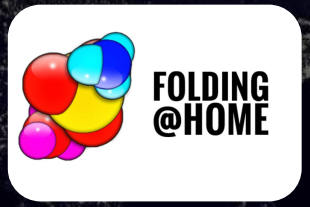
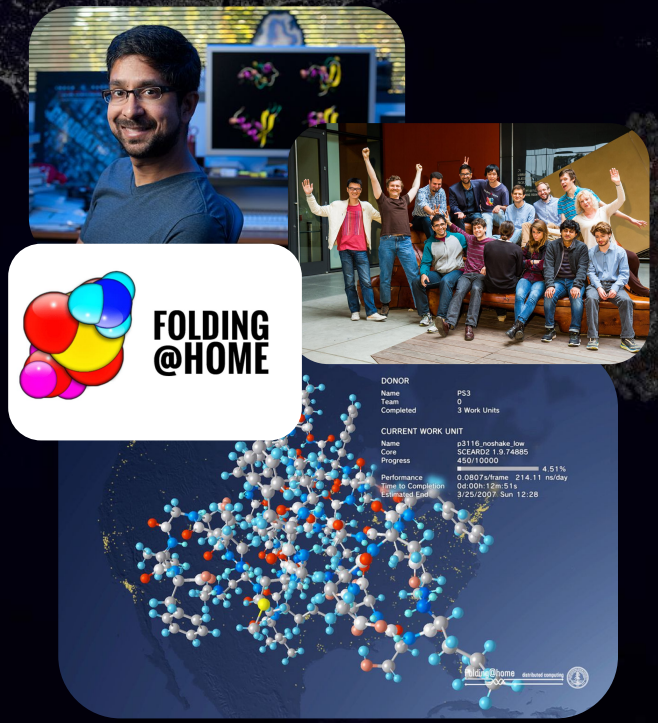


Computação Paralela e Distribuída na Simulação de Proteínas

Uma visão do projeto Folding@Home e da biblioteca OpenMM

MC970 - Introdução à Programação Paralela - 2023.1

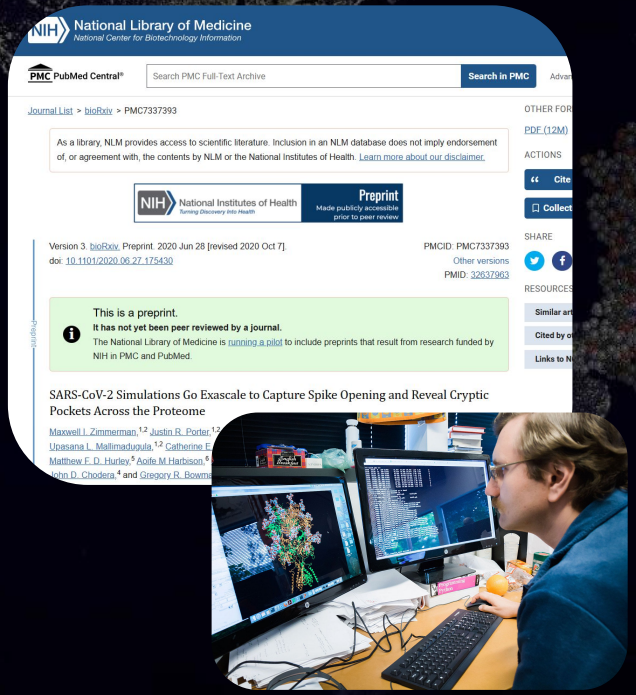
Victor Rigatto
178068



Folding@Home

- Projeto de computação distribuída voluntária
- Início nos anos 2000 na Universidade de Stanford
- Atualmente mantido na Universidade da Pensilvânia
- Distribui a simulação de dinâmica de proteínas para milhares de computadores voluntários espalhados pelo mundo
- Utilizado por diversas universidades e grupos de pesquisa
- Utiliza GPU e CPU em diversas plataformas e hardware
- Ampla utilização de paralelismo, com OpenMM e outros
- Classificado entre os top 10 supercomputadores
- Primeiro sistema a atingir computação em exascale

- https://foldingathome.org

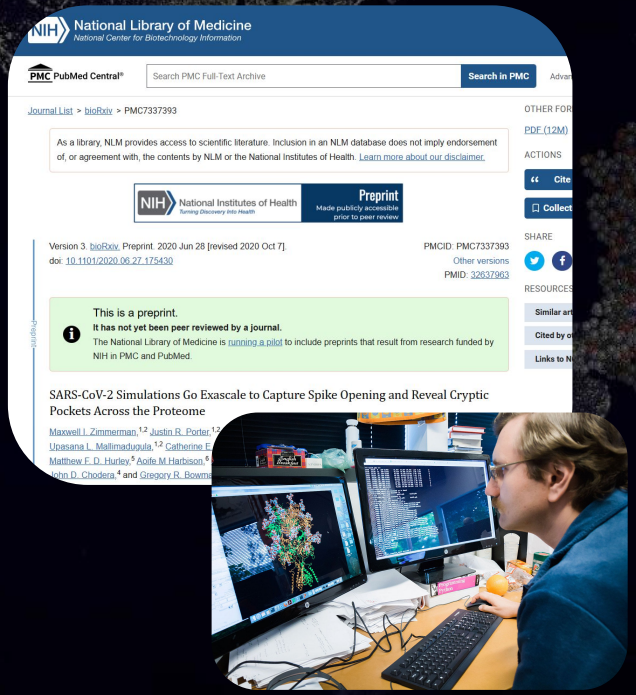


Folding@Home

- Durante a pandemia, atingiu ~1.01 exaFLOPS com ~280.000 GPUs e ~4.8 milhões de núcleos de CPU
- Em 30 de maio de 2023, contabilizava ~49 petaFLOPS
- Distribui a simulação em diferentes pedaços, de acordo com o sistema do usuário
- Centraliza os resultados para gerar a simulação completa
- Cada simulação levaria anos para completar em computadores comuns
- As bibliotecas por trás do Folding@Home fazem ampla utilização de paralelismo com CUDA e OpenCL (GPU), assim como utilizam OpenMP, Pthreads, e instruções SIMD nas diferentes versões do SSE e AVX (CPU)

- https://foldingathome.org

Folding@Home



- As proteínas simuladas são utilizadas para diversos estudos no campo da saúde, algumas são:
  - Terapias para diversos tipos de câncer
  - Alzheimer
  - Huntington
  - Fibrose cística
  - Covid-19
  - Medicamentos
- Recebe amplo suporte de empresas e instituições, como:

NVIDIA · BE · avast · arm · AMD · Microsoft

PURESTORAGE · vmware · ORACLE · neocortix · aws · intel

OpenMM

- Uma das bibliotecas por trás do Folding@Home
- Mantida pela Universidade de Stanford, institutos nacionais de pesquisa nos Estados Unidos e outros
- Especializada na criação de dinâmica de moléculas de maneira mais amigável, e OpenSource
- Especializada principalmente no processamento em GPUs, altamente paralelizada
- Utiliza CUDA (NVIDIA), OpenCL (AMD/Intel), OpenMP e Pthreads (CPUs), e MPI para clusterização
- API em C++ e em Python

- https://openmm.org

OpenMM

- Permite que usuários criem simulações de diversas formas, por exemplo:
  - Através de scripts que utilizam funcionalidades prontas, como kernels, escolhendo diversos parâmetros como substâncias, temperatura, pressão, etc. Para usuários como biologistas computacionais, que saberiam ou não programar
  - Através da criação das funcionalidades e dos kernels de forma personalizada, com completo controle, para usuários que sabem programar

- https://openmm.org

OpenMM

- Dividida em diferentes arquiteturas, notavelmente:
  - Plataforma CPU (Pthreads e OpenMP)
  - Plataforma OpenCL
  - Plataforma CUDA
  - Common Compute: criada para reduzir a complexidade aproveitando-se das semelhanças entre o OpenCL e CUDA. Permite uma implementação única para kernels que podem ser utilizadas por ambas as tecnologias
- Cada plataforma é utilizada para cada caso em que é mais eficiente, de acordo com o tipo de computação, além do sistema disponível

- https://openmm.org

OpenMM
Paralelização

Let $\mathbf{A}_1$ and $\mathbf{A}_2$ be rotation matrices

Let $\mathbf{r}_{12}$ be the vector pointing from particle 1 to particle 2

$$\mathbf{S}_i = \begin{bmatrix} a_i & 0 & 0 \\ 0 & b_i & 0 \\ 0 & 0 & c_i \end{bmatrix}$$

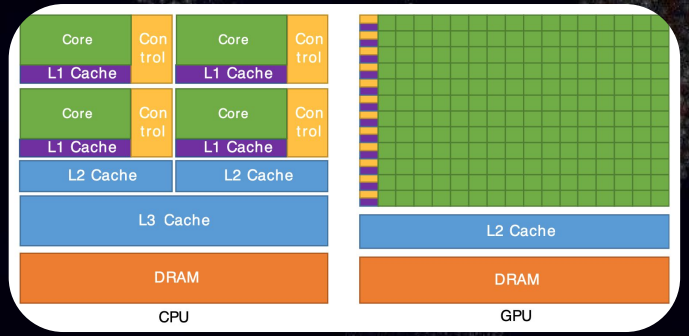
$$E = U_r(\mathbf{A}_1, \mathbf{A}_2, \mathbf{r}_{12}) \cdot \eta_{12}(\mathbf{A}_1, \mathbf{A}_2) \cdot \chi_{12}(\mathbf{A}_1, \mathbf{A}_2, \hat{\mathbf{r}}_{12})$$

$$m_i \ddot{\boldsymbol{x}}_i = -\gamma m_i \dot{\boldsymbol{x}}_i - \boldsymbol{\nabla}_i U(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N) + \sqrt{2k_B T\gamma}\, \boldsymbol{\eta}_i(t)$$

- Resumidamente, a dinâmica de moléculas envolve o cálculo de diversas funções que descrevem fenômenos físicos e químicos e suas interações, envolvendo forças, ângulos, distâncias, interpolação, entre outros
- São calculadas principalmente através de operações com matrizes e vetores, tornando-as altamente paralelizáveis e aceleráveis com instruções SIMD e SIMT
- Outro ponto favorável é que as simulações utilizam o modelo estatístico de Markov (o estado futuro depende apenas do estado atual), o que permite agregar simulações sem dependências de dados, e portanto melhor paralelizáveis

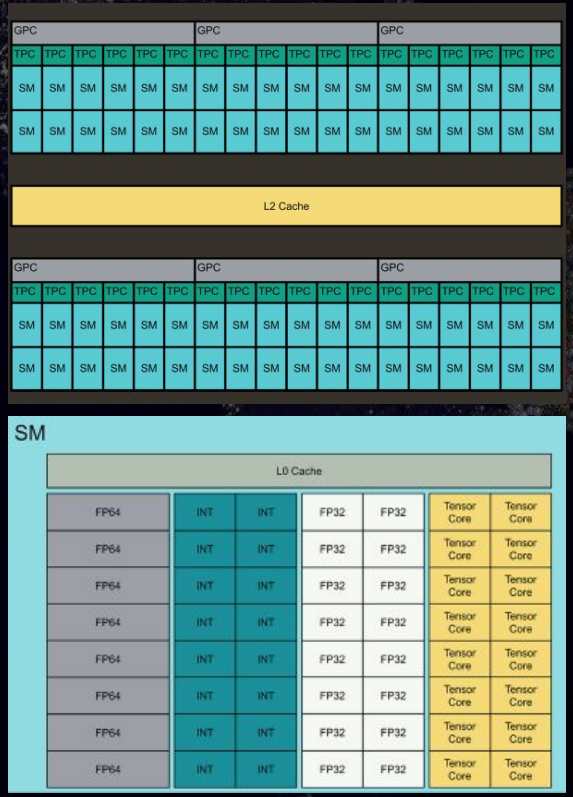
- https://openmm.org

OpenMM
Paralelização



- Como aprendemos, GPUs são especialmente eficientes em realizar operações repetidas com matrizes e vetores, e em grandes quantidades de dados, devido ao grande número de núcleos especializados em executar SIMD/SIMT e a arquitetura de memória minimizando transferências entre CPU-GPU
- Essa característica torna essas simulações em GPUs muito mais eficientes que em CPUs, que são projetadas para executar muitas instruções de natureza diferente e mais complexa, em menos núcleos
- Algumas dessas funções são especialmente ideais para GPUs, por requerer apenas ponto flutuante de precisão simples, como campos de força clássicos

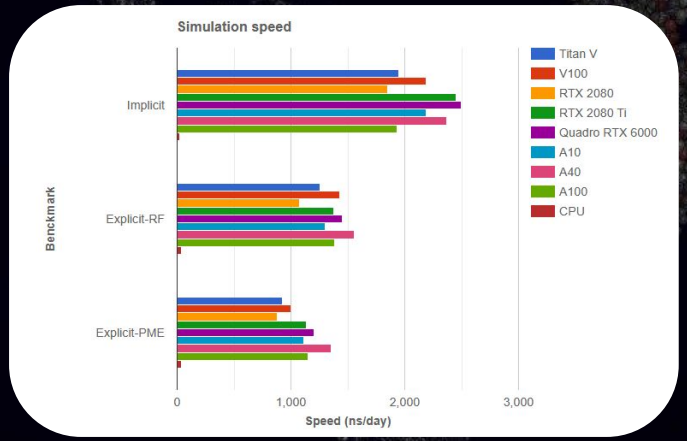
- https://openmm.org



OpenMM
Paralelização

- Por exemplo, no caso da arquitetura da NVIDA Tesla V100 ao lado, temos os GPU Processing Clusters (GPCs), cada um composto de Texture Processing Clusters (TPCs), e esses compostsos de multiplos Streaming Multiprocessors (SMs)
- Os SMs são os núcleos de computação de pontos flutuantes e aritmética, e de tensor cores que aceleram operações de multiplicação de matrizes
- Um thread pode ser a execução de um kernel, e grupos de threads são chamados thread blocks. Esses blocos podem executar em paralelo em diversos SMs
- A hierarquia dos grupos de SMs permitem o compartilhamento de recursos de memória em vários níveis de caches e da DRAM, além de existir o NVLink para comunicação rápida entre GPUs
    - https://openmm.org

OpenMM
Paralelização



ns/day: nanosegundos de simulação da dinâmica por dia

- Ao lado, benchmark do OpenMM de CPU x GPU
- Constata-se uma enorme diferença na capacidade de cálculo entre CPU e GPU, como esperado
- Abaixo, ganho em gigaFLOPS de SIMD em CPU, porém insuficiente para alcançar GPUs
- Ainda assim, CPUs são essenciais para fazer outras computações específicas que GPUs não podem fazer

- https://openmm.org

OpenMM
Plataforma CPU

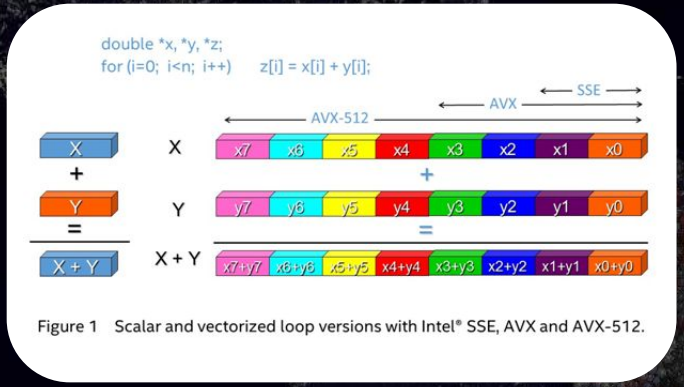


Figure 1 Scalar and vectorized loop versions with Intel® SSE, AVX and AVX-512.

```
 * Given a table of floating-point values and a set of indexes, perform a gather read into a pair
 * of vectors. The first result vector contains the values at the given indexes, and the second
 * result vector contains the values from each respective index+1.
 */
static inline void gatherVecPair(const float* table, ivec8 index, fvecAvx2& out0, fvecAvx2& out1) {
    const double* tableAsDbl = (const double*)table;

    // The input is a set of 8 indexes, each of which refers to a pair of floating point
    // values. The most efficient way to load from indexes in a vector is the gather instruction,
    // and the 64-bit variant should be used to get the pairs.

    // Given indexes ABCDEFGH, load the pairs corresponding to A C E G. This gives a set of
    // 4 pairs. The high indexes (in the upper part of each 64-bit index) are cleared.
    const auto lowerIdx = _mm256_and_si256(index, _mm256_set1_epi64x(0xFFFFFFFF));
    const auto lowerGather = _mm256_castpd_ps(_mm256_i64gather_pd(tableAsDbl, lowerIdx, 4));

    // Load indexes B D F H, this time by shifting the high 32-bit indexes into the lower 32-bits.
    const auto upperIdx = _mm256_srli_epi64(index, 32);
    const auto upperGather = _mm256_castpd_ps(_mm256_i64gather_pd(tableAsDbl, upperIdx, 4));
```

- Utiliza OpenMP e Pthreads, e tenta utilizar sempre que possível instruções SSE e AVX (AVX2 e AVX-512 estão sendo implementados)
- Desenvolvedores utilizam perfilamento para avaliar quanto do processamento está sendo feito com essas instruções
- O código ao lado foi extraído de um arquivo fonte do OpenMM que cria classes e funções para vetorizar dados utilizando AVX
- Nesse caso, uma operação de gathering em múltiplos índices em um par de vetores, útil para operações de álgebra linear, por exemplo

- https://openmm.org

OpenMM
Plataforma OpenCL

```
#pragma OPENCL EXTENSION cl_khr_global_int32_base_atomics : enable
#pragma OPENCL EXTENSION cl_khr_byte_addressable_store : enable

/**
 * Find a bounding box for the atoms in each block.
 */
__kernel void findBlockBounds(int numAtoms, real4 periodicBoxSize, real4
        __global const real4* restrict posq, __global real4* restrict blo
        __global real2* restrict sortedBlocks) {
    int index = get_global_id(0);
    int base = index*TILE_SIZE;
    while (base < numAtoms) {
        real4 pos = posq[base];
#ifdef USE_PERIODIC
        APPLY_PERIODIC_TO_POS(pos)
#endif
        real4 minPos = pos;
```

```
/**
 * An alternate kernel for sorting short lists.  In this version every thread does a full
 * scan through the data to select the destination for one element.  This involves more
 * work, but also parallelizes much better.
 */
__kernel void sortShortList2(__global const DATA_TYPE* restrict dataIn, __global DATA_TYPE*
    __local DATA_TYPE dataBuffer[64];
    DATA_TYPE value = dataIn[get_global_id(0) < length ? get_global_id(0) : 0];
    KEY_TYPE key = getValue(value);
    int count = 0;
    for (int blockStart = 0; blockStart < length; blockStart += get_local_size(0)) {
        int numInBlock = min((int) get_local_size(0), length-blockStart);
        barrier(CLK_LOCAL_MEM_FENCE);
        if (get_local_id(0) < numInBlock)
            dataBuffer[get_local_id(0)] = dataIn[blockStart+get_local_id(0)];
        barrier(CLK_LOCAL_MEM_FENCE);
        for (int i = 0; i < numInBlock; i++) {
            KEY_TYPE otherKey = getValue(dataBuffer[i]);
            if (otherKey < key || (otherKey == key && blockStart+i < get_global_id(0)))
                count++;
        }
    }
    if (get_global_id(0) < length)
        dataOut[count] = value;
}
```

- Existem classes responsáveis por realizar inúmeras funções no contexto do OpenCL, por exemplo para gerenciar a memória entre host e device
- Ao lado, códigos do OpenMM para criar um kernel, ordenar listas de forma paralela com sincronização barrier() e copiar dados para o device através de uma classe

```
/**
 * Copy the values in a vector to the device memory.
 *
 * @param data      the data in host memory to copy
 * @param convert   if true, automatic conversions between single and double
 *                  precision will be performed as necessary
 */
template <class T>
void upload(const std::vector<T>& data, bool convert=false) {
    if (convert && data.size() == getSize() && sizeof(T) != getElementSize()) {
        if (sizeof(T) == 2*getElementSize()) {
            // Convert values from double to single precision.
            const double* d = reinterpret_cast<const double*>(&data[0]);
            std::vector<float> v(getElementSize()*getSize()/sizeof(float));
            for (int i = 0; i < v.size(); i++)
                v[i] = (float) d[i];
```

- https://openmm.org

OpenMM
Plataforma CUDA

```
// Sum the errors over threads and store the total for this block.

buffer[threadIdx.x] = make_real2(sumErrors, sumPolarErrors);
__syncthreads();
for (int offset = 1; offset < blockDim.x; offset *= 2) {
    if (threadIdx.x+offset < blockDim.x && (threadIdx.x&(2*offset-1)) == 0) {
        buffer[threadIdx.x].x += buffer[threadIdx.x+offset].x;
        buffer[threadIdx.x].y += buffer[threadIdx.x+offset].y;
    }
    __syncthreads();
}
if (threadIdx.x == 0)
    errors[blockIdx.x] = make_float2((float) buffer[0].x, (float) buffer[0].y);
```

- Semelhante ao OpenCL, com classes para gerenciar funções
- Utiliza apenas operações atômicas de 64 bits
- O OpenMM cuida da compilação em runtime
- Ao lado, uma sincronização de threads para somar dados, um kernel para somar forças, e o carregamento de um átomo

```
/**
 * Sum the forces computed by different contexts.
 */

extern "C" __global__ void sumForces(long long* __restrict__ force, long long* __restrict__ buffer, int buffer
    int totalSize = bufferSize*numBuffers;
    for (int index = blockDim.x*blockIdx.x+threadIdx.x; index < bufferSize; index += blockDim.x*gridDim.x) {
        long long sum = force[index];
        for (int i = index; i < totalSize; i += bufferSize)
            sum += buffer[i];
        force[index] = sum;
    }
}
```

```
inline __device__ void loadAtomData(AtomData& data, int atom, const real4* __restri
        const real3* __restrict__ inducedDipolePolar, const float2* __restrict__ dar
        const real3* __restrict__ inducedDipolePolarS, const real* __restrict__ bor
#else
inline __device__ void loadAtomData(AtomData& data, int atom, const real4* __restri
        const real3* __restrict__ inducedDipolePolar, const float2* __restrict__ dar
#endif
    real4 atomPosq = posq[atom];
    data.pos = make_real3(atomPosq.x, atomPosq.y, atomPosq.z);
    data.inducedDipole = inducedDipole[atom];
```

- https://openmm.org

OpenMM
Common Compute

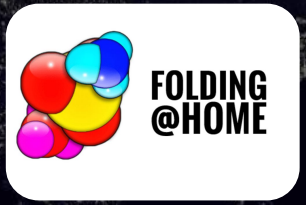
```
__kernel void addArrays(__global const float* restrict a,
                        __global const float* restrict b,
                        __global float* restrict c
                        int length) {
    for (int i = get_global_id(0); i < length; i += get_global_size(0))
        c[i] = a[i]+b[i];
}
```

```
__extern "C" __global__ void addArrays(const float* __restrict__ a,
                                       const float* __restrict__ b,
                                       _float* __restrict__ c
                                       int length) {
    for (int i = blockIdx.x*blockDim.x+threadIdx.x; i < length; i += blockD
        c[i] = a[i]+b[i];
}
```

```
KERNEL void addArrays(GLOBAL const float* RESTRICT a,
                      GLOBAL const float* RESTRICT b,
                      GLOBAL float* RESTRICT c
                      int length) {
    for (int i = GLOBAL_ID; i < length; i += GLOBAL_SIZE)
        c[i] = a[i]+b[i];
}
```

- Contém classes e métodos que são aplicáveis de forma análoga no CUDA e no OpenCL, simplificando a utilização da biblioteca
- Ao lado, o primeiro código é a adição de duas arrays em OpenCL. O segundo é a mesma operação utilizando CUDA. O terceiro é o Common Compute

- https://openmm.org



Referências

Slides da disciplina;
https://foldingathome.org, Acesso em 05 jul 2023;
https://openmm.org, Acesso em 05 jul 2023;
http://docs.openmm.org/latest/developerguide/01_introduction.html, Acesso em 05 jul 2023;
https://github.com/openmm, Acesso em 05 jul 2023;
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6680170, Acesso em 05 jul 2023;
https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2933958, Acesso em 05 jul 2023;
https://escholarship.org/content/qt6n48n8gd/qt6n48n8gd.pdf, Acesso em 05 jul 2023;
https://www.nature.com/articles/s41557-021-00707-0, Acesso em 05 jul 2023;
https://ftp.mi.fu-berlin.de/pub/ag-cmb/pyemma-workshop-2019/slides/intro.pdf, Acesso em 05 jul 2023;
https://comp.anu.edu.au/courses/acceleratorsHPC/slides/Synchronization.pdf, Acesso em 05 jul 2023;


Obrigado