UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

HENRIQUE COLAO ZANÚZ

In-Transit Molecular Dynamics Analytics with Apache Flink

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia de Computação.

Orientador no Brasil: Prof. Dr. Claudio F. R. Geyer
Orientador na França: Prof. Dr. Bruno Raffin

Porto Alegre
2018

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitor: Profª. Jane Fraga Tutikian
Pró-Reitor de Graduação: Prof. Vladimir Pinheiro do Nascimento
Diretor do Instituto de Informática: Profª. Carla Maria Dal Sasso Freitas
Coordenador do Curso de Engenharia de Computação: Prof. Renato Ventura Henriques
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

PREFÁCIO

Este trabalho foi desenvolvido como projeto de final de curso para a universidade francesa INP (*Institut National Polytecnique*) de Grenoble, avaliado pela faculdade francesa ENSIMAG. Este projeto será utilizado, também, como trabalho de graduação do curso de Engenharia de Computação da UFRGS, no âmbito do acordo de dupla diplomação existente entre as duas universidades.

O presente trabalho contém um resumo estendido, em português, do projeto realizado no exterior, bem como o texto completo avaliado pela instituição francesa, em inglês.

Análise De Dinâmica Molecular *In-Transit* com Apache Flink - Resumo Estendido

Henrique C. Zanúz[1], Claudio R. Geyer[1]

[1] Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

{hczanuz, geyer}@inf.ufrgs.br

Abstract. *During its execution, Molecular Dynamic (MD) simulations output molecular trajectories that are analyzed by scientists to allow them to acquire a better understanding of the evolution of the system. Although parallel MD normally generate huge trajectories, scientists continue using sequential algorithms for their analysis, which quickly becomes impractical time-wise. Scientists keep on using this approach due to the complexity of implementing and deploying parallel workflows using standard high-performance computing (HPC) tools; tasks such as load balancing and data partitioning can be troublesome for scientists. Therefore, in this work, an on-line parallel analytics framework is proposed to process and store in-transit the MD trajectories, relying on Apache Flink, a scalable stream processing engine from the Big Data domain. Flink enables to implement analyses using a simple window-based map/reduce model, while Flink's runtime takes care of the deployment, load balancing and data distribution. A complete in transit analytics workflow was built, connecting an MD simulation to Apache Flink and to a distributed database, Apache HBase, to persistently store all the desired data. To demonstrate the expressivity of this programming model and its suitability for this domain of application, two common analytics in the MD field were implemented. The performance of this framework was assessed, concluding that it can handle simulations of sizes used in the literature while proving to be an effective and versatile tool for scientists to easily incorporate on-line parallel analytics in their current workflows.*

Resumo. *Simulações de dinâmica molecular (DM) geram trajetórias moleculares, que precisam ser analisadas por cientistas para que eles possam adquirir um melhor entendimento da evolução do sistema. Apesar de simulações paralelas de DM normalmente gerarem trajetórias enormes, os cientistas continuam utilizando algoritmos sequenciais em suas análises, o que se torna rapidamente impraticável em termos de tempo. Eles usam esta abordagem por causa da complexidade associada à implementação e à implantação (deployment) de análises paralelas que utilizam abordagens clássicas da computação de alto desempenho; questões como o balanço de carga e a distribuição de dados pode ser bastante complicado para eles. Portanto, neste trabalho, um framework de* *análise paralela de Dinâmica Molecular (DM) é proposto para processar e armazenar os dados gerados por uma simulação de DM utilizando nodos dedicados do mesmo cluster que executa a simulação. Este framework faz uso do Apache Flink, uma ferramenta escalável de processamento de Stream proveniente do mundo do Big Data, como elemento principal de processamento. O Flink permite ao usuário implementar análises utilizando um modelo de Map/Reduce baseado em janelas, enquanto seu runtime é encarregado da implantação, balanço de carga e distribuição dos dados. Construiu-se um sistema completo de análise in-transit conectando uma simulação de DM ao Apache Flink e a uma base de dados distribuída, Apache HBase, para o armazenamento dos dados desejados. Para demostrar a expressividade deste modelo de programação, foram implementadas 2 análises comuns nesta área. O desempenho do sistema foi avaliado, concluindo que ele é capaz de lidar com os tamanhos de simulações que são usados na literatura ao mesmo tempo que se mostra uma ferramenta versátil para os cientistas incorporarem análises paralelas aos seus estudos.*

1. Introdução

Simulações de dinâmica molecular geram grandes volumes de dados que têm de ser analisados para que os cientistas possam extrair informações relevantes da evolução do sistema. O tamanho destes dados, das trajetórias moleculares, é tão grande ao ponto de se tornar o fator limitante de desempenho, por causa do seu armazenamento e principalmente das suas análises [MURES et al, 2016]. No entanto, estas análises são normalmente feitas de maneira sequencial, já que são os próprios cientistas que as implementam e eles não possuem tanta expertise em computação, portanto possuem dificuldade para criar algoritmos paralelos, pensar na distribuição dos dados e no balanceamento da carga de trabalho.

Com base nisso, esse trabalho propõe um novo *framework* de análise paralela para DM, que utiliza como base o conceito de processamento de Stream, que vem sendo largamente utilizado no mundo do Big Data para analisar dados de negócio e facilitar a exploração de paralelismo dos códigos [CARBONE et al., 2015]. Nossa proposta, é, então fazer uso do Apache Flink [CARBONE et al., 2015] [Apache Flink] no contexto de alta performance de DM.

A seção 2 explica os conceitos envolvidos neste trabalho e realiza uma breve análise do estado da arte. Na seção 3, o *framework* proposto, sua construção e a implementação de duas análises comuns na área de DM são descritos. Na seção 4, são descritos os experimentos de avaliação do desempenho da solução criada. Por último, na conclusão, é discutida a aplicabilidade deste *framework* em situação real, considerando os resultados obtidos, e são descritos os trabalhos futuros.

2. Contexto

Esta seção apresenta os conceitos básicos sobre os quais este trabalho é desenvolvido, bem como a motivação e o problema a ser resolvido. Não somente isso, uma análise do estado-da-arte também é apresentada.

2.1. Simulações e Análises de Dinâmica Molecular

Simulações de Dinâmica Molecular (DM) simulam o movimento dos átomos e das moléculas em um sistema fechado. Através das leis da dinâmica de Newton, são calculados, para todo o intervalo de tempo, a posição, a velocidade, a energia cinética para cada átomo do sistema, assim como a interferência de cada átomo em seus vizinhos. Essas simulações possuem uma vasta gama de aplicações, do refinamento de estruturas proteicas nas áreas da química e da biologia, assim como no estudo de propriedades de nanodispositivos na física [TIANKAI et al., 2008].

Comumente, para essas simulações, é importante observar a evolução do sistema como um todo, ao invés de apenas se atentar a seu estado final. Para tal, os usuários podem especificar um intervalo de iterações a cada qual a simulação emitirá fotos do sistema. Denomina-se o conjunto de fotos de uma simulação de DM uma Trajetória Molecular. Analisá-las permite que cientistas adquiram um entendimento melhor do comportamento do sistema.

Por exemplo, em [ARYAL et al., 2015], os autores estudam o comportamento de portões hidrofóbicos em canais de íons, que funcionam como nanopóros nas membranas celulares, permitindo ou não a passagem de íons para dentro e fora da célula. Estes portões podem estar ou abertos ou fechados, estados que são determinados pela hidratação dos poros, ou seja, pela quantidade de moléculas de H_2O. Portanto, contar a quantidade de moléculas d’água nestas regiões nos permite inferir o estado do canal em um determinado instante.

Até a primeira década dos anos 2000, a maior parte dos avanços na área de DM estavam concentrados em otimizar a performance das simulações, através de paralelismo e de outras técnicas de software e de hardware. Isso tornou-as altamente paralelizáveis e escaláveis [ABRAHAM et al., 2015]. No entanto, não houve este mesmo esforço no que tange as análises das trajetórias, que continuam sendo majoritariamente sequenciais [MICHAUD-AGRAWAL et al., 2011].

Mais tarde, com o aumento da resolução que as simulações de DM conseguiam computar, o que culmina em mais átomos para processar e na geração de trajetórias maiores (com mais iterações), armazená-las e analisá-las começou a tornar-se o principal gargalo. Ou seja, com o aumento do tamanho das fotos do sistema e com uma frequência maior de captura, a pressão de E/S também cresceu. O problema de escalabilidade advém, então, de duas frentes: das baixas velocidades de escrita em sistemas de armazenamento persistentes e do tempo inviavelmente maior requerido para executar as análises sequenciais sobre estas trajetórias maiores [MURES et al., 2016] [TIANKAI et al., 2008].

Até então, as análises eram executadas de forma *post-mortem* [DREHER et al., 2014], de maneira offline. Ou seja, as trajetórias eram geradas e armazenadas em disco, para posteriormente serem lidas e analisadas. A solução mais simples tomada pelos cientistas para lidar com estas trajetórias maiores era, então, reduzir a frequência das fotos que eram feitas. No entanto, essa escolha impacta negativamente a capacidade de observar fenômenos de movimentação rápida, que não seriam mais capturados pelas fotos menos frequentes.

Como solução para esses problemas, o paradigma de análise *in-situ* [DREHER et al., 2014], atraiu o interesse da comunidade cientifica. Ele é caracterizado pela execução dos algoritmos de análise de trajetória nos mesmos nodos em que a simulação roda e de forma online, ao mesmo tempo que a simulação. Isto pode ser feito intercalando simulação e análise ou dedicando certos núcleos de processadores para esta. Para lidar com a pressão de entrada, o foco deste paradigma se baseia nos seguintes conceitos:

- Reduzir a quantidade de dados escrita em disco, devido ao fato de discos rígidos serem diversas ordens de magnitude mais lentos que a memória principal e o processador;
- Executar as análises de forma online e junto com a simulação permite que as análises rodem em uma escala similar à da simulação.

No entanto, programar análises *in-situ* introduz uma série de limitações e desafios [H. YU et al., 2010]. Já que a análise roda junto com a simulação e nos mesmos nodos, executar análises mais complexas, que demandem mais tempo, pode atrasar a simulação. A decomposição e particionamento dos dados podem não ser os mesmos para a simulação e o algoritmo de análise, dado que estes podem ter diferentes padrões de acesso, o que culmina em mais movimentação de dados. Outro fator complicador é que algumas análises podem necessitar de ajuste fino em seus parâmetros, dependendo da simulação realizada, que pode afetar diretamente a performance da análise; porém, os parâmetros ótimos não poderiam ser encontrados antes do término da execução da simulação.

Uma outra abordagem possível é a *in-transit* (em-trânsito) [DREHER et al., 2014], onde a análise e a redução dos dados também ocorrem de maneira *online*, porém nesta, isso é feito em nodos dedicados para a análise. Portanto, esta estratégia é levemente menos acoplada à simulação que a *in-situ*. Mesmo tendo o custo adicional da transferência dos dados para os nodos dedicados, essa abordagem oferece mais flexibilidade para o usuário. Ou seja, torna-se possível rodar análises mais complexas computacionalmente, já que não há competição pelos mesmos recursos de processamento que a simulação.

Mesmo estas duas abordagens sendo viáveis e bem conhecidas pela comunidade de processamento de alta performance (do inglês, HPC), elas não resolvem a dificuldade que os cientistas têm para implementar as análises que eles desejam e de maneira paralelizada. Na verdade, o oposto ocorre. Essas técnicas acabam por exigir mais expertise do programador que a técnica *post-mortem*, devido às questões envolvendo o balanceamento da carga de trabalho (*workload*), compartilhamento de recursos, temporização e movimentação dos dados.

2.2. Trabalhos Relacionados

Esta seção faz uma breve análise da evolução dos *frameworks* para análise de DM no que tange a usabilidade e a facilidade de utilizá-los para desenvolver novas análises.

Gromacs [LINDAHL et al., 2001] é um simulador de DM maduro e bastante conhecido neste meio. Mesmo não tendo as análises como seu foco, desde sua versão 3.0 ele é distribuído com diversos programas de análise *post-mortem*. Assim como a própria simulação, essas análises codificadas em C, sendo muitas delas já paralelas. No entanto, em termos de usabilidade, essa solução não provê nenhuma ferramenta que ajude os usuários a implementar novos algoritmos. Portanto, se um cientista necessita desenvolver uma nova análise, ele há de desenvolve-la do zero.

Por sua vez, MDAnalysis [MICHAUD-AGRAWAL et al., 2011] é um *framework* de análise de trajetória de DM feito para simplificar a criação de novas análises. Implementado em Python, ele explora o conceito da programação orientada a objetos para fornecer abstrações de mais alto nível para os usuários, como átomos, resíduos, trajetórias e entre outros. Além disto, ele ainda fornece diversos módulos prontos de sub-rotinas bastante usadas nas análises, como por exemplo, cálculos de distância. No entanto, o código que esse *framework* é capaz de gerar é estritamente sequencial. Para tornar estes códigos paralelos, faz-se necessário utilizar-se das técnicas convencionais da área da computação de alta performance, como Interface de Passagem de Mensagens (do inglês, MPI) e/ou OpenMP.

O *framework* Himach [TIANKAI et al, 2008] foi a primeira ferramenta da área a se basear no paradigma de Map-Reduce. Atraídos pela sua facilidade de uso, que mascara a distribuição dos dados e o balanço de carga para o usuário, os autores adaptaram este modelo de programação ao contexto de DM. Para tal, eles estabelecem uma nova abordagem para os algoritmos, que consiste em 3 fases: definição da trajetória, onde o usuário determina quais são as fotos de interesse; aquisição de dados e análise por foto individual, onde toda a informação potencialmente relevante é extraída das fotos, lembrando uma operação de Map e finalmente uma análise cruzando todas as fotos, assemelhando-se a um Reduce. Este framework utiliza MPI para implementar o paralelismo do Map-Reduce e possui boa performance e escalabilidade, segundo seus experimentos. Contudo, esta ferramenta só é capaz de gerar código para análises off-line (*post-mortem*).

Em [DREHER et al., 2014], os autores adicionam ao *framework* de análises FlowVR, a capacidade de gerar análises assíncronas *in-situ* e *in-transit*. Nesta adição, as análises são descritas através de um grafo de *dataflow*, conectando componentes. Neste grafo, os vértices são módulos e as arestas, a comunicação entre eles. Cada modulo é então um trecho de código em *loop* infinito que implementa uma interface contendo: uma operação de espera; uma operação de ingestão de dados e uma de saída de dados. Uma análise consiste, então, em descrever estes módulos e conectá-los. A ferramenta possui um *runtime daemon* que inicializa os módulos nos nodos desejados (podendo ser *in-situ* ou *in-transit*) e transmite as mensagens entre eles. No entanto, a distribuição dos dados entre os módulos ainda fica a cargo do usuário, o que se torna difícil para um cientista sem tanta expertise em programação paralela.

Em [PARASKEVAKOS et al., 2018], os autores comparam a performance de dois *task-parallel frameworks* de caráter generalista quando aplicados no contexto de trajetórias de DM. São eles: Spark e Radical-Pilot. Posteriormente, estas ferramentas são comparadas com a abordagem clássica de HPC, utilizando MPI. Nestas comparações, os autores escolheram duas analises *post-mortem*: a similaridade de trajetórias e o algoritmo *Leaflet Finder*. Os experimentos indicam que o Spark possui uma performance melhor quando as tarefas possuem bastante comunicação de dados ou algoritmos iterativos, devido aos seus *Resilient-Distributed-Datasets* (RDD) mantidos em memória. Por outro lado, Radical Pilot mostrou-se mais versátil devido ao seu mais baixo nível de abstração. Já, em relação as implementações em MPI, apesar de terem tido uma performance aquém, a diferença de performance não foi tão grande a ponto de inviabilizar o uso destas ferramentas de *Big Data* no mundo de HPC.

2.3. Apache Flink

Apache Flink [Apache Flink] [CARBONE et al., 2015] é uma ferramenta de *Big Data* para análise de dados, normalmente de negócios, de código aberto, feita para ser distribuída, de alta performance, altamente disponível e tolerante a falhas. O paradigma de programação do Flink é baseado um grafo de *dataflow* e provê operações similares ao Map-Reduce, permitindo aos usuários facilmente implementar algoritmos paralelos.

Por sua vez, Map-Reduce [DEAN et al.,2008] é um modelo de programação proposto inicialmente pelo Google que consiste em duas operações: Map, uma função que gera pares chave-valor e a função Reduce, que agrupa estes pares com base na chave e opera sobre os valores selecionados. Neste modelo, operações de Map podem ser executadas em paralelo, assim como as operações de Reduce.

Além de prover este modelo de programação, o Flink suporta o que é chamado de Processamento de *Stream*, utilizado em situações onde elementos novos chegam continuamente ao longo do tempo no sistema para serem processados, o que se encaixa perfeitamente no contexto das trajetórias de DM, onde gera-se um fluxo de fotos do sistema simulado.

Um programa para o Flink é, portanto, uma sequência de operadores por onde os dados de entrada passarão. Por sua vez, estes operadores são intrinsicamente paralelos, cujo paralelismo é explorado automaticamente pelo escalonamento dinâmico feito pelos componentes de *runtime* do Flink (o *Job Manager* e o *Task-Manager),* sendo totalmente transparente para o usuário.

Para que o usuário possa compor seus programas, o Flink fornece ainda outras abstrações que facilitam o desenvolvimento. Entre elas destaca-se a noção de janelas [Documentação Apache Flink, 2018]. Dado que um *stream* possa ser infinito, é necessário um mecanismo para agrupar os dados a serem operados, um gatilho que disparará a execução sobre os dados agrupados até então e marcas d'agua que definirão a passagem do tempo. Para tal, existem as janelas, que podem agrupar os dados por tempo, quantidade, sessão ou outros critérios implementados pelo usuário.

3. Framework de Análise Proposto

Dado o panorama apresentado na sessão 2, em que os cientistas têm necessidade de implementar análises paralelas, mas não conseguem devido as complexidades de fazer código em paralelo, mesmo utilizando *frameworks* (de abordagens *in-situ* e *in*-transit), devido a questões como distribuição e movimentação de dados, sincronizações e a própria logica algorítmica. A proposta deste trabalho é se fazer valer do Processamento de *Stream (*e Map-Reduce), que já é amplamente conhecido na área de *Big Data* e que surgiu para suprir estas mesmas necessidades – criar código paralelo-, para criar um *framework* de análise paralela de trajetórias moleculares que seja mais simples para os cientistas programarem, já que todo o paralelismo é extraído dos próprios operadores.

Foi construído então um *framework* utilizando como cerne o Apache Flink, devido aos fatos que fora concebido desde o princípio para lidar com *Streams*, diferentemente do Apache Spark, por exemplo, e que possui abstrações mais flexíveis que simplificam o desenvolvimento, como as diferentes noções de tempo e as janelas. Como a execução do Flink pode demandar muitos recursos das maquinas, optou-se por uma configuração *in-transit*, para evitar interferências do próprio Flink com a simulação, bem como para possibilitar que o usuário tenha mais liberdade e possa criar analises que sejam mais complexas computacionalmente.

Uma das principais preocupações tidas para este projeto foi mantê-lo o mais próximo possível de um cenário de utilização real de análise de DM, para tal foram utilizados os componentes que serão explicados nas seções 3.1 a 3.3. A seção 3.4 detalha a maneira como cada parte foi montada no sistema.

3.1. A Simulação: CoMD

CoMD [EXMATEX] é uma mini-aplicação que simula o comportamento e as operações de uma simulação real, porém oferece uma interface mais simples para sua uso. A versão utilizada neste trabalho utiliza MPI para explorar o paralelismo da aplicação, portanto faz uso de diversos processos com espaço de memória isolada. Cada processo é responsável por calcular o estado de um sub-conjunto de partículas, não havendo trocas de partículas entre dois processos.

3.2. A Comunicação: ZeroMQ

ZeroMQ [HINTJENS, 2013] é uma biblioteca de mensagens largamente utilizada no mundo HPC que suporta a comunicação entre diferentes linguagens de programação e fornece uma abstração de sockets de mais alto nível.

3.3. O Armazenamento: Hbase

HBase [HINTJENS, 2011] é um banco de dados distribuído projetado para *Big Data* que normalmente opera como uma camada de abstração sobre o Haddoop File System.

3.4. A Montagem Do Sistema

A Figura 1 mostra o sistema proposto e como ele é conectado. São utilizados três conjuntos separados de nodos, sendo todos do mesmo cluster: os nodos de simulação, onde os processos MPI do CoMD são executados; os nodos de análise, que são dedicados para a execução do Flink, e por último os nodos de armazenamento, que têm por responsabilidade executar o HBase e realizar a permanência dos dados.

Os processos do CoMD se conectam com uma *sub-task* do Flink, com base no seu ranking MPI, através de *sockets* do ZeroMQ. A cada determinado intervalo de iterações, cada processo envia em seu *socket* uma mensagem contendo seu ranking MPI, o número da iteração atual, um *array* de identificadores das partículas, das quais este processo é encarregado, e um *array* das respectivas posições.

Do outro lado, no Flink, cada *sub-task* recebe estas mensagens e emitem uma nova tupla (contendo as mesmas informações que a mensagem) em um único *DataStream* distribuído que está sendo gerado. No jargão do Flink, esta ingestão de dados é chamada de fonte. Neste momento, o *DataStream* gerado pela fonte pode ser manipulado como o usuário bem entender; é nesta parte que podem ser adicionados os *scripts* de análise.

Ao fim, o *DataStream* gerado ou manipulado pode ser encaminhado para o sumidouro do HBase, que faz a interface entre o Flink e a persistência no HBase, através de requisições de inserção para cada elemento.

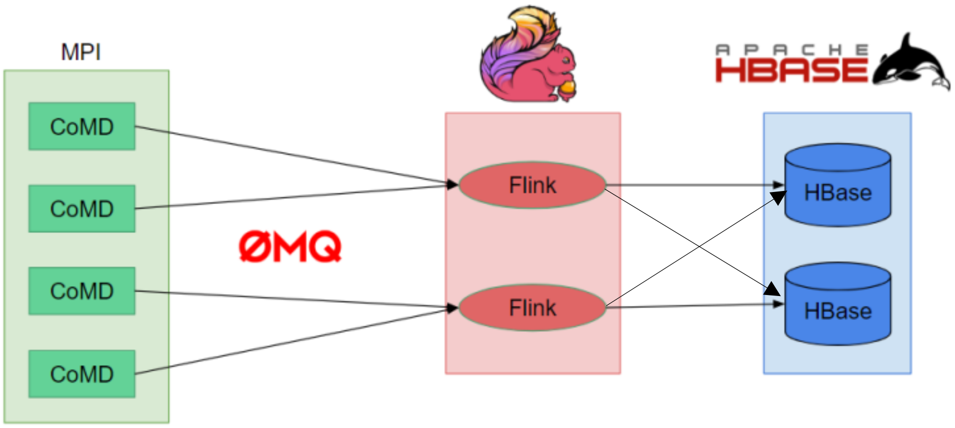


Figura 1 - Diagrama do sistema proposto. Cada forma geométrica com um rotulo inscrito representa um nodo.

Após a construção e validação deste sistema, buscou-se implementar algumas análises, como exemplos da expressividade e usabilidade do processamento de *Streams*

3.5. Análises Implementadas

Duas análises muito comuns nesta área foram desenvolvidas como exemplo de uso para este *framework*.

A primeira delas é um histograma de posições dos átomos, o que é um passo essencial para criar mapas de densidade. Para implementar esta análise, divide-se o espaço total em baldes de mesmo tamanho; em seguida, realiza-se a contagem de átomos em cada balde. O pseudocódigo referente a esta análise pode ser visto na Listagem 1a.

A outra análise desenvolvida foi a identificação de átomos vizinhos com base em uma distância de corte, que é um procedimento importante para identificar estruturas de átomos. Esta análise foi escolhida principalmente pelo fato de ser custosa computacionalmente e com um padrão de acesso aos dados bastante distinto da primeira análise. Um algoritmo ingênuo para esta análise consistiria em calcular a distância entre todos os átomos do sistema, com custo O(n^2). Para reduzir o número de operações, dividiu-se o espaço em cubos de lado igual a distância de corte. Isto permite inferir que para um dado cubo, partículas vizinhas só podem estar no próprio cubo ou nos cubos que fazem fronteira com ele. A partir deste ponto, associa-se cada átomo com cada cubo em que ele pode ter vizinhos, permitindo que ao agrupar por cubo, todos os átomos pertinentes já sejam corretamente separados. Em seguida, basta aplicar uma função para calcular a distância entre todas as partículas selecionadas. Na listagem 1b, vê-se o pseudocódigo para este algoritmo.

```
comdStream
  .FlatMap:
    Emitting one tuple for each atom
    <Timestep,Bucket Index, 1>

  .KeyBy( Bucket Index  )
  .Window by Timestep
  .sum(2)

              (a)
```

```
comdStream
  .FlatMap:
   For each atom a1 and each Neighbor Cell:
    emit Tuple <Atom a1, Cell-Index >

  .KeyBy( Cell Index)
  .Window by Timestep
  .apply( Select_Neighbor_Atoms )

              (b)
```

Listagem 1: Pseudocódigo do Histograma de Posições dos Átomos (a) e da Identificação de Vizinhos (b)

4. Avaliação de Performance

Com o intuito de mensurar o desempenho do framework proposto e testar a sua aplicabilidade em condições similares as encontradas na pratica, utilizou-se o *testbed Grid5000* [BALOUEK et al.,2013], que provê uma serie de clusters para serem usados em pesquisas da computação.

Mais especificamente, utilizou-se o cluster *Paravance*, localizado em Rennes na França, cuja configuração encontra-se na Tabela 1.

Hardware		**Software**	
CPU	2 x Intel Xeon E5-2630 v3	Sistema Operacional	Debian Jessie
Cores/CPU	8	Versão do Flink	1.3.3
Threads/Core	2	Coletor de lixo usado pelo Flink	G1GC
Memória	128 GB	Versão do HBase	1.2.6
Armazenamento	2 x 558 GB HDD	Versão do CoMD	1.1
Rede	2 x 10 Gbps	Versão do ZeroMQ	4.1.4
		Versão do Java	1.8
		Versão do Hadoop	2.7.6

Tabela 1 – Descrição do Hardware e dos Softwares usados.

Para os experimentos fixou-se os seguintes parâmetros: memória do Heap do Flink em 6GB e do HBase em 30GB; 32 milhões de átomos simulados, gerando trajetórias de 62 GB (2 vezes o tamanho definido para memória de Heap do HBase). 125 processos CoMD distribuídos em 8 nodos; 1 nodo para o Flink e outro para o HBase.

Para aferir o sistema proposto é crucial evitar o uso crescente de buffers ao longo da sua execução, já que as simulações de dinâmica molecular podem durar semanas e gerar enormes quantidades de dados. Portanto, se em uma determinada configuração a vazão de saída é menor que a de entrada, classificou-se o sistema como instável (ou desbalanceado), já que em algum ponto no futuro o sistema teria falta de memória ou precisaria atrasar a simulação; ambos seriam efeitos colaterais intoleráveis. Para exemplificar, as Figuras 2 e 3 representam configurações estável e instável, respectivamente.

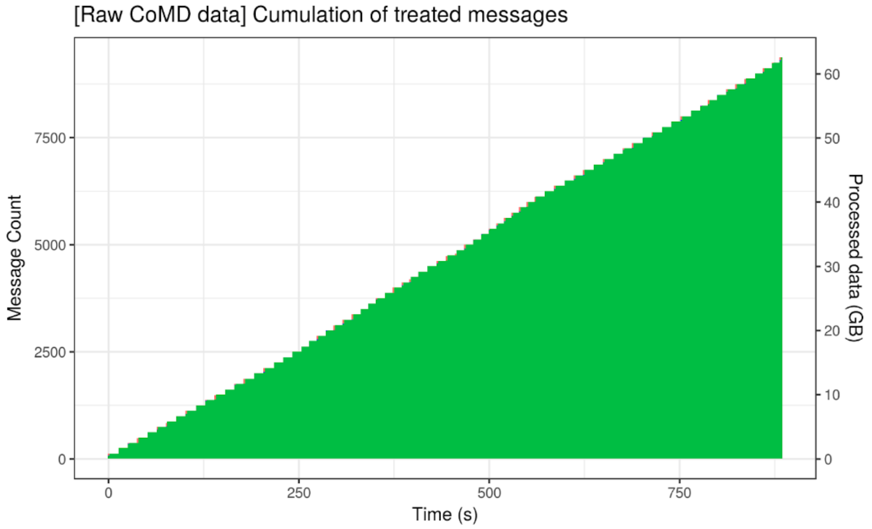


Figure 2- Contagem de mensagens processadas ao longo do tempo em cada parte do sistema em configuração estável. Em verde, HBase; em azul, Flink e em laranja CoMD.

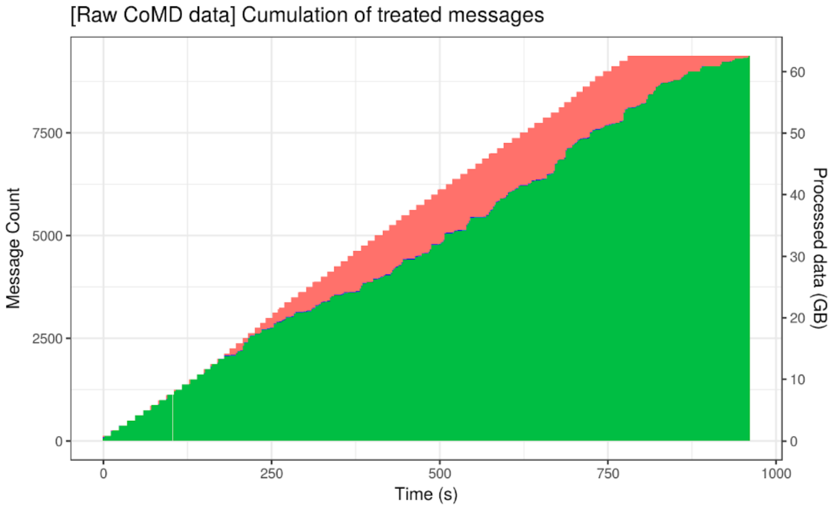


Figura 3- Contagem de mensagens processadas ao longo do tempo em cada parte do sistema com comportamento instável. Em verde, HBase; em azul, Flink e em laranja CoMD.

Buscou-se então medir a máxima vazão estável do sistema, para validar se a proposta é capaz de ser aplicada em condições de análise de trajetórias moleculares reais. Quatro cenários diferentes foram testados: "Sem analise" (*Raw CoMD Data)*, quando Flink apenas redireciona as mensagens para o HBase*; "*Análise do Histograma", adicionando o processamento do Histograma ao caso "Sem Análise"; "Identificação de vizinhos", computando a identificação de vizinho em cima do cenário "Sem Analise" e um experimento de escalabilidade fraca da análise do Histograma.

No cenário "Sem Análise", o sistema atingiu 71 MB/s, salvando 1 foto a cada 11 iterações; já no caso do Histograma, 78 MB/s (1 a cada 10 iterações). Este valor maior deve-se ao fato da imprecisão do experimento e da classificação de estabilidade manual dos pontos.

Normalizando estas vazões obtidas pelo número de nodos da simulação (8 nodos), chegamos a um valor de pressão de entrada de 8,87 MB/s/nodo. Em [DREHER et al,.2014], os autores testam o seu sistema de análise *in-transit* e *in-situ* com abordagem HPC com pressões de 2 MB/s/nodo, salvando 1 a cada 100 iterações. Portanto, o nosso framework consegue lidar com pressões muito maiores que o valor utilizado como *benchmark* para o *FlowVR*. Na prática, a situação é mais leve; as pressões são ainda menores, já que se salva somente 1 a cada 1000 iterações [DREHER et al,.2014]. Portanto, os números obtidos indicam que o sistema é capaz de lidar com pressões muito mais altas do que se está usando atualmente, o que viabiliza seu uso.

No teste de escalabilidade, o número de nodos, o tamanho da trajetória e o número de átomos simulados foram multiplicados 4. Os resultados indicam que o sistema não conseguiu atingir mais de 100 MB/s (1 a cada 30 iterações), quando esperar-se-ia 315MB/s se a escalabilidade fosse linear. Porém, com base nestes e outros experimentos, identificamos que o fator limitante é o HBase, já que verificamos que ao reduzir a pressão de dados armazenados no HBase a performance melhora bastante.

No caso da análise dos vizinhos, devido à grande quantidade de operações e elementos emitidos pelo algoritmo e da pequena quantidade de memória disponível para o Flink, limitada propositalmente, o sistema não foi capaz de lidar com uma simulação tão grande, o que nos obrigou a diminuir o seu tamanho para 256 mil átomos, que é o valor utilizado para aferir o mesmo tipo de algoritmo em [PARASKEVAKOS et al., 2018], porém neste artigo a análise é feita de maneira off-line. Já que os parâmetros de simulação não são os mesmos, uma comparação direta dos valores não pode ser feita. Os resultados mostram que o sistema tolerou 1,10MB/s, salvando 1 a cada 400 iterações. Apesar do valor absoluto ser baixo, o tamanho da simulação é condizente com a literatura e a frequência das fotos também é maior do que o utilizado pelos cientistas na prática, o que é mais um sinal que corrobora a sua aplicabilidade.

5. Conclusão

Foi proposto um novo *framework* para análise de trajetórias de dinâmica molecular *in-transit* que abraça o conceito de Processamento de Stream, através do uso do Apache Flink como elemento central. Este, o Flink, é utilizado principalmente em contextos de análise de dados de negócios, e não em contextos de HPC. Porém, a principal vantagem de utilizar o Flink sobre o estado da arte atual é a facilidade de implementação de analises, já que toda a distribuição dos dados, comunicação e paralelismo são mascarados pela plataforma, de forma que se tornem transparentes ao usuário, no caso aos cientistas (biólogos), que não possuem tanta expertise em computação.

Construiu-se esse sistema utilizando uma mini-aplicação de simulação de DM, CoMD, uma biblioteca de mensagens, ZeroMQ e uma base de dados distribuída, HBase. Foram implementadas duas análises como exemplos do poder de expressividade da ferramenta.

Avaliou-se sua aplicabilidade em situações reais através da aferição da vazão máxima que o sistema pode suportar. Os experimentos mostraram que o sistema é capaz de lidar com pressões muito maiores do que a comunidade vem utilizando para testar seus *frameworks*, porém ainda é necessário melhorar a escalabilidade do sistema, que tem como gargalo o armazenamento. O HBase, apesar de ser distribuído, mostrou-se tanto como fator limitante da vazão tanto quanto pela escalabilidade.

Outros sistemas de armazenamento podem ser utilizados, em investigações futuras, como o Apache Cassandra ou mesmo um sistema MemCached. No entanto, ter uma base de dados ou um componente de persistência com indexação de dados se faz necessário para permitir que os cientistas realizem consultas com filtros (predicados) sobre as enormes trajetórias ou mesmos sobre os resultados das análises

Referências

ABRAHAM, Mark James et al. 2015. *Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers*. SoftwareX, 1-2:19 – 25.

APACHE. *Apache Flink 1.4 Documentation*. https://ci.apache.org/projects/flink/flink-docs-release-1.4/, 2018. Online; acessado 2018-22-05.

APACHE. *Apache Flink*. https://flink.apache.org, 2018. Online; acessado 27-10-2018.

ARYAL, Prafulla et al. 2015. *Hydrophobic gating in ion channels*. Journal of Molecular Biology, 427(1):121 – 130. Understanding Functions and Mechanisms of Ion Channels.

BALOUEK, Daniel et al. *Adding virtualization capabilities to the Grid'5000 testbed. E*m *Cloud Computing and Services Science, volume 367 of Communications in Computer and Information Science*, páginas 3–20. Springer International Publishing, 2013

CARBONE, Paris et al. 2015. *Apache flink™: Stream and batch processing in a single engine*. IEEE Data Eng. Bull., 38(4):28–38, 2015

DEAN, Jeffrey et al. 2008. *Mapreduce: Simplified data processing on large clusters.* Commun. *ACM*, 51(1):107–113, Janeiro 2008.

DREHER, M. et al. 2014. *A flexible framework for asynchronous in situ and in transit analytics for scientific simulations*. Em 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, páginas 277–286, May 2014

EXMATEX. *CoMD*. https://github.com/ECP-copa/CoMD, 2016. Online; acessado 2018-28-05.

GEORGE, Lars . *HBase - The Definitive Guide: Random Access to Your Planet-Size Data.* O'Reilly, 2011.

H. YU et al. 2010. *In situ visualization for largescale combustion simulations*. Em IEEE Computer Graphics and Applications, 30(3):45–57, May 2010.

HINTJENS, P. *ZeroMQ: Messaging for Many Applications*. Oreilly and Associate Series. O'Reilly Media, Incorporated, 2013.4

LINDAHL, Erik et al. 2001 .*Gromacs 3.0: A package for molecular simulation and trajectory analysis.* 7:306–317, 08 2001.

MICHAUD-AGRAWAL, Naveen et al. 2011. *Mdanalysis: A toolkit for the analysis of molecular dynamics simulations*. J. Comput. Chem., 32: 2319-2327. doi:10.1002/jcc.21787.

MURES, Omar A et al. 2016. *Leveraging the power of big data tools for large scale molecular dynamics analysis*, 2016. XXVII Jornadas de Paralelismo (JP2016).

PARASKEVAKOS, Ioannis et al. 2018. *Task-parallel analysis of molecular dynamics trajectories*. *CoRR*, abs/1801.07630.

TIANKAI Tu, et al. 2008. *A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories*. Em 2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis, páginas 1–12.

Master of Science in Informatics at Grenoble
Master Informatique
Specialization Parallel, Distributed and Embedded Systems

In-Transit Molecular Dynamics Analysis with Apache Flink

Henrique Colao Zanúz

25th June, 2018

Research project performed at INRIA

Under the supervision of:
Bruno Raffin

Defended before a jury composed of:
Martin Heusse
Olivier Gruber
Vania Marangozova-Martin
Bruno Raffin

June 2018

Abstract

Parallel Molecular Dynamic (MD) simulations generate an enormous amount of data throughout their execution. At each determined interval of time steps, they output a snapshot of the molecular system they are computing. This snapshot is called a frame and the collection of frames is called a trajectory. The trajectories need to be saved and analyzed to allow scientists to infer more information from them. Traditionally, the simulations are executed first, to only then, when they finish and the trajectories are already saved on disk, computing the analytics. This approach is called *Post-Mortem* Analytics. As the MD simulations improved, thus being able to handle more atoms and computing more time steps, the output and its analysis, usually done sequentially, have started to become the bottleneck. Solving this bottleneck requires parallelizing the analytics, running them on-line and in a similar scale. This can be done *in-situ*, using the same nodes as the simulation, or *in-transit*, by using staging nodes. However, implementing these techniques with standard HPC tools (such as MPI) can get really complicated for the scientists, since it is now necessary to think about data partitioning and load balancing. Our idea is to leverage Apache Flink, a task-parallel analytics engine from the Big data world, in this HPC context. Benefiting from its easier Map-reduce-like programming model to allow scientist to easily extract parallelism from their analytics algorithms. We have built a complete in-transit analytics framework, connecting a MD simulation to Apache Flink and to a distributed database, Apache HBase. To demonstrate the expressibility of this programming model, we have implemented two common analytics, a position histogram and the identification of neighbor atoms. We assessed the performance of this framework, concluding that it can handle simulations of sizes used in the literature.

Résumé

Les simulations parallèles de dynamique moléculaire (DM) génèrent une énorme quantité de données tout au long de leur exécution. A chaque intervalle de temps déterminé, ils produisent un instantané du système moléculaire qu'ils calculent. Cet instantané s'appelle une image et la collection d'images s'appelle une trajectoire. Les trajectoires doivent être sauvegardées et analysées pour permettre aux scientifiques d'en déduire plus d'informations. Traditionnellement, d'abord les simulations sont exécutées, ensuite faire l'analyse. Cette approche est appelée Analytique *Post-Mortem*. Au fur et à mesure que les simulations MD s'améliorent, ce qui permet de traiter plus d'atomes et de calculer plus de pas de temps, la sortie et sa analyse, généralement effectuée séquentiellement, ont commencé à devenir le goulot d'étranglement. Pour résoudre ce goulot d'étranglement, il faut paralléliser les analyses, les exécuter en ligne et à une échelle similaire de la simulation. Ceci peut être fait *in-situ*, en utilisant les mêmes nœuds que la simulation, ou *in-transit*, en utilisant des nœuds de staging. Cependant, la mise en œuvre de ces techniques avec des outils HPC standard (tels que MPI) peut s'avérer très compliquée pour les scientifiques, puisqu'il est maintenant nécessaire de penser au partitionnement des données, à l'équilibrage de charge. Notre idée est de tirer parti d'Apache Flink, un moteur d'analyse parallèle aux tâches du monde des grandes données, dans ce contexte HPC. Bénéficier de son modèle de programmation de type Map-reduce-like pour permettre aux scientifiques d'extraire facilement le parallélisme de leurs algorithmes d'analyse. Nous avons construit un cadre d'analyse en transit complet, reliant une simulation MD à Apache Flink et à une base de données distribuée, Apache HBase. Pour démontrer l'expressivité de ce modèle de programmation, nous avons mis en œuvre deux analyses communes, un histogramme de position et l'identification des atomes voisins. Nous avons évalué le rendement de ce cadre, concluant qu'il peut traiter des simulations de tailles utilisées dans la littérature.

Acknowledgement

I would like to express my sincere gratitude to Professor Bruno Raffin, to Professor Emilio Padrón and to Professor Claudio Geyer for their invaluable assistance during this research.

Contents

— 1 —
Introduction

Molecular dynamics (MD) simulations emulate the motion of the atoms in a closed system. For that, they compute the position, the speed, the energy, the accelerations, the forces on each atom and how they affect the other atoms. The simulations are configured in order that at each *t* time steps computed they store a snapshot of the molecular system, this is called as a *frame*. A collection of frames defines a *trajectory*.

Computational biologists have interest on the frames and the trajectories to be able to study the phenomena that happen in the system, which would not be possible to resolve if only the final result was stored. They might want to perform multiple analysis on these data to add additional information to the trajectories.

MD simulations are highly parallel, distributed and optimized. However, traditionally, the trajectory and frame analysis are done in a *post-mortem* approach, i.e, storing the frames on local disks, on the simulation nodes, and executing the analysis later [7]. Furthermore, these analysis would frequently be computed sequentially. As the simulations improve, thus being able to cope with more and more atoms and longer simulations periods, the output started to become the bottleneck, slowing the simulations down and making the analysis time prohibitively long [19, 15].

One of the possible solutions to this is sending the data to other nodes, using high speed network links, to perform the analytics on-line before storage. This technique is called *In-transit* [7] analytics. By using dedicated extra nodes, it is possible to perform complex analysis and deal with storing at the same time. However, computational biologists may have difficulties when implementing their analysis with this technique using classical high performance computing (HPC) approaches, such as MPI, due to the complexity associated to the data partitioning, communications, synchronizations and simultaneously using task programming (e.g. OpenMP).

Therefore, this work aims to solve this issue by leveraging Apache Flink and its stream processing programming model to provide an easier way of implementing parallel analysis, so that scientists can be able of developing themselves any analytics they want.

Apache Flink [5] is a distributed stream processing framework from the Big Data world, usually used for computing analytics over streams of businesses data. In our case, we use it to process the streams of frames outputted by MD simulations. In spite of being highly scalable, fault tolerant and conceived from scratch to work with stream-based data, its main characteristic, for this work, is its programming model, the dataflow programming paradigm, which is based on assembling operators to describe the programs. This higher level of abstraction allows the scientists to program their parallel analysis without having to take care of complicated details, such as data distributions and communication.

Flink's programming model not only provides Map and Reduce operators, typical in the Big Data world, but it also offers a wider range of transformations for the programmer to use, such as join , split, filter functions, etc. Flink's engine is capable of extracting the underlying parallelism from the data flow by itself. The user does not need to explicitly tell where and when each computation will take place, nor the communication of data. This makes of Flink a powerful and easy-to-program tool.

Nonetheless, as Flink is a Big Data tool, it may not privilege the same characteristics as an HPC tool does. For instance, Flink has emphasis on fault tolerance, through storing checkpoints and being able to restore from those backed-up states, while in an HPC point of view this tends to be neglected, due to its costs in performance, even though it is starting to became increasingly important when going to exascale machines.

Therefore, in this work we have built a complete on-line parallel analytics framework for MD with an in-transit approach, and assessed Flink as the core component of such a framework. The framework built uses Flink as processing engine, Apache HBase as storage system, which is a non-relational distributed data-base, and ZeroMQ messaging framework for making the connection with a simulation. To assess the viability of this framework, we have connected it to CoMD, a MD proxy application that emulates the regular workload in MD simulations. In this way, the over-all set-up consisted in using multiple dedicated nodes of a cluster to run CoMD (simulation), Flink (analysis) and HBase (storage). Later on, two common analytics in this field were implemented to demonstrate the expressibility of Flink's programming paradigm: a position histogram of atoms and the identification of neighbor atoms.

For performance evaluation purposes, we assessed the maximal throughput the system can handle when Flink just relays the data to HBase and when it also computes the analytics. A scalability experiment was done, by scaling the system by a factor of 4 and measuring the maximal throughput it achieved.

Results indicate that the system was able to handle the pressure of big simulations, around 32 million atoms, achieving throughputs of 78 MB/s while also computing the Histogram analytics. A thorough analysis of data collected during the experiments indicates the bottleneck of the system was HBase.

This report is organized as follows: chapter 2 details the context and concepts this work involves and also describes the related work; chapter 3 presents the framework built, its components and analysis implemented; chapter 4 describes the experiments done as well as the performance evaluation of the solution and, finally, the conclusion chapter makes a brief summary of this work, as well as analyses the results obtain in perspective of the literature and describes the further works.

— 2 —
Context

This section presents the context in which this work is inserted in and the main concepts it relies on. Subsection 2.1 introduces the MD simulations and the trajectory analysis; section 2.2 presents an overview of related work, presenting frameworks that were developed to address the problem of creating parallel analysis and section 2.3 describes the key component which is used in our framework proposal, Apache Flink

2.1 Molecular Dynamics Simulations and Analytics

Molecular Dynamics (MD) simulations simulate the motion of the atoms in a closed system. Through Newton's Laws of motion, they compute all the necessary information to characterize the state of each atom, position, speed, forces applied to it, kinetic energy and how it affects the other atoms. They have a wide range of applications, from refining the structure of proteins in chemistry and biochemistry to studying the proprieties of nanotechnological devices, in physics [19].

Commonly, for these simulations, it is interesting to keep track of the evolution of the systems instead of only storing the final state. Therefore, the users can specify the rate, in terms of a number of time steps, with which the simulation emits its data. The data of one time step is called a frame, and a collection of frames describes a molecular trajectory. Analyzing these frames and trajectories allows the scientists to acquire a better understanding of what is happening in the system.

For instance, on [3], they study the behavior of hydrophobic gates in ion channels, which act like nanopores on cell membranes that regulates the passage of ions in and out. These gates can be either opened or closed, which is determined by the pore hydration, the amount of water molecules inside the pore. Therefore, by making a water count on those regions of the system, it is possible to enrich the results of the simulation by having the information of the state of the channels.

Until around the first decade of the XXI century, most of the improvements made in the MD field were to optimize the performance of the simulation itself, through parallelization and through the use of other software and hardware techniques. This made these simulations highly parallelized and very scalable[1]. Meanwhile, as the trajectory analysis did not received the same parallelization and optimization effort, they continued to be mostly sequential[14].

Later on, with the resolution increase the simulations were capable to handle, which means more atoms to process and bigger trajectories (more time steps computed) being generated, storing and analyzing the trajectories started to become the bottleneck. This was caused by the growth of the frames size and the increased frequency of the simulation (number of time steps computed per second), which generated more I/O pressure. The scalability problems came from two fronts: the slower speeds of the persistent storage systems and the prohibitively longer times needed to perform the sequential analytics over the bigger trajectories [19, 15].

At that time, the analytics were performed *post-mortem* [7], in a off-line way, meaning that the simulation would generate the trajectories that would be first stored to disk, to later be read for the analytics. The simplest solution taken by scientists to reduce the I/O pressure was reducing the frequency of snapshots of the system. However, this approach negatively impacts the capability of observing fast moving phenomena, that would no longer be captured by the less frequent snapshots.

To cope with this, the *in-situ* analysis paradigm started to draw interest of the community [20]. It is characterized by computing the analysis in-site, inside the simulation nodes on helper cores, and in an on-line manner, happening while the simulation is running. To deal with the input pressure, the focuses of this new approach were based on the following concepts:

- Reducing the size of the data stored on disk, since hard drives are several orders of magnitude slower than the main memory and the CPU.

- Computing the analysis on-line and alongside the simulation, allowing the analysis to run in a similar scale as the simulation itself and reducing the total amount of time necessary to run the simulation and the analytics *a posteriori*. One other advantage of this approach is the ability to get feedback and partial analytics results from the simulation while it is still running.

However, computing analytics *in-situ* introduces a series of constraints and challenges [20]: As it is running with the simulation, complex analysis that take more time to be computed might slow the simulation down. The data decomposition of the simulation and the analysis algorithms do not always match, since they might have very different data-access patterns, generating more data transfers. Some analyses might require tunning input parameters specifically to the simulation that would change the overall performance of the algorithm, but the optimal values are not known prior to the end of the simulation.

Another different approach is called *in-transit*[7]. Here, the analysis and data reduction are also processed on-line, but in staging nodes, which do not process the simulation. This is a slightly more decoupled version of *in-situ*. Despite paying the price of the raw data transfer between the simulation nodes and the analysis ones, this offers more flexibility. For instance, more complex analytics might be performed since the hardware is not being shared with the simulation processes.

Even though these two different approaches are viable and are already well known in the field, this does not solve the difficulty scientists have to implement the analytics they want. In fact, the opposite happens. These techniques end up requiring more expertise of the programmer than the classical *post-mortem* strategy, due to workload, data distribution, shared resources and timing issues.

2.2 Related Work

This section covers a brief review of the evolution of the MD analytics frameworks, when it comes to easing the developer of the analysis programs.

Gromacs [12] is a well known, mature and open source molecular simulation application for the field of chemistry and biology. Even though it is not its focus, since its 3.0 version, its distribution also comes with several trajectory analysis programs. These algorithms are for *post-mortem* analytics. As the simulation application itself, these analysis tools are mostly coded in C, but only a few of them are parallel.

In terms of usability, each analysis is a different program. No framework for developing new analytics was provided. Therefore, if a scientist needs a different type of analysis, this user would have to code it from scratch.

In turn, MDAnalysis [14] is already a MD trajectory analysis toolkit designed to simplify the development of new analytics. It is implemented in Python and has some performance-critical routines implemented in C. This toolkit is object-oriented, providing the abstractions of atoms, residues, segments, trajectories, time steps and so on, as objects. Not only that, MDAnalysis provides several modules to ease the implementation of analytics. For example, there are packages to compute distances, densities, to select groups atoms based on a specific characteristic, etc.

Even though MDAnalysis succeeds offering a higher level abstraction for implementing MD trajectory analytics, when it comes to parallelizing the algorithms, it relies on the same traditional complicated approaches, MPI and/or Open MP. Also as Gromacs, it is designed for *Post-Mortem* analytics.

In terms of performance, the authors compare the framework's performance with Gromacs and other two analysis softwares: VMD and CHARMM. They analyze the execution time of 6 different analytics over the same trajectory. MDAnalysis was faster than the others when computing simple calculations, however on cases with more complex operations, where NumPy is used, the performance starts to degrade. They claim this could be improved by using Cython or C on the critical paths.

Himach [19] was the first MD analysis framework to provide parallel execution capabilities based on Google's Map-Reduce. Attracted to its ease of use, since it masks the data-distribution and the load balancing for the user, they decided to apply the same paradigm to the MD field. To use this programming model, they established a new of design for the analytics algorithms, based on 3 phases:

The first one is the Trajectory Definition, where the user determines the frames of interest. Instead of specifying a time-series of frames, this creates a set of frames that will be processed, allowing not only for out-of-order processing but also exposing parallelism between them. The second phase is called 'Per-frame data acquisition and analysis', where all the potentially useful data from the frames are extracted. This phase resembles the map of Map-Reduce. Finally, the third part is the cross-frame analysis, which behaves like a reduce. This last part may also be conducted more than once depending on the algorithm the user wants to implement.

Himach is implemented in Python.

Himach has a run-time which is responsible for assigning the tasks to the processors, coordinating the parallel I/O requests, storing and managing intermediate values (temporary key-value pairs) and orchestrating the data exchange between processes. Another important aspect of Himach is that it uses MPI calls (through the PyMPI library) for the interprocess communications and does so in a distributed way, instead of having Map-Reduce's single master node (centralized) controlling the progression of the execution and communications. Thus, avoiding this as a bottleneck.

Their results showed that this solution scales almost linearly (around 90% of the linear curve) until 32 MPI processes, but from that point the I/O and the communications start to degrade the scalability, reaching only 45% of the linear curve when with 512 processes.

Despite its reasonable scalability and good performance, this approach is still for off-line and post-Mortem analytics only.

FlowVR [7] is a framework for asynchronous *In-Situ* and *In Transit* analytics. With this framework, the analytics are programmed as a data-flow graph, by assembling components together. On this graph, the edges are communication channels, and the vertices are the modules. The description of these data pipelines is also done in Python.

Each module is a piece of code running an infinite loop and implements an interface containing: a wait operation, that suspends the module until it has new input data available; a get operation, to fetch new input data; and a non-blocking put operation, for outputting data.

Given the modules, the application script simply consists in describing how the data go through them and how they are connected. FlowVR uses a run-time daemon, that runs in each node, for deploying the application in the nodes and for relaying the messages from module to module, according to the application description. As these modules can run in different nodes, it is the deamons' responsibility to manage the communications between nodes.

Therefore, by using FlowVR, scientists can build more complex analytics set-ups, mixing in-situ and in-transit scenarios, while counting with the abstraction it offers in terms of synchronizations for the communications.

One of the analytics the authors have implemented using this framework is a QuickSurf algorithm, which computes a iso-surface from a volumetric density map. For instance, one module of this algorithm is the computation of Morton indices (or Z-Order index), which is also used later on in this work.

In addition to the previous algorithm, the authors have also integrated MDAnalysis and Gromacs analysis tools as modules for their framework, even though, individually, these tools are still sequential, requiring whole frames as input.

In this paper, when they experiment their Quicksurf implementation, they run it in similar scales to those being used by the in-situ community to benchmarks their frameworks. Therefore, the simulation they use to generate their input runs on 128 nodes, where each node generates 2 MB/s.

On [17], they compare three already existing general purpose task-parallel frameworks, Spark, Dask and RADICAL-Pilot, with respect to their ability of supporting *post-mortem* MD analytics on HPC environments. They also assess them in comparison to classical HPC MPI approaches.

Apache Spark is a Big Data framework for data analytics which extends Map-Reduce by providing a set of transformations over Resilient Distributed Datasets (RDD). This data structure is cached in memory, thus being well suited for iterative computations. A Spark job is a sequence of multiple stages, which means a series of local operations (e.g. maps) followed by a distributed action (e.g. reduce). The dataflow generated by the combination of these transformations and actions is translated into an execution plan by Spark's DAGScheduler.

Dask is a parallel computing library for Python. Differently from Spark, it also provides a lower-level task API (called ´Delayed API') that allows build more complex execution graphs.

Radical-Pilot is a Python implementation of the Pilot systems paradigm, which consists in decoupling the workload specification, resource selection, and task execution via the declaration of job placeholders and late-binding for the execution of the applications [13].

For the comparison, they have implemented 2 different *post-mortem* analytics with each framework: the Path Similarity of groups of atoms based on the evolution of their pair-wise distances along the time steps and the Leaflet Finder algorithm, to identify bi-lipid membranes, which involves discovering all the neighbor atoms (based on a cut-off distance) and identifying the two biggest connected components.

After assessing the scalability of each framework and each analysis, they conclude that Spark outperformed Dask when it comes to communication intensive tasks and also on iterative algorithms, due to the in-memory RDDs. Dask's lower and higher level APIs proved to be more versatile than Spark, although both have presented some limitations, requiring work-arounds. Radical-Pilot proved to be more useful for coarse-grained task-level parallelism and when it was necessary to integrate other existing HPC analytics frameworks (like MDAnalysis).

Even though none of them out-performed their MPI counter-part, their easier programming paradigm and not so big performance gap still creates a strong case for using them for HPC analytics.

2.3 Apache Flink

This section introduces Apache Flink, which is the main component of the framework proposal of this work.

Apache Flink [2, 5] is a Big Data open source framework for processing analytics on business data, in the form of streams and also batches of data. It is meant to be distributed, high-performing, highly available, fault-tolerant, and accurate.

Flink is Java-based and supports programs (scripts) written either in Java, Scala or even in Python (with some limitations). Its programming model is based on data flow and provides Map-Reduce-like [1] operations, allowing users to easily implement parallel algorithms.

The concept of stream refers to data that are continuously being produced, ingested and processed. Streams can be either bounded (finite) or unbounded (infinite). For example, the data stream generated by a set of meteorological sensor is unbounded, while the one generated by an aircraft during one flight is bounded.

Flink uses the paradigm of stream processing to unify batch processing, real time processing and stream processing itself. A batch can be seen as a specific case of a bounded stream that triggers the computation only once. However, in practice, Flink provides a different API for batches (called Data Sets), to benefit from some possible optimizations and to improve compatibility with legacy applications which see streams as a sequence of batches [5]

[1] Map-Reduce [6] is a programming model for processing large data-sets in parallel, proposed by Google. Programs of this model implement a Map function, that generates key-value pairs, and a Reduce function, that groups the pairs by key and operate over the list of values. On this model, the maps may occur in parallel and so does the reduction for each key.

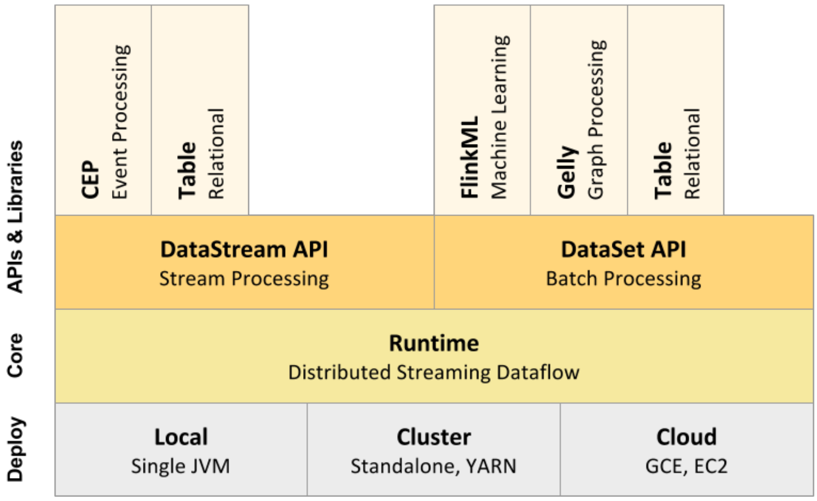


Figure 2.1 – Diagram of Flink's software stack [2]

Figure 2.1 depicts the software stack of the framework. Its core layer, the run-time, is a distributed streaming dataflow engine, which run flink (dataflow) jobs. More specifically, every flink job is a Directly Acyclic Graph (DAG) of stateful operators linked to input and output streams. Not only the Batch and Stream processing APIs are different, but they also count with different sets of additional libraries[5]. On top of DataSet, there is *Flink ML*, for machine learning applications and *Gelly* for graph processing. On top of Datastream, there is the *Complex event processing* library (for detecting event patterns in unbounded streams). Both processing modes provide a *Table API* for SQL integration.

Figure 2.2 shows how the three main Flink components work. A Flink program is transformed in a dataflow graph and is submitted by the Client to the Job Manager. The Job Manager is the process responsible for coordinating the distributed execution of the programs. It keeps track of the state of each operator, of the progression of each stream, it coordinates the checkpoints (for fault-tolerance) and schedules new operators. However, the data processing itself occurs in the Task Managers (TM), the workers. There is one TM for each Flink worker node. Each of them is executed by a different Java Virtual Machine and perform the operations over the streams. They also report their status to the Job Manager, so that it can keep track of the state of the system. Each TM provides one task slot (or sub-task) for each core in its CPU.

By the time the DAG of the job is computed, the possible parallelization of the operators are taken into account. At specific points of the execution, the streams might be split into stream partitions that can be distributed or re-assigned to other sub-tasks. Therefore, the user does not need to explicitly control the distribution of the data.

To process the streams and their operator chain, the TM uses intermediate streams and pools of reusable buffers. Each operator consumes the input stream from a buffer, processes it and then serializes the output stream into another buffer. As these buffer pools are shared between every task of each TM, this ends up providing a natural way of dealing with *Back pressure*. Back pressure is the name given to the situation where some operator is not able to consume and compute the data at the same rate as it is receiving, i.e. a bottleneck; therefore, the previous operators in the chain must be slowed down, for the system not to collapse. In other words, when such an event happens, more buffers will be used to store the input of the slow operator. As the amount of buffers is shared by tasks in a TM, the other operators run out of free buffers to produce their output stream, thus halting the pipeline until the buffers are freed.

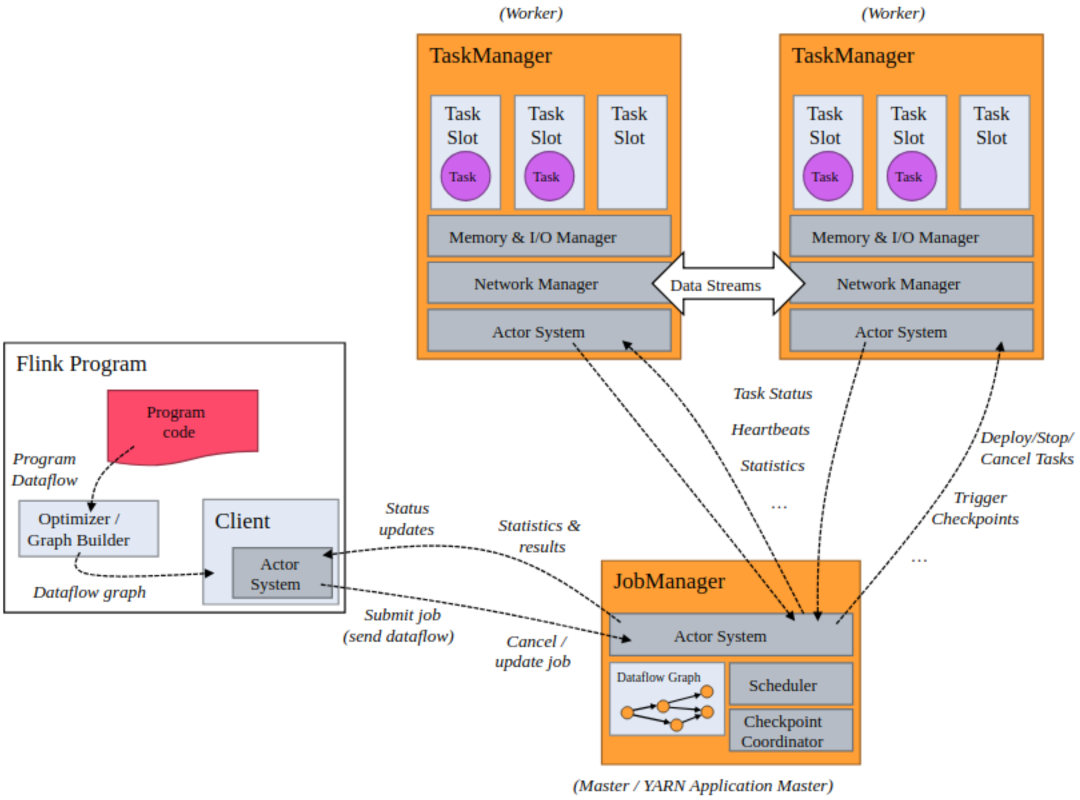


Figure 2.2 – Diagram of the Flink's architecture [2]

2.3.1 DataStream Operators

A Flink program is composed by, at least, a data source, a data pipeline and a data sink. The data source ingests the data into Flink. The data pipeline describes operators that the data will pass through. The data sink is the output of a stream. Therefore, a Flink program is expressed by assembling operators and chaining transformations.

Table 2.1 describes the most important operators for data transformation and Listing 2.1 shows an example of Flink program. This code is a simplified version of the Word Count implementation bundled in the Flink distribution [2]. Line 9 represents the data source, generating a DataStream, while line 20 is a sink to the standard output. As the reader can observe in this code, the code is agnostic to the data distribution and communication. Flink's run-time is responsible from inferring this information by itself.

Listing 2.1 – Word Count Example

```
1 public static void main(String[] args) throws Exception {
2
3     // set up the execution environment
4     final StreamExecutionEnvironment env = StreamExecutionEnvironment.
          getExecutionEnvironment();
5
6     String inputFile = "inputData.data";
7
8     // read the text file from given input path
9     DataStream<String> text = env.readTextFile(inputFile);
```

```

    DataStream<Tuple2<String, Integer>> counts =
    // split up the lines in 2-tuples containing: (word,1)
    text.flatMap(new Tokenizer())
    // group by the tuple field "0" (word)
    .keyBy(0)
    .window()
    //sum up tuple field "1" (accumulator)
    .sum(1);

    //Output to stdout
    counts.print();

    // execute program
    env.execute("WordCount Example");
}
```

In addition to the operators presented on Table 2.1, Flink also provides a wide range of different data sources and sinks.

2.3.2 Windows and Notions of Time

Due to the continuous character of the streams, Flink needs a way to group data together in terms of time, to know how the time passes (watermarks) and when to fire the computations. Differently from Apache Spark, that groups the records in *resilient distributed dataset* (RDD), Flink provides the abstraction of Windows, grouping data based on a time-based criteria.

Flink offers a lot of flexibility regarding the types of windows. They may gather data by a certain time interval, which is called window length, and may slide by a different amount, overlapping or not. A *tumbling window* has fixed length and does not overlap. A *sliding window* allows the overlapping of windows. Other types of windows are: *Counting window*, which gathers per quantity of elements; *Session window*, which assigns the elements to the windows regarding the time passed since the previous element and the *Global window*, which gathers every element in single window.

Orthogonal to the type of the windows, Flink provides 3 different notions of time for them to work with: *Ingestion time*, when the record was first created inside Flink's engine (e.g., by a data source); *Processing time*, when it is processed by each operator and *Event time* which represents the time when the source of data, outside Flink, created it. The latter offers more flexibility to the user, since the information used to assign an event timestamp to the elements might not be a timestamp, and actually, any value that the user might want.

Once Flink is able to group data into windows, it still needs to know when it should trigger the execution of the operators for that window or wait for more elements to come. This is done through *Watermarks*, which do nothing more that signaling the end of the current window, meaning that Flink can fire the execution of operators. Emitting watermarks is of crucial importance when it comes to implementing your own data-source, since all the down-stream operators will depend on this information to work correctly. Flink also provides ways to deal with late elements, elements that came after a watermark has been emitted. In this case, however, all the pipeline will be re-executed, thus it requires the down-stream sinks to be idempotent.

Table 2.1 – Transformations available in the Data Stream API. This table is based on the official Flink documentation web page [2].

Map	Takes each element of the stream and produces a new one. The datatypes may be different
FlatMap	Takes each element of the input stream and emits zero or more elements.
Filter	Filters the elements of the input stream
KeyBy	Partitions the DataStream, grouping elements with the same key value
Reduce	Rolls over the elements, combining the current element with the last reduced value, through pre-aggregations (before the window is fired)
Fold	Waits until the window is completed and fired, by buffering the elements. Combines each element with the previous folded value. It is more flexible then a reduce operation since the folded value may be a custom object and pre-initialized
Aggregation	Similar to reduce but only for these operations: min; max; sum; minBy; maxBy and sumBy
Window	Used only on KeyedStreams, groups the data according to their time charachteristic
WindowAll	Used only on non-KeyedStreams, groups all the data according to their time characteristic. This is frequently a non-parallel transformation
Window Apply	For KeyedStreams. Applies a custom function over the data grouped by the window and the key. Allows access to every single element
WindowAll Apply	Same as Window Apply, but for non Keyed Streams.
Window Reduce, Window Fold, Aggregation on Windows	Same as their other counterparts, but work on windowed streams
Union	Merges two streams of the same type
Window Join	Analogous to an SQL left join.
Connect	Connect two streams of different types
Window CoGroup, CoMap, CoFlatMap	Similar to their regular counterparts, but for connected streams
Split	Separates one stream in one or more sub-streams
Selects	Recover one sub-stream after a split function
Iterate	Allows the creation of a loop by feeding back elements to the original stream, that originated the loop.
Extract Timestamps	Extracts the timestamps of an element. This is usually used when working with event time

— 3 —
The Framework

The idea of this work is to use Apache Flink to perform in-transit molecular dynamics analysis. Our specific interest on Apache Flink comes from being able to leverage all the benefits it offers to the Big Data world, bringing them to the MD analytics context. Some of its characteristics are: easy-to-use task programming model, stream processing capability, low latency, distributed and scalable.

Indeed, Apache Flink is not alone as Big Data analytics engine, Apache Spark is more mature tool and is largely used in this industry. Both engines process streams and batches. However, as opposed to Spark, Flink was meant to work with streams from its very beginning, offering more powerful and flexible abstractions (e.g. different notions of time and window operators), better memory management and lower latency with higher throughput.

We propose to use of Flink for in-transit MD analytics in order to bring its qualities to this HPC context. For that, we have built a complete analytics framework that resembles as close as possible a real scenario used for molecular dynamics. Figure 3.1 depicts it. Each component will be detailed from section 3.1 to 3.3. Following that, we explain how they were assembled together (section 3.4) and we describe the implementation of two common MD analytics in section 3.5.

3.1 The simulation: CoMD

CoMD [8] is a proxy application for MD simulations. It was created by ‘The Exascale Co-Design Center for Materials in Extreme Environments (ExMatEx)’ team.

In the high performance computing (HPC) field, proxy application stands for a simplified version of a real application. Even though they are smaller, these mini-apps share the same characteristics of the original program in terms of operations, of workload and of work balance. Therefore, they are performance models of the real applications, used when the result of the simulation itself is not the end-goal, but instead only the simulation’s characteristics are interesting, which is the case of this work.

In practice, CoMD relieves the user of having to provide several input parameters that in a regular simulation it would be necessary, e.g. the initial configuration of atoms, requiring much less information to be able to run.

CoMD offers 4 different approaches for computing the simulation: A sequential version; One using Open MP ; one using MPI and another combining MPI and OpenMP. In this work, we have used exclusively the MPI version . In CoMD-MPI, each MPI process computes a fixed and different set of atoms.

The initial reference for the system CoMD computes is a face-centered cubic (FCC) lattice of copper atoms. CoMD replicates and stacks them in each axis of the 3D space to increase the number of particles on the system. Each FCC unit is called a unit cell.

Through command line parameters, it is possible to tune the system the CoMD will compute. The main parameters used on this work were: the parallelism of the execution, the number of unit cell per axis(nx,ny,nz), the number of processes (or tasks) on each axis(xproc,yproc,zproc), the total number of time steps to compute and the frame output interval.

In terms of processes and workload, to perform a strong scalability study, it is necessary to keep the total problem size fixed and compare the execution time while increasing (or decreasing) the number of processes. Therefore, it is necessary to keep nx,ny,nz constant while changing xproc,yproc,zproc. To perform a weak scalability test, where each process has a fixed workload nx.ny,nz should follow the changes in xproc,yproc,zprox.

3.2 The communication: ZeroMQ

ZeroMQ [10] or ØMQ, is a high-performance asynchronous messaging framework used for interconnecting applications and designed to be easy to use. It provides its own sockets, which are able to send and receive atomic messages over different transport layers, such as TCP, multicast, inter-process and in-process communications. These sockets can be connected in a wide range of configurations, 1-1, N-N and N-M, forming different communications patterns, such as ‘Request-reply’,‘Pub-Sub’,‘Push-Pull’ (pipeline) and ‘Exclusive pair’.

From those, the connection pattern ’Push-Pull´ was the one used in this work. The push sockets are the senders and the pull ones represent the receivers. This allows for a one way N-M connection (N senders, M receivers). When a message is pushed, it will be sent to all the available Pull sockets and it will be kept in a queue . Upon executing a Pull, the next message not-yet-pulled by any other pull socket will be removed from the queue and delivered to the calling application. Thus, despite of having N-M connections, only the first receiver to pull the message will read it. This message ends up being discarded from all the other receivers’ buffers. In other words, two receivers never get to read the same messages.

More precisely, on this work, we have implemented a N-1 Push-Pull communication pattern. Therefore, multiple producers may push and only one consumer will pull the messages.

In terms of the content of the messages, ZeroMQ is agnostic to the datatype, meaning that it transfers only byte blobs and it is the user’s duty to parse the information correctly on each end-point of the communication.

3.3 The storage system: Apache HBase

Apache HBase is an open-source distributed database based on Google’s proprietary ‘BigTable’. It is a non-relational database that runs on top of another distributed file system, most commonly the Hadoop File System (HDFS). HBase has near-optimal write performance, when it comes to I/O channel saturation, and has an excellent read performance, according to [9].

HBase, by its own, does not provide any declarative query language support and it is not compliant with SQL, however it is possible to use it in combination with Apache Phoenix or Apache Trafodion to obtain similar features.

HBase can be seen as distributed, sparse, persistent and multidimensional sorted map. This map is indexed by a row-key, a column-key and a timestamp. Each table in HBase is a set of rows, which are alphabetically sorted and identified by a row-key. Each row contains columns. Each column is actually a column family, which might contain multiple sub-columns, which are called 'Column Qualifiers'. Together, the column family and the column qualifier form the column name (i.e 'column_family:column_qualifier'). The number of column qualifiers a column family has might change from row to row.

Each value (and each update of a value) inside HBase has a timestamp assigned to it, which can be assigned by the user directly or even automatically when the put request is handled by the system.

The worker nodes in HBase are called Region Servers. Each region server contains an arbitrary number of regions. Each region is responsible for storing rows of one specific table, based on an interval of row-keys. The actual content of the rows are stored in HFiles on the underlying File System. Who coordinates which region server is responsible for which region and their row-key intervals is the master node.

Even though HBase has master and slave nodes (Region Servers), it does not mean that all the requests go through the master and are later processed by the slaves. Each HBase client maintains a cache of the Meta table, which is used to keep track of the servers, the region identifiers and the interval of row-keys accepted by each region server. This allows the clients to directly communicate with the correct worker nodes, without having to go through the master node every time. The master node is used for handling administrative requests, managing the regions assignments, flushing data and so on.

Regarding HBase's parallelism, when HBase is set to split the tables automatically, it stars by creating one single region. When this region gets bigger than the threshold (defined by the parameter 'hbase. hregion. max. filesize'), it splits the current region in two (its content and its row-key interval), then the master node assign these new regions to the region servers it considers convenient. This means that, in the automatic mode, HBase might take some time to start distributing the data.

To avoid that, it is possible to pre-split the table right from the beginning. However, the drawback is that it is possible to create hotspots when the row-key intervals are not correctly balanced. Even though, once these pre-split regions reach their maximal size, the region and its row-key interval continues to be automatically split. As a side-note, the standard automatic split does not offer any guarantee on load balancing, except when a custom region split policy, specified by the user, is used.

It is also important noticing that the choice of the row-key has a direct impact on the overall performance. Typically, sequential keys (such as time-series) yield a bad performance since the put requests get directed to a single region, not benefiting from parallelism. Therefore, the best scenario is when the row-keys are randomly uniformly distributed. However, this might not be possible in some cases. One possible work-around is salting the key by prefixing it with a more random information.

3.4 Putting the Pieces Together

Figure 3.1 depicts how the system is connected. The nodes in green run CoMD-mpi, CoMD parallelized with MPI. In red, there are the Flink worker nodes. In blue, there are the HBase region server nodes, which are responsible for storing the data received by Flink.

In addition to the nodes shown in the figure, there is an extra node, the master, which has the role of Flink's and HBase's master nodes. Even though, these services are centralized, they are not in charge of the actual processing of the data, but instead only managing the server nodes of both services. For instance, as explained in the previous sections, on HBase's case, the master node is responsible for mapping the key intervals to the slaves and not actually treating the request and storing the data.

On Flink's side, each Flink worker node runs one Task manager, providing one Task Slot per core. The master node runs the single Job Manager. The Job Manager has full autonomy to chose where and when to schedule the tasks on the workers. On this system, Flink is responsible for receiving the data generated by CoMD, processing it and sending to the storage nodes.

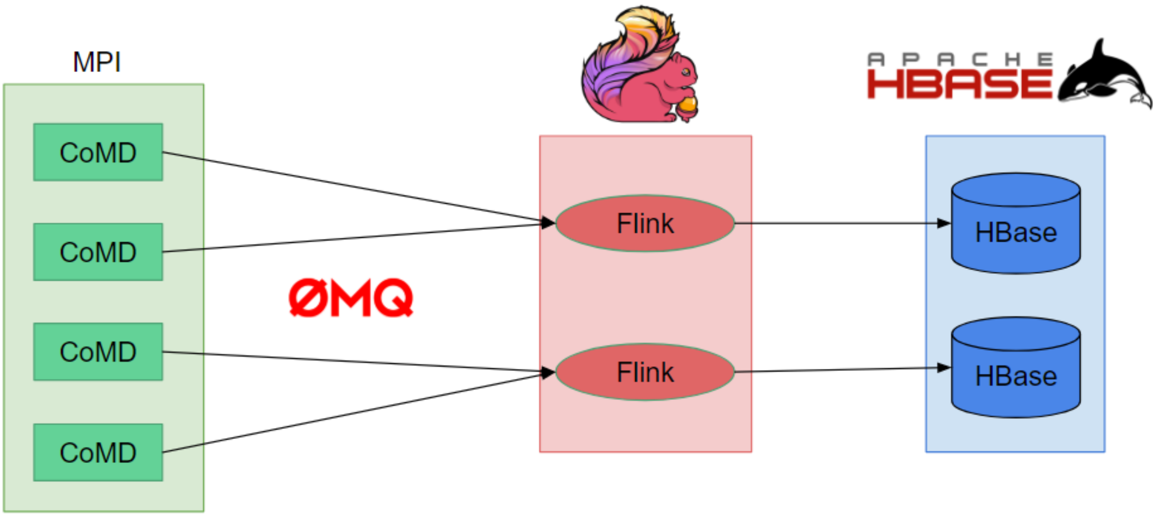


Figure 3.1 – Diagram of the set-up made. Every labeled figure denotes one separate node. CoMD refers to the MD simulation, Flink for nodes running the stream processing engine; HBase the storage nodes and ØMQ (ZeroMQ) to the messaging framework.

To connect all of those components together, it was necessary to implement a Flink program and to slightly modify CoMD. The first point was making the CoMD nodes communicate with Flink's ones, through ZeroMQ, and then connecting Flink to HBase.

3.4.1 Connecting CoMD to Flink

We decided to use ZeroMQ to connect the CoMD MPI processes to Flink. As Flink did not have a ZeroMQ source by the time this work was being developed, we had to develop our own connector. The implementation of this data source had to deal with lower level aspects, such as threads and the partitioning of the ZeroMQ connections throughout every Flink sub-task.

To better understand how the data source was implemented, it is important to explain the interaction protocol we have defined for Flink and CoMD. They communicate through ZeroMQ push-pull sockets, where Flink is the consumer (pull) and CoMD, the producer (push). This framework has as parameter a port interval, that is used by Flink to bind the consumer sockets and by the CoMD processes to establish the communications channels. In the set-up phase, each Flink sub-task picks a port number from the interval received as parameter, based on its sub-task ID. Then, they write a file, named after the chosen port number, on a known location in the underlying network file system (accessible by all the other nodes), containing the hostname of the node that picked that port number. This is done to allow CoMD to discover the hostname of the node responsible for each port. After writing this file, the Flink sub-tasks open a new ZeroMQ socket in pull mode, bind them to the chosen port and start waiting for messages.

On CoMD's side, each MPI process computes the port number it should use based on the port interval passed as parameter and its MPI rank. Then, they read the files that correspond to their port number to find out the address of the Flink's sub-task responsible for that port, which allows them to open a ZeroMQ connection with them, in pull mode. It is important to notice that due to the nature of this application, more than one CoMD process might connect to the same port. From that point, the simulation starts computing the time steps. At each given interval of time steps, the simulation is configured to output its current frame. For that, each CoMD MPI process sends a message to Flink through the socket.

The content of the messages is: its MPI rank (an int); the simulation time step which the data refers to (an int); the current timestamp (long); an array of the atoms' IDs (int[]) and an array of the position of the atoms (double[][3]). Benefiting from an abstraction provided by ZeroMQ, the multi-part messages, each information sent is sent separately, through consecutive send calls. However, ZeroMQ buffers them together and only sends them when the final message part is given - the position of the atoms. This helps mitigating the latency effects when multiple short messages are sent, optimizing the transfer bandwidth. Nonetheless, on the consumer side, Flink still has the illusion of receiving multiple messages, thus requiring multiple receive calls. Another important feature from ZeroMQ that we benefit from is the atomic messages. Even though there are several producers sending multi-part messages, from the consumer point of view they - the messages - do not get scrambled when reading the socket. This means that when a consumer starts to read the first part of a multi-part message, the subsequent read operations will return the next parts of the same message. Only after finishing reading an entire message, it will be able to read the pieces from another one.

On Flink's side, when a Flink sub-task receives and reads a complete message, it emits a tuple containing the same content and an extra timestamp (of the moment the message was read by Flink). The emitted tuples get assigned to the same distributed stream, meaning that each sub-task will receive different messages which, in turn, will generate elements on a single stream that has its content distributed throughout the Flink nodes. Thus, from the user point of view, it is a single stream, not requiring him or her to be aware of the fact that data are, in practice, spread throughout the Flink cluster.

One important aspect of developing a Flink source is to set up correctly the notion of time and how it progresses. On this project, the notion of time used was 'Event Time', using the simulation's time step as timestamp.

Expressing how the time progresses is done through the watermarks. They symbolize that from that point on that window is already over, firing the execution of all the windows up until that timestamp. Due to the characteristics of our design, where one socket might receive messages from several clients (CoMD MPI processes), each sub-task must be aware of the number of clients communicating with it and keep track of number of messages received for each time step. Thus, emitting a watermark whenever it received all the expected message for each time step. Flink is capable of managing the coherence of the watermarks between the sub-tasks and also the downstream operators in a transparent way. The user only needs to describe how the watermarks are emitted considering one single sub-task. In practice, Flink keeps track of the watermarks from all the upstream operators (and sub-tasks), so that it only triggers the execution of the window when this operator has already received the expected watermarks from all of its input sources.

As a side note, Flink provides a mechanism to deal with late elements, but it requires the subsequent sink operators in the chain to be idempotent, since the operations pipeline might be execute multiple times.

This completes the description of how Flink ingests the CoMD contents. From this point, the stream is ready to go through Flink's data pipelines, finishing in the HBase sinks.

3.4.2 Connecting Flink to HBase

Differently from ZeroMQ, HBase had already an official connector available, therefore implementing the sink was more straight-forward. It was only necessary to extend the 'OutputFormat' class in combination with the HBase API, specifying how each element from the stream would be outputted and how to configure, to open and to close the connections. In general, what the implemented code does is generating a put request for HBase for each element that the sink consumes.

During this work three HBase sinks were implemented, one for storing all the trajectories' data and two others for storing the results of the analytics we have implemented. In this section, only former will be detailed, while the latter will be explained in the next section.

Basically, the trajectories' data is stored in one table where the rows are identified by a key of the form :" MPI Rank + _+time step". This salted key allowed for easy recovering of the key while making sure of spreading the row throughout the HBase's region servers. Each row has one column family, called 'data', with 4 different qualifiers. The qualifiers identify the content of the column; they are: 'id','array' (for the positions); 'sent_timestamp',for the timestamp got from CoMD ; 'received_timestamp', for the moment when Flink received the message.As this content is stored on the same column family, the data from one row is not distributed through the HBase's region servers, staying together in only one.

Regarding the configuration of HBase, the trajectories' table is pre-split in up to 10 pre-splits, so that it starts distributing the put requests between the region servers right from the beginning. The distribution of the row-key values throughout the 10 splits was done so that each split is responsible for storing all the rows that start with one specific decimal digit (the first one of the message's MPI rank, in this case). In practice, this is very effective when the number of MPI processes is power of 10 (i.g 100), otherwise it unbalances the workload a bit. However, this is still a better option than using auto-splitting, since it only splits when the file reaches a threshold and it does not have information on the pattern of the row-keys. In addition, the table was set to store information on disk-only, even though it does use buffers and caches to process the operations, and it was also set to not compress the data.

3.5 Analytics

The main motivation for applying Flink in the context of Molecular Dynamics analytics is its easy-to-use dataflow programming model. To exemplify its expressiblity, we have implemented two commom analytics in the MD domain: a position histogram and the identification of neighbor atoms. The histogram is an important analysis for computing density maps, while the identification of neighbors is useful to identify molecule structures.

3.5.1 Position Histogram

The idea of this algorithm is to divide the space into non-overlaping bins and count the number of atoms on each of them for every time step. The complexity of this algorithm is O(n), where n is the number of atoms.

In order to split the space into disjoint bins, we chose to use the Morton code (Z Order Curve) [16], which is a classical approach on this field and also on computer graphics. It is a way of indexing a higher dimensional space in 1D, filling its space while following a Z-shaped curve, as seen in Figure 3.2. The Z-index of a position in a 3D space is obtained by interleaving the bits of each axis. The number of bits that were interleaved designates the order of the Z Curve. For example, the point in binary (001 , 010, 100) is mapped to 001 010 100.

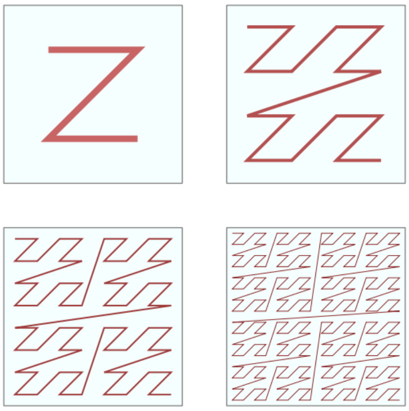


Figure 3.2 – Z-Order Curves in a 2D space of degrees 1,2,3 and 4, respectively

To keep the code independent of the total maximal space the simulation can occupy, since it is an unknown value for Flink, the discretization of the space is done through converting the double values to integer (by applying a floor function), applying a mask on them to select only the N least significant bits and then computing the Morton code with them. The mask is used to control the total amount of bins. This way, we obtain a more balanced workload distribution on each bin and we keep the algorithm agnostic of the size of the system. If we haven't done so, the system size would need to be a parameter and be properly set for each different CoMD run.

Having explained how the algorithm works, it is now possible to describe the implementation, shown in a summarized version on Listing 3.1 and complete in Appendix A.1. After obtaining the message input stream from the ZeroMQ source (in line 2), a flat map operator (in line 10) is used to split the messages that contains arrays of atoms into one element per atom, computing the its Z-index and add an extra integer field to allow the counting, set to 1. From that point, in line 19, it is just a matter of keying the stream by (grouping by) the Z-Index, establishing a 'Tumbling Event Time Window´ of size 1 and reducing the stream by summing the count values.

Listing 3.1 – Short version of the Histogram analysis' code with output on the standard output. The complete version is on Appendix A.1

```
1  //Get the Message stream from ZeroMQ
2  DataStream<Tuple6<Integer , Integer , Long ,Long , byte[] ,byte[]>> text =
       env
```

```
        .addSource(new ZeroMQBinaryStreamFunction(hwm, port, nports, -1,
            configFilePath, nComdClients))
.name("Queue")
;

/* Compute Histogram */

//Messages are split into Atoms (Tuple3<Timestep, ZIndex, Count>)
DataStream< Tuple3<Integer,Long,Integer> > atoms = text
      .flatMap(new MessageParser())
      .name("Split atoms");

int timestep = 0;
int zIndex = 1;
int count = 2;
long windowSize = 1;

        DataStream<Tuple3<Integer, Long, Integer>> analyticsResults = atoms
                    .keyBy(zIndex)
        .window(TumblingEventTimeWindows.of(Time.milliseconds(windowSize)))
                        .sum(count).name("Count atoms")
                        .project(timestep,zIndex,count)
                        ;

//Print the results in the standard IO
analyticsResults.print();
```

As it is possible to see in the code snippet above, Flink abstracts the key-value pairs from the elements it processes. This simplifies the code for the user, since he or she only needs to provide the number (or name) of the fields used as key a value.

In this algorithm, the flat maps are local operations. The keyBy operator, which in the batch processing context is called groupBy, is exploited by Flink's runtime to extract parallelism from the code for the subsequent sum operation. Each Key value, will be assigned and sent to a sub-task based on a hash function. This assignment persists throughout all the windows that are processed. Thus the window operator does not influence the parallelism, it just groups the data according to its timestamp. The 'sum´ operator is, therefore, also a local operation, not requiring data exchange between sub-tasks.

Regarding the output of this analytics, 2 possibilities were implemented: printing the output in the standard output, which is the one seen in the presented snippet, and writing to a separate HBase table, called 'flink_analytics´. This table has a key of the format 'Z-index_time step´ and its splits follow the same logic as the ones of the trajectories' table, each one is responsible for storing the rows that start with one specific decimal digit. Therefore, by having the Z-index first, the rows get spread throughout the different regions.

3.5.2 Neighbors Identification

Identifying neighbor atoms is another common analysis in the MD world. Useful for identifying groups of atoms, this is one of the major steps to identify atom or molecule structures, for instance the Leaflet Finder algorithm presented in [14], identify the neighbor atoms in order to locate the two leaflets that form a lipid bilayer.

The base-line for the neighbor identification algorithm is computing the distances between all the atoms and using a cut-off distance to determine when the atoms are effectively neighbors. The complexity of the naïve approach for this algorithm is O(N^2) in terms of distance computations, where N is the number of atoms in the system. To reduce this amount of operations we divide the molecular system domain in a 3D grid, re-using the Z-Order curve, introduced on the previous section, as an acceleration data structure. By making the side size of the bins in this grid equal to the cut-off distance R, we make sure that the only possible neighbors atoms are located inside itself and the neighbors bins (or Z cells), thus relieving us of computing useless distances between atoms that are far away from each other. Figure 3.3 depicts the relevant neighbor bins for the yellow bin, in a 2D scenario for clarity. No single point in the yellow bin may be closer than R and not be included the union of yellow and blue cells. In 3D, each cell has 27 neighbor cells.

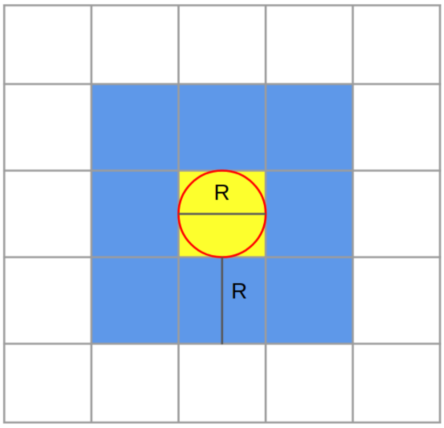


Figure 3.3 – 2D Domain discretization with cut-off distance equal to the bins's side length.

For mapping the positions to the bins, some modifications on the discretization model and Z index encoding used for the histogram analysis were necessary. Fixing the bins side length to R required the code to be aware of the maximal extent of the simulation space, becoming an extra parameter. By knowing the maximal space-size and the cut-off distances, it was possible to deduce how many cells are there in each dimension and the precise bin location in Cartesian coordinates for every possible position. The bins' cartesian index were then encoded to acquire the Z-order index, by interleaving the 21 least significant bits of each coordinate, which results in a 63-bit Z-index. We continued to use the Morton encoding, instead of using the Cartesian coordinates, for the only purpose of keeping this algorithm closer to what would be implemented in MPI. On those cases Morton indexes are frequently used to sort the data, allowing for better exploiting the data locality due to its Z shaped passing.

Not only the neighbor identification analytics has a much more expensive computational cost and less local character, in comparison with the position histogram, but it also demands a more complex data distribution and communication pattern when parallelizing it with regular approaches, MPI or OpenMP. Some issue of the possible issues are orchestrating the propagation of the neighbor cells, mapping more than one cell the same MPI rank and translating the cell identifier to the responsible MPI process.

However, when implementing this analysis in Flink, by composing a dataflow of operators, the resulting algorithm is much simpler and all of those issues becomes transparent to the user.

Listing 3.2 – Pseudo code for the Neighbor Identifying Analysis

```
  DataStream<Tuple2<Long, Atom>> atoms = messages.flatMap(new EmitAtoms());

  final int zIndex = 0;

  DataStream<Tuple2<Integer, Integer>> neighbors =  atoms
          .keyBy(zIndex)
   .window(TumblingEventTimeWindows.of(Time.milliseconds(windowSize)))
   .apply( new ApplyCutOffDistance() );


  DataStream<Tuple2<Integer,Integer>> count =neighbors.map(new MapFunction<
      Tuple2<Integer,Integer>, Tuple2<Integer,Integer>>() {

   @Override
   public Tuple2<Integer, Integer> map(Tuple2<Integer, Integer> arg0)
       throws Exception {
    // TODO Auto-generated method stub
    arg0.f1 = new Integer(1);
    return arg0;
   }

  })
  .windowAll(TumblingEventTimeWindows.of(Time.milliseconds(windowSize)))
  .sum(1);

   count.print()
    .setParallelism(1)
    .name("Sink to stdout");



}
```

The pseudo-code on Listing 3.2 summarizes the implemented algorithm. Basically, the CoMD messages are split into tuples <Z-index; Atom>,the latter object actually contains the atom's ID and position. A set of all the neighbors Z-cells of this atom is computed from their Cartesian coordinates. For each Z-index associated with this particle, a tuple <Z-Index,Atom> is emitted. Until this point, all operations are local. Next, the stream is keyed by the Z-Index, which implies data movement based on a hash function over the z-index, it is windowed by the simulation's time step and a Window Function is applied to compute the distance between each pair of atoms. The window function allows the user to freely iterate over the content of the keyed window. If the distance is smaller than the cut-off distance, a tuple containing both atoms' ID is emitted, symbolizing a neighbor relation.

To verify that all the elements were computed by Flink, an windowAll aggregation is done to count the number of neighbor relations found, emitting one single tuple per time step. This operation implies gathering all the data in one single node, thus impacting the performance. An actual analysis would instead apply other operations, not considered here for simplification purpose.

This algorithm is very costly in terms of memory and operations, because of the emission of the multiple copies of the atoms to all the related Z cells at the same time. During this phase, all the tuples are kept in memory.

Indeed, with lower level programming paradigms there is more room for optimizations, for instance the union of the Z-cells could have been done two by two instead of grouping all 27 cells at the same time. However, this Flink program stands out in terms of expressivity and ease of use. For instance, the mapping between Z-indices and sub-tasks was entirely abstracted by Flink, through an internal default hash function.

3.5.3 Flink Limitations Found

During the process of developing the analysis with Flink, we came across with 2 major limitations: Flink is not capable of switching from Stream processing to Batch processing in run-time and the IterativeStream API might not work correctly when a complex loop logic is used.

We have discovered these two restrictions while trying to implement the Leaflet Finder analytics, as done in [17]. This algorithm is normally used to find Phospholipid Bilayer membranes, which are the cells' exterior membrane. As its name says, its inner and outer surface are made of phospholipid molecules. The proposed algorithm was to identify neighbor phospholipid molecules based on a cut-off distance, compute the connected components algorithm and, finally, select the two biggest components for each time step.

Flink already had an implementation of the Connected Components using the Gelly API, for DataSets (batches). However, the equivalent API for streaming, the experimental Gelly Streaming API, was not as complete and versatile as its batch counterpart. Gelly Streaming was capable of computing connected components, but only a single-pass version of it and where only one graph was constructed along the progression of time. Nonetheless, Gelly Streaming did have an API for computing discrete separated graphs for each window, called 'GraphWindowStream'. However, this latter API, that would fit the requirements we had, did not have a connected components implementation.

We also discovered that, differently from Spark, a DataStream cannot be converted to a DataSet without storing it somewhere and reading it with another Flink instance running in batch mode. From that point, the only choice was to implement the algorithm from scratch using IterativeStream API.

This IterativeStream API is supposed to allow iterative processing over the streams. Using it is just a matter of calling the iterate method on a DataStream, performing some computations and separating the elements that shall be fed back to the loop and the ones that shall leave it. Without using this API, it is not possible to have iterative behavior in Flink, since the core representation of a Flink job is a directed acyclic graph (DAG) for the data-flow, which does not allow loops. This design choice was taken to avoid uncontrolled data duplication.

The classical connected components algorithm has as its input, a list of vertices a the list of edges connecting the neighbor vertices. For each vertex, a unique label is assigned. Then it sends its label to all its directly connected neighbors. Upon receiving these messages from its neighbors, the vertex changes his label, keeping the lowest label. If its label was changed, it would need to send its new label to its neighbors once again. By doing this iterations, the algorithm converges.

Due to the iterative nature of the connected components, it was absolute necessary to use IterativeStreams. The actual problem came from the use of complex operations inside the loop, such as join, which Flink cannot correctly deal with. The effect was that only one iteration was done.

After entering in contact with Flink's development team, through the official mailing list, we were informed that this issue was already identified and it is being addressed by Flink

Improvement Proposal 15, which does not have yet an estimation of when it will be accepted and be part of the official distribution of Flink.

Therefore, we had to abandon the Leaflet Finder algorithm and we chose to use only its first step, the identification of neighbors atoms.

— 4 —
Experiments

To be able to experiment with this framework, we have deployed it inside Grid 5000 [4] (G5K), a testbed for experiments on high performance computing, distributed systems, big-data and cloud. Grid 5000 infrastructure is composed of several clusters on several sites throughout France and Luxembourg. In all the experiments, all the nodes used were equal and part of one single cluster, in order to avoid introducing an other parameters that can alter the results. The specifications of the machines used for all the experiments are presented on Table 4.1.

Table 4.1 – Hardware description

Cluster Name	Paravance
CPU	2 x Intel Xeon E5-2630 v3
Cores/CPU	8
Threads/Core	2
Memory	128 GB
Storage	2 x 558 GB HDD
Network	2 x 10 Gbps

To enable experiment reproducibility, we relied on different software tools that enabled us to control the full deployment and execution of the experiments, from the operating system (OS) up to the last lever software used. Due to the strong coupling that these tools had with Grid 5000, a separate git repository was created, called Vebida-Deployment. It has two main purposes: The first one is building the system image that will be used on the experiment nodes, allowing us to precisely control what is installed on them. This is done through the use of Kameleon Image Builder[18], a tool that allows the creation on systems images based on scripts. The final software stack installed on system we used is shown in Table 4.2.

The second purpose is describing and executing the experiments. For that a Python script was made. This script uses Execo[11], a G5K Python API that allows running remote commands over SSH and lower level commands, such as requesting and deploying nodes. In more details, this Python script was used to set-up the configuration of the software stack, to configure the clusters for Flink, HBase and CoMD, to launch the experiments and to recover and log the interesting data from the experiments in a result file. This script receives as input a YALM file which describes which experiments will be executed, their parameters and the amount of nodes dedicated for each role in this environment.

Regarding the execution of the experiments, this Python script was executed on a separate node from the ones used on the set-up. This node runs the regular Grid5000 system image, not the ones customized by us. After being launched, this script is the one in charge of reserving the master and slave nodes for the experiments. As explained on section 3, the slave nodes were used to run the CoMD-MPI processes, the Flink's Task Managers (workers) and HBase's Region Servers (workers). The master node is used to launch the MPI processes on the other nodes, run Flink's Job Manager and HBase's Master process. Even though the master node was assigned with several tasks, those tasks were not compute-intensive. In addition, it has not showed any symptom of being overloaded during our experiments, staying with low memory and CPU usages.

Table 4.2 – Software stack in the nodes

Operating System	Ubuntu
Flink Version	1.3.3
Flink's Garbage Collector	G1GC
HBase Version	1.2.6
CoMD Version	1.1
ZeroMQ Version	4.1.4
Java Version	1.8
Hadoop Version	2.7.6

In the following sections, we detail the experiments we have made to assess our analytics framework. We first try to measure the maximal throughput we can achieve when Flink is only relaying the messages from CoMD to HBase. Next, we assess the impact of computing the Histogram analysis while still storing the CoMD messages. Then, we do it for the Neighbor identification analysis. Finally, we make a scalability test, by repeating the Histogram test in a system 4 times bigger.

4.1 Raw Data Handling and Storage Capability

The purpose of this experiment is to check how the system handles the input pressure generated by CoMD when Flink receives the data and forwards them to the storage nodes.

In regular *post-mortem* approaches, this is done by storing the data directly in a parallel file system, distributed throughout external nodes. Keeping in mind that this set-up has two additional components, Flink for processing the analysis and HBase to store the trajectories. Even though we largely benefit from the functionalities these two components offer to the framework, this additional software stack might degrade the raw performance. Therefore, the goal of this experiment is to study how capable these tools are to handle this pipeline.

One major aspect for designing this experiment was avoiding the buffering effect. As MD simulations might run for extensive periods and generate huge trajectories, we needed to make sure that the system would reach a steady state, where it would not need more and more buffers to be able to cope with the incoming data throughout the execution. In other words, we needed to find the sweet spot where the input flow was equivalent the output flow, and the throughput was maximal. In case the flows are not balanced, the system would eventually either drop data, crash or slow the simulation down. Neither of those side effects would be tolerable in this domain.

As we had time limitations to use Grid5000's resources, to measure the throughput while obeying to the aforementioned constraint without letting the experiments run for an exhaustive amount of hours, or even days, until the first symptoms appear, we started from the following assumption: the overall latency of each package received and stored would have to be contained within a bounded interval and not exhibit a growing trend. When the latencies increase, it means that the system's buffers are full and it needs more time to treat the data it already has, therefore generating back-pressure. Given this hypothesis, monitoring the latencies for a long enough period of time and generating a big enough trajectory would allow us to assume the system achieved its steady-state.

All the nodes used in the experiments belonged to the same cluster, on one single location, and their clocks were synchronized using Network Time Protocol (NTP). We assumed the timing drift in this case was negligible, allowing us to directly compare timestamps of different nodes. Given that, we measured the latencies through getting the timestamp in 3 different points of the system. The first one is when the CoMD processes start sending a new message; the second is as soon as Flink receives the messages and the third is the timestamp automatically assigned by HBase to the put requests it handles.

The HBase timestamp does not refer to the time the information was actually persisted in disk. Even though the tables are set to be stored in disk, HBase still uses in-memory caches to process its Put requests. HBase also uses 'memstore' blocks to sort the rows before flushing the data to the HFiles. Nonetheless, this timestamp was still chosen because, to the best of our knowledge, HBase lacks a system of reporting when a Put operation was persisted in disk. However, having this memory buffers in HBase does not mean it will not generate back-pressure. When any memstore reaches a certain threshold, HBase blocks incoming requests while it flushes the data. Thus, making Flink slow down and increasing the latency of Flink's packets. Therefore, by generating a big enough trajectory for a long enough, HBase's in-memory buffers will either be unstable, eventually generating back-pressure, or be stable, not blocking incoming put requests.

In the attempt to capture the HBase and Flink buffer usage, we also monitor the heap usage of the Java Virtual Machines that run the Flink's Task Managers and HBase's Region Servers.

By classifying the behavior of the latencies as stable or not, measuring the highest throughput was just a matter of running the experiments for different throughput values and keeping the highest value that was stable (that the latencies did not grow). The measure of the throughput itself was obtained by dividing the total amount of data the CoMD MPI processes sent to Flink by the time it needed to compute the simulation. To be more precise, it refers to the time interval between the sending CoMD's very first message and sending the very last.

Since the throughput is not a direct parameter of CoMD, varying it was done through varying the time step interval with which the simulation outputs its frames (referred on this work as output interval). The lower it is, the higher the throughput, since it will emit data more frequently.

For following series of experiments, we had: 8 simulation nodes (8x16 physical cores); 1 Flink node (16 physical cores); 1 HBase node (16 physical cores). The simulation was big, it had 32 million atoms, which represent half of the HIV virus molecular structure [7]. It was parallelized in 125 MPI processes, due to the 3D-data-partitioning that CoMD requires. Each CoMD MPI process emited messages of 6.83 MB, totalizing 853.75 MB per frame, which is around 35 times the frame size used on [7] to benchmark their framework. Since we had limited resources in Grid 5000 and the interconnection between the nodes was done with a fast 10 Gbps network, we assigned 16 MPI processes per CoMD node, i.e. one per core. The maximal throughput that CoMD could generated (stable or not) with this configuration (when the output interval is set to 1) was around 774 MB/s.

Flink was configured to use 4GB of memory for the Job Manager's heap (in the master node); 6GB for the Task Manager's heap (Flink worker node, referred as Flink node) and use G1 Garbage collector.

HBase was set to not replicate data and to not compress data. HBase also allows the users to set a wide range of parameters, however they are very coupled to each other, meaning that if not properly set, HBase can suffer from huge performance degradations. For this reason, we decided to use the default values for the machines we were using. The Region Server's (HBase slave nodes, referred as HBase nodes) heap size was 30 GB. Other configurations were: the memstore block multiplier set to 2. The lower and higher limit for the total size of the memstores were the defaults,35% and 40%. These last 2 values mean that once 10GB is used by memstores, it starts to get flushed, and when 12 GB is used, it blocks incoming update requests.

For running the experiments, since we vary the throughput by changing the output interval, we decided to fix the overall trajectory size and not the total number of computed time steps, to avoid not capturing important behaviors related to total amount of data stored, that, for instance, might occur inside HBase. We fixed the trajectory size to 62.5 GB, which is 75 outputted frames. This value is twice bigger than HBase's region server's heap (30GB). Therefore, we are certain that HBase cannot cache it entirely in memory. With these parameters, CoMD runs for about 15 to 20 minutes and all the results are already able to be read from the HBase around 2 minutes later.

Figure 4.1 shows the throughput obtained for the tested output intervals and how the points were classified by its stability. The graph shows that we could achieve up to 71 MB/s without presenting back-pressure issues during the run. Due to the time limitation we had on the usage of the cluster and for this work's deadline, we could not run enough experiments generating a bigger trajectory. Each point was executed without repetition, except for the points in the edge between stable and unstable, which were executed twice. In case any of the repetitions proved to be unstable, the point was considered unstable.

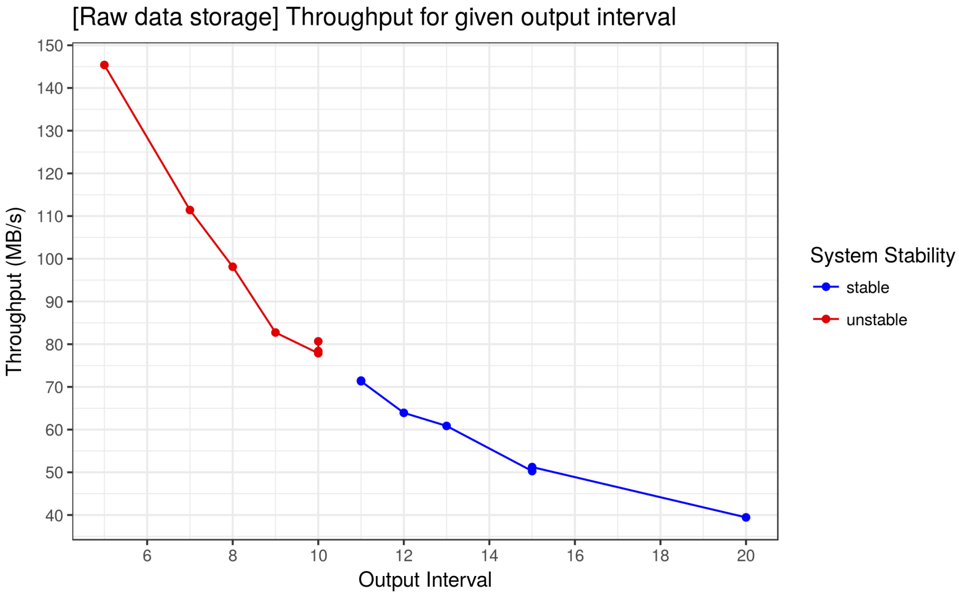
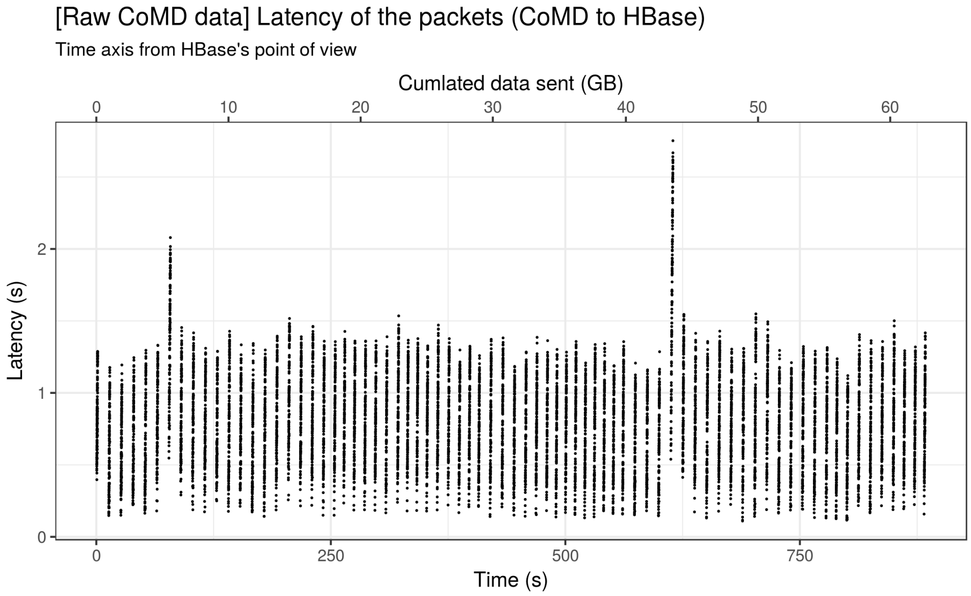


Figure 4.1 – Throughput achieved with different output intervals and their stability

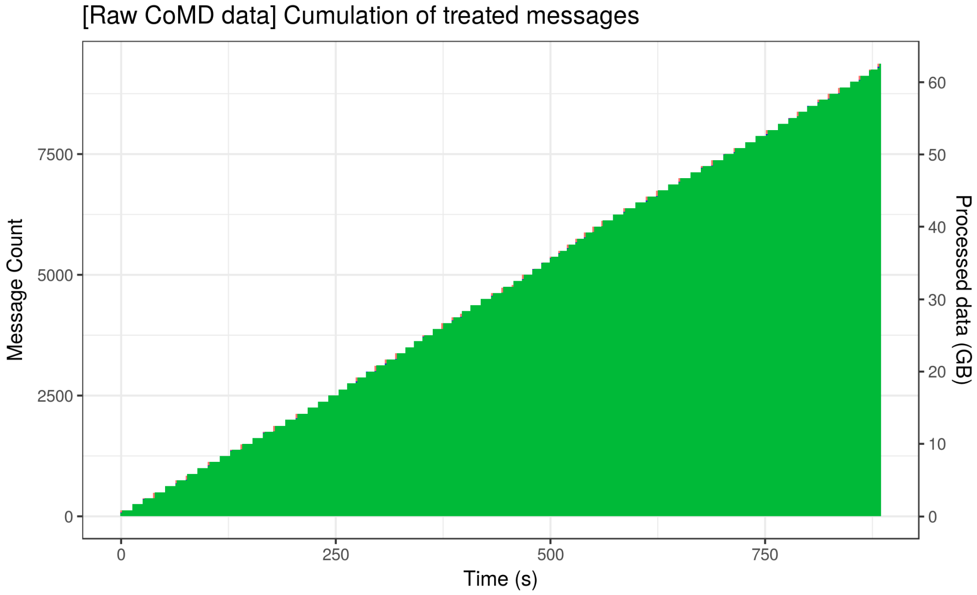
4.1.1 Characteristics of a Stable Point

The measurements made for the stable point with output interval equals to 11 are shown in Figure 4.2. Figure (a) shows the latencies computed and that they do not show a growth trend. Figure (b) shows the number of messages processed throughout time at each point of the system. In this figure, 3 cumulation histograms are overlaid. The orange one counts the number of messages created by CoMD; the blue one, the number of messages received by Flink and the green one, the number of messages handled by HBase. By analyzing it, it is clear that the system is able to cope with the production of the data in the same rate data are produced. That is why it is almost not possible to see the orange (creation of the messages in CoMD) and the blue (Flink ingestion) histograms, which are placed behind the green curve.

In Figure 4.3, by analyzing HBase's Region Server heap usage, it is possible to confirm the predicted behavior for HBase: it works by caching the data in memory before flushing them to disk. Figure 4.3 also shows that throughout time the troughs (when HBase flushes its data) get higher. We did not identify the actual reason for that memory increase. It may be due to some internal data structure.



(a)



(b)

Figure 4.2 – Plots for the stable point with store interval 11. (a) Latency (CoMD to HBase) of packets. The timestamp considered for the x-axis is the HBase's. The secondary axis, on top, shows the amount of data that CoMD has already sent. (b) Cumulation of events through time. In pink, the number of sent messages; in blue, the number of messages received by flink and in green, the number of stored messages.

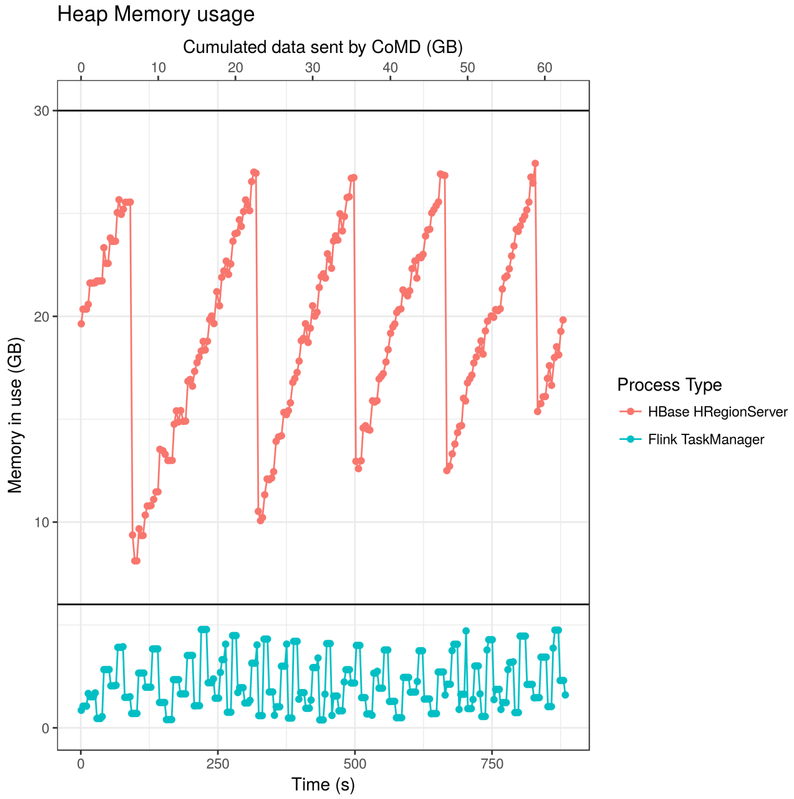


Figure 4.3 – Heap usage of Flink's and HBase's worker nodes for the stable point of output interval 11. Flink's heap size is 6 GB, while HBase's is 30 GB

4.1.2 Characteristics of an Unstable Point

When investigating the latencies of an unstable point, the graphs depict very distinct behaviors from the ones observed on the stable case. For instance, let us analyze the point of lowest throughput for this configuration (when the store interval is 10).

In Figure 4.4, the latencies are clearly growing. In addition, there is also an extra feature that starts happening around 800 seconds, where the points start to be shifted up. This comes from the fact that the system in this case needs more time after CoMD finishes to store all the data it has sent, meaning that there were lots of messages in the system, waiting in buffers. Figure 4.5 shows the intermediate components of these latencies, CoMD to Flink (called ZeroMQ latency) and Flink to HBase (called Flink latency). It is possible to see that the ZeroMQ latency has the major influence over the CoMD-to-HBase latency.

Figure 4.6, shows the latencies from the perspective of the number of messages that were treated throughout the time. It is possible to see that the growth of the latencies in Figure 4.4 correlates to the distance between the the orange and green lines, meaning that those packets are inside the system, waiting somewhere in between.

From these 3 graphs analyzed for the unstable point, it is possible to conclude that the messages wait in the buffers prior to Flink. Indeed, the ZeroMQ sockets were configured to have huge buffers, able to hold up to 3000 messages on both sides of the communication. Having these big buffers also means that ZeroMQ may physically transfer the data to Flink regardless of the rate Flink is consuming the messages, until these ZeroMQ buffers on the

Flink side are full. The choice of having big buffers at these point was made to reduce the impact that the data transfer could have in the experiments.

In addition, it is important to mention that the nodes are interconnected with 10 Gbps network connection and the size of the messages sent with ZeroMQ is 6 MB, which is already a big enough size to mitigate the transfer latency, meaning that the total throughput used (80 MB/s) is far below the theoretical limit for these connections; thus, not being the main candidate for the bottleneck of the system.

Having a more thorough look at figure 4.6, it is possible to notice that the blue histogram is now visible as a thin layer over the green curve, meaning that there are moments where Flink has processed more data than HBase. If we analyze HBase's heap usage (Figure 4.7) together with it, we can see a correlation between a reduction of the slope in the green curve and the moments when HBase is flushing its buffers [1] . These facts lead us to hypothesize that HBase is causing back-pressure on Flink, which, in turn, reacts very quickly to it, stopping the pipeline before accumulating lots of data. These data, then, get accumulated in ZeroMQ's buffers.

To check if our hypothesis was correct, we needed to reduce the pressure in HBase and see if in this case we could achieve higher throughputs. For that we modified the HBase sink to discard all the atoms IDs and positions and replacing them by two 32-bit value. By doing so, we ended up reducing the amount of data HBase would receive, but keeping the same workload on ZeroMQ and Flink. We have run the experiment for the point of output interval 5, which previously was clearly unstable, and the results showed that it became a stable point, with a throughput of 147 MB/s. Thus, this experiment corroborated our hypothesis that HBase was the bottleneck in the framework.

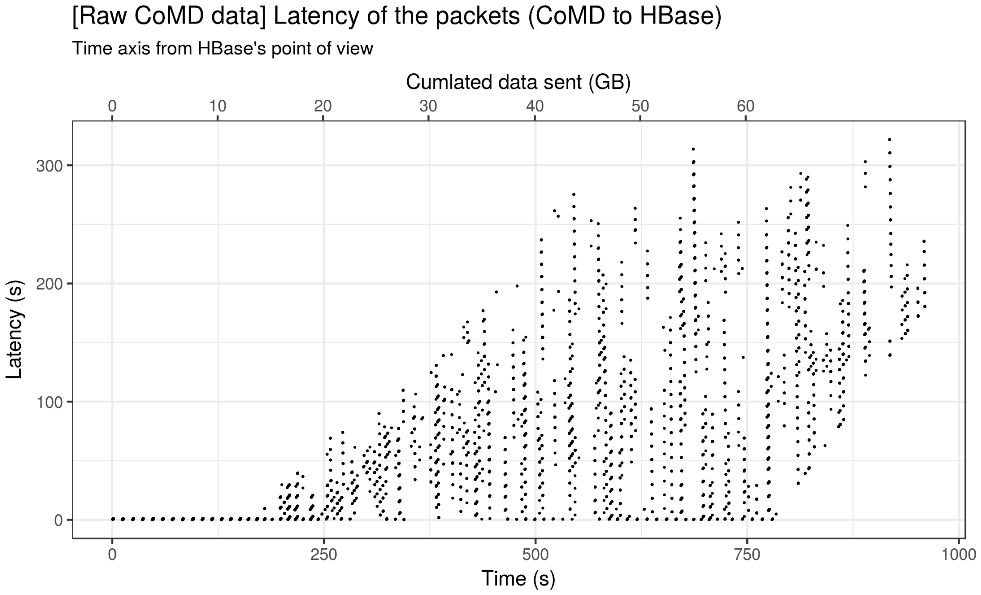


Figure 4.4 – Latency (CoMD to HBase) of packets. The timestamp considered for the x-axis is the HBase's. The secondary axis, on top, shows the amount of data that CoMD has already sent. The store interval is 10

[1] This correlation gets clearer when analyzing the graph without the query perturbation, in Appendix A.2.1. Prior to the appendix, this is further explained in the end of this section.

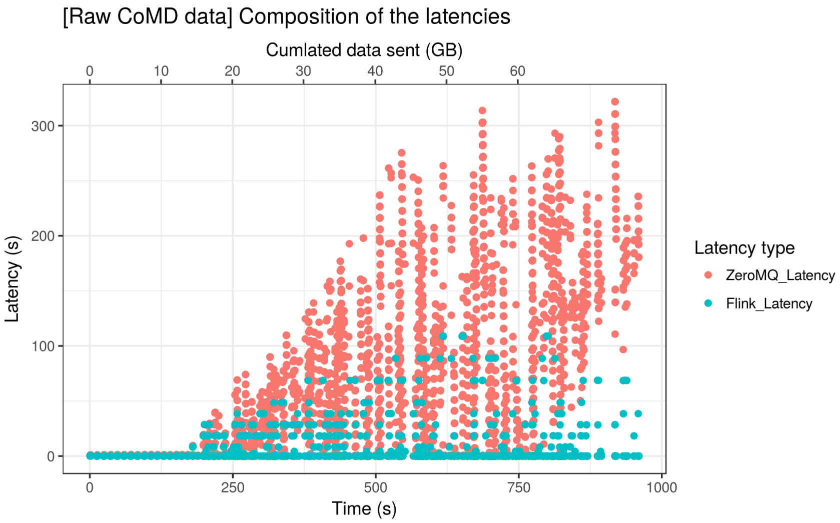


Figure 4.5 – Composition of the latency of the packets. ZeroMQ_Latency stands for the time between the creation of the message (in CoMD) and its ingestion by Flink. Flink_Latency refers to interval between ingesting the packet and storing it in HBase

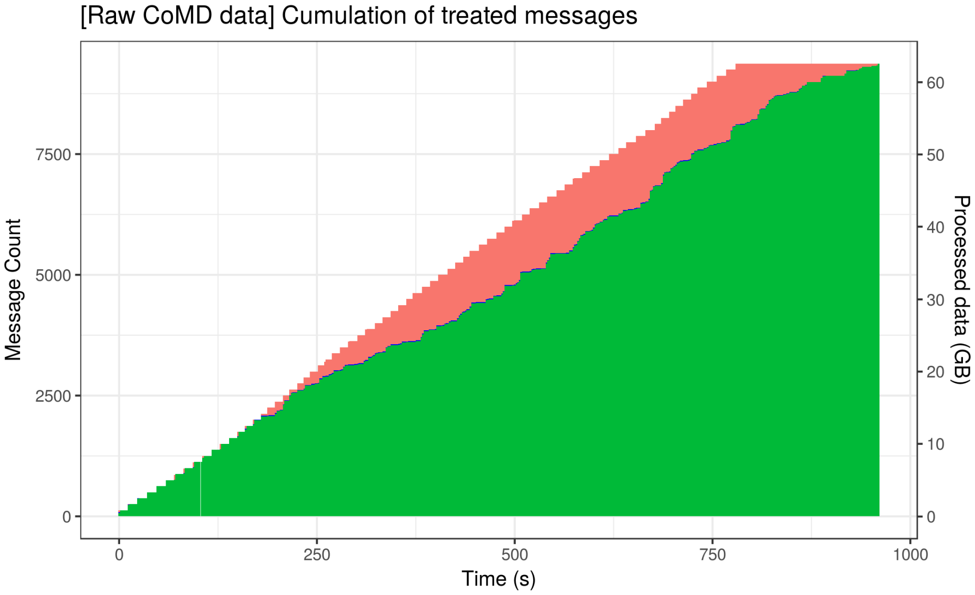


Figure 4.6 – Cumulation of events through time. In pink, the number of sent messages; in blue, the number of messages received by Flink and in green, the number of stored messages.The store interval is 10

Continuing our analysis of the unstable point we were previously detailing (output interval 10), Figure 4.7 shows an unexpected behavior. Here, after CoMD sends all its data, more pressure is put on HBase heap, where it cannot reduce its memory usage neither by flushing more frequently. Surprisingly, this pattern does not seem to correlate with the back-pressure, which is detected in the latencies much earlier. We concluded that this perturbation on HBase's heap comes from the way our experiment script was implemented. In the script, as soon as CoMD finished, it starts querying HBase for all the stored row-key values, in order to know when everything is processed by HBase. Therefore, during this phase HBase needs to keep processing the put requests while it handles the queries, thus applying more pressure on it.

To confirm this hypothesis, we have re-run these 2 experiments, the stable and the unstable, but making the script sleep for 7 minutes after CoMD finished, to guarantee that everything was already stored when it starts making the queries. The results confirmed our supposition, but also showed that this perturbation on the heap had no impact on the other measurements (the latencies). Thus, we did not have to re-run all the other experiments showed in this report. The graphs from those two points re-tested are on Appendix A.2.

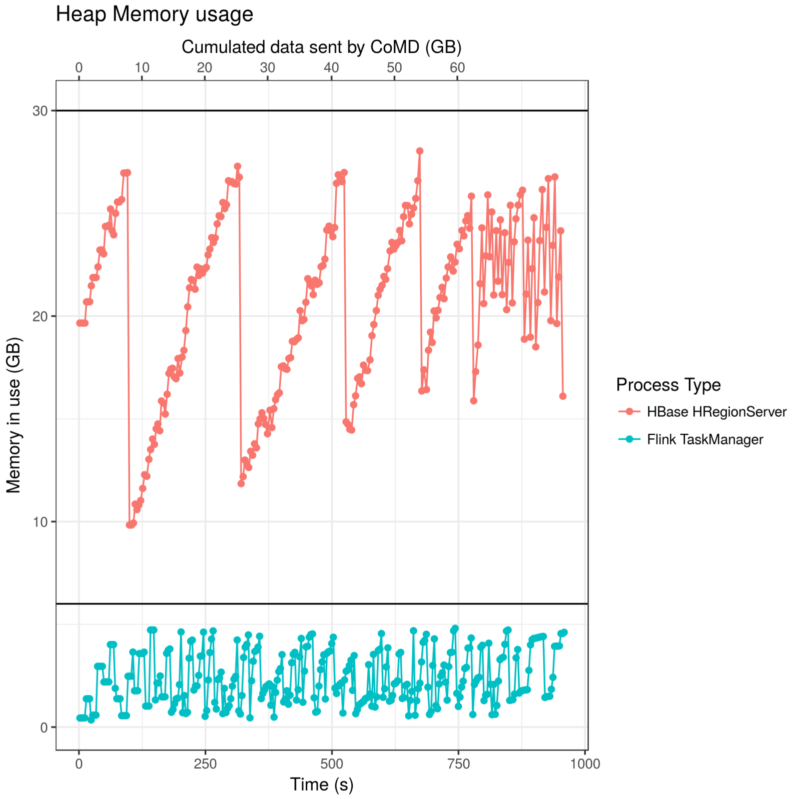


Figure 4.7 – Heap usage of Flink's and HBase's worker nodes for the unstable point of output interval 11. Flink's heap size is 6 GB, while HBase's is 30 GB

4.1.3 Histogram Analytics

For this series of experiments, we wanted to discover the impact of running the Histogram analytics together with storing the CoMD raw data in HBase.

Regarding the analytics code, we decided to store its results in HBase as well, to keep the experiment closer to a realistic scenario, even though storing these results increases the pressure on the storage system. Another separate table was used to store these data, called "flink_analytics". The row-key for this entries was of the form 'ZIndex_time step', thus spreading the workload throughout the regions.

We have fixed the total number of bins to 512, by using only 9 bits for the Z-Index (3 bits per coordinate). This choice was a compromise between exploring more parallelism and not impacting so much the storage.

All the other parameters for the experiments continued to be the same as used on section 4.1, allowing us to directly compare the results. Figure 4.8, depicts the throughputs achieved for the tested output intervals. In this case, we were able to achieve 78 MB/s, with output interval of 10, which is slightly more than what we have seen in the previous section, with just the raw trajectories storing and no analytics at all.

This unexpected better performance might come from the intrinsic inaccuracy of these experiments. As we were limited on time, we had to make these experiments shorter, with smaller trajectories which limits our ability to know how the system would behave in these cases if they continued running for hours. What might happen is that a point that seems to be stable during the first 20 minutes, becomes unstable at some point. Therefore, a false positive result would be obtained.

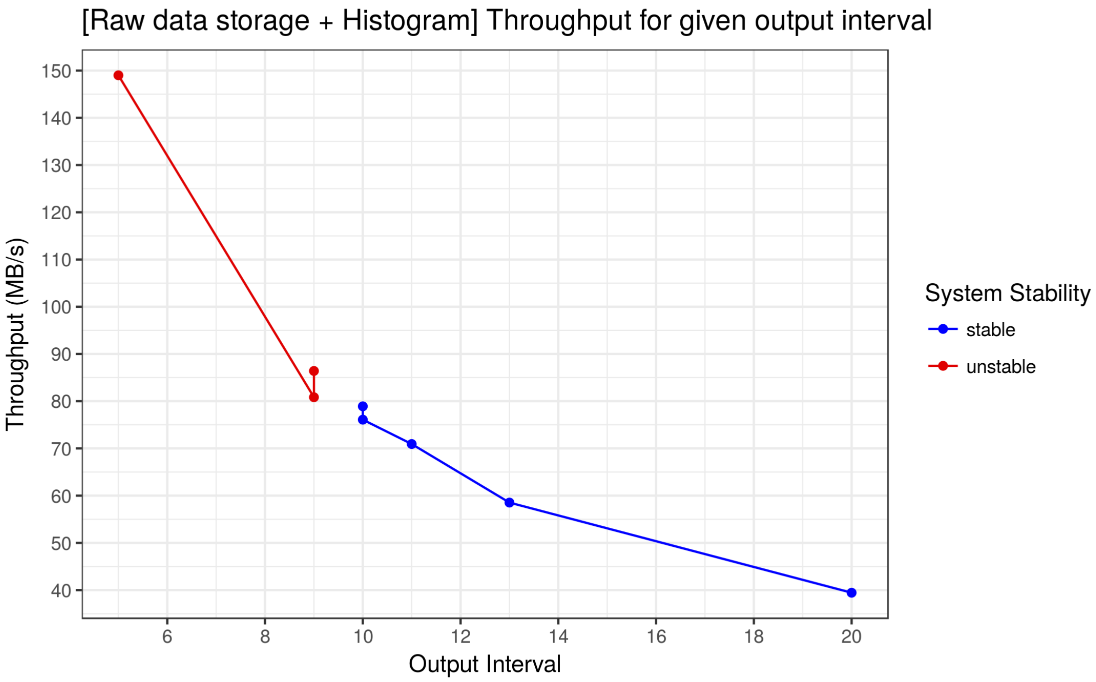


Figure 4.8 – Throughput by output interval and their stability

Let us analyze the stable point of store interval equals to 10. In Figure 4.9, we have the latency of the packets containing raw CoMD data to be stored in HBase (the same measurement from the previous experiment). It is possible to see the stable behavior and also that, in comparison to the previous experiments, the latencies have increased, varying now in the interval (0, 3s] instead of (0, 1.5 s], for the vast majority of messages.

For this experiment, we also measured the latencies of the analytics results. As in this case we did not have a 1-to-1 correspondence from the message sent by CoMD and the results, we measured the latency per time step. More precisely, the considered latency for a given simulation time step is the time between the creation of the last CoMD message of that time step and the last result entry on HBase for the same time step.

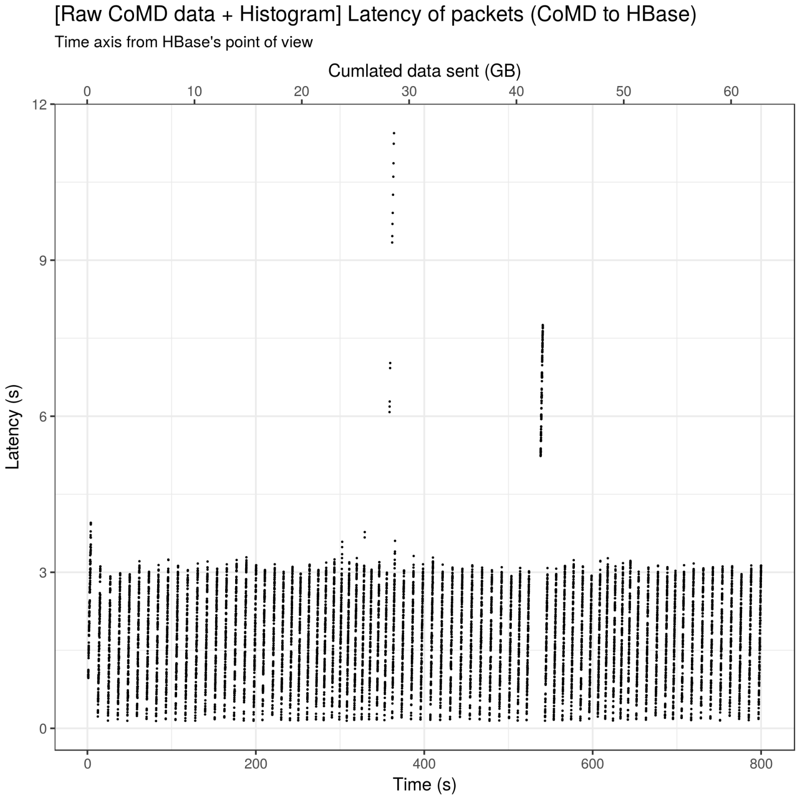


Figure 4.9 – Latency of messages

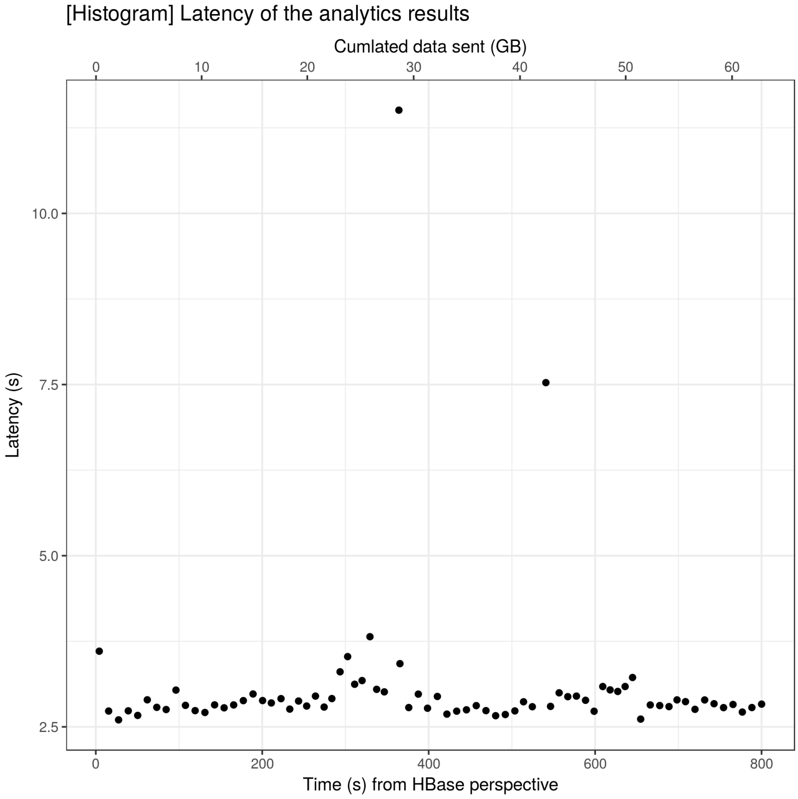


Figure 4.10 – Latency of the analytics results

In Figure 4.10, it is possible to see that they not only correlate to the latency of the messages stored in HBase (Figure 4.9), but also have similar latency values (around 2.5s and 3s). This behavior was expected, since storing the CoMD messages and computing the analytics happen in parallel, at a similar pace.

4.1.4 Neighbors Analytics

Assessing the performance of the Neighbors analytics is not as straight forward as is the Histogram. Due to the huge difference in their complexity, O(N^2) and O(N), respectively, computing the Neighbors analytics over same input data becomes impractical memory-wise, for this set-up. Therefore, some parameters needed to be changed, not allowing us to compare the results directly anymore.

It is worth mentioning that, as for the other experiments, Flink's Task manager was set to use only 6GB of memory and even with our attempt to reduce the total amount of operations for this algorithm, by splitting the domain in a grid of cells, an enormous amount of tuples is still generated because all the computation inside the neighbor cells still O(N^2).

Since, computing this algorithm for 32M atoms would require more memory than what we were using in Flink, the number of the atoms in the simulation was reduced to 256 K, which is still bigger than the value they have used on [17] to compute the Leaflet Finder algorithm. By changing the simulation size, the whole frame size was reduced to 6.83 MB and time required to compute one time step also reduced. We fixed the trajectory size to 130 outputted frames (888.16 MB), which would still end up taking from 15 to 20 minutes of execution.

Regarding the results, only one result tuple is generated per time step, a tuple that contains the time step and the number of neighbor relationships found. This tuple is stored in the second HBase table, called 'flink_analytics', similar to the one used with the histogram. The difference, in this case, is that salting the row-key is not necessary since the data pressure in this table is very low (one tuple per outputted time step).

Figure 4.11 depicts the throughputs obtained. We were able to reach only 1.10 MB/s with this set of parameters and analytics.

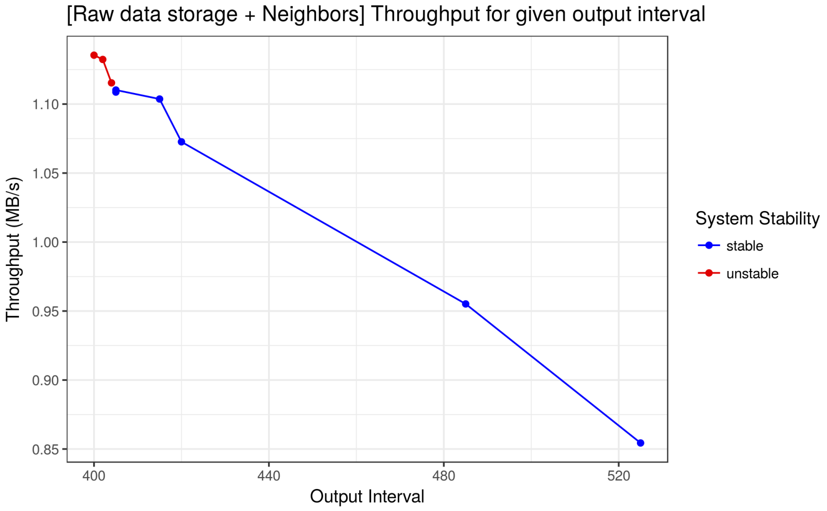


Figure 4.11 – Throughput obtained for the Neighbors Analytics

Figure 4.12 shows the latencies and their composition, for the stable point of output interval 405. Differently from the other stable point found for the other experiments, here, there are much more outliers for the Flink latency, but their overall trend continues to be the same.

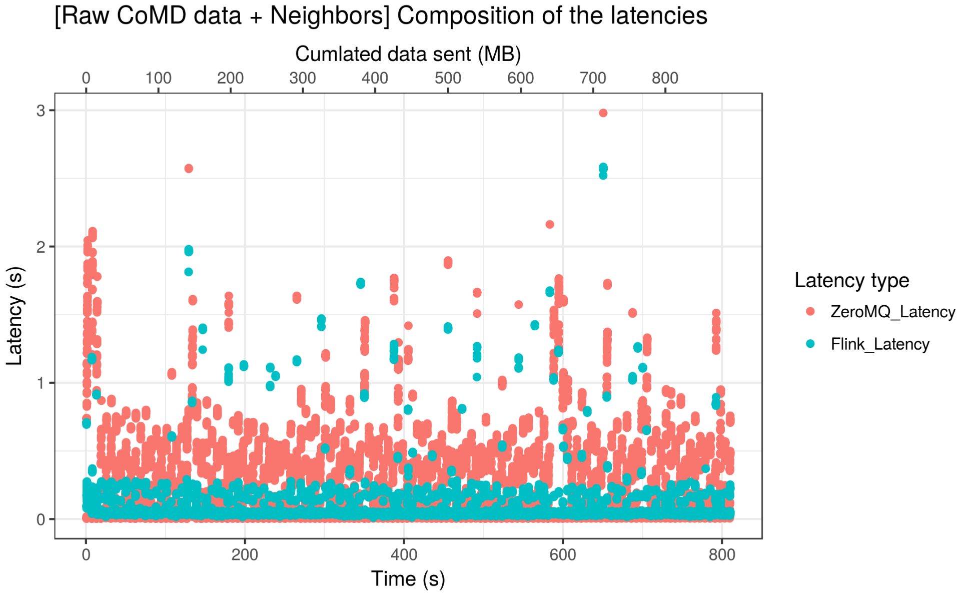


Figure 4.12 – Throughput obtained for the Neighbors Analytics

4.1.5 Scalability

All the previous experiments had 125 CoMD MPI processes (in 8 nodes), 16 cores for Flink (1 node) and 16 cores for HBase (1 node). With this experiment we wanted to assess the scalability of the analytics framework we proposed, by doing a weak scalability experiment, where the workload also scales with the system. We tested on a configuration 4 times bigger: 512 CoMD processes (32 nodes) and 64 cores for Flink and HBase (4 nodes each). The number of atoms in the simulation was multiplied by 4, generating a trajectory of 256 GB. All the other parameters stayed the same as the ones used on the small scale Histogram experiment, which allows us to compare the results directly.

Due to the short time limit we had at our disposal for performing this experiment, we had to crop the variations of the experiments executed. Therefore, we have only experimented with the Histogram analytics, and even so we could only run it for some few points, as shown in Figure 4.13. Unfortunately, we could not find any stable point. However, we can infer that no throughput higher than 100 MB/s will be stable.

For comparison purposes, the experiment in small scale showed that the system was able to handle up to 78 MB/s. If the scalability of this system was linear, we would expect reaching a throughput of 312 MB/s for this configuration. By using the smallest unstable throughput obtained in the scalability test, 100 MB/s, as a rough higher-bound for the actual stable throughput, we can claim that the system has a scalability efficiency lower than 32%.

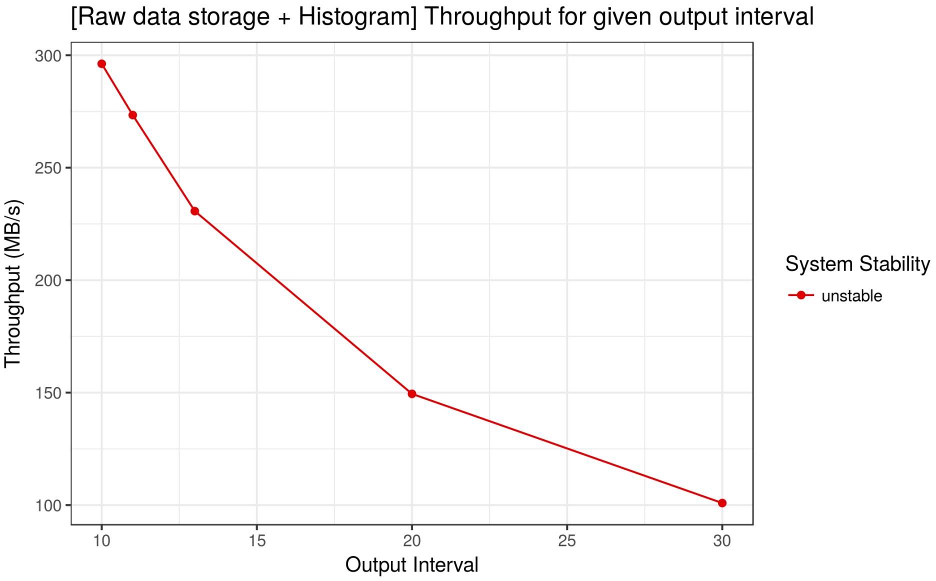


Figure 4.13 – Throughput obtained for the Histogram experiment in scale 4x

— 5 —
Conclusion

We have proposed a new framework for performing parallel in-transit molecular dynamics analytics based on Apache Flink, a stream processing engine made mainly for processing analytics over business data. The main advantage of this framework, compared to the state-of-the-art, is the easiness of its programming model, which does not require the scientist to think about data partitioning and communication for their parallel code, while offering more flexibility, than Map-Reduce, and expressibility for the code.

We have built a system that implements this framework, by connecting Flink to HBase (storage) and to CoMD (the MD mini-app). Next, we have implemented two common analytics for the field: the position histogram and the identification of neighbor atoms. These analytics have very distinct complexities and communication patterns, but thanks to Flink's programming model, both ended up being quite similar since all the complex communication patterns were hidden from the user.

We have experiment on the system built to find the highest stable throughput it can achieve when relaying the CoMD data to HBase and when doing so while computing each of the analytics implemented. We have also experimented the Histogram analytics in a scale 4 bigger, in order to see how the system would scale.

Results showed that for a set-up with 8 CoMD nodes (125 cores):1 Flink node (16 cores):1 HBase node (16 cores) and a simulation of 32M atoms, we have achieved up to 71 MB/s when computing no analytics and 78 MB/s computing the Histogram. When putting these results into perspective, by comparing them to the normalized throughput values used in practice, we see that real simulations are generating a data pressure of 35 MB/s (125 cores) [7]. Therefore, this corroborates the suitability of the framework to the application.

However, it is important mentioning that these results we have obtained refer to 20 minute-long simulations, that generates a 62GB trajectory (and 888.16 MB trajectory for the neighbor analytics). As we were short in time for performing the experiments with this set of parameters, we needed to reduce the time taken for each run. Nonetheless, prior to those experiments, we had experimented with longer periods and other combination of parameters. On those attempts, we verified that the same behavior of being stable and at some point becoming unstable could happen anytime. Therefore, if we run the same simulations presented here for longer periods, it could happen that the actual maximal stable throughput gets lower.

The evidence that we acquired from the measurements of the latencies, the heap usages and the additional experiments we have made during this research let us conclude that the bottleneck of this system is HBase. The main indicators that supports this conclusion come from the following facts:

In stable points, HBase is able to cope with all its input pressure, making the cumulation of events graph entirely green; in unstable points, the same does not happen, letting the blue and orange histogram visible; the correlation between the reductions in the slope of the green curve (HBase) and moments HBase was flushing its data; the quick reaction Flink has to back pressure, making the pipeline stop as soon as it detects it. Last but not least, the experiments showed we could get higher throughputs if we reduced the pressure in HBase.

In respect to neighbor identification analysis, we have achieved up-to 1.10 MB/s. However, this result cannot be directly compared to the previous values, since this experiment was made with very different parameters, due to the algorithm's complexity. Differently from the previous experiments, the identified bottleneck here was the analytics and its execution. Therefore, it is necessary to improve the algorithm in order to expect a better performance.

Regarding the scalability in our experiments, we could not achieve a decent scalability efficiency. However, this could still be an effect of HBase. More experiments in this regards are necessary to properly confirm it.

Since HBase is the component holding the system down in most of the cases, other distributed data storage systems could be investigated as replacement for it, as future works. From trying other databases, like Apache Cassandra, or using directly a distributed file system, like Hadoop, or even going for in-memory storage over the new Non-volatile RAM (NVRAM).

— A —
Appendix

A.1 Histogram Analytics Code

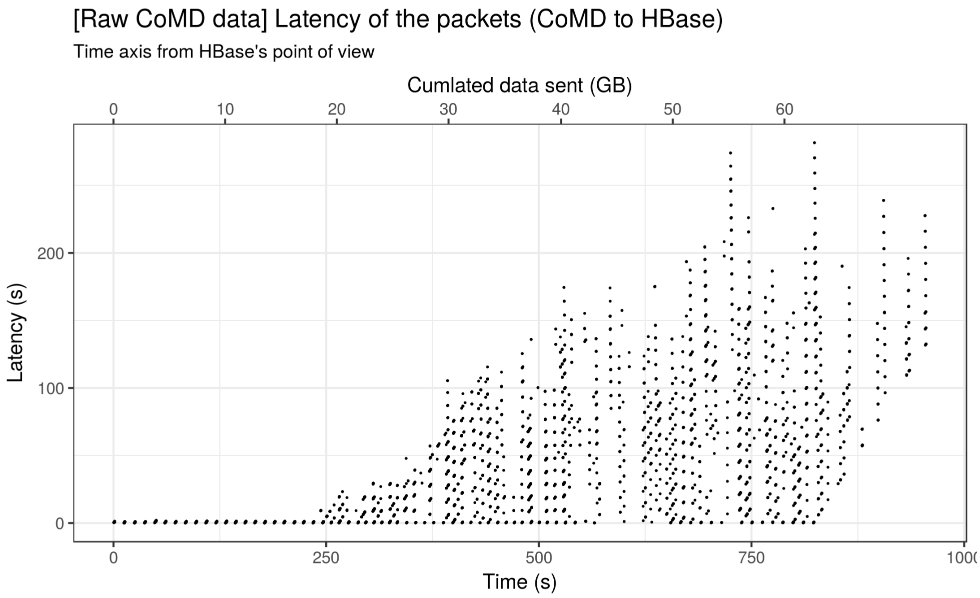
Listing A.1 – Code of the Histogram Analytics with output on the standard output

```
public class Histogram {

 public static void main(String[] args) throws Exception {

  // set up the execution environment
  final StreamExecutionEnvironment env = StreamExecutionEnvironment
   .getExecutionEnvironment();

  //Set the Stream Environemnt to work with Event Time
  env.setStreamTimeCharacteristic(TimeCharacteristic.EventTime);

  //Get the Message stream from ZeroMQ
  DataStream<Tuple6<Integer, Integer, Long ,Long, byte[] ,byte[]>> text =
     env
     .addSource(new ZeroMQBinaryStreamFunction(hwm, port, nports, -1,
        configFilePath, nComdClients))
     .name("Queue")
     ;

  /* Compute Histogram */

  //Messages are split into Atoms (Tuple3<Timestep, ZIndex, Count>)
  DataStream< Tuple3<Integer,Long,Integer> > atoms = text
               .flatMap(new MessageParser())
                                           .name("Split atoms");
        int timestep = 0;
        int zIndex = 1;
  int count = 2;
  long windowSize = 1;
        DataStream<Tuple3<Integer, Long, Integer>> analyticsResults = atoms
                 .keyBy(zIndex)
        .window(TumblingEventTimeWindows.of(Time.milliseconds(windowSize)))
                    .sum(count).name("Count atoms")
                    .project(timestep,zIndex,count)
                    ;

  //Print the results in the standard IO
  analyticsResults.print();

  // execute program
  env.execute("CoMD with Analysis and DB Example");

 }

 private static class MessageParser implements FlatMapFunction< Tuple6<Integer, Integer, Long ,Long, byte[] ,byte[]> ,
                                                                             Tuple3<Integer,Long,Integer> > {

   @Override
   public void flatMap( Tuple6<Integer, Integer, Long ,Long, byte[] ,byte[]>  in , Collector< Tuple3<Integer,Long,Integer> > out) {

    ByteBuffer buffer =  ByteBuffer.wrap(in.f5).order(ByteOrder.nativeOrder());
    buffer.position(0);

    while (buffer.remaining()>0)
      {
       Double[] position  = {new Double(buffer.getDouble()),new Double(buffer.getDouble()),new Double(buffer.getDouble())};
       Long zIndex = ZIndex.compute(position);
     out.collect(new Tuple3<Integer,Long,Integer>(in.f1, //Timestep
                          zIndex,
                          new Integer(1)
                                                     ));
   }
  }
 }

 public static class ZIndex{

  public static Long compute(Double[] position)
  {
   // final int numberOfBits = 21;
   final int numberOfBits = 4;
   int[] truncated = new int[3];
   //Mask is the least significant numberOfBits bits set to 1
   int mask = ( 1 << numberOfBits ) - 1;

   for(int i=0; i<3;i++)
   {
    truncated[i] = (int)(Math.round((position[i].doubleValue()))) & mask;
   }

   return mortonEncodeMagicBits(truncated[0],truncated[1],truncated[2]);

  }
```

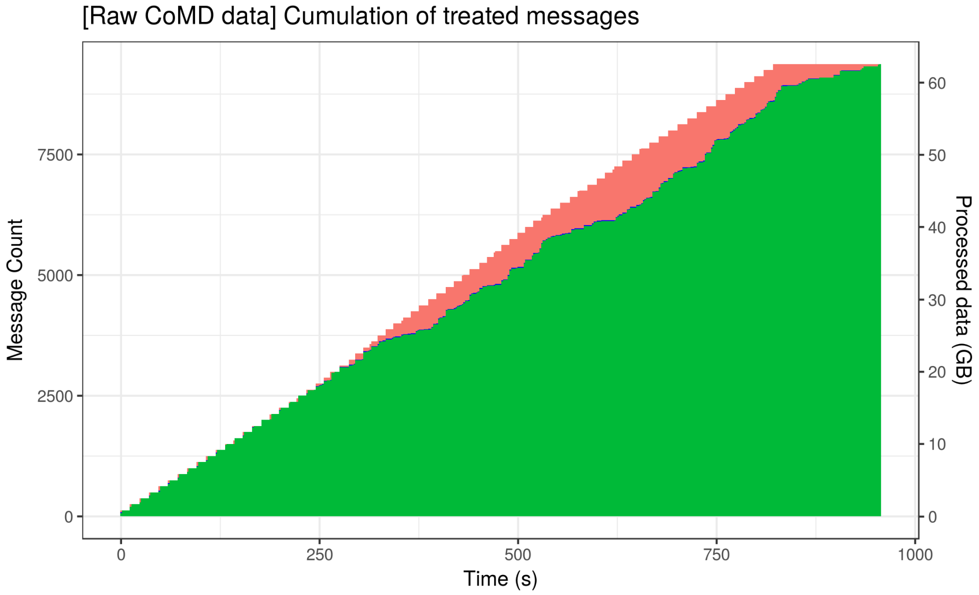
```java

    private static long splitBy3(int a){
        long x = a & 0x1fffff; // we only look at the first 21 bits
        x = (x | x << 32) & 0x1f00000000ffffL;  // shift left 32 bits, OR with self, and 00011111000000000000000000000000000000001111111111111111
        x = (x | x << 16) & 0x1f0000ff0000ffL;  // shift left 32 bits, OR with self, and 00011111000000000000000011111111000000000000000011111111
        x = (x | x << 8) & 0x100f00f00f00f00fL; // shift left 32 bits, OR with self, and 0001000000001111000000001111000000001111000000001111000000000000
        x = (x | x << 4) & 0x10c30c30c30c30c3L; // shift left 32 bits, OR with self, and 0001000011000011000011000011000011000011000011000011000100000000
        x = (x | x << 2) & 0x1249249249249249L;
        return x;
    }

    private static Long mortonEncodeMagicBits(int x, int y, int z){
        long answer = 0;
        answer |= splitBy3(x) | splitBy3(y) << 1 | splitBy3(z) << 2;
        return new Long(answer);
    }
  }
}
```

A.2 Raw CoMD data experiment without the queries perturbation

A.2.1 Unstable point (output interval of 10)



(a)



(b)

Figure A.1 – (a) Latency (CoMD to HBase) of packets. (b) Cumulation of events through time. In pink, the number of sent messages; in blue, the number of messages received by flink and in green, the number of stored messages.

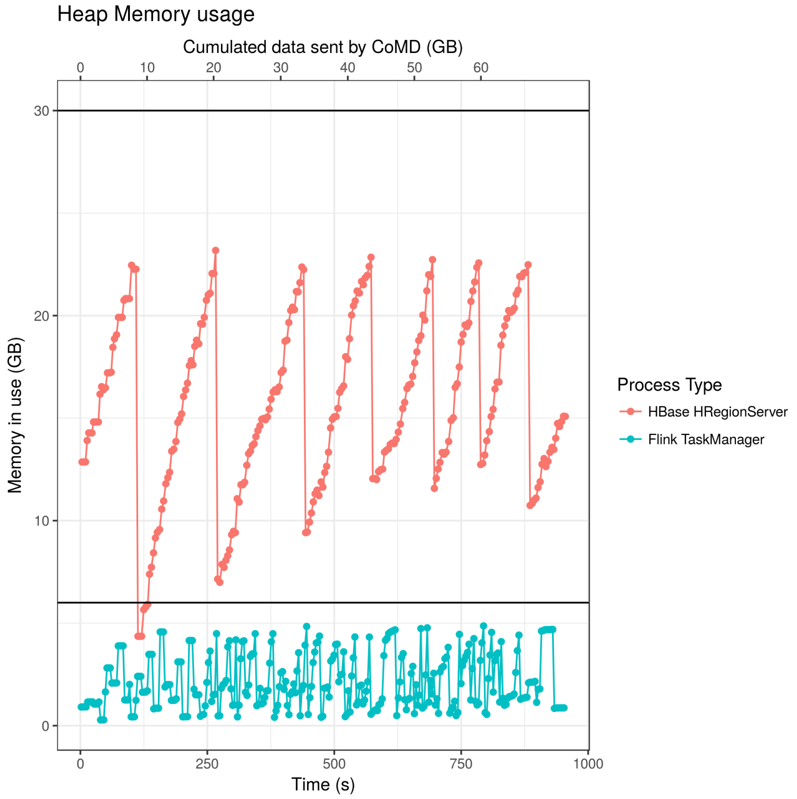
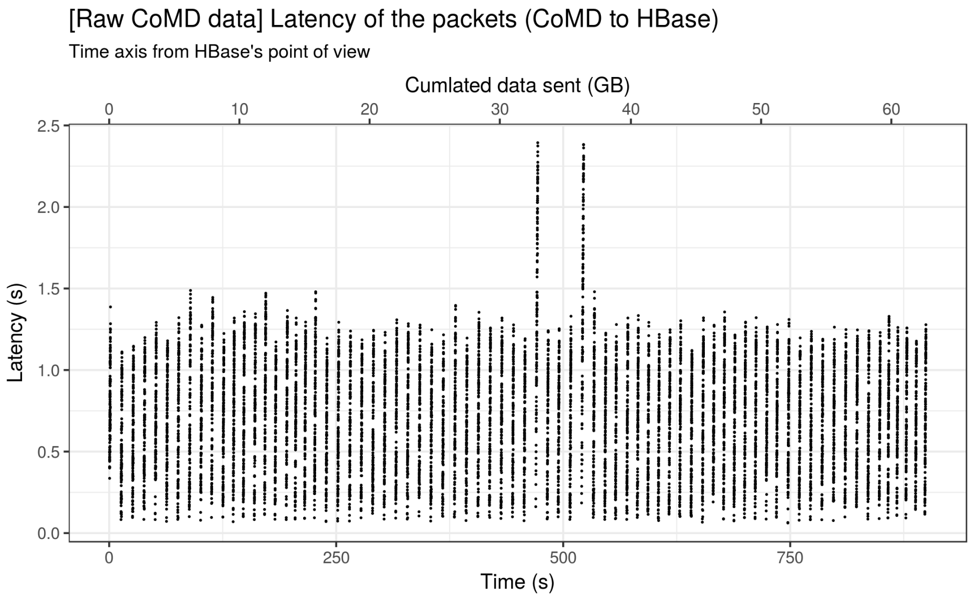
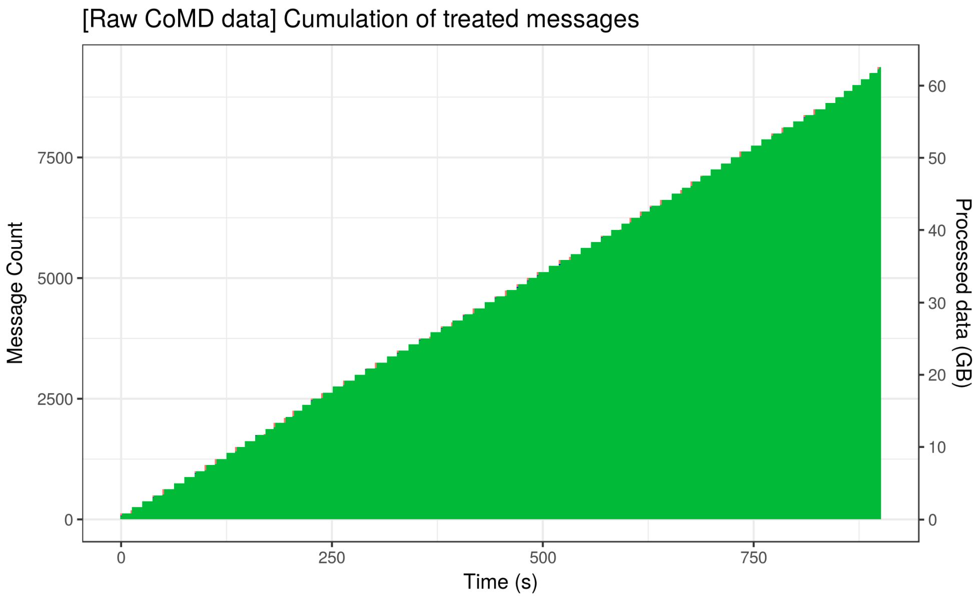


Figure A.2 – Heap usage of Flink’s and HBase’s worker nodes for the unstable point of output interval 10. Flink’s heap size is 6 GB, while HBase’s is 30 GB

A.2.2 Stable point (output interval of 11)



(a)



(b)

Figure A.3 – (a) Latency (CoMD to HBase) of packets. (b) Cumulation of events through time. In orange, the number of sent messages; in blue, the number of messages received by flink and in green, the number of stored messages.

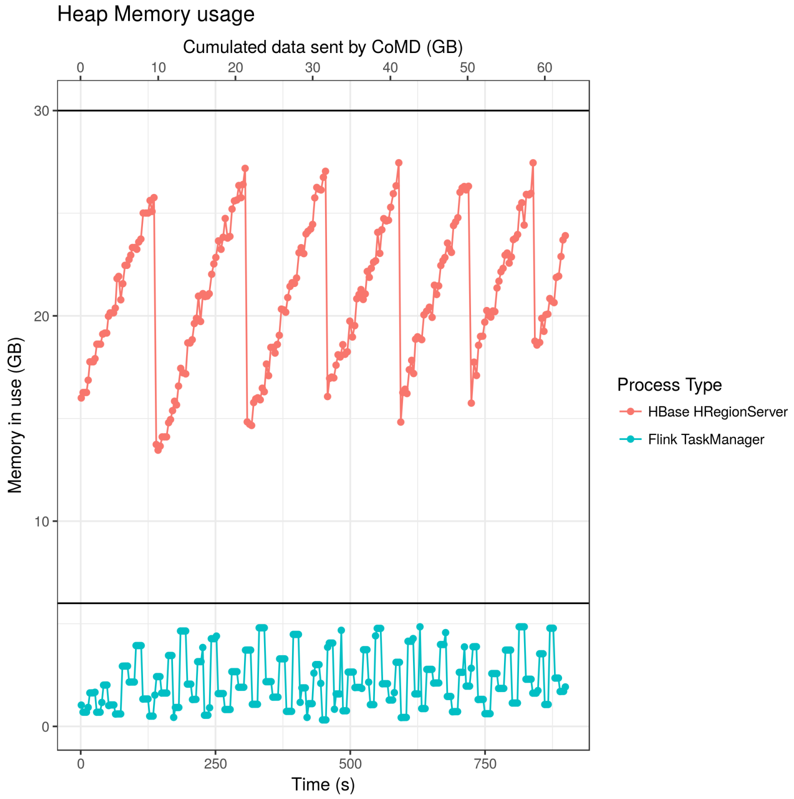


Figure A.4 – Heap usage of Flink's and HBase's worker nodes for the stable point of output interval 11. Flink's heap size is 6 GB, while HBase's is 30 GB

Bibliography

[1] Mark James Abraham, Teemu Murtola, Roland Schulz, Szilárd Páll, Jeremy C. Smith, Berk Hess, and Erik Lindahl. Gromacs: High performance molecular simulations through multi-level parallelism from laptops to supercomputers. *SoftwareX*, 1-2:19 – 25, 2015.

[2] Apache. Apache Flink 1.4 Documentation. `https://ci.apache.org/projects/flink/flink-docs-release-1.4/`, 2018. Online; accessed 2018-22-05.

[3] Prafulla Aryal, Mark S.P. Sansom, and Stephen J. Tucker. Hydrophobic gating in ion channels. *Journal of Molecular Biology*, 427(1):121 – 130, 2015. Understanding Functions and Mechanisms of Ion Channels.

[4] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In Ivan I. Ivanov, Marten van Sinderen, Frank Leymann, and Tony Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013.

[5] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink™: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38(4):28–38, 2015.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[7] M. Dreher and B. Raffin. A flexible framework for asynchronous in situ and in transit analytics for scientific simulations. In *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 277–286, May 2014.

[8] Exmatex. CoMD. `https://github.com/ECP-copa/CoMD`, 2016. Online; accessed 2018-28-05.

[9] Lars George. *HBase - The Definitive Guide: Random Access to Your Planet-Size Data.* O'Reilly, 2011.

[10] P. Hintjens. *ZeroMQ: Messaging for Many Applications*. Oreilly and Associate Series. O'Reilly Media, Incorporated, 2013.

[11] Matthieu Imbert, Laurent Pouilloux, Jonathan Rouzaud-Cornabas, Adrien Lèbre, and Takahiro Hirofuchi. Using the EXECO toolbox to perform automatic and reproducible cloud experiments. In *1st International Workshop on UsiNg and building ClOud Testbeds (UNICO, collocated with IEEE CloudCom 2013*, Bristol, United Kingdom, December 2013. IEEE.

[12] Erik Lindahl, Berk Hess, and David Spoel. Gromacs 3.0: A package for molecular simulation and trajectory analysis. 7:306–317, 08 2001.

[13] André Merzky, Mark Santcroos, Matteo Turilli, and Shantenu Jha. Radical-pilot: Scalable execution of heterogeneous and dynamic workloads on supercomputers. *CoRR*, abs/1512.08194, 2015.

[14] Naveen Michaud-Agrawal, Elizabeth J Denning, Thomas Woolf, and Oliver Beckstein. Mdanalysis: A toolkit for the analysis of molecular dynamics simulations. 32, 07 2011.

[15] Omar A. Mures, Emilio J. Padron, and Bruno Raffin. Leveraging the power of big data tools for large scale molecular dynamics analysis, 2016. XXVII Jornadas de Paralelismo (JP2016).

[16] Jack A. Orenstein. Spatial query processing in an object-oriented database system. In *SIGMOD Conference*, 1986.

[17] Ioannis Paraskevakos, André Luckow, George Chantzialexiou, Mahzad Khoshlessan, Oliver Beckstein, Geoffrey C. Fox, and Shantenu Jha. Task-parallel analysis of molecular dynamics trajectories. *CoRR*, abs/1801.07630, 2018.

[18] Cristian Ruiz, Salem Harrache, Michael Mercier, and Olivier Richard. Reconstructable Software Appliances with Kameleon. *Operating Systems Review*, 49(1):80–89, 2015.

[19] Tiankai Tu, C. A. Rendleman, D. W. Borhani, R. O. Dror, J. Gullingsrud, M. O. Jensen, J. L. Klepeis, P. Maragakis, P. Miller, K. A. Stafford, and D. E. Shaw. A scalable parallel framework for analyzing terascale molecular dynamics simulation trajectories. In *2008 SC - International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2008.

[20] H. Yu, C. Wang, R. W. Grout, J. H. Chen, and K. L. Ma. In situ visualization for large-scale combustion simulations. *IEEE Computer Graphics and Applications*, 30(3):45–57, May 2010.