



University of Minho
School of Engineering

Portfólio

Portfólio de varios trabalhos realizados ao longo dos
semestre

Hugo Afonso Da Gião
PG41073

12 de Julho de 2020

Resumo

Este documento contem relatorios relata o desenvolvimento e resultados obtidos na realizacao de varios trabalhos.

Conteúdo

1	Introdução	3
2	NAS parallel benchmarks	4
2.1	Introdução	4
2.2	Maquinas utilizadas	4
2.3	Metodologia utilizada e testes realizados	5
2.3.1	<i>Kernels</i> e pseudo-aplicações utilizadas	5
2.3.2	Testes realizados	5
2.3.3	Testes,métricas,opções e compiladores utilizados para as diferentes versões dos testes <i>NPB</i>	6
2.4	Análise dos resultados	9
2.4.1	Performance dos <i>benchmarks</i>	9
2.4.2	Impacto no sistema	10
3	Dtrace - Desenvolvimento de programas	12
3.1	Introdução	12
3.2	Ambiente utilizado	12
3.2.1	Traçado das chamadas a <i>system call open</i>	12
3.2.2	Estatísticas relativas a abertura e criação de ficheiros	13
3.2.3	Replicação do programa <i>strace -c</i>	13
3.3	Conclusões	14
4	Dtrace - Utilização de Dtrace para a análise de radix sort	15
4.1	Introdução	15
4.2	Algoritmo utilizado	15
4.2.1	Versão sequencial	15
4.2.2	Versões paralelas	16
4.3	Probes utilizadas	17
4.3.1	Custom probes	17
4.3.2	CPC probes	23
4.3.3	SYSINFO probes	24
4.3.4	PLOCKSTAT probes	24
4.3.5	SCHED probes	25
4.3.6	VMINFO probes	26
4.3.7	libmpi probes	27
4.4	Teste e Resultados	28
4.4.1	Metodologia e ambiente de teste	28

4.4.2	Resultados	29
4.5	Conclusões e trabalho futuro	41
5	<i>Perf</i> - Utilização da ferramenta <i>Perf</i> para a análise de aplicações	42
5.1	Introdução	42
5.2	Análise de diferentes algoritmos de ordenação	42
5.2.1	Hardware de teste	42
5.2.2	Uso de <i>Perf</i> para explicar as diferenças no desempenho dos diferentes algoritmos	43
5.2.3	Análise dos perfis de execução dos diferentes algoritmos	43
5.2.4	Uso de <i>Flamegraphs</i> para analisar as chamadas dos algoritmos	46
5.2.5	Análise do segundo algoritmo a nível <i>assembly</i>	48
5.3	Análise de algoritmos de multiplicação de matrizes com uso do <i>Perf</i>	50
5.3.1	Algoritmos de multiplicação de matrizes utilizados	50
5.3.2	<i>Hardware</i> de teste	50
5.3.3	Tamanhos do problema	51
5.3.4	Estabelecer uma <i>beline</i>	51
5.3.5	Encontro de pontos quentes	51
5.3.6	Análise do programa ao nível <i>assembly</i>	56
5.3.7	Incrementar a frequência das amostras	59
5.3.8	Eventos utilizados	60
5.3.9	Análise das diferentes versões do algoritmo e tamanhos	61
5.3.10	Comparação entre <i>Counting mode</i> e <i>Sampling</i>	65
5.3.11	Utilização de <i>FlameGraphs</i> para análise dos algoritmos	67
5.4	Conclusões e trabalho futuro	69
6	Conclusões e trabalho futuro	70

Capítulo 1

Introdução

Este documento descreve vários trabalhos realizados ao longo do semestre, estes trabalhos enquadram-se na área de análise de desempenho de sistemas e aplicações.

O primeiro trabalho consiste na análise de varias aplicações do *NAS parallel benchmarks*, estas sendo sequenciais e paralelas em memória distribuída, partilhada e híbridas utilizando ferramentas de monitorização e o uso destas ferramentas para avaliar o estado do sistema durante execução desses *benchmarks*.

O segundo e terceiro trabalhos consistem no desenvolvimento de varias aplicações utilizando a ferramenta *Dtrace* e o uso desta ferramenta para análise e avaliação de varias implementações do algoritmo *radix sort*.

O quarto trabalho consiste no uso da ferramenta *Perf* e da biblioteca *FlameGraphs* para a análise de vários algoritmos de ordenação e multiplicação de matrizes.

Capítulo 2

NAS parallel benchmarks

2.1 Introdução

Este trabalho visa a explorar o desempenho de diversos *benchmarks* do pacote *NAS parallel benchmarks*, em vários sistemas. Foram utilizadas as versões sequenciais, *OpenMP*, *MPI* e híbrida desses *benchmarks* nos testes realizados.

Para realizar as medições do desempenho dos *benchmarks* foram utilizadas várias ferramentas de monitorização que permitiram monitorizar o estado do sistema durante a execução dos programas. Os resultados desses utilitários foram posteriormente tratados utilizando *python* e foram criados gráficos para o auxílio da análise dos resultados utilizando a biblioteca *matplotlib*.

Numa primeira fase foram realizados testes com o objetivo de analisar a performance dos *benchmarks* utilizando diferentes compiladores, versões e opções de otimização. Para tal escolhi alguns dos *benchmarks* disponíveis de modo a poder simular diversos perfis de execução e realizei as medições para os diferentes compiladores, versões e opções de otimização.

Numa segunda fase utilizei alguns dos *benchmarks* para medir o impacto no sistema do aumento da carga das aplicações. Para isso foram escolhidos alguns *benchmarks* e utilizadas várias ferramentas de monitorização de modo a observar o estado do sistema na execução de diferentes classes dos mesmos.

Este relatório está organizado primeiramente por uma secção de exposição da metodologia e *benchmarks* e testes utilizados, das diferentes opções, classes e *benchmarks* utilizados para cada uma das versões *NPB*, posteriormente contém uma secção dos resultados obtidos. Será enviado também um ficheiro contendo os resultados dos vários testes realizados.

2.2 Maquinas utilizadas

Os testes foram realizados em sistemas com diferentes características, esses pertencem a um dos seguintes *racks* do *SeARCH*:

- nós **r641**:
 - **CPU**: 2 X *E5-2650v2* cada um com 8 cores e *SMT*
 - **RAM**: 64GiB
 - **L3 CACHE**: 2 x 20480KiB
 - **L2 CACHE**: 16 X 256KiB
 - **L1 CACHE**: 2 X 16 X 32KiB
- nós **r662**:
 - **CPU**: 2 x *E5-2695v2* cada um com 12 cores e *SMT*

- **RAM:**64GiB
- **L3 CACHE:** 2 x 30720KiB
- **L2 CACHE:** 16 X 256KiB
- **L1 CACHE:** 2 X 16 X 32KiB

2.3 Metodologia utilizada e testes realizados

Para cada um dos *benchmarks* corridos foram utilizadas um conjunto de métricas estas foram obtidas correndo cada *benchmark* um número de vezes dependendo do seu tamanho e guardar os resultados de diferentes comandos em ficheiros de modo a poder visualizar o estado do sistema durante a execução do programa. As medições foram repetidas 5 vezes e a média das 3 melhores execuções foi utilizada para criar os gráficos utilizados para a sua interpretação. Estes testes foram corridos utilizando o sistema *PBS* e para facilitar o uso de diferentes compiladores, opções e os uso de vários *benchmarks* foi utilizado *python* e biblioteca *os* para poder automatizar as invocações dos *scripts PBS* com diferentes parâmetros. Os ficheiros utilizados para obter os vários *benchmarks* são depois guardados com um nome e numa diretoria apropriada para depois serem utilizados para gerar os gráficos.

2.3.1 *Kernels* e pseudo-aplicações utilizadas

Utilizei os seguintes *kernels* nos testes realizados:

- **IS:** Utilizei este *kernel* para a realização dos testes Sequenciais, *OpenMp* e *MPI*. Este *kernel* permite avaliar acessos à memória.
- **EP:** Utilizado nos testes *OpenMp* e *Mpi*. Permite visualizar a performance duma aplicação com elevada escalabilidade.
- **MG:** Utilizado nos testes Sequenciais, *OpenMp* e *MPI*. Permite avaliar comunicação e performance em aplicações com uso intensivo de memória.
- **CG:** Utilizado nos testes Sequenciais. Permite medir acessos a memória irregulares.

Utilizei as seguintes pseudo-aplicações nos testes realizados:

- **BT:** Utilizado nos testes híbridos.
- **SP:** Utilizado nos testes híbridos.

2.3.2 Testes realizados

As metrcas relativas aos tempos de execução, *system time*, *user time*, *idle*, *iowait* e utilização máxima de *CPU* foram obtidos utilizando o comando *sar* 1 essas são:

- **Tempo total** O tempo total de uma execução do programa em segundos foi obtido dividindo o numero de linhas contendo informação relativa ao uso de *CPU* do comando pelo numero de execuções do programa.
- **Tempo system** Esta métrica foi obtida somando os valores de *system time* em cada linha do comando multiplicados por 0.01 dividido pelo numero de execuções do programa.
- **Tempo user** Esta métrica foi obtida somando os valores de *user time* em cada linha do comando multiplicados por 0.01 dividido pelo numero de execuções do programa.

- **Tempo iowait** Esta métrica foi obtida somando os valores de *iowait* em cada linha do comando multiplicados por 0.01 dividido pelo numero de execuções do programa.
- **Tempo idle** Esta métrica foi obtida somando os valores de *idle* em cada linha do comando multiplicados por 0.01 dividido pelo numero de execuções do programa.
- **Utilização máxima de CPU** Foi obtida obtendo subtraindo 1 ao valor mínimo do campo de idle das diferentes linhas produzidas pelo comando.

As métricas relativas ao uso de memoria no sistema durante a execução do programa foram obtidas utilizando o comando *sar -r 1*.

- **Memoria utilizada KiB** Esta métrica foi obtida medindo o valor máximo do campo *memory used KB* produzido pelo comando.
- **Memoria utilizada percentagem** Esta métrica foi obtida medindo o valor máximo do campo *memory used per* produzido pelo comando.

Para as métricas relacionadas com o uso de rede e disco foi utilizado o comando *sar -b 1*.

- **Utilização de disco** Foi obtida somando os valores do campos *breads* e *bwrtns* produzidos pelo comando.
- **Utilização da rede** Foi obtida somando os valores do campo *tps* produzidos pelo comando.

Apesar de apenas ser possível verificar o estado de uma maquina utilizando os comandos acima, os testes foram realizados da mesma maneira para as varias versões. Os resultados obtidos apesar de não darem uma visão completa do sistema para as versões *MPI* e híbrida permitem observando uma da duas maquinas obter grande parte da informação que seria retirada se fossem realizadas medições em ambas as maquinas.

2.3.3 Testes, métricas, opções e compiladores utilizados para as diferentes versões dos testes *NPB*

Versão sequencial

As tabelas seguintes contém os vários *benchmarks*, classes, compiladores, níveis de otimização, versões dos compiladores e métricas utilizadas. Apenas utilizei uma versão da suite de compilação da Intel por ser a única das disponíveis no *SeARCH* com licença. Nos gráficos os compiladores, versões e nível de otimização utilizados são identificados da forma (compilador).(nível de otimização).(versão).

Suites de compilação	Versões	Opções de optimização
Intel	19.0.5.281(2019)	O1,O2,O3
GNU	5.3.0,6.1.0,7.2.0	O1,O2,O3

Benchmarks	Tamanhos	Metricas
IS, MG, CG	S, W, A, B, C	Tempo total Tempo system Tempo user Tempo iowait Tempo idle Utilização maxima de CPU Memória utilizada KiB Memoria utilizada percentagem Utilização de disco

Para a realização dos testes relativos a performance dos *Benchmarks* foram utilizados todos os compiladores, opções de otimização e versões acima. Todos os testes foram corridos com opção *mcmmodel = medium*. Nestes testes foram também utilizadas todas as métricas acima exceto a utilização máxima de *CPU*. Foram utilizados os tamanhos B e C

Para os testes do impacto no sistema foram utilizados os compiladores de C e Fortran da GNU, versão 5.3.0 e opção -O3. Foram utilizadas as métricas Tempo total, Utilização máxima de *CPU*, Memória utilizada *KiB*, Memória utilizada percentagem e utilização de disco. Foram utilizadas as todas as classes na tabela.

Os testes relacionados com as versões sequenciais dos *benchmarks* foram corridos em apenas 1 máquina.

Versão OpenMP

Para estes *benchmarks* foram utilizados os mesmos compiladores e opções que anteriormente. Nos gráficos os compiladores versões e nível de otimização utilizados são identificados da forma (compilador).(nível de otimização).(versão).

Suites de compilação	Versões	Opções de otimização
Intel	19.0.5.281(2019)	O1,O2,O3
GNU	5.3.0,6.1.0,7.2.0	O1,O2,O3

Benchmarks	Tamanhos	Métricas
IS,MG,EP	S,W,A,B,C,D	Tempo total Tempo system Tempo user Tempo iowait Tempo idle Utilização máxima de CPU Memória utilizada KiB Memória utilizada percentagem Utilização de disco

Para a realização dos testes relativos a performance dos *Benchmarks* foram utilizados todos os compiladores, opções de otimização e versões. Todos os testes foram corridos com opção *mcmmodel = medium*. Nestes testes foram também utilizadas todas as métricas acima exceto a utilização máxima de *CPU*.

Para os testes do impacto no sistema foram utilizados os compiladores de C e Fortran da GNU, versão 5.3.0 e opção -O3. Foram utilizadas as métricas Tempo total, Utilização máxima de *CPU*, Memória utilizada *KiB*, memória utilizada percentagem e utilização de disco.

Os testes relacionados com as versões *OpenMp* dos *benchmarks* foram corridos em apenas 1 máquina.

Versão MPI

Para a realização dos testes dos *benchmarks MPI* foram utilizados varias versões das implementações *OpenMpi_eth* e *Mpich_eth* da Intel e da GNU. Os compiladores são identificados nos gráficos da forma (compilador).(versão). Nos gráficos o termo *gnu_eth* refere-se à implementação *OpenMpi* da GNU, *intel_eth* à implementação *OpenMpi* da Intel e os termos com *mpich* ou *mpich2* referem-se a essas implementações do respetivo fabricante.

Implementação de MPI	Versões	Fabricantes
OpenMpi_eth	1.8.2,1.6.3,	Intel,GNU
MPICH_eth	1.2.7,1.5	Intel,GNU

Benchmarks	Tamanhos	Metricas
IS, MG, EP	S, W, A, B, C, D	Tempo total Tempo system Tempo user Tempo iwait Tempo idle Utilização máxima de CPU Memória utilizada KiB Memória utilizada percentagem Utilização de disco Utilização de rede

Para a realização dos testes relativos a performance dos *Benchmarks* foram utilizados todos os compiladores, opções de otimização e versões. Todos os testes foram corridos com opção *medium*. Nestes testes foram também utilizadas todas as métricas acima exceto a utilização máxima de *CPU*. Foram utilizadas as classes C e D.

Para os testes do impacto no sistema foram utilizados os compiladores de C e Fortran da GNU, versão 5.3.0 e opção -O3. Foram utilizadas as métricas Tempo total, Utilização máxima de *CPU*, Memória utilizada *KiB*, memória utilizada percentagem e utilização de disco. Foram utilizadas todas as classes na tabela.

Os testes *MPI* foram corridos utilizando os comandos *mpirun -np cores -mca btl self, sm, tcp bin* para as versões *openmpi* da GNU e *mpirun -np cores bin* para as restantes.

Os testes relacionados com as versões *MPI* dos *benchmarks* foram corridos em apenas 2 máquinas do mesmo rack.

Versão híbrida

Para os testes da versão híbrida foram utilizadas várias versões das implementações *OpenMpi_ -eth* da *Intel* e *GNU*, utilizando as implementações *MPICH* utilizadas para os testes *MPI* ocorriam problemas em termos de acesso as bibliotecas de *OpenMp*. Os compiladores são identificados nos gráficos da forma (compilador).(versão). Nos gráficos o termo *gnu_ -eth* refere-se à implementação *OpenMpi* da *GNU*, *intel_ -eth* à implementação *OpenMpi* da *Intel*.

Implementação de MPI	Versões	Fabricantes
OpenMpi_ -eth	1.8.2(Intel e GNU) ,1.6.3(Intel) ,1.8.4(GNU),	Intel, GNU

Benchmarks	Tamanhos	Metricas
SP, BT	S, W, A, B, C	Tempo total Tempo system Tempo user Tempo iwait Tempo idle Utilização máxima de CPU Memória utilizada KiB Memória utilizada percentagem Utilização de disco Utilização de rede

Para a realização dos testes relativos a performance dos *Benchmarks* foram utilizados todos os compiladores, opções de otimização e versões. Todos os testes foram corridos com opção *medium*. Nestes testes foram também utilizadas todas as métricas acima exceto a utilização máxima de *CPU*. Foram nestes testes utilizadas as classes A e B.

Para os testes do impacto no sistema foram utilizados os compiladores de C e *Fortran* da *GNU*, versão 5.3.0 e opção -O3. Foram utilizadas as métricas Tempo total, Utilização máxima de *CPU*, memória utilizada *KiB*, memória utilizada percentagem e utilização de disco. Foram nestes testes utilizadas todas as classes na tabela.

Os testes da versão híbrida foram corridos utilizando os comandos *mpirun -np cores -mca btl self,sm,tcp bin* para as versões *openmpi* da *GNU* e *mpirun -np cores bin* para as restantes e antes de correr cada teste é invocado este comando *export OMP_NUM_THREADS = THREADS* com *THREADS* = 32 para máquinas *r641* e *THREADS* = 48 para máquinas *r662*

Estes testes foram corridos em duas máquinas do mesmo *rack*.

2.4 Análise dos resultados

2.4.1 Performance dos *benchmarks*

Versão sequencial

Os testes das versões sequenciais dos *benchmarks* mostram que é gasto um tempo considerável em *idle* isto acontece porque não são utilizados a maioria dos recursos de *CPU*, o restante tempo é gasto consideravelmente mais em *user time*, isto é visto também pela maior correlação entre as proporções dos diferentes valores nos testes de tempo total e *user time*, isto indica que os possíveis problemas de performance ocorrem devido a limitações dos *benchmarks*. Estes resultados foram observados para os vários *benchmarks* e tamanhos. O tempo de *iowait* é também consistentemente de baixa proporção comparativamente ao tempo total dos *benchmarks*.

Contrariamente ao que seria esperado utilizar opções de otimização mais elevadas não conduz necessariamente a melhores resultados nos *benchmarks*.

Versão OpenMP

Nos testes *OMP* notei que para os testes da classe C que o compilador da Intel apresenta tempos substancialmente maiores para os testes IS e MG, nos testes EP pode-se verificar o oposto. Notei também que nos testes OMP existe uma maior correlação entre o tempo gasto em *user time* e tempo total em relação ao *system time*. Notei que os tempos em *idle* são proporcionalmente bastante inferiores do que os da versão sequencial. Em termos de uso de memória não foram encontradas grandes disparidades entre as diferentes otimizações e compiladores.

Para os testes da classe D os compiladores da Intel continuam a ter tempos superiores nos testes IS e MG apesar de terem uma menor discrepância comparativamente ao tamanho inferior, para o teste EP os tempos foram consideravelmente menores para os compiladores da Intel tal como na classe menor em proporções similares. O uso de memória continua similar para as diferentes opções de compilação e compiladores.

Versão Mpi

No *benchmark* IS da classe C podemos ver que os tempos de execução apesar de variarem com as implementações de *MPI* utilizadas também variam com as máquinas utilizadas. Podemos observar tempos superiores nas máquinas *r641*, nessas máquinas consegui observar uma tendência para melhores tempos nas implementações da Intel. Notei que os tempos de *iowait* são insignificantes e as distribuições dos tempos de *idle* e *user time* são similares as do tempo total. Sendo que o tempo gasto em *idle* é de proporção significativa. Em termos de escrita no disco encontrei que as implementações da Intel são mais eficientes. Em termos de memória utilizada verifiquei que esta é bastante superior nas versões da Intel.

Para os testes do *benchmark* MG de classe C os tempos de execução são inferiores para as versões da Intel. Notei que os tempos de *user time* são insignificantes quando comparados com os de *user time* exceto para a versão *OpenMpi* de *GNU* versão 1.6.3. O tempo gasto em *idle* é também significativo

sendo maior para as versões da GNU especialmente a versão *OpenMpi* 1.6.3. Verificamos em ambas as máquinas que o número de escritas em disco é superior para as versões *OpenMpi* de GNU comparativamente as outras. Notei semelhantemente ao *benchmrk* anterior que o uso de memória é superior nas implementações da *intel*.

Para o *benchmark* EP da classe C, notei maiores tempos de execução para as máquinas r641 nas implementações da GNU, nas máquinas r662 não encontrei essas diferenças. Notei que foi gasto mais tempo em *user time* relativamente ao *system time* sendo que o tempo gasto em *system time* é insignificante com a exceção mais uma vez do compilador *OpenMpi* versão 1.6.3 da GNU. O número de blocos escritos é superior nas implementações *OpenMpi* da GNU. Verifiquei que as implementações da *Intel* utilizam mais memória.

No *benchmark* IS da classe D podemos verificar que os tempos de execução são similares para todos as implementações. Os tempos de *usertime* seguem uma distribuição similar ao tempo total. Podemos notar mais uma vez que o tempo de *system time* da implementação *OpenMpi* 1.6.3 da GNU é bastante superior aos de todas as outras implementações neste caso notei também que as versões da *Intel* têm maiores *system time* do que as restantes versões da GNU. Notei em termos de tempo *idle* tempos muito superiores para as máquinas r662 relativamente as restantes, nestas máquinas esses tempos são semelhantes nas máquinas r641 notei tempos inferiores para as versões da *Intel*. Mais uma vez em termos de leituras de disco o as implementações de *OpenMpi* da GNU são superiores. Os usos de memória são similares em todas as versões.

No *benchmark* MG da classe D notei tempos ligeiramente melhores para as implementações da *Intel*. Mais uma vez *system time* muito maior do que as outras implementações para a implementação *OpenMpi* da GNU versão 1.6.3. Em termos de *iowait* os valores não são significativos apesar disso os valores de *iowait* observados são em ambas as máquinas consideravelmente superiores para as implementações *OpenMpi* da GNU. Similarmente a outros testes notei tempos em *idle* bastante superiores nas máquinas r662 relativamente as r641, em ambas notei também tempos *idle* bastante inferiores para as implementações da *intel*. O numero de escritas no disco é superiores para as implementações *OpenMpi* da GNU. O uso de memória é semelhante.

Os resultados do *benchmark* EP são semelhantes aos do MG para a classe D, com a exceção do uso de memória que é superior para as implementações da *Intel*.

Em suma podemos concluir que para os vários tamanhos e *benchmarks* consegui obter geralmente melhores resultados utilizando as implementações de *OpenMpi* e *MPICH* da *Intel*. Sendo que essas demonstram realizar um melhor uso dos recursos.

Versão híbrida

No *benchmark* SP da classe A, podemos notar uma anomalia em que o tempo de execução utilizando a implementação *OpenMpi* da GNU versão 1.6.3 nas maquinas r662, sendo que os tempos são bastante superiores as restantes implementações e aos da mesma implementação nas maquina r641. Com essa exceção notei tempos menores nas implementações da *Intel* para ambos os benchmarks. Notei também um menor número de escritas e leituras nos compiladores da *Intel* e maior uso de memória nos mesmos. Notei também maiores tempos de *system time* e *user time* para os compiladores da *Intel* e menores de *iowait* e *idle* sendo que os de *idle* representam uma taxa significativa do tempo global.

2.4.2 Impacto no sistema

Versão sequencial

Em termos de tempo podemos notar que o crescimento do mesmo é mais acentuado para os *benchmarks* IS e MG. Notei também que para os testes IS e MG o uso de memória é crescente quando o tamanho dos *benchmarks* é aumentado. Para o *benchmark* CG o uso de memória varia pouco com o tamanho dos *benchmarks*. Em termos de blocos escritos no disco os valores crescem com os testes para o *benchmark* CG, nos outros *benchmarks* podemos notar repetidamente um maior número de escritas para os testes da classe A, B, C em relação aos outros.

Versão OpenMP

Em termos de tempo da aplicação podemos notar um aumento do tempo de execução elevado para todos os testes com o aumento de carga. Em termos de utilização máxima de *CPU* notei que esta é maior do que as do A e B nas classes S e W, mas estas são inferiores do que as dos testes C e D para o *benchmark* IS. No *benchmark* MG temos uma utilização máxima de *CPU* com pouca variação. No *benchmark* EP a utilização de *CPU* aumenta até ao tamanho B onde estagna. Em termos de uso de memória o uso de memória é significativamente maior para o tamanho D e valores similares para os outros tamanhos no *benchmark* IS. Algo similar acontece para o *benchmark* MG. No *benchmark* EP o uso de memória tem pouca variação independentemente das classes dos *benchmarks*. Em termos de utilização de disco encontrei que esta foi maior para a classe C para o *benchmark* IS. O mesmo foi encontrado para o *benchmark* MG e EP.

Versão Mpi

Para o teste IS encontrei um crescimento acentuado dos tempos de execução com o tamanho de carga. Em termos de uso máximo de *CPU* notei que este se situa perto dos 100% nas máquinas r641 em todos os *benchmarks* e possui valores oscilatórios nas máquinas r662 sendo maior nas duas classes maiores, mas não chegando a 70%. O uso de memória apesar de ter alguma oscilação aumenta com o aumento da carga. O número de blocos lidos são superiores nas classes S e W diminuem passando para a classe A e depois aumentam até a classe D. Notei também que o número de *packets* enviados é superior nas classes S e W nas máquinas r662 e depois inferior nas classes A, B e C com a classe D tendo um valor de *packets* superior aos das ultimas nas máquinas r641 e sendo o máximo nas máquinas r662.

Nos testes MG notei mais uma vez uso de *CPU* perto dos 100% para as máquinas r641 e com valores oscilatórios com tendências a ter maiores valores com o aumento de carga, mas sempre menos de 70% nas máquinas r662. A utilização de memória aumenta com a carga. Em termos de blocos escritos este é superior para a classe D e maior nas classes S e W do que nas restantes em ambas as máquinas. O número de *packets* enviados e recebidos é superior nas classes S e W nas máquinas r641 e na classe D nas máquinas r662.

No testes EP notei usos de *CPU* crescentes com a carga estagnando a partir da classe B, apesar disso estes estão mais uma vez perto dos 100% nas máquinas r641 e não chegam aos 70% nas máquinas r662. Em termos de memória encontrei os maiores nas classes A, B e C para as máquinas r641 e usos similares para as máquinas r662. Em termos de leituras de disco encontrei valores bastante superiores para a classe D nas máquinas, seguidas das classes S e W. Em termos de transmissões *tcp* nas máquinas r641 as classes S e W tem valores maiores seguidos da classe D e depois as restantes, nas máquinas r662 a classe D tem valores maiores do que as classes S e W.

Versão híbrida

Notei uma utilização de *CPU* muito baixa isto poderá ser devido a uma má configuração. Para o teste SP podemos notar um incremento do uso dos vários recursos com o aumento da carga este sendo mais acentuado no uso de disco e rede. Em termos de memória e *CPU* o uso tende a estagnar por volta dos 4-5% dependendo da máquina para o *CPU* e 3% para memória nas duas máquinas. Para o teste BT obtive resultados semelhantes, salientando que os testes da classe S utilizam ainda mais recursos de rede e disco proporcionalmente as restantes classes que no teste anterior.

Capítulo 3

Dtrace - Desenvolvimento de programas

3.1 Introdução

O trabalho apresentado neste documento consiste na criação de um conjunto de *scripts Dtrace* com funcionalidades distintas. O objetivo por detrás da criação destas *scripts* consiste em ganhar uma maior familiaridade com a linguagem D e o utilitário *Dtrace* de modo a conseguir utilizar os mesmos num trabalho futuro esse consistente em analisar diferentes implementações de um algoritmo.

Este documento descreve as várias *scripts* criadas e elabora sobre as estratégias utilizadas para a implementação das mesmas.

3.2 Ambiente utilizado

Para efeitos de teste e desenvolvimento das *scripts* foi utilizada uma maquina virtual *Oracle linux*, o virtualizador utilizado foi o *VirtualBox* e foram utilizados dois cores e 1 *GiB* de memória *RAM* nesta maquina virtual

3.2.1 Traçado das chamadas a *system call open*

A primeira *script* implementada traça as chamadas a *system call open* para abrir ficheiros com */etc* no seu caminho, para cada chamada detetada é imprimida uma linha contendo o nome do ficheiro executável associado ao processo, os *pid,uid* e *gid* associados ao processo que realizou a chamada, o caminho para o ficheiro a ser aberto, as *flags* passadas como argumento, a *probe* responsável por detetar esta chamada e o valor de retorno da chamada ao sistema.

A estratégia utilizada para implementar as funcionalidades descritas acima consistiu em primeiro guardar os valores de *arg0* e *arg1* da *probes syscall:open:entry*, esses valores correspondem ao ficheiro e *flags* utilizadas respetivamente, posteriormente é utilizada a *probe syscall:open:return* para os valores cujo caminho do ficheiro contenha */etc*, a distinção dessas chamadas com as restante é realizada utilizando a função *strstr*, é depois imprimida a informação descrita anteriormente com o auxilio da informação guardada e os valores nos campos *execname,pid,uid,gid* e *arg1* desta *probe*. O *script* associado a este programa consiste no `ex1.d`:

```
[dtrace@dtraceBase assignment2]$ ./ex1.d
```

```
cat(pid:1822,uid:1000,gid:1000) tried to open file /etc/ld.so.cache with
flags 524288 called open with return value 3
```

```
cat(pid:1822,uid:1000,gid:1000) tried to open file /etc/inittab with flags
0 called open with return value 3
```

Listing 3.1: Output do primeiro programa

```
cat /etc/inittab >> /tmp/test
```

Listing 3.2: Comandos utilizados para produzir o output do programa

3.2.2 Estatísticas relativas a abertura e criação de ficheiros

O segundo programa implementado imprime periodicamente com um valor em segundos passado como argumento os valores associados ao número de tentativas de abertura de ficheiros, o número de tentativas de criação de ficheiros e as tentativas de criar ficheiros bem-sucedidas agrupados por *PID*.

Para implementar este programa utilizei as *probes syscall:open:entry* e *syscall:open:return*. Para contabilizar as tentativas de abertura de ficheiros existentes e de novos ficheiros o programa verifica para cada entrada na *probe syscall:open:entry* se as *flags* incluem a *flag O_CREATE*, posteriormente o valor das *flags* é guardado. Para contabilizar o sucesso das operações é utilizado o argumento *errno* da *probe syscall:open:return*. A impressão periódica é feita utilizando a *probe tick* com o argumento passado na linha de comandos. O ficheiro associado a esta *script* é o *ex2.d*:

```
Date: 2020 Jun 13 23:11:12
```

```
Number of tentatives to open existing files
```

1995	cat	4
------	-----	---

```
Number of tentatives to create new files
```

1995	bash	1
------	------	---

```
Faillure to open existing files
```

```
Faillure to create news files
```

```
Success to open existing files
```

1995	bash	1
1995	cat	4

```
Success to create news files
```

1995	bash	1
------	------	---

Listing 3.3: Output do segundo programa

3.2.3 Replicação do programa *strace -c*

O terceiro programa implementado consiste numa reimplementação de algumas funcionalidades do comando *strace -c* utilizando o *Dtrace*. Estas funcionalidades consistem em medir durante a execução

de um programa passado como argumento o número de chamadas a cada *system calls* realizadas pelo mesmo e tempo despendido nas mesmas em segundos, para além disso também é imprimido o tempo de execução do programa.

Para implementar as funcionalidades descritas utilizei as *probes syscall::entry* e *syscall::return*, essas *probes* medindo o número de chamadas a cada *system call* associada ao processo a ser observado e para cada acionamento da *syscall::entry* é guardado o *timestamp* da mesma que é utilizado no acionamento da *syscall::return* no momento do seu acionamento pela mesma *syscall*. O programa é depois corrido com a opção *-c* e com o programa a ser analisado como argumento. O ficheiro associado a esta *script* é o *ex3.d*.

```
time spent in seconds since the start of this program 12
system calls frequency

access 2
arch_prctl 2
exit_group 2
fadvise64 2
munmap 2
write 2
read 6
brk 8
mprotect 8
open 8
newfstat 10
close 12
mmap 16

time spent on each system call in seconds

fadvise64 0
arch_prctl 0
access 0
write 0
munmap 0
read 0
newfstat 0
brk 0
close 0
mprotect 0
open 0
mmap 0
```

Listing 3.4: Output do terceiro programa

3.3 Conclusões

A implementação destes programas permitiu ganhar alguma familiaridade com a ferramenta *Dtrace* esta que será útil em trabalhos futuros nomeadamente uso desta ferramenta para a análise e deteção de problemas de performance de aplicações existentes tal como a própria monitorização e recolha de estatísticas sobre o uso de vários sistemas.

Capítulo 4

Dtrace - Utilização de *Dtrace* para a análise de radix sort

4.1 Introdução

Este documento relata o trabalho realizado para a análise de implementações sequenciais, paralelas de memória distribuída e partilhada do algoritmo *radix sort MSD* utilizando a ferramenta *Dtrace*.

O *Dtrace* é uma ferramenta de traçado dinâmico desenvolvida pela *Sun Microsystems*, esta ferramenta foi criada com intuito de ser utilizada para diagnosticar problemas relacionados com as aplicações e *kernel* em tempo real. Relativamente a outras ferramentas de traçado como por exemplo o *strace* esta tem a vantagem de poder ser utilizada em produção.

Este documento está organizado em vários capítulos o primeiro capítulo descreve os algoritmos e algumas opções tomadas nas várias implementações no capítulo subsequente são apresentadas as *probes* criadas e utilizadas na análise dos mesmos, posteriormente está o capítulo contendo os resultados e alguma discussão e análise dos mesmos, posteriormente são apresentadas algumas conclusões relativas ao trabalho realizado.

4.2 Algoritmo utilizado

4.2.1 Versão sequencial

O algoritmo utilizado neste caso de estudo consiste no *radix sort MSD*. A implementação utilizada ordena um *array* dividindo-o em *buckets* de acordo com os seus *n bits* mais significativos, nesta implementação $n = 128$.

Após a sua ordenação inicial em *buckets* esta função é chamada recursivamente para cada um dos *buckets* com os *n bits* mais significativos seguintes, até ao caso de paragem em que o *array* a ser ordenado tem tamanho inferior ou igual a 1 ou não existirem mais *bits* para ordenar.

Algorithm 1 Radix Sort sequential with fixed number of buckets

Input:array of integers array,integer size,integer digit.

Output:none

```
1: for  $i \in [0, size[$  do
2:    $digit \leftarrow get\_digit(array[i], 0)$ 
3:    $count[digit]++$ 
4: end for
5: for  $i \in [0, size[$  do
6:    $digit \leftarrow get\_digit(array[i], 0)$ 
7:    $temp[count[digit] + inserted[digit]++] \leftarrow array[i]$ 
8: end for
9: for  $i \in [0, size[$  do
10:   $array[i] \leftarrow temp[i]$ 
11: end for
12: for  $i \in [1, NR\_BUCKETS[$  do
13:   $start[i] \leftarrow start[i-1] + count[i]$ 
14: end for
15: for  $i \in [0, size[$  do
16:   $radix\_sort(array[start[i]:], count[i], digit + 1)$ 
17: end for
```

4.2.2 Versões paralelas

Versão em memória partilhada *Omp*

A estratégia utilizada nesta implementação consistiu em após a primeira iteração ordenar cada um dos *buckets* em paralelo. Esta implementação também realiza outras secções do algoritmo como o cálculo do tamanho de cada *bucket*, a ordenação em *buckets* e a copia do *array* temporário para o *array* principal em paralelo.

Versão em memória distribuída *MPI*

O algoritmo de base para a implementação em memória distribuída consiste em inicialmente realizar um primeira ordenação em *buckets* de acordo com os primeiros n bits mais significativos pelo processo *master*. Após ter realizado esta ordenação este processo divide a carga pelos processos *slave* e por si mesmo de modo a esta ser distribuída o mais equilibradamente possível, para tal este algoritmo utiliza um algoritmo de distribuição de carga que ordena os **buckets** da forma "*highest size bucket, lowest size bucket, second highest size bucket, second lowest size bucket, ...*", o que permite mitigar problemas de distribuição de carga ocorrentes no caso de um processo possuir um número significativo de *buckets* maiores. Esta implementação utiliza as primitivas *scatterv* e *gatherv* para distribuir os *buckets* pelos restantes processos e a primitiva *broadcast* para partilhar os tamanhos e os *buckets* a serem ordenados por cada um dos processos.

Versão em memória partilhada *Pthreads*

Esta implementação é semelhante a implementação *Omp*, a principal diferença relativamente a esta consiste na distribuição dos *buckets* ordenados em paralelo ser feito utilizando o algoritmo de divisão de carga utilizado para a versão *MPI*. As restantes diferenças comparativamente com a versão *Omp* consistem no uso de primitivas da biblioteca *Pthreads* em vez das de *Omp*. O número de *threads* é também passado como argumento ao programa em vez de ser controlado por uma variável de ambiente.

Versão em memória partilhada *C++11*

Para utilizar mais funcionalidades do *C++* em vez de utilizar um *array* auxiliar optei por utilizar a primitiva *vector* de *c++*, a utilização desta primitiva permite utilizar menos recursos relativamente a computação dos *digitos* sendo que apenas é necessário percorrer o *array* uma vez por cada chamada a função em vez de duas uma para contar e outra para inserções e esta primitiva preserva alguma eficiência nos acessos a memória sendo que as suas inserções tem um peso similar a inserções num *array*. As restantes diferenças comparativamente a implementação *Pthreads* consistem em adaptações para o uso da primitiva *Thread* do *C++* em vez das primitivas *Pthreads*.

4.3 Probes utilizadas

4.3.1 Custom probes

Sequencial

Probe	argumentos	descrição
<i>start_sorting_into_buckets</i> <i>finish_sorting_into_buckets</i>	(int, int)	Indicam o início e fim da ordenação dos elementos de um <i>array</i> em <i>buckets</i> de acordo com os seus $n * d - n * (d + 1)$ <i>bits</i> . O argumento arg_0 corresponde ao tamanho do <i>array</i> e arg_1 a d .
<i>start_seq_radix</i> <i>finish_seq_radix</i>	(int, int)	Indicam o início e fim de uma execução da função de <i>radix sort</i> sequencial. O argumento arg_0 indica o tamanho do <i>array</i> e arg_1 os <i>bits</i> pelos quais o <i>array</i> esta a ser ordenado.
<i>start_count_digits</i> <i>finish_count_digits</i>	(int, int)	Indicam o início e fim das contagens de cada ocorrência dos bits pelos quais o <i>array</i> esta a ser ordenado. Os argumento arg_0 representa o tamanho do <i>array</i> e arg_1 esses <i>bits</i> .
<i>start_insert_into_buckets</i> <i>finish_insert_into_buckets</i>	(int, int)	Indica o fim e o início da inserção nos <i>buckets</i> . O argumento arg_0 indica o tamanho do <i>array</i> e arg_1 os <i>bits</i> pelos quais o <i>array</i> esta a ser ordenado.
<i>start_copy_to_main_array</i> <i>finish_copy_to_main_array</i>	(int, int)	Indicam o fim e o início da copia do <i>array</i> temporário para o <i>main array</i> . O argumento arg_0 indica o tamanho do <i>array</i> e arg_1 os <i>bits</i> pelos quais o <i>array</i> esta a ser ordenado.
<i>start_allocate_temp_array</i> <i>finish_allocate_temp_array</i>	(int, int)	Indicam o início e fim da alocação do array temporário. O argumento arg_0 indica o tamanho do <i>array</i> e arg_1 os <i>bits</i> pelos quais o <i>array</i> esta a ser ordenado.

Tabela 4.1: *Custom probes* definidas para a versão sequencial

time spent sorting into buckets

3	3
1	3
2	3

array sizes per digit

1			
value	—————	Distribution	————— count
4194304			0

```

8388608 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 5
16777216 |                                                                 0

      2
value  ----- Distribution ----- count
2097152 |                                                                 0
4194304 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 10
8388608 |                                                                 0

      3
value  ----- Distribution ----- count
16384  |                                                                 0
32768  |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1280
65536  |                                                                 0

time spent per digit

      4          0
      3          4
      2          8
      1         12

calls by digit

      1          5
      2         640
      3        1280
      4       163840

time spent in different sections of the program

allocate temp array          0
copy to main array          0
sorting into buckets        10
total                      38

```

Listing 4.1: Output das *probes* sequenciais para a implementação sequencial utilizando tamanho 10000000

O uso destas *probes* consiste maioritariamente em examinar o tempo gasto em várias secções do programa para tal foram criadas várias sondas que permitem obter quando o programa entra e sai de determinadas secções. Tal como informação acerca dos tamanhos dos *arrays* a serem ordenados e o "dígito" a ser ordenado.

O resultado destas *probes* consiste no tempo gasto na secção de ordenação dos elementos em *buckets* em segundos, a distribuição dos tamanhos dos *arrays* pelos dígitos a serem ordenados, o tempo gasto na ordenação de cada dígito incluindo as chamadas recursivas e as chamadas para a ordenação de cada "dígito".

OpenMp

Para além das *probes* definidas para a versão sequencial que foram adaptadas para medir não apenas as chamadas recursivas mas também as ocorrências em todas as secções do algoritmo e devolver informação relativamente ao *bucket* paralelo ou secção sequencial a ser ordenado ,a implementação *OpenMp* possui também as seguintes *probes*.

Probe	argumentos	descrição
<i>start_par_radix</i> <i>finish_par_radix</i>	(int, int, int)	Indicam o inicio e fim da execução do função paralela. O argumento arg_0 indica o tamanho do <i>array</i> , o arg_1 os <i>bits</i> pelos quais o <i>array</i> esta a ser ordenado e arg_2 devolve o <i>bucket</i> .

Tabela 4.2: Custom probes definidas para a versão OpenMp

time spent sorting into buckets

```

1          1
3          1
2          1

```

array sizes per digit

```

1
value ----- Distribution ----- count
2097152 | 0
4194304 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 4
8388608 | 0

```

```

2
value ----- Distribution ----- count
1048576 | 0
2097152 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 10
4194304 | 0

```

```

3
value ----- Distribution ----- count
8192 | 0
16384 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1280
32768 | 0

```

time spent per digit

calls by digit

```

2          640
3          1280
4          163840

```

time spent in different sections of the program

```

allocate temp array          0
copy to main array          0
total                        3
sorting into buckets        4

```

Listing 4.2: Output das probes Omp para a implementação sequencial utilizando tamanho 5000000

Tal como para as *probes USDT* criadas para a versão sequencial a função principal destas *probes* consiste em medir o tempo gasto em diferentes secções do programa sejam estas do componentes do algoritmo ou chamadas recursivas para a ordenação de um dígito.

O resultado do *script* criado utilizando estas *probes* é o mesmo que na versão anterior.

Mpi

Para a versão *Mpi* para além das *probes* definidas para as versões *Omp* para as quais o terceiro argumento passou a ser o processo em vez do *bucket* ,foram definidas as seguintes *probes*.

Probe	argumentos	descrição
<i>start_workload_distribution</i> <i>finish_workload_distribution</i>	(int, int, int)	Indicam o início e fim do processo de distribuição do trabalho pelos vários processos pelo processo <i>Master</i> .O argumento arg_0 indica o tamanho do <i>array</i> ,o arg_1 os <i>bits</i> pelos quais o <i>array</i> esta a ser ordenado e o arg_2 corresponde ao <i>rank</i> do processo.
<i>start_scatter_workload</i> <i>finish_scatter_workload</i>	(int, int, int)	Indicam o início e fim do envio das porções do <i>array</i> a serem ordenados pelos <i>Slaves</i> por parte do processo <i>Master</i> .O argumento arg_0 indica o tamanho do <i>array</i> ,o arg_1 os <i>bits</i> pelos quais o <i>array</i> esta a ser ordenado e o arg_2 indica o <i>rank</i> do processo.
<i>start_gather_workload</i> <i>finish_gather_workload</i>	(int, int, int)	Indicam o início e fim da receção das porções do <i>array</i> a serem <i>ordenadas</i> pelos processos <i>Slave</i> por parte do <i>Master</i> .O argumento arg_0 indica o tamanho do <i>array</i> ,o arg_1 os <i>bits</i> pelos quais o <i>array</i> esta a ser ordenado e o arg_2 o <i>rank</i> do processo.
<i>start_receive_workload</i> <i>finish_receive_workload</i>	<i>int</i>	Indicam o início e fim das receções das porções do <i>array</i> a serem ordenadas pelos processos <i>Slave</i> do processo <i>Master</i> .O argumento arg_0 refere-se ao <i>rank</i> do processo.
<i>start_send_workload</i> <i>finish_send_workload</i>	<i>int</i>	Indicam o início e fim do envio das porções do <i>array</i> ordenadas pelos processos <i>Slave</i> ao processo <i>Master</i> .O argumento arg_0 representa o <i>rank</i> do processo.

Tabela 4.3: Custom probes definidas para a versão Mpi

time spent sorting into buckets

3	1
1	1
2	1
array sizes per digit	
1	
value	Distribution
2097152	0
4194304	5
8388608	0
2	
value	Distribution
1048576	0
2097152	10
4194304	0
3	
value	Distribution
	count


```

262144 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 10
524288 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 0

3
value  Distribution count
1024 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 0
2048 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1278
4096 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 2
8192 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 0

time spent per digit

3 0
2 1
1 4

calls by digit

1 4
2 640
3 1280
4 163840

time spent in different sections of the program

workload distribution 0
allocate temp array 0
copy to main array 0
sorting into buckets 4
total 5
recursive calls 6

```

Listing 4.4: Output das *probes Pthreads* para a implementação *Pthreads* utilizando tamanho 1000000 e 8 *threads*

C++11

Para esta implementação foram definidas estas *probes* presentes para a versão *Pthreads* *start_sorting_into_buckets*, *finish_sorting_into_buckets*, *start_seq_radix*, *finish_seq_radix*, *start_par_radix*, *finish_par_radix*, *start_copy_to_main_array*, *finish_copy_to_main_array*, *start_workload_distribution* e *finish_workload_distribution*.

```

time spent sorting into buckets

2 0
3 1
1 4

array sizes per digit

1
value  Distribution count
262144 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 0
524288 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 4
1048576 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 0

2
value  Distribution count

```



```

131072 | 0
262144 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 10
524288 | 0

3
value  Distribution count
1024 | 0
2048 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1278
4096 | 2
8192 | 0

time spent per digit

3 1
2 2
1 6

calls by digit

1 4
2 640
3 1280
4 163840

time spent in different sections of the program

workload distribution 0
copy to main array 0
sorting into buckets 6
total 8
recursive calls 10

```

Listing 4.5: Output das *probes C++11* para a implementação *C++11* utilizando tamanho 1000000 e 8 *threads*

4.3.2 CPC probes

Probe	Uso
<i>PAPI_l1_dcm-all-5000</i>	Obter de uma forma aproximada as <i>l1 data cache misses</i> .
<i>PAPI_l2_dcm-all-5000</i>	Obter de uma forma aproximada as <i>l2 data cache misses</i> .
<i>PAPI_l1_icm-all-5000</i>	Obter de uma forma aproximada as <i>l1 instruction cache misses</i> .
<i>PAPI_l2_icm-all-5000</i>	Obter de uma forma aproximada as <i>l2 instruction cache misses</i> .

Tabela 4.4: *Custom probes* definidas para a versão *Mpi*

```

data cache misses l1

11 cache data misses 5510000

data cache misses l2

12 cache data misses 695000

```

```

instruction cache misses l1

    l1 cache instruction misses                                1600000

instruction cache misses l2

    l2 cache instruction misses                                100000

```

Listing 4.6: Output das *probes cpc* para a implementação *Omp* utilizando tamanho 50000000 e 1 *thread*

O objetivo por detrás do uso destas *probes* consiste em medir vários tipos de *cache misses* ocorrentes durante a execução do programa.

O resultado destas *probes* consiste nas *cache misses l1* e *l2* de dados e instruções.

4.3.3 SYSINFO probes

Probe	Uso
<i>nthreads</i>	Obter o número de <i>threads</i> criado.
<i>pswitch</i>	Obter o número de <i>cpu switches</i> das <i>threads</i> .
<i>procovf</i>	Obter falhas na criação de processos.
<i>bwrite</i>	Obter número e tamanho das escritas.
<i>bread</i>	Obter número e tamanho das escritas.
<i>inv_swch</i>	Obter número de <i>switches</i> involuntarios.
<i>sysfork</i>	Obter número de chamadas a <i>system call fork</i> .
<i>sysexec</i>	Obter número de chamadas a <i>system call exec</i> .

Tabela 4.5: *Custom probes* definidas para a versão *Mpi*

```

info collected from the system

    number of threads created                                1
    number of times threads where forced to give up cpu      50
    number of cpu switches between threads                   87

```

Listing 4.7: Output das *probes sysinfo* para a implementação *Omp* utilizando tamanho 50000000 e 1 *thread*

O objetivo destas *probes* consiste em medir informação relativa a processos do sistema.

O resultado destas *probes* consiste em contagem e quantificações destes eventos.

4.3.4 PLOCKSTAT probes

Probe	Uso
<i>mutex-block</i> <i>mutex-spin</i> <i>mutex-release</i> <i>mutex-error</i>	Obter os tempos de espera pelos e retenção dos <i>mutexes</i> .

Tabela 4.6: Custom probes definidas para a versão *Mpi*

```

info collected from the system

```

```

wait for mutex
value      Distribution      count
1024      |                  0
2048      |@@@@             2
4096      |@@              1
8192      |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 12
16384     |@@@@@@@@@@       4
32768     |                  0

time holding mutex
value      Distribution      count
2048      |                  0
4096      |@@              1
8192      |@@@@             2
16384     |                  0
32768     |                  0
65536     |                  0
131072    |                  0
262144    |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 16
524288    |                  0

```

Listing 4.8: Output das probes plockstat para a implementação *Omp* utilizando tamanho 10000000 8 threads

O objetivo destas *probes* consiste em criar quantificações relativas aos tempos de espera e de retenção de *mutexes*.

4.3.5 SCHED probes

Probe	Uso
<i>on-cpu</i>	Obter os tempo total gasto em cada <i>CPU</i> e os tempos por cada
<i>off-cpu</i>	<i>strech</i> por cada em cada <i>CPU</i> .

Tabela 4.7: *Custom probes* definidas para a versão *Mpi*

time spent per stretch per cpu

```

6
value      Distribution      count
8192      |                  0
16384     |@@@@@@@@@@       3
32768     |@@@@@@@@@@       3
65536     |                  0
131072    |                  0
262144    |                  0
524288    |                  0
1048576   |                  0
2097152   |                  0
4194304   |                  0
8388608   |@@@              1
16777216  |                  0
33554432  |                  0

```

```

67108864 |@@@@@@@@@@@@@@@@@@@@ 6
134217728 |@@@@@@ 2
268435456 | 0

...

5
value      Distribution      count
8192      | 0
16384     |@@ 1
32768     |@@ 1
65536     | 0
131072    | 0
262144    | 0
524288    | 0
1048576   | 0
2097152   | 0
4194304   | 0
8388608   | 0
16777216  |@@@@@ 2
33554432  |@@@@@ 2
67108864  |@@@@@ 2
134217728 |@@@@@@@@@@@@@@@@@@@@ 9
268435456 | 0

```

time per cpu

```

6      0
4      1
2      1
7      1
1      1
0      1
3      1
5      2

```

Listing 4.9: Output das probes sched para a implementação *Omp* utilizando tamanho 100000000 e 8 *threads*

O objetivo destas *probes* consiste em quantificar o tempo em cada *core* por *stretch* e somar o tempo total do programa em cada *core*.

4.3.6 VMINFO probes

Probe	Uso
<i>todas</i>	Obter as ocorrências dos diferentes eventos na execução do programa.

Tabela 4.8: Probes VMINFO

info collected from the system

```

number of threads created                                7
number of times threads where forced to give up cpu      131
number of cpu switches between threads                   263

```

Listing 4.10: Output das *probes vminfo* para a implementação *Omp* utilizando tamanho 100000000 e 8 *threads*

Quantifica varias informações dessas *probes*.

4.3.7 libmpi probes

Probe	Uso
<i>MPI_Bcast</i> <i>MPI_Scatterv</i> <i>MPI_Gathrev</i> <i>MPI_Send</i> <i>MPI_Recv</i>	Obter o tempo gasto em envios de mensagens utilizando esta primitiva e tamanhos das mensagens enviadas com a mesma.

Tabela 4.9: Probes libmpi

Communication times using different primitives

```

gatherv                                                    0
bcast                                                       0
Integers sent or received for each function call

```

```

bcast
value  Distribution count
  64   | 0
 128   | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 10
 256   | 0

gatherv
value  Distribution count
2097152 | 0
4194304 | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 80
8388608 | 0

```

Communication times using different primitives

```

gatherv                                                    0
bcast                                                       0
Integers sent or received for each function call

```

```

bcast
value  Distribution count
  64   | 0
 128   | @@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 10
 256   | 0

gatherv

```

value	Distribution	count
-1		0
0	@@@	5
1		0
2		0
4		0
8		0
16		0
32		0
64		0
128		0
256		0
512		0
1024		0
2048		0
4096		0
8192		0
16384		0
32768		0
65536		0
131072		0
262144		0
524288		0
1048576		0
2097152		0
4194304	@@	75
8388608		0

Listing 4.11: Output das probes libmpi para a implementação mpi utilizando tamanho 100000000 com 8 processos

Quantifica várias informações relativas as *probes*.

Estas *probes* foram utilizadas para quantificar o tamanho das mensagens enviadas e medir o tempo de comunicação total utilizado por cada uma das primitivas utilizadas na implementação em memória distribuída.

4.4 Teste e Resultados

4.4.1 Metodologia e ambiente de teste

As varias implementações foram corridas utilizando números variáveis de processos e *threads* para as implementações paralelas numa maquina virtual *Solaris*,com um *CPU AMD*,os vários *scripts* criados foram utilizados para obter estatísticas acerca da execução dos programas.

Para o efeito de testes foram utilizados *arrays* de tamanhos 1000000, 5000000 e 10000000 e cada medição efetuada mede a ordenação de um *array* de um determinado tamanho 5 vezes. Os testes foram automatizados utilizando *bash scripts* e foram também criados gráficos que permitam melhor visualizar alguns dos resultados obtidos.

4.4.2 Resultados

Versão Sequencial

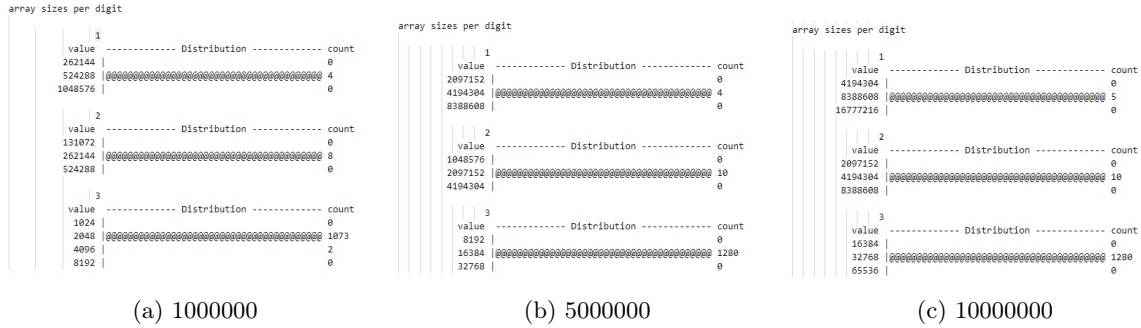


Figura 4.1: Distribuição do tamanho dos *arrays* a serem ordenados pelas chamadas as funções para diferentes *bits* para diferentes tamanhos do *array* original.

Os testes realizados permitem verificar que existe uma distribuição uniforme do *array* utilizado para teste, isto acontece porque este *array* é gerado aleatoriamente com uma distribuição uniforme. Visto que os *arrays* são gerados sequencialmente para todas as versões do programa é esperado que os valores do mesmo sejam uniformes para todas as versões, isto foi verificado com os testes realizados nas outras versões.

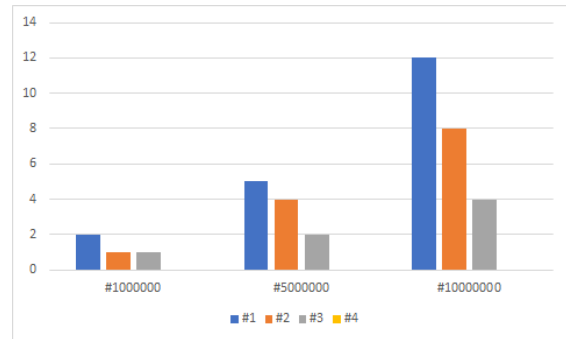
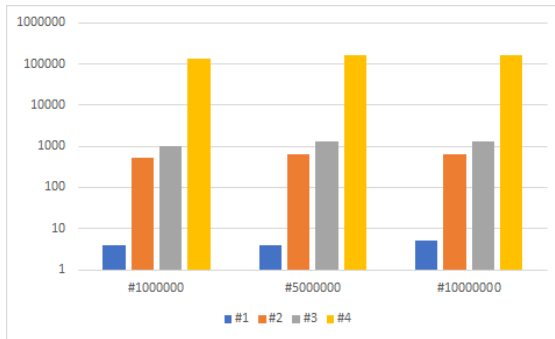


Figura 4.2: número de chamadas as funções *radixsort* por dígito

Figura 4.3: Tempo gasto por dígito em segundos em escala logarítmica com base 10

A partir dos testes pude verificar um aumento de tempo considerável gasto no cálculo do primeiro conjunto de *bits* comparativamente com os restantes a medida que o tamanho de input aumenta. Como seria esperado pelo facto dos *arrays* serem distribuídos uniformemente e por serem de tamanho substancial o número de chamadas as funções por cada dígito tem pouca variação com o aumento do tamanho de dados.

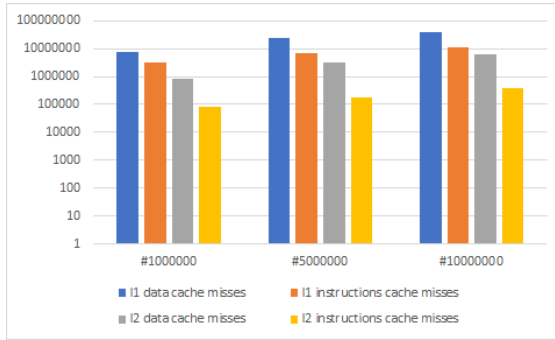


Figura 4.4: número de cache misses para l1 e l2
valor absoluto escala logarítmica base 10

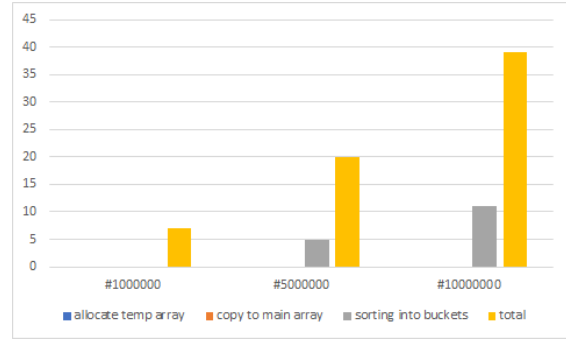


Figura 4.5: tempo gasto por secção em segundos

Para os vários tamanhos pode-se verificar que o número de *cache misses* relacionadas com os dados é bastante superior as de instruções, neste programa isto era esperado pelo tamanho dos dados e por este possuir varias operações que envolvem copiar valores entre posições de memória. Pude também notar um aumento dos vários tipos de *cache misses* com o aumento do tamanho do *array* original. Pude notar também a secção *sorting into buckets* representa uma maior secção de execução do programa relativamente as restantes.

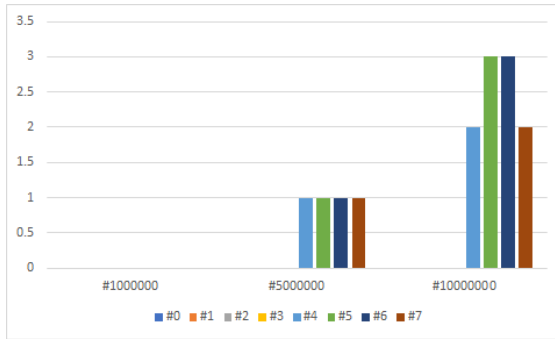


Figura 4.6: Tempo total por *thread* em segundos

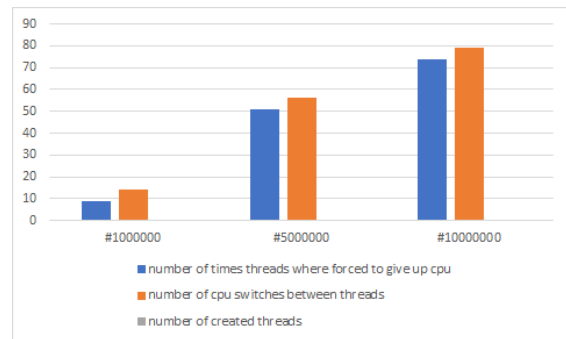


Figura 4.7: *threads* criadas pelo programa, *CPU switches* e *threads* serem obrigadas a abdicar de *CPU* valor absoluto

A partir dos testes pude também notar que a versão sequencial do programa executa em diferentes *threads* do processador, este salto entre *threads* poderia explicar alguns problemas de performance devido ao uso menos eficiente de cache sendo que não permite que o programa utilize os valores já incluídas numa cache. Este problema subsede parcialmente por o ambiente de teste ser partilhado, o mesmo poderia não acontecer no caso de este ser o único programa em execução. Com o aumento do tamanho verifiquei também um aumento considerável das comutações de contexto.

Versão OpenMp

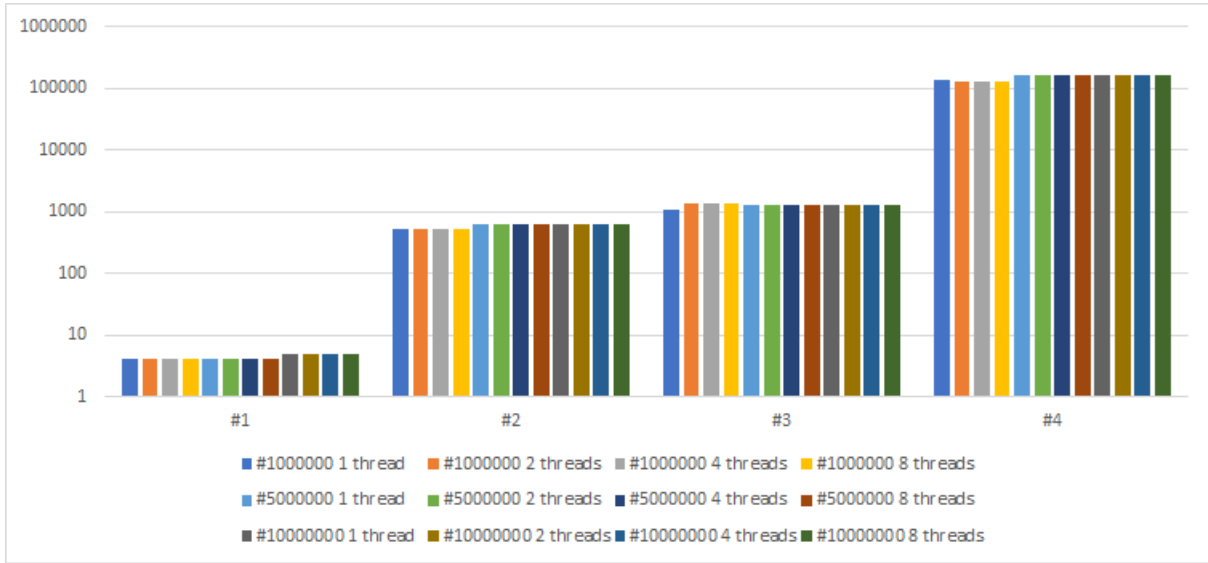


Figura 4.8: número de chamadas as funções *radix sort* por dígito

Como seria esperado pude verificar que o número de chamadas não varia consoante o tamanho ou o número de *threads*.

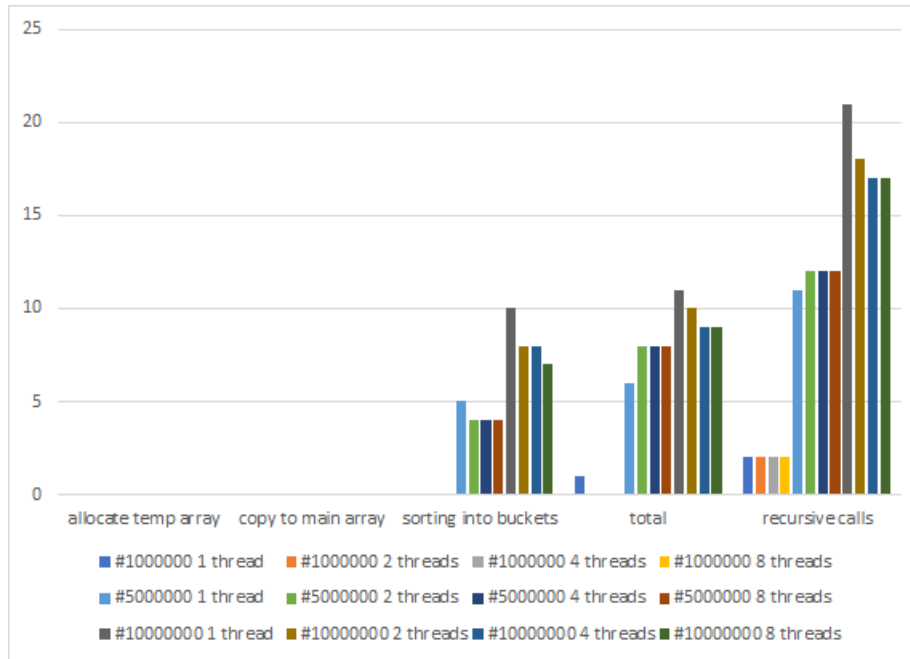


Figura 4.9: tempo gasto por secção em segundos

Em termos de tempo gasto por secção do programa verifiquei que o tempo gasto na ordenação em *buckets* continua a ser superior ao tempo gasto na copia para o *array* original e o tempo de alocação do *array* temporário. Algo verificado que seria inesperado é o facto de o tempo cumulado das chamadas recursivas ser consideravelmente superior para o maior tamanho, isto poderá acontecer devido a existir possivelmente um maior número de saltos entre diferentes localizações das *threads*.

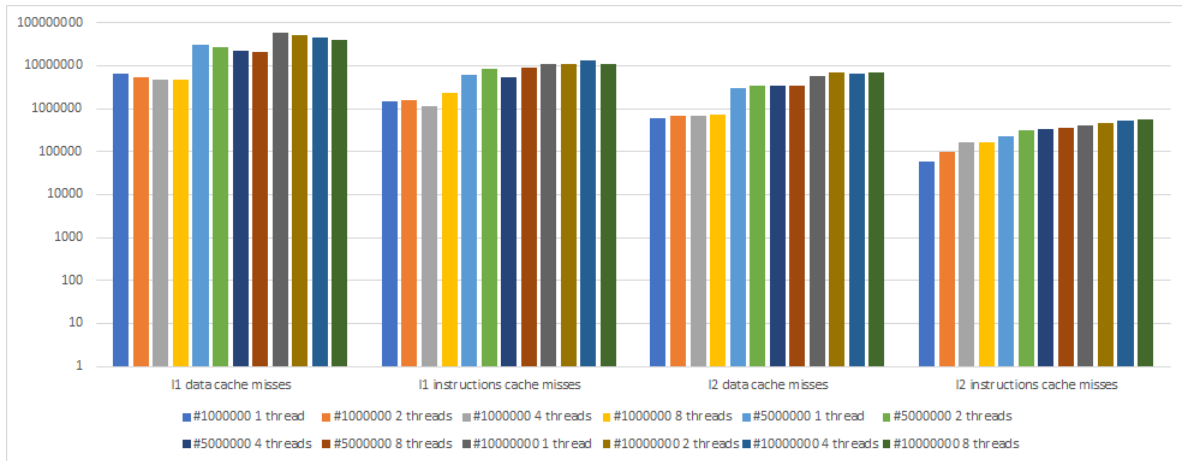


Figura 4.10: número de cache misses para $l1$ e $l2$ valor absoluto escala logarítmica base 10

Em termos de *cache misses* como seria esperado estas são maiores para dados e para $l1$ e tendem a aumentar com o tamanho dos dados. Relativamente ao número de *threads* verifica-se que estas diminuem com o número de *threads* para dados em $l1$ mas possuem uma tendência a aumentar com o número de *threads* para os restantes tipos de *misses* e níveis.

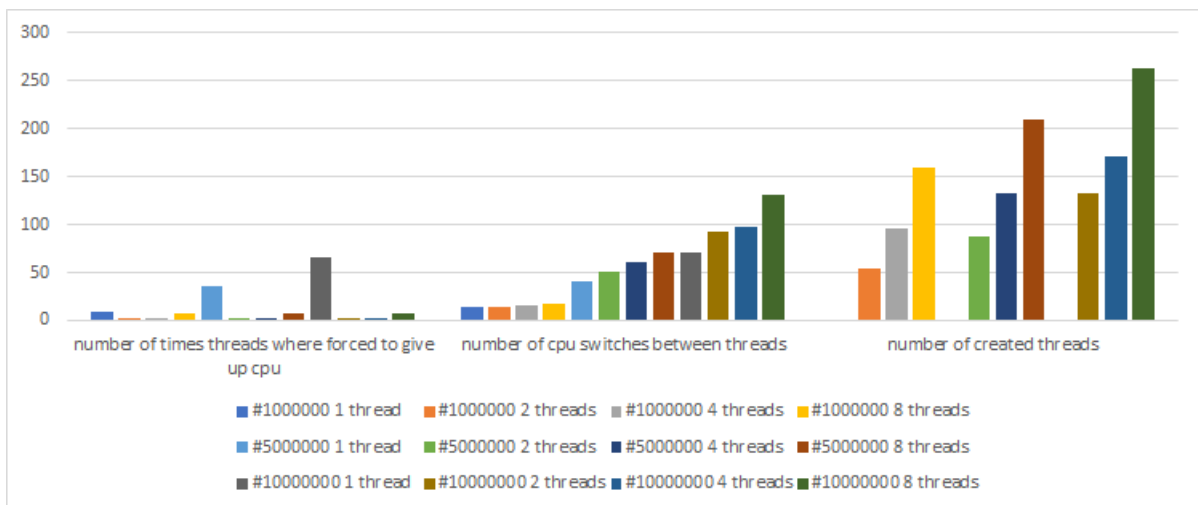


Figura 4.11: *threads* criadas pelo programa, *CPU switches* e *threads* serem obrigadas a abdicar de *CPU* valor absoluto

Verifiquei também um aumento das *threads* criadas com o aumento do número de *threads* máximo, verifiquei também que ocorre um maior número de comutações de contexto com um aumento do número de *threads* para o tamanho maior o que explica os resultados vistos anteriormente.

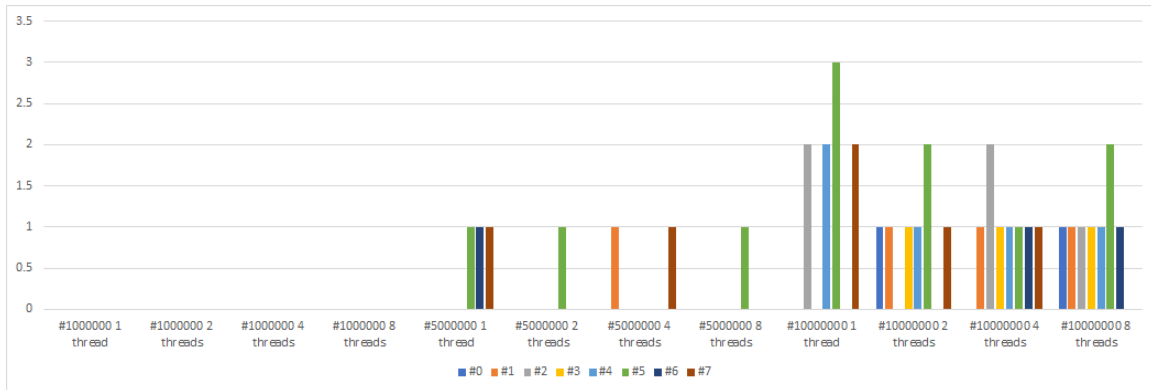


Figura 4.12: Tempo total por *thread* em segundos

É possível concluir que a carga foi realizada por varias *threads* do *CPU*. Verifiquei também que a carga não está bem distribuída pela varias *threads* do *CPU* mesmo para os maiores tamanhos isto pode ser dado por existir uma secção do programa realizada sequencialmente.

Versão Mpi

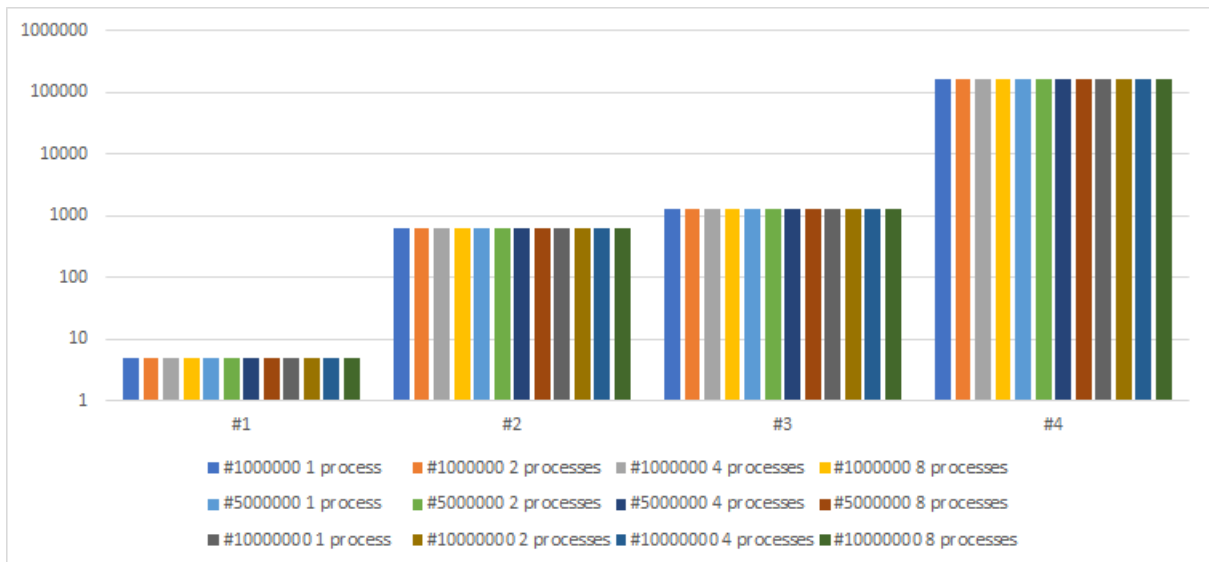


Figura 4.13: número de chamadas as funções *radix sort* por dígito

Mais uma vez o número de chamadas por cada dígito apresenta pouca variação com o número de *threads* e tamanho do problema.

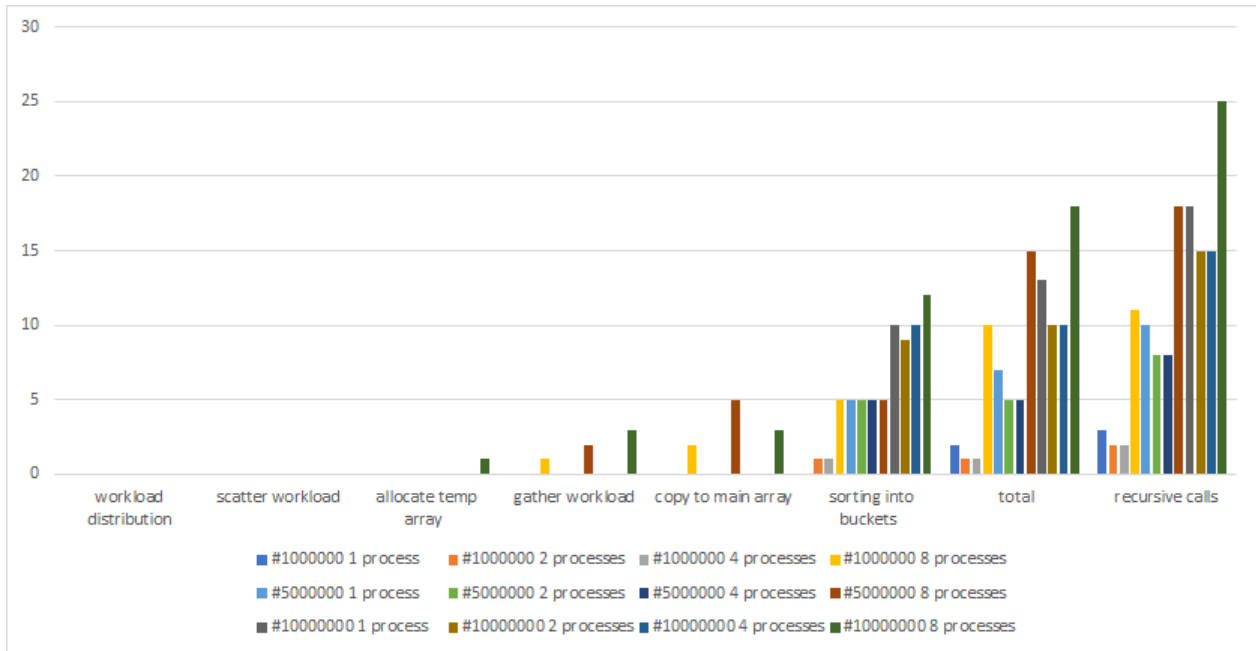


Figura 4.14: tempo gasto por secção em segundos

Comparativamente a versão em memória partilhada a versão em memória distribuída apresenta tempos mais elevados de copia para o *array* principal, isto poderá acontecer devido ao maior uso de memória devido a existência de comunicação entre os processos. verifiquei também piores tempos passando de 2 e 4 *threads* para 8 *threads* essas são dadas maioritariamente pelos acréscimos no tempo de comunicação.

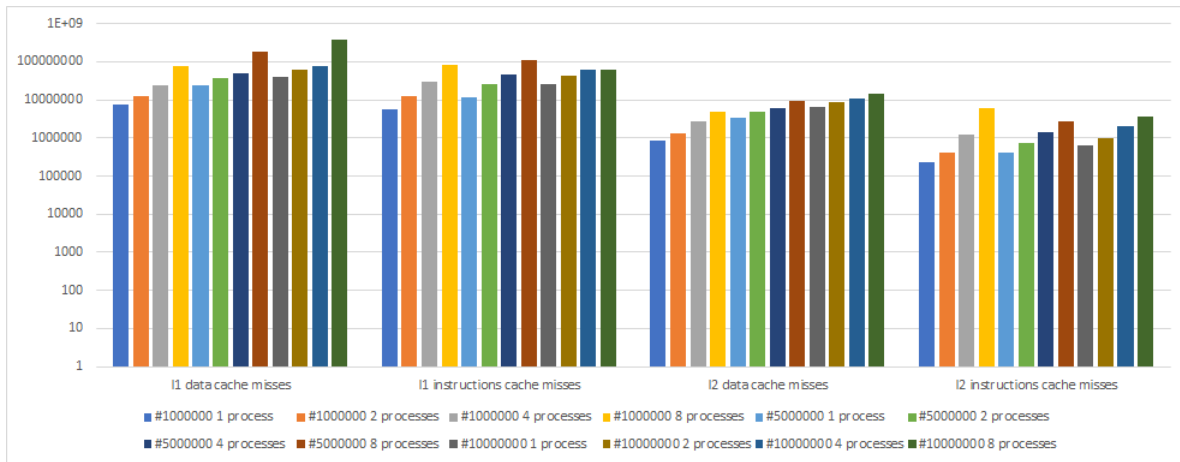


Figura 4.15: número de cache misses para l1 e l2 valor absoluto escala logarítmica base 10

Em termos de cache misses os resultados são semelhantes a versão em memória partilhada *OMP*.

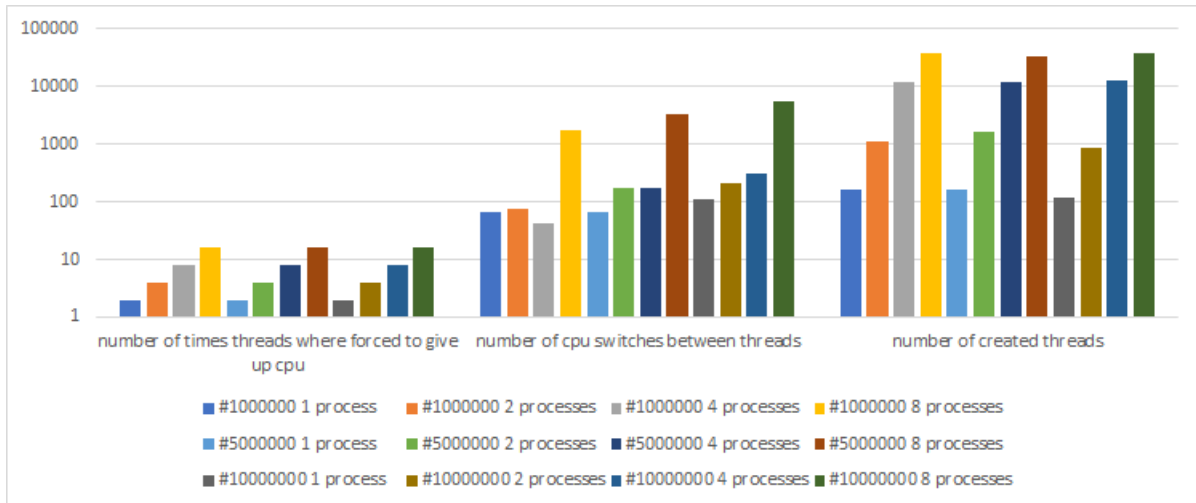


Figura 4.16: *threads* criadas pelo programa, *CPU switches* e *threads* serem obrigadas a abdicar de *CPU* valor absoluto

Em termos de comutações de contexto estes aumentam com o número de cores, mas não com o tamanho do problema, estes valores explicam alguns dos problemas de performance encontrados.

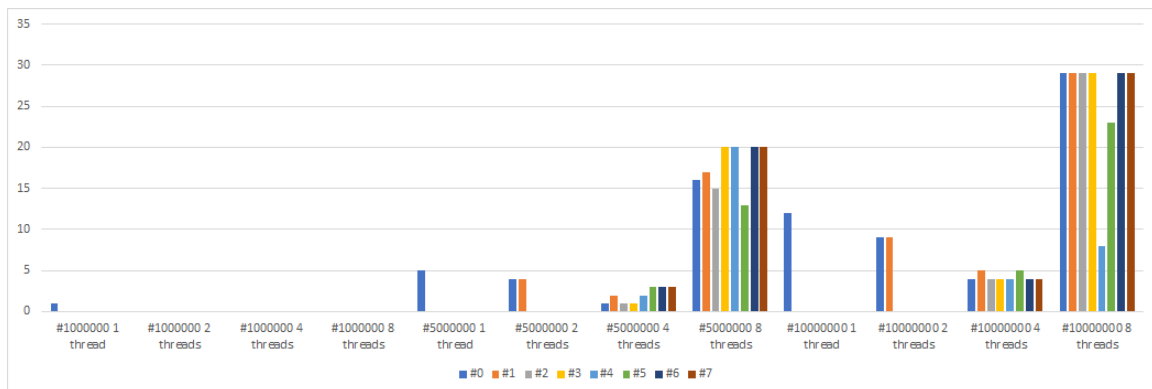


Figura 4.17: Tempo total por *thread* em segundos

Para 8 *threads* e os dois maiores tamanhos é possível verificar uma distribuição não uniforme da carga.

bcast	
value	count
64	0
128	10
256	0

gatherv	
value	count
-1	0
0	5
1	0
2	0
4	0
8	0
16	0
32	0
64	0
128	0
256	0
512	0
1024	0
2048	0
4096	0
8192	0
16384	0
32768	0
65536	0
131072	0
262144	0
524288	0
1048576	0
2097152	0
4194304	75
8388608	0

Figura 4.18: Distribuição das mensagens pelo seu tamanho

Relativamente a comunicação realizada pode verificar que esta é realizada maioritariamente utilizando as primitivas *gatherv* e *scatterv* e as restantes utilizando *broadcast* em termos de tempo gasto nas comunicações medir o tempo passado entre as entradas e retornos das funções permitiu visualizar corretamente os tempos de comunicação totais sendo que estas não chegam a um segundo o que não indica eventuais tempos de espera para o acesso a estes dados visto que o programa poderá estar a utilizar *buffer* para realizar as escritas nas novas posições de memória. Em termos de distribuição do tamanho de dados enviado por mensagens este é distribuído uniformemente em cada uma das primitivas utilizadas.

Versão Pthreads

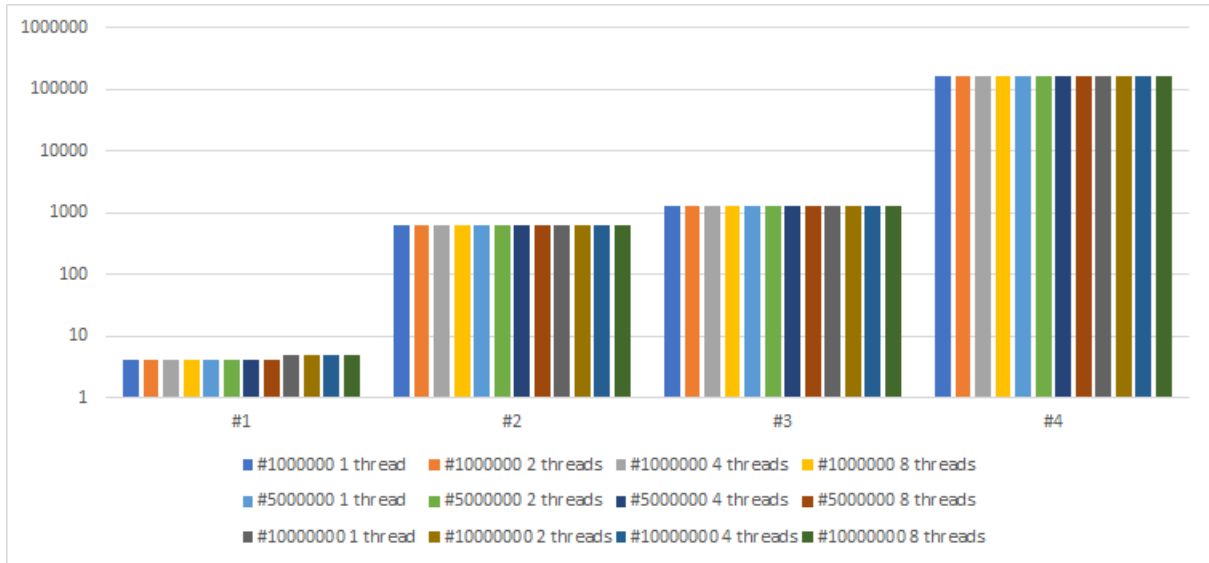


Figura 4.19: número de chamadas as funções *radix sort* por dígito

Como seria esperado pude verificar que o número de chamadas não varia consoante o tamanho ou o número de *threads*.

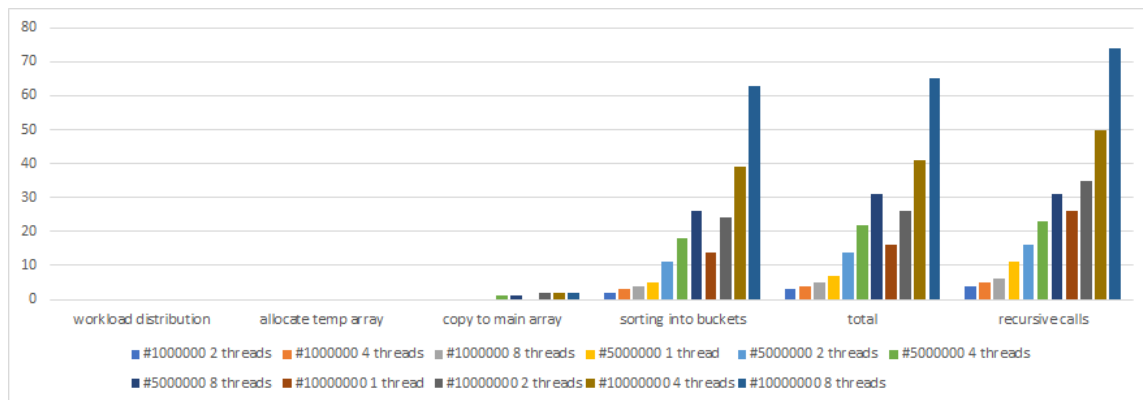


Figura 4.20: tempo gasto por secção em segundos

Algo que foi possível verificar na execução desta aplicação foi que para os vários tamanhos e nas varias secções os tempos de execução aumentam com o número de *threads* utilizado.

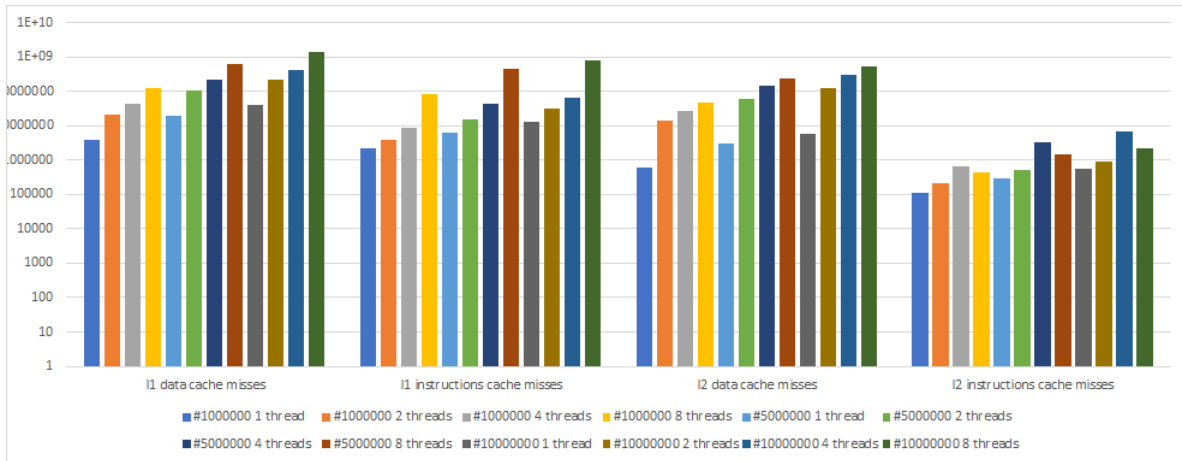


Figura 4.21: número de cache misses para l1 e l2 valor absoluto escala logarítmica base 10

Mais uma vez as *cache misses* para *l1* são maiores do que as de *l2* e as de dados do que as de instruções. Verifica-se também um aumento das *misses* com o número de *threads*, isto acontece devido a menor eficiência nos acessos a memória devido a vários fatores como a partilha de recursos pelas *threads*, *false sharing* e as *threads* serem movidas entre os *CPUs* da máquina.

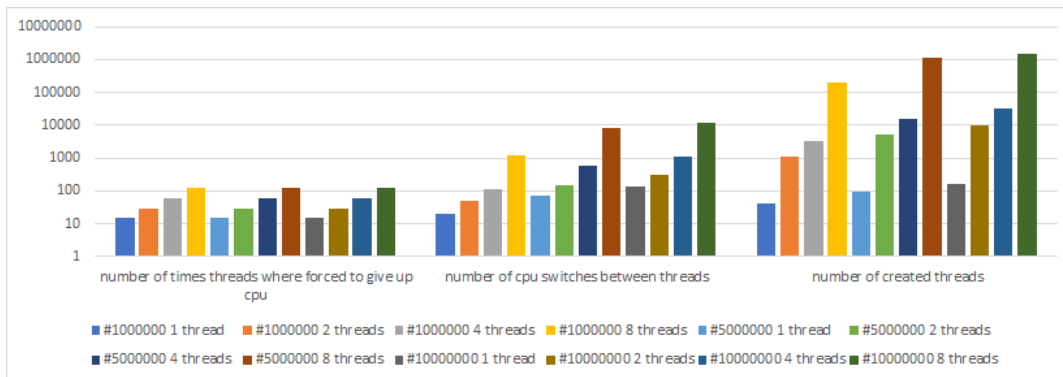


Figura 4.22: *threads* criadas pelo programa, *CPU switches* e *threads* serem obrigadas a abdicar de *CPU* valor absoluto

Mais uma vez os números de comutações de contexto e criações de *threads* aumentam com o número de *threads* criadas.

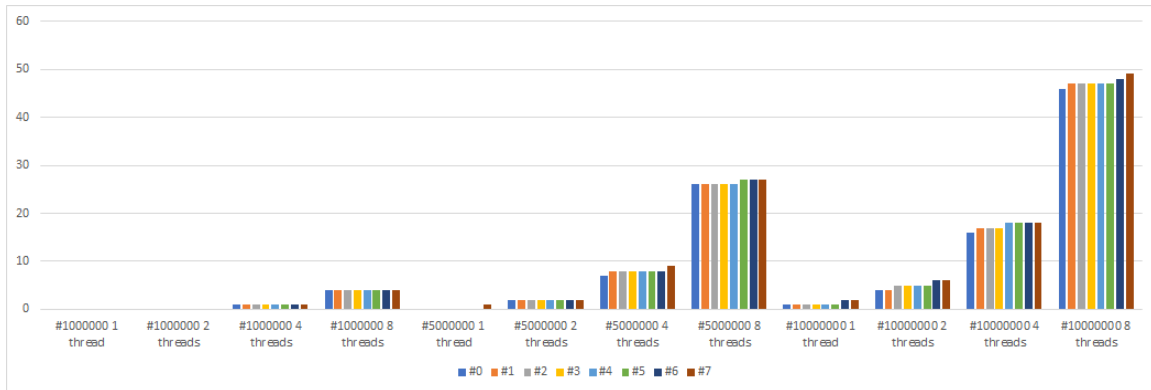


Figura 4.23: Tempo total por *thread* em segundos

A partir deste gráfico pode verificar que a repartição do trabalho é feita de maneira equilibrada entre os *cores* lógicos do *CPU*, mesmo em casos em que o programa utiliza um número máximo de *threads* inferior aos *cores* lógicos do *CPU*. Estes resultados explicam também alguns dos problemas relacionados com o mau uso da memória quando são utilizadas mais do que 1 *thread*.

Versão C++11

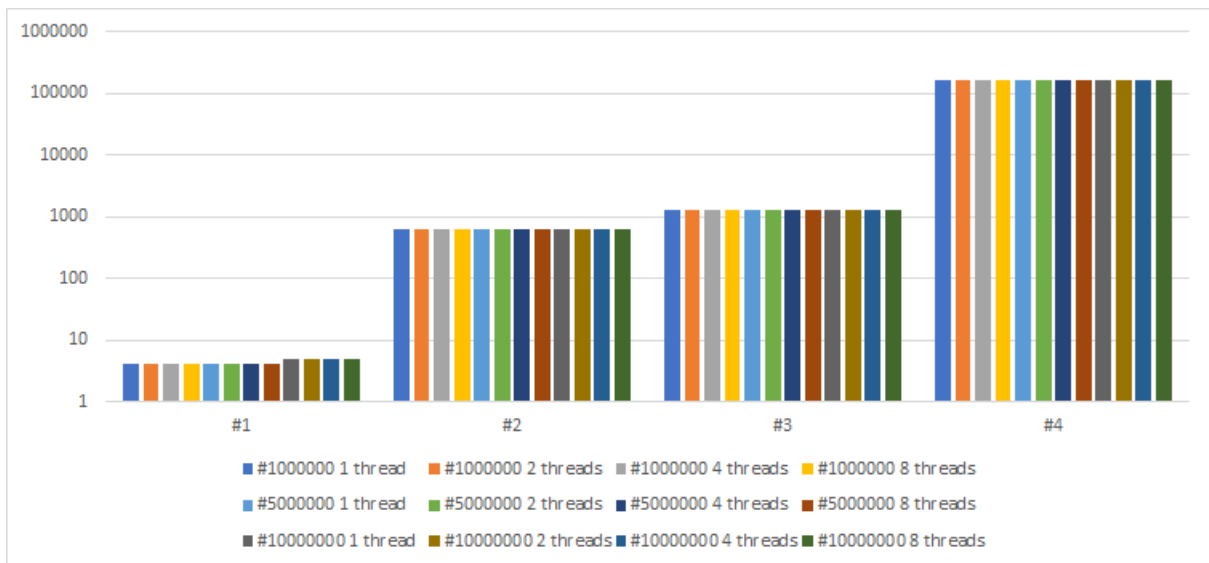


Figura 4.24: número de chamadas as funções *radix sort* por dígito

Como seria esperado pude verificar que o número de chamadas não varia consoante o tamanho ou o número de *threads*.

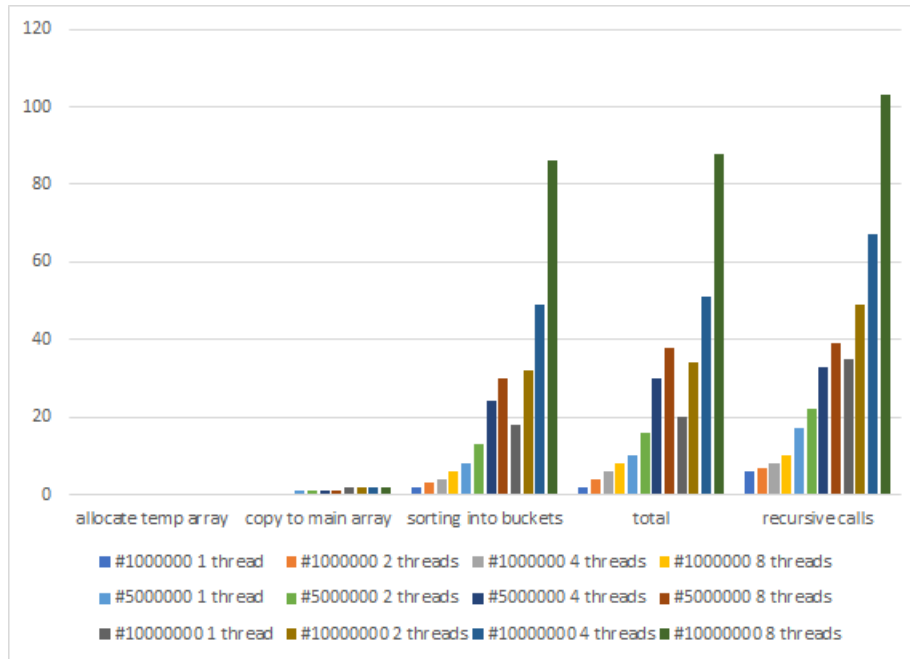


Figura 4.25: tempo gasto por secção em segundos

Tal como na implementação *Pthreads* foi possível verificar na execução desta aplicação que para os vários tamanhos e nas varias secções os tempos de execução aumentam com o número de *threads* utilizado.

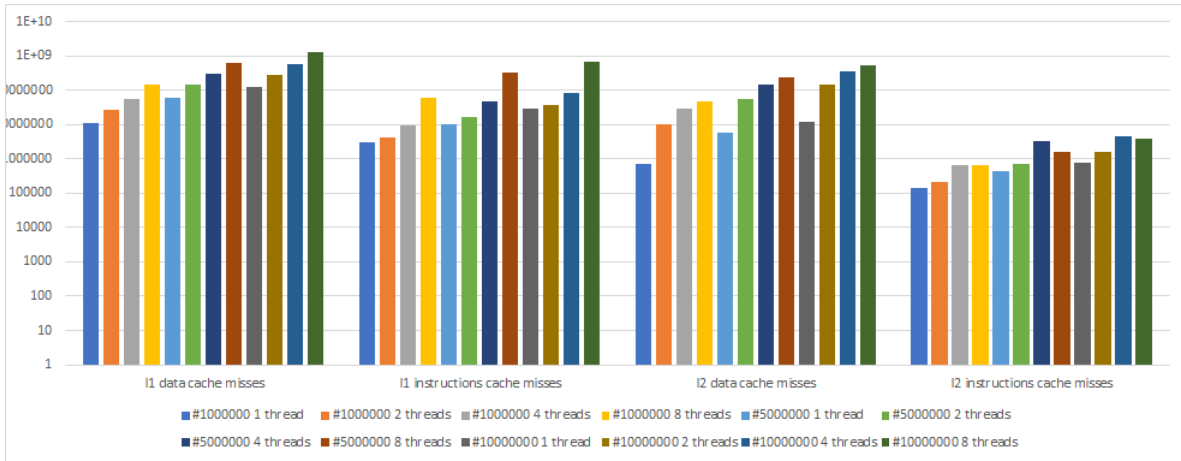


Figura 4.26: número de cache misses para *l1* e *l2* valor absoluto escala logarítmica base 10

Os resultados foram similares aos das versões anteriores, maior número de *misses* para *l1* do que *l2* e para dados relativamente a instruções.

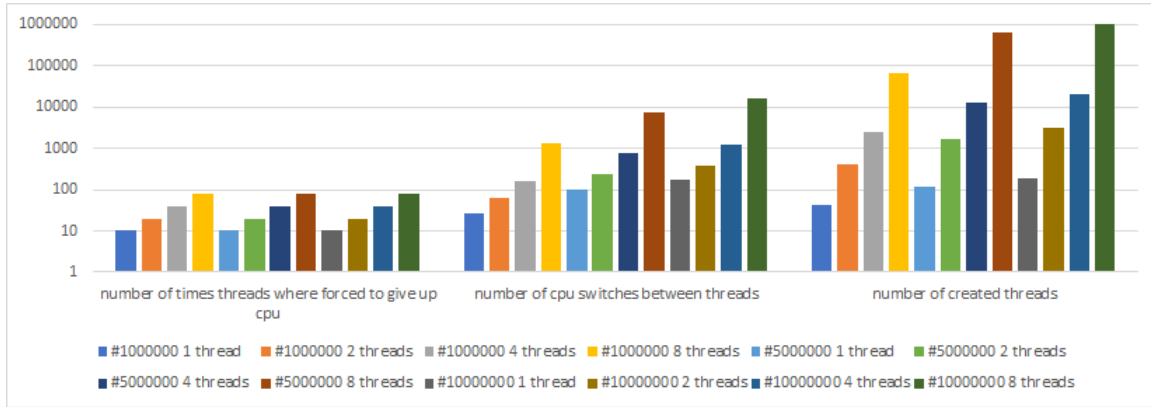


Figura 4.27: threads criadas pelo programa, *CPU switches* e *threads* serem obrigadas a abdicar de *CPU* valor absoluto

Como em versões anteriores as comutações de contexto e número de *threads* criadas aumentam com o número de *threads* utilizado pelo programa.

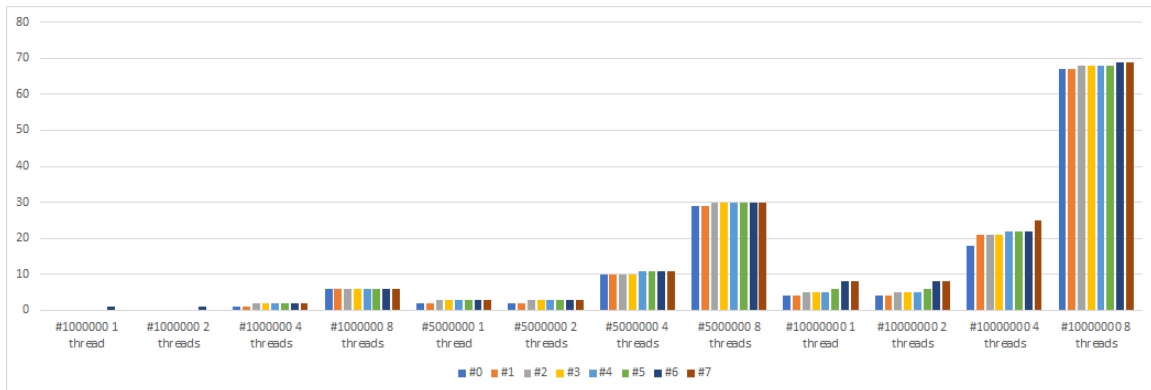


Figura 4.28: Tempo total por *thread* em segundos

Os resultados foram similares aos da versão *Pthreads* em que o trabalho está repartido de uma forma equilibrada pelos *cores* lógicos do *CPU*.

4.5 Conclusões e trabalho futuro

O trabalho realizado para a concretização das tarefas descritas neste relatório permitiu obter algum conhecimento sobre o funcionamento e uso da ferramenta *Dtrace* bem como outros conhecimentos relacionados com sistemas operativos, *hardware* e na sua influência na execução de programas.

Em termos de trabalho futuro este poderá consistir numa exploração mais profunda das funcionalidades da ferramenta *Dtrace* e o uso desta e outras ferramentas de traçado e monitorização para a monitorização e análise de outros programas e sistemas.

Capítulo 5

Perf - Utilização da ferramenta *Perf* para a análise de aplicações

5.1 Introdução

Este documento relata o trabalho realizado na análise de vários algoritmos utilizando a ferramenta *Perf*, a análise dos resultados obtidos e também a visualização dos mesmos com recurso a *FlameGraphs*.

No primeiro capítulo é detalhado o trabalho realizado para a análise de vários algoritmos de ordenação de modo a obter possíveis explicações no desempenho dos mesmos, obter e analisar os perfis de execução dos vários algoritmos, visualizar os seus comportamentos com o recurso a *FlameGraphs* e analisar o perfil de alguns dos algoritmos a nível *assembly*.

O segundo capítulo relata o trabalho realizado para a análise de duas versões do algoritmo de multiplicação de matrizes, este consiste inicialmente na análise e realização de *benchmarks* da versão *naive* do algoritmo para diferentes tamanhos e posteriormente na comparação entre os dois algoritmos e visualização do comportamento dos algoritmos com o auxílio dos *FlameGraphs*.

Posteriormente são apresentadas algumas conclusões relacionadas com os resultados obtidos, a aprendizagem realizada e trabalho futuro.

5.2 Análise de diferentes algoritmos de ordenação

5.2.1 Hardware de teste

L1 data cache	32 <i>KiB</i>
L1 instructions cache	32 <i>KiB</i>
L2 data cache	256 <i>KiB</i>
L3 data cache	12228 <i>KiB</i>
Sockets	2
CPU cores per socket	6
SMT	<i>yes</i>

Tabela 5.1: Especificações de *hardware* da maquina r431 utilizada para os testes realizados

Para os testes realizados nesta secção foram utilizadas maquinas do *rack r431* do *cluster Se-ARCH*.O uso destas maquinas para a resolução deste trabalho deu-se pela sua conveniência por possuir previamente uma instalação do *Perf*,pela sua disponibilidade e por possuir vários contadores que permitam a avaliação dos vários algoritmos.

5.2.2 Uso de *Perf* para explicar as diferenças no desempenho dos diferentes algoritmos

Metric/Algorithm	1	2	3	4
instructions	79,142,070,310	83,795,234,588	136,598,913,137	177,008,053,513
cache-misses	15,953,867	31,073,839	1,049,451,527	45,791,788
branch-misses	1,267,781,147	446,627	1,441,193,590	1,459,663,522
cpu-migration	2	0	0	6
branches	13,100,153,040	7,018,274,144	14,995,852,221	24,794,572,013
time	27.156050625	23.550020823	57.118593935	47.881616187

Tabela 5.2: Especificações de *hardware* da maquina r432 utilizada para os testes realizados

A partir da tabela acima podemos verificar que o algoritmo 2 tem os tempos de execução mais baixos seguido do algoritmo 1 ambos com tempos de execução entre 20 e 30 segundos e posteriormente os 3 e 4 com tempos significativamente piores do que os anteriores sendo estes entre 2 e 3 vezes piores que os dois melhores.

Estes tempos justificam-se parcialmente pelo número de instruções sendo que os algoritmos 1 e 2 apresentam consideravelmente menos instruções e tempos de execução comparativamente aos 3 e 4. No entanto o algoritmo 2 apresenta tempos de execução inferiores apesar de o seu numero de instruções ser superior ao algoritmo 1. O numero de cache misses similarmemente justifica algumas das diferenças de performance nomeadamente justifica a pior performance da versão 3 relativamente a 4 dada apesar da primeira possuir menos instruções.

Algo que justifica a melhor performance do algoritmo 2 relativamente ao 1 é o numero de *branches* e *branch-misses* sendo que ambos são inferiores para o algoritmo 2 especialmente o numero de *branch-misses*. A existência de um menor numero de *branches* e *branch-misses* permite que o processador faça um menor uso da *cache*, como os dados carregados são os utilizados em vez de serem descartados. Algo similar acontece com instruções em que as instruções necessárias são carregadas mais eficientemente.

5.2.3 Analise dos perfis de execução dos diferentes algoritmos

```
> perf record -F 99 ./sort algorithm 1 100000000
> perf report -n --stdio
```

```
[pg41073@compute-432-2 assignment4]$ perf report -n --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 2K of event 'cycles'
# Event count (approx.): 8044771144
#
# Overhead      Samples  Command      Shared Object      Symbol
# .....
#
# 93.68%        2559    sort sort      [.] sort1(int*, int, int)
# 1.97%         90     sort sort      [.] ini_vector(int**, int)
# 1.16%         58     sort sort      [.] copy_vector(int*, int*, int)
# 0.88%         35     sort libc-2.12.so [.] __random_r
# 0.80%         40     sort libc-2.12.so [.] __random
# 0.36%         18     sort libc-2.12.so [.] rand
# 0.31%          8     sort [kernel.kallsyms] [k] hrtimer_interrupt
# 0.28%         14     sort [kernel.kallsyms] [k] clear_page_c
# 0.15%          1     sort [kernel.kallsyms] [k] _d_lookup
# 0.12%          1     sort [kernel.kallsyms] [k] page_fault
# 0.10%          5     sort sort      [.] rand@plt
# 0.04%          1     sort [kernel.kallsyms] [k] rcu_process_gp_end
# 0.04%          1     sort [kernel.kallsyms] [k] physflat_send_IPI_mask
# 0.04%          1     sort [kernel.kallsyms] [k] _spin_lock_irqsave
# 0.02%          1     sort [kernel.kallsyms] [k] idle_cpu
# 0.02%          1     sort [kernel.kallsyms] [k] __mem_cgroup_commit_charge
# 0.02%          1     sort [kernel.kallsyms] [k] ____pagevec_lru_add
# 0.02%          1     sort [kernel.kallsyms] [k] __rcu_process_callbacks
# 0.00%         20     sort [kernel.kallsyms] [k] native_write_msr_safe
```

Figura 5.1: resultados para um array de inteiros de tamanho 100,000,000 para o algoritmo 1

```
[pg41073@compute-432-2 assignment4]$ perf report -n --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 2K of event 'cycles'
# Event count (approx.): 72382577977
#
# Overhead      Samples  Command      Shared Object      Symbol
# .....
#
# 92.63%        2396    sort sort      [.] sort2(int*, int)
# 2.17%         96     sort sort      [.] ini_vector(int**, int)
# 1.32%         51     sort sort      [.] copy_vector(int*, int*, int)
# 1.18%         44     sort libc-2.12.so [.] __random
# 0.63%          9     sort sort      [.] rand@plt
# 0.54%         24     sort libc-2.12.so [.] __random_r
# 0.44%         17     sort [kernel.kallsyms] [k] clear_page_c
# 0.27%          7     sort [kernel.kallsyms] [k] hrtimer_interrupt
# 0.20%          1     sort [kernel.kallsyms] [k] page_fault
# 0.18%          1     sort [kernel.kallsyms] [k] audit_putname
# 0.16%          7     sort libc-2.12.so [.] rand
# 0.09%          2     sort [kernel.kallsyms] [k] idle_cpu
# 0.06%          2     sort [kernel.kallsyms] [k] irq_exit
# 0.04%          1     sort [kernel.kallsyms] [k] rcu_process_dyntick
# 0.04%          1     sort [kernel.kallsyms] [k] tick_program_event
# 0.03%          1     sort [kernel.kallsyms] [k] get_page_from_freelist
# 0.02%          1     sort [kernel.kallsyms] [k] ktime_get
# 0.00%         20     sort [kernel.kallsyms] [k] native_write_msr_safe
```

Figura 5.2: Resultados para um *array* de inteiros de tamanho 100,000,000 para o algoritmo 2

```
[pg41073@compute-432-2 assignment4]$ perf report -n --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 5K of event 'cycles'
# Event count (approx.): 174903982073
#
# Overhead      Samples  Command      Shared Object      Symbol
# .....
#
# 96.74%        5483    sort sort      [.] sort3(int*, int)
# 0.94%         102    sort sort      [.] ini_vector(int**, int)
# 0.52%          37    sort sort      [.] copy_vector(int*, int*, int)
# 0.47%          51    sort libc-2.12.so [.] __random
# 0.35%          21    sort [kernel.kallsyms] [k] hrtimer_interrupt
# 0.26%          28    sort libc-2.12.so [.] __random_r
# 0.19%          16    sort [kernel.kallsyms] [k] clear_page_c
# 0.13%          14    sort libc-2.12.so [.] rand
# 0.09%           1    sort [kernel.kallsyms] [k] security_socket_sendmsg
# 0.09%           2    sort [kernel.kallsyms] [k] _spin_lock
# 0.05%           5    sort sort      [.] rand@plt
# 0.04%           2    sort [kernel.kallsyms] [k] __percpu_counter_add
# 0.04%           2    sort [kernel.kallsyms] [k] account_user_time
# 0.02%           1    sort [kernel.kallsyms] [k] run_timer_softirq
# 0.02%           1    sort [kernel.kallsyms] [k] rcu_process_gp_end
# 0.02%           1    sort [kernel.kallsyms] [k] scheduler_tick
# 0.02%           1    sort [kernel.kallsyms] [k] update_cfs_shares
# 0.02%           1    sort [kernel.kallsyms] [k] native_read_tsc
# 0.02%           1    sort [kernel.kallsyms] [k] irq_exit
# 0.00%           8    sort [kernel.kallsyms] [k] native_write_msr_safe
```

Figura 5.3: Resultados para um *array* de inteiros de tamanho 100,000,000 para o algoritmo 3

```
[pg41073@compute-432-2 assignment4]$ perf report -n --stdio
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 4K of event 'cycles'
# Event count (approx.): 136561446239
#
# Overhead      Samples  Command      Shared Object      Symbol
# .....
#
# 86.69%        3864    sort sort      [.] aux_sort4(int*, int, int, int)
# 2.89%         130    sort sort      [.] sort4(int*, int, int)
# 2.00%          89    sort libc-2.12.so [.] _int_malloc
# 1.90%          85    sort libc-2.12.so [.] _int_free
# 1.21%          99    sort sort      [.] ini_vector(int**, int)
# 1.00%          45    sort libc-2.12.so [.] malloc
# 0.67%          49    sort sort      [.] copy_vector(int*, int*, int)
# 0.63%          31    sort [kernel.kallsyms] [k] clear_page_c
# 0.49%          40    sort libc-2.12.so [.] __random
# 0.43%          35    sort libc-2.12.so [.] __random_r
# 0.40%          18    sort libc-2.12.so [.] free
# 0.34%          16    sort [kernel.kallsyms] [k] hrtimer_interrupt
# 0.34%          15    sort libc-2.12.so [.] malloc_consolidate
# 0.19%          15    sort libc-2.12.so [.] rand
# 0.13%           1    sort [kernel.kallsyms] [k] local_bh_enable_ip
# 0.11%           5    sort sort      [.] free@plt
# 0.09%           1    sort [kernel.kallsyms] [k] __wake_up_bit
# 0.09%           4    sort sort      [.] malloc@plt
# 0.06%           5    sort sort      [.] rand@plt
```

Figura 5.4: Resultados para um *array* de inteiros de tamanho 100,000,000 para o algoritmo 4

A partir dos testes realizados pode verificar que a maioria do tempo gasto de execução dos vários algoritmos é gasto em funções de utilizador mais precisamente nas várias funções de ordenação e auxiliares.

Mais precisamente pode verificar que a percentagem do tempo gasto nas funções de utilizador é superior para os 3 primeiro algoritmo, especialmente para o segundo algoritmo e consideravelmente inferior para o quarto algoritmo, sendo que neste os testes evidenciam que um tempo considerável da execução do mesmo é gasto em funções de alocação e libertação estáticas de memória.

Denotasse também que a percentagem do tempo despendido nas funções de inicialização é superior para o primeiro e segundo algoritmo. Isto deve-se por estes possuírem tempos de execução consideravelmente inferiores como visto anteriormente.

5.2.4 Uso de *Flamegraphs* para analisar as chamadas dos algoritmos

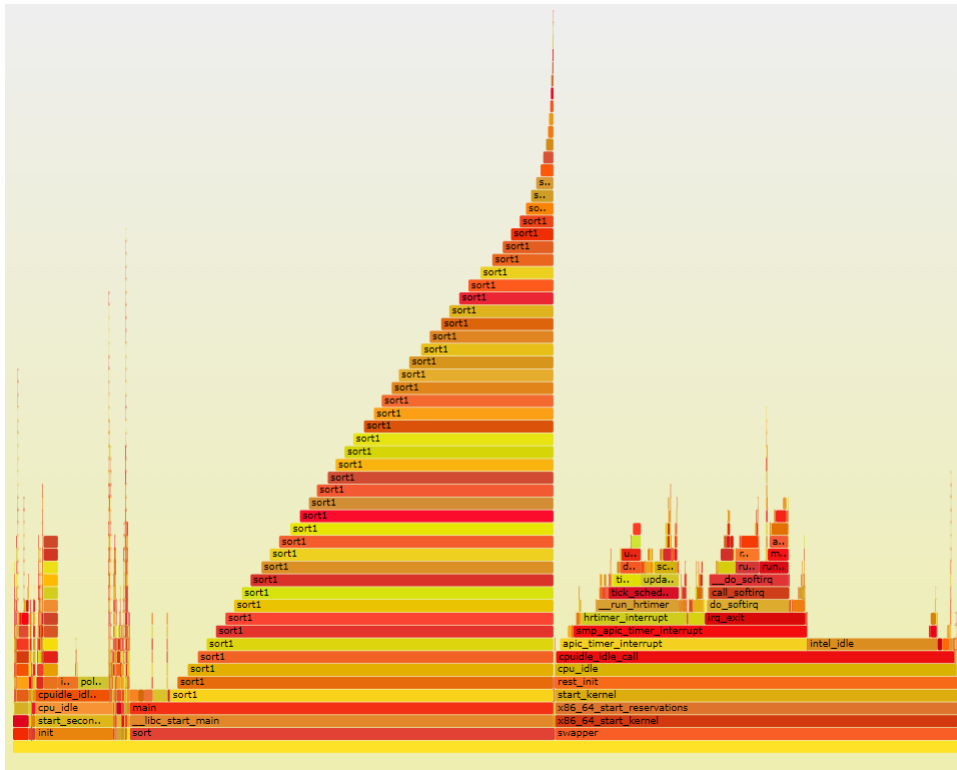


Figura 5.5: *Flamegraph* para um *array* de inteiros de tamanho 100,000,000 para o algoritmo 1

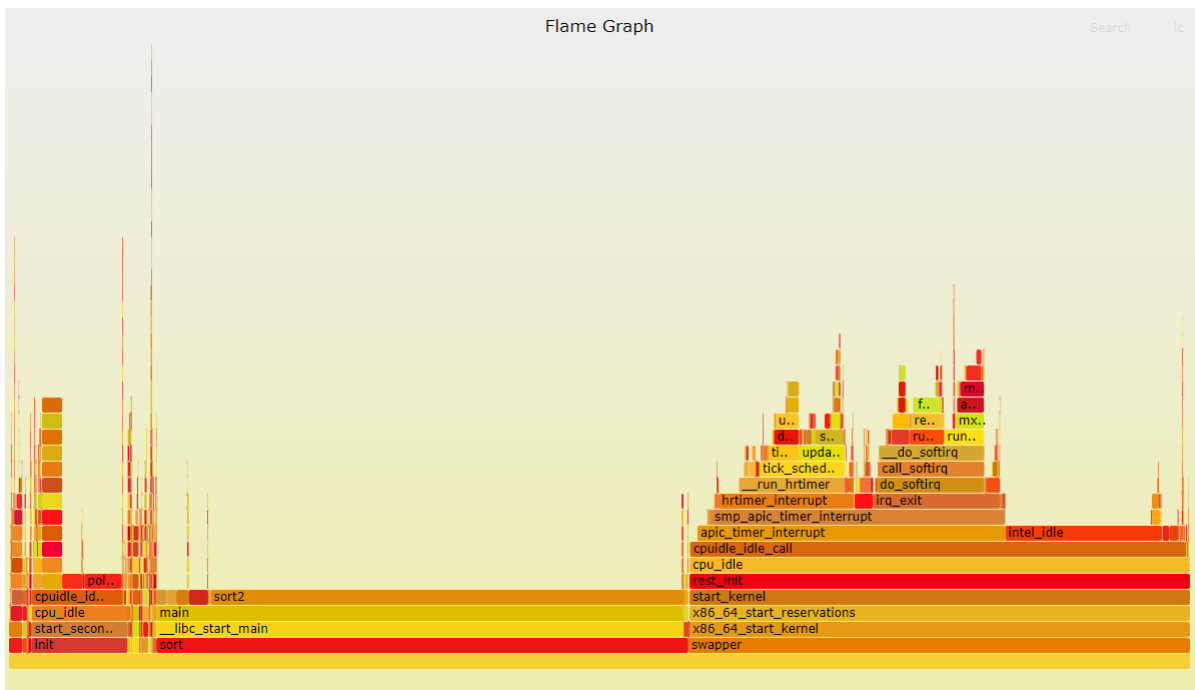


Figura 5.6: *Flamegraph* para um *array* de inteiros de tamanho 100,000,000 para o algoritmo 2

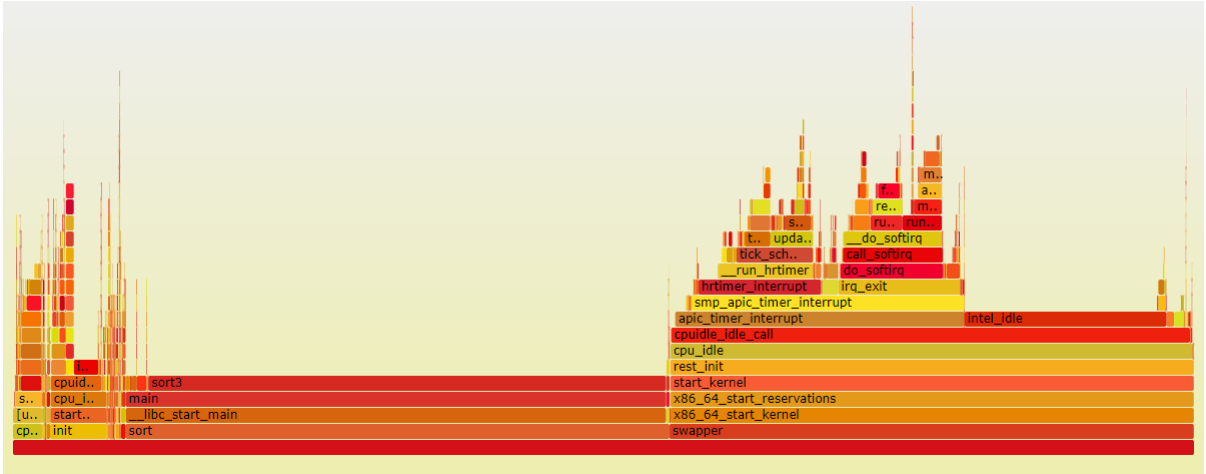


Figura 5.7: *Flamegraph* para um *array* de inteiros de tamanho 100,000,000 para o algoritmo 3

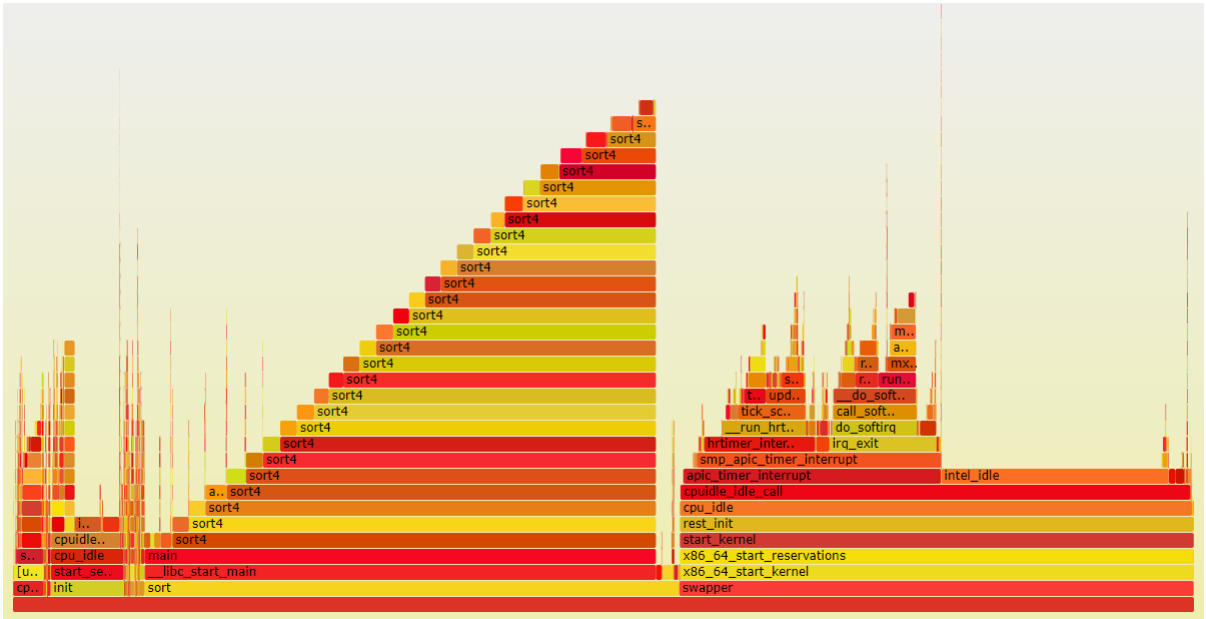


Figura 5.8: *Flamegraph* para um *array* de inteiros de tamanho 100,000,000 para o algoritmo 4

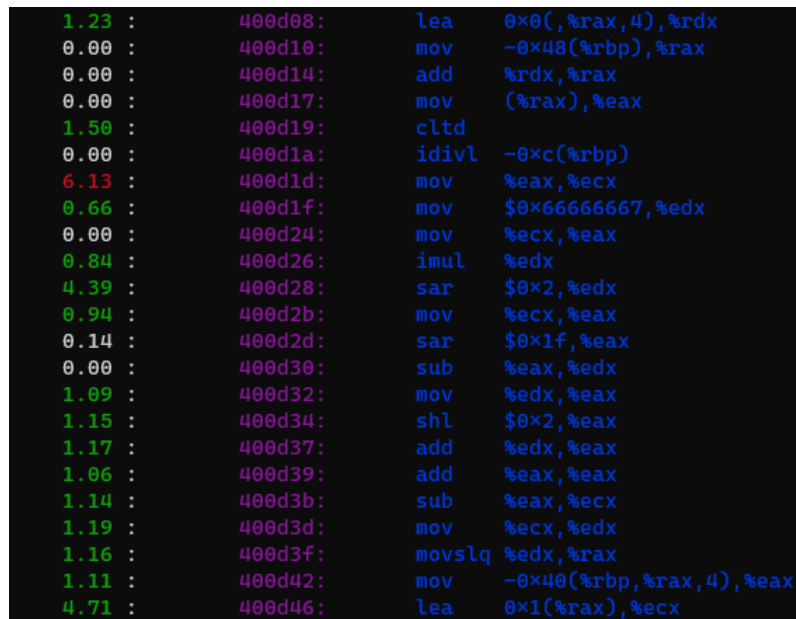
Com base no aninhamento das chamadas pode verificar que os algoritmos 1 e 4 tem um comportamento similar em termos de chamadas recursivas. Podemos verificar também a partir dos gráficos que no algoritmo 4 as funções de ordenação fazem chamadas a uma função auxiliar que tem um tempo de execução relativamente constante relativamente a chamada recursiva da mesma função a medida que são realizadas as chamadas. Verifica-se também que são realizadas significativamente menos chamadas para o algoritmo 4 do que para o 1. Visto que no algoritmo de *merge sort* em todos os níveis das chamadas recursivas é realizado o *merge* de todos os elementos e este poderá ter tempos de execução semelhantes para cada um dos níveis podemos assumir que a função auxiliar ao algoritmo 4 realiza esta tarefa e o gráfico do algoritmo gráfico 4 demonstra que durante as chamadas as função auxiliar do algoritmo 4 são invocadas varias funções relacionadas com alocações de memória, por estas razões podemos assumir que o algoritmo 4 é o algoritmo *merge sort*. Como o algoritmo possui um comportamento similar em termos de chamadas recursivas possuindo mais níveis do que o anterior, dá-se a possibilidade de este ser o algoritmo de *quick sort*, o que justifica a existência de um maior numero de níveis das chamadas

recursivas por este algoritmo não ser estável.

A partir dos gráficos é similarmente verificável que ambos os algoritmos 2 e 3 não realizam chamadas recursivas. Sendo que os comportamentos destas funções são similares a nível de chamadas pode-se verificar a partir das medições realizadas anteriormente em que realizamos que existe um numero de cache-misses bastante inferior para o algoritmo 2 e que este possui tempos de execução também significativamente inferiores aos vários outros assumir que este algoritmo seja o *radix sort* e por possuir um numero elevado quando comparado de *cache-misses* e tempos comparativamente aos outros algoritmos podemos assumir que o algoritmo 3 é o *heap sort*, visto que este algoritmo apresenta fraca localidade temporal e espacial nos acessos aos dados.

5.2.5 Analise do segundo algoritmo a nível *assembly*

```
> perf record -g ./bin/sort 2 1 100000000  
> perf annotate --stdio
```



The image shows a terminal window displaying assembly code for the second part of algorithm 2. Each line of assembly is preceded by two performance metrics: a green number representing time and a red number representing cache misses. The assembly instructions are color-coded: green for data movement, red for arithmetic/logic, and blue for control flow. The code includes instructions like `lea`, `mov`, `add`, `mov`, `cltd`, `idivl`, `mov`, `mov`, `imul`, `sar`, `mov`, `sar`, `sub`, `mov`, `shl`, `add`, `add`, `sub`, `mov`, `movslq`, `mov`, and `lea`.

Time	Cache Misses	Address	Instruction
1.23		400d08:	lea 0x0(%rax,4),%rdx
0.00		400d10:	mov -0x48(%rbp),%rax
0.00		400d14:	add %rdx,%rax
0.00		400d17:	mov (%rax),%eax
1.50		400d19:	cltd
0.00		400d1a:	idivl -0xc(%rbp)
6.13		400d1d:	mov %eax,%ecx
0.66		400d1f:	mov \$0x66666667,%edx
0.00		400d24:	mov %ecx,%eax
0.84		400d26:	imul %edx
4.39		400d28:	sar \$0x2,%edx
0.94		400d2b:	mov %ecx,%eax
0.14		400d2d:	sar \$0x1f,%eax
0.00		400d30:	sub %eax,%edx
1.09		400d32:	mov %edx,%eax
1.15		400d34:	shl \$0x2,%eax
1.17		400d37:	add %edx,%eax
1.06		400d39:	add %eax,%eax
1.14		400d3b:	sub %eax,%ecx
1.19		400d3d:	mov %ecx,%edx
1.16		400d3f:	movslq %edx,%rax
1.11		400d42:	mov -0x40(%rbp,%rax,4),%eax
4.71		400d46:	lea 0x1(%rax),%ecx

Figura 5.9: *assembly* anotado para o algoritmo 2 parte 1

```

1.17 :      400da1:      lea     0x0(,%rax,4),%rdx
0.02 :      400da9:      mov     -0x48(%rbp),%rax
0.07 :      400dad:      add     %rdx,%rax
0.02 :      400db0:      mov     (%rax),%eax
1.67 :      400db2:      cld
0.05 :      400db3:      idivl    -0xc(%rbp)
6.88 :      400db6:      mov     %eax,%ecx
0.66 :      400db8:      mov     $0x66666667,%edx
0.00 :      400dbd:      mov     %ecx,%eax
0.70 :      400dbf:      imul    %edx
4.43 :      400dc1:      sar     $0x2,%edx
1.14 :      400dc4:      mov     %ecx,%eax
0.03 :      400dc6:      sar     $0x1f,%eax
0.00 :      400dc9:      sub     %eax,%edx
1.15 :      400dcb:      mov     %edx,%eax
1.14 :      400dcd:      shl     $0x2,%eax
1.16 :      400dd0:      add     %edx,%eax
1.22 :      400dd2:      add     %eax,%eax

```

Figura 5.10: *assembly* anotado para o algoritmo 2 parte 2

```

7.02 :      400e19:      subl    $0x1,-0x4(%rbp)
0.05 :      400e1d:      jmpq     400d92 <sort2(int*, int)+0x153>
0.00 :      400e22:      movl     $0x0,-0x4(%rbp)
1.24 :      400e29:      mov     -0x4(%rbp),%eax
0.06 :      400e2c:      cmp     -0x4c(%rbp),%eax
0.04 :      400e2f:      jge     400e63 <sort2(int*, int)+0x224>
0.03 :      400e31:      mov     -0x4(%rbp),%eax
0.00 :      400e34:      cltq
1.24 :      400e36:      lea     0x0(,%rax,4),%rdx
0.03 :      400e3e:      mov     -0x48(%rbp),%rax
0.08 :      400e42:      add     %rax,%rdx
0.04 :      400e45:      mov     -0x4(%rbp),%eax
1.19 :      400e48:      cltq
0.01 :      400e4a:      lea     0x0(,%rax,4),%rcx
0.02 :      400e52:      mov     -0x18(%rbp),%rax
0.03 :      400e56:      add     %rcx,%rax
1.17 :      400e59:      mov     (%rax),%eax
6.83 :      400e5b:      mov     %eax,(%rdx)
1.22 :      400e5d:      addl    $0x1,-0x4(%rbp)

```

Figura 5.11: *assembly* anotado para o algoritmo 2 parte 3

A partir dos resultados deste comando pude verificar que grande parte do tempo de execução é gasto no carregamento e escrita de dados e no cálculo das posições do *array* onde inserir e retirar dados. Tendo isto em conta uma maneira de melhorar a performance deste algoritmo passaria por aumentar a localidade dos dados por exemplo utilizar a variante *MSD* em vez de *LSD* como está a utilizar o algoritmo, visto que esta possui melhor localidade temporal. Algo que também poderia melhorar a performance deste algoritmo passaria por realizar um alinhamento do *array* dos dígitos e dos *arrays* temporários de modo a diminuir *cache misses* e fazer uso da vectorização.

5.3 Análise de algoritmos de multiplicação de matrizes com uso do *Perf*

5.3.1 Algoritmos de multiplicação de matrizes utilizados

Para o trabalho realizado nesta secção foi utilizado uma implementação do algoritmo *naive* de multiplicação de matrizes encontrado em [2]. Para alguns dos testes realizados foi também implementado um algoritmo de multiplicação de matrizes alternativo, este algoritmo modifica o ordem dos ciclos k e j , o que garante uma maior localidade temporal no acesso aos dados o que permite obter melhor uso de cache e da memória e das extensões vetoriais.

Para facilitar a análise e os testes realizados o programa foi modificado para aceitar o tamanho das matrizes e o algoritmo a utilizar como argumento. Foi também utilizada a diretiva `#pragma GCC ivdep`, como as matrizes são passadas como argumento e para garantir que os resultados são mais semelhantes aos da versão anterior em que as matrizes são alocadas dinamicamente e consistem em variáveis globais, o uso desta diretiva permite que o compilador assuma que não existem dependências entre as matrizes.

Algorithm 2 Naive matrix multiplication algorithm

```

1: for ( $i \leftarrow 0; i < M_{size}; i++$ ) do
2:   for ( $j \leftarrow 0; j < M_{size}; j++$ ) do
3:      $sum \leftarrow 0$ 
4:     for ( $k \leftarrow 0; k < M_{size}; k++$ ) do
5:        $sum += M_a[i][k] * M_b[k][j]$ 
6:     end for
7:      $M_r[i][j] \leftarrow sum$ 
8:   end for
9: end for

```

Algorithm 3 Loop nest interchange matrix multiplication algorithm

```

1: for ( $i \leftarrow 0; i < M_{size}; i++$ ) do
2:   for ( $k \leftarrow 0; k < M_{size}; k++$ ) do
3:     for ( $j \leftarrow 0; j < M_{size}; j++$ ) do
4:        $M_r[i][j] += M_a[i][k] * M_b[k][j]$ 
5:     end for
6:   end for
7: end for

```

5.3.2 Hardware de teste

L1 data cache	32 KiB
L1 instructions cache	32 KiB
L2 data cache	256 KiB
L3 data cache	20480 KiB
Sockets	2
CPU cores per socket	8
SMT	yes

Tabela 5.3: Especificações de *hardware* da máquina r431 utilizada para os testes realizados

Para efeitos de teste foram utilizadas como para o exercício anterior máquinas do *rack r431* do *cluster SeARCH* estas máquinas foram utilizadas por terem o *software* necessário a resolução destes exercícios previamente instalados e por possuírem contadores de *hardware* e eventos similares aos do *Hardware* similar aos das máquinas utilizadas nos tutoriais utilizados como base para este exercício.

5.3.3 Tamanhos do problema

Os testes foram realizados com 4 tamanhos distintos estes foram escolhidas de modo as estruturas de dados utilizadas pelo programa caberem em *cache* nível 1, *cache* nível 2, *cache* nível 3 e em memória *RAM*.

fits in L1	32
fits in L2	128
fits in L3	512
fits in RAM	2048

Tabela 5.4: Especificações de *hardware* da máquina r432 utilizada para os testes realizados

5.3.4 Estabelecer uma *beline*

Para estabelecer a *baseline* foram medidos os eventos *cpu – clocks* e *faults* decorridos durante a execução de uma multiplicação de matrizes utilizando o algoritmo *naive*, para os diferentes tamanhos utilizando o primeiro algoritmo. Para tal foram utilizados os comandos seguintes.

```
> perf stat -e cpu-clock ./bin/naive 1 size
> perf stat -e cpu-clock , faults ./bin/naive 1 size
```

size	cpu-clocks events in milliseconds	faults	time in seconds
32	2.322247	308	0.003654498
128	7.190261	355	0.008417665
512	314.661839	1081	0.317850716
2048	47168.633150	12650	47.341694693

Tabela 5.5: medições do evento *cpu – clock* e tempos de execução do primeiro algoritmo para diferentes tamanhos

O cálculo da *baseline* permite avaliar que o número de eventos de *cpu – clocks* é aproximado do número real sendo que os valores coincidem com os que seriam de esperar de acordo com o tempo de execução registado. Apesar disso encontram-se algumas discrepâncias para os dois tamanhos inferiores.

5.3.5 Encontro de pontos quentes

A fim de analisar possíveis *bottlenecks* e possibilidades de otimização no programa foi utilizado o comando abaixo para poder visualizar as funções que mais registaram os eventos *cpu – clock* e *faults* para a versão *naive* para os diversos tamanhos.

```
> perf record -e cpu-clock , faults ./bin/naive 1 size
> perf report --stdio --sort comm,dso
```

```
[pg41073@compute-432-2 assignment4]$ perf report --stdio --sort comm,dso
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 167K of event 'cpu-clock'
# Event count (approx.): 167323
#
# Overhead Command Shared Object
# .....
#
# 99.40% naive naive
# 0.37% naive [kernel.kallsyms]
# 0.23% naive libc-2.12.so
# 0.00% naive ld-2.12.so
# 0.00% naive [nfs]
#
# Samples: 356 of event 'faults'
# Event count (approx.): 12772
#
# Overhead Command Shared Object
# .....
#
# 48.90% naive libc-2.12.so
# 48.22% naive naive
# 2.86% naive ld-2.12.so
# 0.02% naive [kernel.kallsyms]
```

Figura 5.12: Output do comando para o algoritmo *naive* com matrizes de tamanho 2048x2048.

```
[pg41073@compute-431-5 assignment4]$ perf report --stdio --sort comm,dso
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 1K of event 'cpu-clock'
# Event count (approx.): 1277
#
# Overhead Command Shared Object
# .....
#
# 96.48% naive naive
# 2.19% naive libc-2.12.so
# 0.78% naive [kernel.kallsyms]
# 0.55% naive ld-2.12.so
#
# Samples: 11 of event 'faults'
# Event count (approx.): 1100
#
# Overhead Command Shared Object
# .....
#
# 70.73% naive libc-2.12.so
# 29.00% naive ld-2.12.so
# 0.27% naive [kernel.kallsyms]
```

Figura 5.13: Output do comando para o algoritmo *naive* com matrizes de tamanho 512x512.

```
[pg41073@compute-432-2 assignment4]$ perf report --stdio --sort comm,dso
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 31 of event 'cpu-clock'
# Event count (approx.): 31
#
# Overhead Command Shared Object
# .....
#
# 61.29% naive naive
# 16.13% naive ld-2.12.so
# 12.90% naive [kernel.kallsyms]
# 6.45% naive libc-2.12.so
# 3.23% naive [nfs]
#
# Samples: 15 of event 'faults'
# Event count (approx.): 396
#
# Overhead Command Shared Object
# .....
#
# 76.77% naive ld-2.12.so
# 22.73% naive libstdc++.so.6.0.21
# 0.51% naive [kernel.kallsyms]
```

Figura 5.14: Output do comando para o algoritmo *naive* com matrizes de tamanho 128x128.

```
[pg41073@compute-432-2 assignment4]$ perf report --stdio --sort comm,dso
# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 10 of event 'cpu-clock'
# Event count (approx.): 10
#
# Overhead Command Shared Object
# .....
#
# 80.00% naive ld-2.12.so
# 10.00% naive [kernel.kallsyms]
# 10.00% naive naive
#
# Samples: 15 of event 'faults'
# Event count (approx.): 390
#
# Overhead Command Shared Object
# .....
#
# 76.92% naive ld-2.12.so
# 22.56% naive libc-2.12.so
# 0.51% naive [kernel.kallsyms]
```

Figura 5.15: Output do comando para o algoritmo *naive* com matrizes de tamanho 32x32.

Em termos de *cpu – clock* é possível verificar que a maioria do tempo gasto pelo programa, para os 3 tamanhos maiores, para o menor tamanho verifica-se que o tempo gasto maioritariamente no *Dynamic linker* e funções de *kernel*.

Em termos de *fault* verifica-se que estas acontecem para o maior tamanho maioritariamente em *system calls* e no programa. Para os outros tamanhos estas acontecem maioritariamente em *system calls*.

Para restringir os resultados as funções do programa e da biblioteca de *runtime* do *c* foi utilizado o comando abaixo.

```
> perf report --stdio --dsos=naive,libc-2.12.so,ld-2.12.so,libstdc++.so.6.0.21
/
```

```

#
# Samples: 162K of event 'cpu-clock'
# Event count (approx.): 162429
#
# Overhead  Command      Shared Object      Symbol
# .....
#
# 99.60%    naive    naive              [.] multiply_matrices(float**, float**, float**, int)
# 0.07%    naive    naive              [.] initialize_matrices(float**, float**, float**, int)
# 0.05%    naive    libc-2.12.so       [.] __random
# 0.05%    naive    libc-2.12.so       [.] __random_r
# 0.02%    naive    naive              [.] rand@plt
# 0.01%    naive    libc-2.12.so       [.] rand
# 0.00%    naive    libc-2.12.so       [.] _int_malloc
# 0.00%    naive    ld-2.12.so         [.] _dl_lookup_symbol_x
# 0.00%    naive    ld-2.12.so         [.] _dl_relocate_object
# 0.00%    naive    ld-2.12.so         [.] do_lookup_x
# 0.00%    naive    libc-2.12.so       [.] __brk
# 0.00%    naive    libstdc++.so.6.0.21 [.] _GLOBAL__sub_I_locale_inst.cc

# Samples: 300 of event 'faults'
# Event count (approx.): 12687
#
# Overhead  Command      Shared Object      Symbol
# .....
#
# 48.48%    naive    libc-2.12.so       [.] _int_malloc
# 48.44%    naive    naive              [.] initialize_matrices(float**, float**, float**, int)
# 0.76%    naive    ld-2.12.so         [.] match_symbol
# 0.76%    naive    ld-2.12.so         [.] strcmp
# 0.69%    naive    libstdc++.so.6.0.21 [.] call_gmon_start
# 0.31%    naive    ld-2.12.so         [.] _dl_sysdep_start
# 0.18%    naive    naive              [.] multiply_matrices(float**, float**, float**, int)

```

Figura 5.16: Output do comando para o algoritmo *naive* com matrizes de tamanho 2048x2048.

```

# Event count (approx.): 1268
#
# Overhead  Command      Shared Object      Symbol
# .....
#
# 95.43%    naive    naive              [.] multiply_matrices(float**, float**, float**, int)
# 1.18%    naive    naive              [.] initialize_matrices(float**, float**, float**, int)
# 0.63%    naive    libc-2.12.so       [.] __random_r
# 0.32%    naive    libc-2.12.so       [.] __random
# 0.32%    naive    libc-2.12.so       [.] rand
# 0.08%    naive    ld-2.12.so         [.] _dl_lookup_symbol_x
# 0.08%    naive    ld-2.12.so         [.] _dl_map_object
# 0.08%    naive    ld-2.12.so         [.] _dl_relocate_object
# 0.08%    naive    ld-2.12.so         [.] dl_main
# 0.08%    naive    ld-2.12.so         [.] do_lookup_x
# 0.08%    naive    libc-2.12.so       [.] _int_malloc
# 0.08%    naive    naive              [.] rand@plt

# Samples: 20 of event 'faults'
# Event count (approx.): 1117
#
# Overhead  Command      Shared Object      Symbol
# .....
#
# 3.67%    naive    ld-2.12.so         [.] memset
# 1.07%    naive    ld-2.12.so         [.] _dl_new_object
# 0.72%    naive    ld-2.12.so         [.] _dl_load_cache_lookup
# 0.63%    naive    ld-2.12.so         [.] _dl_sysdep_start
# 0.63%    naive    ld-2.12.so         [.] _start
# 0.54%    naive    ld-2.12.so         [.] _dl_map_object_from_fd
# 0.09%    naive    ld-2.12.so         [.] _dl_next_tls_modid

```

Figura 5.17: Output do comando para o algoritmo *naive* com matrizes de tamanho 512x512.


```

# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 22 of event 'cpu-clock'
# Event count (approx.): 22
#
# Overhead Command Shared Object Symbol
# .....
#
# 50.00% naive naive [.] multiply_matrices(float**, float**, float**, int)
# 13.64% naive ld-2.12.so [.] strcmp
# 9.09% naive libc-2.12.so [.] rand
# 4.55% naive ld-2.12.so [.] do_lookup_x
#
# Samples: 15 of event 'faults'
# Event count (approx.): 556
#
# Overhead Command Shared Object Symbol
# .....
#
# 39.39% naive ld-2.12.so [.] _dl_lookup_symbol_x
# 28.96% naive libstdc++.so.6.0.21 [.] _GLOBAL__sub_I_compatibility_thread_c__0x.cc
# 18.88% naive ld-2.12.so [.] _dl_map_object
# 4.86% naive ld-2.12.so [.] _dl_cache_libcmp
# 3.24% naive ld-2.12.so [.] dl_main
# 1.26% naive ld-2.12.so [.] _dl_start
# 1.08% naive ld-2.12.so [.] _dl_sysdep_read_whole_file
# 0.90% naive ld-2.12.so [.] _dl_map_object_from_fd
# 0.54% naive ld-2.12.so [.] _start
# 0.18% naive ld-2.12.so [.] _dl_next_tls_modid
# 0.18% naive ld-2.12.so [.] memset

```

Figura 5.18: Output do comando para o algoritmo *naive* com matrizes de tamanho 128x128.

```

# To display the perf.data header info, please use --header/--header-only options.
#
# Samples: 8 of event 'cpu-clock'
# Event count (approx.): 8
#
# Overhead Command Shared Object Symbol
# .....
#
# 37.50% naive ld-2.12.so [.] do_lookup_x
# 12.50% naive ld-2.12.so [.] _dl_lookup_symbol_x
# 12.50% naive ld-2.12.so [.] _dl_map_object_deps
# 12.50% naive ld-2.12.so [.] check_match.12442
#
# Samples: 15 of event 'faults'
# Event count (approx.): 421
#
# Overhead Command Shared Object Symbol
# .....
#
# 25.18% naive ld-2.12.so [.] match_symbol
# 25.18% naive ld-2.12.so [.] strcmp
# 25.18% naive libc-2.12.so [.] malloc
# 13.54% naive ld-2.12.so [.] dl_main
# 4.99% naive ld-2.12.so [.] _dl_cache_libcmp
# 1.43% naive ld-2.12.so [.] _dl_setup_hash
# 1.43% naive ld-2.12.so [.] _dl_sysdep_read_whole_file
# 1.19% naive ld-2.12.so [.] _dl_map_object_from_fd
# 0.71% naive ld-2.12.so [.] _dl_start
# 0.24% naive ld-2.12.so [.] _dl_next_tls_modid
# 0.24% naive ld-2.12.so [.] _start
# 0.24% naive ld-2.12.so [.] memset
:

```

Figura 5.19: Output do comando para o algoritmo *naive* com matrizes de tamanho 32x32.

Os resultados obtidos com estas opções são similares aos anteriores. Com este comando pode-se verificar que a função *multiply_matrices* é a onde é gasto a maioria do tempo para os 3 tamanhos maiores. Verifica-se também que algum tempo é gasto na função de *initialize_matrices* para os dois maiores tamanhos. Em termos de *faults* verifica-se que estes estão relacionados com alocação de memória para o tamanho maior para os restantes tamanhos varias *system calls* estão associadas aos

mesmos.

5.3.6 Análise do programa ao nível *assembly*

De modo a poder analisar as secções da função de multiplicação de matrizes *naive* que consomem mais tempo de execução foi utilizado o comando *perf annotate* para essa função.

```
> perf annotate --stdio --dsos=naive
--symbol="multiply_matrices(float**, float**, float**, int)"
```

```
      :      for (i = 0 ; i < msize ; i++) {
0.00 :      40080b:      xor     %edi,%edi
0.00 :      40080d:      mov     (%rbx,%rdi,8),%r10
0.00 :      400811:      mov     0x0(%rbp,%rdi,8),%r11
0.00 :      400816:      xor     %r9d,%r9d
0.00 :      400819:      nopl    0x0(%rax)
0.01 :      400820:      movaps  %xmm2,%xmm1
0.00 :      400823:      xor     %eax,%eax
0.00 :      400825:      nopl    (%rax)
      :      for (j = 0 ; j < msize ; j++) {
      :      float sum = 0.0 ;
      :      #pragma GCC ivdep
      :      for (k = 0 ; k < msize ; k++) {
      :      sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
5.11 :      400828:      mov     (%rsi,%rax,8),%r8
0.14 :      40082c:      movss   (%r8,%r9,1),%xmm0
68.09 :      400832:      mulss   (%r10,%rax,4),%xmm0
16.42 :      400838:      add     $0x1,%rax
      :
```

Figura 5.20: Output do comando para o algoritmo *naive* com matrizes de tamanho 2048x2048 parte 1.

```
      :      for (i = 0 ; i < msize ; i++) {
      :      for (j = 0 ; j < msize ; j++) {
      :      float sum = 0.0 ;
      :      #pragma GCC ivdep
      :      for (k = 0 ; k < msize ; k++) {
10.23 :      400842:      jg      400828 <multiply_matrices(float**, float**, float**, int)+0x38>
      :      sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
      :      }
      :      matrix_r[i][j] = sum ;
      :
```

Figura 5.21: Output do comando para o algoritmo *naive* com matrizes de tamanho 2048x2048 parte 2.

```
      :      for (j = 0 ; j < msize ; j++) {
      :      float sum = 0.0 ;
      :      #pragma GCC ivdep
      :      for (k = 0 ; k < msize ; k++) {
      :      sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
13.78 :      400828:      mov     (%rsi,%rax,8),%r8
0.33 :      40082c:      movss   (%r8,%r9,1),%xmm0
30.27 :      400832:      mulss   (%r10,%rax,4),%xmm0
32.49 :      400838:      add     $0x1,%rax
      :
```

Figura 5.22: Output do comando para o algoritmo *naive* com matrizes de tamanho 512x512 parte 1.

```

:      for (i = 0 ; i < msize ; i++) {
:          for (j = 0 ; j < msize ; j++) {
:              float sum = 0.0 ;
:              #pragma GCC ivdep
:              for (k = 0 ; k < msize ; k++) {
22.97 :          400842:    jg     400828 <multiply_matrices(float**, float**, float**, int)+0x38>
:              sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
:              }
:          matrix_r[i][j] = sum ;
:      }

```

Figura 5.23: Output do comando para o algoritmo *naive* com matrizes de tamanho 512x512 parte 2.

```

:          for (j = 0 ; j < msize ; j++) {
:              float sum = 0.0 ;
:              #pragma GCC ivdep
:              for (k = 0 ; k < msize ; k++) {
:                  sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
44.44 :          400828:    mov    (%rsi,%rax,8),%r8
0.00 :          40082c:    movss  (%r8,%r9,1),%xmm0
5.56 :          400832:    mulss  (%r10,%rax,4),%xmm0
16.67 :          400838:    add    $0x1,%rax
:      }
:  }

```

Figura 5.24: Output do comando para o algoritmo *naive* com matrizes de tamanho 128x128 parte 1.

```

:      for (i = 0 ; i < msize ; i++) {
:          for (j = 0 ; j < msize ; j++) {
:              float sum = 0.0 ;
:              #pragma GCC ivdep
:              for (k = 0 ; k < msize ; k++) {
27.78 :          400842:    jg     400828 <multiply_matrices(float**, float**, float**, int)+0x38>
:              sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
:              }
:          matrix_r[i][j] = sum ;
0.00 :          400844:    movss  %xmm1,(%r11,%r9,1)
5.56 :          40084a:    add    $0x4,%r9
:      }
:  }
void multiply_matrices(float **matrix_a, float **matrix_b, float **matrix_r, int msize)

```

Figura 5.25: Output do comando para o algoritmo *naive* com matrizes de tamanho 128x128 parte 2.

```

:          for (j = 0 ; j < msize ; j++) {
:              float sum = 0.0 ;
:              #pragma GCC ivdep
:              for (k = 0 ; k < msize ; k++) {
:                  sum = sum + (matrix_a[i][k] * matrix_b[k][j]) ;
0.00 :          400828:    mov    (%rsi,%rax,8),%r8
0.00 :          40082c:    movss  (%r8,%r9,1),%xmm0
0.00 :          400832:    mulss  (%r10,%rax,4),%xmm0
100.00 :          400838:    add    $0x1,%rax
:      }
:  }

```

Figura 5.26: Output do comando para o algoritmo *naive* com matrizes de tamanho 32x32.

De acordo com os resultados obtidos podemos verificar que grande parte do tempo do programa é gasto em instruções relacionadas com as somas e multiplicações necessárias para calcular a matriz resultado. Estas instruções consistem em instruções de multiplicação, de movimentação de dados e de adição. A percentagem das mesmas varia consideravelmente com os tamanhos. Verifica-se também para os vários tamanhos que é gasto um tempo considerável na operação de salto condicional associado ao *loop* mais interno.

Como existem complicações na interpretação do resultado do comando anterior devido a otimizações realizadas pelo compilador e dificuldades em associar o output em *assembly* ao respetivo código foi também utilizado este comando com a opção `--no-source` para obter apenas o *dissassembly* anotado.

```
> perf annotate --stdio --dsos=naive
--symbol="multiply_matrices(float**, float**, float**, int)" --no-source
```

```

:      Disassembly of section .text:
:
:      00000000004007f0 <multiply_matrices(float**, float**, float**, int)>:
0.00 : 4007f0: test    %ecx,%ecx
0.00 : 4007f2: jle     40085d <multiply_matrices(float**, float**, float**, int)+0x6d>
0.00 : 4007f4: lea     -0x1(%rcx),%eax
0.00 : 4007f7: pxor    %xmm2,%xmm2
0.00 : 4007fb: push    %rbp
0.00 : 4007fc: mov     %rdx,%rbp
0.00 : 4007ff: push    %rbx
0.00 : 400800: lea     0x4(%rax,4),%rdx
0.00 : 400808: mov     %rdi,%rbx
0.00 : 40080b: xor     %edi,%edi
0.00 : 40080d: mov     (%rbx,%rdi,8),%r10
0.00 : 400811: mov     0x0(%rbp,%rdi,8),%r11
0.00 : 400816: xor     %r9d,%r9d
0.00 : 400819: nopl    0x0(%rax)
0.01 : 400820: movaps  %xmm2,%xmm1
0.00 : 400823: xor     %eax,%eax
0.00 : 400825: nopl    (%rax)
5.11 : 400828: mov     (%rsi,%rax,8),%r8
0.14 : 40082c: movss   (%r8,%r9,1),%xmm0
68.09 : 400832: mulss   (%r10,%rax,4),%xmm0
16.42 : 400838: add     $0x1,%rax
0.01 : 40083c: cmp     %eax,%ecx
0.00 : 40083e: addss   %xmm0,%xmm1
10.23 : 400842: jg      400828 <multiply_matrices(float**, float**, float**, int)+0x38>
0.00 : 400844: movss   %xmm1,(%r11,%r9,1)
0.01 : 40084a: add     $0x4,%r9
0.00 : 40084e: cmp     %rdx,%r9
0.00 : 400851: jne     400820 <multiply_matrices(float**, float**, float**, int)+0x30>
0.00 : 400853: add     $0x1,%rdi
0.00 : 400857: cmp     %edi,%ecx
0.00 : 400859: jg      40080d <multiply_matrices(float**, float**, float**, int)+0x1d>
0.00 : 40085b: pop     %rbx
0.00 : 40085c: pop     %rbp
0.00 : 40085d: repz    retq

```

Figura 5.27: Output do comando para o algoritmo *naive* com matrizes de tamanho 2048x2048.

```

:      Disassembly of section .text:
:
:      00000000004007f0 <multiply_matrices(float**, float**, float**, int)>:
0.00 : 4007f0: test    %ecx,%ecx
0.00 : 4007f2: jle     40085d <multiply_matrices(float**, float**, float**, int)+0x6d>
0.00 : 4007f4: lea     -0x1(%rcx),%eax
0.00 : 4007f7: pxor    %xmm2,%xmm2
0.00 : 4007fb: push    %rbp
0.00 : 4007fc: mov     %rdx,%rbp
0.00 : 4007ff: push    %rbx
0.00 : 400800: lea     0x4(%rax,4),%rdx
0.00 : 400808: mov     %rdi,%rbx
0.00 : 40080b: xor     %edi,%edi
0.00 : 40080d: mov     (%rbx,%rdi,8),%r10
0.00 : 400811: mov     0x0(%rbp,%rdi,8),%r11
0.00 : 400816: xor     %r9d,%r9d
0.00 : 400819: nopl    0x0(%rax)
0.08 : 400820: movaps  %xmm2,%xmm1
0.00 : 400823: xor     %eax,%eax
0.00 : 400825: nopl    (%rax)
13.78 : 400828: mov     (%rsi,%rax,8),%r8
0.33 : 40082c: movss   (%r8,%r9,1),%xmm0
30.27 : 400832: mulss   (%r10,%rax,4),%xmm0
32.49 : 400838: add     $0x1,%rax
0.00 : 40083c: cmp     %eax,%ecx
0.00 : 40083e: addss   %xmm0,%xmm1
22.97 : 400842: jg      400828 <multiply_matrices(float**, float**, float**, int)+0x38>
0.00 : 400844: movss   %xmm1,(%r11,%r9,1)
0.08 : 40084a: add     $0x4,%r9
0.00 : 40084e: cmp     %rdx,%r9
0.00 : 400851: jne     400820 <multiply_matrices(float**, float**, float**, int)+0x30>
0.00 : 400853: add     $0x1,%rdi
0.00 : 400857: cmp     %edi,%ecx
0.00 : 400859: jg      40080d <multiply_matrices(float**, float**, float**, int)+0x1d>
:

```

Figura 5.28: Output do comando para o algoritmo *naive* com matrizes de tamanho 512x512.

```

:
: Disassembly of section .text:
:
: 0000000004007f0 <multiply_matrices(float**, float**, float**, int)>:
0.00 : 4007f0: test %ecx,%ecx
0.00 : 4007f2: jle 40085d <multiply_matrices(float**, float**, float**, int)+0x6d>
0.00 : 4007f4: lea -0x1(%rcx),%eax
0.00 : 4007f7: pxor %xmm2,%xmm2
0.00 : 4007fb: push %rbp
0.00 : 4007fc: mov %rdx,%rbp
0.00 : 4007ff: push %rbx
0.00 : 400800: lea 0x4(%rax,4),%rdx
0.00 : 400808: mov %rdi,%rbx
0.00 : 40080b: xor %edi,%edi
0.00 : 40080d: mov (%rbx,%rdi,8),%r10
0.00 : 400811: mov 0x0(%rbp,%rdi,8),%r11
0.00 : 400816: xor %r9d,%r9d
0.00 : 400819: nopl 0x0(%rax)
0.00 : 400820: movaps %xmm2,%xmm1
0.00 : 400823: xor %eax,%eax
0.00 : 400825: nopl (%rax)
44.44 : 400828: mov (%rsi,%rax,8),%r8
0.00 : 40082c: movss (%r8,%r9,1),%xmm0
5.56 : 400832: mulss (%r10,%rax,4),%xmm0
16.67 : 400838: add $0x1,%rax
0.00 : 40083c: cmp %eax,%ecx
0.00 : 40083e: addss %xmm0,%xmm1
27.78 : 400842: jg 400828 <multiply_matrices(float**, float**, float**, int)+0x38>
0.00 : 400844: movss %xmm1,(%r11,%r9,1)
5.56 : 40084a: add $0x4,%r9
0.00 : 40084e: cmp %rdx,%r9
0.00 : 400851: jne 400820 <multiply_matrices(float**, float**, float**, int)+0x30>
0.00 : 400853: add $0x1,%rdi
0.00 : 400857: cmp %edi,%ecx
0.00 : 400859: jg 40088d <multiply_matrices(float**, float**, float**, int)+0x1d>
0.00 : 40085b: pop %rbx
0.00 : 40085c: pop %rbp
0.00 : 40085d: repz retq

```

Figura 5.29: Output do comando para o algoritmo *naive* com matrizes de tamanho 128x128.

```

:
: Disassembly of section .text:
:
: 0000000004007f0 <multiply_matrices(float**, float**, float**, int)>:
0.00 : 4007f0: test %ecx,%ecx
0.00 : 4007f2: jle 40085d <multiply_matrices(float**, float**, float**, int)+0x6d>
0.00 : 4007f4: lea -0x1(%rcx),%eax
0.00 : 4007f7: pxor %xmm2,%xmm2
0.00 : 4007fb: push %rbp
0.00 : 4007fc: mov %rdx,%rbp
0.00 : 4007ff: push %rbx
0.00 : 400800: lea 0x4(%rax,4),%rdx
0.00 : 400808: mov %rdi,%rbx
0.00 : 40080b: xor %edi,%edi
0.00 : 40080d: mov (%rbx,%rdi,8),%r10
0.00 : 400811: mov 0x0(%rbp,%rdi,8),%r11
0.00 : 400816: xor %r9d,%r9d
0.00 : 400819: nopl 0x0(%rax)
0.00 : 400820: movaps %xmm2,%xmm1
0.00 : 400823: xor %eax,%eax
0.00 : 400825: nopl (%rax)
0.00 : 400828: mov (%rsi,%rax,8),%r8
0.00 : 40082c: movss (%r8,%r9,1),%xmm0
0.00 : 400832: mulss (%r10,%rax,4),%xmm0
100.00 : 400838: add $0x1,%rax
0.00 : 40083c: cmp %eax,%ecx
0.00 : 40083e: addss %xmm0,%xmm1
0.00 : 400842: jg 400828 <multiply_matrices(float**, float**, float**, int)+0x38>
0.00 : 400844: movss %xmm1,(%r11,%r9,1)
0.00 : 40084a: add $0x4,%r9
0.00 : 40084e: cmp %rdx,%r9
0.00 : 400851: jne 400820 <multiply_matrices(float**, float**, float**, int)+0x30>
0.00 : 400853: add $0x1,%rdi
0.00 : 400857: cmp %edi,%ecx
0.00 : 400859: jg 40088d <multiply_matrices(float**, float**, float**, int)+0x1d>
0.00 : 40085b: pop %rbx
0.00 : 40085c: pop %rbp
0.00 : 40085d: repz retq

```

Figura 5.30: Output do comando para o algoritmo *naive* com matrizes de tamanho 32x32.

Como seria de esperar os resultados obtidos foram semelhantes aos obtidos anteriormente.

5.3.7 Incrementar a frequência das amostras

Dado que as medições do *Perf* são realizadas estatisticamente medi o número de amostras obtido para cada um dos eventos *clock – rate* e *faults* para vários de tamanhos para poder visualizar o

impacto resultante do aumento da frequência no número de amostras recolhidas. Utilizei o comando abaixo para obter a frequência inicial.

```
> perf evlist -F
```

Inicialmente a frequência era 4000. Com o comando seguinte medi o numero de amostras utilizando as frequências de 8000 e 4000.

```
> perf record -e cpu-clock --freq=freq ./bin/naive 1 size
```

Sample frequency	clock-rate	32 faults
4000	115471	12663
8000	242611	12654

Tabela 5.6: *Samples* recolhidas para os diferentes eventos utilizando o algoritmo de multiplicação de matrizes *naive* com matrizes de tamanho 2048x2048

Sample frequency	clock-rate	32 faults
4000	9952	3416
8000	22132	3409

Tabela 5.7: *Samples* recolhidas para os diferentes eventos utilizando o algoritmo de multiplicação de matrizes *naive* com matrizes de tamanho 512x512

Sample frequency	clock-rate	32 faults
4000	36	469
8000	64	423

Tabela 5.8: *Samples* recolhidas para os diferentes eventos utilizando o algoritmo de multiplicação de matrizes *naive* com matrizes de tamanho 128x128

Sample frequency	clock-rate	32 faults
4000	13	713
8000	25	313

Tabela 5.9: *Samples* recolhidas para os diferentes eventos utilizando o algoritmo de multiplicação de matrizes *naive* com matrizes de tamanho 32x32

Os resultados obtidos mostram que para o evento *cpu - clock* o número de amostras duplica com a duplicação do tamanho da frequência utilizada. Para o evento *faults* verifica-se que os resultados são similares para os 3 maiores tamanhos tendo apenas alguma variação para o menor tamanho.

5.3.8 Eventos utilizados

Foi utilizado o comando abaixo a fim de verificar quais os eventos disponíveis no sistema de teste. A partir da análise dos eventos disponíveis verifiquei que a maioria dos eventos utilizados para os testes subsequentes estão disponíveis na máquina em questão.

```
> perf list
```

```

List of pre-defined events (to be used in -e):
cpu-cycles OR cycles                [Hardware event]
instructions                        [Hardware event]
cache-references                    [Hardware event]
cache-misses                       [Hardware event]
branch-instructions OR branches    [Hardware event]
branch-misses                      [Hardware event]
stalled-cycles-frontend OR idle-cycles-frontend [Hardware event]
stalled-cycles-backend OR idle-cycles-backend  [Hardware event]

cpu-clock                          [Software event]
task-clock                        [Software event]
page-faults OR faults             [Software event]
context-switches OR cs            [Software event]
cpu-migrations OR migrations      [Software event]
minor-faults                      [Software event]
major-faults                     [Software event]
alignment-faults                  [Software event]
emulation-faults                  [Software event]

L1-dcache-loads                   [Hardware cache event]
L1-dcache-load-misses             [Hardware cache event]
L1-dcache-stores                  [Hardware cache event]
L1-dcache-store-misses            [Hardware cache event]
L1-dcache-prefetches              [Hardware cache event]
cpu-clock                        [Software event]
task-clock                        [Software event]
page-faults OR faults             [Software event]
:
LLC-load-misses                   [Hardware cache event]
LLC-stores                       [Hardware cache event]
LLC-store-misses                  [Hardware cache event]
LLC-prefetches                   [Hardware cache event]
LLC-prefetch-misses              [Hardware cache event]
dTLB-loads                       [Hardware cache event]
dTLB-load-misses                  [Hardware cache event]
dTLB-stores                      [Hardware cache event]
dTLB-store-misses                [Hardware cache event]
iTLB-loads                       [Hardware cache event]
iTLB-load-misses                 [Hardware cache event]
branch-loads                     [Hardware cache event]
branch-load-misses               [Hardware cache event]

```

Figura 5.31: Eventos disponíveis na máquina de teste

5.3.9 Análise das diferentes versões do algoritmo e tamanhos

Foram medidos para os vários tamanhos e para os dois algoritmos os valores de vários eventos e estes foram subsequentemente utilizados para calcular um conjunto de métricas. Para facilitar os testes e evitar que os vários contadores interfiram nos valores dos outros foi utilizado uma *shell script* que permita automatizar as medições e medir cada evento independentemente.

```

> perf record -e $metric ./bin/naive 1 2048
> perf report --stdio --show-nr-samples --dsos=naive | grep "Event"

```

Event	naive	interchange
cpu-cycles	153579835565	18872420661
cpu-clock	172034	18524
L1-dcache-load-misses	11676490203	553111515
L1-dcache-loads	25998720494	6631326934
L1-dcache-store-misses	7039985	1391208
instructions	60876847831	22086792694
cache-misses	439586633	198526441
branch-misses	4542740	4312800
cpu-migrations	-	-
branches	8768057460	2311156792
L1-dcache-loads	25997343784	6631155117
L1-dcache-load-misses	11569448154	553059738
L1-dcache-stores	124533957	2246988346
L1-dcache-store-misses	6739880	1387443
L1-icache-loads	492611767	491902057
LLC-loads	1769729847	103888720
LLC-load-misses	441026564	91188369
LLC-store-misses	1124507	871207
dTLB-load-misses	3211089527	13032985
iTLB-load-misses	6492	2513
branch-loads	8768875350	2309001206
branch-load-misses	212944277	192790716

Tabela 5.10: Resultados dos diferentes eventos para as diferentes implementações do algoritmo utilizando como *input* matrizes de tamanho 2048x2048

Metric	naive	interchange
Instructions per cycle	1.035	2.07
L1 cache miss ratio	1074	980
L1 cache miss RATE PTI	9.2	1.09
Data TLB miss rate PTI	9.2	1.09
Branch mispredicted ratio	0.002	0.006
Branch mispredict rate PTI	0.303	0.76

Tabela 5.11: Diferentes métricas calculadas utilizando os valores dos eventos obtidos anteriormente para as duas versões do algoritmo com input matrizes de tamanho 2048x2048

Event	naive	interchange
cpu-cycles	946050773	182284638
cpu-clock	1313	267
L1-dcache-load-misses	9187142	221568
L1-dcache-loads	414816403	113304751
L1-dcache-store-misses	385986	115571
instructions	979881005	377382887
cache-misses	77272	73946
branch-misses	297242	288866
cpu-migrations	-	-
branches	144118235	44374194
L1-dcache-loads	414672586	113348901
L1-dcache-load-misses	161378094	9193286
L1-dcache-stores	6969231	40054004
L1-dcache-store-misses	386466	117283
L1-icache-loads	27057094	27434975
LLC-loads	8790308	1804307
LLC-load-misses	13608	13006
LLC-store-misses	56781	52756
dTLB-load-misses	9015504	414743
iTLB-load-misses	1253	1425
branch-loads	144090002	44342765
branch-load-misses	11276553	11293725

Tabela 5.12: Resultados dos diferentes eventos para as diferentes implementações do algoritmo utilizando como *input* matrizes de tamanho 512x512

Metric	naive	interchange
Instructions per cycle	0.39	1.17
L1 cache miss ratio	2.25	11.99
L1 cache miss RATE PTI	190	25
Data TLB miss rate PTI	52.7	0.59
Branch mispredicted ratio	0.00051	0.086
Branch mispredict rate PTI	0.07	0.20

Tabela 5.13: Diferentes métricas calculadas utilizando os valores dos eventos obtidos anteriormente para as duas versões do algoritmo com input matrizes de tamanho 2048x2048

Event	naive	interchange
cpu-cycles	12772194	8018015
cpu-clock	30	17
L1-dcache-load-misses	221568	235263
L1-dcache-loads	8037070	3538381
L1-dcache-store-misses	1184906	1083633
instructions	19396	17889
cache-misses	22026	18639
branch-misses	40373	25559
cpu-migrations	-	-
branches	3559816	2054391
L1-dcache-loads	8165605	3352742
L1-dcache-load-misses	224487	233555
L1-dcache-loads	242611	12654
L1-dcache-stores	824538	1298700
L1-dcache-store-misses	19087	17995
L1-icache-loads	3404898	4275438
LLC-loads	27576	24772
LLC-load-misses	10052	9254
LLC-store-misses	9196	9481
dTLB-load-misses	12436	11190
iTLB-load-misses	1910	1257
branch-loads	3479978	2040020
branch-load-misses	1290122	1474193

Tabela 5.14: Resultados dos diferentes eventos para as diferentes implementações do algoritmo utilizando como *input* matrizes de tamanho 128x128

Metric	naive	interchange
Instructions per cycle	0.0015	0.0022
L1 cache miss ratio	36.27	15.04
L1 cache miss RATE PTI	414367.4	197796.47
Data TLB miss rate PTI	641.16	625.52
Branch mispredicted ratio	0.011	0.012
Branch mispredict rate PTI	2081.5	1428

Tabela 5.15: Diferentes métricas calculadas utilizando os valores dos eventos obtidos anteriormente para as duas versões do algoritmo com input matrizes de tamanho 128x128

Event	naive	interchange
cpu-cycles	5186860	4707675
cpu-clock	10	9
L1-dcache-load-misses	64811	68625
L1-dcache-loads	1184906	1083633
L1-dcache-store-misses	9973	10821
instructions	4991671	4656035
cache-misses	18196	16385
branch-misses	25175	24280
cpu-migrations	-	-
branches	923332	881412
L1-dcache-loads	1253916	1150298
L1-dcache-load-misses	66573	68785
L1-dcache-stores	424239	436621
L1-dcache-store-misses	11366	11349
L1-icache-loads	1988026	2019354
LLC-loads	26817	23894
LLC-load-misses	8581	8700
LLC-store-misses	4663	4433
dTLB-load-misses	10223	8816
iTLB-load-misses	1429	1213
branch-loads	914612	899932
branch-load-misses	674893	659745

Tabela 5.16: Resultados dos diferentes eventos para as diferentes implementações do algoritmo utilizando como *input* matrizes de tamanho 32x32

Metric	naive	interchange
Instructions per cycle	499167.1	517337.22
L1 cache miss ratio	18.28	15.79
L1 cache miss RATE PTI	237.4	232.74
Data TLB miss rate PTI	2.048	1.89
Branch mispredicted ratio	0.02	1
Branch mispredict rate PTI	5.04	5.2

Tabela 5.17: Diferentes métricas calculadas utilizando os valores dos eventos obtidos anteriormente para as duas versões do algoritmo com input matrizes de tamanho 32x32

Em termos de instruções por ciclo podemos verificar que para os vários tamanhos a performance da versão *interchange* é superior. Em termos de *miss rates* para o maior tamanho encontrei o *cache miss ratio* é inferior para a versão *interchange*, para o segundo maior tamanho verifica-se o oposto, no entanto para ambas estas versões ambos os *L1 cache miss RATE PTI* e *Data TLB miss rate PTI* são consideravelmente inferiores para a versão *interchange* isto justifica o maior numero de instruções por ciclo desta versão por perder menos tempo em acessos a memoria. Os resultados para os dois menores tamanhos foram inconclusivos e poderão ser atributivos a falta de dados.

5.3.10 Comparação entre *Counting mode* e *Sampling*

O *perf* suporta a realização das medições dos vários eventos utilizando *sampling*, esta técnica de medição consiste em recolher várias amostras durante a execução do programa e posteriormente

juntá-las de modo a obter informação necessária à sua avaliação. Foram comparados os resultados das medições realizadas com os dois modos para os dois tamanhos maiores e foram também calculadas métricas utilizando esses resultados. As métricas utilizando o *Sampling mode* foram recolhidas da mesma forma do que as métricas anteriores utilizando *shell scripts* e o comando abaixo. O período utilizado foi 100000, este período foi utilizado por ser o mesmo do que o utilizado no tutorial.

```
> perf record -e metric -c period ./bin/naive 1 size
> perf report --stdio --show-nr-samples --dsos=naive | grep "Event"
```

Event	naive 2048x2048	interchange 2048x2048	naive 512x512	interchange 512x512
cpu-cycles	142389883755	13261216763	957819437	184828719
cpu-clock	158948	17344	1277	260
cache-references	1925133684	103747696	89777771	1852816
cache-misses	431869403	85727780	49341	29002
instructions	61058300000	22171400000	982800000	378200000
LLC-loads	1988954894	101944331	8748003	1818427
LLC-load-misses	432113925	83135408	11994	9720
dTLB-load-misses	2328224247	12988596	9015770	412392
branches	8763879676	2309228272	144096534	44367590
branch-misses	4519347	4293829	288968	288711

Tabela 5.18: Resultados dos diferentes eventos para as diferentes implementações do algoritmo utilizando como *input* matrizes de tamanho 32x32

Event	naive 2048x2048	interchange 2048x2048	naive 512x512	interchange 512x512
cpu-cycles	145812500000	16049600000	953000000	183900000
cpu-clock	42867400000	4434900000	321800000	69600000
cache-references	1531700000	191000000	8900000	1800000
cache-misses	438200000	81200000	-	-
instructions	60861147730	22086293233	980082628	377841466
LLC-loads	2144100000	97500000	8700000	1700000
LLC-load-misses	437700000	80900000	-	-
dTLB-load-misses	2421900000	12800000	9000000	400000
branches	8756400000	2309400000	143900000	44100000
branch-misses	4400000	4200000	200000	200000

Tabela 5.19: Resultados dos diferentes eventos para as diferentes implementações do algoritmo utilizando como *input* matrizes de tamanho 32x32

Metric	naive 2048x2048	interchange 2048x2048	naive 512x512	interchange 512x512
Instructions per cycle	0.429	1.672	1.026	2.046
Cache miss ratio	4.458	1.21	181.954	63.886
Cache miss rate PTI	31.529	4.679	9.135	4.899
LLC load miss ratio	4.603	1.226	729.365	187.081
LLC load miss rate PTI	7.077	3.75	0.012	0.026
dTLB load miss rate PTI	38.131	0.586	9.174	1.09
Branch mispredict ratio	0.0015	0.0025	0.0013	0.005
Branch mispredict rate PTI	0.076	0.22	0.210	0.550

Tabela 5.20: Resultados dos diferentes eventos para as diferentes implementações do algoritmo utilizando como *input* matrizes de tamanho 32x32

Metric	naive 2048x2048	interchange 2048x2048	naive 512x512	interchange 512x512
Instructions per cycle	0.417	1.376	1.028	2.055
Cache miss ratio	3.495	2.352	-	-
Cache miss rate PTI	25.167	8.648	9.081	4.764
LLC load miss ratio	4.899	1.205	-	-
LLC load miss rate PTI	7.192	3.663	-	-
dTLB load miss rate PTI	39.794	0.58	9.183	1.059
Branch mispredict ratio	0.001	0.002	0.001	0.005
Branch mispredict rate PTI	0.072	0.19	0.204	0.529

Tabela 5.21: Resultados dos diferentes eventos para as diferentes implementações do algoritmo utilizando como *input* matrizes de tamanho 32x32

Com a exceção das medições relacionadas com o numero de *cache misses* que não foram obtidas para o tamanho 512x512, a não existência dessas medições poderá ser atribuída ao tamanho insuficiente do tempo de amostra, as medições apresentam-se similares entre os dois métodos utilizados.

5.3.11 Utilização de *FlameGraphs* para análise dos algoritmos

Foram também utilizados *FlameGraphs* para avaliar as chamadas ao sistema dos vários algoritmos para os dois maior tamanhos.

```
> perf record -F 99 -ag ./bin/naive version size
> perf script | ./FlameGraph/stackcollapse-perf.pl
| ./FlameGraph/flamegraph.pl > tests/name.svg
```

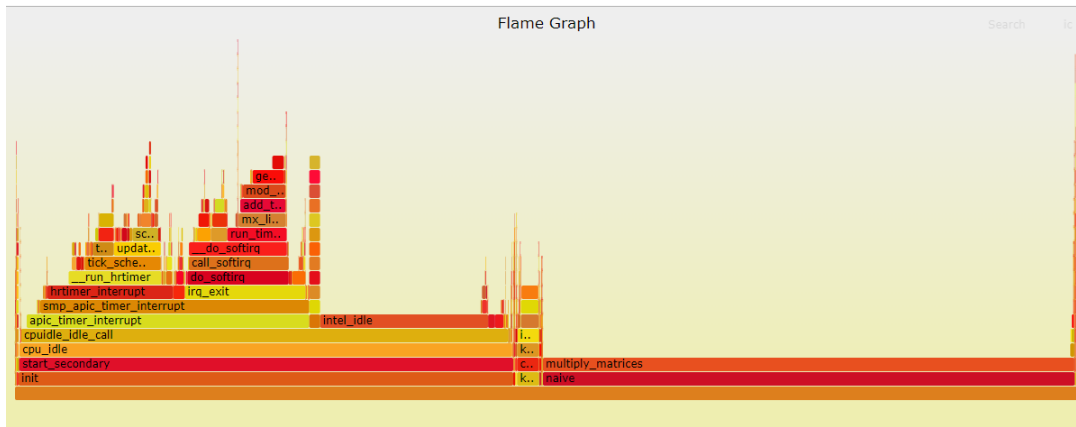


Figura 5.32: *FlameGraph* para o algoritmo *naive* e matrizes de tamanho 2048x2048.

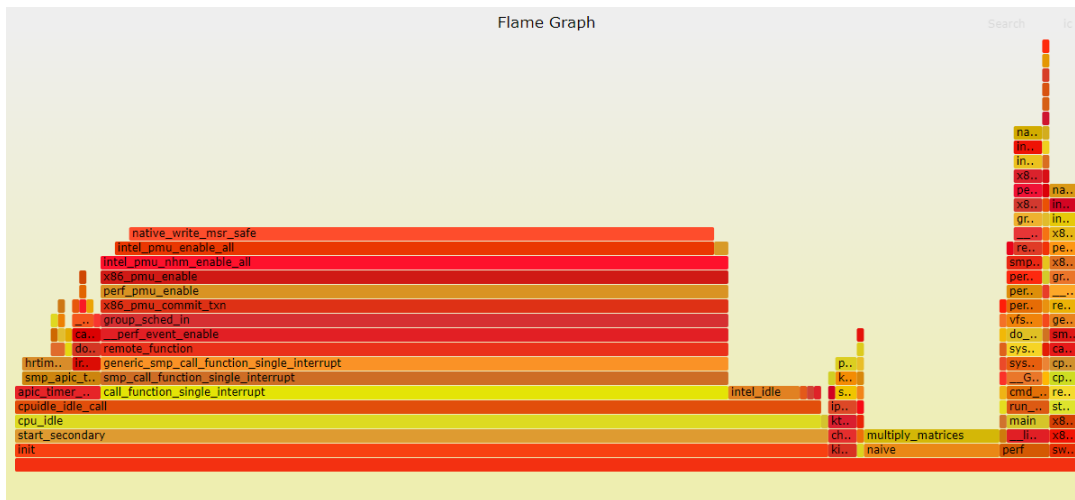


Figura 5.33: *FlameGraph* para o algoritmo *naive* e matrizes de tamanho 512x512.

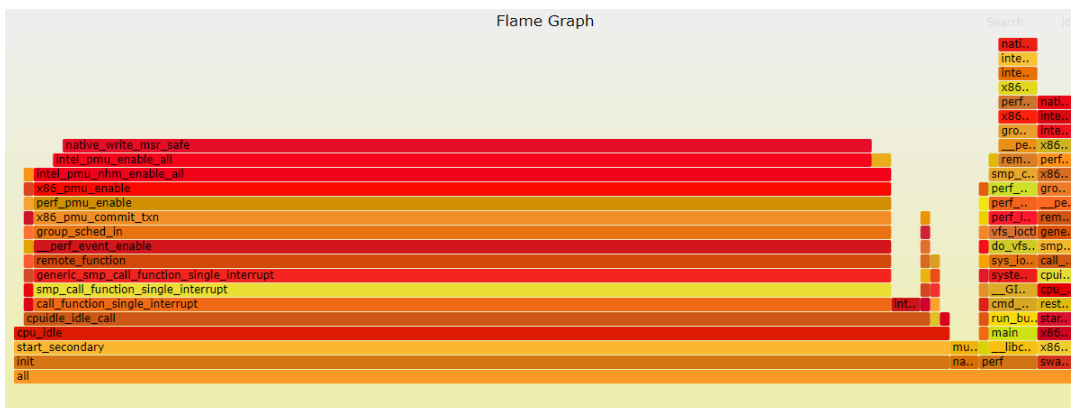


Figura 5.34: *FlameGraph* para o algoritmo *interchange* e matrizes de tamanho 2048x2048.

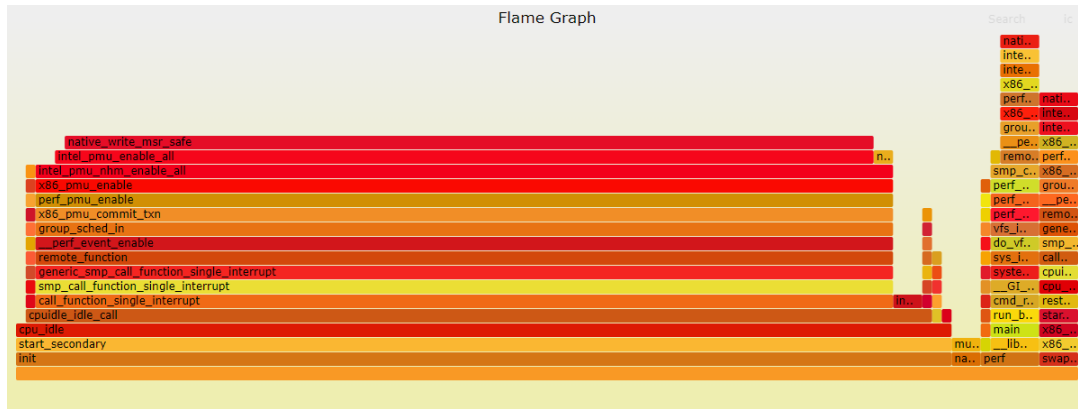


Figura 5.35: *FlameGraph* para o algoritmo *interchange* e matrizes de tamanho 512x512.

Os *FlameGraphs* permitem verificar para o tamanho maior que a versão *naive* ocupa uma percentagem maior do tempo no seu nível da *stack* relativamente a versão *interchange* isto acontece porque esta função é menos eficiente.

5.4 Conclusões e trabalho futuro

Este trabalho permitiu ganhar familiaridade com as ferramentas *Perf* e *FlameGraph*, permitiu também estudar diferentes algoritmos e perceber algumas das motivações por detrás das diferenças de performance.

Em termos de algoritmos de ordenação pude verificar que o algoritmo *radix-sort LSD* apresenta melhor desempenho relativamente aos outros e que estas diferenças assentam principalmente no menor numero de *branches* e *branch-misses* mas também pela sua complexidade e comportamento no que toca a chamadas recursivas. Encontrei também alguns *bottlenecks* no desempenho da aplicação este encontra-se principalmente no acesso aos dados.

No que toca aos algoritmos de multiplicação de matrizes o trabalho realizado permitiu visualizar diferenças em ambos os algoritmos, sendo que estas assentam principalmente em acessos a memória, permitiu também visualizar os pontos quentes da aplicação.

Em termos de trabalho futuro, as ferramentas *Perf* e *FlameGraphs* poderão ser utilizadas em vários outros programas para medir, estudar e melhor o desempenho de diversas aplicações.

Capítulo 6

Conclusões e trabalho futuro

A realização dos vários trabalhos permitiu ganhar familiaridade com várias ferramentas e utilitários que permitem avaliar o desempenho de aplicações e sistemas, permitiu também ganhar conhecimento que permite melhor avaliar os mesmos e identificar problemas que limitem o seu desempenho. A realização dos mesmos possibilitou também ganhar experiência no desenvolvimento de programas em *c++* moderno e com recurso a biblioteca *pthread*.

Em termos de trabalho futuro os conhecimentos ganhos com a realização destes trabalhos poderá ser aplicada em varias tarefas diretamente e indiretamente relacionadas com o trabalho realizado nomeadamente em *High Performance Computing* e em varias áreas em que o desempenho das aplicações tenha algum relevo.

Bibliografia

- [1] Paul J. Drongowski. *PERF tutorial: Counting hardware performance events*. 2015. URL: <http://sandsoftwaresound.net/perf/perf-tut-count-hw-events/>.
- [2] Paul J. Drongowski. *PERF tutorial: Finding execution hot spots*. 2015. URL: <http://sandsoftwaresound.net/perf/perf-tutorial-hot-spots/>.
- [3] Paul J. Drongowski. *PERF tutorial: Profiling hardware events*. 2015. URL: <http://sandsoftwaresound.net/perf/perf-tut-profile-hw-events/>.
- [4] *dtrace.org*. 2020. URL: <http://dtrace.org/blogs/about/>.
- [5] *DTracing Hardware Cache Counters*. 2013. URL: <https://www.joyent.com/blog/dtracing-hardware-cache-counters>.
- [6] Brendan Gregg. *Cpuu FlameGraphs*. 2015. URL: <http://www.brendangregg.com/FlameGraphs/cpuflamegraphs.html>.
- [7] *Oracle Linux DTrace Tutorial*. 2019.
- [8] *Oracle Solaris 11.4 DTrace (Dynamic Tracing) Guide*. 2019.
- [9] *Sun HPC ClusterTools 8 Software User's Guide*. 2008. URL: <https://docs.oracle.com/cd/E19356-01/820-3176-10/Dtrace-mpiperuse.html>.