修士 (工学) 学位論文

Master's Thesis of Engineering

Code Segment と Data Segment によって構成される Gears OS の設計

Design of Gears OS with consist of Code and Data Semgment

2016 年 3 月

March 2016

小久保 翔平

Shohei KOKUBO

琉球大学

大学院理工学研究科

情報工学専攻

Infomation Engineering Course

Graduate School of Engineering and Science

Universty of the Ryukyus

要 旨

目 次

図 目 次

表 目 次

第1章　並列分散環境下におけるプログラミング

第2章　並列プログラミングフレームワーク Cerium

Cerium は PlayStation 3(PS3) に搭載された Cell Broadband Engine(Cell) 向けの Fine-Grain TaskManager として当研究室で設計・開発されたフレームワークである。本章では Cerium の実装について説明する。

2.1　Cerium の概要

Cerium は、TaskManager, SceneGraph, Rendering Engine の 3 つの要素から構成される。Cell 用のゲームフレームワークとして開発されたが、現在では Multi-Core CPU, GPU も計算資源として利用可能な汎用計算フレームワークとなっている。

2.2　TaskManager

TaskManager は、Task と呼ばれる分割されたプログラムを管理する。サブルーチンまたは関数が Task の単位となる。TaskManager が提供する API を表:2.1 に示す。

create_task	Task の生成
allocate	環境のアライメントに考慮した allocator
set_inData	Task への入力データのアドレスを追加
set_outData	Task からのデータ出力先アドレスを追加
set_param	Task のパラメータ (32 bits)
wait_for	Task の依存関係を設定
set_cpu	Task を実行する Device の設定
spawn	Task を Queue に登録
iterate	データ並列で実行する Task として Queue に登録

表 2.1: TaskManager API

TaskManager は ActiveTaskList と WaitTaskList の 2 種類の Queue を持つ。依存関係を解決する必要がある Task は WaitTaskList に入れられる。TaskManger によって依存関係が解決されると ActiveTaskList に移され、実行可能な状態となる。実行可能な状態となった Task は set_cpu で指定された Device に対応した Scheduler に転送し実行される。図:2.1 は Cerium が Task を生成/実行する場合のクラスの構成である。

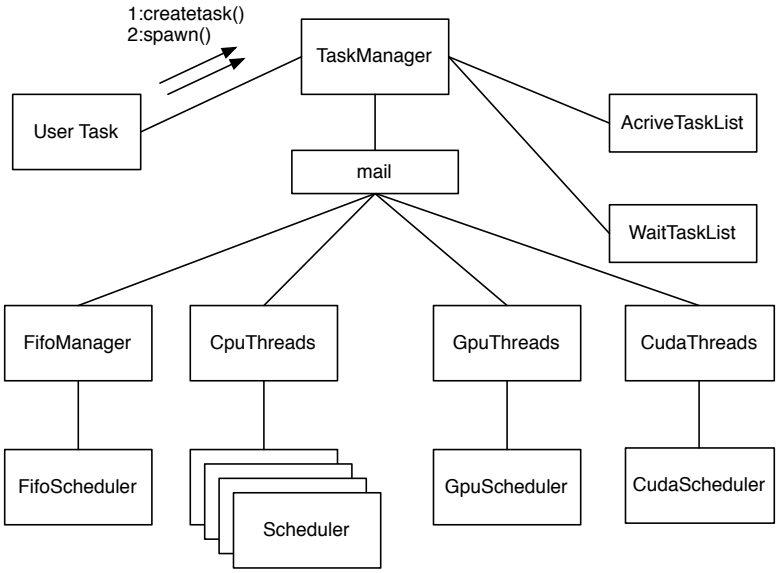

図 2.1: TaskManager

2.3　Cerium における Task

Task は TaskManager の API を利用して生成する。生成された Task には以下の要素を設定することができる。

- input data
  set_inData を用いて設定する Task が実行する処理に必要なデータの入力元となるアドレス。関数を呼び出す際の引数に相当する。汎用ポインタ (void* 型) なので Task 側で適切なキャストを行う必要がある。

- output data
  set_outData を用いて設定する Task が処理したデータの出力先となるアドレス。関数の戻り値に相当する。

- parameter
  set_param を用いて設定するデータの処理に必要な実数値 (index 等)。

- cpu type
  set_cpu を用いて設定する Task が実行される Device の組み合わせ。Cell, Multi-Core CPU, GPU またはこれらの組み合わせを指定することができる。

- dependency
  wait_for を用いて設定する他の Task との依存関係。依存関係が解決された Task は実行可能な状態となる。

2.4　Task のパイプライン実行

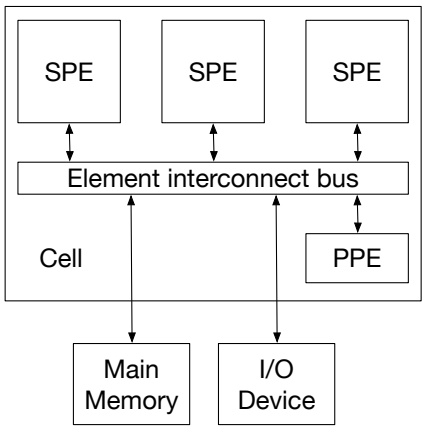

図 2.2: Cell Architecture

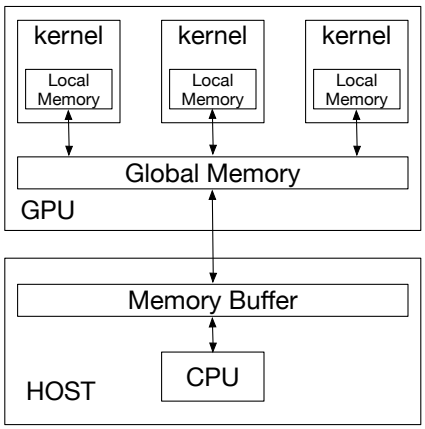

図 2.3: GPU Architecture

Cell(図:2.2) や GPU(図:2.3) のように異なるメモリ空間を持つ Device を計算資源として利用するにはデータの転送が必要になる。このデータ転送がボトルネックとなり、並列度が低下してしまう。転送処理をオーバーラップし、並列度を維持するために Cerium では Task のパイプライン実行をサポートしている。

TaskManager である程度の Task をまとめた TaskList を生成し、実行する Device に対応した Scheduler に転送する。受け取った TaskList に沿ってパイプラインを組み Task を実行していく。TaskList でまとめられている Task は依存関係が解決されているので自由にパイプラインを組むことが可能である。実行完了は TaskList 毎ではなく、Task 毎に通知される。図:2.4 は TaskList を受け取り、Task をパイプラインで処理していく様子である。

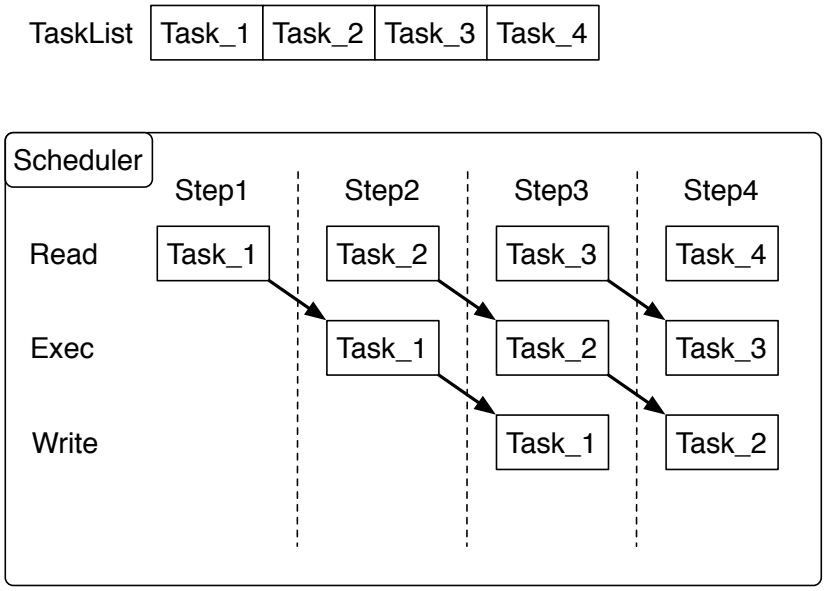

図 2.4: Scheduler

第3章　CbC

第4章　Gears OS

Cerium と Alice の開発を通して得られた知見から並列分散処理には Code の分割だけではなく Data の分割も必要であることがわかった。当研究室で開発している Code Segment を基本的な処理単位とするプログラミング言語 Continuation based C(CbC) を用いて Data Segment を定義し、Gears OS の設計と基本的な機能の実装を行なった。

本章では Gears OS の設計と実装した基本的な機能について説明する。

4.1　Code Gear と Data Gear

Gears OS ではプログラムの単位として Gear を用いる。Gear は並列実行の単位、データの分割、Gear 間の接続等になる。

Code Gear はプログラムの処理そのものになる。これは OpenCL/CUDA の kernel, Cerium の Task に相当する。Code Gear は任意の数の Data Gear を参照し、処理が完了すると 任意の数の Data Gear に書き込む。Code Gear は接続された Data Gear 以外にアクセスできない。Code Gear から次の Code Gear への処理の移動は goto の後に Code Gear の名前と引数を指定することで実現できる。Code Gear は Code Segment そのものである。

Data Gear はデータそのものを表す。int や文字列などの Primitive Data Type を持っている。

Gear の特徴として処理やデータの構造が Code Gear, Data Gear に閉じていることにある。これにより実行時間、メモリ使用量などを予測可能なものにすることが可能になる。

4.2　Gears OS の構成

Gears OS は以下の要素で構成される。

- Context
  接続可能な Code/Data Gear のリスト、TaskQueue へのポインタ、Persistent Data Tree へのポインタ、Temporal Data Gear のためのメモリ空間等を持っており、Context を通してアクセスすることができる。メインとなる Context と Worker 用の Context があり、TaskQueue と Persistent Data Tree は共有される。Temporal Data Gear のためのメモリ空間は Context 毎に異なり、互いに干渉することはできない。Persistent Data Tree への書き込みのみで相互作用を発生させ目的の処理を達成する。

- TaskQueue
  ActiveTaskQueue と WaitTaskQueue の 2 つの TaskQueue を持つ。先頭と末尾の Element へのポインタを持つ Queue を表す Data Gear である。Element は Task を表す Data Gear へのポインタと次の Element へのポインタを持っている。Compare and Swap(CAS) を使ってアクセスすることでスレッドセーフな Queue として利用することが可能になる。

- TaskManager
  Task には Input Data Gear, Output Data Gear が存在する。Input/Output Data Gear から依存関係を決定し、TaskManager が解決する。依存関係が解決された Task は WaitTaskQueue から ActiveTaskQueue に移される。TaskManager はメインとなる Context を参照する。

- Persistent Data Tree
  非破壊木構造で構成された Lock-free なデータストアである。Red-Black Tree として構成することで最悪な場合の挿入・削除・検索の計算量を保証する。

- Worker
  TaskQueue から Task の取得・実行を行う。Task の処理に必要なデータは Persistent Data Tree から取得する。処理後、必要なデータを Persistent Data Tree に書き出して再び Task の取得・実行を行う。

図:4.1 は Gears OS の構成図である。

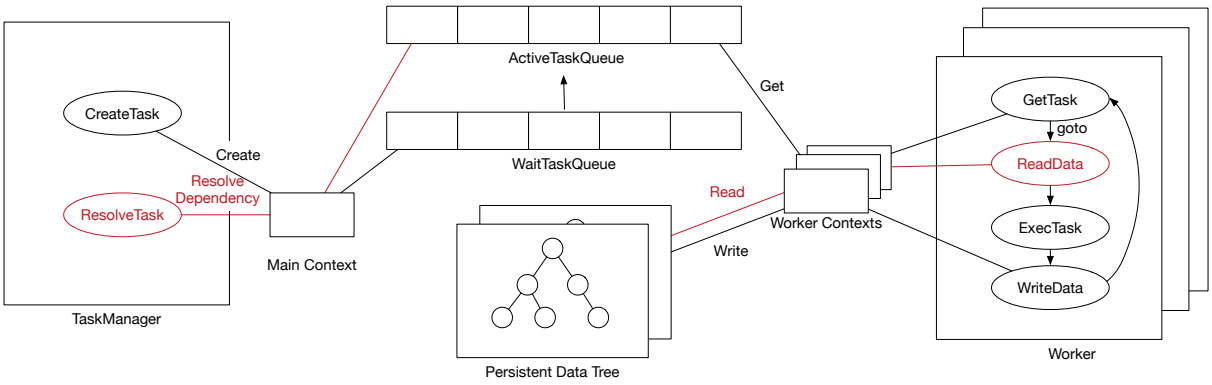

図 4.1: Gears OS

4.3 Allocator

Gears OS では Context の生成時にある程度の大きさのメモリ領域を確保する。Context には確保したメモリ領域を指す情報が格納される。このメモリ領域を利用して Task の実行に必要な Data Gear を生成する。

Context の定義と生成はソースコード:4.1, ソースコード:4.2 の通りである。

```
1  /* Context definition example */
2  #define ALLOCATE_SIZE 1000
3
4  // Code Gear Name
5  enum Code {
6      Code1,
7      Code2,
8      Allocator,
9      Exit,
10 };
11
12 // Unique Data Gear
13 enum UniqueData {
14     Allocate,
15 };
16
17 struct Context {
18     enum Code next;
19     int codeNum;
20     __code (**code) (struct Context*);
21     void* heapStart;
22     void* heap;
23     long heapLimit;
24     int dataNum;
25     union Data **data;
26 };
27
28 // Data Gear definition
29 union Data {
30     // size: 4 byte
31     struct Data1 {
32         int i;
33     } data1;
34     // size: 5 byte
35     struct Data2 {
36         int i;
37         char c;
38     } data2;
39     // size: 8 byte
40     struct Allocate {
41         long size;
42     } allocate;
43 };
```

ソースコード 4.1: Context

```
1  #include <stdlib.h>
2
3  #include "context.h"
4
5  extern __code code1_stub(struct Context*);
6  extern __code code2_stub(struct Context*);
7  extern __code allocator_stub(struct Context*);
8  extern __code exit_code(struct Context*);
9
```

```
10 __code initContext(struct Context* context, int num) {
11     context->heapLimit = sizeof(union Data)*ALLOCATE_SIZE;
12     context->heapStart = malloc(context->heapLimit);
13     context->heap = context->heapStart;
14     context->codeNum = Exit;
15
16     context->code = malloc(sizeof(__code*)*ALLOCATE_SIZE);
17     context->data = malloc(sizeof(union Data*)*ALLOCATE_SIZE);
18
19     context->code[Code1] = code1_stub;
20     context->code[Code2] = code2_stub;
21     context->code[Allocator] = allocator_stub;
22     context->code[Exit] = exit_code;
23
24     context->data[Allocate] = context->heap;
25     context->heap += sizeof(struct Allocate);
26
27     context->dataNum = Allocate;
28 }
```

ソースコード 4.2: initContext

Context はヒープサイズを示す heapLimit, ヒープの初期位置を示す heapStart, ヒープの現在位置を示す heap を持っている。必要な Data Gear のサイズに応じて heap の位置を動かすことで Allocation を実現する。

allocate を行うには allocate に必要な Data Gear に情報を書き込む必要がある。この Data Gear は Context 生成時に生成する必要があり、ソースコード:4.1 14 行目の Allocate がそれに当たる。UniqueData で定義した Data Gear は Context と同時に生成される。

Temporal Data Gear にある Data Gear は基本的には破棄可能なものなので heapLimit を超えたら heap を heapStart の位置に戻し、ヒープ領域を再利用する (図:4.2)。必要な Data Gear は Persistent Data Tree に書き出すことで他の Worker からアクセスすることが可能になる。

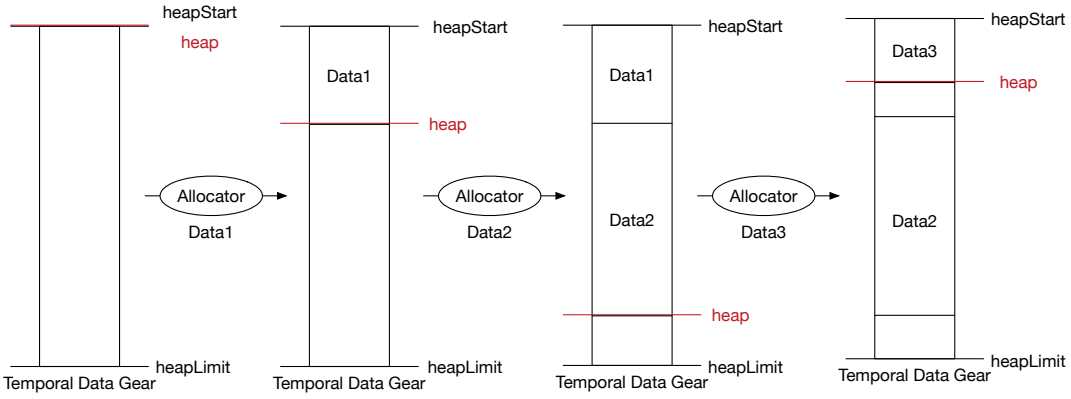

図 4.2: Allocation

実際に allocate を行う Code Gear はソースコード:4.3 の通りである。

Context 生成時に実行可能な Code Gear と名前が対応付けられる。その対応付けられた Code Gear が Context の code に格納される。この code を介して遷移先の Code Gear

を決定する。

Code Gear には Context が接続されるが Context を介して Data Gear にアクセスすることはない。stub を介して間接的に必要な Data Gear にアクセスする。

```
// Code Gear
__code start_code(struct Context* context) {
    // start processing
    goto meta(context, context->next);
}

// Meta Code Gear
__code meta(struct Context* context, enum Code next) {
    // meta computation
    goto (context->code[next])(context);
}

// Code Gear
__code code1(struct Context* context, struct Allocate* allocate) {
    allocate->size = sizeof(struct Data1);
    context->next = Code2;

    goto meta(context, Allocator);
}

// Meta Code Gear(stub)
__code code1_stub(struct Context* context) {
    goto code1(context, &context->data[Allocate]->allocate);
}

// Meta Code Gear
__code allocator(struct Context* context, struct Allocate* allocate) {
    context->data[++context->dataNum] = context->heap;
    context->heap += allocate->size;

    goto meta(context, context->next);
}

// Meta Code Gear(stub)
__code allocator_stub(struct Context* context) {
    goto allocator(context, &context->data[Allocate]->allcate);
}

// Code Gear
__code code2(struct Context* context, struct Data1* data1) {
    // processing
}

// Meta Code Gear(stub)
__code code2_stub(struct Context* context) {
    goto code2(context, &context->data[context->dataNum]->data1);
}
```

ソースコード 4.3: allocate

4.4 Synchronized Queue

Gears OS における Synchronized Queue は TaskQueue として利用される。メインとなる Context と Worker 用の Context で共有され、Woker が TaskQueue から Task を取得し実行することで並列処理を実現する。

Gears OS での Queue を Queue を表す Data Gear と Queue の構成要素である Element によって表現する。Queue を表す Data Gear には先頭の Element を指す first, 末尾の Element を指す last, Element の個数を示す count が格納される。Element を表す Data Gear には Task を示す task, 次の Element を示す next が格納される。

ソースコード:4.4 は Context の定義 (ソースコード:4.1) に追加する Queue と Element の定義である。

```
1  // Code Gear Name
2  enum Code {
3      PutQueue,
4      GetQueue,
5  };
6
7  // Unique Data Gear
8  enum UniqueData {
9      Queue,
10     Element,
11 };
12
13 // Queue definication
14 union Data {
15     // size: 20 byte
16     struct Queue {
17         struct Element* first;
18         struct Element* last;
19         int count;
20     } queue;
21     // size: 16 byte
22     struct Element {
23         struct Task* task;
24         struct Element* next;
25     } element;
26 }
```

ソースコード 4.4: queue

新たに Queue に対する操作を行う Code Gear の名前を追加し、UniqueData には Queue の情報が入る Queue(ソースコード:4.4 9 行目) と Enqueue に必要な情報を書き込む Element(ソースコード:4.4 10 行目) を定義している。

通常の Enqueue, Dequeue を行う Code Gear はソースコード:4.5 と ソースコード:4.6 の通りである。

```
1  // allocate Element
2  __code putQueue1(struct Context* context, struct Allocate* allocate) {
3      allocate->size = sizeof(struct Element);
4      allocator(context);
5
6      goto meta(context, PutQueue2);
7  }
8
9  // Meta Code Gear(stub)
```

```
10 __code putQueue1_stub(struct Context* context) {
11     goto putQueue1(context, &context->data[Allocate]->allocate);
12 }
13
14 // write Element infomation
15 __code putQueue2(struct Context* context, struct Element* new_element, struct
       Element* element, struct Queue* queue) {
16     new_element->task = element->task;
17
18     if (queue->first)
19         goto meta(context, PutQueue3);
20     else
21         goto meta(context, PutQueue4);
22 }
23
24 // Meta Code Gear(stub)
25 __code putQueue2_stub(struct Context* context) {
26     goto putQueue2(context,
27                    &context->data[context->dataNum]->element,
28                    &context->data[Element]->element,
29                    &context->data[ActiveQueue]->queue);
30 }
31
32 // Enqueue(normal)
33 __code putQueue3(struct Context* context, struct Queue* queue, struct Element*
       new_element) {
34     struct Element* last = queue->last;
35     last->next = new_element;
36
37     queue->last = new_element;
38     queue->count++;
39
40     goto meta(context, context->next);
41 }
42
43 // Meta Code Gear(stub)
44 __code putQueue3_stub(struct Context* context) {
45     goto putQueue3(context,
46                    &context->data[ActiveQueue]->queue,
47                    &context->data[context->dataNum]->element);
48 }
49
50 // Enqueue(nothing element)
51 __code putQueue4(struct Context* context, struct Queue* queue, struct Element*
       new_element) {
52     queue->first = new_element;
53     queue->last = new_element;
54     queue->count++;
55
56     goto meta(context, context->next);
57 }
58
59 // Meta Code Gear(stub)
60 __code putQueue4_stub(struct Context* context) {
61     goto putQueue4(context,
62                    &context->data[ActiveQueue]->queue,
63                    &context->data[context->dataNum]->element);
64 }
```

ソースコード 4.5: Enqueue

```
1 // Dequeue
2 __code getQueue(struct Context* context, struct Queue* queue, struct Node* node) {
3     if (queue->first == 0)
4         return;
```

```
5
6     struct Element* first = queue->first;
7     queue->first = first->next;
8     queue->count--;
9
10    context->next = GetQueue;
11    stack_push(context->code_stack, &context->next);
12
13    context->next = first->task->code;
14    node->key = first->task->key;
15
16    goto meta(context, GetTree);
17 }
18
19 // Meta Code Gear(stub)
20 __code getQueue_stub(struct Context* context) {
21    goto getQueue(context,
22                  &context->data[ActiveQueue]->queue,
23                  &context->data[Node]->node);
24 }
```

ソースコード 4.6: Dequeue

ソースコード:4.5 とソースコード:4.6 はシングルスレッドでは正常に動作するが、並列実行すると期待した値にならない。

4.5 Red-Black Tree

第5章　比較

5.1　Cerium

5.2　従来の OS

第6章　評価

6.1　Twice

第7章　結論

謝辞

参考文献

[1] 宮國渡, 河野真治, 神里晃, 杉山千秋. Cell 用の fine-grain task manager の実装. 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS), April 2008.

[2] 赤嶺一樹, 河野真治. Datasegment api を用いた分散フレームワークの設計. 日本ソフトウェア科学会第 28 回大会論文集, Sep 2011.

[3] Sony Corporation. Cell broadband engine architecture, 2005.

[4] 河野真治, 杉本優. Code segment と data segment によるプログラミング手法. 第 54 回プログラミング・シンポジウム, Jan 2013.

[5] 河野真治, 島袋仁. C with continuation と、その playstation への応用. 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS), May 2000.

[6] 徳森海斗, 河野真治. Continuation based c の llvm/clang 3.5 上の実装について. 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS), May 2014.

[7] Eugenio Moggi. Computational lambda-calculus and monads. *Proceedings of the Fourth Annual Symposium on Logic in computer science*, 1989.

[8] 下地篤樹, 河野真治. 線形時相論理による continuation based c プログラムの検証. 情報処理学会システムソフトウェアとオペレーティング・システム研究会 (OS), April 2007.

[9] Aaftab Munshi, Khronos OpenCL Working Group. *The OpenCL Specification Version 1.0*, 2007.

[10] CUDA. https://developer.nvidia.com/category/zone/cuda-zone/.

[11] Messagepack. http://msgpack.org/.