# Speedup Evaluation of the Training Algorithm for a Multilayer Perceptron based on Colored Petri Nets Simulations

**Matheus Dias Gama**[1], **Stéphane Julia**[1], **Rita Maria Silva Julia**[1]

[1]Faculdade de Computação
Universidade Federal de Uberlândia (UFU) – Uberlândia – MG – Brazil

{matheus.dias,stephane,rita}@ufu.br

***Abstract.*** *The complexity of distributed algorithm's distribution policies often impedes the use of established mathematical approaches for performance analysis, such as asymptotic analysis, recurrence techniques and probabilistic analysis. The aforementioned techniques have no proper means to assess the impact on the algorithm's run time caused by the gradual increment of the number of processors. Therefore, this article proposes both a visual and formal approach for evaluating the processor saturation point and speedup of distributed algorithms utilized in Artificial Intelligence, based on automatic simulations of Hierarchical Colored Petri Nets in the Colored Petri Nets Tools software. The aforementioned approach was demonstrated in a case study on the training of Multilayer Perceptrons based on error backpropagation.*

***Resumo.*** *Frequentemente, a complexidade da política de distribuição dos algoritmos distribuídos os tornam extremamente hostis a consagradas abordagens matemáticas de análise de desempenho, tais como análise assintótica, técnicas de recorrências e análise probabilística. Isso se deve ao fato de que tais métodos não provêm recursos adequados que lhes permitam avaliar o quão o gradual aumento do número de processadores impacta no tempo de execução do algoritmo. Diante disso, este artigo propõe uma abordagem visual e formal, baseada em simulações automáticas de modelos de Redes de Petri Coloridas Hierárquicas no ambiente gráfico Colored Petri Nets Tools (CPN Tools), para avaliar o speedup e o ponto de saturação de processadores de algoritmos distribuídos usados em Inteligência Artificial. Será usado como estudo de caso o algoritmo de treinamento dos Perceptrons de Múltiplas Camadas baseado na retro-propagação do erro.*

## 1. Introdução

No cenário fortemente automatizado da sociedade moderna, a análise da eficiência de algoritmos se torna um foco primordial de estudo. Duas métricas são essenciais nessa análise: a memória requerida e o tempo de execução gasto [Cormen 2009]. Com a evolução e a desoneração do hardware, a primeira dessas métricas passou a representar um desafio relativamente menor quando comparado à segunda. Dessa forma, a avaliação do tempo de execução se tornou o foco mais relevante na análise da qualidade das soluções algorítmicas. Dentre as abordagens analíticas consagradas disponíveis para tal fim, destacam-se [Cormen 2009]: a análise assintótica, a qual mede o impacto do aumento da

dimensão do dado de entrada no tempo de execução; as técnicas que resolvem as recorrências, usadas para avaliar os algoritmos recursivos (método da substituição, método da árvore de recursão ou método Máster); e a análise probabilística, que usa probabilidades de distribuição dos dados de entrada para estimar o tempo de execução.

Entretanto, há algoritmos que apresentam certas peculiaridades que tornam o uso dessas técnicas puramente matemáticas de avaliação de desempenho inadequadas ou excessivamente árduas. É o caso dos algoritmos distribuídos em que, dependendo da complexidade da política de distribuição utilizada, a avaliação por meio de métodos analíticos do efeito de um gradual incremento de processadores no seu tempo de execução pode tornar-se impraticável (o que ocorre, por exemplo, sempre que o tempo de execução depende de atualizações de valores de parâmetros efetuadas via troca de mensagens [Barbosa 1996]).

Outros autores propuseram métodos empíricos baseados na complexidade computacional para entender a escalabilidade de programas [Goldsmith et al. 2007]. Entende-se por escalabilidade a capacidade de ganho de desempenho do programa em função do incremento das cargas de trabalho [Coulouris 2012]. Em [Borovska and Lazarova 2007], foi proposto um método empírico baseado na comparação de desempenho de modelos de programação paralelos para avaliar o *speedup* (relação entre os tempos gastos para executar um algoritmo com um único processador e com N processadores, avaliando, assim, a variação produzida pelo paralelismo no tempo de processamento), a *eficiência* e a escalabilidade de um algoritmo de busca paralelo. Contudo, tais alternativas apresentam o seguinte inconveniente: elas requerem a implementação dos algoritmos analisados, o que pode ter um efeito perverso particularmente no caso dos algoritmos distribuídos, uma vez que isso demandaria a aquisição prévia de recursos de *hardware* dispendiosos de multiprocessamento, antes mesmo de saber se a proposta de distribuição investigada, de fato, vale a pena.

Vários trabalhos utilizaram técnicas analíticas (focadas em métodos puramente matemáticos) para verificar a complexidade das Redes Neurais Artificiais (RNA). Em [Serpen and Gao 2014], os autores analisaram a complexidade computacional do treinamento de um Perceptron Multicamadas implementado de forma distribuída em uma rede de sensores sem fio (RSSF). Em [Kon and Plaskota 2000], os autores apresentaram uma análise de complexidade de uma rede neural de três camadas na qual cada neurônio pode ter uma função de ativação distinta na alimentação direta (feedfoward). Foram obtidos resultados inéditos para todas as classes de funções de ativação. O incoveniente das abordagens puramente analíticas é a necessidade de uma nova análise para cada novo tipo de arquitetura de algoritmo ou de rede neural. Se algumas características mudam na arquitetura da distribuição, novos resultados teóricos precisam ser produzidos.

Considerando o panorama acima apresentado do Estado da Arte, o presente trabalho tem como objetivo principal produzir uma abordagem visual e formal, baseada nos resultados apresentados em [Júnior et al. 2018], para avaliar o desempenho de algoritmos distribuídos usados em Inteligência Artificial por meio do cálculo do *speedup* e do ponto de saturação de processadores (número máximo de processadores que contribuem para reduzir o tempo de processamento de tais algoritmos). Tal abordagem será implementada por meio de modelos baseados em Redes de Petri Coloridas Hierárquicas (RdPCH) simulados no ambiente gráfico *CPN Tools* [Jensen and Kristensen 2009].

Usa-se como estudo de caso o algoritmo de treinamento de um perceptron multicamadas (PMC) [Hassoun et al. 1995].

## 2. Fundamentação Teórica

Resumem-se aqui os fundamentos teóricos essenciais à compreensão do presente trabalho.

### 2.1. Redes de Petri

Como resultado de suas pesquisas de Doutorado, Carl Adam Petri apresentou um tipo de grafo bipartido com estados associados para estudar a comunicação entre autômatos [Murata 1989]. Posteriormente, a título de homenagem da comunidade científica, tal proposta recebeu o nome de *Redes de Petri (RdP)*. As RdP são ferramentas matemáticas, gráficas e formais que permitem a modelagem, a análise, o controle e a supervisão de sistemas dinâmicos, produzindo uma abordagem discreta dos mesmos. Os elementos básicos das RdP são: o lugar, representado por um círculo, que pode modelar um estado parcial, uma condição, uma espera, ou um procedimento; a transição, representada por uma barra ou retângulo, que modela um evento de início ou fim de um estado; o arco, representado por um arco direcionado, que liga transição(ações) a(aos) lugar(res) e viceversa; a ficha ou marca, representada por um ponto em um lugar, que modela a ocorrência de um estado.

As RdP são ferramentas de modelagem dinâmica, pois permitem acompanhar a evolução do sistema modelado por meio da noção de fluxos de fichas. A mudança da ficha de um lugar ao outro depende do modelo de RdP utilizado, o qual definirá uma política de disparo específica. Por exemplo, se o modelo é uma RdP temporizada com tempos de execução de operações associados com as transições do modelo, os disparos de transições não são instantâneos e dependem das durações associadas às transições do modelos. Porém, se o modelo é uma RdP Predicado/Transição, os disparos de transições são instantâneos, mas ocorrem somente quando são satisfeitas condições associadas às transições do modelo e que correspondem a Predicados de uma lógica de primeira ordem.

### 2.2. Redes de Petri Coloridas

Uma Rede de Petri Colorida (RdPC) é uma rede na qual cada lugar possui um tipo abstrato de dado que define um conjunto $U$ [Jensen and Kristensen 2009]. Tal conjunto é um universo contendo todos os possíveis valores pertencentes a este tipo. Um multi-conjunto sobre $U$ é então um mapeamento $f : U \rightarrow N$ que atribui para cada elemento $u \in U$ um número natural $f(u)$. Cada elemento $u \in U$ especifica um valor do tipo do lugar representado por $U$. Além disso, um arco em uma RdPC pode ter uma etiqueta (ou rótulo) associada à ele. Esta etiqueta é uma expressão com algumas variáveis que são avaliadas sobre um multi-conjunto. Uma transição pode ter uma Guarda ou Expressão de guarda associada a ela. Esta Guarda é uma expressão booleana que pode ter variáveis que correspondem às etiquetas dos arcos de entrada e saída da referida transição. Uma vez que cada lugar de uma RdPC possui um tipo (cor) associado a ele, as fichas que são inseridas nesse lugar devem estar em conformidade com seu tipo que deve ser previamente declarado. Com isso, as RdPC são uma linguagem de modelagem gráfica que permite a associação da potencialidade das RdP - representação da sincronização e da concorrência em

sistemas distribuidos, por exemplo - com o formalismo das linguagens de programação funcionais.

Um ambiente de modelagem e simulação que tem contribuído muito com a "popularização" do uso das RdPC e de suas extensões é o CPN Tools [Jensen et al. 2018]. Esse ambiente é composto por um conjunto de ferramentas que permitem ações, tais como editar, simular, analisar espaços de estados e avaliar o desempenho dos modelos criados. Na ferramenta, o usuário pode trabalhar explorando um ambiente gráfico com diversos recursos. Além disso, é possível o uso combinado da programação funcional com o modelo a ser criado de forma bem intuitiva e prática.

## 3. Modelo de Simulação de uma Rede Neural Artificial

O treinamento de um PMC tem por objetivo ajustar os pesos de seus neurônios em função dos erros que ele produz em sua saída ao processar o conjunto de dados (ou amostras) de treinamento. No caso, tais reajustes serão efetuados de forma a tentar minimizar os erros produzidos, sendo conduzidos sob a estratégia de *backpropagation*, ou seja, os erros produzidos nos neurônios de saída servirão de base para sucessivos cálculos dos gradientes locais dos neurônios da rede (partindo dos neurônios da última camada e prosseguindo em direção aos das camadas ocultas) [Hristev 1998]. Tais gradientes representam os parâmetros fundamentais no dimensionamento do valor dos reajustes de peso a serem efetuados nos neurônios do PMC.

Em [Hanzálek 2003], Hanzáleck propõe um modelo introdutório baseado em RdP ordinárias (RdP originais apresentadas na Seção 2.1) para analisar partes do algoritmo de treinamento de um PMC que podem ser paralelizadas. Contudo, seu modelo não conta com recursos que lhe permitam avaliar o maior valor possível de *speedup* que o algoritmo de treinamento pode atingir sendo processado com *n* processadores, nem o ponto de saturação dos processadores - que são os objetivos primordiais do presente trabalho. Assim sendo, as subseções 3.1 e 3.2 apresentam, respectivamente, o modelo introdutória de Hanzáleck e os modelos expandidos aqui criados com a finalidade de atingir tais objetivos (modelos baseado em RdPC simulado no ambiente *CPN Tools*).

### 3.1. Modelo de Hanzálek

Hanzálek [Hanzálek 2003], por meio das RdP ordinárias, propõe um modelo para detectar partes do algoritmo de treinamento de um PMC que podem ser paralelizadas. O PMC usado por ele como estudo de caso apresenta uma camada de entrada (camada *0*) com dois neurônios, duas camadas ocultas (camadas *1* e *2*) com quatro neurônios e uma camada de saída (camada *3*) com dois neurônios. As Figuras 1(a) e 1(b) mostram, respectivamente, a arquitetura do PMC e o modelo proposto, sendo que este último retrata as conhecidas etapas inerentes ao treinamento de um PMC, as quais se referem aos cálculos dos seguintes valores relativos aos neurônios: sinais de ativação e de saída; erro; gradientes locais; reajuste dos vetores de peso [Hristev 1998].

Em seu modelo, Hanzálek representa as entradas da rede pelo lugar $P0$ e a aplicação da função sigmóide aos sinais de entrada pela transição $T0$.

Os processos de ativação são representados pelas transições $T1$, $T2$ e $T3$ na Figura 1 (b). O sinal de ativação de um neurônio *i* de uma das camadas ocultas ou da camada de

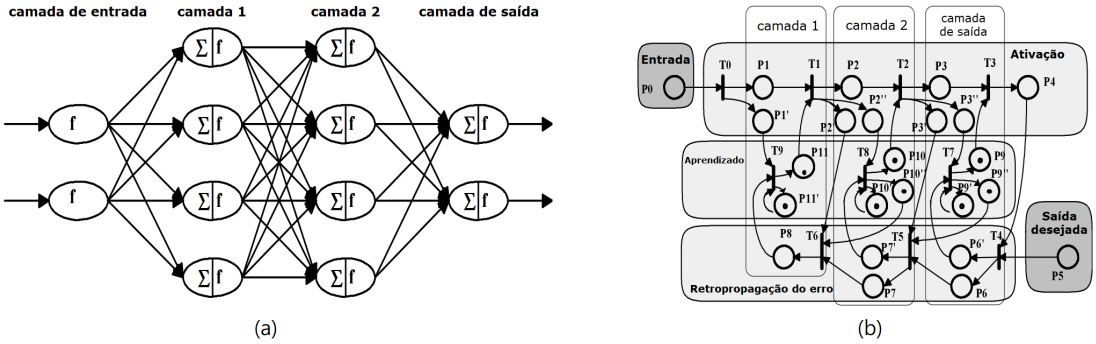


**Figura 1. Arquitetura do PMC (Figura (a)) e Modelagem em RdP de seu algoritmo de aprendizagem [Hanzálek 2003] (Figura (b))**

saída é calculado a partir dos pesos e dos sinais de saída de todos os neurônios da camada *i-1*.

Os lugares $P1$, $P2$, $P3$ e $P4$ representam as saídas de todos os neurônios de cada camada, sendo $P1'$ a representação de uma cópia das saídas representadas em $P1$. $P2'$ e $P2''$ são representações de cópias das saídas em $P2$. $P3'$ e $P3''$ são representações de cópias das saídas em $P3$.

Os lugares $P11$, $P10$ e $P9$ representam, respectivamente, os pesos entre a camada de entrada e a primeira camada oculta, entre a primeira e a segunda camadas ocultas, e entre a segunda camada oculta e a camada de saída. $P11'$ representa uma cópia de $P11$. $P10'$ e $P10''$ representam cópias de $P10$. $P11'$ e $P11''$ representam cópias de $P11$.

Após a transição $T3$ ser disparada e aparecer uma ficha no lugar $P4$, a transição $T4$, que representa o cálculo do gradiente local dos neurônios da camada de saída, é sensibilizada, utilizando, para tal cálculo, o valor do *erro*, que é obtido a partir da representação do sinal de saída a que se refere o lugar *P4* e da representação do sinal desejado a que se refere o lugar *P5*. O gradiente local calculado fica representado por fichas no lugar $P6$, e uma cópia desse gradiente no lugar $P6'$.

Utilizando o conceito de retropropagação, o gradiente local de um neurônio *i* de uma camada oculta *l* é calculado a partir do valor do gradiente local dos neurônios da camada *l+1* e dos pesos que ligam o neurônio *i* aos neurônios da camada *l+1*. No modelo, os gradientes locais dos neurônios da primeira e da segunda camadas ocultas estão representados, respectivamente, pelos lugares *P7* e *P8*.

Finalmente, cada *link* de peso $w_{ij}$ de cada neurônio *j* da rede é ajustado com base nos seguintes valores: gradiente local de *j*, valor corrente de $w_{ij}$ e sinal de saída produzido pelo neurônio *i* da camada anterior dos neurônios de cada camada. Tais reajustes estão representados nas transições *T7*, *T8* e *T9*. Apenas ao final do disparo de *T9*, um novo dado de entrada poderá ser processado, já que a transição *T1* será sensibilizada novamente com a criação de uma ficha em *P11*.

As equações que apresentam o comportamento detalhado do algoritmo de treinamento em cada etapa de sua execução, em particular as equações associadas a cada transição do modelo apresentado na Figura 1 (b) do Hanzálek, se encontram em [Hanzálek 2003].

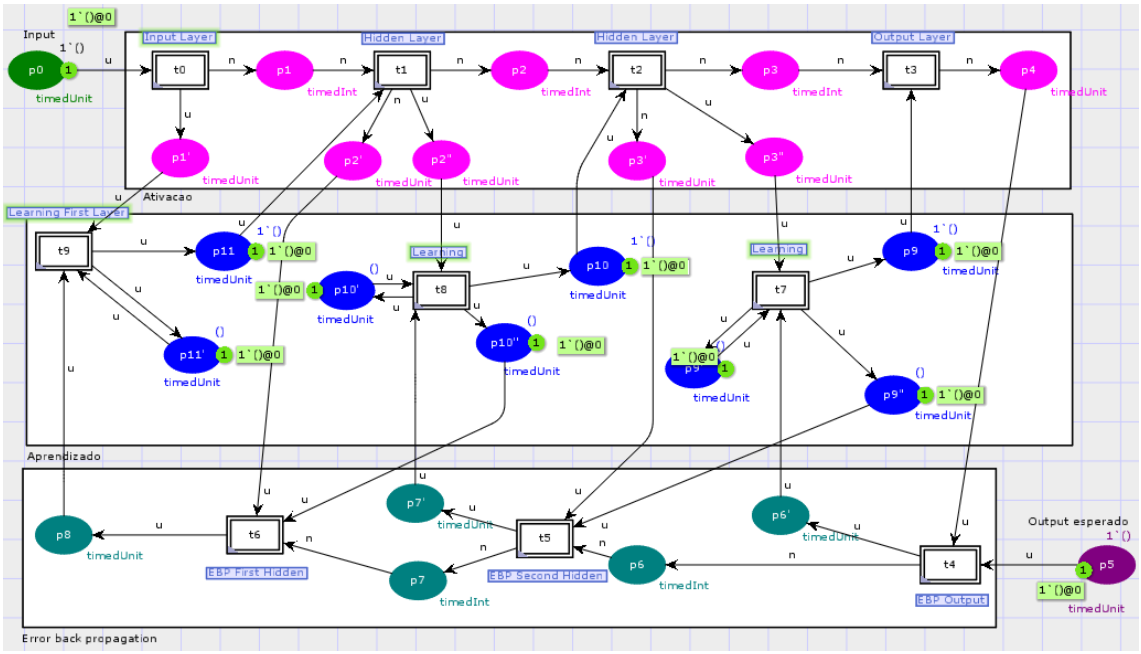


**Figura 2. RdPCH que representa o treinamento on-line do PMC**

### 3.2. Modelo RdPC simulado no CPN Tools

Esta subseção apresenta a implementação, no ambiente do *CPN Tools*, de dois modelos baseados em RdPCH para cálculo do *speedup* máximo do algoritmo do treinamento de PMCs executado em $n$ processadores $(n \geq 1)$, o que representa a proposta deste artigo. Ambos os modelos apresentam a mesmas potencialidades, diferindo apenas na forma do treinamento modelado (on-line e em mini-batches).

Tais modelos hierarquizados permitem expandir cada transição do modelo original de Hanzáleck apresentado na subseção 3.1, tornando possível a representação individual de cada neurônio e de cada recurso compartilhado (processador), o que, consequentemente, permite a representação detalhada do processamento de cada neurônio, por cada processador (o modelo original somente permite representar o processamento em bloco de todos os neurônios de cada camada).

Além da hierarquização, nos modelos aqui propostos também foram adicionadas cores aos lugares, possibilitando a utilização de fichas com valores inteiros e de fichas temporizadas, que são necessárias para simular o tempo necessário para a realização de cada operação elementar do algoritmo.

As principais formas de treinamento de um PMC são [Brownlee 2019]: treinamento *on-line*, onde os pesos são reajustados a cada dado de entrada apresentado (com base no erro individual produzido por cada amostra); treinamento em *mini-batches*, em que os dados de entrada são divididos em lotes menores e os pesos somente são reajustados após o processamento de todos as amostras de cada mini-batch (com base no erro médio produzido pelo conjunto de amostras do mini-batch); treinamento em *batches*, que é um caso extremo de mini-batch em que as amostras de treinamento são agrupadas em um único lote.

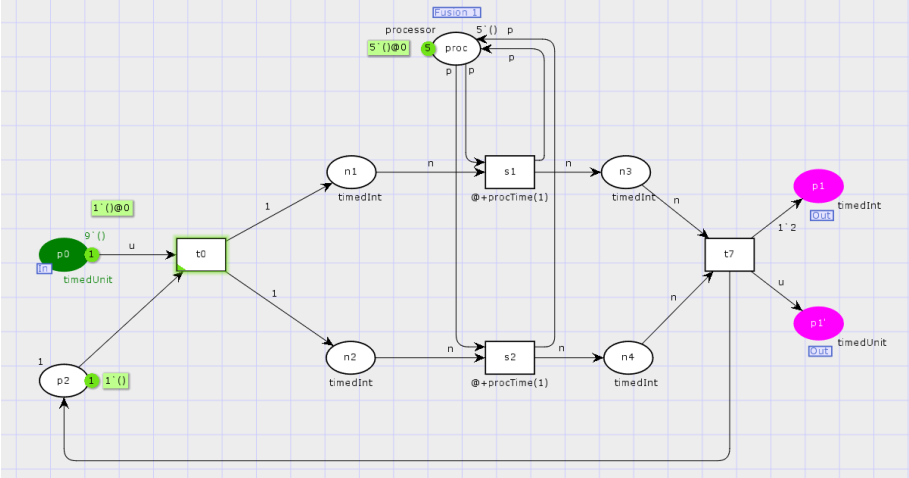


**Figura 3. Expansão da transição t0 da Figura 2**

### 3.2.1. Modelo de Treinamento On-line do PMC

A Figura 2 mostra o modelo aqui proposto para processamento on-line. Por meio dele pode-se observar que os recursos de hierarquização e temporização da RdPCH permitem, por exemplo, expandir a transição *t0* da Figura 2, gerando a sub-rede ilustrada na Figura 3. Cada ficha no lugar de fusão $proc$ $(Fusion1)$ representa um processador, cada um com um tempo discreto associado, tempo esse que simboliza a data de disponibilidade do processador. Esse lugar $proc$, que utiliza a ferramenta de lugar de fusão, está conectado a todas as transições que representam operações elementares na rede. A ferramenta lugar de fusão permite especificar um conjunto de lugares que são considerados idênticos [Jensen and Kristensen 2009]. Isto significa que todos esses lugares representam um único lugar conceitual, ainda que sejam desenhados como vários lugares individuais. Assim o lugar *proc* que representa a disponibilidade dos processadores poderá ser associado a todas as transições que representam operações no modelo. Para representar uma quantidade infinita de processadores, é necessário apenas remover o lugar $proc$ ou os arcos que o ligam a todas as transições que representam operações elementares. Dessa forma, não há nenhuma restrição de utilização de recurso e as transições podem ser disparadas assim que forem sensibilizadas, independentemente da disponibilidade de um recurso do tipo processador.

Ressalte-se que a transição $t0$ na Figura 2 representa a ativação da função sigmóide na camada de entrada da rede e que cada uma das transições $s1$ e $s2$ da Figura 3 representa a ativação da função sigmóide em cada um dos neurônios dessa camada. Ao disparar essas transições, as fichas geradas em $n3$, $n4$ e $proc$ têm os tempos associados aumentados em uma unidade de tempo discreta (+1), fazendo com que elas só possam ser utilizadas em uma unidade de tempo depois do disparo da transição. Leva-se em consideração que esse tempo discreto (+1), do ponto de vista da análise de algoritmos, representa a execução de uma operação elementar, como geralmente é o caso em exemplos de análises de complexidade assintótica de algoritmos tradicionais.

A inscrição no arco entre a transição $t7$ e o lugar $p1$ contém a informação de quantos neurônios existem na camada de entrada, a qual será necessária para calcular o

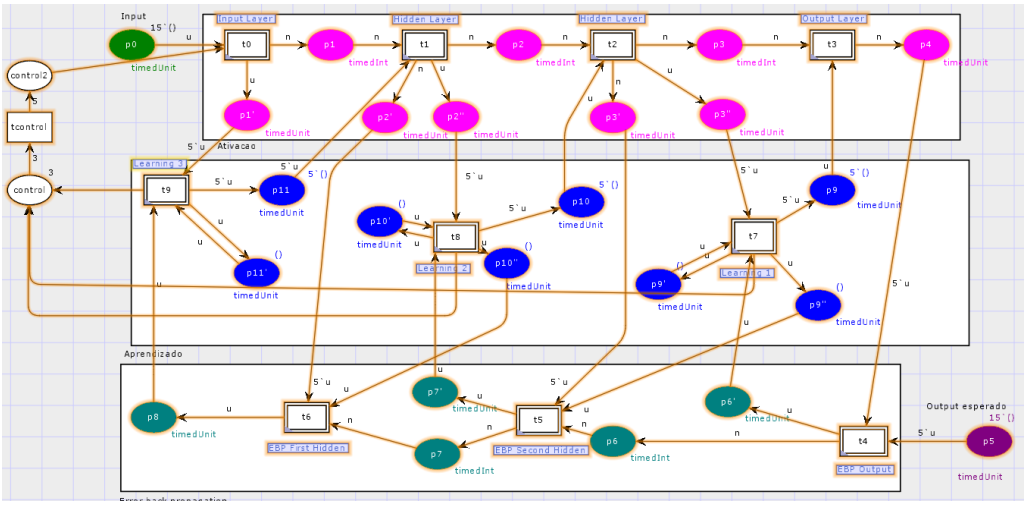


**Figura 4. RdPCH que representa o treinamento de um PMC em mini-batches**

tempo computacional das ativações das camadas subsequentes. O lugar $p2$ garante que os sinais de ativação dos neurônios de uma camada oculta ou da camada de saída somente sejam calculados depois que tiverem sido concluídos os cálculos do sinal de saída de todos os neurônio da camada anterior.

Assim como para a transição $t0$, todas as outras transições do modelo na Figura 2 foram detalhadas em sub-redes no *CPN Tools* de acordo com as equações apresentadas em [Hanzálek 2003].

### 3.2.2. Modelo de Treinamento em Mini-Batches do PMC

A Figura 4 mostra a adaptação do modelo hierárquico da Figura 2 para um treinamento em mini-batches (que serve também para batches, uma vez que este último tipo de treinamento representa um caso particular do primeiro). Observa-se que, na referida figura, a título de exemplo, o tamanho de cada lote é *5*.

Nota-se também na Figura 4 a adição dos lugares *control* e *control2* e da transição *tcontrol*. Sempre que se concluir o processo de atualização dos pesos entre um par de camadas consecutivas do PMC estudado, será gerada uma ficha no lugar *control*. Logo, a cada vez que se concluir um processo de retropropagação em que todos pesos da rede tiverem sido ajustados, haverá *3* fichas no referido lugar (uma para cada par), o que sensibiliza e provoca o disparo da transição *tcontrol*. Após tal disparo, *n* fichas serão alocadas em *control2*, onde *n* representa o tamanho do *mini-batch* (no exemplo da Figura 4, *n*=*5*). As fichas em $control2$ serão consumidas, uma a uma, a cada disparo de $t0$. Quando todas elas tiveram sido consumidas, novos disparos de $t0$ só serão possíveis após um novo disparo de $tcontrol$, no próximo lote.

Além dessas inclusões de lugares e transição de controle, o peso de alguns arcos também foram modificados no modelo da Figura 4. De fato, por exemplo, o peso original *1* do arco que liga o lugar *p4* à transição *t4* no modelo da Figura 2 é alterado para o valor *5* no modelo da Figura 4, significando que, neste último, a transição *t4* somente poderá ser disparada quando tiverem sido produzidos os resultados do processamento de todas as

*5* amostras de um lote (ao passo que, no processamento on-line modelado na Figura 2, tal transição é disparada após concluído o processamento de cada amostra).

## 4. Simulação dos Modelos Propostos

Esta seção simula os modelos produzidos neste trabalho no ambiente *CPN Tools* com o objetivo de detectar o melhor *speedup* do algoritmo de aprendizagem executado com *n* processadores, bem como o ponto de saturação desses processadores (seção 1). A título de esclarecimento, diz-se que o *speedup* obtido é o *melhor* por refletir o desempenho em uma situação ideal em que sempre que uma tarefa precisar ser executada e houver um processador disponível, este último será acionado imediatamente, sem se levar em conta consumo de tempo com, por exemplo, troca de mensagens ou comunicação entre processadores.

Ambos os modelos de treinamento serão simulados a partir de uma base de treinamento contendo 15 amostras, sendo que os lotes usados na simulação do modelo em *mini-batches* terão tamanho 5.

Conforme mencionado anteriormente, cada operação que envolve um processador é associada a uma unidade de tempo igual a *1*, o que significa que o processador (ficha) em questão ficará indisponível para outras operações durante esta unidade de tempo.

### 4.1. Simulação do Modelo de Treinamento On-line

A fim de definir o menor tempo possível de execução do treinamento no modelo on-line, este foi inicialmente simulado em uma situação ideal em que há um conjunto infinito de processadores disponíveis. Para tanto, o arcos que ligam o lugar *proc* às transições foram retirados, o que permite que as transições sejam disparadas independentemente dos processadores.

Na sequência, foram executadas sucessivas simulações do modelo rodando com *n* processadores, sendo que em cada simulação o valor de *n* é gradativamente aumentado (a partir de *1*). A sequência de simulações termina tão logo uma delas atinge o tempo mínimo calculado na simulação com infinitos processadores.

O resultado dessas simulações é apresentado na Figura 5 (b). A partir do gráfico obtido, é possível observar que o tempo de simulação diminui consideravelmente com o aumento do número de processadores, uma vez que, a princípio, mais operações podem ser realizadas em paralelo na medida em que se aumenta o número de processadores.

Observa-se também na Figura que, a partir de *30* processadores, o acréscimo de tal recurso deixará de provocar uma redução do tempo de simulação, o que caracteriza o ponto de saturação de processadores. No caso, os tempos de simulação para *1* e *30* processadores são, respectivamente, *3690* e *422* unidades de tempo.

Além disso, fica claro através do gráfico que a diminuição do tempo de simulação é mais acentuada quando há poucos processadores e mais atenuada a partir de 5 processadores. Há, por exemplo, uma queda de aproximadamente 77% do tempo de simulação (2854 unidades de tempo) de 1 para 5 processadores e apenas 50% de redução no tempo de simulação (414 unidades de tempo) de 5 para 30 processadores.

O gráfico apresentado na Figura 5 (a) mostra o *speedup* calculado para $N$ processadores. Com 2 processadores, o speedup obtido foi de aproximadamente $1,97$, signifi-

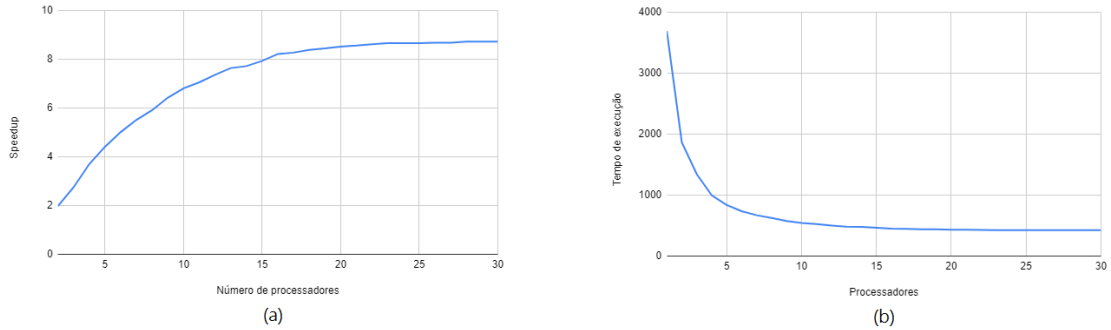


**Figura 5. Gráficos de tempo de simulação e speedup do modelo com treinamento on-line**

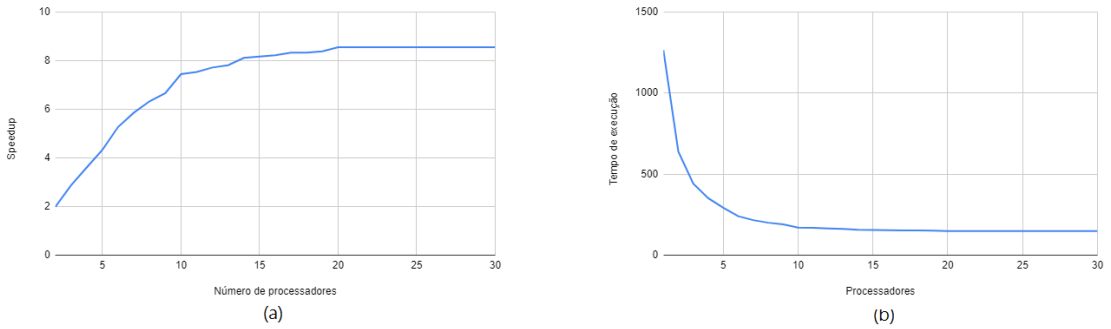


**Figura 6. Gráficos de tempo de simulação e speedup do modelo de treinamento em mini-batches**

cando que o tempo de execução com dois processadores foi $1,97$ vezes mais rápido do que o tempo de execução com apenas um processador. Com 30 processadores, o *speedup* obtido foi de 8,74. Isso significa que o maior ganho possível de velocidade em tempo de execução seria 8,74 vezes em relação à execuções com apenas um processador, já que 30 é o limitante superior para número de processadores que diminuiria o tempo de execução.

## 4.2. Simulação do Modelo de Treinamento *Mini-batches*

O modelo de treinamento em *mini-batches* foi simulado a partir de um conjunto de 15 amostras (assim como o modelo *on-line*). Além disso, o tamanho dos lotes usados no treinamento foi igual a *5*.

Os resultados apresentados no gráfico da Figura 6 (b) são coerentes com a expectativa de uma redução no tempo de simulação, uma vez que o processo da retropropagação e atualização dos pesos no treinamento em *mini-batches* acontece 5 vezes menos, já que os lotes têm tamanho 5.

No caso da simulação em mini-batches, o ponto de saturação de processadores foi atingido com 20 processadores. Observa-se também que o tempo de simulação deste modelo rodando com 1 processador foi de 1266 unidades de tempo, o que representa uma redução de 2424 unidades de tempo (65,7%) em comparação com essa mesma simulação efetuada no modelo *on-line*. Na simulação com 20 processadores, o tempo medido para o modelo *mini-batches* foi de 148 unidades de tempo, o que representa uma redução de 1118 unidades de tempo (88%) com relação ao tempo medido para o mesmo modelo simulado

com um único processador, bem como uma redução de 285 unidades de tempo (66%) em relação ao tempo medido para o modelo *on-line* simulado com 20 processadores.

Assim como na simulação do modelo *on-line*, observa-se também aqui, na simulação do modelo *mini-batches*, que quanto mais processadores são adicionados, menor é a redução produzida no tempo de simulação, gerando curvas com comportamento muito semelhantes para ambos os modelos.

Com relação ao speedup do modelo *mini-batches*, a Figura 6 (a) mostra que as simulações com 2 e 20 processadores foram, respectivamente, 1,98 e 8,55 vezes mais rápidas do que a simulação com um único processador (que representa o processamento serial).

Finalmente, a título de investigação, efetuou-se aqui também a simulação do treinamento em *batches*, ou seja, caso particular do processamento *mini-batches* em que o tamanho do lote é *15*. Neste caso, o desempenho máximo foi obtido para 20 processadores (ponto de saturação). Com 1 processador, o tempo de simulação obtido foi 862 unidades de tempo, valor 77% menor do que no modelo on-line e 32% menor do que no modelo em mini-batches com lote de tamanho 5. Com 20 processadores, o tempo obtido foi 100 unidades de tempo, o que corresponde a uma redução de 762 unidades de tempo (88%) em relação à simulação com 1 processador. Comparando com as simulações efetuadas na mesma situação nos modelos *on-line* e *mini-batches* com tamanho de lote 5, as redução foram respectivamente, de 77% e 32%. Com relação ao speedup, as simulações do modelo em *batches* com 2 e 20 processadores apresentaram valores de 1,98 e 8,62, respectivamente.

## 5. Conclusão

Este trabalho formaliza e implementa o modelo de um Perceptron de Múltiplas Camada (2-4-4-2) proposto por Hanzálek em [Hanzálek 2003] considerando as Redes de Petri Coloridas no ambiente software CPN Tools. Com a utilização de uma versão hierárquica do modelo é possível particularizar cada transição do modelo, definindo, de acordo com as equações fornecidas por Hanzálek, as operações realizadas em cada momento do algoritmo, possibilitando um aperfeiçoamento na precisão do modelo e acarretando em medições de tempo mais precisas durante as simulações. Um lugar de fusão $proc$, que representa a quantidade de processadores que ficam alocados a todas as operações significativas da RNA. O modelo pronto, foi possível realizar as simulações para que pudesse ser feito os cálculos do *speedup*. O modelo é então simulado no ambiente do *CPN Tools* com diferentes números de processadores e com dois tipos de processamentos diferentes, *on-line* e em *mini-batches*. Observa-se uma aceleração no tempo de simulação já esperada conforme o acréscimo do número de processadores. Após obter os tempos de simulação com um e mais processadores, é possível então calcular o *speedup* para cada número de processadores sem a necessidade de implementar o algoritmo em si. A grande vantagem da abordagem proposta é que ela pode ser aplicada a todo tipo de arquitetura, simplesmente adaptando o modelo, sem que seja necessário um estudo analítico completo para cada nova projeto de RNA.

Uma possível abordagem futura é a utilização de técnicas de *process mining* [Sahu et al. 2018] para a detecção de partes que poderão ser paralelizadas no modelo através do levantamento de *event logs* por monitores associados a cada transição do modelo

e disponíveis no *CPN Tools*. Os resultados fornecidos por técnicas de *Process Mining* indicarão em particular os processos paralelos que poderão se beneficiar de um projeto de arquitetura distribuída.

## Referências


Barbosa, V. C. (1996). *An Introduction to Distributed Algorithms*. The MIT Press, Cambridge, Massachusetts.

Borovska, P. and Lazarova, M. (2007). Efficiency of parallel minimax algorithm for game tree search. In *Proceedings of the 2007 international conference on Computer systems and technologies*, pages 1–6.

Brownlee, J. (2019). A gentle introduction to mini-batch gradient descent and how to configure batch size.

Cormen, T. H. (2009). *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts.

Coulouris, G. F. (2012). Distributed systems: concepts and design/george coulouris...[et al.].

Goldsmith, S. F., Aiken, A. S., and Wilkerson, D. S. (2007). Measuring empirical computational complexity. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 395–404.

Hanzálek, Z. (2003). *Parallel Algorithms for Distributed Control-A Petri Net Based Approach*. PhD thesis, Citeseer.

Hassoun, M. H. et al. (1995). *Fundamentals of artificial neural networks*. MIT press.

Hristev, R. (1998). The ann book.

Jensen, K., Christensen, S., Kristensen, L. M., and Westergaard, M. (2018). Cpn tools.

Jensen, K. and Kristensen, L. M. (2009). *Coloured Petri nets: modelling and validation of concurrent systems*. Springer Science & Business Media.

Júnior, C. M. M., Julia, R. M. S., Julia, S., and Silva, L. d. F. (2018). A new approach to evaluate the complexity function of algorithms based on simulations of hierarchical colored petri net models. In *Information Technology-New Generations*, pages 555–564. Springer.

Kon, M. A. and Plaskota, L. (2000). Information complexity of neural networks. *Neural Networks*, 13(3):365–375.

Murata, T. (1989). Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580.

Sahu, M., Chakraborty, R., and Nayak, G. (2018). A task-level parallelism approach for process discovery. *International Journal of Engineering & Technology*, 7(4):2446–52.

Serpen, G. and Gao, Z. (2014). Complexity analysis of multilayer perceptron neural network embedded into a wireless sensor network. *Procedia Computer Science*, 36:192–197.