Universidade Nova de Lisboa

OMNIS CIVITAS CONTRA SE DIVISA NON STABIT

Faculdade de Ciências e Tecnologia

Sistema de Optimização de Amplificadores em Ambiente Distribuído

Relatório de Mestrado em Engenharia Electrotécnica e de Computadores

José Filipe Gonçalves Higino

Nº 22827

2008

Resumo/Abstract

Este trabalho apresenta um ambiente distribuído de optimização de circuitos amplificadores baseado na metodologia "Message Passing Interface" (MPI). O software desenvolvido integra dois componentes: algoritmos genéticos e código de um simulador "open-source", NGSPICE. O trabalho utiliza uma arquitectura cliente-servidor para o processamento de simulações em paralelo / distribuído. Desta forma o desempenho do optimizador aumenta e consegue-se reduzir substancialmente o tempo de optimização dos circuitos. Utilizou-se modelos de simulação BSIM3v3 que garantem resultados mais precisos. Este novo método de optimização possibilita ainda (re)aproveitamento de computadores, e.g. computadores pessoais ou máquinas profissionais topo de gama. A capacidade de processamento total pode ser tanto maior quanto maior o número de máquinas disponíveis e suas capacidades de processamento.

This paper presents a distributed environment for optimization of circuits amplifiers based on the methodology "Message Passing Interface" (MPI). The software developed includes two components: genetic algorithms and code of a "open-source" simulator, NGSPICE. The work uses a client-server architecture for the processing of simulations in parallel / distributed. Thus the performance of the Optimizer increases and is able to reduce substantially the time for optimization of circuits. We used simulation models BSIM3v3 that ensure more accurate results. The new method also allows optimization of (re)use of computers, f. e. personal computers or high-end professional machines. The total processing capacity may be greater the larger the number of machines available and their capacity for processing.

Autor: José Filipe Gonçalves Higino

Orientador: Professor João Goes

Assistente: Professor Rui Tavares

Índice Geral

Índice de Figuras

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

A motivação deste trabalho prende-se com a necessidade de reduzir o tempo de desenvolvimento de novos circuitos e ao mesmo tempo tentar obter resultados mais precisos. Focando o trabalho na etapa da optimização de circuitos, a adopção do paradigma de processamento distribuído/paralelo mostrou ser uma boa forma de reduzir o tempo de optimização. Este aumento de desempenho torna viável a análise de circuitos integrados baseada em simulações no domínio do tempo, com recurso a modelos de transístores mais complexos e mais precisos, e.g. BSIM3v3 [10].

1.2 ENQUADRAMENTO

Na concepção de circuitos electrónicos está implícita uma vasta gama de etapas estratégicas com um objectivo comum entre elas, viabilizar a construção de um circuito integrado,como ilustrado na Figura 1. Esta figura representa uma síntese da concepção de um circuito electrónico: No canto inferior esquerdo da Figura 1, temos representados os primeiros quatro passos da Figura 2, ou seja, "Definição das especificações", a "Escolha da arquitectura", a "Optimização do circuito" e "Verificação por simulação". No canto superior esquerdo da Figura 1, temos representado o quinto passo descrito na Figura 2, o "Desenho das máscaras do circuito (layout)". Por último numa terceira fase indicada pelo desenho no qual convergem duas setas dos desenhos falados anteriormente nesta secção, está representado o sexto passo da Figura 2, a "Extração e Verificação (simulação)". Só depois de validados os objectivos para o circuito é que é possível proceder à sua produção, exemplificada na Figura 1 como uma placa *"wafer"* composta por uma matriz de circuitos impressos.

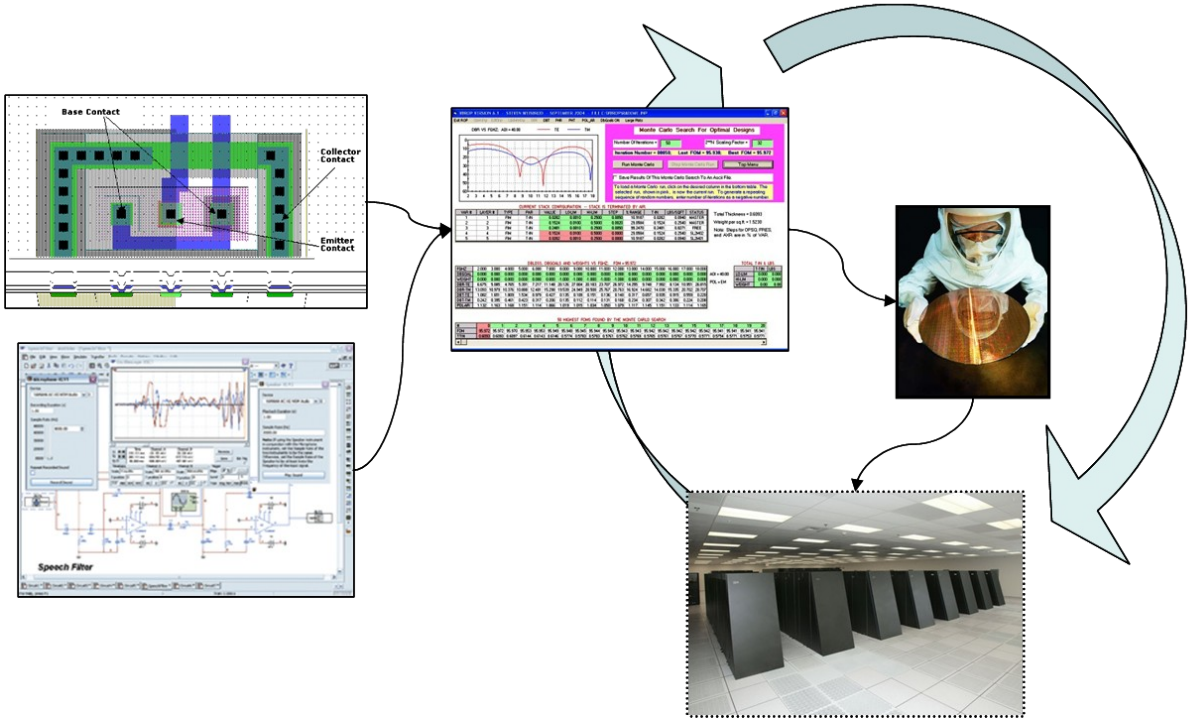

Figura 1: Ciclo de vida do processo de concepção circuitos electrónicos integrados

Depois da produção destes circuitos é possível construir engenhos electrónicos como os computadores por exemplo, que dispostos em configurações especificas dão o lugar a máquinas que servem como ferramentas de processamento às próprias optimizações/simulações/verificações dos circuitos implementados.

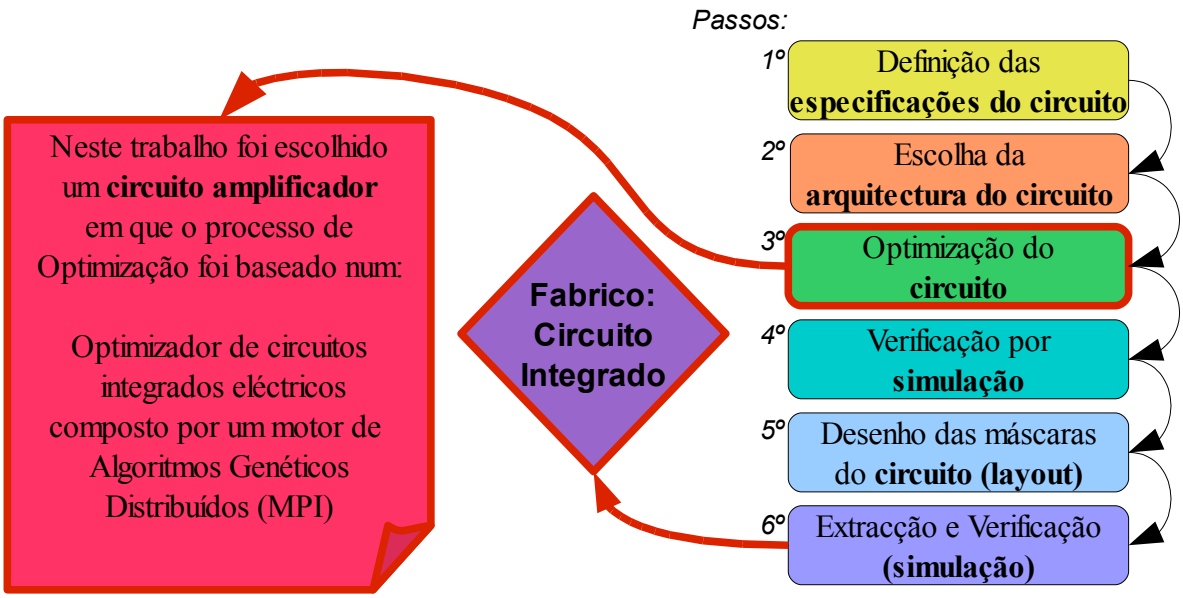

Figura 2: Esquemático do processo de concepção de um circuito electrónico integrado

Entre as muitas tarefas do desenho de circuitos existe a optimização de circuitos electrónicos, sobre a qual incide este trabalho. Esta fase encontra-se entre o plano de arquitectura do circuito e a concepção do desenho do circuito integrado que é posteriormente enviado para fabricação, como indicado no ciclo de vida da Figura 2.

A optimização de um circuito electrónico pode realizar-se de diversas formas, sendo que, maioritariamente todas elas exigem um elevado nível de processamento computacional bem como um nível de repetição elevado, o que prolonga o tempo de espera na concepção de um circuito integrado electrónico.

O processo de concepção de um circuito integrado analógico tem por si só uma evolução histórica muito própria na concepção de circuitos eléctricos. Este processo, demorado e cada vez mais repetitivo à medida que a tecnologia alcança novas metas, tem também como objectivo, inovar na implementação de circuitos eléctricos bem como nos custos associados à sua produção. Nos dias que decorrem a concepção de um circuito integrado analógico passa por diferentes etapas até que este possa assumir um formato físico com utilidade no mundo electrónico, como ilustrado na Figura 2. Começando pela escolha das especificações do circuito integrado, à sua arquitectura, seguido da sua implementação física em materiais específicos, encontra-se uma necessidade de repetição periódica nos processos que envolvem o fabrico de um circuito integrado. Nesta repetição periódica está implícita uma evolução tecnológica em que o circuito deverá ainda sofrer um conjunto de simulações de todos os passos descritos, a fim de validar o seu funcionamento correcto entre todos os passos anteriormente descritos para que, finalmente se proceda ao seu fabrico.

Para tal, esta tese de mestrado apresenta uma forma, embora mais complexa, mais eficiente em termos de tempo de computação para optimizar circuitos amplificadores analógicos integrados através de processamento distribuído/paralelo. Através desta técnica de processamento é possível diminuir o tempo de processamento, utilizando modelos de transístores mais complexos e mais precisos.

Deste trabalho resultou a publicação [7], com revisão internacional, onde se demonstra um caso concreto de aplicação do trabalho desenvolvido, e se valida, internacionalmente, o seu conteúdo.

1.3 Descrição e Objectivos

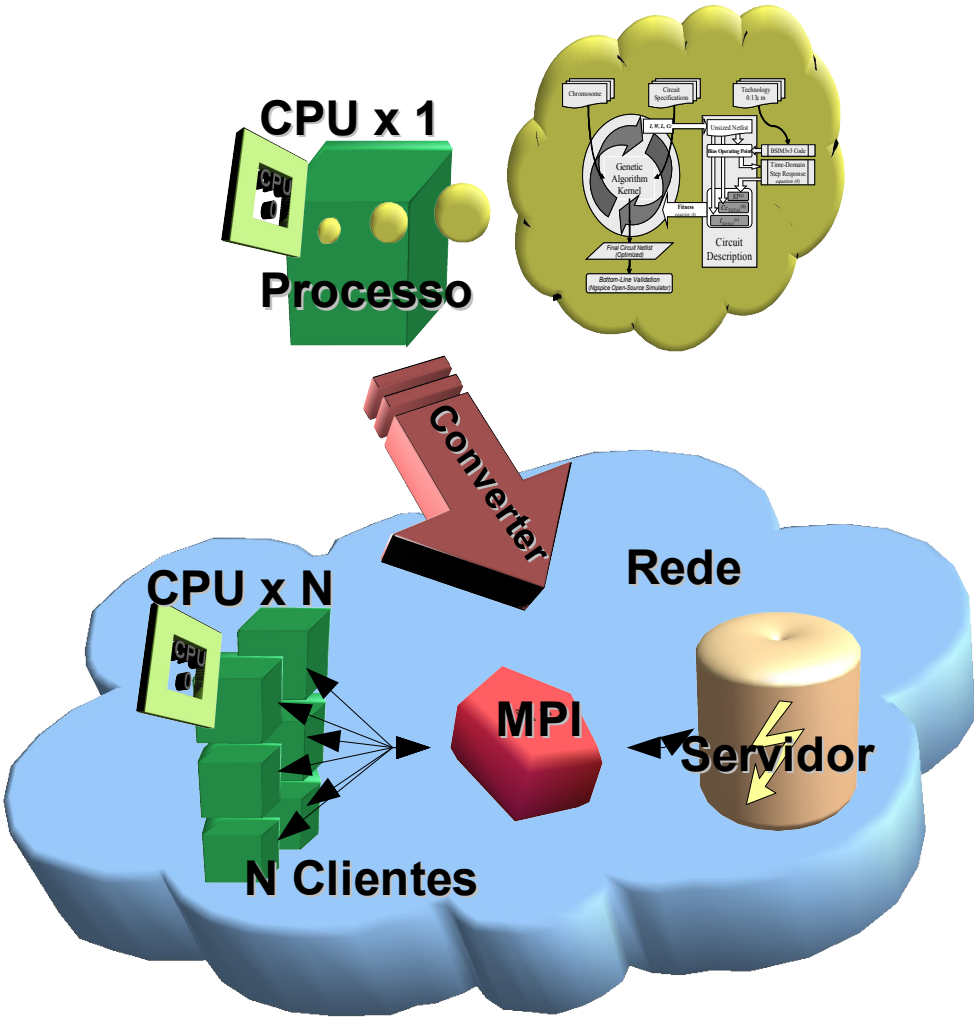

Figura 3: Ilustração do Objectivo da Tese

Este trabalho tem como objectivo demonstrar que é possível implementar um optimizador de circuitos electrónicos baseado em processamento paralelo e algoritmos genéticos, alcançando-se resultados num menor período de tempo. A integração do paradigma de processamento distribuído permite passar de apenas uma máquina física, e um único processo, para um ambiente de múltiplas máquinas físicas, como indica a Figura 3. Assim, todo o processamento decorre num menor período de tempo e em recursos, potencialmente, mais baratos.

Com o principal propósito de diminuir o tempo necessário para optimizar um circuito electrónico utilizando a tecnologia existente, como por exemplo os Computadores Pessoais mais vulgares dos dias de hoje, os chamados computadores pessoais. Através destes Computadores Pessoais e um motor de algoritmos genéticos, paralelizar o processamento das simulações no processo de optimização, como exemplificado na Figura 3, reduzindo tanto o tempo de optimização como os custos associados ao decorrer de todo o processo. Deste modo passa a ser possível a utilização de múltiplas máquinas, "N Clientes" como indica a Figura 3, diferentes entre si, de forma a viabilizar a reutilização de recursos que possam já não ter utilidade comum.

Não obstante, o facto dos algoritmos genéticos actuarem como mecanismos independentes de avaliação de cada indivíduo na simulação do circuito, criam um ambiente propício ao paralelismo e torna-se vantajosa a sua utilização face à implementação local. No entanto embora se gaste mais tempo na comunicação entre os processos de paralelismo, esta não é significativa comparada com o tempo gasto para a avaliação/simulação de cada indivíduo do circuito, acabando também por fornecer a vantagem do aumento de recursos, permitindo efectuar optimizações com um alargado espaço de pesquisa, conduzindo a optimização do circuito em ambiente distribuído a um melhor conjunto de resultados.

1.4 Estrutura da Tese

O trabalho é composto por oito capítulos dos quais sete serão sumariados nos seguintes pontos. Na primeira e actual secção (Secção I) é apresentado o principal contributo da tese em 1.1 – *Motivação*, é ainda descrita uma breve introdução que nos contextualiza no assunto do trabalho e por fim são apresentados os objectivos propostos. Nas restantes secções serão discutidos os respectivos assuntos:

Secção II – Ferramentas Integradas: Relata as características das tecnologias existentes e utilizadas no âmbito deste trabalho, evidenciando vantagens e desvantagens face a outras tecnologias.

Secção III – Trabalho Realizado e Opções Tomadas: Alterações e decisões com descrição ao pormenor sobre o estado inicial da aplicação, os objectivos delineados e como foram planeadas as acções de modificação.

Secção IV – Testes e Resultados Obtidos: Demonstração e descrição dos resultados obtidos nos testes realizados. Estão catalogadas todas versões dos componentes com que foram testados.

Secção V – Conclusões: São validados os principais contributos desta tese e sintetizados os objectivos cumpridos através da discussão dos resultados obtidos

Secção VI – Trabalho Futuro: Ideias idealizadas como possíveis componentes a adicionar à aplicação desenvolvida e que não foram implementadas na realização deste trabalho.

Secção VII – Anexos e Bibliografia: Anexos e bibliografias de assuntos discutidos na tese.

Secção VIII – Referências: Referências de documentos usados para explicar assuntos na tese.

2 Ferramentas Integradas

2.1 Contexto

Existem hoje em dia existem diversos ambientes de optimização de circuitos. Alguns são baseado em modelos de transístores simples que permitem resultados num curto espaço de tempo, mas sem grande precisão. Outros, baseados em modelos mas complexos e, consequentemente mais precisos, tornam-se lentos..

Neste trabalho é demonstrada uma metodologia para optimizar circuitos através da integração de algoritmos genéticos e simulações paralelas recorrendo ao ambiente MPI ("Messaging Passing Interface"). O trabalho consiste na evolução de uma plataforma, anteriormente desenvolvida e baseada em processamento sequencial. Os desenvolvimentos apresentados permitiram a integração do código de um optimizador baseado em algoritmos genéticos, onde o processamento de simulações passou a ser distribuído, e realizado em paralelo. A comunicação entre os diversos processos distribuídos é baseado no paradigma MPI . As simulações continuam a ser suportadas pelo simulador NGSPICE, que incorpora modelos de transístores mais complexos e, consequentemente, mais precisos, BSIM3v3. A plataforma desenvolvida não necessita de recorrer, obrigatoriamente, a recursos caros. É possível utilizar computadores comuns para escalonar a capacidade de processamento na optimização de circuitos, oferecendo uma alternativa mais económica e potencialmente mais rápida que outros aplicativos profissionais comerciais.

Em casos de necessidade extrema é possível aplicar o trabalho proposto em *"hardware"* empresarial e de alto desempenho. Nesses casos o desempenho é igualmente alto e comparado com aplicativos profissionais, muito superior por beneficiar da possibilidade de correr sobre ambientes *"cluster"*, conjuntos de múltiplas máquinas ligadas entre si Sabendo que existem configurações de máquinas em *"cluster"* que ultrapassam os 100 e 200 processadores lógicos [8], podemos prever algumas das potencialidades que um ambiente de processamento em paralelo poderá ter numa aplicação de um optimizador de processamento em paralelo.

Considerando agora capacidades de desempenho muito superiores como por exemplo os *"clusters"* de "computação de alto desempenho" (HPC, "High Performance Computing")

mais rápidos do mundo da empresa IBM ("International Business Machines") e fazendo uma simples pesquisa na *Internet* [8], obtém-se que, o maior *"cluster"* actual possui cerca de 212992 processadores , "BlueGene" (da IBM), ilustrado na Figura 4. Para estes casos a configuração apresentada não seria capaz de aproveitar todo o processamento da máquina mas nas condições actuais desempenharia muito melhor que muitas aplicações devido à sua capacidade de escalonamento conseguida pela utilização do MPI. Seriam possíveis optimizações com um número total de iterações na ordem dos milhões ou mesmo biliões de simulações, processáveis em apenas umas horas.

Figura 4: "BlueGene" em fase final de instalação

2.2 Optimização: algoritmos genéticos

O aumento da complexidade do processo de optimização de um determinado circuito, resulta maioritariamente do aumento do número de variáveis a optimizar nesse mesmo circuito. Estas variáveis podem ainda ser alteradas a fim de se atingir os objectivos pretendidos. As variáveis podem ser por exemplo: as dimensões dos transístores (W e L), as correntes nos vários ramos do circuito, as capacidades de compensação, etc. Contudo, muitos dos circuitos de hoje em dia não sofrem alterações desta natureza. Na maioria dos casos e conforme evidenciado em [5], os circuitos sofrem um processo de evolução na tecnologia, geração após geração em que são inicialmente concebidos/implementados. Nas posteriores gerações do mesmo circuito no seu processo evolutivo, a tensão de alimentação dos circuitos tende a baixar gradualmente e novas adaptações/optimizações terão de ser feitas aos dimensionamentos dos circuitos já implementados para que estes se mantenham eficientes às novas tensões. Na prática o circuito desempenha as mesmas funções mas apresenta um tamanho físico mais reduzido e um consumo potencialmente menor.

Para eliminar esta barreira de complexidade momentânea, é necessário utilizar uma técnica que no seu processo de optimização não dependa da construção complexa do circuito a optimizar quando este sofre uma alteração de optimização para uma nova tecnologia a usar. Para tal, é possível a utilização de um método de optimização por tentativa como o Algoritmo Genético, evidenciado em [1]. Deste modo é possível simplificar o código de optimização tornando-o num processo cíclico mais fluente, ou mesmo baseado na arquitectura de um Kernel genérico suportado por algoritmos genéticos [5], permitindo ainda a possibilidade de utilizar mais recursos para adquirir melhor desempenho, através do processamento distribuído/paralelo.

Os algoritmos genéticos, têm como principal função/característica, a aprendizagem. Através de uma função de classificação específica de cada circuito, chamada de *"fitness"*, são calculados valores resultantes da análise do circuito. É possível juntar as propriedades de um algoritmo genético com as propriedades de um motor que executa o processamento das simulações. Conjuntamente estes dois componentes, optimizam um circuito através de indivíduos que evoluem consoante o valor de *"fitness"* de cada simulação. Os indivíduos são

um conjunto de atributos (e este de genes), gerados aleatoriamente dentro de um intervalo configurável (descrito num ficheiro designado por cromossoma) de valores que configuram um circuito para que possa ser simulado, como em [1] e [5]. Ao conjunto de indivíduos, chamamos população, em que cada indivíduo desta população origina um diferente resultado na simulação do circuito a optimizar. Através do resultado dessa simulação é calculado um valor de *"fitness"*, que serve de elemento comparativo na determinação do melhor indivíduo, classificando-o relativamente aos objectivos propostos para o circuito. Quanto mais alto for o valor de *"fitness"*, mais o circuito se aproxima dos objectivos propostos. Após todos os elementos analisados são realizadas operações de selecção/cruzamento/mutação, como descrito em [5], a fim de se gerar um nova população para que de novo seja simulado o circuito com os atributos de cada indivíduo.

As selecções baseiam-se numa percentagem configurável de indivíduos copiados directamente para a próxima população. Esta cópia poderá ser feita ainda de duas formas: forma roleta e forma ordenada. No primeiro caso cada indivíduo terá associado a si mesmo um determinado valor de *"fitness"* no qual está directamente associado uma probabilidade de ser copiado directamente para a nova população. A segunda forma baseia-se numa simples ordenação de todos os indivíduos e é então copiada a percentagem escolhida dos primeiros indivíduos com melhor *"fitness"*.

As operações de *"cross-over"* ou cruzamentos são baseadas numa variável *"pc"*, indicadora da taxa de *"cross-over"* em que determinada parte do valor de dois indivíduos a informação poderá ser trocada mediante a sua igualdade ou não entre os dois valores. Estas operações são aplicadas nos restantes indivíduos após a selecção.

Finalmente as operações de mutação são aplicadas a todos os elementos da população e a técnica consiste na probabilidade, *"pm"*, indicadora da taxa de mutação que um atributo da informação do indivíduo tem.

2.3 SIMULAÇÃO: NGSPICE

NGSPICE [3] é uma ferramenta "Open Source" de Design Electrónico Automático, composto por três pacotes de *"software",* o "Spice3", o "Cider" e o "Xspice". O NGSPICE é disponibilizado e desenvolvido pela equipa do projecto "gEDA" ("Electronic Design Automation tools") em conformidade com a "GNU GPL" ("GNU", «GNU's Not Unix», "General Public License"). Na sua composição o "Spice3" é um simulador de circuitos integrados muito popular, utilizado há mais de 30 anos, que uniformizou a simulação de circuitos electrónicos em geral. O "Cider" é uma extensão da versão do "Spice3f5" que permite simulações com maior detalhe, vocacionadas para circuitos críticos. O "Xspice" é outra extensão do "Spice3C1" que permite a modelação da simulação de circuitos digitais através de um algoritmo de eventos.

Foi escolhido por ser um código "Open Source" e teve como propósito adaptar as necessidades do trabalho (facilitando a sua integração com os restantes componentes) a fim de possibilitar a simulação de circuitos e consequentemente a respectiva optimização, em ambiente distribuído. O facto de já se encontrar desenvolvido e com documentação simplificou o trabalho da tese, facilitando a implementação dos objectivos propostos.

Esta ferramenta inclui os modelos de transístores BSIM3v3 [10] que permitem uma análise de circuitos mais precisa, embora seja necessário mais tempo de processamento na análise dos mesmos. Deste modo nasce uma necessidade de recorrer ao processamento distribuído com o objectivo de melhorar o desempenho e reduzir o tempo necessário para realizar uma optimização.

Na aplicação entregue o NGSPICE é utilizado para simular o circuito gerado a partir dos valores de um individuo que configuram o circuito a ser simulado. Depois de todos os indivíduos de uma população completarem as respectivas simulações, a população é regerada através de regras impostas pelo comportamento matemático dos algoritmos genéticos. Durante o restante comportamento o simulador processa um indivíduo de cada vez até que se esgotem todos os indivíduos da população de uma geração.

2.4 Processamento em Paralelo

O processamento em paralelo é uma técnica conhecida pela capacidade de uma unidade física de processamento executar e processar dois processos em simultâneo. Contudo, é ainda possível dividir a actividade lógica de processamento em dois tipos de processamentos virtuais, o *Processamento em Paralelo Local* e o *Processamento em Paralelo Distribuído*.

Esta técnica permite efectuar em dois ou mais diferentes lugares físicos e no mesmo espaço temporal, processamentos de dados com objectivos comuns entre si. Estes processamentos poderão dar lugar a simulações executadas em nós/computadores remotos ligados por um meio de comunicação produzindo no final um conjunto de resultados com características semelhantes. No final é possível recolher deste conjunto de resultados o melhor deles.

Com esta arquitectura é possível diminuir o tempo de processamento total até obter o melhor resultado possível, sendo que em vez do processamento tomar uma linha contínua de evolução, enverga agora por múltiplos caminhos paralelos que poderão assumir comprimentos de processamento dinâmicos, dependendo da sua capacidade de processamento. Desta forma o processamento em paralelo adiciona uma nova dimensão no desempenho e nas capacidades de processamento aplicacionais.

2.4.1 Processamento em Paralelo Local

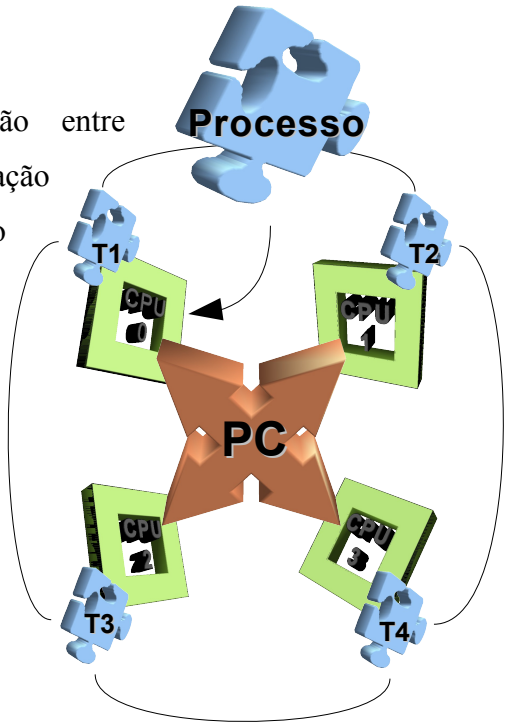

Figura 5: Processamento Local

Distingue-se pelo facto da comunicação entre processos ser feita na mesma camada de comunicação que das unidades físicas de processamento. Isto é, o processamento realiza-se dentro do mesmo sistema operativo que engloba um conjunto de *"hardware"* próprio do mesmo. Este conjunto de *"hardware"* pode possuir uma ou mais unidades de processamento central em que o processamento realiza-se entre CPU's ("Central Process Unit") ou processadores e não entre sistemas operativos distintos.

Como é possível verificar na Figura 5, o processo principal, tipicamente lançado no CPU 0, lança uma ou mais *"threads"* (T1, T2, T3 e T4 da Figura 5), das quais poderão ligar-se a diferentes CPU's, ao CPU0, CPU1, CPU2 ou ao CPU3, conforme as definições do sistema operativo. Uma diferença importante que distingue o tipo de processamento da presente secção do da próxima secção 2.4.2 - *Processamento em Paralelo Distribuído*, é a capacidade de cada *"thread"* poder comunicar directamente com outra sem que ter de atravessar um mecanismo muito lento e altas latências de comunicação em relação a um processador, como por exemplo redes IP, redes de fibra óptica, discos partilhados concorrentes, etc. No processamento local a velocidade de comunicação entre *"threads"* é mais rápida que em maior parte das comunicações entre processos. Quando estas se fazem dentro do mesmo processador e respectiva cache, tornam-se as mais rápidas possíveis.

2.4.2 Processamento em Paralelo Distribuído

Pode englobar o conceito de processamento em paralelo local mas distingue-se pelo facto de fazer-se também entre várias máquinas com diferentes sistemas operativos. Tal interoperacionalidade apenas é possível através de uma camada de transporte adicional mais lenta e um ambiente configurador, como o MPI por exemplo que administre a comunicação entre as máquinas, como computadores pessoais por exemplo.

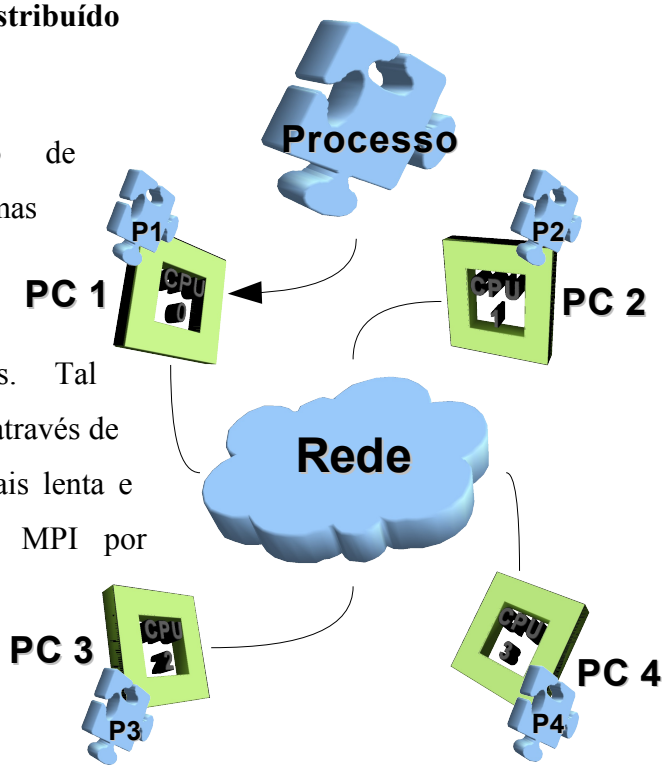

Figura 6: Processamento Distribuído

Através da Figura 6 é possível verificar que de modo geral no processamento em paralelo distribuído o processo inicial P1, o primeiro a ser executado, começa por lançar a primeira ou o primeiro conjunto de *"threads"*. Estas *"threads"* estão sobre controlo do processo P1 no primeiro nó/máquina, constituído-se assim o nó de arranque ou muitas vezes chamado de nó Servidor. Este nó poderá então lançar através de uma rede de comunicação, tipicamente ponto a ponto, os restantes processos P2, P3 e P4 como ilustrado na Figura 6. Estes processos arrancam nos nós configurados e estes por sua vez lançam *"threads"* se o processamento paralelo local for possível. Neste caso os processos distribuídos P1, P2, P3 e P4 não podem comunicar a uma velocidade tão alta como a velocidade de comunicação entre *"threads"* do mesmo processo. Esta diferença na velocidade de comunicação constitui uma desvantagem para o processamento em paralelo distribuído, sendo menos eficiente por unidade de processamento lógico. Contudo, este tipo de processamento possui uma capacidade enorme de processamento porque pode expandir o processamento para múltiplas máquinas físicas, ao contrário do processamento em paralelo local, onde a sua capacidade de processamento se restringe às potencialidades de uma única máquina física.

2.4.2.1 Ambiente MPI (Activo ou Passivo)

"Messaging Passing Interface" é um ambiente "Open Source" desenvolvido para permitir a comunicação básica entre máquinas distintas heterogéneas ou homogéneas. Este ambiente disponibiliza um nível abstracto para o suporte de implementações que têm como objectivo, expandir o processamento de uma aplicação até mais do que uma máquina física. Através destes ambientes é possível estender a memória e a capacidade de processamento de uma aplicação através de redes e máquinas tecnologicamente diferentes, fornecendo o melhor desempenho para cada uma das tecnologias de processador ou rede, deixando a responsabilidade de gestão de cada um dos componentes do conjunto heterogéneo para o ambiente MPI, [6].

O MPI é frequentemente comparado com o PVM, "Parallel Virtual Machine" que é um ambiente distribuído e sistema de troca de mensagens muito popular desenvolvido em 1989. Foi também um dos sistemas que motivou a necessidade de uma norma para a troca de mensagens em sistemas de processamento paralelo. Modelos de programação em "Threads" que utilizam memória partilhada (como Pthreads e OpenMP) e a programação de troca de mensagens (MPI / PVM) podem ser considerados como um complemento na abordagem de programação. Ocasionalmente são usados em conjunto quando as aplicações beneficiam desta arquitectura, como por exemplo, em grandes servidores de múltiplos nós com memória partilhada.

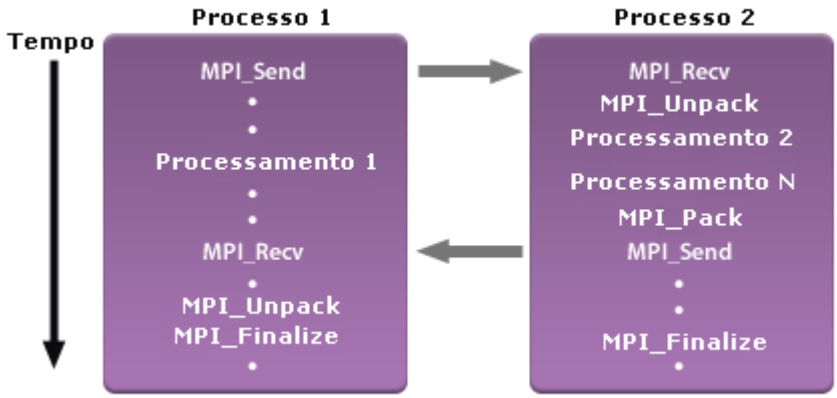

Figura 7: Funções típicas de MPI

Escritas em linguagem de programação C, as distribuições de MPI como por exemplo o "OpenMPI" indicado em [4], fornecem um conjunto de funções, ilustradas na Figura 7, que nos permitem estender a implementação do processamento útil de uma aplicação a mais do que um computador/máquina. Na Figura 7 estão detalhadas quais as funções base de uma implementação MPI, usadas para paralelizar uma aplicação simples. A descrição do funcionamento básico passa pela troca da informação a processar através da função *MPI_Send(...)* e recebida através da função *MPI_Recv(...)*. Depois de desempacotar a informação através da função *MPI_Unpack(...)* é possível começar a processar em ambos os processos, como indicado na Figura 7. No final do processamento poderão ser empacotados eventuais resultados e enviados para o processo principal, neste caso o "Processo 1" indicado na Figura 7, através das funções *MPI_Pack(...)* e *MPI_Send(...)*, respectivamente. Quando todo o processamento acaba e não existem mais dados a serem processados, o ambiente é abandonado através da função *MPI_Finalize()*. As funções mencionadas neste parágrafo e suas implementações na aplicação desenvolvida, serão ainda sumariadas na secção 3.4 – *Alterações Implementadas* e detalhadas na secção 3.4.1.3 – *Camada de Transporte (Mensagens MPI)*. No entanto, todas as aplicações implementadas através desta interface terão obrigatoriamente que respeitar um conjunto de regras de implementação impostas pelo *"standard"* do MPI e consequentemente possuir um tempo de implementação superior ao de uma aplicação simples de processamento local que não utilize MPI. Consequentemente a complexidade na estrutura de comunicação interna da aplicação aumenta porque passa a existir código adicional, do MPI. Contudo, este aumento na complexidade do código e a possibilidade de poder estender o processamento da aplicação a mais recursos físicos, independentes entre si (máquinas diferentes) permite alcançar capacidades de processamento muito superiores às de uma aplicação que não usa as vantagens do processamento em paralelo utilizando código MPI para a troca de mensagens entre processos As aplicações implementadas com código MPI possuem a vantagem de poderem assumir dois tipos de comunicação: a comunicação passiva ou bloqueante que se baseia no envio de um pedido e na espera por tempo indefinido da resposta; e a comunicação activa ou não-bloqueante em que são enviados um ou mais pedidos e depois recebidas as respostas não necessáriamente de forma ordenada. Estes dois modos de comunicação tornam mais flexíveis as implementações de arquitecturas e permitem minimizar complexidades de estruturas de comunicação quando não são necessárias. Distinguindo os dois tipos de comunicação, temos que: quando a ligação de todos os nós MPI no mesmo ambiente de trabalho e a comunicação entre eles é um

requisito ao funcionamento da aplicação, trata-se de uma comunicação passiva; quando parte dos nós podem abandonar, juntarem-se ou cessarem simplesmente a comunicação com o ambiente, embora não abandonando o ambiente e ao mesmo tempo manterem a sincronização de todas as suas acções de modo a poderem restabelecerem a comunicação de novo, estamos neste caso perante uma comunicação activa em que a ausência de comunicação não bloqueia os nós, deixando-os executar outras tarefas. A implementação da comunicação passiva para estruturas de comunicação pequenas é tendencialmente mais simples de implementar e produz melhores resultados. No entanto a implementação da comunicação não bloqueante tem cada vez mais um peso importante nos nossos dias e a sua utilização é cada vez mais um requisito das aplicações como descrito em [4].

2.4.2.2 Ambiente BOINC (Passivo)

"Berkeley Open Infrastructure for Network Computing" ("BOINC", Figura 8) é uma plataforma desenvolvida para suportar projectos de voluntariado em processamento computacional. Está vocacionada para quaisquer aplicações distribuídas que exijam um grande tempo computacional e/ou uma capacidade de armazenamento elevada. Para além de um suporte vasto, desenvolvido e aperfeiçoado pela equipa da "BOINC", este ambiente dispõe ainda de um variado leque de "features", tais como, segurança, redundância, administração remota, compatibilidade para diferentes arquitecturas, disponibilidade do código fonte, entre muitas outras.

Figura 8: Ícone da plataforma "BOINC"

Como princípio geral, o seu funcionamento actua de modo passivo começando por ser gerada a informação a processar e os objectivos a atingir. De seguida, o pacote de informação é fragmentado em blocos de dados com tamanhos publicamente aceitáveis para os requisitos médios ou comuns de processamento das máquinas do voluntariado em que se requer um estudo do tempo em que se processa todo o projecto de forma a conseguir-se a melhor optimização possível. Estes pacotes são então enviados para os clientes e processados. No retorno de cada pacote são realizados testes de validação que determinam se o pacote foi processado com sucesso, se a informação resultante é útil ou não e ainda se, no caso do retorno do pacote chegar fora do

prazo permitido, registar e proceder de forma a possibilitar reenvio de novos pacotes de dados respeitantes ao bloco em teste e seguidamente do retorno de um número configurado de resultados que viabilizam se a resposta pode ser compreendida como válida ou não. Depois do processamento de todos os pacotes são validados os resultados e é gerado um resumo para que sejam então tiradas as conclusões necessárias através de intervenção humana.

No entanto este sistema não trata do código que será implementado na aplicação distribuída ou ainda computação pública, apenas fornece as funcionalidades para administrar vários ambientes de computação distribuída como por exemplo o projecto SETI@Home, Figura 9 (*"Search for Extra-Terrestrial Intelligence"*, *"@Home"*, em casa) e referido em [2].

Figura 9: Ícone de um Projecto "BOINC", chamado "SETI@HOME"

Apesar da extrema facilidade de implementação e utilização para o utilizador final, a estrutura implementada é bastante consistente e própria, dificultando ou mesmo impossibilitando a sua funcionalidade em aplicações cujo o objectivo principal é datar uma linha temporal de execução onde é facilmente conhecido o fim do processo. Esta dificuldade associada a requisitos de desempenho mais acentuados ou outras quaisquer implicações que necessitem de um elevado nível de modificação aplicacional tornam este ambiente menos vantajoso em aplicações como a que foi desenvolvida neste trabalho.

A principal diferença resume-se ao facto de numa implementação do estilo "BOINC" não se conhecer com alguma exactidão quando é que finalizará o processamento total, podendo demorar dias na recepção dos últimos resultados do processamento. Enquanto que na implementação de MPI adoptada, é possível estimar com a precisão de alguns minutos o termo do processamento total.

No entanto para o estilo "BOINC" é préviamente definido um tempo de expiração para os resultados, permitindo que o processamento não fique dependente de clientes que nunca mais completem o processamento ou demorem muito mais tempo a completar o processamento da informação que lhes foi requisitada. Na implementação MPI esta capacidade de não ficar indefinidamente à espera do retorno de um resultado terá de ser programada. Na ausência de programação que previna este acontecimento, a aplicação poderá bloquear por tempo indefinido.

3 Trabalho Realizado e Opções Tomadas

3.1 Delimitação dos Objectivos

Para melhor estruturar e clarificar os objectivos a atingir procedeu-se a uma divisão dos objectivos em dois grupos diferentes. Nestes dois grupos foram evidenciados os objectivos de requisito técnico, relativos a alterações no código, como Objectivos Estruturais e os objectivos de conceito semântico como Objectivos Funcionais.

3.1.1 Objectivos Funcionais

Os objectivos funcionais delimitados visam estabelecer metas semânticas que suportam a tese apresentada. Os objectivos a atingir são:

- Possibilidade de optimizar circuitos mais complexos.
- Diminuir abruptamente o tempo de uma optimização.
- Melhorar as possibilidades de obter um melhor resultado, em tempo útil.
- Alcançar melhores resultados e em menos tempo, comparativamente a um sistema que utiliza um simulador, sem processamento paralelo.

3.1.2 Objectivos Estruturais

Os objectivos estruturais estabelecem as metas de implementação no código aplicacional de forma a comprovar a tese apresentada. Os objectivos a atingir são:

- Expandir a memória disponível para a aplicação, usando recursos distribuídos.
- Processar simulações em máquinas diferentes em paralelo.
- Optimizar código antigo.
- Aumentar a capacidade de optimizar com populações maiores dividindo-a em populações menores para processamento em ambiente distribuído.

3.2 Funcionamento Geral

A aplicação anteriormente desenvolvida, integra dois componentes: o simulador "NGSPICE" e os "algoritmos genéticos", como se ilustra na Figura 10. O simulador é baseado em código "Open Source" e inclui os modelos BSIM3v3. O outro componente, o núcleo de algoritmos genéticos, foi desenvolvido por ex-alunos da UNL-FCT (Universidade Nova de Lisboa – Faculdade de Ciências e Tecnologia), e portanto o seu código também está acessível. Na Figura 10 está também representado o terceiro elemento de software, o MPI, que permite a evolução da plataforma para englobar processamento distribuído. O MPI é baseado na ideia de "Open Source".. O código desenvolvido no âmbito deste trabalho

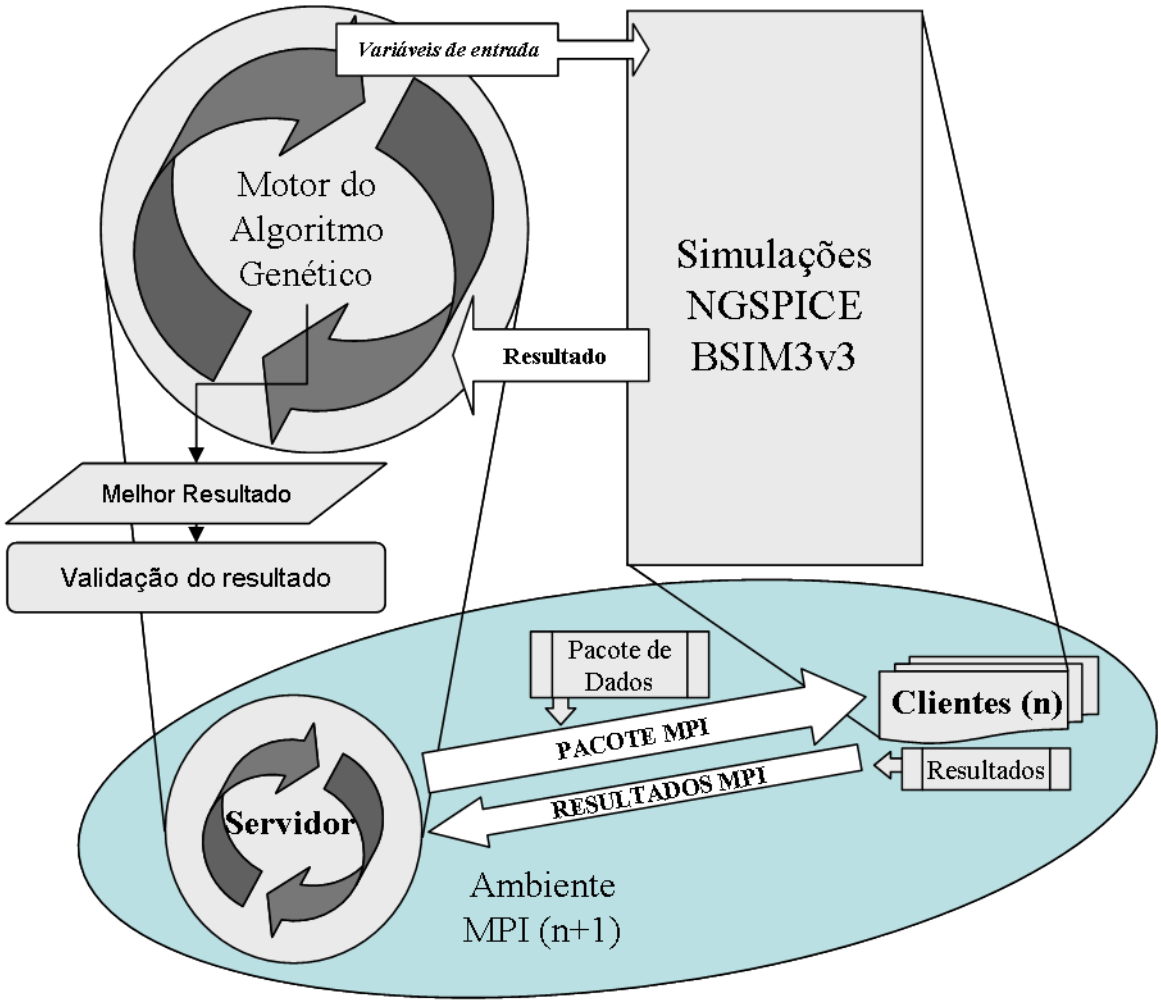

Figura 10: Integração dos componentes, Algoritmos Genéticos, NGSPICE e MPI

permitiu implementar um optimizador de circuitos integrados electrónicos com processamento distribuído

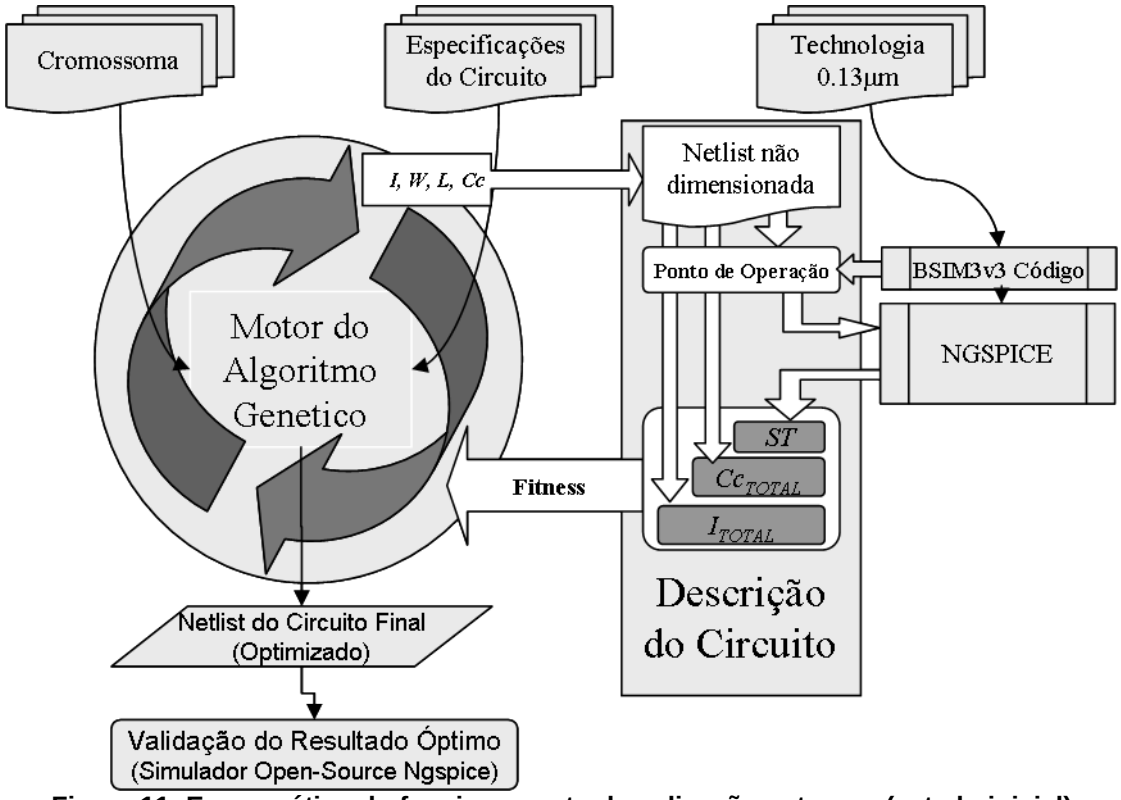

Figura 11: Esquemático do funcionamento da aplicação entregue (estado inicial)

Para melhor compreender o funcionamento da aplicação entregue e acompanhando o esquemático de funcionamento ilustrado na Figura 11 bem como outros detalhes mais complexos e presentes no conceito de processamento distribuído/paralelo, foi necessário averiguar como se desenrola a estrutura do código da aplicação. O código produzido é essencialmente vocacionado para ambientes de um processador lógico porque a aplicação é constituída por um único processo. Este processo no arranque inicial da aplicação inicializa um conjunto de variáveis e componentes. De seguida segue-se um conjunto de processos cíclicos que na sua conclusão disponibilizam um conjunto de resultados numéricos.

Detalhando o processo no parágrafo acima temos que, na fase inicial da aplicação são criadas a maior parte das variáveis que servem de canal de comunicação para toda a aplicação. Estas variáveis são inicializadas com valores predefinidos no código da aplicação ou importadas de um ficheiro de configuração. De seguida são inicializados os ambientes de simulação da NGSPICE e é criada uma população baseada nos máximos e nos mínimos de cada atributo. Estes máximos e mínimos estão especificados num elemento chamado de cromossoma, definido por intermédio de um ficheiro cujo os seus valores serão

posteriormente importados para dentro da aplicação. Cada indivíduo configura um possível funcionamento do circuito "teste" a ser optimizado, sendo que, depois de simulado através do NGSPICE, retornará um valor de *"fitness"*, calculado através de um conjunto de resultados disponibilizados pelo NGSPICE. Este *"fitness"* vai servir de elemento comparativo entre todas as restantes simulações realizadas. Depois de todos os indivíduos esgotados nas configurações/funcionamentos do circuito "teste" e suas respectivas simulações é realizada uma geração, definida por um conjunto de processos configuráveis e implementados pelos algoritmos genéticos. Estes últimos condicionam a evolução da população existente com base num conjunto de técnicas responsáveis pela beneficiação do elemento simulado que estiver mais perto dos objectivos óptimos do circuito. Posteriormente ao processo de geração é novamente realizada a simulação do circuito com todos os indivíduos dessa nova população e realizada uma outra geração, sucessivamente até que se esgote o número de gerações configurado. No final da última geração o melhor indivíduo que configura o circuito é divulgado juntamente com os valores das variáveis que foram determinadas e usadas na simulação do circuito.

3.3 Plano de Modificação

Para alcançar os objectivos delimitados traçou-se um plano de modificação que visa definir e estruturar quais e onde serão realizadas alterações no código da aplicação existente para que esta possa cumprir tanto os objectivos funcionais como os estruturais.

3.3.1 Dinâmica a Implementar

No processo de implementação é necessário definir quais as modificações a fazer. Sendo um dos objectivos, a criação de um ambiente aplicacional distribuído, é necessário escolher uma arquitectura de comunicação que se adapte ao fluxo de dados da aplicação mas que ao mesmo tempo seja simples de implementar e organizar estruturalmente em código aplicacional, facilitando o processo de adaptação ao código já existente.

3.3.1.1 Arquitectura Escolhida

Sendo a aplicação inicial escrita em código C e sabendo da secção 3.2 - *Funcionamento Geral*, que se trata de uma aplicação preparada para tirar partido apenas de um processador lógico, escolheu-se a plataforma de comunicação MPI, também escrita em código C e de muito fácil integração do código com o do resto da aplicação. A escolha da arquitectura tem também como objectivo o de suportar a estrutura que irá paralelizar as simulações e no qual o código exige mais processamento.

Focando a simplicidade como atributo principal da implementação de uma arquitectura de comunicação como o MPI, optou-se pela estrutura do tipo cliente e servidor, ilustrada na Figura 12. Nesta arquitectura de comunicação classificou-se de nós Cliente, os processos que recebem informação complexa para ser processada e de nó Servidor o processo que envia a informação a ser processada e recebe os resultados desses processamentos para posterior análise/conclusão. Numa camada presente entre estes dois últimos componentes, existem mensagens de comunicação inseridas num meio do tipo MPI no qual foi classificando de camada transporte, representado pelas duas "setas" em sentidos contrários no centro da Figura 12. Esta camada trata também de todas as comunicações entre o código MPI e o código não MPI da aplicação como por exemplo, processos de controlo das mensagens. O código MPI permite o transporte de mensagens (números inteiros, texto, vectores, etc) entre

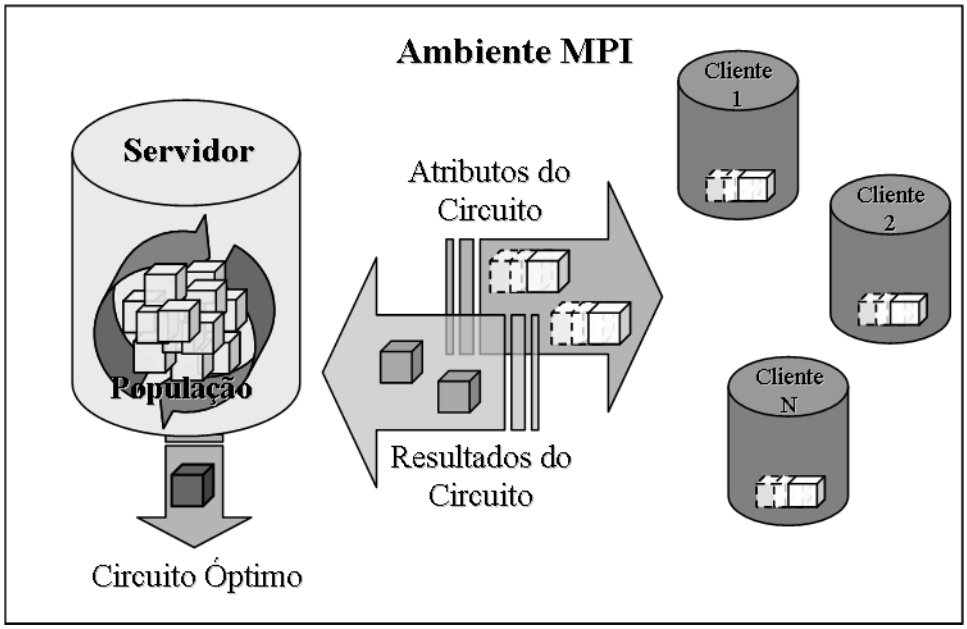

Figura 12: Esquema representativo da arquitectura (MPI) escolhida para aplicação

processos e assemelhando-se a um RPC, "Remote Procedure Call", pode também transportar mensagens com a finalidade de despoletar funções noutros processos.

A capacidade destas mensagens poderá preencher toda a variável/vector/matriz que se desejar carregar, respeitando algumas regras no seu acondicionamento, explicadas mais adiante na secção 3.3.1.3 - *Camada de Transporte (Mensagens MPI)*. Na Figura 12 o conteúdo das mensagens do Servidor para os Clientes transportam os atributos dos indivíduos, as mensagens de cada Cliente para o Servidor transportam o resultado do melhor *"fitness"* e respectiva *"netlist"*.

A camada MPI permite expandir o processamento aplicacional a mais do que uma máquina física. Este facto é possível através da comunicação entre processos resultantes da divisão do processamento total a realizar, em pequenas porções facilmente processadas em máquinas simples. Internamente, a de troca de mensagens entre o processo de cada nó, representa simples cópias de variáveis entre funções de processo distintos. Existe ainda a possibilidade de tornar a responsabilidade destes nós dinâmica, não exigindo o número inicial de nós como um requisito até ao final do processamento, tornando os processos mais simples de administrar e mais fáceis de manobrar para uma determinada quantidade de recursos. Desta forma, o controlo do processo genérico sobre os nós é mais complexa e também mais robusta, habilitando os processos de cada nó a circularem como opção, entre o interior do processo genérico e o seu exterior (funcionalidade da versão 2 do MPI, não desenvolvida neste trabalho), representando este último a destruição do mesmo ou simplesmente a sua hibernação para a poupança de recursos. Resumindo, implementando uma estrutura de processamento com clientes que permitem entrar e sair do processamento activo através de um "login" ou "logout" por exemplo (funcionalidades do MPI versão 2)

Colocando os últimos dois parágrafos em foco é ainda necessário perceber qual o conceito de arranque dos processos num ambiente MPI como ilustrado na Figura 13. Embora a comunicação para anunciar os arranques das aplicações entre nós/máquinas MPI seja um misto de comunicação sequencial/paralela, o arranque dos processos é feito em paralelo. O processamento dentro da aplicação distribuída é realizado em paralelo entre os vários processos ou *"MPI tasks"* sobre controlo local de cada nó, e quando o hardware permitir processamento paralelo local. Neste caso o nó Servidor, o primeiro a ser inicializado, arranca necessariamente na máquina onde se lança a aplicação distribuída. A Figura 13 representa de forma simples o processo de inicialização/arranque dos processos MPI, em que cada nó MPI representa uma máquina apenas. Neste arranque, cada nó MPI regista-se no ambiente MPI e

classifica-se como sendo apto a comunicar utilizando funções MPI. A este processo de caracterização chamamos de "Classificação MPI" como indicado na Figura 13. A razão da comunicação entre nós MPI ser um misto de comunicação sequencial/paralela é explicada pela organização em estrutura de árvore dos nós MPI. Neste ambiente MPI o número de ramificações da estrutura em árvore poderá ser configurada e adquirir um comportamento mais ou menos paralelo na comunicação entre nós MPI consoante o número de ramificações por cada nível da árvore.

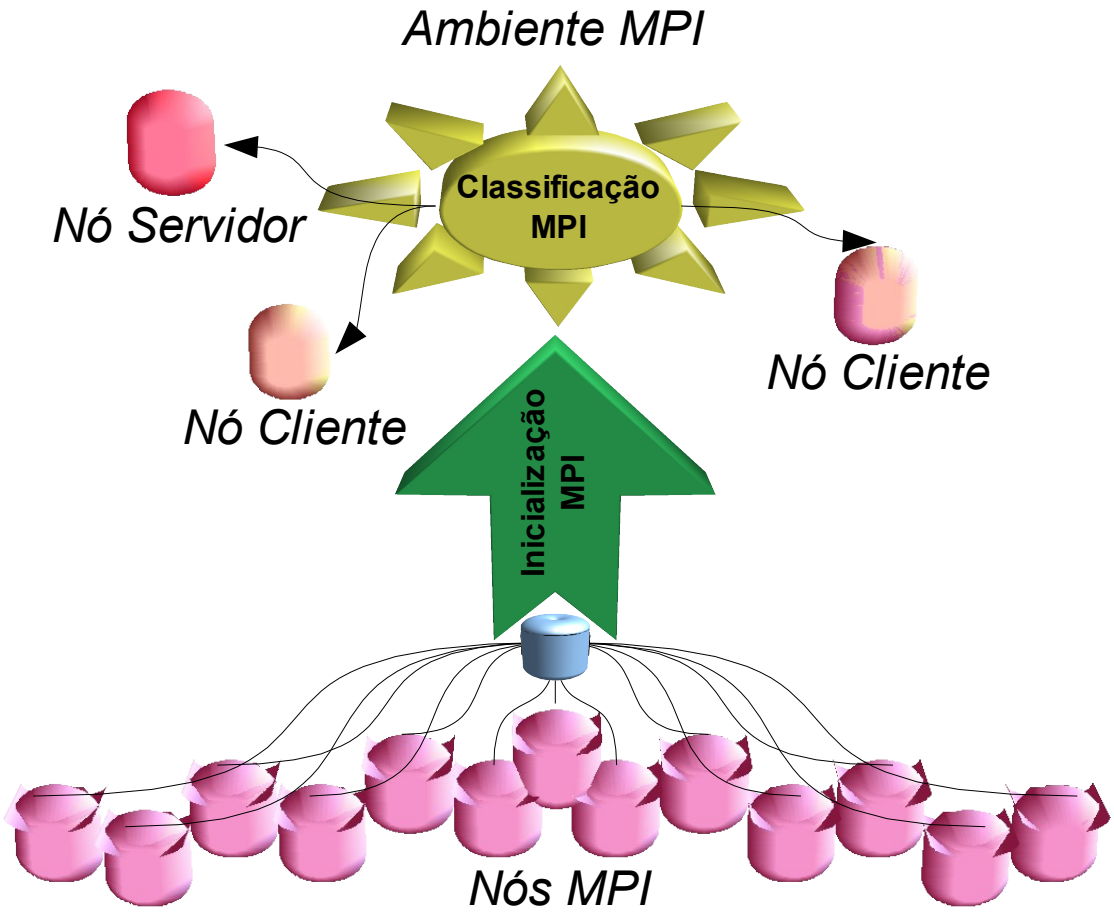

Figura 13: Inicialização dos nós no ambiente MPI

3.3.1.2 Dinâmica da Implementação

Finalmente, a dinâmica entre estes três últimos componentes, o nó Servidor, o nó Cliente e camada de transporte será tal, de modo a que se respeite os seguintes conceitos e estruturas como ilustrado em resumo na Figura 14, localizada no final da presente secção. Numa primeira implementação deste trabalho e tomando em conta que poderá ser continuado o trabalho aqui proposto em posteriores implementações, como descrito e exemplificado no capítulo 8 - *Trabalho Futuro* deste trabalho, temos que:

1. O programa deve lançar o ambiente MPI e realizar configurações consoante os nós disponíveis.
2. O simulador NGSPICE deve arrancar em todos os nós, excluindo o nó Servidor, acompanhado de um conjunto de informações necessárias ao processo que poderão estar em ficheiro acessíveis de todos as máquinas em que o processo corra ou predefinidas na própria aplicação.
3. O nó Servidor deve gerar todos os dados a serem processados pelos nós Cliente, inicializando todas as variáveis e criando todas as configurações necessárias aos nós Cliente.
4. Após a arquitectura cliente-servidor estar inicializada, o nó Servidor deve configurar os nós Cliente com as informações necessárias para o processamento seguinte através do envio de informação gerada no ponto anterior e/ou predefinida localmente.
5. Neste ponto a aplicação está pronta para começar o ciclo de processamento.
6. O nó Servidor deve então começar a enviar pacotes de informação para os nós Cliente e. Assim que todos os nós Cliente tenham pacotes para processar, a aplicação fica à espera de resultados para que na sua recepção torne a enviar nova informação a ser processada até que todos os pacotes sejam processados.
7. Depois de toda a informação processada nos nós Cliente e recolhida no nó Servidor, este último deverá efectuar uma nova geração baseada nos resultados recolhidos da geração anterior entre outros processos relacionados de forma a gerar esse novo conjunto de dados a serem processados. A informação recolhida da geração presente fica guardada para futuras comparações com as seguintes gerações.

8. Seguindo os anteriores pontos, 6 e 7, até se esgotarem o número de gerações configuradas e dentro destas, o número máximo de informação a processar em cada geração, chega-se ao fim do processo evolutivo para achar um resultado óptimo para o circuito a optimizar. Após cada geração é efectuada uma comparação para averiguar se o resultado presente é melhor que o anterior.
9. Terminadas todas as gerações, o melhor resultado é apresentado gerando uma *"netlist"* do circuito, representando o individuo óptimo encontrado.
10. São enviadas mensagens a todos os nós de Cliente requisitando que abandonem o ambiente MPI e terminarem a sua existência.
11. Os nós Cliente devem libertar alocações de memória e abandonarem de seguida o ambiente MPI para poderem cessar a sua actividade.
12. O nó Servidor deve igualmente libertar as suas alocações de memória e depois de enviar as mensagens aos nós Cliente, terminar a aplicação.

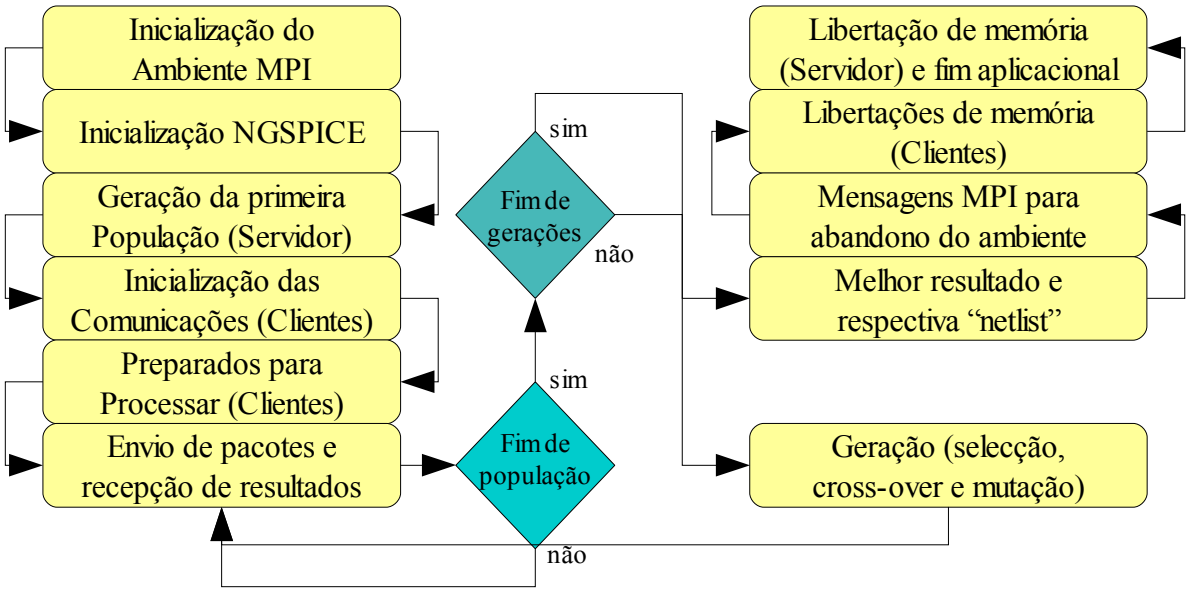

Figura 14: Esquema da dinâmica de implementação

3.3.2 Pré-Organização Estrutural

Antes de implementar quaisquer funções ou modificações ao código da aplicação entregue foi necessário organizar de diferente forma os seguintes pormenores de forma a facilitar a execução do plano de modificação da aplicação:

- Reorganizar as variáveis da aplicação de modo a facilitar a implementação das funções de controlo do código MPI.
- Simplificar o ficheiro "main.c" de forma a que neste sejam apenas lançados os módulos principais do programa, e exportar todo o código de inicialização do simulador NGSPICE para um outro ficheiro.
- Criar um outro ficheiro dedicado a todas as funções de MPI, incluindo as que chamam outras do programa já desenvolvidas.
- Preparar o código do simulador NGSPICE, para a implementação distribuída.

3.4 Alterações Implementadas

Nas seguintes secções irão ser detalhadas todas as alterações semânticas e lógicas ao código desenvolvido na aplicação entregue. Para validar correctamente os assuntos tratados na presente secção e seguintes subsecções é necessário ter em conta a versão da aplicação vigente à data da última alteração deste documento era a versão 1.0.011. Enumerando todas as funções por ficheiro com interesse significativo neste trabalho e relativas às alterações efectuadas, temos:

- (main.c) – int main(int argc, char **argv) – Função de arranque da Aplicação.
- (mpi_main.c) – Funções da estrutura MPI.
  - void master(void)
  - void slave(void)
  - void set_next_work_item(int)

- void unpack_work_item(void)
- void process_results(int)
- int do_work(void)
- int init_mpi_work(int, char**)
- int init_defs_master(int ntasks)
- int init_defs_slave(void)
- int slave_send_result(void)
- int master_recieve_result(int)
- void calculate_buffers_size(void)

- (ng_main.c) – Funções da estrutura do simulador NGSPICE.
  - int ng_main(int , char**)
  - void init_ng_load(void)
  - void init_auto_change(void)
  - void finalize_ng(void)
  - void master_ng_init(void)
  - void master_ng(int)
  - void master_ng_finalize(void)
  - void slave_ng_init_alloc(void)
  - void slave_ng_init_pop(void)
  - void slave_ng(void)
  - void slave_ng_finalize(void)
- (circuit_main.c) – Funções de adaptação ao ficheiro do circuito a simular.
- (circuit_objective.c) – Ficheiro do circiuto a simular.
  - doubledouble FitnessFunction(int , double*, double*, FILE*)
- (MultiThreadFitness.c) – Funções de adaptação ao *"multi-thread"*.
  - void AutoThreadFitness(int)
  - void SingleThreadFitness(void)
  - void MultiThreadFitness(void)
  - void FitnessLoop(double**, double**, int)
  - void SelectBestFitness(double**, double**)
  - void *FitnessThread(void*)
- (ga.c) – Funções da estrutura de algoritmos genéticos.
  - void Master_Alg_Gen_Init(void)
  - void Master_Alg_Gen_Process(int)
  - void Master_Alg_Gen_Finalize(void)
  - void Slave_Alg_Gen(void)
- (funcs.c) – Funções de apoio aos algoritmos genéticos.
  - void Init_Alloc_Slave(void)
  - void Alloc_Pop_Slave(void)
  - void Free_Pop_Slave(int npop_to_free)

- (ngaEDA.ini) – Ficheiro de inicializações/configurações de arranque (serão sobrepostas às configurações). Para mais detalhes consultar o Anexo B - Ficheiro de Configuração (ngaEDA.ini).

Sumariando de forma hierárquica todas as alterações/implementações ao código existente da aplicação desenvolvida neste trabalho, podemos sintetizar a estrutura modificada no seguinte diagrama ilustrado na Figura 15. Caso se pretenda consultar em detalhe o resto da estrutura gráfica da Figura 15 deverá consultar-se os seguintes documentos: Anexo D - Diagrama do Código Modificado (simples), Figura 40, Anexo E - Diagrama do nó Servidor, Figura 41 e Anexo F - Diagrama do nó Cliente, Figura 42:

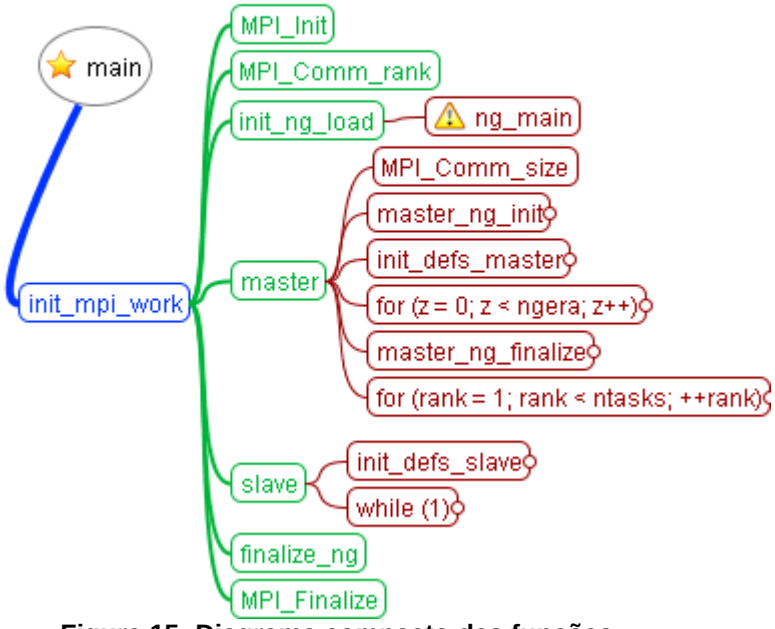

Figura 15: Diagrama compacto das funções modificadas/implementadas.

Na Figura 15 é ilustrado um excerto da estrutura de código construída, que esquematiza de forma hierárquica as funções da aplicação. Esta estrutura em árvore permite facilmente localizar determinadas funções e o ciclo de execução das mesmas. Os detalhes das funções anteriormente referidas estão descritos nos anexos D, E e F no parágrafo acima referidos.

3.4.1 Implementação da Estrutura MPI (Cliente-Servidor)

Nesta secção são apresentadas as funções e o código aplicacional produzido, para a implementação da estrutura MPI, de acordo com o que foi descrito na secção 3.3.1 - *Dinâmica a Implementar*, e suas subsecções. Contudo nos seguintes pontos e antes do detalhe do seu funcionamento, são introduzidos os conceitos básicos de cada uma das funções:

- void master(void) – Função que caracteriza o ciclo de um nó Servidor.
- void slave(void) – Função que caracteriza o ciclo de um nó Cliente.
- void set_next_work_item(int) – Função que prepara novos pacotes de dados para serem processados num nó Cliente.
- void unpack_work_item(void) – Desempacota um pacote de dados no nó Cliente.
- void process_results(int) – O nó Servidor processa todos os resultados recebidos dos nós Cliente.
- int do_work(void) – O nó Cliente inicia o processamento de um pacote de dados.
- int init_mpi_work(int, char**) – Inicia todo o processo MPI.
- int init_defs_master(int ntasks) – Inicializa definições de comunicação no nó Servidor.
- int init_defs_slave(void) – Inicializa definições de comunicação num nó Cliente.
  int slave_send_result(void) – Um nó Cliente envia o resultado do processamento ao nó Servidor.
- int master_recieve_result(int) – O nó Servidor recebe um resultado de um nó Cliente
- void calculate_buffers_size(void) – Aloca “buffers” para futuras comunicações estabelecidas após a chamada da respectiva função.

Na implementação da camada MPI da aplicação e tomando em conta os aspectos referidos na secção 3.3.1.1 - *Arquitectura Escolhida*, começou-se por implementar uma função que é invocada em todos os nós, apartir da função principal, *main(...)*. Esta função regista, por ordem de chegada, qual a identidade e número de ordem de chegada do processo a inicializar-se no ambiente MPI: nó Cliente ou nó Servidor. Por sua vez os processos MPI ou

"MPI tasks" utilizam uma área de comunicação para trocarem informação entre si. Esta área de comunicação denomina-se de *"MPI_COMM_WORLD"* e baseia-se num registo de memória por onde todos os processos MPI podem trocar mensagens entre si. Existe ainda outro comunicador pré-definido para utilizações semelhantes. Existe ainda a possibilidade de definir outros comunicadores para utilizações especiais, que só se justificam em casos muito específicos. A primeira função a ser chamada denomina-se de *init_mpi_work(...)* e o seu conteúdos é definido pelo seguinte:

```
int init_mpi_work(int argc, char **argv)
{

 MPI_Init(&argc, &argv);

 MPI_Comm_rank(MPI_COMM_WORLD, &myrank);

{...}*1

 MPI_Finalize();
 return 0;
}
```

A função *MPI_Init(...)* inicializa uma identidade MPI para o processo que a chama. Este processo depois de fazer parte da estrutura MPI, é registado no ambiente MPI e indicado qual o comunicador pré-definido a utilizar, neste caso o "*MPI_COMM_WORLD*", através da função *MPI_Comm_rank(...)*. Na variável *"myrank"* fica referenciado qual o número de chegada do respectivo processo que chamou a função *MPI_Comm_rank(...)*. No final da função, altura em que o processo sai do ambiente MPI, é chamada a função *MPI_Finalize()* indicando o abandono do ambiente MPI, bem como quaisquer referências a ele.

No interior (*1) da função previamente descrita, *init_mpi_work(...)* situa-se o código (abaixo indicado, no final deste parágrafo) que diferencia o nó Servidor dos nós Cliente. Caso o processo seja o primeiro a registar-se no ambiente MPI através da função *MPI_Comm_rank(...),* então este assume o comportamento de um nó Servidor. Os restantes nós automaticamente assumem o perfil de nó Cliente. Antes desta diferenciação é ainda necessário inicializar de forma homogénea o simulador NGSPICE através da função

init_ng_load(). Esta função por sua vez chama outra função, chamada de *ng_main()*, implementada com o objectivo de adaptar todas as funções que constituíam o código de inicialização do simulador NGSPICE no ficheiro principal da aplicação entregue representada pela função *main*(...). No final da aplicação a função *finalize_ng()* liberta quaisquer alocações de memória ou registos associados a ficheiros em uso. De seguida duas funções free(...), libertam as variáveis *"work_buffer"* e *"result_buffer"* alocadas previamente para as comunicações do MPI:

```
{...}
 init_ng_load();

 if (myrank == 0) {
  master();

 } else {
  slave();

 }
 finalize_ng();
 free(work_buffer);
 free(result_buffer);
{...}
```

Nas seguintes secções irão ser explicados os conteúdos das funções *master()* e *slave()*, respectivamente representando o nó Servidor e os nós Cliente. É ainda detalhada a estrutura das Mensagens MPI, trocadas entre nós e qual a configuração adoptada para este caso específico.

3.4.1.1 Nó de Gestão (Servidor)

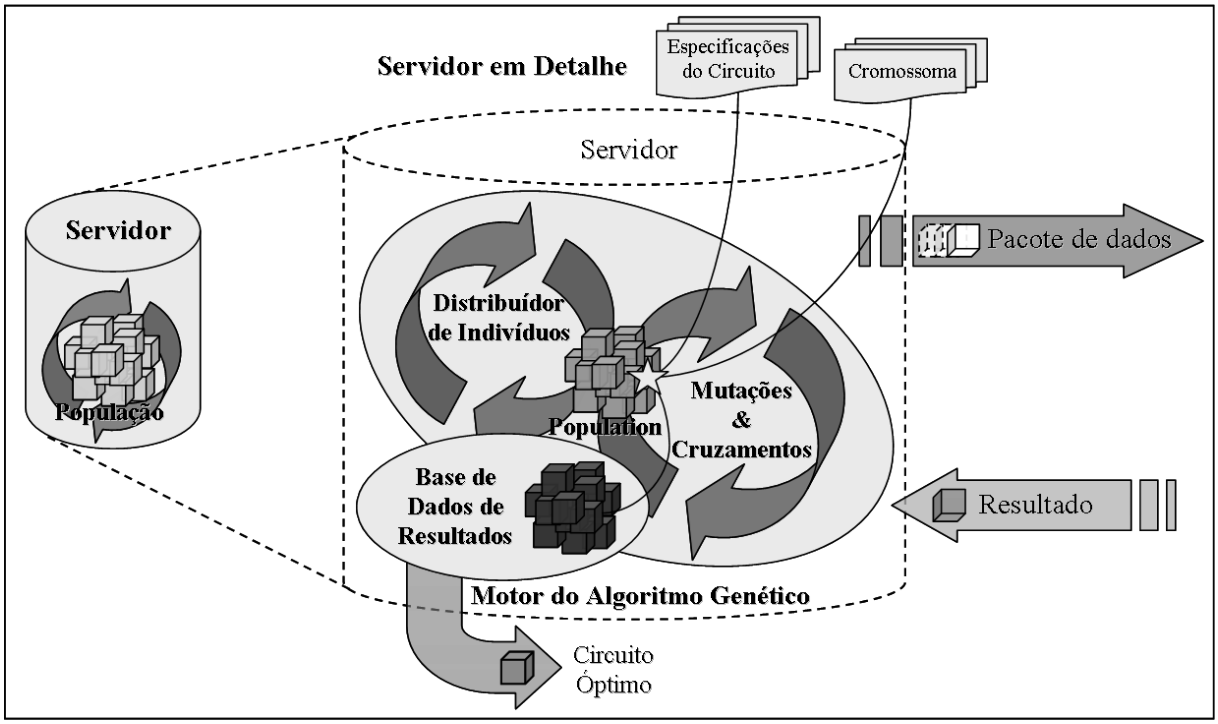

Figura 16: Detalhe da estrutura do Nó de Gestão (Servidor)

Caracterizando conceptualmente este nó pelo que a secção 3.3.1.1 - *Arquitectura Escolhida*, nos diz, é evidente que este nó tem a capacidade de controlo da aplicação, como ilustrado na Figura 16. Todo o processamento começa neste nó , desencadeando-se de seguida o arranque dos restantes nós, os nós Clientes. Após a estrutura definida, é realizada a configuração de todos os nós por intermédio de um ficheiro "*ngaEDA.ini*" e outras configurações pré-definidas na própria aplicação. Após todo o sistema inicializado este nó gera a primeira população baseada num ficheiro de configuração que contém os intervalos de valores admitidos, e respectiva resolução (nº de bits), para cada gene do indivíduo a gerar.

Este nó é responsável pela distribuição de pacotes com a informação necessária para se processar a análise de cada individuo, representado na Figura 16 como o "Distribuidor de Indivíduos". Estes pacotes contêm um ou mais indivíduos, que representam o circuito a ser avaliado. O resultado da avaliação são as "Especificações do Circuito", previamente estabelecidas, e que serão utilizadas no calculo do *"fitness"*. Dos atributos que possuem uma meta/objectivo a atingir, é guardada numa "Base de Dados de Resultados" que o indivíduo

simulado foi o melhor de todos os processados pelo nó Cliente, como indicado na Figura 16. O nó Servidor após receber um resultado e se ainda possuir mais pacotes a serem processados, envia mais um pacote para o nó do qual acabou de receber esse mesmo resultado.

Após receber todos os resultados dessa geração o nó Servidor executa alguns procedimentos de algoritmia genética como acções de *selecção*, *mutação* e *crossover* como ilustrado na Figura 16 a fim de melhorar os indivíduos a gerar na próxima população da geração que se segue e que potencialmente melhor configuram o circuito a optimizar.

Depois de todas as gerações processadas o nó Servidor deverá realizar a selecção do melhor indivíduo e apresentar os resultados e características do mesmo. Este resultado e todos os resultados intermédios são disponibilizados através de funções de *printf(...)* (*"stdout"* da consola) através da consola de onde se executou o arranque da aplicação distribuída, podendo estes ser redireccionados para ficheiro se desejável. O nível de detalhe das mensagens poderá ser configurado no ficheiro de configuração do nó Servidor, "ngaEDA.ini". Este é consultado no início de cada geração para possíveis alterações do nível de detalhes das mensagens.

Pormenorizando os parágrafos anteriores da presente secção e analisando o código da aplicação em anexo, constata-se que, o nó Servidor começa por referenciar um apontador *"ntasks"* para o número total de nós registados no comunicador escolhido, "*MPI_COMM_WORLD*", através da função *MPI_Comm_size(...)*. De seguida são inicializadas as seguintes variáveis, com os respectivos objectivos:

- slave_npop_size – número predefinido de indivíduos da população contida em cada pacote enviado para processamento aos nós Cliente caso não seja configurado nenhum valor.
- slave_npop_position – posição do próximo indivíduo a enviar para um nó Cliente.
- slave_result_position – posição da chegada do próximo resultado de um nó Cliente.

A função *master_ng_init()*, descrita, seguidamente, na Nota 17 gera a primeira população baseada nas definições de um ficheiro de configuração localizado nó Servidor, "*ngaEDA.ini*". Após a criação da população, é também configurado através do ficheiro "ngaEDA.ini" o número de indivíduos a processar por nó. Caso sejam verdadeiras as condições de existência de população, variável *existe_pop com valor "verdadeiro",* e estejam definidas as especificações do circuito a atingir, variável *existe_ind com valor "verdadeiro"*, a função *Mater_Alg_Gen_Init()* iniciará o procedimento de avaliação dos indivíduos sobre a forma de simulação dos correspondentes circuitos. Logo de seguida a função *init_defs_master(...)* descrita, seguidamente, na Nota 18, alocará a quantidade de memória necessária para as variáveis de comunicação do MPI. Depois de calculadas, o valor destas variáveis serão enviadas para cada um nós Cliente.

```
void master(void)
{
{...}
 slave_npop_size = 2;
 slave_npop_position = 0;
 slave_result_position = 0;


        master_ng_init();
        init_defs_master(ntasks);

for (z = 0; z < ngera; z++) {*1
        master_time_start = time(NULL);
        init_auto_change();
        sent_npop_size = slave_npop_size;

 for (rank = 1; rank < ntasks; ++rank) {*2
  set_next_work_item(z);
  MPI_Send(work_buffer,
        position,
        MPI_PACKED,
        rank,
        WORKSENT,
        MPI_COMM_WORLD);
  }
{...}
```

```
void master_ng_init(void)
{
Cria_Pop(iniparser_getstr(initSettings, "ga:crm"));
slave_npop_size = iniparser_getint(initSettings, "MPI:popsize", 1);
        if ((existe_pop)&&(existe_ind)){
                Master_Alg_Gen_Init();
        }
}
```

Nota 17: Detalhe da função master_ng_init()

```
int init_defs_master(int ntasks)
{
calculate_buffers_size();
work_buffer = (char*) calloc(work_buffer_size, sizeof(char));
result_buffer = (char*) calloc(result_buffer_size, sizeof(char));
for (rank = 1; rank < ntasks; ++rank) {
MPI_Pack(...);
{...}
MPI_Send(...);
}
```

Nota 18: Detalhe da função init_defs_master(...)

```
void set_next_work_item(int gera)
{
if ((slave_npop_position+sent_npop_size) > npop ) {
sent_npop_size=(npop-slave_npop_position);
}
MPI_Pack(...);
{...}
}
```

Nota 19: Detalhe da função set_next_work_item(...)

Depois de inicializado todo o ambiente aplicativo é possível iniciar o ciclo de envio da informação necessária aos nós Cliente. O primeiro ciclo, *"for (z = 0; z < ngera; z++)"* (*[1]), administra as gerações da população. A variável *master_time_start* inicializa um marcador de tempo, utilizado para fornecer detalhes do tempo de execução da optimização. A função *init_auto_change()* fornece a possibilidade de se poder alterar as mensagens de saída da aplicação a cada ciclo de geração através da leitura de duas variáveis, *mastermsg* e *slavemsg* contidas no ficheiro anteriormente referido como "*ngaEDA.ini*". Finalmente, a variável *sent_npop_size* é actualizada com o valor de *slave_npop_size*, que indicam o número de indivíduos a enviar em cada pacote de dados e o número máximo de indivíduos que os nós

Cliente devem processar respectivamente. O valor da variável *sent_npop_size* é alterado para o número de indivíduos restantes e por enviar de uma população. Esta alteração é realizada quando este número de indivíduos restantes for inferior ao número máximo de indivíduos que os nós Cliente deverão processar, ou seja, quando for inferior à variável *sent_npop_size*.

Por fim o segundo ciclo, *"for (rank = 1; rank < ntasks; ++rank)"* (*[2]), envia pacotes de indivíduos para todos os nós Cliente através da função *MPI_Send(...)* e da função *set_next_work_item(...)*, que envia e prepara o pacote respectivamente. A função que prepara os pacotes para envio, descrita anteriormente, na Nota 19, analisa se o tamanho de pacote pré-configurado é válido para o próximo envio e caso não seja, redimensiona-o. De seguida são empacotados os elementos necessários ao processamento do nó Cliente para que sejam enviados pela função *MPI_Send*(...) ao nó respectivo. O conteúdo das mensagens trocadas entre nós é explicado na secção 3.4.1.3 - *Camada de Transporte (Messages MPI)*, localizada na segunda secção adiante.

```
{...}
 while (slave_npop_position < npop) {

          master_recieve_result(z);
          set_next_work_item(z);

   MPI_Send(work_buffer,
          position,
          MPI_PACKED,
          status.MPI_SOURCE,
          WORKSENT,
          MPI_COMM_WORLD);

   }
{...}
```

```
int master_recieve_result(int gera)
{
MPI_Recv(...);
MPI_Pack(...);
{...}
}
```

Nota 20: Detalhe da função master_recieve_result(...)

Após todos os nós Cliente terem recebido informação para processar, o nó Servidor inicia um novo ciclo, "*for (rank = 1; rank < ntasks; ++rank)*" (adiante em *[3]) onde dedica a sua rotina à seguinte ordem de procedimentos até que o número total de indivíduos da população se esgote:

1. master_recieve_result(...) – fica à espera que um nó Cliente lhe envie um resultado, como indica a Nota 20. Esta função é bloqueante.
2. set_next_work_item(...) – prepara outro pacote de indivíduos a serem processados, como indica a Nota 19.
3. MPI_Send(...) – envia o pacote anteriormente preparado ao mesmo nó Cliente do qual recebeu um resultado de um anterior pacote de indivíduos enviado.

Depois de se esgotarem todos os indivíduos da população da actual geração, o nó Servidor entra num ciclo em que executa a recolha de todos os resultados ainda por receber, executado pela função *master_recieve_results(...)* como indica a Nota 20.

Para finalizar o ciclo da geração, o nó Servidor deverá então processar todos os resultados recebidos, da população actual, através da função *master_ng(...)*, chamada de dentro da função *process_results(...)* como indica a Nota 21. Através desta última função o nó Servidor deverá efectuar um conjunto de selecções, mutações e cruzamentos, para que no final se volte a gerar uma nova população, pronta para ser novamente processada até que o número de gerações configuradas se esgote. Mesmo antes do processo acabar uma geração, é guardada na variável *master_time_stop* uma marca de tempo, para que no final se possa contabilizar o tempo de processamento total da optimização. Este tempo de processamento é ainda convertido num formato facilmente interpretável (*[4]) como ilustrado na Nota 22. Este tempo é calculado em segundos e depois disponibilizado no formato: "dias" "horas" "minutos" e "segundos".

```
{...}
*3  for (rank = 1; rank < ntasks; ++rank) {
        master_recieve_result(z);
  }
        process_results(z);
        master_time_stop = time(NULL);
```

```
void master_ng(int n_gera)
{
if ((existe_pop)&&(existe_ind)){
Master_Alg_Gen_Process(n_gera);
}
}
```

Nota 21: Detalhe da função master_ng(...) chamada de dentro da função process_results(...)

```
*4
```

```
master_total_time = master_time_stop - master_time_start;
TotalTimeRun = TotalTimeRun + (long)master_total_time;
long total_time_estimage = TotalTimeRun*ngera/(z+1);
long time_wait_estimage = total_time_estimage - TotalTimeRun;
long tw_days = time_wait_estimage/86400;
long tw_hours = time_wait_estimage/3600 - (tw_days*24);
long tw_minutes = time_wait_estimage/60 - (tw_hours*60+tw_days*24*60);
long tw_seconds = time_wait_estimage -
(tw_minutes*60+tw_hours*3600+tw_days*86400);
long tt_days = total_time_estimage/86400;
long tt_hours = total_time_estimage/3600 - (tt_days*24);
long tt_minutes = total_time_estimage/60 - (tt_hours*60+tt_days*24*60);
long tt_seconds = total_time_estimage -
(tt_minutes*60+tt_hours*3600+tt_days*86400);
```

Nota 22: Código para estatística dos tempos de processamento

```
}

        master_ng_finalize();

 for (rank = 1; rank < ntasks; ++rank) {
   MPI_Send(0, 0, MPI_PACKED, rank, DIETAG, MPI_COMM_WORLD);
 }

}
```

Para finalizar todo o processo do nó Servidor, depois de todas as gerações esgotadas é necessário libertar recursos alocados e desmantelar quaisquer variáveis alocadas através da função *master_ng_finalize()*. Para sinalizar os nós Cliente, que o mesmo procedimento deverá ser completado, o nó Servidor enviará a todos os restantes nós uma mensagem especial, marcada com uma "etiqueta" denominada de "DIETAG" que indica o abandono do ambiente MPI por parte de todos os nós nele registados.

3.4.1.2 Nós de Processamento (Clientes)

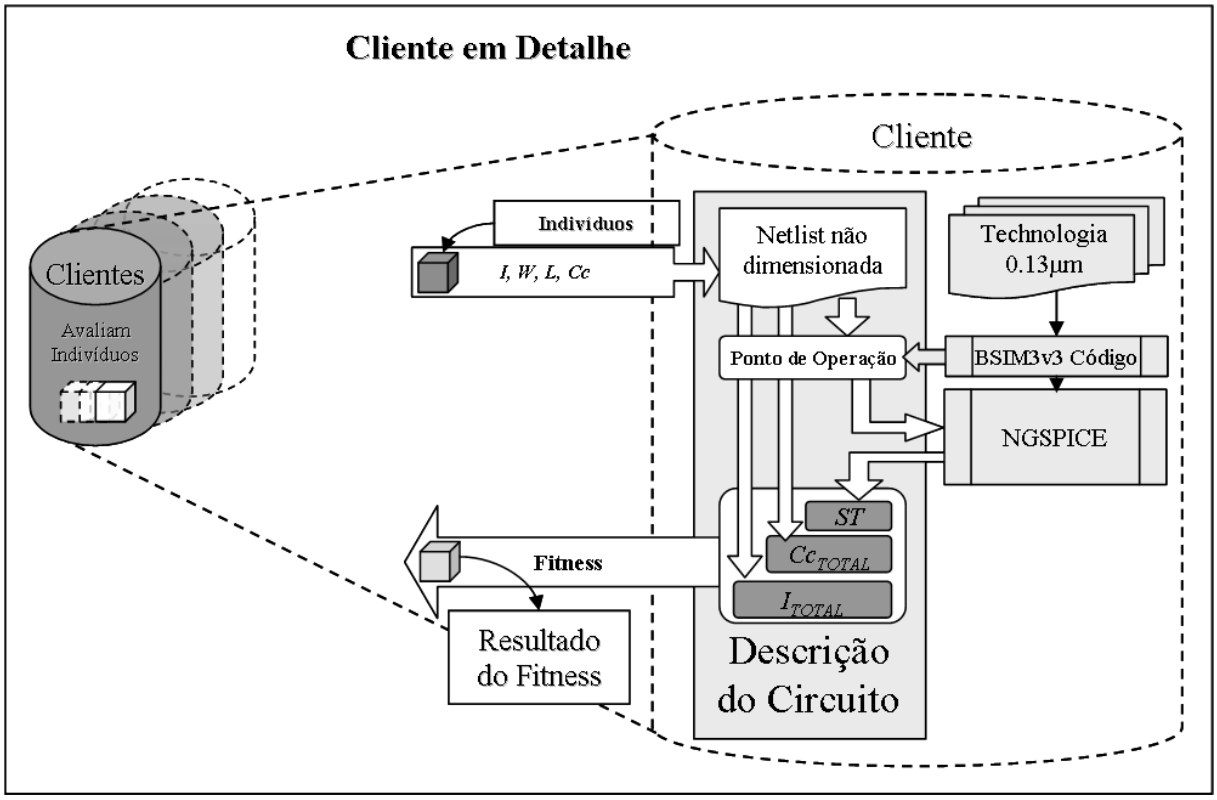

Figura 23: Detalhe da estrutura do Nó de Processamento (Cliente)

Tomando em conta o segundo parágrafo da secção anterior, 3.4.1.1 - *Nó de Gestão (Servidor),* podemos caracterizar os nós Cliente como os de maior importância em todo o processo de optimização da aplicação. Estes nós deverão realizar o trabalho "pesado" da aplicação e também o mais repetitivo. Assim sendo, é possível dividir os dados a processar em pequenas parcelas que requerem muito menos tempo de processamento, consoante a relação entre o volume total de dados e o número de divisões realizadas nesse volume de dados, reduzindo assim o tempo de processamento aplicacional.

Sendo a simulação de um circuito um processo complexo e demorado à medida que o tamanho da população adquire maiores proporções, o principal objectivo dos nós Cliente é reduzir esta complexidade e tempo de processamento, assumindo a responsabilidade do processamento das simulações. Contudo, estas simulações também envolvem uma quantidade de aritmética muito elevada relativa a um conjunto de componentes aplicacionais já

desenvolvidos, o «*NGSPICE*» e algumas estruturas dos algoritmos genéticos, como ilustra a Figura 23.

Partindo do estado da aplicação inicial, ilustrado na Figura 11 isolou-se o processamento cíclico das simulações para que este passe a ser executado em vários processos em simultâneo. Desta forma é possível transformar o retorno de resultados um acontecimento mais frequente através do processamento de simulações em dois ou mais nós em paralelo como representado na Figura 23.

Para melhor preparar a aplicação em futuras modificações e assumindo que a simulação de um circuito tende a potencialmente ficar mais lenta com o aumento da complexidade desse mesmo circuito em optimização, preparou-se a implementação para, no futuro, evoluir para uma implementação que incluísse processamento *"multi-thread"*. No futuro esta estrutura poderá ser completada, através da implementação de um sistema de gestão de *"threads"* para o processamento em multi-tarefa. Desta forma é possível reaproveitar melhor os recursos da tendência da tecnologia dos nossos dias através do aumento do número de processadores lógicos e físicos por máquina. Este parágrafo vai ser discutido mais pormenorizadamente na secção 3.4.4 - *Implementações Secundárias*, mais adiante.

Detalhando o desenvolvimento do código nos nós Cliente constata-se que, antes do nó entrar no seu único ciclo de trabalho, começa por inicializar-se, chamando a função *init_defs_slave()*, como detalhado na Nota 24 *(mais adiante). Esta função inicializa e aloca memória para todas as variáveis necessárias à comunicação do nó Cliente, de forma a que este, consiga receber, processar e enviar todos os pacotes recebidos do nó Servidor. Sendo uma função bloqueante, MPI_Recv(...), fica em modo de espera até que receba um pacote do nó Servidor com a informação necessária para carregar as definições de comunicação, como "buffers" e alocações de memória necessárias à comunicação dos nós. As variáveis armazenadas são: work_buffer_size que define o tamanho do "buffer" para armazenar o número de indivíduos por pacote; result_buffer_size que define o tamanho do "buffer" para armazenar o resultado de cada nó Cliente; NVAR define o número de variáveis (genes) por indivíduo; lim_max e lim_min definem o máximo e mínimo dos intervalos admitidos por cada gene do individuo*; e *ngen* que define o número de genes por atributo de um indivíduo. Juntamente com esta informação de comunicação vem o valor da capacidade necessária para alocar algumas variáveis necessárias ao controlo do estado da simulação. As alocações das variáveis referidas anteriormente são efectuadas por intermédio da função *slave_ng_init_alloc()* que chama a função *Init_Alloc_Slave()*, onde são então efectuadas as alocações, como detalhado na alínea 1 do Anexo I – Código de Funções.

```
void slave(void)
{
 init_defs_slave();
{...}
```

```
int init_defs_slave(void)
{
MPI_Recv(...);
MPI_Unpack(work_buffer_size);
MPI_Unpack(result_buffer_size);
MPI_Unpack(NVAR);
slave_ng_init_alloc();
MPI_Unpack(lim_max);
MPI_Unpack(lim_min);
MPI_Unpack(ngen);
}
```

Nota 24: Detalhe da função init_defs_slave()

Após configuradas as opções que permitem interagir correctamente com o nó Servidor, os nós Cliente iniciam um ciclo que termina com o retorno da própria função. Porém estes ciclos terão a duração do número de iterações necessárias, até que se esgote o

número total de indivíduos contidos na população da geração actual, enviados pelo nó Servidor.

Generalizando, esta arquitectura cliente-servidor, adquire a capacidade do auto-balanceamento porque o envio da carga para os nós Cliente é realizada consoante a capacidade de processamento de cada cliente. Inicialmente, todas as máquinas clientes recebem o mesmo número de indivíduos para processar. As máquinas mais rápidas continuam a receber novos indivíduos para processar, enquanto as máquinas mais lentas continuam a processar os seus indivíduos.. Todos os nós esperaram que a última simulação seja processada, da máquina mais lenta. A optimização só termina quando a máquina mais lenta terminar a avaliação do seu último individuo. Esta arquitectura simplifica a organização do código de controlo de toda a aplicação e como não necessita de aplicações adicionais que executem a gestão dos processamento distribuído, as dependências de execução da aplicação são mais reduzidas.

Novamente bloqueante, a primeira função dentro do ciclo dos nós Cliente é a função *MPI_Recv()*, destinada a receber os pacotes de indivíduos a processar, que no final de cada simulação ou conjunto de simulações de cada pacote é enviado para o nó Servidor o melhor resultado processado através da função *slave_send_result*() detalhada na Nota 26.

No seguimento da função que recebe os pacotes de indivíduos, *MPI_Recv(...)*, é inicializada uma variável *slave_time_start* com o mesmo objectivo que a do mesmo tipo *master_time_start* mencionada na secção anterior, 3.3.1.1 - *Nó de Gestão (Servidor)*. A cada iteração do ciclo é contabilizado o tempo de simulação, terminando a contagem do tempo quando se regista o valor da variável *slave_time_stop*.

```
{...}
 while (1) {

   MPI_Recv(work_buffer, work_buffer_size, MPI_PACKED, 0, MPI_ANY_TAG,
                    MPI_COMM_WORLD, &status);
          slave_time_start = time(NULL);


   if (status.MPI_TAG == WORKSENT) {
          unpack_work_item();
          do_work();
          slave_time_stop = time(NULL);

          slave_total_time = slave_time_stop - slave_time_start;
          slave_send_result();
   }

   if (status.MPI_TAG == DIETAG) {
          slave_ng_finalize();
     return;
   }
  }
}
```

```
void unpack_work_item(void)
{
MPI_Unpack(npop);
slave_ng_init_pop();
for (i = 0; i < npop; i++) {
          for (j = 0; j < NVAR; j++) {
                    for (k = 0; k < ngen[j]; k++) {
                    MPI_Unpack(pop[i][j][k]);
{...}
```

Nota 25: Detalhe da função unpack_work_item()

```
int slave_send_result(void)
{
MPI_Pack(max_fit_geracao);
MPI_Pack(npop);
for (i = 0; i < npop; i++) {
          MPI_Pack(pop[i]);
}
MPI_Pack(temp_long);
for (i = 0; i < NIND; i++) {
          MPI_Pack(Ind_Act_Melhor[i]);
}
for (i = 0; i < NVAR; i++) {
          MPI_Pack(melhor_cromossoma[i]);
}
}
```

Nota 26: Detalhe da função slave_send_result()

```
void slave_ng_finalize(void)
{
          if (existe_pop) {
          Free_Pop_Slave(npop);
          }
          if (existe_ind) Liberta_Ind();
}
```

Nota 27: Detalhe da função slave_ng_finalize()

Iniciada a contagem de tempo da simulação, são testadas duas possíveis situações ou mais propriamente, são testados dois tipos de pacotes. Caso o pacote contenha a variável "*status.MPI_TAG*" igual ao valor simbólico de *"WORKSENT"*, então estamos perante um pacote com indivíduos para se realizarem simulações. Caso o pacote contenha a mesma variável anterior igual ao valor simbólico *"DIETAG"*, então estamos perante uma situação

final em que o nó Cliente abandona o ambiente MPI e liberta alguns recursos alocados através da função *slave_ng_finalize()*, detalhada na Nota 27. Na situação de um pacote para processamento, este é descompactado e guardado em variáveis locais através da função *unpack_work_item()* detalhada na Nota 25. De seguida é executada a simulação através da função *do_work()*, que por sua vez chama um conjunto de funções umas a seguir às outras, começando pela *slave_ng()*. Caso exista população e indivíduos esta última função chama a função *Slave_Alg_Gen()*. Esta última função, finalmente chama a função *AutoThreadFitness()* que é explicada na secção 3.4.4 - *Implementações Secundárias*, mais adiante. A razão desta construção, de chamada de funções "em corrente", deveu-se ao facto de se querer separar cada função nas suas respectivas áreas conceptuais, onde se tratam assuntos semelhantes, como o MPI, o NGSPICE, os algoritmos genéticos, etc. Segue-se depois a marcação do tempo de termo da simulação e é efectuado o cálculo do tempo total da simulação. Este último é guardado na variável *slave_total_time* para que se possa enviar juntamente com o com o melhor resultado da(s) simulação(s) através da função *slave_send_result()*, para o nó Servidor como ilustrado na Nota 26.

3.4.1.3 Camada de Transporte (Mensagens MPI)

Suportando toda a interacção entre o nó Cliente e o nó Servidor (secções anteriores 3.4.1.1 - *Nó de Gestão (Servidor)* e 3.3.1.2 - *Nós de Processamento (Clientes)*) existe a Camada de Transporte (Mensagens MPI). Todo o processo de comunicação é assegurado por esta camada, através de funções do MPI com o objectivo de executar, administrar, monitorizar, e reciclar todas as comunicações estabelecidas entre o nó Servidor e nós Cliente.

Estas mensagens são de estrutura pré-definida mas com capacidades dinâmicas dentro de cada tipo de transporte, envio e recepção. Nesta camada existem variáveis internas ao código MPI e variáveis não MPI que servem de auxílio à quantidade de informação transportada dentro dos pacotes de dados. No domínio da camada MPI existem regras de empacotamento e desempacotamento antes do envio e recepção de pacotes respectivamente. Existem também regras de manuseamento das mensagens relativamente ao seu envio e recepção, necessárias ao contexto de comunicação MPI afim de manter o sentido lógico e funcional da comunicação entre dois pontos.

As mensagens MPI que circulam pela Camada de Transporte resultam do empacotamento/desempacotamento de registos representados como variáveis, vectores, matrizes, caracteres ou conjuntos destes, respectivamente, que viajam entre dois pontos, nó Servidor e nó Cliente. Estas mensagens quando chegam aos destinatários necessitam de ser desempacotadas a fim de se reaver o seu conteúdo, ou seja, a informação previamente empacotada antes do pacote ter sido enviado.

Detalhando melhor os dois métodos de transporte, temos que no envio é necessário empacotar a informação que passa a ser representada por um único objecto do tipo "*MPI_PACKED*". Na recepção é necessário desempacotar o mesmo objecto para que seja extraída a sua informação. As funções que empacotam e desempacotam estão detalhadas de seguida e são respectivamente *MPI_Pack(...)* e *MPI_Unpack(...)*:

```
MPI_Pack(     <variable_to_pack>,
              <variable_to_pack_vector_size>,
              <MPI_VAR_TYPE>,
              <buffer>,
              <buffer_size>,
              <&current_position_in_buffer>,
              <MPI_COMUNICATION_ENVIRONMENT>);
```

```
MPI_Unpack(   <buffer>,
              <buffer_size>,
              <&current_position_in_buffer>,
              <variable_unpacked>,
              <variable_unpacked_vector_size>,
              <MPI_VAR_TYPE>,
              <MPI_COMUNICATION_ENVIRONMENT>);
```

Em ambas as funções existe uma variável temporária, <*buffer*> para onde são empacotados ou de onde são desempacotados os dados para envio ou recepção respectivamente. Esta variável terá um tamanho pré-definido, <*buffer_size*>, que não poderá ser excedido pela variável que aponta a próxima posição, <*&current_position_in_buffer*> e onde pode ser escrita ou lida informação da ou para a variável <*buffer*>. A informação/dados

é guardada na variável <*variable_unpacked*> e lida da variável <*variable_to_pack*> de respectivos tamanhos, <*variable_unpacked_vector_size*> e <*variable_to_pack_vector_size*>.

Depois de empacotar para um "*buffer*" todos os registos necessários é então possível enviar o objecto do tipo *MPI_PACKED* para outro nó através da função *MPI_Send(...)*. Do mesmo modo na recepção o nó destinatário poderá receber o objecto através da função *MPI_Recv(...)*. Estas duas últimas funções estão detalhadas já a seguir:

```
MPI_Send(   <buffer>,
            <buffer_size>,
            MPI_PACKED,
            <node_rank_id_to>,
            WORKTAG,
            <MPI_COMUNICATION_ENVIRONMENT>);
```

```
MPI_Recv(   <buffer>,
            <buffer_size>,
            MPI_PACKED,
            <node_rank_id_from>,
            WORKTAG,
            <MPI_COMUNICATION_ENVIRONMENT>,
            <&message_information_status>);
```

A variável <*buffer*> é um registo de onde e para onde são lidos ou guardados os dados úteis a enviar ou receber, respectivamente. A variável <*buffer_size*> indica o tamanho da variável <*buffer*> que é enviado para o nó de número igual à variável <*node_rank_id_to*> ou recebido do nó de número igual à variável <*node_rank_id_from*>. Na recepção são ainda guardadas algumas variáveis dentro de uma estrutura apontada por <*&message_information_status*>. A variável <*MPI_COMUNICATION_ENVIRONMENT*> indica qual o comunicador a utilizar para o ambiente MPI de onde se está a realizar as operações de empacotamento, envio, recepção e desempacotamento.

3.4.2 Alterações na implementação dos algoritmos genéticos

Nesta secção serão apresentadas as funções e o código produzido, ao nível da aplicação, para a implementação da estrutura de algoritmos genéticos, de acordo com o que foi descrito na secção 3.3.1 - *Dinâmica a Implementar* e suas subsecções. Contudo nos seguintes pontos e antes do detalhe do seu funcionamento, serão introduzidos os conceitos básicos de cada uma das funções:

- void Master_Alg_Gen_Init(void) – Inicializa a estrutura de algoritmos genéticos.
- void Master_Alg_Gen_Process(int) – Processamento de informação (resultados) da estrutura de algoritmos genéticos.
- void Master_Alg_Gen_Finalize(void) – Liberta variáveis respeitantes à estrutura de algoritmos genéticos.
- void Slave_Alg_Gen(void) – Função principal dos algoritmos genéticos (representa um *"kernel"* de algoritmos genéticos como falado em [5]), modificada para permitir a sua chamada de um ambiente preparado para processamento em *"multi-thread"*.

De igual modo todas as que serviram de apoio aos algoritmos genéticos:

- void Init_Alloc_Slave(void) – Inicializa recursos (alocações de memória) da estrutura de algoritmos genéticos.
- void Alloc_Pop_Slave(void) – Aloca recursos de dados na estrutura de algoritmos genéticos.
- void Free_Pop_Slave(int npop_to_free) – Liberta recursos de dados na estrutura de algoritmos genéticos.

As primeiras cinco funções anunciadas nesta secção e referidas nos primeiros 5 pontos, resultaram da separação do antigo código de forma a que fosse possível implementar as restantes funções da camada MPI sem que se perdesse todo o código já desenvolvido. No entanto poderá ser consultado o seu código nas alíneas 1, 2, 3, 4 e 5 do Anexo I – Código de Funções. As restantes duas funções foram implementadas para completar a separação das funções antigas nas apresentadas na presente secção.

```
void Free_Pop_Slave(int npop_to_free)
{
  for (i = 0; i < npop_to_free; i++){
    for (j = 0; j < NVAR; j++){
      free(pop[i][j]);
      free(novapop[i][j]);
    }
    free(pop[i]);
    free(novapop[i]);
  }
  free(pop);
  free(novapop);
  free(fit);
  free(cumfit);
  existe_pop = 0;
}
```

Nota 28: Detalhe da função Free_Pop_Slave(...)

Começando pelo detalhe da mais simples, *Free_Pop_Slave(...)* ilustrada na Nota 28, tem como objectivo libertar todas as alocações de memórias realizadas nos nós Cliente relativas aos dados das populações e dos resultados provenientes das simulações.

Por último a função, *Alloc_Pop_Slave()*, detalhada na Nota 29, foi implementada a fim de permitir a alocação dinâmica de menos memória para a matriz da população e para outros vectores que dependem linearmente do tamanho dos pacotes que os nós Cliente recebem, como por exemplo as variáveis *fit* e *cumfit*. Este fenómeno tem lugar quando os nós recebem pacotes com uma dimensão, número de indivíduos, menor que aquela com que começaram o processamento inicial de uma determinada geração. A condição que indica se deve ou não ser modificado o tamanho de memória alocada está definido através da condição inicial que representa a situação de quando um novo tamanho requerido ao nó Cliente é diferente do actual e existe memória alocada, terceira

```
void Alloc_Pop_Slave(void)
{
 if ((existe_pop == 1) && (old_npop != npop)) {
          Free_Pop_Slave(old_npop);
          }
 if ((existe_pop == 0) && (old_npop != npop)) {
  fit = (double*) calloc(npop, sizeof(double));
  cumfit = (double*) calloc(npop, sizeof(double));
  pop = (char***) calloc(npop, sizeof(char**));
  novapop = (char***) calloc(npop, sizeof(char**));
  for (j = 0; j < npop; j++){
    pop[j] = (char**) calloc(NVAR, sizeof(char*));
    novapop[j] = (char**) calloc(NVAR, sizeof(char*));
    for (k = 0; k < NVAR; k++){
      pop[j][k] = (char*) calloc(ngen[k], sizeof(char));
      novapop[j][k] = (char*) calloc(ngen[k], sizeof(char));
    }
  }
          existe_pop = 1;
          old_npop = npop;
 }
}
```

Nota 29: Detalhe da função Alloc_Pop_Slave(...)

linha da Nota 29. Neste caso é libertada a memória e novamente alocada memória de acordo com os requisitos do tamanho em memória ocupado pela nova quantidade de indivíduos a ser entregue ao nó Cliente.

3.4.3 Alterações na implementação do NGSPICE

Nesta secção serão apresentadas as funções e o código produzido, ao nível da aplicação, para a implementação da estrutura do simulador NGSPICE, de acordo com o que foi descrito na secção 3.3.1 - *Dinâmica a Implementar* e suas subsecções. Contudo nos seguintes pontos e antes do detalhe do seu funcionamento, serão introduzidos os conceitos básicos de cada uma das funções:

- int ng_main(int , char**) – Inicializa o ambiente de simulação da NGSPICE.
- void init_ng_load(void) – Inicializa algumas variáveis para o ambiente de simulação e lança-o através da função do ponto anterior (*ng_main(...)*).
- void init_auto_change(void) – Relê o ficheiro de configurações e actualiza variáveis consoante alterações induzidas no ficheiro de configuração (*"ngaEDA.ini"*).
- void finalize_ng(void) – Liberta memória alocada no arranque da aplicação (ponteiro para um ficheiro e a variável que permite apontar e interpretar o ficheiro de configurações).
- void master_ng_init(void) – Cria a primeira população através das definições de configuração e prepara todo o ambiente dos algoritmos genéticos para que se possa posteriormente começar a processar/distribuir pacotes no nó Servidor.
- void master_ng(int) – Processa os resultados da geração actual (escolhe o melhor) e cria uma nova população com base no melhor resultado e nas operações de selecção, mutação e cruzamento (tudo no nó Servidor).
- void master_ng_finalize(void) – Liberta memória alocada pelos algoritmos genéticos no nó Servidor.
- void slave_ng_init_alloc(void) – Aloca memória para variáveis necessária à estrutura dos algoritmos genéticos nos nós Cliente.

- void slave_ng_init_pop(void) – Aloca memória para a criação de uma nova população na estrutura de algoritmos genéticos nos nós Cliente. Esta função reorganiza ainda a memória alocada para a população em processamento (conjunto de indivíduos) quando o número de indivíduos se altera.
- void slave_ng(void) – Inicia o processamento do(s) indivíduo(s) do pacote que foi recebido no nó Cliente.
- void slave_ng_finalize(void) – Liberta memória alocada pelos algoritmos genéticos no nó Cliente.

Finalmente irá ser explicado como introduzir novos circuitos através do ficheiro de circuito "circuit_objective.c" e onde introduzir adaptações ao código caso estas sejam necessárias no ficheiro "circuit_main.c". Relativamente à função que representa o código do circuito "circuit_objective.c":

- double FitnessFunction(int , double*, double*, FILE*) – Retorna o valor da avaliação dos objectivos a cumprir, o *"fitness"*. Estes objectivos são calculados a partir de resultados da simulação do circuito que se pretende optimizar, também processada nesta função. Esta função é processada apenas nos nós Cliente.

Caso sejam necessárias adaptações ou novas inicializações de variáveis no ficheiro anterior, "circuit_main.c", onde é chamada a anterior função, estas deverão ser realizadas no ficheiro da própria função, o "circuit_objective.c" sempre que possível. Desta forma é possível manter a coerência de programação, alterando apenas os ficheiros vocacionados para os circuitos. Caso se evidencie uma forma mais universal de alterar estas inicializações, declaradas no ficheiro anterior, "circuit_objective.c", então deverá ser implementada uma função tal que, uniformize esse mesmo o formato. Este deverá pelo menos englobar um conjunto de circuitos suficientemente volumoso, para que possa ser usado em eventuais funções do ponto anterior com sucesso e sem quaisquer outras alterações de código.

No que diz respeito ao parágrafo anterior e simplificando o método de introdução de novos circuitos na aplicação, temos que, a introdução de um novo circuito/ficheiro "circuit_objective.c" deverá respeitar um conjunto de regras. Estas regras fortificam a compatibilidade do código desenvolvido. Em caso contrário, é necessário implementar

adaptações ao código existente, se possíveis no ficheiro "circuit_main.c", de forma geral e acessível a outros circuitos potencialmente semelhantes:

- Respeitar as variáveis externas de toda a aplicação, não removendo nenhumas e ignorando caso não necessárias).
- Reutilizar a estrutura de dados imposta pela aplicação, e caso seja necessário implementar novas estruturas adaptar a já implementada de forma a que seja incorporada na nova e se mantenha compatível do ponto de vista da retro-compatibilidade.
- Permitir sempre o processamento isolado de todas as funções implementadas ou utilizadas do NGSPICE de forma a manter a compatibilidade do processamento em paralelo e a sua portabilidade caso seja necessário.

As alterações introduzidas na estrutura do simulador da NGSPICE da aplicação desenvolvida, devem-se apenas a reestruturações na aplicação. Existiram ainda outras pequenas implementações de código com o objectivo de facilitar a separação das funções MPI das restantes funções. Para tal foram criadas as funções enunciadas nesta secção, com excepção da função *FitnessFunction(...)* como funções do tipo "ponte". Estas têm como objectivo, chamarem outras funções relativas à estrutura dos algoritmos genéticos, entre outras, implementadas para controlo local.

Relativamente ao detalhe das funções referidas no inicio desta secção, poderão ser consultadas nas alíneas 1, 2, 3, 4 e 5 do Anexo I – Código de Funções. Contudo, para completar, um exemplo do procedimento de integração de um novo circuito no código da aplicação desenvolvida, poderá ser consultado o Anexo H – Exemplo de integração de um novo circuito.

3.4.4 Implementações Secundárias

Nesta secção são apresentadas as funções e o código produzido, ao nível da aplicação, para a implementações secundárias. Deste modo todas as funções que serviram de apoio à adaptação da aplicação para processamento em *"multi-thread"*, são:

- void AutoThreadFitness(int) – Função que permite num futuro próximo, através da implementação de um sistema automático, detectar de forma automática ou manual a forma de processamento das simulações nos nós Cliente, isto é, a configuração do número de *"threads"* a lançar para o processamento das simulações.
- void SingleThreadFitness(void) – Modo chamado pelo ponto anterior quando se configura os nós Cliente para processarem as simulações uma de cada vez, em apenas uma *"Thread"*.
- void MultiThreadFitness(void) – Modo chamado pelo segundo ponto anterior quando se configura os nós Cliente para processarem mais do que uma simulação de cada vez, em mais do que uma *"thread"*.
- void FitnessLoop(double**, double**, int) – Função que chama o ciclo principal dos nós nós Cliente.
- void SelectBestFitness(double**, double**) – Função que selecciona o melhor resultado das simulações processadas num nó Cliente. Desta forma apenas é enviado o melhor resultado optimizando a comunicação da aplicação.
- void *FitnessThread(void*) – Função que lança a *"Thread"* que processa a simulação. Esta função está preparada para processamento *"multi-thread"* usando "Pthreads". No trabalho desenvolvido apenas processa uma simulação faltando implementar o motor de controlo que realiza a gestão do lançamento de *"Threads"*.

Tomando em conta a tendência das aplicações presentes e futuras, em subdividirem-se em *"threads"* para tomarem partido do aumento dos processadores lógicos existentes por máquina, decidiu-se implementar um sistema que adapta-se o processamento da aplicação

desenvolvida neste trabalho a sistemas de multiprocessamento local através do uso de várias *"threads"* por processo e por máquina.

Este sistema baseia-se num conjunto de funções tais, que possibilitam aos nós Cliente processarem simulações em paralelo na mesma máquina virtual ou física, exigindo que se envie mais do que um indivíduo por pacote, utilizando *"threads"* para o efeito pretendido. Por outro lado, de forma a garantir compatibilidade e simplicidade para as diferentes configurações, manteve-se um modo de processamento sem *"threads"* proveniente da aplicação inicial em que apenas é lançada uma única simulação por processo de nó Cliente. A implementação com *"threads"* não está completa, restando por implementar o sistema que controla o número de *"threads"* a lançar. Para tal é necessário saber perante qualquer máquina, através da recolha de informação de *"hardware"*, quantas *"threads"* deverá lançar para que o sistema de processamento seja o mais optimizado possível. Tipicamente deverá lançar-se uma *"thread"* por processador lógico, admitindo a tecnologia usada hoje em dia.

Será de referir que a implementação das funções desta secção foram marioritáriamente obtidas a partir de funções da antiga aplicação, com a excepção da estrutura de multi-processamento, não tendo sido necessário realizar alterações à estrutura do código. O código relativo às funções da presente secção está detalhado nas alíneas 6, 7, 8, 9, 10 e 11 do Anexo I – Código de Funções.

4 Testes e Resultados Obtidos

4.1 Realização de Testes

Antes e durante a realização de testes foi necessário determinar objectivos estratégicos a atingir para que se possa concluir sobre eventuais resultados esperados. Foram então elaborados alguns planos de teste e determinados os factores condicionantes dos resultados destes testes através do detalhe da dinâmica de processamento da aplicação

4.1.1 Tempos de execução da "Aplicação Inicial"

Para que seja comparado todo o processo de evolução é necessário possuir elementos comparativos do tempo de execução da aplicação entregue. Esta, no seu estado inicial assumia os seguintes tempos de execução, ilustrados na Tabela 30, para os determinados valores de configuração abaixo indicados:

- ✔ Testes realizados com aplicação inicial configurados para:
  - 100 Gerações
  - Populações referidas nas colunas
- ✔ A máquina utilizada para este teste foi:
  - 1 x AMD Sempron(tm) Processor 2800+ (1.6GHz) 256 KB L2 (1GB RAM)

Resultados/População	50 Indivíduos	100 Indivíduos
Tempo de Processamento	125m49.647s	277m25.529s
Fitness	7.252E-05	8.339E-05

Tabela 30: Resultados da aplicação inicial (aplicação entregue)

4.1.2 Planos de Teste

Tomando em conta a arquitectura escolhida, cliente-servidor, constata-se que existe um limite de nós Cliente para um único nó Servidor. A fim de perceber até que ponto esse limite estaria próximo, bem como tentar perceber quais os factores mais importantes na configuração, procedeu-se à realização de testes nas seguintes situações:

- Uma só máquina, para determinar o limite de desempenho
- Muitas máquinas
- Poucas máquinas
- Máquinas com poucos processadores (1 processador)
- Máquinas com muitos processadores (2 ou mais)
- Muitas máquinas, umas com menos processadores que outras
- Muitas máquinas com mais processos que processadores lógicos
- Muitas máquinas com os mesmos processos que processadores lógicos

Nas situações acima procedeu-se de forma a tentar perceber, usando o resultado de *"fitness"* como elemento comparativo, qual a influência dos seguintes factores na alteração dos resultados obtidos:

- Muitas ou poucas Gerações.
- Maiores ou menores Populações.
- Tempos de processamento para as situações dos dois pontos anteriores sendo que, para o mesmo tempo determinar qual é a melhor combinação a fim de se aumentar a probabilidade de obter o melhor resultado.

4.1.3 Factores Condicionantes

Apesar de neste trabalho ser demonstrado um novo método de processamento que expande grandiosamente a capacidade de processamento da anterior aplicação é irrisório não considerar factores condicionantes que poderão afectar o desempenho da aplicação. Os seguintes factores alteram o desempenho de determinados componentes da aplicação que quando configurados correctamente proporcionam o melhor desempenho possível para a arquitectura escolhida:

- Número de nós por processador lógico.
- Nó com menor quantidade de memória RAM disponível pode criar situações limitativas por falta de memória e bloquear a aplicação.
- Estrutura da rede com congestionamento do tráfico em redes IP a partir do nó Servidor.

4.2 Configuração de Testes

Os testes realizados foram efectuados em ambientes totalmente abertos, sujeitos a intervenções humanas planeadas e não planeadas com *"software Open Source"*. Para os testes realizados foi utilizada uma plataforma de trabalho descrita no Anexo J – Plataforma de Trabalho.

4.2.1 Versões de UNIX Utilizadas

Na realização de todos os testes da aplicação em desenvolvimento neste trabalho foram utilizadas as seguintes distribuições de UNIX, sendo que as mais adaptadas, são as distribuições de Linux. Contudo, para objectivos profissionais mais exigentes da aplicação em desenvolvimento aconselha-se o uso de sistemas mais robustos como o AIX (da IBM) ou o Solaris (da SUN):

- ***Circuito 1***

✔ Linux (Ubunto) 2.6.17-11-generic #2 SMP Tue Mar 13 23:32:38 UTC 2007 i686 GNU/Linux
✔ Linux 2.6.18-1.2798.fc6xen #1 SMP Mon Oct 16 15:11:19 EDT 2006 i686 i686 i386 GNU/Linux

- ***Circuito 2***

✔ Linux (Ubunto) 2.6.17-11-generic #2 SMP Tue Mar 13 23:32:38 UTC 2007 i686 GNU/Linux
✔ Linux 2.6.18-1.2798.fc6xen #1 SMP Mon Oct 16 15:11:19 EDT 2006 i686 i686 i386 GNU/Linux
✔ Linux 2.6.21-1.3194.fc7 #1 SMP Wed May 23 22:35:01 EDT 2007 i686 i686 i386 GNU/Linux
✔ Linux 2.6.21-1.3228.fc7 #1 SMP Tue Jun 12 15:37:31 EDT 2007 i686 athlon i386 GNU/Linux
✔ Linux 2.6.22.1-27.fc7 #1 SMP Tue Jul 17 17:13:26 EDT 2007 i686 i686 i386 GNU/Linux

4.2.2 Versões de MPI Utilizadas

Das distribuições de MPI disponíveis, optou-se principalmente pelo uso da distribuição MPICH2. Embora as restantes distribuições aconselhadas na secção 7.10.12 - *Ambientes MPI Recomendados*, sejam igualmente suportadas e funcionais como a distribuição MPICH2, decidiu-se usar apenas um distribuição para efeitos comparativos nos testes da aplicação desenvolvida neste trabalho:

- ***Circuito 1 - Amplificador de Um Andar***

✔ MPICH2 – 1.0.4p1
✔ MPICH2 – 1.0.5p4

- ***Circuito 2 - Amplificador Cascode-Dobrado de Dois Andares***

✔ MPICH2 – 1.0.4p1
✔ MPICH2 – 1.0.5p4
✔ MPICH2 – 1.0.6p1

4.3 Resultados Obtidos

Todos os resultados apresentados nas seguintes secções são valores únicos e não resultam de médias de outros resultados. Os mesmos são provas escritas e documentadas da tentativa de cumprimento dos objectivos propostos na presente teste:

4.3.1 Circuito 1 - Amplificador de Um Andar

Embora se tenham realizado testes para o circuito da presente secção, este serviu apenas de cobaia na fase de implementação/teste inicial da aplicação desenvolvida. Os resultados obtidos para este circuito foram diferentes, nomeadamente os valores de *"fitness"*, tempos de processamento totais, objectivos do circuito entre outros, dos resultados do segundo circuito testado, detalhado na secção 4.3.2 - *Circuito 2 - Amplificador Cascode-Dobrado de Dois Andares*. Contudo o desempenho e o comportamento da aplicação neste primeiro circuito obedece às mesmas condições usadas no segundo circuito e detalhado na secção 4.3.2 - *Circuito 2 - Amplificador Cascode-Dobrado de Dois Andares*.

Do mesmo modo decidiu-se apresentar resultados temporais apenas para o segundo circuito com o objectivo de simplificar os testes e a objectividade dos mesmos. No entanto, fica documentado que foi possível executar a optimização com outro circuito mais simples e obter resultados que comprovam a validade dos testes realizados, como ilustrado na Tabela 31. Este teste foi realizado para uma população de 10 indivíduos e 100 gerações, pacotes de 4 indivíduos e apenas uma máquina de processamento igual à primeira máquina do primeiro ponto do Anexo A - Bateria de Testes (x86 32-bit).

Para validar os resultados dos testes realizados na presente secção foi necessário ter em conta e respeitar os assuntos discutidos na secção 3.4.3 - *Alterações na implementação dos algoritmos genéticos* sobre os ficheiros "circuit_main.c" e "circuit_objective.c", nomeadamente a forma de introduzir novos circuitos para a mesma aplicação sem necessitar de reformular o código todo, apenas alterando os dois ficheiros referidos anteriormente.

Tempo de Processamento	**--- (sem contabilização)**
Resultado de "Fitness"	**0.000048**
Total Current (uA)	1.94E+00 uA
Error (*Vout)	1.00E-03 %Vout
Output Swing (V)	7.70E-01 V
SetTime (nSec)	1.05E+03 nSec
Vo (V)	2.22E-16 V
Av Gain (db)	-1.37E+01 db
Noise ?	2.78E+00 ?
CCapArea (pf)	1.52E-01 pf

Tabela 31: Detalhes do processamento para um população de 10 indivíduos e 100 gerações

Os dados da Tabela 31 encontram-se dentro dos valores esperados e a sua divulgação serve apenas como prova de que a aplicação permite utilizar diferentes circuitos no processo de optimização.

4.3.2 Circuito 2 - Amplificador Cascode-Dobrado de Dois Andares

Os resultados dos testes realizados para o presente circuito foram tabelados numa bateria de testes detalhada no Anexo A - Bateria de Testes (x86 32-bit), com o principal objectivo de identificar a dinâmica de escalonamento da aplicação e de comparar os mesmos resultados com os da aplicação inicial.

Para a bateria de testes foram utilizadas as máquinas descritas no Anexo A - Bateria de Testes (x86 32-bit).

Na tabela da Tabela 32 são apresentados os primeiros testes de comparação de resultados obtidos com a aplicação, optimizador de processamento distribuído. Os primeiros testes têm como objectivo estudar a evolução da aplicação com o tamanhos diferentes de população e gerações. Como se pode verificar, existe um custo de processamento muito superior para as gerações do que para uma população muito grande. Esta evidência provém do facto de que para o mesmo número de simulações, obtém-se um valor de *"fitness"* em menor tempo numa configuração com uma população de 500 indivíduos e 100 gerações que numa configuração com uma população de 100 indivíduos e 500 gerações. Os valores de *"fitness"*

apresentam-se na segunda, quinta e oitava colunas das últimas quatro linhas da tabela apresentada na Tabela 32. Os tempos de processamento encontram-se nas colunas a seguir às referidas na frase anterior.

Indivíduos	100 Gerações			300 Gerações			500 Gerações		
50	78(1T)	28m53s (6h43m22s)	1N	96(5T)	1h26m33s (19h28m11s)	4N	96(8T)	2h25m53s (1d9h53m58s)	8N
100	99(2T)	53m14s (12h47m31s)	2N	103(6T)	2h43m27s (1d15h11m26s)	5N	101(9T)	4h32m58s (2d17h23m25s)	9N
500	106(3T)	3h17m42s (2d13h47m52s)	3N	107(7T)	9h35m16s (7d11h32m19s)	6N	---	~16h22m41s (~12d12h47m43s)	---
1000	113(4T)	6h39m47s (5d8h24m25s)	4N	---	~19h10m32s (~15d8h3m15s)	---	---	~1d8h45m22s (~25d16h37m49s)	---

Tabela 32: Verificação do efeito do tamanho da população/nº de gerações

Na tabela da Tabela 33 foram testados os efeitos que terão o número de máquinas presentes no processamento em paralelo. Verifica-se que quanto mais máquinas mais rápido se processa todo o trabalho. No entanto verifica-se também que este desempenho não é linear, sofrendo alguma entropia no aumento do desempenho para um maior número de máquinas. Este acontecimento deveu-se a diferentes situações:

- ✔ No primeiro e segundo teste da Tabela 33, realizado com duas e quatro máquinas respectivamente, foram utilizadas as duas máquinas e as quatro máquinas mais rápidas respectivamente.
- ✔ No último teste da Tabela 33 foram adicionadas outras máquinas mais lentas que as primeiras, sendo notório que o desempenho não progrediu de forma linear.

Máquinas em Paralelo	2			4			10		
	99(23T)	2h23m50s (9h21m24s)	23N	91(18T)	1h6m57s (8h19m45s)	18N	99(2T)	53m14s (12h47m31s)	2N

Tabela 33: Verificação do efeito do número de máquinas do em paralelo

Na tabela da Tabela 34 testou-se, à semelhança do que se testou na tabela da Tabela 33, a evolução do desempenho com o aumento de número de máquinas. No entanto, com o objectivo de perceber qual o melhor número de nós por máquina através dos testes da Tabela 34 fez-se variar também o número de processos por máquina. sendo que os valores apresentados facilmente identificam que deverá corresponder ao número médio de CPU's por

máquina. Nestes testes a evidência não se verifica em todos os casos devido ao reduzido número de máquinas para teste. Esta evidência é tanto maior quanto maior for o número de máquinas utilizadas para processar.

Relativamente aos valores de *"fitness"* existem algumas divergências causadas pelo comportamento do motor de algoritmos genéticos, que embora significativas nos testes realizados, são aceitáveis do ponto de vista da probabilidade que o mesmo motor poderá induzir na convergência dos valores de *"fitness"* para um determinado número de simulações.

No que diz respeito ao número de processos por máquina que melhor desempenho trás na contabilização do tempo total de optimização, não é possível indicar um número ideal nestes testes. Esta baixa diferenciação no número ideal de processos por máquina seria mais evidente para um número maior de máquinas. No entanto e considerando o teste com mais máquinas, é possível verificar que o número ideal de processos por máquina aproxima-se do número médio de processadores lógicos por máquina que no caso do teste das 10 máquinas é de 1,4 processadores por máquina e o melhor tempo é obtido utilizando 2 processos por máquina.

Processos	2 Máquinas			4 Máquinas			10 Máquinas		
1	82(24T)	2h48m35s (5h34m51s)	24N	93(19T)	1h35m21s (6h14m3s)	19N	115(14T)	49m39s (7h44m39s)	14N
2	89(25T)	2h19m43s (9h11m58s)	25N	100(20T)	1h8m25s (8h50m48s)	20N	87(15T)	42m42s (12h26m14s)	15N
5	93(26T)	2h21m12s (22h32m6s)	26N	87(21T)	1h6m28s (20h13m13s)	21N	97(16T)	44m20s (1d4h50m52s)	16N
10	97(27T)	2h15m2s (1d17h22m34s)	27N	90(22T)	1h8m24s (1d14h39m52s)	22N	91(17T)	1h12m18s (2d17h29m4s)	17N

Tabela 34: Verificação do efeito do número de processos (em cada máquina cliente)

Para detalhar as configurações usadas na bateria de testes do Anexo A - Bateria de Testes (x86 32-bit) iremos utilizar uma representação parcial da estrutura usada e ilustrada tabela da Tabela 35. Na primeira coluna desta tabela está representado o número de indivíduos utilizados em cada um dos testes, ou seja, um teste por cada linha e coluna únicos. Na mesma tabela estão representados apenas 4 testes, cada um deles para 100 gerações e respectivas populações de, 50, 100, 500 e 1000 indivíduos. Nestes testes foram utilizadas 10 máquinas e dois nós Cliente por cada máquina, mais um nó Servidor. Os pacotes foram configurados para transportarem 2 indivíduos de cada vez, querendo igualmente dizer que os nós Cliente recebem e processam dois indivíduos de cada vez.

Podemos analisar na Tabela 35 que os valores de *"fitness"* das optimizações são tanto mais óptimos quanto maior for a população a simular confirmando com a segunda coluna da Tabela 35. Este valor de *"fitness"* está não normalizado, tendo sido multiplicado por 1E05. A evolução para a arquitectura escolhida deverá apresentar um comportamento linear para uma quantidade de indivíduos/gerações muito alta. Os tempos de processamento totais devem-se ao facto de que para 10 máquinas e dois nós Cliente por máquina, temos um total de 20 nós Cliente. Assumindo os dois indivíduos por pacote, na primeira distribuição de pacotes aos nós Cliente são distribuídos 40 indivíduos. Para o primeiro e segundo teste da Tabela 35, a quantidade de indivíduos a simular é muito pequena para a quantidade de máquinas do ambiente, provocando muita irregularidade nos tempos de processamento. Assim sendo, a partir dos 500 indivíduos é notória a melhoria nos tempos de processamento sendo que podemos traduzir e comprovar estas melhorias na medida em que:

- Para 50 indivíduos e 100 gerações temos uma média de 350 milisegundos por simulação.
- Para 100 indivíduos e 100 gerações temos uma média de 320 milisegundos por simulação.
- Para 500 indivíduos e 100 gerações temos uma média de 240 milisegundos por simulação.
- Para 1000 indivíduos e 100 gerações temos uma média de 240 milisegundos por simulação.

Indivíduos	100 Gerações		
50	78(1T)	28m53s (6h43m22s)	1N
100	99(2T)	53m14s (12h47m31s)	2N
500	106(3T)	3h17m42s (2d13h47m52s)	3N
1000	113(4T)	6h39m47s (5d8h24m25s)	4N

Tabela 35: Explicação da Tabela da Bateria de Testes (Anexo A - Bateria de Testes (x86 32-bit))

É possível constatar algumas diferenças entre os resultados da aplicação inicial detalhados na Tabela 30 e os resultados do circuito da presente secção. Através do seguinte quadro ilustrado na Tabela 36 denotam-se algumas diferenças para o mesmo número de gerações, evidentes e que nos permitem tirar algumas conclusões:

Resultados/População	50 Indivíduos	100 Indivíduos
Tempo de Processamento (Aplicação Inicial)	125m49.647s ***(2h5m50s)***	277m25.529s ***(4h37m26s)***
Fitness (Aplicação Inicial)	7.252E-05	8.339E-05
Tempo de Processamento (Aplicação Actual)	***(28m53s)***	***(53m14s)***
Fitness (Aplicação Actual)	7.8E-05	9.9E-05

Tabela 36: Comparação de resultados entre a "Aplicação Actual" e a " Aplicação Inicial"

Podemos verificar na Tabela 36, que para um conjunto de 10 máquinas a "Aplicação Actual" produz resultados iguais ou melhores que os resultados da "Aplicação Inicial". Sendo que uma dessas 10 máquinas foi utilizada para os testes da "Aplicação Inicial", temos que nos testes apresentados na Tabela 36, a relação entre o número de máquinas necessárias numa aplicação local, comparado com outra versão da aplicação em modo distribuído/paralelo de forma a desempenhar o processamento da mesma quantidade de dados terá de ser inferior em aproximadamente 4 vezes em relação ao tempo total de processamento do modelo local. Assumindo uma evolução mais ou menos linear entre o número de indivíduos processados na "Aplicação Inicial" temos que o valor aproximado de tempo de processamento esperado para uma população de 500 indivíduos seria de 23 horas, 7 minutos e 10 segundos. Ou seja, a relação entre a "Aplicação Inicial" e a "Aplicação Actual" neste caso, idealizando os valores calculados, seria de 7 vezes. Como temos 10 máquinas para a "Aplicação Actual" podemos dizer que para os testes realizados obtivemos um desempenho da arquitectura de 70%.

Resumindo, temos que a "Aplicação Actual" para a optimização do circuito da presente secção, "Circuito 2" na arquitectura de máquinas utilizada, ou seja, 10 máquinas como especificado no Anexo A - Bateria de Testes (x86 32-bit), tem as seguintes funcionalidades/capacidades:

- ✔ Tempo médio aproximado de processamento de uma simulação: 240 milisegundos por simulação
- ✔ Eficiência de arquitectura aproximada em relação ao processamento num só nó, equiparada à "Aplicação Inicial": 70%

As conclusões tiradas na presente secção tiveram em conta alguns valores não regulares de *"fitness"* que, muitas vezes são muito mais altos para populações/gerações menores que em comparação com outras maiores.

4.3.2.1 Melhores Resultados (Circuito Óptimo)

Antes de dar por terminados os testes e a fim de solidificar os resultados já obtidos, realizou-se um último teste com mais uma máquina, 11 no total, uma maior população, 10 mil indivíduos, um menor número de gerações, apenas 10 e também um maior tamanho de pacote, 3 indivíduos por pacote:

- Para os testes de 11 máquinas foram usados os seguintes processadores:
  - 4 x Intel(R) Pentium(R) 4 CPU HT (3.00GHz) 2048KB L2 (512MB RAM)
  - 1 x AMD Sempron(tm) Processor 2800+ (1.6GHz) 256 KB L2 (1GB RAM)
  - 5 x Intel(R) Pentium(R) 4 CPU (1.70GHz) 256KB L2 (256MB RAM)
  - 1 x Intel(R) Core(TM)2 Duo CPU T7300 (2.0GHz) 4096KB L2 (512MB RAM)

O sistema operativo da máquina introduzida corre sobre "VMware Server", um *"software"* de virtualização. Deste modo é possível demonstrar a versatilidade da aplicação em ambientes completamente heterogéneos e até emulados no caso da máquina introduzida.

Neste teste foi possível desvendar alguns pormenores interessantes no sentido de permitirem uma melhor optimização da aplicação e consequentes melhores resultados no sentido de se obterem em menor tempo. Embora apenas se ilustre um teste como válido foram executadas várias instâncias de teste, com variações nas variáveis de população, tamanho de pacote e número de gerações, de forma concluir sobre os seguintes aspectos:

- ✔ Existe melhor desempenho quando se utiliza mais a capacidade da MTU (Maximum Transfer Unit) – testado com o tamanho de pacote – Considerações sobre a MTU: Tipicamente de 1500 bytes, podendo chegar a 9000 bytes para"jumbo-frames", como indicado no documento [9]. Contudo existem dois cabeçalhos a contabilizar para o tamanho total da mensagem de forma a que os dados transmitidos pelo MPI não superem na totalidade da mensagem, o valor da MTU. Este são o cabeçalho necessário para indexar o pacote IP e próprio cabeçalho incluído na mensagem MPI.
- ✔ A aplicação dá origem a melhores resultados quando se utiliza populações maiores, para a mesma quantidade de simulações totais – testado com a verificação de que para populações mais extensas o valor de *"fitness"* é mais alto para o mesmo número de

simulações, quando comparado com um teste de menor população e igual número de simulações, ou seja, mais gerações – Comprovado perante os valores do Anexo A - Bateria de Testes (x86 32-bit) e da Tabela 37.

- ✔ Existe melhor desempenho quando se utiliza o mesmo número de processos que o número de processadores lógicos existentes – testado através da relação entre o número de processos para o número de processadores lógicos – Comprovado através do mesmo número de iterações utilizadas entre o Anexo A - Bateria de Testes (x86 32-bit) e a Tabela 37, e o menor tempo registado no caso do exemplo apresentado na presente secção ilustrado na Tabela 37.
- ✔ Fácil visualização que para populações elevadas, a evolução do valor de *"fitness"* estagna após o processamento das primeiras simulações – testado através da ausência de evolução durante algumas gerações em diversos testes. Antes de estagnar existe evoluções em todas as gerações. Quando o processo de evolução estagna, não se verificam mais evoluções até ao fim das gerações configuradas. Este ponto visa apenas clarificar que num tipo de configuração em que existe uma população muito grande é muito mais fácil e rápido perceber quando é que o cromossoma utilizado para simular o circuito, está ou não a produzir os seus melhores resultados – Contactado na evolução de dois testes reproduzíveis, um com uma população muito elevada e poucas gerações, outro com uma população muito pequena mas com um elevado número de gerações.

Finalmente o melhor resultado obtido, em detalhe na Figura 37 obteve-se quando se usou uma **população de 10 mil** indivíduos e **10 gerações**. Aumentou-se o tamanho do pacote para **3 indivíduos** juntou-se uma máquina bastante mais recente para verificar a longevidade da compatibilidade da aplicação:

Tempo de Processamento	**5h 34m 57s**
Resultado de "Fitness"	**0.000127**
Total Current (uA)	7.74E+00 uA
Error (*Vout)	4.00E-05 *Vout
Output Swing (V)	6.99E-01 V
SetTime (nSec)	1.97E+01 nSec
Vo (V)	5.00E-01 V
Av Gain (db)	8.10E+01 db
Noise ?	4.73E+00 ?
CCapArea (pf)	8.00E+00 pf

Tabela 37: Melhor resultado - 10000 indivíduos com 10 gerações apenas

Os resultados obtidos na Tabela 37 foram os melhores resultados obtidos em todos os testes realizados neste trabalho. Comparando o tempo de processamento e o resultado de *"fitness"* para o mesmo número de iterações/simulações temos que: para o anterior melhor teste antes realizado e ilustrado na Tabela 35, com 1000 indivíduos de população e 100 gerações, o tempo de processamento foi de 6 horas, 38 minutos e 47 segundos com um valor de *"fitness"* de 0.000113; para os resultados comparativos da Tabela 37, o tempo de processamento para o mesmo número de iterações/simulações, 10000 indivíduos de população e apenas 10 gerações foi de 5 horas, 34 minutos e 57 segundos, com um valor de *"fitness"* de 0.000127.

Esta optimização foi considerada a optimização "óptima" do "Circuito 2" em questão, *"Circuito 2 - Amplificador Cascode-Dobrado de Dois Andares"* para o cromossoma usado. Os restantes detalhes sobre as dimensões dos transístores do circuito optimizado nesta secção estão na *"Netlist"* do circuito anexado relativo à Tabela 37, estão presentes no Anexo G - Netlist do Melhor Resultado.

5 CONCLUSÕES

Apesar de algo demorada, a presente tese cumpriu com resultados evidentes os objectivos propostos. Foi possível demonstrar que é possível transformar uma aplicação de processamento local numa aplicação de processamento em paralelo, com uma ou mais do que uma máquina física. Foi possível demonstrar também que é possível renovar as técnicas de processamento, utilizando recursos que muitas das vezes julgamos já sem utilidade, como por exemplo, computadores pessoais antigos. De forma mais conclusiva e apenas referindo simples pontos, foi conseguido:

- ✔ Menores tempos de processamento como é possível verificar no Anexo A - Bateria de Testes (x86 32-bit), Tabela 37 e Tabela 30.
- ✔ Reutilização de *"hardware"* que está funcionalmente desabilitado, por falta de desempenho.
- ✔ Possibilidade de alcançar melhores optimizações para o mesmo período de tempo como indicado no Anexo A - Bateria de Testes (x86 32-bit), Tabela 37 e Tabela 30.
- ✔ Reaproveitar código já desenvolvido da aplicação entregue de forma a exemplificar como paralelizar uma aplicação baseada em rotinas de elevada recursividade.

5.1 Resumo Abstracto

Dos objectivos propostos foi possível cumprir dentro de uma área de resultados consideravelmente aceitáveis, com as metas traçadas para esses mesmos objectivos, ao ponto de ser possível reclamar os seguintes pontos sem quaisquer dúvidas:

Objectivos propostos:

A) Possibilidade de optimizar circuitos mais complexos.
B) Diminuir abruptamente o tempo de uma optimização.
C) Melhorar as possibilidades de obter um melhor resultado em tempo útil.

Dos quais é possível conferir que:

- ✔ Objectivo A – Provado através das máquinas usadas para os testes (secção 4.3.2.1 - *Melhores Resultados (Circuito Óptimo)*).
- ✔ Objectivo B – Visivelmente provado nos resultados da Tabela 36.
- ✔ Objectivo C – Foi possível passar da optimização do "circuito 1" (menos complexo) para o "Circuito 2" (mais complexo) com apenas alguns diferenças de implementação no ficheiro que terá sempre de ser trocado para cada circuito "circuit_objecive.c".
- ✔ Objectivo D – Através do aumento do desempenho é possível chegar a melhores resultados num menor período de tempo para optimizações de circuitos analógicos.

Utilizando uma breve descrição como a Figura 38:

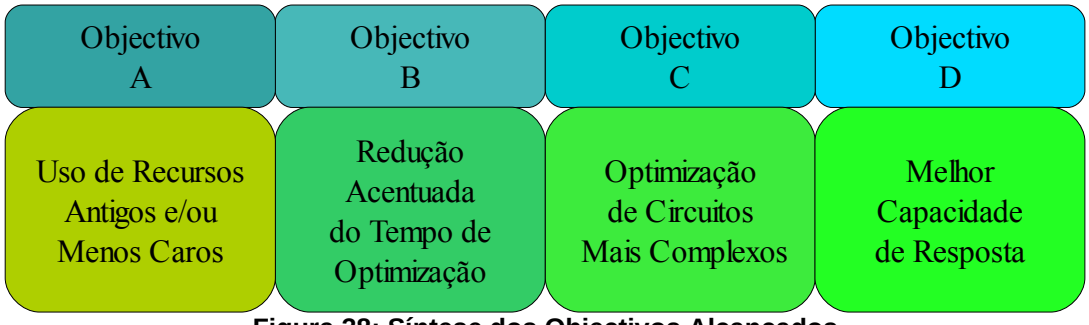

Figura 38: Síntese dos Objectivos Alcançados

6 Trabalho Futuro

6.1 Visual Interactivo

Para melhor administrar todo o ambiente, a implementação de um interface gráfico será uma possível opção futura na continuação do desenvolvimento da aplicação deste trabalho.

Esta interface gráfica pode ser facilmente implementada utilizando a linguagem C ou Java, sendo que nesta última nada de novo é necessário acrescentar ao presente código da aplicação, mantendo-a inalterada e independente. Utilizando o código Java é possível por exemplo monitorizar um ficheiro de debug da aplicação, como os actuais "printf(...)" de toda a aplicação e trabalhar a informação contida nesse ficheiro. Outros comandos de chamada e administração do ambiente MPI podem ser chamados facilmente através de um *"script"* ou directamente numa *"shell"*.

Existem ainda outras formas mais eficientes de implementar uma interface gráfica que não altere muito o conteúdo da aplicação já desenvolvida mas, por outro lado, a complexidade na estrutura de controlo gráfica irá aumentar muito com a implementação de uma interface gráfica em que sejam necessárias validações do conteúdo de mensagens e de estados da comunicação passiva ou activa. Esta alternativa será mais robusta e concisa embora requeira um conhecimento/estudo maior de como funciona a aplicação.

6.2 Estrutura Aplicacional

Outras opções de implementação possíveis serão a nível estrutural, no código da aplicação. Como exemplos temos as seguintes secções, 6.2.1 – *Arquitecturas de Comunicação*, 6.2.2 – *Evolução da Camada MPI* e 6.2.3 – *Exploração e Evolução do NGSPICE Implementado*.

6.2.1 Arquitecturas de Comunicação

Actualmente a arquitectura cliente-servidor é uma arquitectura muito conhecida e provavelmente a mais simples e pragmática de implementar. A escolha desta arquitectura para a presente aplicação rebateu apenas na facilidade de implementação, sendo que existem outras arquitecturas possíveis de implementar e que fornecem melhor desempenho à aplicação.

Focando o limite de um sistema relativamente complexo para o cenário da rede Internet de hoje em dia, é facilmente visível a falha da arquitectura escolhida. Para o escalonamento da aplicação é necessário implementar um sistema inteligente que se adapte à estrutura de rede no qual irá ser lançado consoante um conjunto de pormenores e informações simples, fornecidas pelo utilizador. Esta condição de adaptado deve ser tão ou mais parecida com um algoritmo que visa estabelecer o equilíbrio entre a capacidade de fluxo de dados do troço de rede e os caudais a abrir nesses mesmos troços de forma a não estrangular a rede em nenhum ponto. É por isso necessária a condição de vários nós Servidores interagirem entre si de forma cooperativa e activa. Desta forma o nó poderá configurar-se de forma a lançar ou destruir nós Cliente à sua volta, consoante os seus graus de actividade ou inactividade respectivamente.

Para sistemas pequenos e homogéneos, pouco mais do que uma arquitectura cliente-servidor é necessária a fim de obter um melhor desempenho, sendo que para este caso, melhorar é uma condição pouco provável.

Para sistemas médios/grandes e homogéneos, um sistema de um ou mais servidores resolverá o problema de desempenho, sendo que, poucas alterações serão necessárias a sua adaptação à aplicação já implementada. Apenas terá de ser estudada a distribuição de carga de forma passiva ou activa sobre a configuração de rede nesse conjunto de máquinas de forma a obter o melhor desempenho possível. Por outro lado poderão ainda ser utilizadas outras aplicações para realizar esta distribuição de carga, como por exemplo o “Globus Toolkit”.

Finalmente para sistemas heterogéneos de qualquer tamanho mas não abrangendo o mundo Internet, poderá optar-se pela solução já implementada com modificações na acessibilidade do nó Servidor que terá um requisito de disponibilidade superior e imperativo para que todo o processo de optimização finalize. Este fim poderá ser alcançado, usando

múltiplos Servidores de comunicação cooperativa que dividem trabalho entre si. A partir deste ponto toda a comunicação entre cliente-servidor deve ser ponderada com a capacidade de processamento de cada nó Cliente.

6.2.2 Evolução da Camada MPI

Acompanhando a secção anterior, 6.2.1 - *Arquitecturas de Comunicação* e compreendendo o funcionamento da camada MPI na secção 2.4.2.1 - *Processamento em Paralelo Local*, é perceptível que a camada MPI terá como condição necessária a sua evolução, acompanhando a estrutura de rede e suas principais directivas de funcionamento.

Transladando as evoluções possíveis da camada de MPI, será possível encaixar novidades como:

- Clientes com "login" e "logout" nos Servidores.
- Clientes com mobilidade e em processamento entre Servidores.
- Reinicialização de Cliente sem necessidade de retransmissão de dados.
- Configuração de Grupos para executar diferentes optimizações dentro da mesma rede.

6.2.3 Exploração e Evolução do NGSPICE Implementado

O facto de não se ter optado pela exploração do pacote NGSPICE a fim de focar melhor os objectivos deste trabalho, disponibiliza automaticamente uma oportunidade de desenvolvimento e aperfeiçoamento desse mesmo pacote de *"software"*, tendo este um papel importante no desempenho da aplicação desenvolvida neste trabalho.

6.3 Evolução Tecnológica

Existem pelo menos dois caminhos de evolução muito importantes que, quando desenvolvidos especificamente para o código da aplicação, habilitam tecnologicamente a aplicação a dispor de um poder de processamento superior, quando este existe. Desempenhando todo o seu trabalho mais rapidamente é ainda possível dispor ao mesmo tempo de um suporte mais robusto e recente.

Estes dois caminhos surgem através da divisão do código em processos separados e independentes, *"threads"*, que desbloqueiam o processamento lógico sequencial e a simples optimização de código que eliminará ocasiões em que tecnologias diferentes de CPU's (Central Processing Unit) desempenham de forma diferente.

Porém, existe ainda outro grande passo tecnológico que tornará possível o aumento da velocidade de processamento aplicacional e o reaproveitamento de recursos contidos numa única máquina. Este avanço baseia-se na implementação de código útil, tipicamente para processamento em CPU, modificado para ser processado em GPU (Graphic Processing Unit), cuja arquitectura de processamento está muito mais optimizada para cálculos matriciais com operações de soma, multiplicação, divisão e subtracção, ou ainda cálculos de números não inteiros, ordenações, procuras, entre muitos outros em constante desenvolvimento.

6.3.1 Processos *Multi-Threaded*

A separação de porções do código, quando possíveis de paralelizar e em que sejam evidentes ciclos independentes ou funções pouco utilizadas, fornece vantagens notórias no desempenho da aplicação e tornam mais simples a organização estrutural do programa. A estes tipos de componentes de processamento, controlados por um processo aplicacional chamam-se *"threads"*, transformando aplicações *"single-threaded"* em aplicações *"multi-threaded"* ou aplicações de processamento múltiplo.

A aplicação desenvolvida neste trabalho possui modificações experimentais, numa porção de código, preparadas para trabalhar em conjuntos de *"threads"* configuráveis,

embora ainda falte completar a sua implementação, que paralelizam a gestão do lançamento de simulações nos nós Cliente.

É ainda evidente que na implementação de componentes adicionais, a construção de uma *"thread"* de gestão para cada componente, seja também um caminho importante a tomar em conta.

6.3.2 Implementação de Processamento Aplicacional de CPU em GPU

Um nova tecnologia em crescente desenvolvimento que permite transportar código semântico, normalmente executado no CPU, para o GPU. Este último possui de uma arquitectura de processamento simples, e não permite executar cálculos muito complexos em poucos ciclos de processamento. Contudo possui uma grande capacidade de processamento e está optimizado para execução em paralelo ou optimizado para processamento em multi-tarefa. Ao libertar algum processamento simples do CPU, que ocupa valiosos ciclos de processamento úteis por exemplo no controlo de procedimentos, executando-os no GPU, a aplicação que dependia apenas de um componente de processamento passa agora a dispor de dois motores de processamento lógico, completamente distintos do ponto de vista da sua arquitectura. Este factor favorece não só a capacidade de implementação de código e processamento em paralelo mas também porque, ao tratarem-se de dois componentes de arquitecturas diferentes, a sua evolução é diferente ao longo do tempo e isso permite obter vantagens a nível de optimização nas aplicações dos nossos dias.

A aplicação desenvolvida neste trabalho possui código capaz de ser transportado para execução em GPU. Estas linhas de código apresentam nomeadamente muitos cálculos repetidos ou uma baixa complexidade de cálculo como somas, subtracções, multiplicações, operações matriciais, ordenações, procuras, etc.

A identificação correcta do código a transportar e o cálculo da viabilidade da solução exige que sejam conhecidas as operações a executar no GPU e quais as dimensões da estrutura em paralelo a implementar, para que o processamento no GPU seja optimizado para a sua memória gráfica disponível, bem como a sua capacidade de processamento.

Existem duas tecnologias que permitem a implementação desta ideia de forma simples e rápida: CUDA ("Compute Unified Device Architecture") da empresa NVIDIA e o CTM ("Close-to-Metal") da empresa ATI/AMD, ou recentemente chamado de CAL ("Compute Abstraction Layer"). Estes dois conjuntos de bibliotecas preparadas para aceder directamente aos GPU's das respectivas marcas, disponibilizam funções simples que replicam funções tipicamente executadas em CPU para serem executadas em GPU. As funções são chamadas em linguagem C de forma transparente em relação ao código chamado e executado no GPU, produzindo o efeito desejado sem que o programador se preocupe com a gestão interna de execução no GPU. Preferencialmente, a plataforma que mais se ajustará no futuro a uma eventual aplicação neste trabalho, será a ideia da NVIDIA, devido aos seus desenvolvimentos recentes em bibliotecas especializadas para processamento de sinais, vectores e matrizes.

Figura 39: Logotipo "Cuda Zone" da NVIDIA

6.3.3 Optimização do Código

Apesar de todos os aumentos de desempenho da aplicação apresentada e desenvolvida neste trabalho, ficam ainda por desenvolver e optimizar muitas lacunas da aplicação. Estas, alheias ao utilizador comum, poderão melhorar substancialmente determinados processos cíclicos em que a redução de apenas meio segundo ou mesmo alguns milésimos de segundo num ciclo, reduzem em minutos ou mesmo horas o tempo total de optimização do circuito, dependendo do tamanho população e do número de gerações pretendidas.

7 Anexos e Bibliografia

Para efeitos de pesquisas, estudos e compreensão de alguns assuntos relacionados com a aplicação desenvolvida neste trabalho e ainda sobre todo o conceito implementado na distribuição em paralelo de componentes aplicativos, foram consultados os seguintes documentos:

- MPICH2 - User's Guide [http]
- MPICH2 - Installer's Guide [http]
- Open MPI - [http://www.open-mpi.org/]
- LAM/MPI - [http://www.lam-mpi.org/]
- Fedora Linux - [http://fedoraproject.org/]
- Ubunto Linux - [http://www.ubuntu.com/]
- Funções MPI - [http]

Informação sobre os dois pacotes de *"software"* mais conhecidos para paralelismo/processamento de código em GPU ("Graphical Processing Unit") que geralmente é executado em CPU ("Central Processing Unit"), CUDA ("Compute Unified Device Architecture") da empresa NVIDIA e o CTM ("Close To Metal") da empresa AMD/ATI, ou recentemente chamado de CAL ("Compute Abstraction Layer"):

- CUDA - http://developer.nvidia.com/object/cuda.html
- CTM Press News - http://www.amd.com/us-en/Corporate/VirtualPressRoom/0,,51_104_543~114147,00.html
- CTM Guide - http://ati.amd.com/companyinfo/researcher/Documents.html

Estes últimos componentes não foram desenvolvidos na aplicação apresentada.

7.1 Anexo A – Bateria de Testes (x86 32-bit)

- Para os seguintes testes foram utilizadas as seguintes máquinas:
  - Para os testes de 10 máquinas foram usados os seguintes processadores:
    - 4 x Intel(R) Pentium(R) 4 CPU HT (3.00GHz) 2048KB L2 (512MB RAM)
    - 1 x AMD Sempron(tm) Processor 2800+ (1.6GHz) 256 KB L2 (1GB RAM)
    - 5 x Intel(R) Pentium(R) 4 CPU (1.70GHz) 256KB L2 (256MB RAM)
  - Para os testes de 4 máquinas foram usados os seguintes processadores:
    - 3 x Intel(R) Pentium(R) 4 CPU HT (3.00GHz) 2048KB L2 (512MB RAM)
    - 1 x AMD Sempron(tm) Processor 2800+ (1.6GHz) 256 KB L2 (1GB RAM)
  - Para os testes de 2 máquinas foram usados os seguintes processadores:
    - 1 x Intel(R) Pentium(R) 4 CPU HT (3.00GHz) 2048KB L2 (512MB RAM)
    - 1 x AMD Sempron(tm) Processor 2800+ (1.6GHz) 256 KB L2 (1GB RAM)

Resultado da respectiva simulação abaixo indicada como "**1T**"

- Verificação do efeito do tamanho da população/nº de gerações
  - Com todas as máquinas (2 processos/máquina e 2 indivíduos/processo)

Indivíduos	100 Gerações			300 Gerações			500 Gerações		
50	78(1T)	28m53s (6h43m22s)	1N	96(5T)	1h26m33s (19h28m11s)	4N	96(8T)	2h25m53s (1d9h53m58s)	8N
100	99(2T)	53m14s (12h47m31s)	2N	103(6T)	2h43m27s (1d15h11m26s)	5N	101(9T)	4h32m58s (2d17h23m25s)	9N
500	106(3T)	3h17m42s (2d13h47m52s)	3N	107(7T)	9h35m16s (7d11h32m19s)	6N	---	~16h22m41s (~12d12h47m43s)	---
1000	113(4T)	6h39m47s (5d8h24m25s)	4N	---	~19h10m32s (~15d8h3m15s)	---	---	~1d8h45m22s (~25d16h37m49s)	---

Indivíduos	800 Gerações			1000 Gerações		
50	106(10T)	3h53m21s (2d6h30m26s)	10N	99(12T)	4h50m37s (2d19h29m6s)	12N
100	111(11T)	7h8m9s (4d6h3m20s)	11N	113(13T)	9h6m46s (5d11h44m42s)	13N
500	---	~1d1h41m20s (~19d13h27m20s)	---	---	~1d8h48m22s (~25d6h1m37s)	---
1000	---	~2d3h22m41s ~ (40d2h23m2s)	---	---	~2d17h36m43s (~51d18h21m19s)	

Netlist da respectiva optimização abaixo indicada como "**13N**"

Fitness normalizado da respectiva optimização (factor de 100000)

- Verificação do efeito do número de máquinas do cluster/em paralelo
  - Com o número de indivíduos/gerações, para os quais se obtiveram melhores resultados (fitness) – (100 indivíduos / 100 gerações)
  - Com 2 processos/máquina e 2 indivíduos/processo

Máquinas em Paralelo	2			4			10		
	99(23T)	2h23m50s (9h21m24s)	23N	91(18T)	1h6m57s (8h19m45s)	18N	99(2T)	53m14s (12h47m31s)	2N

- Verificação do efeito do número de processos (em cada máquina cliente)
  - Com o número de indivíduos/gerações e número de máquinas , para os quais se obtiveram melhores resultados – (100 indivíduos / 100 gerações)
  - Com 1 indivíduos/processo

Processos	2 Máquinas			4 Máquinas			10 Máquinas		
1	82(24T)	2h48m35s (5h34m51s)	24N	93(19T)	1h35m21s (6h14m3s)	19N	115(14T)	49m39s (7h44m39s)	14N
2	89(25T)	2h19m43s (9h11m58s)	25N	100(20T)	1h8m25s (8h50m48s)	20N	87(15T)	42m42s (12h26m14s)	15N
5	93(26T)	2h21m12s (22h32m6s)	26N	87(21T)	1h6m28s (20h13m13s)	21N	97(16T)	44m20s (1d4h50m52s)	16N
10	97(27T)	2h15m2s (1d17h22m34s)	27N	90(22T)	1h8m24s (1d14h39m52s)	22N	91(17T)	1h12m18s (2d17h29m4s)	17N

O valor de tempo entre parêntesis resulta da soma dos tempos de processamento de todas as simulações em todos os nós Cliente. De forma equivalente poderá interpretar-se este tempo como o tempo total de processamento de toda a optimização mas executando num só CPU lógico. Contudo não deverá ser obtido necessariamente pela multiplicação do tempo real com o número de CPUs lógicos. Isto porque caso existam mais processos que CPUs lógicos (é o caso da presente bateria de testes) o tempo final de todos os nós Cliente ainda será maior porque os dados de dois processos para 1 CPU lógico demoram sempre mais tempo que o mesmo conjunto de dados num só processo para 1 CPU lógico.

1T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000078 DONE

Total Current (uA)- | Error (*Vout)+ | Output Swing (V)- | SetTime (nSec)- | Vo (V)- | Av Gain (db)- | Noise ? | CCapArea (pf) |
5.66E+00 uA| 4.00E-05 *Vout| 7.26E-01 V| 3.20E+01 nSec| 5.00E-01 V| 7.88E+01 db| 6.21E+00 ?| 8.00E+00 pf|

2T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000099 DONE

Total Current (uA)- | Error (*Vout)+ | Output Swing (V)- | SetTime (nSec)- | Vo (V)- | Av Gain (db)- | Noise ? | CCapArea (pf) |
5.14E+00 uA| 4.00E-05 *Vout| 8.35E-01 V| 2.53E+01 nSec| 5.00E-01 V| 7.88E+01 db| 5.21E+00 ?| 8.00E+00 pf|

3T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000106 DONE

Total Current (uA)- | Error (*Vout)+ | Output Swing (V)- | SetTime (nSec)- | Vo (V)- | Av Gain (db)- | Noise ? | CCapArea (pf) |
6.90E+00 uA| 4.00E-05 *Vout| 7.83E-01 V| 2.35E+01 nSec| 5.00E-01 V| 8.20E+01 db| 6.52E+00 ?| 8.00E+00 pf|

4T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000113 DONE

Total Current (uA)- | Error (*Vout)+ | Output Swing (V)- | SetTime (nSec)- | Vo (V)- | Av Gain (db)- | Noise ? | CCapArea (pf) |

7.06E+00 uA| 4.00E-05 *Vout| 7.13E-01 V| 2.22E+01 nSec| 5.00E-01 V| 7.42E+01 db| 4.27E+00 ?| 8.00E+00 pf|

5T
-- 300 # 300 --
Master_Alg_Gen_Process max_fit_total: 0.000096 DONE

Total Current (uA)- | Error (*Vout)+ | Output Swing (V)- | SetTime (nSec)- | Vo (V)- | Av Gain (db)- | Noise ? | CCapArea (pf) |
5.18E+00 uA| 4.00E-05 *Vout| 7.73E-01 V| 2.61E+01 nSec| 5.00E-01 V| 7.51E+01 db| 2.61E+00 ?| 8.00E+00 pf|

6T
-- 300 # 300 --
Master_Alg_Gen_Process max_fit_total: 0.000103 DONE

Total Current (uA)- | Error (*Vout)+ | Output Swing (V)- | SetTime (nSec)- | Vo (V)- | Av Gain (db)- | Noise ? | CCapArea (pf) |
6.84E+00 uA| 4.00E-05 *Vout| 7.69E-01 V| 2.43E+01 nSec| 5.00E-01 V| 8.14E+01 db| 4.37E+00 ?| 8.00E+00 pf|

7T
-- 300 # 300 --
Master_Alg_Gen_Process max_fit_total: 0.000107 DONE

Total Current (uA)- | Error (*Vout)+ | Output Swing (V)- | SetTime (nSec)- | Vo (V)- | Av Gain (db)- | Noise ? | CCapArea (pf) |
5.53E+00 uA| 4.00E-05 *Vout| 7.04E-01 V| 2.34E+01 nSec| 5.00E-01 V| 9.61E+01 db| 3.78E+00 ?| 8.00E+00 pf|

8T
-- 500 # 500 --
Master_Alg_Gen_Process max_fit_total: 0.000096 DONE

Total Current (uA)- | Error (*Vout)+ | Output Swing (V)- | SetTime (nSec)- | Vo (V)- | Av Gain (db)- | Noise ? | CCapArea (pf) |
6.54E+00 uA| 4.00E-05 *Vout| 7.35E-01 V| 2.62E+01 nSec| 5.00E-01 V| 8.23E+01 db| 4.99E+00 ?| 8.00E+00 pf|

9T
-- 500 # 500 --
Master_Alg_Gen_Process max_fit_total: 0.000101 DONE

Total Current (uA)- | Error (*Vout)+ | Output Swing (V)- | SetTime (nSec)- | Vo (V)- | Av Gain (db)- | Noise ? | CCapArea (pf) |
6.74E+00 uA| 4.00E-05 *Vout| 7.11E-01 V| 2.48E+01 nSec| 5.00E-01 V| 8.24E+01 db| 6.36E+00 ?| 8.00E+00 pf|

10T
-- 800 # 800 --
Master_Alg_Gen_Process max_fit_total: 0.000106 DONE

Total Current (uA)- | Error (*Vout)+ | Output Swing (V)- | SetTime (nSec)- | Vo (V)- | Av Gain (db)- | Noise ? | CCapArea (pf) |
6.96E+00 uA| 4.00E-05 *Vout| 6.54E-01 V| 2.35E+01 nSec| 5.00E-01 V| 8.03E+01 db| 5.14E+00 ?| 8.00E+00 pf|

11T

-- 800 # 800 --
Master_Alg_Gen_Process max_fit_total: 0.000111 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
7.76E+00 uA	4.00E-05 *Vout	8.04E-01 V	2.25E+01 nSec	5.00E-01 V	6.71E+01 db	4.30E+00 ?	8.00E+00 pf

12T
-- 1000 # 1000 --
Master_Alg_Gen_Process max_fit_total: 0.000099 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
5.61E+00 uA	4.00E-05 *Vout	7.86E-01 V	2.52E+01 nSec	5.00E-01 V	8.23E+01 db	3.75E+00 ?	8.00E+00 pf

13T
-- 1000 # 1000 --
Master_Alg_Gen_Process max_fit_total: 0.000113 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
6.84E+00 uA	4.00E-05 *Vout	7.46E-01 V	2.22E+01 nSec	5.00E-01 V	7.77E+01 db	6.08E+00 ?	8.00E+00 pf

14T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000115 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
6.38E+00 uA	4.00E-05 *Vout	6.76E-01 V	2.18E+01 nSec	5.00E-01 V	7.53E+01 db	4.87E+00 ?	8.00E+00 pf

15T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000087 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
4.94E+00 uA	4.00E-05 *Vout	8.39E-01 V	2.88E+01 nSec	5.00E-01 V	7.42E+01 db	3.26E+00 ?	8.00E+00 pf

16T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000097 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
6.68E+00 uA	4.00E-05 *Vout	6.89E-01 V	2.56E+01 nSec	5.00E-01 V	8.42E+01 db	6.02E+00 ?	8.00E+00 pf

17T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000091 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
7.30E+00 uA	4.00E-05 *Vout	7.75E-01 V	2.75E+01 nSec	5.00E-01 V	7.91E+01 db	5.90E+00 ?	8.00E+00 pf

18T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000091 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
7.06E+00 uA	4.00E-05 *Vout	7.63E-01 V	2.73E+01 nSec	5.00E-01 V	8.19E+01 db	3.79E+00 ?	8.00E+00 pf

19T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000093 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
5.97E+00 uA	4.00E-05 *Vout	7.87E-01 V	2.69E+01 nSec	5.00E-01 V	8.95E+01 db	3.68E+00 ?	8.00E+00 pf

20T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000100 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
6.53E+00 uA	4.00E-05 *Vout	6.82E-01 V	2.51E+01 nSec	5.00E-01 V	9.07E+01 db	4.19E+00 ?	8.00E+00 pf

21T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000087 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
7.46E+00 uA	4.00E-05 *Vout	8.36E-01 V	2.87E+01 nSec	5.00E-01 V	7.68E+01 db	5.15E+00 ?	8.00E+00 pf

22T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000090 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
6.04E+00 uA	4.00E-05 *Vout	6.08E-01 V	2.79E+01 nSec	5.00E-01 V	9.15E+01 db	5.13E+00 ?	8.00E+00 pf

23T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000099 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
6.45E+00 uA	4.00E-05 *Vout	7.15E-01 V	2.51E+01 nSec	5.00E-01 V	8.02E+01 db	3.91E+00 ?	8.00E+00 pf

24T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000082 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
7.51E+00 uA	4.00E-05 *Vout	7.65E-01 V	3.04E+01 nSec	5.00E-01 V	8.52E+01 db	4.69E+00 ?	8.00E+00 pf

25T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000089 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
6.73E+00 uA	4.00E-05 *Vout	5.94E-01 V	2.80E+01 nSec	5.00E-01 V	8.24E+01 db	6.57E+00 ?	8.00E+00 pf

26T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000093 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
7.62E+00 uA	4.00E-05 *Vout	7.82E-01 V	2.69E+01 nSec	5.00E-01 V	7.91E+01 db	4.13E+00 ?	8.00E+00 pf

27T
-- 100 # 100 --
Master_Alg_Gen_Process max_fit_total: 0.000097 DONE

Total Current (uA)-	Error (*Vout)+	Output Swing (V)-	SetTime (nSec)-	Vo (V)-	Av Gain (db)-	Noise ?	CCapArea (pf)
7.20E+00 uA	4.00E-05 *Vout	6.71E-01 V	2.58E+01 nSec	5.00E-01 V	8.32E+01 db	5.80E+00 ?	8.00E+00 pf

7.2 Anexo B – Ficheiro de Configuração (ngaEDA.ini)

Cada etiqueta dentro de parêntesis rectos serve de menu para as opções que se seguem dentro desta:

- [ngaeda] - (definições do programa) – Não usado
  - log = "ngaEDA.log" - (ficheiro de LOG do programa) – Não usado
- [ngspice] - (parâmetros do NGSPICE) – Não usado
- [MPI] - (parâmetros do MPI)
  - popsize = 2 - (população máxima a processar por nó Cliente)
  - autothread = 1 - (processamento em "Multi" ou "Single-Thread")
  - mastermsg = 2 - (nível de output de mensagens do nó Servidor)
  - slavemsg = 1 - (nível de output de mensagens dos nós Cliente)
- [ga] - (parâmetros dos algoritmos genéticos)
  - ind = "../cfg/mespcomp.ind" - (parâmetros dos objectivos)
  - crm = "../cfg/mespcomp.crm" - (parâmetros dos limites dos cromossomas)
  - gen = 100 - (número de gerações)
  - pop = 1000 - (número de indivíduos da população)
  - roleta = 0 - (selecção por método de roleta ou não)
  - rank = 1 - (ordenação de resultados, baseado no fitness)
  - elitista = 1 - (selecção elitista)
  - mutvar = 0 - (tem ou não mutação variável)
  - pc = 0.76 - (taxa/probabilidade de cross-over)
  - pm = 1 - (taxa/probabilidade de mutação)
- [circuit] - (parâmetros do Circuito)
  - ininet = "../sim/sim_ngspice_tt.sp" - (parâmetros da simulação)
  - optnet = "../opt/debug_netlist_" - (output para debug) – Não usado
  - optres = "../opt/debug_optdata.dat" - (output para debug) – Não usado

Um exemplo da chamada de um valor contido no anterior ficheiro de configuração será chamado do seguinte modo:

```
{...}
initSettings = iniparser_load("./ngaEDA.ini");
autothread = iniparser_getint(initSettings, "MPI:autothread", 1);
{...}
```

A variável "initSettings" guarda um apontador para o ficheiro de configuração "ngaEDA.ini" através da função *iniparser_load(..)*. De seguida é possível ler um atributo, do ficheiro através da função *iniparser_getint(...)*, que neste caso é um inteiro. O primeiro argumento da anterior função aponta para de onde irá ser feita a leitura, o segundo identifica qual o atributo a ler dentro do ficheiro e o último, caso o anterior atributo não exista, o valor a retornar caso esse atributo não exista.

7.3 Anexo C – Administração do MPICH2

Exemplos de administração para uma rede de 5 máquinas:

```
[dga@pvm5 ~]$ mpdringtest 10
time for 10 loops = 0.0673379898071 seconds
[dga@pvm5 ~]$ mpdringtest 100
time for 100 loops = 0.278882980347 seconds
[dga@pvm5 ~]$ mpdringtest 1000
time for 1000 loops = 2.07774806023 seconds

[dga@pvm5 ~]$ mpdtrace -l
pvm5_51257 (10.164.25.228)
pvm7_45890 (10.164.24.223)
pvm8_53007 (10.164.25.219)
pvm6_42937 (10.164.24.221)
pvm2_34177 (10.164.25.226)
```

7.4 Anexo D – Diagrama do Código Modificado (simples), Figura 40

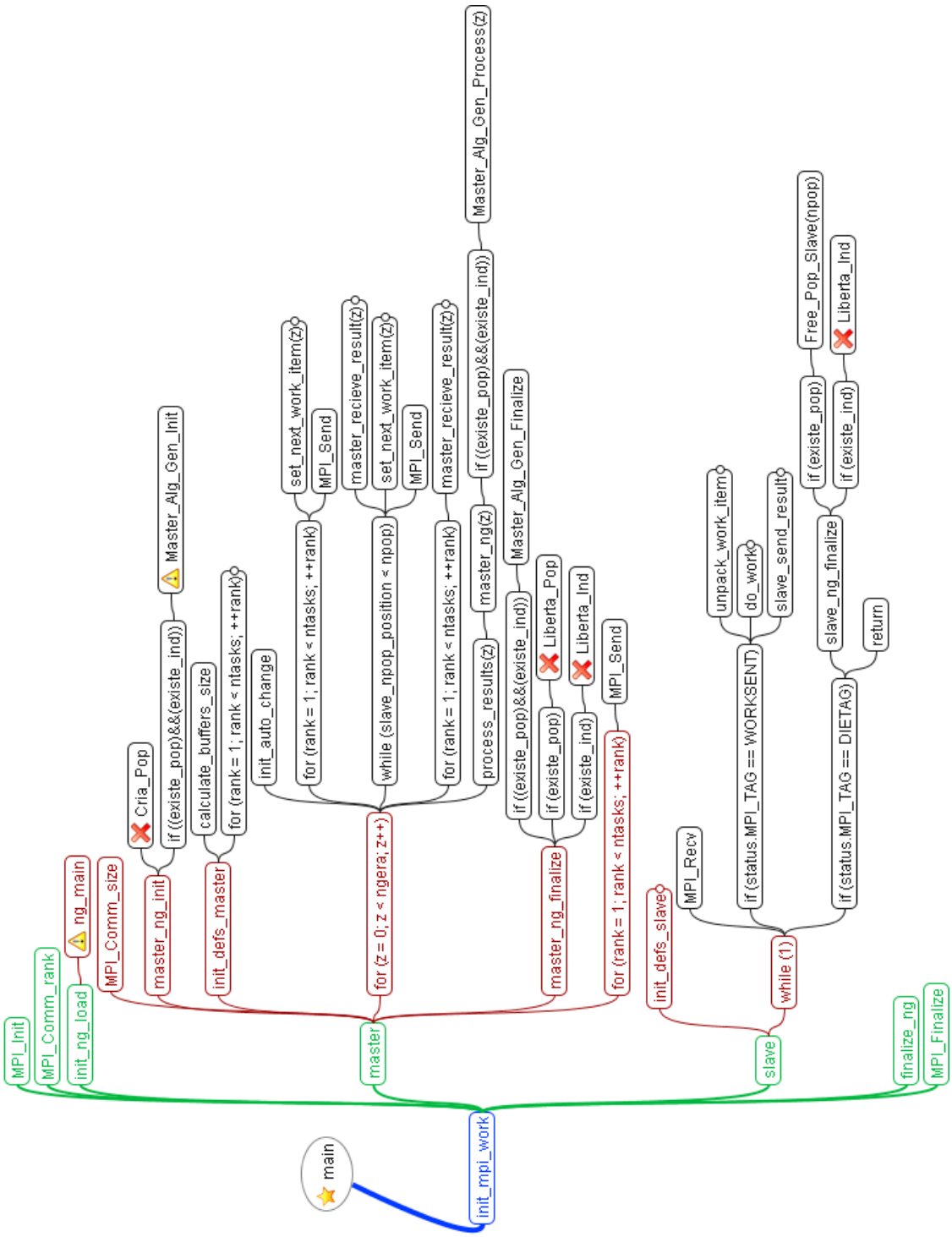

Figura 40: Diagrama compacto das funções principais da aplicação

7.5 Anexo E – Diagrama do nó Servidor, Figura 41

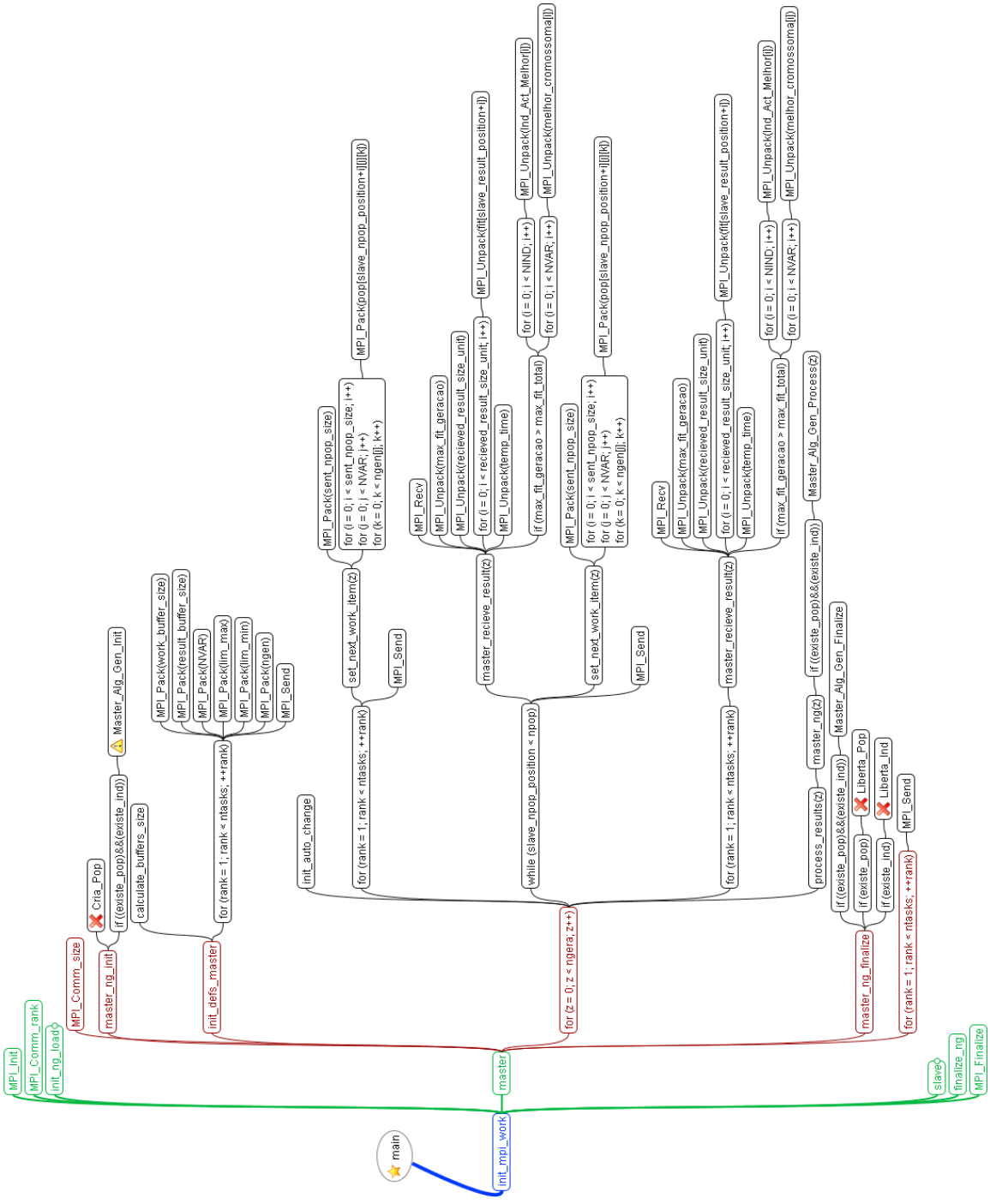

Figura 41: Diagrama de funções do nó Servidor

7.6 Anexo F – Diagrama do nó Cliente, Figura 42

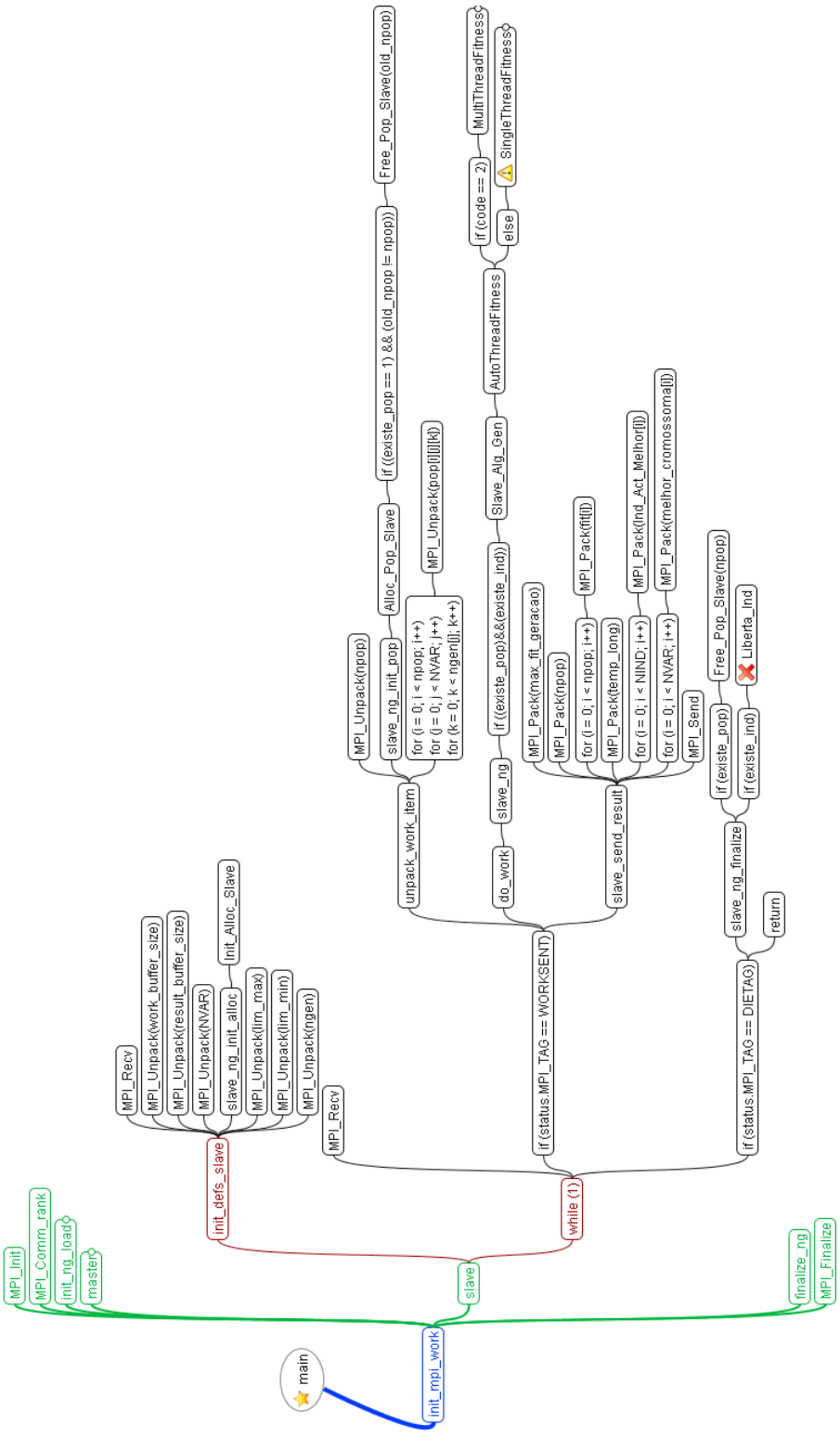

Figura 42: Diagrama de funções do nó Cliente

7.7 Anexo G – Netlist do Melhor Resultado

*c:fb_b2:i2 fb_b2:n3 fb_b2:n2 8000.000fF
*c:fb_b2:i1 fb_b2:n1 fb_b2:n2 8000.000fF
*c:fb_a2:i2 fb_a2:n3 fb_a2:n2 8000.000fF
*c:fb_a2:i1 fb_a2:n1 fb_a2:n2 8000.000fF
*c:fb_b1:i2 fb_b1:n3 fb_b1:n2 8000.000fF
*c:fb_b1:i1 fb_b1:n1 fb_b1:n2 8000.000fF
*c:fb_a1:i2 fb_a1:n3 fb_a1:n2 8000.000fF
*c:fb_a1:i1 fb_a1:n1 fb_a1:n2 8000.000fF
*c_cmfb2 n8 vss 600.000fF
csbp nbs2bp vss 896.532fF
csap nbs2ap vss 896.532fF
csbn nbs2bn vss 929.707fF
csan nbs2an vss 929.707fF
ctb n3b n2b 0.000fF
cta n3a n2a 0.000fF
cfb n1b n2b 0.000fF
cfa n1a n2a 0.000fF
cmb n4b n2b 0.000fF
cma n4a n2a 0.000fF
cbb n4b n3b 4000.000fF
cba n4a n3a 4000.000fF
cab n4b n1b 4000.000fF
caa n4a n1a 4000.000fF
clb n4b vss 28000.000fF
cla n4a vss 28000.000fF
ib2 vb21 vss 0.181mA
ib1 vb3 vss 0.181mA
*vvcmi vcmi vss 800.000mV
*vvib2 vcmi_in vin_b2 0.000mV
*vvia2 vin_a2 vcmi_in 0.000mV
*vvcmi_in vcm_in vss 550.000mV
*vvib1 vin_b1 vcmi_in 0.000mV
*vvia1 vcmi_in vin_a1 0.000mV
*vf2 f2 vss 0.000mV

*vf1 f1 vss 0.000mV
*vvss vss 0 0.000mV
*vvdd vdd vss 1200.000mV
m8 n8 vb8 vss vss n_hsl130 w=1878.022u l=0.454u
m6b n4b n2b n8 vss n_hsl130 w=138.156u l=0.338u
m6a n4a n2a n8 vss n_hsl130 w=138.156u l=0.338u
m5b n3b vb5 vss vss n_hsl130 w=287.070u l=0.989u
m5a n3a vb5 vss vss n_hsl130 w=287.070u l=0.989u
m4b n2b nbs2bp n3b vss n_hsl130 w=253.333u l=1.064u
m4a n2a nbs2ap n3a vss n_hsl130 w=253.333u l=1.064u
m1b n1b vib n0 vss n_hsl130 w=225.458u l=0.263u
m1a n1a via n0 vss n_hsl130 w=225.458u l=0.263u
m0 n0 vb0 vss vss n_hsl130 w=168.728u l=1.422u
mbs4bp nbs3bp vb5 vss vss n_hsl130 w=15.966u l=0.989u
mbs4ap nbs3ap vb5 vss vss n_hsl130 w=15.966u l=0.989u
mbs3bp nbs2bp vb4 nbs3bp vss n_hsl130 w=7.045u l=1.064u
mbs3ap nbs2ap vb4 nbs3ap vss n_hsl130 w=7.045u l=1.064u
mbs4bn nbs3bn vb5 vss vss n_hsl130 w=7.983u l=0.989u
mbs4an nbs3an vb5 vss vss n_hsl130 w=7.983u l=0.989u
mbs3bn nbs2bn vb4 nbs3bn vss n_hsl130 w=7.045u l=1.064u
mbs3an nbs2an vb4 nbs3an vss n_hsl130 w=7.045u l=1.064u
mbs9n nbsdifn vb0 vss vss n_hsl130 w=19.428u l=1.422u
mbsbn nbs1bn n1b nbsdifn vss n_hsl130 w=19.782u l=0.263u
mbsan nbs1an n1a nbsdifn vss n_hsl130 w=19.782u l=0.263u
mbscn vdd np8 nbsdifn vss n_hsl130 w=39.563u l=0.263
umb26 np9 vb5 vss vss n_hsl130 w=37.805u l=0.989u
mb25 vbs2 vb4 np9 vss n_hsl130 w=33.363u l=1.064u
mb22 np10 vb5 vss vss n_hsl130 w=37.805u l=0.989u
mb21 vbs3 vb4 np10 vss n_hsl130 w=33.363u l=1.064u
mb19 vb0 vb0 vss vss n_hsl130 w=16.873u l=1.422u
mb16 np5 vb5 vss vss n_hsl130 w=54.651u l=0.989u
mb15 vb71 vb4 np5 vss n_hsl130 w=48.228u l=1.064u
mb13 np6 vb5 vss vss n_hsl130 w=28.707u l=0.989u
mb12 vb5 vb4 np6 vss n_hsl130 w=25.333u l=1.064u
mb9 vb4 vb4 vss vss n_hsl130 w=5.102u l=1.064u
mb6 vb8 vb8 vss vss n_hsl130 w=187.802u l=0.454u
m7b n4b vb72 vdd vdd p_hsl130 w=1412.821u l=0.870u
m7a n4a vb72 vdd vdd p_hsl130 w=1412.821u l=0.870u
m3b n2b nbs2bn n1b vdd p_hsl130 w=850.672u l=0.240u

m3a n2a nbs2an n1a vdd p_hsl130 w=850.672u l=0.240u
m2b n1b vb22 vdd vdd p_hsl130 w=532.357u l=0.940u
m2a n1a vb22 vdd vdd p_hsl130 w=532.357u l=0.940u
mbs2bp nbs2bp vbs3 nbs1bp vdd p_hsl130 w=23.656u l=0.240u
mbs2ap nbs2ap vbs3 nbs1ap vdd p_hsl130 w=23.656u l=0.240u
mbs1bp nbs1bp vbs2 vdd vdd p_hsl130 w=11.241u l=0.940u
mbs1ap nbs1ap vbs2 vdd vdd p_hsl130 w=11.241u l=0.940u
mbs9p nbsdifp vbs2 vdd vdd p_hsl130 w=44.965u l=0.940u
mbsbp nbs3bp n3b nbsdifp vdd p_hsl130 w=69.236u l=0.263u
mbsap nbs3ap n3a nbsdifp vdd p_hsl130 w=69.236u l=0.263u
mbscp vss np9 nbsdifp vdd p_hsl130 w=138.473u l=0.263u
mbs2bn nbs2bn vbs3 nbs1bn vdd p_hsl130 w=23.656u l=0.240u
mbs2an nbs2an vbs3 nbs1an vdd p_hsl130 w=23.656u l=0.240u
mbs1bn nbs1bn vbs2 vdd vdd p_hsl130 w=22.483u l=0.940u
mbs1an nbs1an vbs2 vdd vdd p_hsl130 w=22.483u l=0.940u
mb24 vbs2 vbs3 np8 vdd p_hsl130 w=112.029u l=0.240u
mb23 np8 vbs2 vdd vdd p_hsl130 w=53.236u l=0.940u
mb20 vbs3 vbs3 vdd vdd p_hsl130 w=5.437u l=0.240u
mb18 vb0 vb3 np7 vdd p_hsl130 w=53.923u l=0.240u
mb17 np7 vb21 vdd vdd p_hsl130 w=25.624u l=0.940u
mb14 vb71 vb71 vdd vdd p_hsl130 w=141.282u l=0.870u
mb11 vb5 vb3 np4 vdd p_hsl130 w=85.067u l=0.240u
mb10 np4 vb21 vdd vdd p_hsl130 w=40.424u l=0.940u
mb8 vb4 vb3 np3 vdd p_hsl130 w=85.067u l=0.240u
mb7 np3 vb21 vdd vdd p_hsl130 w=40.424u l=0.940u
mb5 vb8 vb3 np2 vdd p_hsl130 w=323.892u l=0.240u
mb4 np2 vb21 vdd vdd p_hsl130 w=153.912u l=0.940u
mb3 vb21 vb3 np1 vdd p_hsl130 w=112.029u l=0.240u
mb2 np1 vb21 vdd vdd p_hsl130 w=53.236u l=0.940u
mb1 vb3 vb3 vdd vdd p_hsl130 w=5.437u l=0.240u

7.8 Anexo H – Exemplo de integração de um novo circuito

Apenas considerando modificações numa integração "exemplo" de um novo circuito temos que, serão necessárias modificações/verificações nos ficheiros "circuit_main.c" e "circuit_objective.c":

- Inserir a função, *FitnessFunction(...)* do novo circuito a simular no ficheiro "circuit_objective.c"
- Verificar se todas as variáveis internas da função FitnessFunction(...), que não sejam utilizadas em restantes funções da aplicação, estão declaradas no ficheiro "circuit_objective.c". Caso não estejam, deverão ser declaradas no mesmo ficheiro.
- Se existirem variáveis em que seja estritamente necessário a sua utilização com o resto da aplicação, nomeadamente com o motor de algoritmos genéticos por exemplo, então esta alteração deverá dar origem a uma função declarada no ficheiro "circuit_main.c" e chamada do ficheiro "circuit_objective.c".
- No ficheiro "circuit_objective.c" deverá respeitar-se as variáveis já declaradas com o objectivo de manter futuras compatibilidades com novas funções.

7.9 Anexo I – Código de funções

Neste anexo serão detalhadas algumas funções com o objectivo de acompanharem o texto do trabalho:

1. *Init_Alloc_Slave()*

```
void Init_Alloc_Slave(void)
{
      ind = (double*) calloc(NVAR, sizeof(double));
      lim_max = (double*) calloc(NVAR, sizeof(double));
      lim_min = (double*) calloc(NVAR, sizeof(double));
      ngen = (int*) calloc(NVAR, sizeof(int));
      Titulo_genes = (char**) calloc(NVAR, sizeof(char*));
```

```
      melhor_cromossoma = (double*) calloc(NVAR, sizeof(double));
      melhor_cromossoma_total = (double*) calloc(NVAR, sizeof(double));
}
```

2. *Master_Alg_Gen_Init()*

```
void Master_Alg_Gen_Init(void)
{
   char     *strtmp;
   int            m;
   adopt_index = (double*) calloc(npop, sizeof(double));
   e1 = 0;
   max_fit_total = 0;
   Stop = 0;
   stopAfterGeneration = 0;
   indunits = (char**) calloc(NIND, sizeof(char*));
   strtmp = (char*) calloc(1024, sizeof(char));
   for (m = 0; m < NIND; m++) {
      char *charptr1, *charptr2;
      indunits[m] = NULL;
      charptr1 = strchr(Titulo_Des[m], '(');
      if (!charptr1) continue;
      charptr2 = strchr(charptr1, ')');
      if (!charptr2) continue;
      charptr1++;
      int c = 0;
      while (charptr1 != charptr2)
         strtmp[c++] = *(charptr1++);
      strtmp[c] = '\0';
      indunits[m] = (char*) calloc(strlen(strtmp)+1, sizeof(char));
      strcpy(indunits[m], strtmp);
   }
   free(strtmp);
}
```

3. *Master_Alg_Gen_Process(...)*

```
void Master_Alg_Gen_Process(int ngera_z)
{
      char  aux2;
      int   i, j, k, m, auxi, mut;
        printf("\n-- %d # %d --\n", ngera_z+1, ngera);
      printf("Master_Alg_Gen_Process max_fit_total: %f DONE\n",
max_fit_total);
            for (i = 0; i < NVAR; i++)
                  melhor_cromossoma_total[i] = melhor_cromossoma[i];
            for (i = 0; i < NIND; i++)
                  melhores_parametros[i] = Ind_Act_Melhor[i];
            printf("\n");
            for (i = 0; i < NIND; i++){
                  printf("%10s | ", Titulo_Des[i]);
            }
            printf("\n");
            for (i = 0; i < NIND; i++){
                printf("%10.2E ", melhores_parametros[i]);
                if (indunits[i])
                  printf("%s", indunits[i]);
                else
                  printf("?");
                printf("| ");
            }
            printf("\n");
```

```
        //}
#ifdef GA_MAIN_DEBUG_ON
      printf("Master_Alg_Gen_Process SELECTION DONE\n");
      printf("Master_Alg_Gen_Process fit size: %i\n", sizeof(fit)/8);
#endif
      srand( (unsigned)time( NULL ) );
        if (Rank){
                for (i=0;i<npop;i++)
                        adopt_index[i]= i;
                quicksort(fit, adopt_index, 0, npop-1);
                for (i=0;i<npop;i++)
                        fit[(int)adopt_index[i]]=i+1;
        }
        double sum = 0;
#ifdef GA_MAIN_DEBUG_ON
      printf("Master_Alg_Gen_Process RANK DONE\n");
#endif
        for(i = 0; i < npop; i++)
                sum += fit[i];
        for(i = 0; i < npop; i++)
                fit[i] = fit[i]/sum;
        cumfit[0] = fit[0];
#ifdef GA_MAIN_DEBUG_ON
      printf("Master_Alg_Gen_Process cumfit value1: %f\n", cumfit[0]);
#endif
        for(i = 1; i < npop; i++)
                cumfit[i] = cumfit[i-1] + fit[i];
#ifdef GA_MAIN_DEBUG_ON
      printf("Master_Alg_Gen_Process cumfit value2: %f\n", cumfit[0]);
#endif
        if (elitista){
                for (j = 0; j < NVAR; j++)
                        for (k = 0; k < ngen[j]; k++){
                                novapop[npop-1][j][k] = pop[melhor][j][k];
                        }
        }
        else{
                for (i = 0; i<adopt_round((1-pc)*npop); i++){
                        double aux = adopt_rand(100001)/100000.0;
                        for (j=0; j<npop; j++){
                                if (aux <= cumfit[j]){
                                        e1=j;
                                        break;
                                }
                                e1 = j;
                        }
                        for (j = 0; j<NVAR; j++)
                                for (k = 0; k<ngen[j]; k++)
                                        novapop[npop-(adopt_round((1-
pc)*npop))+i][j][k] = pop[e1][j][k];
                }
        }
#ifdef GA_MAIN_DEBUG_ON
      printf("Master_Alg_Gen_Process elitista DONE\n");
#endif
        if (elitista)
                for (i=0; i<npop-2; i+=2)
                        Cruzamento(i);
        else
```

```
                for (i=0; i < adopt_round((pc*npop)); i+=2)
                        Cruzamento(i);
#ifdef GA_MAIN_DEBUG_ON
      printf("Master_Alg_Gen_Process Cruzamento DONE\n");
#endif
        for (i=0; i<adopt_round(pm*npop); i++){
                if (elitista)
                        auxi =
adopt_round((npop-2)*adopt_rand(100001)/100000.0);
                else
                        auxi =
adopt_round((npop-1)*adopt_rand(100001)/100000.0);
                for (m=0; m<NVAR; m++){
                        if (!MutVar)
                                mut =
adopt_round((ngen[m]-1)*adopt_rand(100001)/100000.0);
                        else
                                mut = adopt_round((-((double)ngen[m]-2)/
(ngera-1)*ngera_z + ngen[m]-1)*adopt_rand(100001)/100000.0);
                        aux2 = novapop[auxi][m][mut];
                        if (aux2 == 0)
                                novapop[auxi][m][mut] = (char)1;
                        else
                                novapop[auxi][m][mut] = (char)0;
                }
        }
#ifdef GA_MAIN_DEBUG_ON
      printf("Master_Alg_Gen_Process Mutacao DONE\n");
#endif
        for (i=0; i<npop; i++)
                for (j=0; j<NVAR; j++)
                        for (k=0; k<ngen[j]; k++){
                                pop[i][j][k] = novapop[i][j][k];
                        }
#ifdef GA_MAIN_DEBUG_ON
      printf("Master_Alg_Gen_Process newpop DONE\n");
#endif
      if (stopAfterGeneration){
            printf("Paused after generation ...\n");
            getchar();
            stopAfterGeneration = 0;
      }
#ifdef GA_MAIN_DEBUG_ON
      printf("Master_Alg_Gen_Process ALL DONE\n");
#endif

}
```

4. *Master_Alg_Gen_Finalize()*

```
void Master_Alg_Gen_Finalize(void)
{
   free(indunits);
   free(adopt_index);
}
```

5. *Slave_Alg_Gen(...)*

```
void Slave_Alg_Gen(void)
{
      max_fit_total = 0;
        max_fit_geracao = 0;
```

```
        melhor = 0;
#ifdef GA_MAIN_DEBUG_ON
      printf("!! MPI MultiThreadFitness !!\n");
#endif
      AutoThreadFitness(autothread);

}
```

6. *AutoThreadFitness(...)*

```
void AutoThreadFitness(int code)
{
if (code == 2) {
      MultiThreadFitness();
}
else
      SingleThreadFitness();
}
```

7. *SingleThreadFitness()*

```
void SingleThreadFitness(void)
{
      double      aux;
      int   j, k, i, m;
      for (i=0; i<npop;i++){
                for (j=0; j<NVAR; j++){
                        aux = 0;
                        for (k=0; k<ngen[j]; k++)
                                aux = aux + pow(2,k)*((int)pop[i][j][k]);
                              ind[j] = lim_min[j] + (lim_max[j]-
lim_min[j])*aux/(pow(2,ngen[j])-1);
                }
            fit[i] = Fitness(ind,Ind_Act, fplog);
                if (fit[i] > max_fit_geracao){
                        max_fit_geracao = fit[i];
                        melhor = i;
                        for (m = 0; m < NIND; m++)
                                Ind_Act_Melhor[m] = Ind_Act[m];
                        for (m = 0; m < NVAR; m++)
                                melhor_cromossoma[m] = ind[m];
                }
      }
}
```

8. *MultiThreadFitness()*

```
void MultiThreadFitness(void)
{
      double      aux;
      int   i, j, k, m;
      MT_io MT_Data;
      MT_io *MT_Data_ptr;
      MT_Data_ptr = &MT_Data;
      MT_Data_ptr->MT_ind = (double**) calloc(npop, sizeof(double*));
      MT_Data_ptr->MT_Ind_Act = (double**) calloc(npop, sizeof(double*));
      MT_Data_ptr->MT_index = (int*) calloc(npop, sizeof(int));
      MT_Data_ptr->MT_lauch_index = 0;
#ifdef MULTITHREAD_MAIN_DEBUG_ON
      printf("->Init\n");
#endif
      pthread_mutex_init(&(MT_Data_ptr->launch_lock), NULL);
#ifdef MULTITHREAD_MAIN_DEBUG_ON
```

```
      printf("->Init-Done\n");
#endif
      for (i=0; i<npop;i++){
            MT_Data_ptr->MT_index[i] = i;
      }
      pthread_t thread1, thread2;
      int iret1, iret2;
      iret1 = pthread_create(&thread1, NULL, FitnessThread, (void*)
MT_Data_ptr);
      pthread_join(thread1, NULL);
      pthread_mutex_destroy(&(MT_Data_ptr->launch_lock));
#ifdef MULTITHREAD_MAIN_DEBUG_ON
      printf("Mutex Destroyed!\n");
#endif
      SelectBestFitness(MT_Data_ptr->MT_ind, MT_Data_ptr->MT_Ind_Act);
}
```

9. *FitnessLoop(...)*

```
void FitnessLoop(double **MT_ind, double **MT_Ind_Act, int pop_index)
{
      double      aux;
      int   j, k;
      MT_ind[pop_index] = (double*) calloc(NVAR, sizeof(double));
      MT_Ind_Act[pop_index] = (double*) calloc(NIND, sizeof(double));
      for (j=0; j<NVAR; j++){
            aux = 0;
            for (k=0; k<ngen[j]; k++)
                  aux = aux + pow(2,k)*((int)pop[pop_index][j][k]);
                  MT_ind[pop_index][j] = lim_min[j] + (lim_max[j]-
lim_min[j])*aux/(pow(2,ngen[j])-1);
      }
      printf("->(NODE:%i)Fit-Entering=%i\n", myrank, pop_index);
      fit[pop_index] = Fitness(MT_ind[pop_index], MT_Ind_Act[pop_index],
fplog);
      printf("->(NODE:%i)Fit-Done=%i\n", myrank, pop_index);
}
```

10. *SelectBestFitness(...)*

```
void SelectBestFitness(double **MT_ind, double **MT_Ind_Act)
{
      int   j, m;
      for (j=0; j<npop; j++){
                  if (fit[j] > max_fit_geracao){
                        max_fit_geracao = fit[j];
                  melhor = j;
            }
      }
      for (m = 0; m < NIND; m++)
            Ind_Act_Melhor[m] = MT_Ind_Act[melhor][m];
      for (m = 0; m < NVAR; m++)
            melhor_cromossoma[m] = MT_ind[melhor][m];
}
```

11. **FitnessThread(...)*

```
void *FitnessThread(void* ptr)
{
#ifdef MULTITHREAD_MAIN_DEBUG_ON
      printf("Lancamento de Thread\n");
#endif
      MT_io *MT_Data_ptr;
```

```
      MT_Data_ptr = (MT_io *) ptr;
      int index;
      while (!Stop) {
#ifdef MULTITHREAD_MAIN_DEBUG_ON
      printf("Thread Locked\n");
#endif
      pthread_mutex_lock(&(MT_Data_ptr->launch_lock));
#ifdef MULTITHREAD_MAIN_DEBUG_ON
      printf("Thread Activated=%d\n", MT_Data_ptr->MT_lauch_index);
#endif
      if (MT_Data_ptr->MT_lauch_index<npop) {
#ifdef MULTITHREAD_MAIN_DEBUG_ON
      printf("Work FOUND!\n");
#endif
}
      else {
      pthread_mutex_unlock(&(MT_Data_ptr->launch_lock));
#ifdef MULTITHREAD_MAIN_DEBUG_ON
      printf("NO Work TO DO!\n");
#endif
      break;
      }
      index = MT_Data_ptr->MT_index[MT_Data_ptr->MT_lauch_index];
      MT_Data_ptr->MT_lauch_index++;
      pthread_mutex_unlock(&(MT_Data_ptr->launch_lock));
#ifdef MULTITHREAD_MAIN_DEBUG_ON
      printf("Thread UnLocked\n");
#endif
      FitnessLoop(MT_Data_ptr->MT_ind, MT_Data_ptr->MT_Ind_Act, index);
#ifdef MULTITHREAD_MAIN_DEBUG_ON
      printf("Thread FitnessLoop Done\n");
#endif
      }
#ifdef MULTITHREAD_MAIN_DEBUG_ON
      printf("Acabou a Thread\n");
#endif
}
```

7.10 Anexo J – Plataforma de Trabalho

- *Hardware (11 máquinas):*
  - *4 x Intel(R) Pentium(R) 4 CPU HT (3.00GHz) 2048KB L2 (512MB RAM)*
  - *1 x AMD Sempron(tm) Processor 2800+ (1.6GHz) 256 KB L2 (1GB RAM)*
  - *5 x Intel(R) Pentium(R) 4 CPU (1.70GHz) 256KB L2 (256MB RAM)*
  - *1 x Intel(R) Core(TM)2 Duo CPU T7300 (2.0GHz) 4096KB L2 (512MB RAM)*

- *Software (SO + MPI + Compiladores)*
  - *Linux (Ubunto) 2.6.17-11-generic*
  - *Linux 2.6.18-1.x.fc6xen*
  - *Linux 2.6.21-1.x.fc7*
  - *MPICH2 – (1.0.4p1, 1.0.5p1, 1.0.6p1)*
  - *GCC*

7.10.1 Versões de UNIX Recomendadas

Existem hoje em dia diversas distribuições de UNIX (anteriormente chamado "Unics", "UNiplexed Information and Computing System"), umas mais comuns, vocacionadas para os utilizadores regulares/ocasionais e outras distribuições mais direccionadas para o mundo empresarial, requerendo estas um *"hardware"* mais profissional e caro, que desempenha tarefas mais optimizadas mas também mais complexas de implementar.

Para correr a aplicação desenvolvida recomenda-se apenas o uso de versões de *Kernel Linux 2.6* de qualquer distribuição, desde que respeite as dependências de compilação da aplicação.

7.10.2 Ambientes MPI Recomendados

Os ambientes MPI mais conhecidos e nos quais a versão de MPI utilizada (versão 1) é suportada em todos eles são:

- OpenMPI
- MPICH2 (*"MPI Chameleon 2"*)
- LAM/MPI (*"Local Area Multicomputer/MPI"*)
- PE (Paralell Environment da IBM)

Figura 43: Símbolo típico do LAM/MPI

Figura 44: Símbolo típico do OpenMPI

As Figuras 43 e 44 representam símbolos típicos de distribuições de MPI. Todos as restantes distribuições de MPI que respeitem o *standard* do MPI serão compatíveis com a aplicação desenvolvida neste trabalho.

7.10.3 Configuração e Instalação da Plataforma

Depois de instalado(s) o(s) sistema(s) operativo(s) no número de máquinas desejadas, a fim de se preparar o ambiente necessário para correr a aplicação desenvolvida neste trabalho, deve-se então proceder da seguinte forma:

1. Descarregar umas das distribuições MPI (neste exemplo o MPICH2).
2. Criar em todas as máquinas um utilizador com o mesmo nome e ID. No caso de utilização de LDAP (Lightweight Directory Access Protocol) não esquecer dar permissões ao utilizador para ter acesso às bibliotecas partilhadas.
3. Criar a seguinte estrutura de directórios num nó apenas do utilizador escolhido:
    - ~/sources
    - ~/install/mpich2

4. Descomprimir o pacote da distribuição de MPI para dentro do directório "~/sources" A estrutura dentro do directório "~/sources" deverá ficar consoante o nome e versão da distribuição usada, "~/sources/<nome_versão_da_distribuição>".
5. [Para o caso de LDAP com directórios por NFS (Network File System) ou CIFS (Common Internet File System) não é necessário este ponto 5] Copiar para as restantes máquinas e para as mesmas subdirectorias, o conteúdo dos directórios no ponto 3. Exemplo (necessita do pacote "coreutils" instalado, no caso da distribuição "Ubunto"):
    - Mudar para o nó onde está o ponto 3 (utilizador escolhido)
    - sync ~/sources <outra_máquina>:~/
    - sync ~/install <outra_máquina>:~/
    - Repetir os dois últimos passos para as restantes máquinas
6. Deverá ser disponibilizado uma forma de ligação às máquinas sem que seja necessário introduzir palavras-passe. Estes poderão ser por exemplo, por SSH (Secured Shell) ou por RSH (Remote Shell). Para proceder de forma a usar SSH deverá realizar-se os seguintes pontos:
    - Executar o seguinte comando em cada uma das máquinas através do utilizador anteriormente escolhido, para gerar as chaves (neste caso com o algoritmo RSA) sem palavra-passe (deverá premir-se <ENTER> para não digitar palavra-passe): ssh-keygen -t rsa
    - Criar o ficheiro *"authorized_keys"* dentro de *"~/.ssh/"* no utilizador acima escolhido em cada uma das máquinas.
    - Copiar o conteúdo do ficheiro *"~/.ssh/id_rsa.pub"* de cada máquina, para todos os ficheiros *"~/.ssh/authorized_keys"* de todas as mesmas máquinas.
    - Modificar as permissões do ficheiro *"~/.ssh/authorized_keys"* de todas as máquinas em questão para leitura e escrita pelo dono do ficheiro.
    - Executar em todas as máquinas o comando de leitura de identidades em que um dos argumentos (máquina) será igual ao nome de cada uma das restantes máquinas (incluindo a própria): *"ssh_keyscan -t rsa «máquina» >> ~/.ssh/known_hosts"*
7. Continuar com o procedimento da secção seguinte (7.10.3.1 - *Método de Configuração/Compilação/ Instalação*) em todas as máquinas.

7.10.3.1 Método de Configuração/Compilação/ Instalação

Após seguir os passos da secção anterior, 7.10.3 - *Configuração e Instalação da Plataforma,* é necessário proceder à configuração do modo de compilação para que a aplicação desenvolvida execute com sucesso num dos sistemas operativos recomendados:

1. Assumindo a presente localização dentro do directório: ~/sources/<distribuição>
2. Executar o *"script"* de configuração: "./configure --prefix=~/install/mpich2"
3. Compilar com: "make"
4. Instalar com: "make install"

Para que a "*shell*" onde se irá utilizar estes novos compiladores use estes novos compiladores em vez dos que foram utilizados por exemplo para compilar o mpich2, é necessário adicionar ou a todos os "*scripts*" de "*configure*" das aplicações a compilar ou simplesmente no arranque de uma nova *"shell"* como por exemplo o ficheiro "~/.bashrc" ou "~/.bash_profile" para uma "*shell bash*". Um exemplo das linhas a inserir será por exemplo:

```
{...}
export PATH=$PATH:$HOME/install/mpich2/bin
export LOCAL_PATH="$HOME/install/mpich2/bin"
export CC=$LOCAL_PATH/mpicc
export CPP="$CC -E"
export CXX=$LOCAL_PATH/mpicxx
export CXXCPP="$CXX -E"
export F77=$LOCAL_PATH/mpif77
echo $CC
echo $CPP
echo $CXX
echo $CXXCPP
echo $F77
echo "Done"
{...}
```

7.10.3.2 Administração do Ambiente (MPICH2)

Para se iniciar uma rede de nós através de um distribuição de MPICH2 deverá proceder-se da seguinte forma:

- Criar um ficheiro na *"home directory"* de cada utilizador de cada máquina com o nome de, *".mpd.conf"* em que o seu conteúdo terá uma palavra chave que distingue o ambiente MPI e se este tem permissões de entrar no ambiente a lançar (exemplo do conteúdo tipo do ficheiro, ".mpd.conf"): "MPD_SECRETWORD=<secret_word>" ou "MPD_SECRETWORD=123456789" por exemplo.
- Criar um ficheiro de nome *"mpd.hosts"*, que poderá estar em qualquer directório, desde que acessível, mas de qualquer forma iremos assumir a sua localização também dentro da *"home directory"* mas apenas do utilizador em que será iniciado o ambiente. Este ficheiro irá conter o nome das máquinas por cada linha e terão de ter resolução local ou por DNS, em que se deseja disponibilizar recursos ao ambiente MPI.
- Arrancar com a rede executando o comando: *"mpdboot -v -n <n_máquinas> -f ~/mpd.hosts"*

Depois de iniciada a rede ficarão disponíveis para execução, um conjunto de comandos que permitirá testar, executar, terminar e listar processos a integrados com MPI:

- mpdrun – Interpreta aplicações com código MPI (também corre aplicações normais)
- mpdringtest <ciclos> – Realiza um teste à volta da rede com um número de ciclos à escolha, retornando o tempo que levou a realizar essas mesmas ciclos.
- mpdtrace – Contacta cada um dos nós actualmente ligados na rede, identificando-os.
- mpdlistjobs – Lista em que nós estão os processos activos na rede.
- mpdkilljob – Termina um membro de um processo MPI num determinado nó.
- mpdallexit – Termina toda a rede ligada ao nó de onde se lança o comando.

Alguns exemplos de output dos comandos poderão ser consultados no Anexo C - Administração do MPICH2.

7.10.4 Configuração e Instalação da Aplicação

Antes de começar os procedimentos da presente e seguintes subsecções confirme todos os passos efectuados na secção 7.10.3 - *Configuração e Instalação da Plataforma* e suas subsecções a fim de conseguir efectuar o seguinte teste: *"mpdrun -np 10 hostname"* em que deverá conseguir visualizar 10 linhas com o nome das máquinas onde foi corrido o comando *"hostname"*.

Se no parágrafo anterior verificou com sucesso o nome das máquinas onde correu o comando *"hostname"* então o seu ambiente de trabalho está bem configurado. Para configurar o pacote aplicativo deverá então proceder da seguinte forma:

- Em cada uma das máquinas onde irá ser executado o programa criar/descomprimir a aplicação, tendo em conta que a estrutura de directórios terá de ser igual em todas as máquinas e em todas as directorias que a aplicação usar. Recomenda-se que se descomprima para uma só máquina e depois se crie nas restantes apenas a árvore de directórios necessária ao funcionamento da aplicação: *"ngaEDA/src/"*, *"ngaEDA/sim"*, *"ngaEDA/cfg"*, *"ngaEDA/tech_files/umc"*.
- Depois de executar os passos descritos na próxima secção 7.10.4.1 - *Método de Configuração/Compilação*, deverá copiar o ficheiro *"ngaEDA/src/ngspice"* para as restantes máquinas.
- Para máquinas de arquitectura diferente é necessário efectuar também os passos descritos na próxima secção 7.10.4.1 - *Método de Configuração/Compilação*: por exemplo para 4 máquinas do tipo x86, 10 máquinas do tipo x86-64bit e ainda 5 máquinas do tipo ppc64 é necessário executar 3 configurações/compilações diferentes.

7.10.4.1 Método de Configuração/Compilação

Após seguir os passos da secção anterior 7.10.4 - *Configuração e Instalação da Aplicação*, é necessário proceder à configuração do modo de compilação para que a aplicação desenvolvida execute com sucesso num dos sistemas operativos recomendados:

1. Assumindo a presente localização dentro do directório: *".../ngaEDA/"*
2. [Opcional] aclocal no caso do sistema usar uma nomenclatura diferente para compilar, da máquina de onde o mesmo código foi préviamente compilado.
3. ./autogen.sh
4. ./configure
5. [Opcional] Dentro da directoria relativa *".../ngaEDA/src/iniparser/"* existe uma biblioteca "libiniparser.a" que terá de ser recompilada em determinados casos e que não está englobada na compilação geral, apenas liga as bibliotecas. Para tal deverá apagar-se esta biblioteca e de seguida executar os comandos, *"make clean"* e *"make"* na respectiva directoria.
6. make

7.10.4.2 Método de Arranque da Aplicação

Para que a aplicação execute com sucesso é necessário tomar em conta os seguintes aspectos:

- ✔ A árvore de directórios deverá ser igual em todas as máquinas para os directórios necessários ao funcionamento da aplicação.
- ✔ As permissões dos directórios deverão permitir ao utilizador escrever, ler e executar o ficheiro da aplicação.
- ✔ O utilizador deverá ser o mesmo em todas as máquinas.

- ✔ Através do comando *"mpdtrace"* serão visualizadas todas as máquinas acessíveis no ambiente MPI e em quais deveram existir as árvores de directórios da aplicação (e o executável da mesma).

Finalmente, para executar a aplicação em, por exemplo 10 máquinas físicas como computadores pessoais, poderão executar-se os seguintes comandos, assumindo que se está na directoria relativa *".../ngaEDA/src/"*:

- mpdrun -np 10 ./ngspice → mensagens para o ecran, sessão bloqueada
- mpdrun -np 10 ./ngspice > output.out → mensagens normais para ficheiro, mensagens de erro para o ecran, sessão bloqueada
- mpdrun -np 10 ./ngspice &> output.out → todas as mensagens para ficheiro, sessão bloqueada
- (mpdrun -np 10 ./ngspice &> output.out &) → todas as mensagens para ficheiro, sessão livre, a aplicação ficará a correr mesmo saindo da máquina

7.10.5 Requisitos de Software

Se quaisquer das situações acima gerar algum erro, deve verificar se contem todos os requisitos recomendados:

- ✔ Compiladores para código C e C++ ou compatíveis para o Sistema operativo utilizado
- ✔ Python 2.3 ou acima
- ✔ Bibliotecas STDC++
- ✔ Ferramentas usuais de configuração: automake, aclocal, m4, make

7.10.6 Opções de Plataforma não Testadas (Mas Compatíveis)

Embora colocada de parte, uma das plataformas mais conhecidas hoje em dia, o Windows, a razão da sua não utilização deve-se apenas ao requisito de manter a simplicidade de teste e de implementação da aplicação desenvolvida neste trabalho. Não obstante, a compreensão de que o código desenvolvido é igualmente suportado por parte das plataformas MPI, nomeadamente a distribuição MPICH2 quer num ambiente Windows ou UNIX, aliciou o uso extensivo da plataforma UNIX a fim de beneficiar das funcionalidades comuns entre o mesmo de tipo de sistema operativo para efeitos de comparação de resultados.

7.11 Anexo K – Errata da Aplicação Inicial

Uma solução possível para resolver o problema evidenciado na Tabela 30 será modificar a secção de código abaixo indicada no ficheiro "main.c" da aplicação inicial **de**:

```
Cria_Pop(iniparser_getstr(initSettings, "ga:crm"));
Carrega_Ind(iniparser_getstr(initSettings, "ga:ind"));
Initialize(Ind_Des);
Strength(strength);
Roleta = iniparser_getint(initSettings, "ga:roleta", 0);
Rank = iniparser_getint(initSettings, "ga:rank", 1);
elitista = iniparser_getint(initSettings, "ga:elitista", 1);
MutVar = iniparser_getint(initSettings, "ga:mutvar", 0);
pc = iniparser_getdouble(initSettings, "ga:pc", 0.76);
pm = iniparser_getdouble(initSettings, "ga:pm", 1.0);
ngera = iniparser_getint(initSettings, "ga:gen", 100);
npop = iniparser_getint(initSettings, "ga:pop", 100);
```

Para:

```
Roleta = iniparser_getint(initSettings, "ga:roleta", 0);
Rank = iniparser_getint(initSettings, "ga:rank", 1);
elitista = iniparser_getint(initSettings, "ga:elitista", 1);
MutVar = iniparser_getint(initSettings, "ga:mutvar", 0);
pc = iniparser_getdouble(initSettings, "ga:pc", 0.76);
pm = iniparser_getdouble(initSettings, "ga:pm", 1.0);
ngera = iniparser_getint(initSettings, "ga:gen", 100);
npop = iniparser_getint(initSettings, "ga:pop", 100);
Cria_Pop(iniparser_getstr(initSettings, "ga:crm"));
Carrega_Ind(iniparser_getstr(initSettings, "ga:ind"));
Initialize(Ind_Des);
Strength(strength);
```

8 REFERÊNCIAS

[1] Nuno Paulino Jorlo Goes I.', Adolfo Steiger-Garção, "Design Methodology for Optimization of Analog Building Blocks Using Genetic Algorithms", Universidade Nova de Lisboa / CRI-UNINOVA

[2] Dr. David P. Anderson, "Public Computing: Reconnecting People to Science", March 21, 2004, http://boinc.berkeley.edu/boinc2.pdf

[3] NGSPICE: a mixed-level/mixed-signal circuit simulator. http://ngspice.sourceforge.net/

[4] Torsten Hoefler, Prabhanjan Kambadur, Richard L. Graham, GalenShipman, and Andrew Lumsdaine, "A Case for Standard Non-Blocking Collective Operations", Paris, France, October 2007, http://www.open-mpi.org/papers/euro-pvmmpi-2007-nb-coll/mpi-vs-nbc.pdf

[5] B. Vaz, R. Costa, N. Paulino, J. Goes, R. Tavares, A. Steiger-Garção, "A General-purpose Kernel based on Genetic Algorithms for Optimization of Complex Analog Circuits", MWSCAS 2001, 2001.

[6] Richard L. Graham, Galen M. Shipman, Brian W. Barrett, Ralph H. Castain, George Bosilca, Andrew Lumsdaine, "Open MPI: A High-Performance, Heterogeneous MPI" – 1-4244-0328-6 – Spain, Barcelona, September 2006 IEEE; http://www.open-mpi.org/papers/heteropar-2006/heteropar-2006-paper.pdf

[7] R. Santos-Tavares, N. Paulino, J. Higino, J. Goes, J. P. Oliveira, "Optimization of Multi-Stage Amplifiers in Deep-Submicron CMOS Using a Distributed/Parallel Genetic Algorithm", Universidade Nova de Lisboa / CRI-UNINOVA 2007/2008, ISCAS' 08 – 2008 IEEE International Symposium on Circuits and Systems Seattle, USA 2008.

[8] TOP500 – SUPERCOMPUTERS, http://www.top500.org

[9] Hong Ong and Paul A. Farrell, "Performance Comparison of LAM/MPI, MPICH, and MVICH on a Linux Cluster connected by a Gigabit Ethernet Network", Department of Mathematics and Computer Science, Kate State University, October of 2000, http://www.usenix.org/publications/library/proceedings/als00/2000papers/papers/full_papers/ong/ong_html/

[10] BSIM3v3.3.0 MOSFET Model - Users' Manual, UC Berkeley – 07-29-2005; http://www-device.eecs.berkeley.edu/~bsim3/ftpv330/Mod_doc/b3v33manu.tar