

# FX1000利用のための プログラミングガイド活用

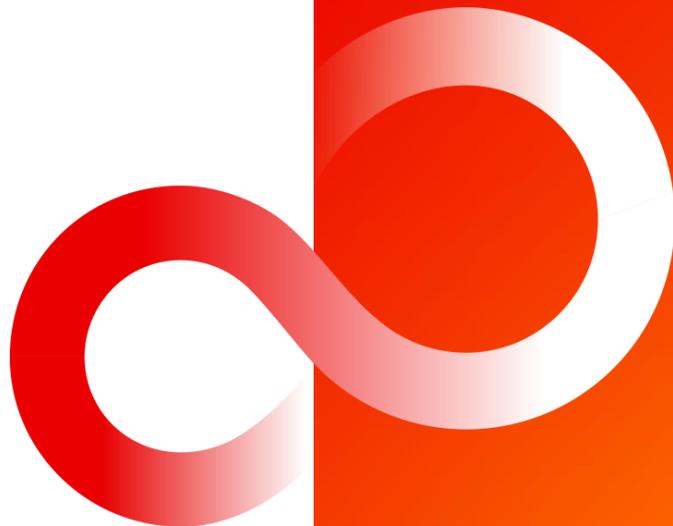
2022年9月13日

富士通株式会社

ミッションクリティカルシステム事業本部

UNIX & FXシステム事業部 FX言語ソフトウェア部

中村朋健



- プログラミングガイド概要・種類
- 各プログラミングガイドの紹介
  - プログラミング共通編
  - プロセッサ編
  - Fortran編
  - チューニング編
  
- プログラミングガイドの活用例

## ○ 概要

- FX1000の利用者がプログラミングやチューニングするうえで役立つ情報をまとめたガイド
  - 使用手引書などの製品マニュアルは仕様や機能を網羅的かつ詳細に記載、一方、プログラミングガイドはプログラミングやチューニングに具体的に適用できる内容を記載
- 利用者ポータル HPC Portal マニュアルの言語製品を選択
  - <https://portal.cc.nagoya-u.ac.jp/cgi-bin/hpcportal.ja/index.cgi>

The screenshot shows the HPC Portal website. The left sidebar contains a menu with items: About, パスワード変更, SSH公開鍵登録, お知らせ, マニュアル, 利用手引書, 言語製品 (highlighted with a green arrow), その他, and 動画. The main content area is titled '言語製品' and contains a warning: 'FX1000 言語製品 (本書の一部、または全部を無断で複製、転載、再配布することを禁じます)'. Below this is a section for 'FX1000チューニングガイド' with a table of documents:

ドキュメント名	最新更新日
<a href="#">プログラミングガイド プログラミング共通編</a>	2022/07/27
<a href="#">プログラミングガイド プロセッサ編</a>	2022/06/30
<a href="#">プログラミングガイド Fortran編</a>	2022/07/27
<a href="#">プログラミングガイド チューニング編</a>	2022/07/27

Below the table is a section for '概説書' with another table:

ドキュメント名	最新更新日
<a href="#">Development Studio 概説書</a>	2020/06/29

The footer of the page reads 'Fujitsu HPC Portal' on the left and 'V03L02' on the right.

## ○プログラミングガイドの種類

- 各ガイドは和文と英文を用意

	内容
プログラミング 共通編	言語製品の構成、コンパイラの種類や使い分けなどをまとめたガイド。
プロセッサ編	FX1000のプロセッサA64FXの特徴、マイクロアーキ、および性能情報をまとめたガイド。プロセッサの諸元や基礎性能値を説明。
Fortran編	Fortranを使用してプログラミングやチューニングするときの注意点をまとめたガイド。
チューニング編	1コア内やスレッド並列でチューニングするときに役立つ情報をまとめたガイド。様々なチューニング手法を適用したときのチューニングの効果について説明。 Fortranのチューニング例のみで記載していたが、C/C++利用者からの要望も多く、2022年7月の更新でC/C++のチューニング方法を追加。

## ○ プログラミング共通編の主な記載内容

言語製品の構成、コンパイラの種類や使い分けなどをまとめたガイド。

- 言語パッケージの内容
- コンパイラ推奨オプション
- C/C++のtradモードとclangモード
- 他社コンパイラが生成するオブジェクトの互換
- clangモードのオプションと最適化指示子
- OpenMPライブラリ（LLVM OpenMPと富士通OpenMP）
- 従来システムからの移行
- コンパイラによる高速化
- tradモードのデバッグ機能
- ラージページ
- 注意事項



- 「プログラミング共通編」の例
  - tradモードとclangモードの使い分け

ユーザ様が迷いやすいところ

## C/C++のtrad/clangモードの選択について FUJITSU

- 想定している利用シーン（使い分け）

利用シーン	シーンに適したコンパイラ	
	tradモード	clangモード
京,FX100で作成した資源をそのまま利用したい	○	×
HPC向けのチューニングをしたい	○	○ (オプション指定が必要)
OSSを利用したい	×	○
C++17の新しい言語規格を利用したい	○ (一部サポート)	○

- コンパイラの構造



## コンパイラ機能（一般、ループ最適化）比較 FUJITSU

	項目	tradモード	clangモード	備考
				2021年3月31日時点
一般	OpenMP	○	○	clangモードはOpenMP 4.5をサポート(ただし、declare simd構文およびtaskloop simd構文のlinear指示節は除く)
	自動並列	○	×	
	ACLE	×	○	
	lto	×	○	
ループ最適化	ソフトウェアパイプライン	○	△	clangモードはオプション指定
	多重ループのループ交換	○	○	
	多重ループの一重化	○	×	
	ループアンローリング	○	○	
	フルアンローリング	○	○	
	ループ分割	△	△	オプション指定
	ループ融合	○	×	
	アンスイッチング	○	○	
マルチバージョン	△	△	ともに個別機能のみ	

## ○主な記載内容

FX1000のプロセッサA64FXの特徴、マイクロアーキ、および性能情報をまとめたガイド。プロセッサの諸元や基礎性能値を説明。

- A64FXプロセッサの概要・諸元
- L2キャッシュの利用
- マイクロアーキ
- 基礎カーネル性能



- 「プロセッサ編」の例
  - A64FXプロセッサの諸元

チューニングする上で基礎となる情報

### A64FXプロセッサ諸元

項目	諸元
プロセッサ・コア数	52 (13 cores / CMG)
CMG数	4
L1Iキャッシュ・サイズ	64KiB / 4way
L1Dキャッシュ・サイズ	64KiB / 4way
L2キャッシュ・サイズ	32MiB / 16way (8MiB / CMG)
キャッシュライン・サイズ	256B
メモリ・サイズ	32GiB(8GiB/CMG)
インターコネクト	Tofu Interconnect D
I/O	PCI-Express Gen3 16 Lanes
命令セット・アーキテクチャ	ARMv8-A, ARMv8.1, ARMv8.2, ARMv8.3 (*1), SVE

(\*1) ARMv8.3 は Complex number support 命令のみサポートする。

7 再配布および不特定多数への公開禁止 Copyright 2021 FUJITSU LIMITED

### A64FXプロセッサ諸元：バンド幅、レイテンシ

	A64FX	備考
周波数 [GHz]	2.0	
CPU数/ノード	1	
演算コア数/ノード	48	
CMG数/ノード	4	
メモリ容量/ノード [GiB]	32	
キャッシュ容量	L1 [KiB/コア]	64(命令)+64(データ)
	L2 [MiB/CMG]	8
キャッシュレイテンシ [cycle]	L1	5 (EX, short) 8 (FL, short) 11 (FL, long)
	L2	37~47
キャッシュバンド幅 [B/cycle/コア]	L1	ヒット時：128
	L2	42.7
ノードあたり演算性能 (1コアあたり) [GFlops]	倍精度	3072 (64)
	単精度	6144 (128)
	半精度	12288 (256)
基本演算レイテンシ [cycle]	整数add命令	1
	整数mult命令	5
	FMA命令	9
メモレイテンシ [ns]	150	
ノードあたり理論メモリバンド幅 (CMGあたり) [GB/秒]	1024(256)	
CMG間バンド幅	128GB/s×2(双方向)	

8 再配布および不特定多数への公開禁止 Copyright 2021 FUJITSU LIMITED

- 「プロセッサ編」の例
  - A64FXプロセッサの基礎性能

各システムの特徴を把握する情報

### 四則演算・平方根 (1/2)

FUJITSU

■ 四則演算・平方根(実数型)の測定結果

		京		FX100		PRIMERGY RX2530 M1		PRIMERGY RX2540 M4		A64FX			
		浮動小数点演算性能 (Gflops)	性能向上比	演算効率 (%)	SIMD化効果								
倍精度	加算	1.38	1.00	3.37	2.48	4.61	3.52	8.57	6.54	7.42	5.38	11.59	6.55
	減算	1.38	1.00	3.40	2.49	4.62	3.52	8.46	6.46	7.42	5.38	11.59	6.55
	乗算	1.38	1.00	3.40	2.50	4.62	3.53	7.51	5.73	7.42	5.38	11.59	6.70
	積和演算	2.87	1.00	6.96	2.46	9.51	3.49	15.79	5.80	14.84	5.18	23.19	6.55
	除算	10.39	1.00	19.06	1.86	3.22	0.33	9.24	0.94	39.57	3.81	61.84	7.94
	逆数	10.79	1.00	19.02	1.79	5.60	0.55	8.38	0.82	39.85	3.69	62.26	8.33
	平方根	11.90	1.00	21.63	1.84	4.84	0.43	8.14	0.72	34.78	2.92	54.34	12.75
単精度	加算	1.69	1.00	6.57	3.93	9.11	5.66	13.59	8.45	13.84	8.18	10.81	12.16
	減算	1.68	1.00	6.62	3.99	9.11	5.71	13.78	8.64	13.84	8.24	10.81	12.16
	乗算	1.69	1.00	6.62	3.96	9.09	5.65	13.49	8.39	13.84	8.18	10.81	12.16
	積和演算	3.46	1.00	13.52	3.96	19.18	5.84	26.60	8.09	27.68	8.00	21.62	12.20
	除算	9.85	1.00	15.97	1.64	26.98	2.88	39.32	4.20	61.13	6.21	47.76	14.42
	逆数	9.99	1.00	16.84	1.71	28.13	2.97	44.43	4.68	72.00	7.21	56.25	18.64
	平方根	9.61	1.00	17.15	1.81	8.41	0.92	49.29	5.40	52.07	5.42	40.68	23.83

146 両配布および不特定多数への公開禁止 Copyright 2021 FUJITSU LIMITED

### 数学関数 (2/2)

FUJITSU

■ 他CPU比較

		京 (GFLOPS)	FX100 (GFLOPS)	Skylake (GFLOPS)	A64FX (GFLOPS)	京比	FX100比	Skylake比
倍精度	atan	8.71	13.25	18.80	15.41	1.77	1.16	0.78
	atan2	8.87	9.69	16.19	6.88	0.78	0.71	0.40
	cos	11.55	19.43	21.10	24.44	2.11	1.26	1.10
	exp	7.93	15.22	22.55	18.04	2.27	1.19	0.76
	exp10	7.99	15.18	22.19	17.25	2.16	1.14	0.74
	log	6.97	7.87	16.01	11.80	1.69	1.50	0.70
	log10	7.62	9.45	20.70	11.84	1.55	1.25	0.54
	sin	11.54	19.47	22.21	24.17	2.10	1.24	1.03
	べき乗	7.48	8.88	13.32	8.47	1.13	0.95	0.60
	単精度	atan	7.29	9.76	29.51	36.30	4.98	3.72
atan2		8.00	6.76	28.70	25.67	3.21	3.80	0.85
cos		10.98	16.37	39.92	48.93	4.46	2.99	1.16
exp		6.74	11.44	47.42	35.75	5.30	3.13	0.72
exp10		7.37	11.65	43.10	37.59	5.10	3.23	0.83
log		5.91	5.49	41.46	28.96	4.90	5.28	0.66
log10		6.29	6.67	52.90	30.70	4.88	4.60	0.55
sin		10.98	16.36	45.36	48.98	4.46	2.99	1.03
べき乗		7.30	6.55	25.56	17.30	2.37	2.64	0.64
GEOMEAN						2.66	2.00	0.76

149 両配布および不特定多数への公開禁止 Copyright 2021 FUJITSU LIMITED

## ○ 主な記載内容

Fortranを使用してプログラミングやチューニングするときの注意点をまとめたガイド。

- 最適化指定
  - 推奨オプション
  - SIMD、SWPL
  - プリフェッチ
  - ループの最適化
  - ロード・ストア
  - リンク時最適化
  - OpenMP
- オプションの注意事項
- 翻訳情報（診断メッセージ・ガイダンスメッセージ）
- プログラム実行
- プログラミング時の注意事項



## ○ 「Fortran編」 の例

Fortran固有の機能で役立つチューニング

### ○ do concurrentによるデータ依存の解消

pointerの性能チューニング (3/3) FUJITSU

- do concurrentによるデータ依存の解消

```
subroutine test(a,b,n1,n2)
real(kind=8),dimension(:,:),
pointer,contiguous :: a,b
do j=1,n2
  do i=1,n1
    a(i,j) = a(i,j) + b(i,j)
  enddo
enddo
end subroutine test
```

➔

```
subroutine test(a,b,n1,n2)
real(kind=8),dimension(:,:),
pointer,contiguous :: a,b
do concurrent(j=1:n2)
  do concurrent(i=1:n1)
    a(i,j) = a(i,j) + b(i,j)
  enddo
enddo
end subroutine test
```

修正差分による文法解釈の相違と最適化効果		pointer+contiguos	左記+do concurrent
定義配列と参照配列のデータ依存		不明	無
繰り返しを跨るデータ依存		不明	無
最内ループ1次元目のアクセス連続性		有	有
多重ループ全体のアクセス連続性		不明	不明
修正による最適化効果の可能性	自動並列	×	○
	S I M D	×	○
	(効果のある) S W P	×	○

162 再配布および不特定多数への公開禁止 Copyright 2022 FUJITSU LIMITED

- データ依存を解消する記載方法
- do concurrentによるデータ依存の有無
- 最適化の効果の有無

## ○ 「Fortran編」 の例

### ○ pointerとallocatableの違い

機能差がわかりにくい仕様

pointerとallocatableの違い

```

subroutine test(a,b,n1,n2)
real(kind=8),dimension(:,,:), pointer :: a,b
do j=1,n2
  do i=1,n1
    a(i,j) = a(i,j) + b(i,j)
  enddo
enddo
end subroutine test
                    
```

➔

```

subroutine test(a,b,n1,n2)
real(kind=8),dimension(:,,:), allocatable :: a,b
do j=1,n2
  do i=1,n1
    a(i,j) = a(i,j) + b(i,j)
  enddo
enddo
end subroutine test
                    
```

修正差分による文法解釈の相違と最適化効果	pointer	allocatable
定義配列と参照配列のデータ依存	不明	無
繰り返しを跨るデータ依存	不明	無
最内ループ1次元目のアクセス連続性	不明	有
多重ループ全体のアクセス連続性	不明	不明
修正による最適化効果の可能性	自動並列	○
	S I M D	○
	(効果のある) S W P	○

※ allocatableは形状明示と比較するとアドレス計算は多い

- pointerとallocatableの指定方法
- データ依存の有無
- 最適化の効果の有無

## ○ 主な記載内容

1コア内やスレッド並列でチューニングするときに役立つ情報をまとめたガイド。様々なチューニング手法を適用したときのチューニングの効果について説明。

- ボトルネックの調査
- 1コアチューニング
  - データ局所性の改善
  - データアクセス待ちの改善
  - スラッシングの改善
  - SIMD化による改善
  - 演算待ちの改善
  - マイクロアーキ依存の改善
- スレッド並列チューニング
  - 並列化率の改善
  - 実行効率の改善
  - ラージページによる改善
  - メモリ使用量の改善

Fortranに加えて  
C/C++の例を記載



- 「チューニング編」の例
- プロファイラ(CPU性能解析レポート)の見方

チューニングでよく使うツール。  
チューニング編を理解するには  
必要な情報

### CPU性能解析レポート：全体

FUJITSU

性能ボトルネックの抽出手段には、CPU性能解析レポートの利用を薦めます。

CPU性能解析レポートでは、以下のような豊富な種類のPA(Performance Analysis)イベントを計測でき、アプリケーションプログラム実行時のCPU動作状態を調べることができます。

The screenshot displays a comprehensive report with several key sections highlighted:

- 統計情報 (Statistics)
- メモリ・キャッシュビジー状況 (Memory/Cash Busy Status)
- キャッシュミス状況 (Cache Miss Status)
- 命令ミックス (Instruction Mix)
- 消費電力量 (Power Consumption)
- ハードウェアプリファッチ情報 (Hardware Prefetch Information)
- 性能情報 (Performance Information)
- その他の性能情報 (Other Performance Information)
- CMG間データ転送状況 (Data Transfer Status between CMGs)
- サイクルアカウンティング (Cycle Accounting)

7 再配布および不特定多数への公開禁止 Copyright 2022 FUJITSU LIMITED

### CPU性能解析レポート：サイクルアカウンティングとは

FUJITSU

サイクルアカウンティングとは、性能ボトルネックの要因分析手法です。

サイクルアカウンティング情報は、CPU性能解析レポートの右上部に掲載されています。

サイクルアカウンティングでは、あるアプリケーションプログラムを実行するためにかかった総時間（CPUサイクル数）をCPUの動作状態で分類してグラフ化します。そのグラフからCPU内のボトルネックが把握できるので、詳細な性能分析やチューニングを行うことができます。

The diagram illustrates the cycle accounting process:

- 命令コミット数 (Instruction Commit Count):** A vertical bar chart showing counts for 0, 1, 2, 3, 4, and Other instructions.
- 実行時間制約要因 (Execution Time Constraint Factors):** A list of factors such as 'メモリアクセス待ち' (Memory access wait), 'キャッシュアクセス待ち' (Cache access wait), '浮動小数点演算待ち' (Floating point calculation wait), and 'ストア待ち' (Store wait).
- Stacked Bar Chart:** A chart showing the total execution time broken down by instruction type and constraint factor. The y-axis is '実行時間 (実測)' (Execution Time (Actual)) from 0.0E+00 to 5.0E+00.
- Legend:** A color-coded key for instruction types: 0命令コミット (0 instruction commit), 1命令コミット (1 instruction commit), 2命令コミット (2 instruction commit), 3命令コミット (3 instruction commit), 4命令コミット (4 instruction commit), and その他の命令コミット (Other instruction commit).

命令コミット数 : 1マシサイクルでN命令実行した時間  
0命令コミット : 何らかの要因で命令がストールした時間

8 再配布および不特定多数への公開禁止 Copyright 2022 FUJITSU LIMITED

- 「チューニング編」の例
- ループ分割による性能向上

A64FX上でチューニングする上ではSWPL化が非常に重要であり、その促進ための方法の一つ

**C/C++ clangモード** ループ分割 (ソフトウェアパイプライン促進) (改善前) FUJITSU

演算の連鎖が長く、多くのレジスタを必要とします。そのため、SWPLなどのスケジューリングの最適化ができず、浮動小数点演算待ちが多くなっています。

```

改善前ソース
19  for (iter = 0; iter < 10000; iter++){
    <<< Loop-Information Start >>>
    <<< [OPTIMIZATION]
    <<< SIMD(VL: 8 Interleave: 1)
    <<< SPILLS:
    <<< GENERAL : SPILL 0 FILL 0
    <<< SIMD&FP : SPILL 0 FILL 8
    <<< SCALABLE : SPILL 4 FILL 6
    <<< PREDICATE : SPILL 0 FILL 0
    <<< Loop-Information End >>>
20  v for (i = 0; i < n; i++){
21      y[i] = sin(x[i]*c0)
22          +cos(x[i]*c1)
23          +atan(x[i]*c2)
24          +log(x[i]*c3);
25  }
26  }
    
```

演算連鎖が長く、多くのレジスタを使用するため、ソフトウェアパイプラインを適用できない

スケジューリングが最適化できていないため、演算待ちが大きくみえる

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	1.76E+00	2.04E+00	0.00	99.85%	0.14%	0.01%	9.95E+03	0.00	88.46%	18.51%	0.00%

※コンパイラの特性を考慮して数字関数を例にした

**C/C++ clangモード** ループ分割 (ソフトウェアパイプライン促進) (ソースチューニング) FUJITSU

ループ分割を行うことで演算の連鎖が短くなり、1ループに使用するレジスタが減少しました。その結果、SWPLなどのスケジューリングの最適化が行われ、演算待ちが改善されました。

```

改善後ソース
19  for (iter = 0; iter < 10000; iter++){
    #pragma loop_confusion
    <<< Loop-Information Start >>>
    <<< [OPTIMIZATION]
    <<< SIMD(VL: 8 Interleave: 1)
    <<< SOFTWARE PIPELINING
    <<< SPILLS:
    <<< GENERAL : SPILL 0 FILL 0
    <<< SIMD&FP : SPILL 0 FILL 0
    <<< SCALABLE : SPILL 8 FILL 17
    <<< PREDICATE : SPILL 0 FILL 0
    <<< Loop-Information End >>>
21  v for (i = 0; i < n; i++){
22      y[i] = sin(x[i]*c0);
23  }
24  v for (i = 0; i < n; i++){
25      y[i] = cos(x[i]*c1);
26  }
27  v for (i = 0; i < n; i++){
28      y[i] += atan(x[i]*c2);
29  }
30  v for (i = 0; i < n; i++){
31      y[i] += log(x[i]*c3);
32  }
33  }
    
```

ループ融合を抑制

ループ分割

1.51倍の効果

演算待ちが減少した

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D miss demand rate (%) (/L1D miss)	L1D miss hardware prefetch rate (%) (/L1D miss)	L1D miss software prefetch rate (%) (/L1D miss)	L2 miss	L2 miss rate (/Load-store instruction)	L2 miss demand rate (%) (/L2 miss)	L2 miss hardware prefetch rate (%) (/L2 miss)	L2 miss software prefetch rate (%) (/L2 miss)
改善前	0.00	1.76E+00	2.04E+00	0.00	99.85%	0.14%	0.01%	9.95E+03	0.00	88.46%	18.51%	0.00%
改善後	0.00	2.67E+00	2.39E+00	0.00	99.91%	0.10%	-0.01%					

※コンパイラの特性を考慮して数字関数を例にした

- 「チューニング編」の例
- False Sharingによるチューニング

ハマると性能劣化の影響が大きく注意が必要

### False Sharingとは

False Sharingとは、スレッド間でキャッシュラインのInvalidateおよびCopy Backが頻発する現象のことです。

**4スレッド並列と仮定した場合の例**

```

改善前ソース
1  subroutine sub(s,a,b,ni,nj)
2  real*8 a(ni,nj),b(ni,nj)
3  real*8 s(nj)
4
5  1 pp      do j = 1, nj
6  1 p      s(j)=0.0
7  2 p 8v   do i = 1, ni
8  2 p 8v   s(j)=s(j)+a(i,j)*b(i,j)
9  2 p 8v   end do
10 1 p      end do
11
12 end
    
```

初期状態: キャッシュにはキャッシュライン単位でデータが載る。各スレッドは、s(1)~s(4)を含む同一キャッシュラインを読み込む。

スレッド0 (コア1) s(1)~s(4)  
 スレッド1 (コア2) s(1)~s(4)  
 スレッド2 (コア3) s(1)~s(4)  
 スレッド3 (コア4) s(1)~s(4)

L2キャッシュ

■ スレッド0が s(1) の更新を指示

- キャッシュヒット
- スレッド0が s(1) の更新を完了
- データの一意性を保つためスレッド1~3のキャッシュラインを無効化 (Invalidate)

① キャッシュヒット  
 ② 更新  
 ③ Invalidate  
 ④ Invalidate  
 ⑤ Invalidate

L2キャッシュ

■ スレッド1が s(2) の更新を指示

- キャッシュミス
- スレッド0からスレッド1へキャッシュラインをCopy Back
- スレッド1が s(2) の更新を完了
- データの一意性を保つためスレッド0のキャッシュラインを無効化 (Invalidate)

① キャッシュミス  
 ② Copy Back  
 ③ 更新  
 ④ Invalidate  
 ⑤ Invalidate

この状態を各スレッドで繰り返すため性能低下

272 再配布および平権定多事への公認禁止 Copyright©2022 FUJITSU LIMITED

### C/C++ False Sharing (ソースチューニング)

ループ交換を行い、外側で並列化することでFalse Sharingを回避できます。その結果、L1キャッシュミス数が削減され、データアクセス待ちが改善されました。

```

改善後ソース
42  #pragma omp for
43  <<< Loop-information Start >>
44  <<< [OPTIMIZATION]
45  <<< SOFTWARE PIPELINING
46  <<< MVE: 3, POL: S)
47  <<< PREFETCH(HARD) Expect
48  <<< a, (unknown)
49  <<< Loop-information End >>>
50  p      for(j=0;j<N;j++)
51  p      {
52  p          <<< Loop-information Start >>
53  p          <<< [OPTIMIZATION]
54  p          <<< SIMD(VL: 8)
55  p          <<< FULL UNROLLING
56  p          <<< Loop-information End >>>
57  p          fv for (i=0;i<M;i++)
58  p          {
59  p              if(flag[i][j]==1)
60  p              {
61  p                  a[i][j]=b[i][j];
62  p              }
63  p          }
    
```

改善後: ループ交換を行い外側で並列化。False Sharing回避。

M=16, N=60000  
 配列宣言 double a[N][M], b[N][M];

9.43倍の効果

False Sharing回避によりL1Dミス数が削減され性能向上した

Cache	L1I miss rate (/Effective instruction)	Load-store instruction	L1D miss	L1D miss rate (/Load-store instruction)	L1D-miss demand rate (%) (/L1D miss)	L1D-miss hardware prefetch rate (%) (/L1D miss)	L2 software prefetch rate (%) (/L2 miss)	L2 miss	L2 miss rate (/Load-store instruction)	hardware demand rate (%) (/L2 miss)	hardware prefetch rate (%) (/L2 miss)	software prefetch rate (%) (/L2 miss)
改善前	0.00	4.04E+08	7.28E+08	0.18	30.45%	69.61%	0.00%	3.94E+04	0.00	48.4%	58.6%	0.00%
改善後	0.00	1.27E+08	4.81E+07	0.04	6.15%	93.8%	0.00%	1.93E+04	0.00	15.6%	89.1%	0.00%

276 再配布および平権定多事への公認禁止 Copyright©2022 FUJITSU LIMITED

# プログラミングガイドの活用例

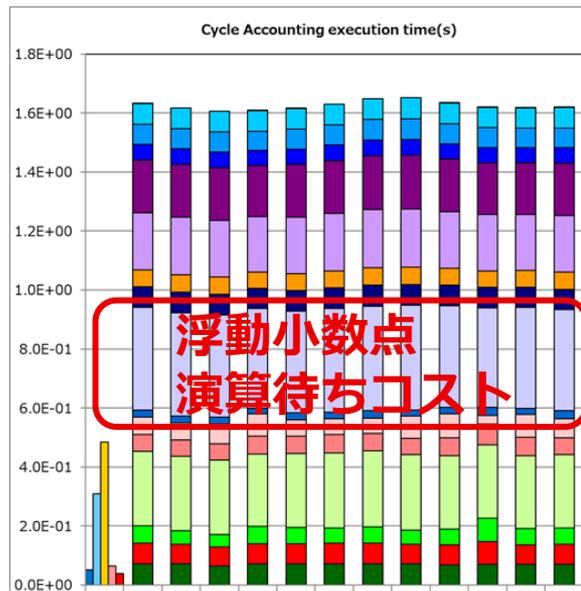
OSSアプリケーションLAMMPSへの適用

## ○ LAMMPSの概要

- 科学技術計算に用いられる著名なOSSアプリケーション (C/C++でプログラミング)
- 富士通プロファイラを用いたボトルネックの調査から以下の情報を得られた
  - NPairHalfBinNewtonOmp::buildの手続きコストが最も大きい
  - 上記手続きでは浮動小数点演算待ちに多くのコストを費やしている

Cost	%	Operation (s)	Barrier	%	Start	End	
348	100.0000	3.4779	0	0.0000	--	--	Process 9
134	38.5057	1.3392	0	0.0000	--	--	LAMMPS_NS::NPairHalfBinNewtonOmp::build(LAM...
96	27.5862	0.9594	0	0.0000	88	175	void LAMMPS_NS::PairLJCutOMP::eval<0, 0, 1>...
29	8.3333	0.2898	0	0.0000	82	160	void LAMMPS_NS::BondFENEOMP::eval<0, 0, 1>(i...
15	4.3103	0.1499	0	0.0000	77	89	LAMMPS_NS::RanMars::uniform()
10	2.8736	0.0999	0	0.0000	--	--	mca_btl_vader_component_progress
9	2.5862	0.0899	0	0.0000	--	--	LAMMPS_NS::NPair::find_special(int const*, i...
6	1.7241	0.0600	0	0.0000	348	358	LAMMPS_NS::Atom::map_find_hash(int)

富士通プロファイラ(基本プロファイラ)による手続きコスト情報



富士通プロファイラによる  
NPairHalfBinNewtonOmpの  
サイクルアカウンティング

## ○2種類のチューニングでSWPL化を促進

(※) ループ分割：チューニング編P180  
ループアンスイッチング：チューニング編P216

```
ibin = atom2bin[i];
for (k = 0; k < nstencil; k++) {
  for (i = binhead[ibin+stencil[k]]; i >= 0; i = bins[i]) {
    jtype = type[j];
    if (exclude && exclusion(i,j,itype,jtype,mask,molecule)) continue;
  }
  delx = xtmp - x[j][0];
  dely = ytmp - x[j][1];
  delz = ztmp - x[j][2];
  rsq = delx*delx + dely*dely + delz*delz;

  if (rsq <= cutneighsq[itype][jtype]) {
    if (molecular) {
      if (!moltemplate)
        which = find_special(special[i],nspecial[i],tag[j]);
      else if (imol >= 0)
        which = find_special(onemols[imol]->special[iatom],
                             onemols[imol]->nspecial[iatom],
                             tag[j]-tagprev);
      else which = 0;
      if (which == 0) neighptr[n++] = j;
      else if (domain->minimum_image_check(delx,dely,delz))
        neighptr[n++] = j;
      else if (which > 0) neighptr[n++] = j ^ (which << SBBITS);
    } else neighptr[n++] = j;
  }
}
```

チューニング前

ループ内の条件が成立した場合と成立しない場合のループに分けることで、分岐を減らしソフトウェアパイプラインを促進させる最適化

ループ分割(※)

ループアンスイッチング(※)

```
#pragma clang loop vectorize(assume_safety)
for (k = 0; k < nstencil; k++) {
  for (i = binhead[ibin+stencil[k]]; j >= 0; j = bins[j]) {
    }
}

#pragma clang loop vectorize(disable)
for (k = 0; k < j_count; k++) {
  }
}

if (molecular && !moltemplate) {
#pragma clang loop vectorize(assume_safety)
  for (k = 0; k < j_count; k++) {
    }
}
}

if (molecular && moltemplate && imol >= 0) {
  for (k = 0; k < j_count; k++) {
    }
}
}

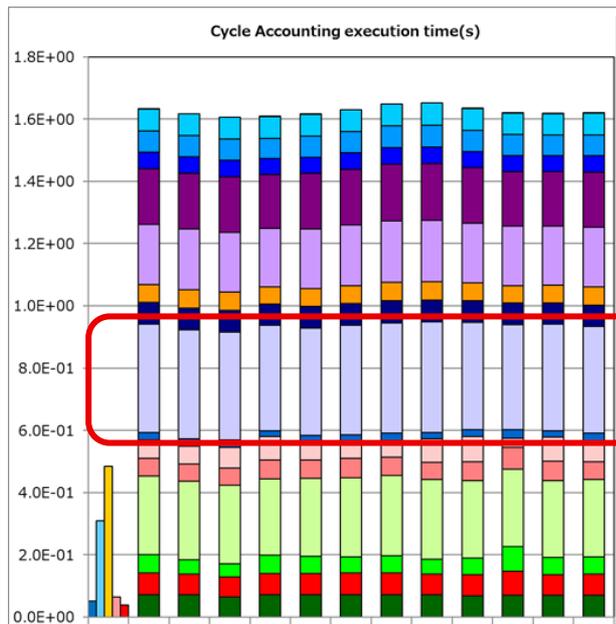
if (molecular && moltemplate && imol < 0) {
  for (k = 0; k < j_count; k++) {
    }
}
}

if (!molecular) {
  for (k = 0; k < j_count; k++) {
    }
}
}
```

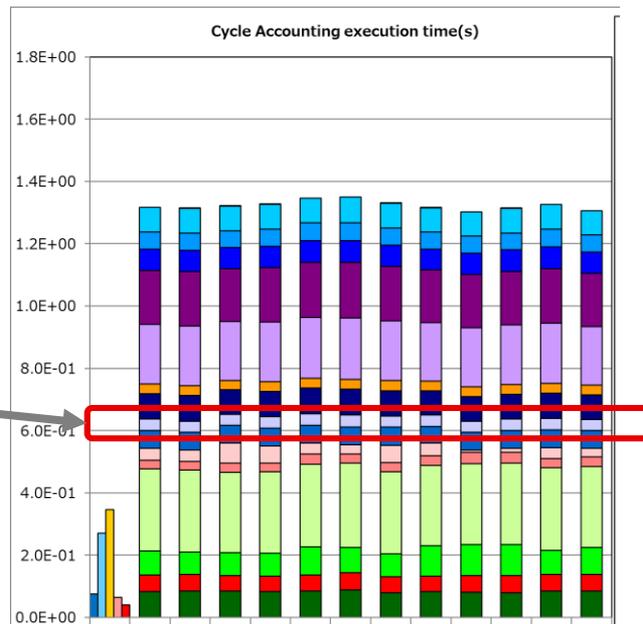
チューニング後

# LAMMPSをチューニングした効果

## ○前ページ部分のチューニング効果



浮動小数点演算  
待ちが減少



ループ分割やループアンスイッチングすることにより、ソフトウェアパイプラインが促進し、浮動小数点演算待ちが減ることによって性能向上した例

- チューニングによるLAMMPSの高速化
    - SWPL化を促進するために適用したチューニング(活用例)
      - ループ分割
      - ループアンスイッチング
    - 上記以外に適用した手法
      - ストライピング
      - ループアンローリング
      - etc
- ⇒ プログラム全体で**1.8倍**の高速化

FX1000上でのプログラミングやチューニングにご活用ください

**Thank you**

