementation of these two parts is the sequential importance resampling (SIR) algorithm [21]. Recursive Bayesian importance sampling [22] of an unobserved and discrete Markov process $\{\mathbf{x}_k\}_{k=1,\ldots,K}$ is based on three components: (i) the measurement vector $\mathbf{Z}^k = \{\mathbf{z}_1, \ldots, \mathbf{z}_k\}$, (ii) the dynamics (i.e., state-transition) model, which is given by a probability distribution $p(\mathbf{x}_k|\mathbf{x}_{k-1})$, and (iii) the likelihood (i.e., observation model) $p(\mathbf{z}_k|\mathbf{x}_k)$. Then, the state posterior $p(\mathbf{x}_k|\mathbf{Z}^k)$ at time k is recursively computed as:

$$\underbrace{p(\mathbf{x}_k|\mathbf{Z}_k)}_{\text{posterior}} = \frac{\overbrace{p(\mathbf{z}_k|\mathbf{x}_k)}^{\text{likelihood}}\,\overbrace{p(\mathbf{x}_k|\mathbf{Z}^{k-1})}^{\text{prior}}}{\underbrace{p(\mathbf{z}_k|\mathbf{Z}^{k-1})}_{\text{normalization}}}, \tag{1}$$

where the prior is defined as:

$$p(\mathbf{x}_k|\mathbf{Z}^{k-1}) = \int p(\mathbf{x}_k|\mathbf{x}_{k-1})\, p(\mathbf{x}_{k-1}|\mathbf{Z}^{k-1})\, \mathrm{d}\mathbf{x}_{k-1}. \qquad (2)$$

Having the posterior, the minimum mean square error (MMSE) or maximum *a posteriori* (MAP) estimators of the state can be obtained [21], for example by

$$\mathbf{x}_t^{\mathrm{MMSE}} = \int \mathbf{x}_t p(\mathbf{x}_t|\mathbf{Z}^t) d\mathbf{x}_t.$$

Due to intractability of the high-dimensional integrals, PFs approximate the posterior at each time point k by N weighted samples (called "particles") $\{\mathbf{x}_k^i, w_k^i\}_{i=1,\dots,N}$. This approximation amounts to Monte-Carlo quadrature and is achieved by sampling a set of particles from an importance function (proposal) $\pi(\cdot)$ and updating their weights according to the dynamics and observation models. This process is called sequential importance sampling (SIS) [21]. SIS suffers from *weight degeneracy* where small particle weights become successively smaller and do not contribute to the posterior any more. To overcome this problem, a *resampling* step is performed [21] whenever the number of particles with relatively high weights falls below a specified threshold. Through the standard notation [21,23], the resulting SIR algorithm is given in Algorithm 1. To parallelize the SIR algorithm, one only needs to focus on the *resampling* step, since all other parts of the SIR algorithm are local and can trivially be executed in parallel.

Algorithm 1 SIR, a generic Particle Filtering Algorithm

```
 1: procedure SIR
 2:     for $i = 1 \to N$ do                              ▷ Initialization, $k$=0
 3:         $w_0^i \leftarrow 1/N$
 4:         Draw $\mathbf{x}_0^i$ from $\pi(\mathbf{x}_0)$
 5:     end for
 6:     for $k = 1 \to K$ do
 7:         for $i = 1 \to N$ do                          ▷ SIS step
 8:             Draw a sample $\tilde{\mathbf{x}}_k^i$ from $\pi(\mathbf{x}_k|\mathbf{x}_{k-1}^i, \mathbf{Z}^k)$
 9:             Update the importance weights
10:             $\tilde{w}_k^i \leftarrow w_{k-1}^i \frac{p(\mathbf{z}_k|\tilde{\mathbf{x}}_k^i)p(\tilde{\mathbf{x}}_k^i|\mathbf{x}_{k-1}^i)}{\pi(\tilde{\mathbf{x}}_k^i|\mathbf{x}_{k-1}^i,\mathbf{Z}^k)}$
11:         end for
12:         for $i = 1 \to N$ do
13:             $w_k^i \leftarrow \tilde{w}_k^i / \sum_{j=1}^N \tilde{w}_k^j$
14:         end for
                                   ▷ Calculate the effective sample size
15:         $\widehat{N}_{\mathrm{eff}} \leftarrow 1/\sum_{j=1}^N (w_k^j)^2$
16:         if $\widehat{N}_{\mathrm{eff}} < N_{\mathrm{threshold}}$ then          ▷ Resampling step
17:             Sample a set of indices $\{s(i)\}_{i=1,\dots,N}$ distributed such that $\Pr[s(i) = l] = w_k^l$ for $l = 1 \to N$.
18:             for $i = 1 \to N$ do
19:                 $\mathbf{x}_k^i \leftarrow \tilde{\mathbf{x}}_k^{s(i)}$
20:                 $w_k^i \leftarrow 1/N$                ▷ Reset the weights
21:             end for
22:         end if
23:     end for
24: end procedure
```

3. Distributed resampling algorithms

Distributed resampling algorithms (DRAs) can be categorized into three classes: The first includes *multiple particle filter* (MPF) algorithms [24], which are banks of truly independent PFs with no communication during and/or after local resampling. This renders MPF an embarrassingly parallel approach, in which every process runs a separate PF. The only communication in this case is sending the local estimates of the state to a master node that forms a combined estimate. However, this low communication complexity comes at the cost of a reduced statistical accuracy due to problems when, for example, some of the independent PFs diverge during the sequential estimation and introduce large errors into the combined estimate. If some of the independent PFs happen to explore the same area of state space, the approach is also computationally wasteful. Using a global resampling scheme, which incurs global communication between the processes, accuracy and efficiency can be improved.

The second DRA category includes *distributed resampling algorithms with non-proportional allocation* (RNA) [11] and *local selection* (LS) algorithms [25]. These algorithms are designed to minimize inter-process communication. However, the way they perform resampling may cause a statistical accuracy loss, since the result generated by these algorithms depends on the number of processes used. A comparison of RNA with LS can be found in [26]. In RNA, the state space is divided into disjoint strata, each of them assigned to a different process. The number of particles is fixed and the same for each process. The resampling step is performed locally by each process in its stratum, leading to an uneven weight distribution across processes. This requires deterministic particle routing (i.e., DLB) strategies. A popular choice is to migrate 10%–50% of the particles of each process to the neighboring process after resampling [9, 26]. Neighborhood between processes is defined by a process topology, which usually is taken to be a ring. Using such a simple, static DLB scheme shortens application development times, but may cause redundant communication especially once the PF has converged onto its target and tracks it through state space. In this case, the PFs in all processes have converged to an equally good approximation of the posterior. The exchange of particles between processes would then not be necessary anymore, but is still wastefully performed in RNA. It is also not clear what the optimal percentage of migrating particles should be. For applications requiring high precision, the number of particles that need to be communicated may also become too large, limiting the scalability of RNA.

The third class of DRAs contains *distributed resampling algorithms with proportional allocation* (RPA) [11]. These algorithms are based on stratified sampling with proportional allocation, which means that particles with larger weights are resampled more often. This causes growing particle imbalance between the processes, since a process with higher-weight particles will generate more particles and thus become overloaded. Similarly, processes with low-weight particles will have even fewer particles after resampling and "starve". To overcome this problem, RPA requires adaptive DLB schemes for particle routing, where the number of communication links is minimized in order to reduce the *latency* in the network. In addition, also the sizes of the communicated messages have to be optimized in order to avoid *bandwidth* problems. We address both the *latency* and the *bandwidth* criteria in RPA algorithms using the DLB schedules presented next.

4. DLB Schedules for RPA

To rebalance the workload (i.e., number of particles) on each process, one has to use a DLB scheme. PPF implements three DLB protocols, which all start by labeling the processes as either *senders* or *receivers*. A good DLB scheduler then minimizes the communication overhead required for routing particles from the *senders* to the *receivers*.

4.1 Greedy Scheduler

The Greedy Scheduler (GS) matches the first sender with the first receiver and then iterates through the senders. For each sender S_i with particle surplus N_{S_i}, it moves as many particles as possible to receiver R_j. Once a receiver is full, it moves on to the next R_{j+1} until the sender is empty. The procedure guarantees that at the end each process has the same number of particles. The pseudocode of GS is given in Algorithm 2.

Algorithm 2 Greedy Scheduler

```
Require: S:=the list of senders, R:=the list of receivers.
Ensure: schedule:=the list of matchings between the elements
    of S and R including the number of particles to be routed.
 1: procedure GreedyScheduler($S,R$)
 2:     $j \leftarrow 0$
 3:     while $S \neq \varnothing$ do
 4:         while $N_{S_i} \neq 0$ do
 5:             if $N_{S_i} \geq N_{R_j} > 0$ then
 6:                 schedule $\leftarrow \{S_i, R_j, N_{R_j}\}$
 7:                 $N_{S_i} \leftarrow N_{S_i} - N_{R_j}$
 8:                 $N_{R_j} \leftarrow 0$
 9:                 $j \leftarrow j+1$
10:             else if $N_{R_j} > N_{S_i} \geq 0$ then
11:                 schedule $\leftarrow \{S_i, R_j, N_{S_i}\}$
12:                 $N_{R_j} \leftarrow N_{R_j} - N_{S_i}$
13:                 $N_{S_i} \leftarrow 0$
14:                 $j \leftarrow 0$
15:             end if
16:         end while
17:         $i \leftarrow i+1$
18:     end while
19:     return schedule
20: end procedure
```

4.2 Sorted Greedy Scheduler

The Sorted Greedy Scheduler (SGS) [24, 27, 28] first sorts the senders in S by their N_{S_i} and the receivers in R by their N_{R_j}, both in descending order. This sorting reduces the number of required communication links. The rest of the SGS algorithm is identical with GS, as seen in Algorithm 3.

Algorithm 3 Sorted Greedy Scheduler

```
Require: S:=the list of senders, R:=the list of receivers.
Ensure: schedule:=the list of matchings between the elements
    of S and R including the number of particles to be routed.
 1: procedure SortedGreedyScheduler($S,R$)
 2:     $S' \leftarrow$ sort($S$)                 ▷ in descending order
 3:     $R' \leftarrow$ sort($R$)                 ▷ in descending order
 4:     return GreedyScheduler($S',R'$)
 5: end procedure
```

4.3 Largest Gradient Scheduler

While GS and SGS aim to balance the loads perfectly, one may be interested in a faster scheduler that causes less communication overhead, but does not guarantee optimal particle balancing. The Largest Gradient Scheduler (LGS) is such a suboptimal heuristic. Similar to SGS, LGS first sorts S and R such that $N_{S_1} > N_{S_2} > \ldots > N_{S_{|S|}}$ and $N_{R_1} > N_{R_2} > \ldots > N_{R_{|R|}}$. After that, each sender is paired with the corresponding receiver of same rank:

$$\begin{gathered} S_1 \rightarrow R_1, \\ S_2 \rightarrow R_2, \\ \vdots \\ S_{\min(|S|,|R|)} \rightarrow R_{\min(|S|,|R|)}. \end{gathered}$$

LGS thus finds the *largest gradients* between S and R and limits the number of communication links to

$$C = \min(|S|, |R|).$$

The pseudocode of LGS is given in Algorithm 4.

Algorithm 4 Largest Gradient Scheduler

```
Require: S:=the list of senders, R:=the list of receivers.
Ensure: schedule:=the list of matchings between the elements
    of S and R including the number of particles to be routed.
 1: procedure LargestGradientScheduler($S,R$)
 2:     $S' \leftarrow$ sort($S$)                 ▷ in descending order
 3:     $R' \leftarrow$ sort($R$)                 ▷ in descending order
 4:     for $i = 1 \rightarrow \min(|S|,|R|)$ do
 5:         if $N_{S_i} \geq N_{R_i}$ then
 6:             $N_{S_i} \leftarrow N_{S_i} - N_{R_i}$
 7:             schedule $\leftarrow \{S_i, R_i, N_{R_i}\}$
 8:         else
 9:             $N_{S_i} \leftarrow 0$
10:             schedule $\leftarrow \{S_i, R_i, N_{S_i}\}$
11:         end if
12:     end for
13:     return schedule
14: end procedure
```

5. Parallel Particle Filtering Library

The PPF library is written in Java and uses the Message-Passing Interface (MPI) [12] for inter-process communication,

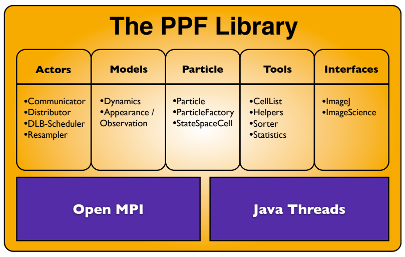

Figure 1. Software structure of the PPF library. The library is divided into five modules and hides the complexity of MPI and multithreading from the application programmer. It also supports using `imagescience` classes, as well as functions from ImageJ and Fiji.

which is the *de facto* standard for parallel high-performance computing. The OpenMPI team [13] recently started providing a Java interface based on `mpiJava` [29], which covers all MPI 1.2 functions and is currently maintained on provisional basis in the development trunk of OpenMPI [30]. The PPF library also provides built-in support for ImageJ [15], Fiji [16], and `imagescience` [17] for file I/O, image analysis, and image editing. A schematic of the structure of the PPF library is shown in Fig. 1.

The library consists of five modules: (i) actors, (ii) models, (iii) particle, (iv) tools, and (v) interfaces. The *actors* module encapsulates functionality that is common to PF algorithms (e.g., resampling) and provides support for parallel PFs via its communication, data distribution, and DLB sub-modules. The *models* module includes dynamics and observation models. By default, the library includes simple standard models that can be sub-classed to include application-specific models. The *particle* module contains the particle data structure and related methods (e.g., particle generation). The *tools* module contains a set of helper methods for sorting, statistical calculations, efficient particle neighbor lists, etc. The *interfaces* module provides APIs to link the PPF library to ImageJ [15], Fiji [16], and `imagescience` [17]. This allows ImageJ/Fiji plugins to access the functionality provided by the PPF library, but it also allows PPF methods to use functions provided by ImageJ/Fiji, such as functions for image processing, file I/O, and graphical user-interface building.

A client code that implements a parallel PF application can directly call the PPF API. Most of the intricacies arising from parallel programming and code optimization are hence hidden from the application programmer. Sitting as a middleware on top of MPI and Java threads, the PPF library makes the parallelization of PF applications on shared- and distributed-memory systems easier. Below, we highlight some of the features of the PPF library.

5.1 Multi-level Hybrid Parallelism

As HPC systems grow in size, multi-level hybrid parallelization techniques emerge as a viable tool to achieve high parallel efficiency in scalable scientific applications. Many applications [31] realize hybrid parallelization strategies by combining MPI with OpenMP [32]. In order to have full thread control and also enable job-level multi-threading, we here follow the trend of combining MPI for inter-process communication with native threads for intra-process parallelism. We employ Java's native thread concurrency model in the PPF library, which provides full thread control and avoids additional software/compiler installation and maintenance. Furthermore, we provide an intra-process load-balancing scheme for threads specifically designed for PF applications, whose implementation using Java threads is straightforward and easy to extend.

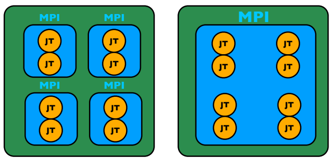

Figure 2. Two possible ways to use hybrid parallelism with MPI and Java threads (JT) in PPF: On the left, each MPI process is bound to a *CPU* (blue squares), and each JT is assigned to a *core* (orange circles). On the right, there is one MPI process per *node/computer* (green rectangle), and again one JT per *core*. The PPF library implements both models and lets the application developer choose which one to use for a specific application.

The PPF library lets the user choose between two different concurrency models, as illustrated in Fig. 2: In the first model (left panel), the number of MPI processes is equal to the number of available CPU chips, and the number of threads per process is equal to the number of cores per CPU. Compared to an all-MPI paradigm, this allows benefitting from shared caches between cores and reduces communication latency. The second model (right panel) uses a single MPI process per *node/computer* and and one Java thread for each core. This hybrid model keeps the shared memory on the node coherent and causes a lower memory latency than an all-MPI implementation [33]. Which model one chooses for a particular application depends on the specific hardware (cache size, number of memory banks, memory bus speed, etc.) and application (data size, computational cost of likelihood evaluation, etc.).

5.2 Non-blocking MPI Operations

During execution of DLB strategies, such as GS, SGS, or LGS, we use non-blocking point-to-point MPI operations in order to overlap communication with computation. This is especially useful during the DLB phase, where a sender sends its message to a receiver and then immediately carries on with generating new local particles.

5.3 Input-space Domain Decomposition

In the PPF library, MPI-level parallelization is done at the particle-data level. This means that each MPI process has full knowledge of the input, e.g. the image to be processed, but only knows a part of the particles in state space. At the thread level, we then also decompose the input (e.g., image) into smaller subdomains. This provides a convenient way of introducing thread-level parallelism. For images, the pixels containing particles are directly assigned to local threads. Thus, both the state space is distributed via MPI, and the input space is distributed across threads.

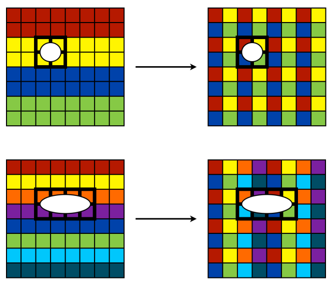

Figure 3. Big boxes represent the binned state space (where in the case of images the pixels can be used as bins) and the colors represent which part of the state space will be processed by which thread. Once particles concentrate around an object to be tracked, i.e. the posterior approximation converged, thread balancing accelerates the computations. The PPF library implements a checkerboard-like thread balancing scheme with patch sizes that depend on the support of the posterior distribution and on the number of threads. Two examples are shown here: In the 2×2 scheme (upper row), the area covered by the posterior (white circle) is distributed across four threads (upper right corner). Similarly, the *2×4* thread-balancing scheme (lower row) distributes a larger posterior (white ellipse) across eight threads.

5.4 Thread Balancing

Thread-level load balancing is as important as process-level DLB. When using PFs for object tracking, for example, once an object is found and locked onto, many particles are drawn to the vicinity of that object. This worsens thread load-balance, since if consecutive pixels belong to the same thread, that thread becomes overloaded while others are idle.

The PPF library hence uses an adaptive checkerboard-like load balancing scheme for threads, as illustrated in Fig. 3. Depending on the number of threads and the support of the posterior distribution, the size of the checkerboard patch is automatically adjusted.

5.5 Image Patches

When using PFs for image processing, the likelihood computation involves image data and may be computationally costly due to, e.g., invoking a numerical simulation of the image-formation process. Performance improvements in the likelihood calculation hence have the largest impact on the overall performance of a PF image-processing application.

In many image-based applications, a separate likelihood estimation is carried out for each particle, where the full image is loaded and then the likelihood kernel is applied. In image-based likelihood computations, these kernels are typically symmetric (e.g., Gaussian) and local (i.e., span only a few pixels). Thus, it is sufficient to visit pixels one by one and only load the *image patch* centered at the visited pixel. The size of the patch is given by the likelihood kernel support, which is typically much smaller than the whole image. Once a patch is loaded, computing the particle weights in the central pixel requires less time. In fact, if there are N_{pix} pixels in the image and N particles in state space, the overall computational complexity of the likelihood calculation is reduced from $\mathcal{O}(NN_{\text{pix}})$ to $\mathcal{O}(N)$, which is usually a significant reduction.

Since threads are distributed in a checkerboard-like fashion, only one thread needs to load an image patch and all neighboring threads can simply access the patch data from shared cache. This results in better cache efficiency.

5.6 Approximate Sequential Importance Resampling

The PPF library also provides an implementation of a fast approximate SIR algorithm that uses a piecewise constant approximation of the likelihood function to estimate the posterior distribution faster. This algorithm is called Piecewise Constant Sequential Importance Sampling and can offer significant speedups [34].

6. An Example Application

We demonstrate the capabilities of the PPF library by using it to implement a PF application for tracking sub-cellular objects imaged by fluorescence microscopy [35, 36]. In this example from biological microscopy imaging, sub-cellular structures such as endosomes, vesicles, mitochondria, or viruses are fluorescently labeled and imaged over time with a microscope. Many biological studies start from analyzing the dynamics of those structures and extracting parameters that characterize their behavior, such as average velocity, instantaneous velocity, spatial distribution, motion correlations, etc. First, we describe the dynamics and appearance models implemented in the PPF library for this biological application, and then we explain the technical details of the experimental setup.

6.1 Dynamics Model

The motion of sub-cellular objects can be represented using a variety of motion models, ranging from random walks to constant-velocity models to more complex dynamics where switching between motion types occurs [37, 38].

Here, we use a near-constant-velocity model, which is frequently used in practice [39, 40]. The state vector in this case is $\mathbf{x} = (\hat{x}, \hat{y}, v_x, v_y, I_0)^T$, where $\hat{x}$ and $\hat{y}$ are the estimated x- and y-positions of the object, (v_x, v_y) its velocity vector, and I_0 its estimated fluorescence intensity.

6.2 Observation Model

Many sub-cellular objects are smaller than what can be resolved by a visible-light microscope, making them appear in a fluorescence image as diffraction-limited bright spots, where the intensity profile is given by the impulse-response function of the microscope, the so-called point-spread function (PSF) [38, 39, 41, 42].

In practice, the PSF of a fluorescence microscope is well approximated by a 2D Gaussian [43]. The object appearance in a 2D image is hence modeled as:

$$I(x,y;x_0,y_0) = I_0 \exp\left(-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma_{\text{PSF}}^2}\right) + I_{\text{bg}}, \quad (3)$$

where (x_0, y_0) is the position of the object, I_0 is its intensity, I_{bg} is the background intensity, and σ_{PSF} is the standard deviation of the Gaussian PSF. Typical microscope cameras yield images

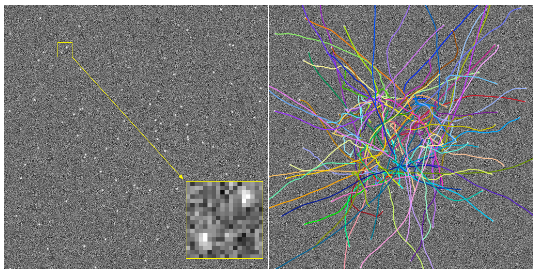

Figure 4. Examples of low SNR synthetic images used in the experiments. Left: The first 512×512 frame from a movie sequence of 50 frames, showing the typical object appearance due to the Gaussian PSF model. Right: Trajectories of the moving bright objects, generated using the nearly-constant-velocity dynamics model, overlaid with the first image frame.

with pixel edge lengths corresponding to 60 to 200 nm physical length in the specimen. For the images used here, the pixel size is 67 nm, and the microscope has $\sigma_{\mathrm{PSF}} = 78$ nm (or 1.16 pixels). During image acquisition, the "ideal" intensity profile $I(x,y)$ is corrupted by measurement noise, which in the case of fluorescence microscopy has mixed Gaussian-Poisson statistics. For the resulting noisy image $\mathbf{z}_k = Z_k(x,y)$ at time point k, the likelihood $p(\mathbf{z}_k|\mathbf{x}_k)$ is:

$$p(\mathbf{z}_k|\mathbf{x}_k) \propto \exp\left(-\frac{1}{2\sigma_\xi^2}\sum_{(x_i,y_i)\in\mathscr{S}_\mathbf{x}}[Z_k(x_i,y_i) - I(x_i,y_i;\hat{x},\hat{y})]^2\right), \quad (4)$$

where σ_ξ controls the peakiness of the likelihood, (x_i,y_i) are the integer coordinates of the pixels in the image, $(\hat{x},\hat{y})$ are the spatial components of the state vector $\mathbf{x}_k$, and $\mathscr{S}_\mathbf{x}$ defines a small region in the image centered at the location specified by the state vector $\mathbf{x}_k$. Here, $\mathscr{S}_\mathbf{x} = [\hat{x} - 3\sigma_{\mathrm{PSF}}, \hat{x} + 3\sigma_{\mathrm{PSF}}] \times [\hat{y} - 3\sigma_{\mathrm{PSF}}, \hat{y} + 3\sigma_{\mathrm{PSF}}]$.

6.3 Experimental Setup

We consider a single-object tracking problem where $\sigma_{\mathrm{PSF}} = 1.16$ pixels and the images have a signal-to-noise ratio (SNR) of 2 (equivalent to 6 dB). An example of an input image is shown in Fig. 4 (left). It is a synthetic image showing a number of bright PSF spots moving according to the above-described dynamics and appearance models. The task considered here for the PF is to detect the spots and track their motion over time, reconstructing their trajectories. The ground-truth trajectories used to generate the synthetic movies are shown in Fig. 4 (right). Comparing PF tracking results with them allows quantifying the tracking error. For other applications, the PPF library can easily be extended to include also other dynamics and observation models.

Using double-precision arithmetics, a single particle requires 52 kB (i.e., six doubles and one integer) of computer memory. The particles are initialized uniformly at random and all tests are repeated 20 times with different random synthetic movies on the MadMax computer cluster of MPI-CBG, which consists of 44 nodes each having two 6-core Intel® Xeon® E5-2640 2.5 GHz CPU with 128 GB DDR3 800 MHz memory. All algorithms are implemented in Java (v. 1.7.0_13), and

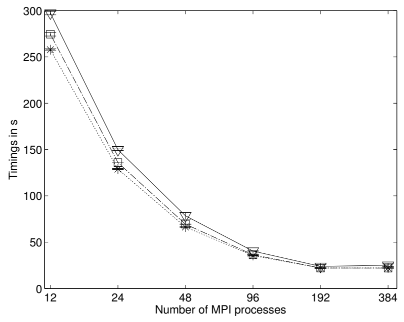

Figure 5. Absolute wall-clock times of RNA with 10% (□) and 50% (▽) particle-exchange ratios and of ARNA (⋆) for a strong scaling with 38.4 million particles. This problem size allows efficient parallelism on up to 192 cores.

OpenMPI (v. 1.9a1r28750) is used for inter-process communication.

We test the PPF library using both the RNA and RPA algorithms with different problem sizes and computer system sizes up to 384 cores.

6.4 Results with RNA

In RNA, each processor maintains the same amount of particles and thus a DLB scheme is not required. The PPF library implements both the classical RNA as well as *adaptive RNA* (ARNA) [44]. ARNA dynamically adjusts the particle-exchange ratio by keeping track of the number of processes actually involved in successfully tracking an object. These processes are then arranged in a ring topology, such that particle routing is straightforward. In order to find the tracked object again if it has been lost for a couple of frames, ARNA randomizes the process order in the ring topology whenever the object is lost, thus ensuring faster information exchange to re-locate the object.

In the example described above, the PPF library reduces the runtime from about 50 minutes on a single core to about 20 seconds on 192 cores. Figure 5 shows the wall-clock runtimes for a strong scaling with a constant number of 38.4 million particles (1.86 TB of particle data) distributed across an increasing number of cores. RNA and ARNA show parallel efficiencies of 65% and 67%, respectively, on 192 cores for the selected problem size (Fig. 6). Beyond 192 cores, the parallel efficiencies drop below 40% for all RNA variants, as the number of particles per process becomes too small to amortize the communication overhead. For all RNA simulations, we use a hybrid parallelism model defined in Fig. 2 (left) where each MPI process is assigned a single Java Thread.

6.5 Results with RPA

For RPA, we compare three different DLB schemes. The tracking accuracy is measured by the root-mean-square error (RMSE) and was the same for all tests (about 0.063 pixels). All DLB schemes hence lead to results of equal quality. We use six Java threads per MPI process, since each CPU of the benchmark machine has six cores, and one MPI process per CPU. The wall-clock times are shown in Fig. 7 for a weak

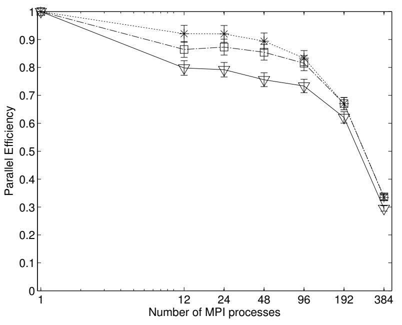

Figure 6. Parallel efficiencies of RNA with 10% (□) and 50% (▽) particle-exchange ratios and of ARNA (⋆) for a strong scaling with 38.4 million particles. This problem size allows efficient parallelism on up to 192 cores.

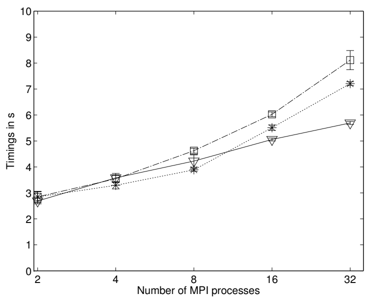

Figure 7. Weak scaling runtime results with less than a second standard deviation of a RPA run with 60k particles per MPI process. Each MPI processes is mapped onto a single CPU with six logical cores. There are six Java threads per MPI process. Three DLB schemes are used: Greedy (□), SortedGreedy (⋆), and LGS (▽).

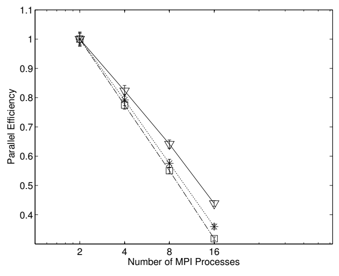

Figure 8. Strong-scaling parallel efficiencies of RPA with a constant number of 3.84 million particles distributed across an increasing number of processes. Three different DLB schemes are compared: Greedy (□), SortedGreedy (⋆), and LGS (▽). Each MPI process is pinned to one of the two available CPUs on each node, each running six Java threads.

scaling with 60 000 particles per process. The corresponding parallel efficiency is shown in Fig. 8. Overall, LGS provides the best scalability, due to its linear communication complexity. Nevertheless, RPA scales less well than RNA and ARNA. For all RPA tests, we use a hybrid parallelism model defined in Fig. 2 (right) where each MPI process is assigned six Java Threads.

7. Conclusions

We presented the PPF library that enables parallel particle filtering applications on commodity as well as on high-performance parallel computing systems. The library uses multi-level hybrid parallelism combining OpenMPI with native Java threads. As a proof of concept for the recently developed OpenMPI Java bindings, we showed the capability of the PPF library in a biological imaging application. The PPF library reduces parallel runtimes of the RNA and RPA methods by integrating dynamic load balancing with thread balancing and also implements PF-specific algorithmic improvements such as domain decomposition, image patches, and approximate sequential importance resampling (ASIR). The PPF library renders using parallel computer systems easier for application developers by hiding the intricacies of parallel programming and providing a simple API to design parallel PF applications. We presented a test case where 1.86 TB of particle data were distributed across 192 cores at 67% efficiency with either the RNA or ARNA methods used.

Future work includes adding support for graphics processing units (GPU) to further accelerate certain parts of PF algorithms and designing new DLB algorithms for RPA that exploit MPI 3.0 extensions, such as non-blocking collective operations.

The PPF library is available as open source and free of charge from the MOSAIC Group's web page `http://mosaic.mpi-cbg.de`, under the "Downloads" section.

Acknowledgments

The authors thank the MOSAIC Group (MPI-CBG, Dresden) for fruitful discussions and the MadMax cluster team (MPI-CBG, Dresden) for operational support. Ömer Demirel was funded by grant #200021–132064 from the Swiss National Science Foundation (SNSF), awarded to I.F.S. Ihor Smal was funded by a VENI grant (#639.021.128) from the Netherlands Organization for Scientific Research (NWO).

References

[1] Thomas Flury and Neil Shephard. Bayesian inference based only on simulated likelihood: particle filter analysis of dynamic economic models. *Econometric Theory*, 27(5):933, 2011.

[2] Hashem Tamimi, Henrik Andreasson, André Treptow, Tom Duckett, and Andreas Zell. Localization of mobile robots with omnidirectional vision using particle filter and iterative sift. *Robotics and Autonomous Systems*, 54(9):758–765, 2006.

[3] Michael D Breitenstein, Fabian Reichlin, Bastian Leibe, Esther Koller-Meier, and Luc Van Gool. Robust tracking-by-detection using a detector confidence particle filter. In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 1515–1522. IEEE, 2009.

[4] Zhaowen Wang, Xiaokang Yang, Yi Xu, and Songyu Yu. Camshift guided particle filter for visual tracking. *Pattern Recognition Letters*, 30(4):407–413, 2009.

[5] Matthew A Goodrum, Michael J Trotter, Alla Aksel, Scott T Acton, and Kevin Skadron. Parallelization of particle filter algorithms. In *Computer Architecture*, pages 139–149. Springer, 2012.

[6] James Anthony Brown and David W Capson. A framework for 3D model-based visual tracking using a GPU-accelerated particle filter. *Visualization and Computer Graphics, IEEE Transactions on*, 18(1):68–80, 2012.

[7] Gustaf Hendeby, Rickard Karlsson, and Fredrik Gustafsson. Particle filtering: the need for speed. *EURASIP Journal on Advances in Signal Processing*, 2010:22, 2010.

[8] Simon Maskell, Ben Alun-Jones, and Malcolm Macleod. A single instruction multiple data particle filter. In *Nonlinear Statistical Signal Processing Workshop, 2006 IEEE*, pages 51–54. IEEE, 2006.

[9] Sven Zenker. Parallel particle filters for online identification of mechanistic mathematical models of physiology from monitoring data: performance and real-time scalability in simulation scenarios. *Journal of clinical monitoring and computing*, 24(4):319–333, 2010.

[10] Lawrence M Murray. Bayesian state-space modelling on high-performance hardware using libbi. *arXiv preprint arXiv:1306.3277*, 2013.

[11] Miodrag Bolic, Petar M Djuric, and Sangjin Hong. Resampling algorithms and architectures for distributed particle filters. *Signal Processing, IEEE Transactions on*, 53(7):2442–2450, 2005.

[12] MPI Forum. Message passing interface (MPI) forum home page. `http://www.mpi-forum.org/` (Dec. 2009).

[13] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[14] Oscar Vega-Gisbert, Jose E Roman, Siegmar Groß, and Jeffrey M Squyres. Towards the availability of java bindings in open MPI. In *Proceedings of the 20th European MPI Users' Group Meeting*, pages 141–142. ACM, 2013.

[15] Michael D Abràmoff, Paulo J Magalhães, and Sunanda J Ram. Image processing with ImageJ. *Biophotonics international*, 11(7):36–42, 2004.

[16] Johannes Schindelin, Ignacio Arganda-Carreras, Erwin Frise, Verena Kaynig, Mark Longair, Tobias Pietzsch, Stephan Preibisch, Curtis Rueden, Stephan Saalfeld, Benjamin Schmid, et al. Fiji: an open-source platform for biological-image analysis. *Nature methods*, 9(7):676–682, 2012.

[17] Imagescience.org, 10 2013.

[18] Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010.

[19] Nicolas Chopin, Pierre E Jacob, and Omiros Papaspiliopoulos. Smc2: an efficient algorithm for sequential analysis of state space models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 2012.

[20] Ba-Ngu Vo, Sumeetpal Singh, and Arnaud Doucet. Sequential Monte Carlo implementation of the PHD filter for multi-target tracking. In *Proc. Int'l Conf. on Information Fusion*, pages 792–799, 2003.

[21] Arnaud Doucet, Nando De Freitas, Neil Gordon, et al. *Sequential Monte Carlo methods in practice*, volume 1. Springer New York, 2001.

[22] John Geweke. Bayesian inference in econometric models using monte carlo integration. *Econometrica: Journal of the Econometric Society*, pages 1317–1339, 1989.

[23] Anwer S Bashi, Vesselin P Jilkov, X Rong Li, and Huimin Chen. Distributed implementations of particle filters. In *Proc. of the Sixth Int. Conf. of Information Fusion*, pages 1164–1171, 2003.

[24] Ingvar Strid. Parallel particle filters for likelihood evaluation in dsge models: An assessment. Technical report, Society for Computational Economics, 2006.

[25] Joaquín Míguez, Mónica F Bugallo, and Petar M Djurić. A new class of particle filters for random dynamic systems with unknown statistics. *EURASIP Journal on Applied Signal Processing*, 2004:2278–2294, 2004.

[26] Joaquín Míguez. Analysis of parallelizable resampling algorithms for particle filtering. *Signal Processing*, 87(12):3155–3174, 2007.

[27] Ömer Demirel and Ivo F Sbalzarini. Balanced offline allocation of weighted balls into bins. *arXiv preprint arXiv:1304.2881*, 2013.

[28] Ömer Demirel and Ivo F Sbalzarini. Balancing indivisible real-valued loads in arbitrary networks. *arXiv preprint arXiv:1308.0148*, 2013.

[29] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. mpijava: An object-oriented java interface to MPI. In *Parallel and Distributed Processing*, pages 748–762. Springer, 1999.

[30] Open MPI: Trunk nightly snapshot tarballs, 09 2013.

[31] Michael Lange, Gerard Gorman, Michèle Weiland, Lawrence Mitchell, and James Southern. Achieving efficient strong scaling with PETSc using hybrid MPI/OpenMP optimisation. In *Supercomputing*, pages 97–108. Springer, 2013.

[32] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[33] Ranger Virtual Workshop (Cornell University). Why hybrid? or why not?, 07 2013.

[34] Ömer Demirel, Ihor Smal, Wiro Niessen, Erik Meijering, and Ivo F. Sbalzarini. Piecewise constant sequential importance sampling for fast particle filtering. *arXiv preprint arXiv:1310.5541*, 2013.

[35] A. Akhmanova and C. C. Hoogenraad. Microtubule plus-end-tracking proteins: Mechanisms and functions. *Current Opinion in Cell Biology*, 17(1):47–54, 2005.

[36] Yulia Komarova, Christian O De Groot, Ilya Grigoriev, Susana Montenegro Gouveia, E Laura Munteanu, Joseph M Schober, Srinivas Honnappa, Rubén M Buey, Casper C Hoogenraad, Marileen Dogterom, et al. Mammalian end binding proteins control persistent microtubule growth. *The Journal of cell biology*, 184(5):691–706, 2009.

[37] William J Godinez, Marko Lampe, Peter Koch, Roland Eils, Barbara Muller, and Karl Rohr. Identifying virus-cell fusion in two-channel fluorescence microscopy image sequences based on a layered probabilistic approach. *Medical Imaging, IEEE Transactions on*, 31(9):1786–1808, 2012.

[38] Ihor Smal, E Meijering, Katharina Draegestein, Niels Galjart, Ilya Grigoriev, A Akhmanova, ME Van Royen, Adriaan B Houtsmuller, and W Niessen. Multiple object tracking in molecular bioimaging by Rao-Blackwellized marginal particle filtering. *Medical Image Analysis*, 12(6):764–777, 2008.

[39] I. Smal, K. Draegestein, N. Galjart, W. Niessen, and E. Meijering. Particle filtering for multiple object tracking in dynamic fluorescence microscopy images: Application to microtubule growth analysis. *Medical Imaging, IEEE Transactions on*, 27(6):789–804, 2008.

[40] X Rong Li and Vesselin P Jilkov. Survey of maneuvering target tracking. Part I. Dynamic models. *Aerospace and Electronic Systems, IEEE Transactions on*, 39(4):1333–1364, 2003.

[41] Jo Helmuth, Grégory Paul, and Ivo Sbalzarini. Beyond co-localization: inferring spatial interactions between sub-cellular structures from microscopy images. *BMC bioinformatics*, 11(1):372, 2010.

[42] Jo A Helmuth and Ivo F Sbalzarini. Deconvolving active contours for fluorescence microscopy images. In *Advances in Visual Computing*, pages 544–553. Springer, 2009.

[43] D Thomann, DR Rines, PK Sorger, and G Danuser. Automatic fluorescent tag detection in 3d with super-resolution: application to the analysis of chromosome movement. *Journal of Microscopy*, 208(1):49–64, 2002.

[44] Ömer Demirel, Ihor Smal, Wiro Niessen, Erik Meijering, and Ivo F. Sbalzarini. Adaptive distributed resampling algorithm with non-proportional allocation. *arXiv preprint arXiv:1310.4624, 2013*, 2013.