2007年12月　Tokyo

Think Parallel

並列に考えましょう

ジェイムス･レインダース
ディレクター＆エバンジェリスト
インテル® ソフトウェア開発製品部

Because of multi-core processors, concurrency is now a critical aspect of software design.

マルチコア・プロセッサーの浸透により、並列性はソフトウェア設計における重要な要素となっています

What does this mean?

これは何を意味するでしょう？

同時性（並列性）

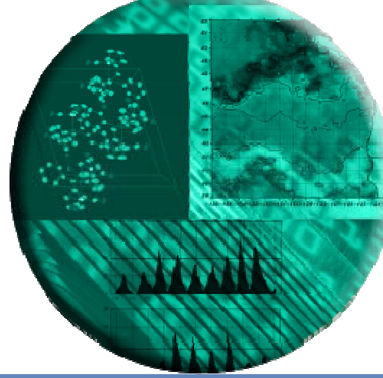
- Hardware offers concurrency –
  how can software use it?
- ハードウェアは同時処理性を提供します　ー
  ソフトウェアはそれをどのように活用できるでしょう？
- What is needed to write concurrent software?
- 並列ソフトウェアを記述するには何が必要でしょう？

ハードウェアは同時処理性を提供 –
ソフトウェアはそれをどう活用するか？

認識

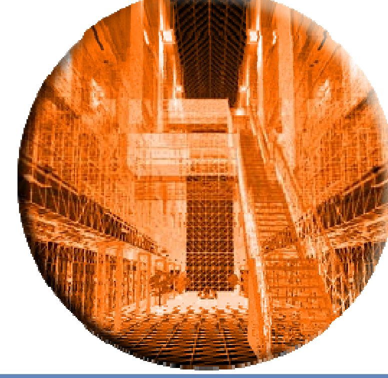
Feature Extraction
Identification
Cancer Cell Detection

検索



Financial Analytics
Media Search &
Retrieval

解析

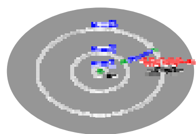


Visualization
Ray Tracing
Physical Simulation

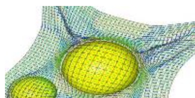
Better *modeling* internal to programs, leads to better human interaction.

優れたプログラムのモデリングはより良い人との相互関係を先導する

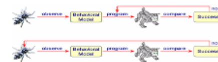
モデルベースのコンピューティング

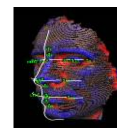

Atomic Model

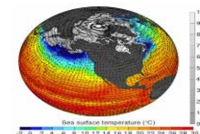
Body Model


Cloth Model

Financial Model


Behavioral Model


Facial Model


Climate Model

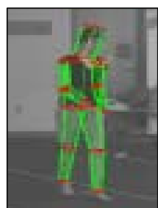
Mathematical rules and variables that approximate reality
Allow computers to recognize, manipulate, and represent ideas

現実に近い数学の規則や数値は、コンピューターが認識、操作そして表現することを可能にする

Example: Modeling body motion…

例：ボディーモーションのモデリング…

Raw イメージ


イメージの解析

```
Type Body {
 Vector arm
 Vector leg
 Move(arm)
 Move(leg)
```

モデルの明示化

表示表現

モデル化は並列性を活用するための重要な傾向

Examples:

- Gesture recognition – Is that a command?
- Image enhancement – Is that noise?
- Character recognition – Is that the target character?

例:

- ジェスチャー認識 – それは命令ですか？
- イメージ拡大 – それはノイズですか？
- 文字認識 – それは目標とする文字ですか？

同時性（並列性）

- Hardware offers concurrency –
  how can software use it?
- What is needed to write concurrent software?

- ハードウェアは並列性を提供する –
  ソフトウェアはそれをどのように利用できるか？
- 並列ソフトウェアを記述するために必要なことは？

My eight "Tips" for concurrent software development

並列ソフトウェアを開発するための8つのヒント

ヒント #1: Think Parallel

Approach all problems looking for the parallelism.

すべての処理で並列性の可能性を探してください

Organize your thinking to express the parallelism.

あなたの思考を体系づけて並列性を表現してください

Think parallel *first* (not an afterthought).

最初に並列性について考えてください（後で考えてはだめです）

ヒント #2… 抽象化を活用…

Why?

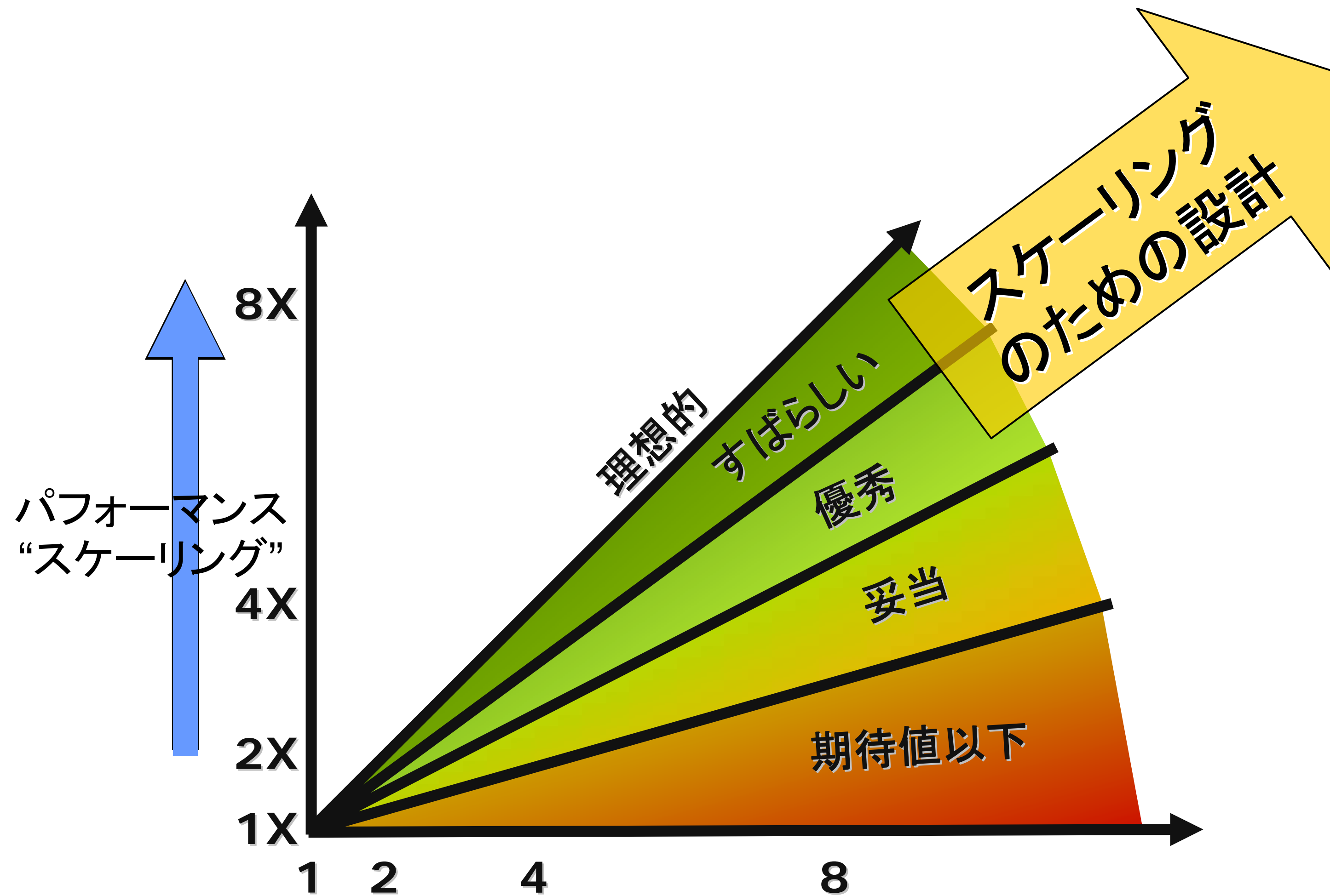
なぜか？

並列化における3つの課題

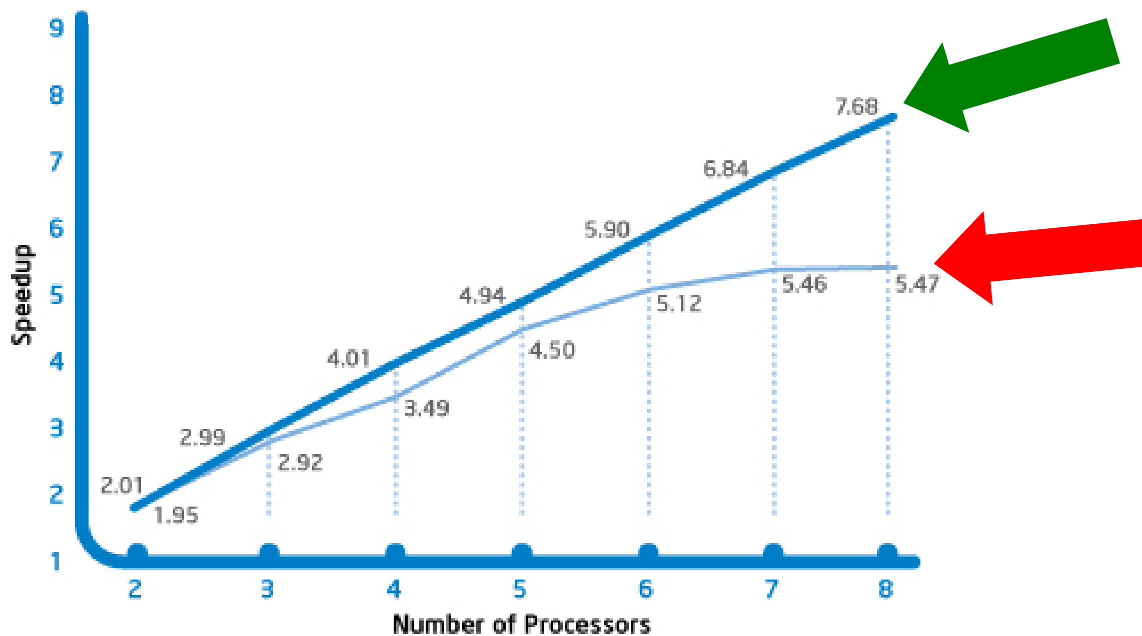
- Scalability
- Correctness
- Maintainability

- スケーラビリティー
- 正当性
- 保守性

スケーラビリティー



例: TBB 抽象化によるスケーリング vs. “ハンドコード”
(3D レイトレーシング・アプリケーション)



“Higher level” programming model *must* aim to yield better scaling.

Increasingly important as we move to heterogeneous multi-core processors.

“高度な” プログラミング・モデルではスケーリングを実現するには計画性がもとめられる

私たちがヘテロジニアスなマルチコア・プロセッサーへ移行するとさらに重要となる

intel

並列化における3つの課題

- Scalability
- Correctness
- Maintainability

- スケーラビリティー
- 正当性
- 保守性

並列プログラミングにおける特殊な“バグ”

- レースコンディション（競合）

- デッドロック

“Higher level” programming model *must* aim to yield better scaling.

“高度な” プログラミング・モデルではスケーリングを実現するには計画性がもとめられる

Increasingly important as we move to heterogeneous multi-core processors.

私たちがヘテロジニアスなマルチコア・プロセッサーへ移行するとさらに重要となる

intel

並列化における3つの課題

- Scalability
- Correctness
- Maintainability

- スケーラビリティー
- 正当性
- 保守性

保守性：抽象化の活用

"ハンドコード"

This example includes software developed by John E. Stone.

Thread Setup and Initialization

```
CRITICAL_SECTION MyMutex, MyMutex2, MyMutex3;
int get_num_cpus (void) {
  SYSTEM_INFO si;
  GetSystemInfo(&si);
  return (int)si.dwNumberOfProcessors;}
int nthreads = get_num_cpus ();
HANDLE *threads = (HANDLE *) alloca (nthreads * sizeof (HANDLE));
InitializeCriticalSection (&MyMutex);
InitializeCriticalSection (&MyMutex2);
InitializeCriticalSection (&MyMutex3);
for (int i = 0; i < nthreads; i++) {
  DWORD id;
  &threads[i] = CreateThread (NULL, 0, parallel_thread, i, 0, &id);}
for (int i = 0; i < nthreads; i++) {
  WaitForSingleObject (&threads[i], INFINITE);
}
```

Parallel Task Scheduling and Execution

```
const int MINPATCH = 150;
const int DIVFACTOR = 2;
typedef struct work_queue_entry_s {
  patch pch;
  struct work_queue_entry_s *next;
} work_queue_entry_t;
work_queue_entry_t *work_queue_head = NULL;
work_queue_entry_t *work_queue_tail = NULL;
void generate_work (patch* pchin)
{  int startx, stopx, starty, stopy;
  int xs,ys;
  startx=pchin->startx; stopx= pchin->stopx;
  starty=pchin->starty; stopy= pchin->stopy;
  if(((stopx-startx) >= MINPATCH) || ((stopy-starty) >= MINPATCH)) {
    int xpatchsize = (stopx-startx)/DIVFACTOR + 1;
    int ypatchsize = (stopy-starty)/DIVFACTOR + 1;
    for (ys=starty; ys<=stopy; ys+=ypatchsize)
    for (xs=startx; xs<=stopx; xs+=xpatchsize) {
      patch pch;
      pch.startx = xs;
      pch.starty = ys;
      pch.stopx = MIN(xs+xpatchsize-1,stopx);
      pch.stopy = MIN(ys+ypatchsize-1,stopy);
      generate_work (&pch);}
} else {
    /* just trace this patch */
    work_queue_entry_t *q = (work_queue_entry_t *) malloc (sizeof
(work_queue_entry_t));
    q->pch.starty = starty; q->pch.stopy = stopy;
    q->pch.startx = startx; q->pch.stopx = stopx;
    q->next = NULL;
    if (work_queue_head == NULL) {
      work_queue_head = q;
    } else {
      work_queue_tail->next = q;
    }
    work_queue_tail = q;
  }
}
void generate_worklist (void)
{
  patch pch;
  pch.startx = startx;
  pch.stopx = stopx;
  pch.starty = starty;
  pch.stopy = stopy;
  generate_work (&pch);
}
bool schedule_thread_work (patch &pch)
{
  EnterCriticalSection (&MyMutex3);
  work_queue_entry_t *q = work_queue_head;
  if (q != NULL) {
    pch = q->pch;
    work_queue_head = work_queue_head->next;
  }
  LeaveCriticalSection (&MyMutex3);
  return (q != NULL);
}
generate_worklist ();

void parallel_thread (void *arg)
{
  patch pch;
  while (schedule_thread_work (pch)) {
    for (int y = pch.starty; y <= pch.stopy; y++) {
      for (int x=pch.startx; x<=pch.stopx; x++) {
        render_one_pixel (x, y);}}
    if (scene.displaymode == RT_DISPLAY_ENABLED) {
      EnterCriticalSection (&MyMutex3);
      for (int y = pch.starty; y <= pch.stopy; y++) {
        GraphicsDrawRow(pch.startx-1, y-1, pch.stopx-pch.startx+1,
(unsigned char *) &global_buffer[((y-starty)*totalx+(pch.startx-startx))*3]);
      }
      LeaveCriticalSection (&MyMutex3);
    }
  }
}
```

抽象化の活用 (TBB)

Thread Setup and Initialization

```
#include "tbb/task_scheduler_init.h"
#include "tbb/spin_mutex.h"
tbb::task_scheduler_init init;
tbb::spin_mutex MyMutex, MyMutex2;
```

Parallel Task Scheduling and Execution

```
#include "tbb/parallel_for.h"
#include "tbb/blocked_range2d.h"
class parallel_task {
public:
  void operator() (const tbb::blocked_range2d<int> &r) const {
    for (int y = r.rows().begin(); y != r.rows().end(); ++y) {
      for (int x = r.cols().begin(); x != r.cols().end(); x++) {
        render_one_pixel (x, y);
      }
    }
    if (scene.displaymode == RT_DISPLAY_ENABLED) {
      tbb::spin_mutex::scoped_lock lock (MyMutex2);
      for (int y = r.rows().begin(); y != r.rows().end(); ++y) {
        GraphicsDrawRow(startx-1, y-1, totalx, (unsigned char
*) &global_buffer[(y-starty)*totalx*3]);
      }
    }
  }
  parallel_task () {}
};
parallel_for (tbb::blocked_range2d<int> (starty, stopy + 1,
grain_size, startx, stopx + 1, grain_size), parallel_task ());
```

保守性：抽象化の活用

“hand-coded”

```
Thread Setup and Initialization
CRITICAL_SECTION MyMutex, MyMutex2, MyMutex3;
int get_num_cpus (void) {
    SYSTEM_INFO si;
    GetSystemInfo(&si);
    return (int)si.dwNumberOfProcessors;}
int nthreads = get_num_cpus ();
HANDLE *threads = (HANDLE *) alloca (nthreads * sizeof (HANDLE));
InitializeCriticalSection (&MyMutex);
InitializeCriticalSection (&MyMutex2);
InitializeCriticalSection (&MyMutex3);
for (int i = 0; i < nthreads; i++) {
    DWORD id;
    &threads[i] = CreateThread (NULL, 0, parallel_thread, i, 0, &id);}
for (int i = 0; i < nthreads; i++) {
    WaitForSingleObject (&threads[i], INFINITE);
}

Parallel Task Scheduling and Execution
const int MINPATCH = 150;
const int DIVFACTOR = 2;
typedef struct work_queue_entry_s {
    patch pch;
    struct work_queue_entry_s *next;
} work_queue_entry_t;
work_queue_entry_t *work_queue_head = NULL;
work_queue_entry_t *work_queue_tail = NULL;
void generate_work (patch* pchin)
{  int startx, stopx, starty, stopy;
   int xs,ys;
```

```
}
void generate_worklist (void)
{
    patch pch;
    pch.startx = startx;
    pch.stopx = stopx;
    pch.starty = starty;
    pch.stopy = stopy;
    generate_work (&pch);
}
bool schedule_thread_work (patch &pch)
{
    EnterCriticalSection (&MyMutex3);
    work_queue_entry_t *q = work_queue_head;
    if (q != NULL) {
        pch = q->pch;
        work_queue_head = work_queue_head->next;
    }
    LeaveCriticalSection (&MyMutex3);
    return (q != NULL);
}
generate_worklist ();
```

```
(pch)) {
= pch.stopy; y++) {
=pch.stopx; x++) {
y);}}
RT_DISPLAY_ENABLED) {
yMutex3);
 <= pch.stopy; y++) {
.startx-1, y-1, pch.stopx-pch.startx+1,
[((y-starty)*totalx+(pch.startx-startx))*3]);
yMutex3);
```

Using abstraction (TBB)

```
d Initialization
_scheduler_init.h"
_mutex.h"
er_init init;
MyMutex, MyMutex2;

Parallel Task Scheduling and Execution
#include "tbb/parallel_for.h"
#include "tbb/blocked_range2d.h"
class parallel_task {
public:
    void operator() (const tbb::blocked_range2d<int> &r) const {
        for (int y = r.rows().begin(); y != r.rows().end(); ++y) {
            for (int x = r.cols().begin(); x != r.cols().end(); x++) {
                render_one_pixel (x, y);
            }
        }
        if (scene.displaymode == RT_DISPLAY_ENABLED) {
            tbb::spin_mutex::scoped_lock lock (MyMutex2);
            for (int y = r.rows().begin(); y != r.rows().end(); ++y) {
                GraphicsDrawRow(startx-1, y-1, totalx, (unsigned char
*) &global_buffer[(y-starty)*totalx*3]);
            }
        }
    }
    parallel_task () {}
};
parallel_for (tbb::blocked_range2d<int> (starty, stopy + 1,
grain_size, startx, stopx + 1, grain_size), parallel_task ());
```

TBB, OpenMP, ライブラリーには多くの抽象化の選択肢がある

“高度な” プログラミング・モデルではスケーリングを実現するには計画性がもとめられる
私たちがヘテロジニアスなマルチコア・プロセッサーへ移行するとさらに重要となる

ヒント #2: 抽象化を使用してプログラム

Helps with the three key challenges:

- Scalability
- Correctness
- Maintainability

3つの主要な課題の解決を助ける:

- スケーラビリティー
- 正当性
- 保守性

ヒント #3: スレッド(核心)ではなく、タスク(雑用)に注目してプログラミング

Mapping tasks to threads should be done by some form of abstraction.

タスクのスレッドへの割り当てはいくつかの抽象化の法則にって行われるべき

Program *what* to do, not *how*.

どう制御するかではなく何をするかに注目してプログラミング

Increasingly important as we move to heterogeneous multi-core processors.

私たちがヘテロジニアスなマルチコア・プロセッサーへ移行するとさらに重要となる

ヒント #4: 並列性をオフにして設計してみてください

To make debugging simpler, create programs that can run **without** concurrency.

デバッグを簡単にするため、並列性を無効にしシリアル処理ができるようプログラムを設計する

First, run with CONCURRENCY OFF – and debug like you have always debugged.

最初に、並列処理オフで実行、そしていつものようにデバッグを行う

Then, run with CONCURRENCY ON and debug the "special" concurrency issues.

次に、並列処理をオンで実行、そして並列性に関する"特殊な"問題をデバッグ

ヒント #5: いつロックをするかを学ぶ、
そしてどのように多用せず最小限に抑えるか

Learning to keep locks under control – is an expertise to build up over years of experience.

ロックを完全に制御するのを学ぶのは、経験の年を経て確立できる専門的技術

Transactional memory – may be very important but will NOT eliminate this tip.

トランザクショナル・メモリー – 非常に重要だが、このヒントでは排除できない

ヒント #6: 並列性を補助するよう設計されたツールやライブラリーを使用する

Don't "tough it out" with old tools.

古いツールを使って“苦境に耐える”ことをしない”

This is critical to effective software development.

効果的なソフトウェア開発には不可欠

ヒント #7: スケーラブルなメモリー・アロケーターを使用

Why?

- reduce needless contention
- optimize for concurrency
- avoid pitfalls like false-sharing or just good old cache thrash

なぜか？

- 必要のないコンテンションを減らす
- 並列性のために最適化
- フォルス・シェアリングやキャッシュ・スラッシングなどの落とし穴に落ちいらないようにする

Many solutions exist – including a very good one available open source from Intel
http://threadingbuildingblocks.org

多くのソリューションがある – インテルからオープンソース化された非常に優れたひとつ
http://threadingbuildingblocks.org

ヒント #8: ワークロードを増やしてスケールするように設計

The amount of work your program needs to handle increases over time.

Plan for that.

あなたのプログラムが扱う仕事量は時間と共に増加します

計画を立ててください

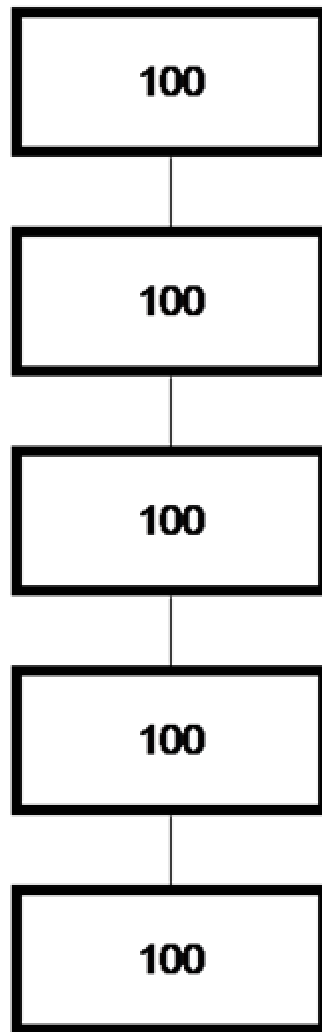
Designed with scaling in mind, a program will handle more work as the number of processor cores increase.

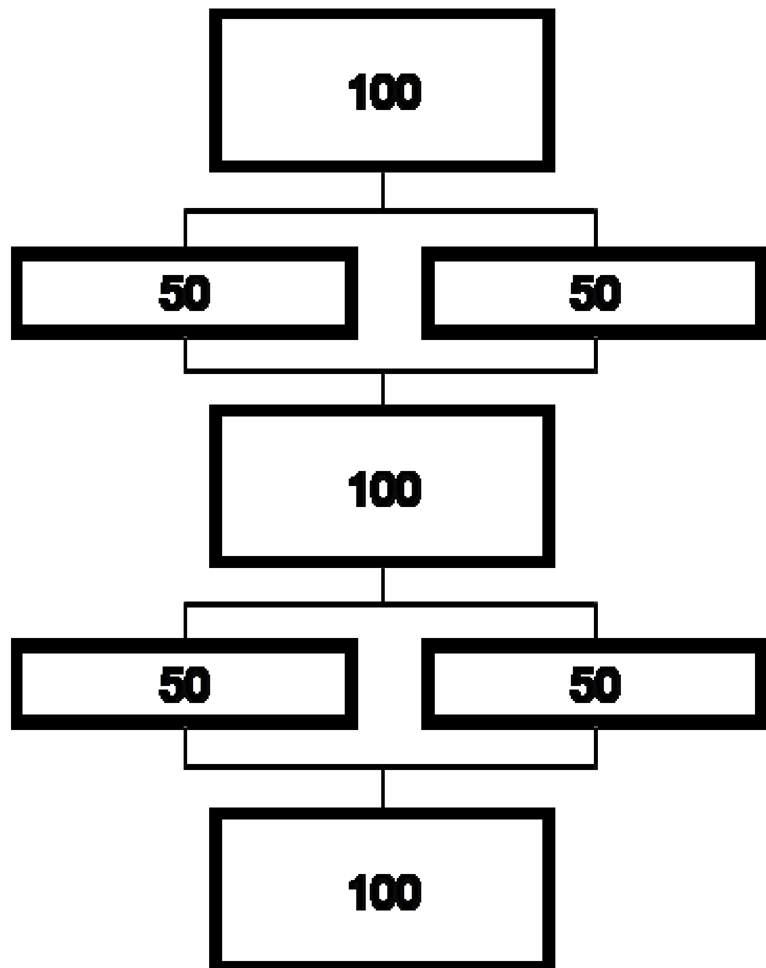
スケーリングを考慮して設計されたプログラムは、プロセッサーのコア数が増加するとより多くの処理を行う

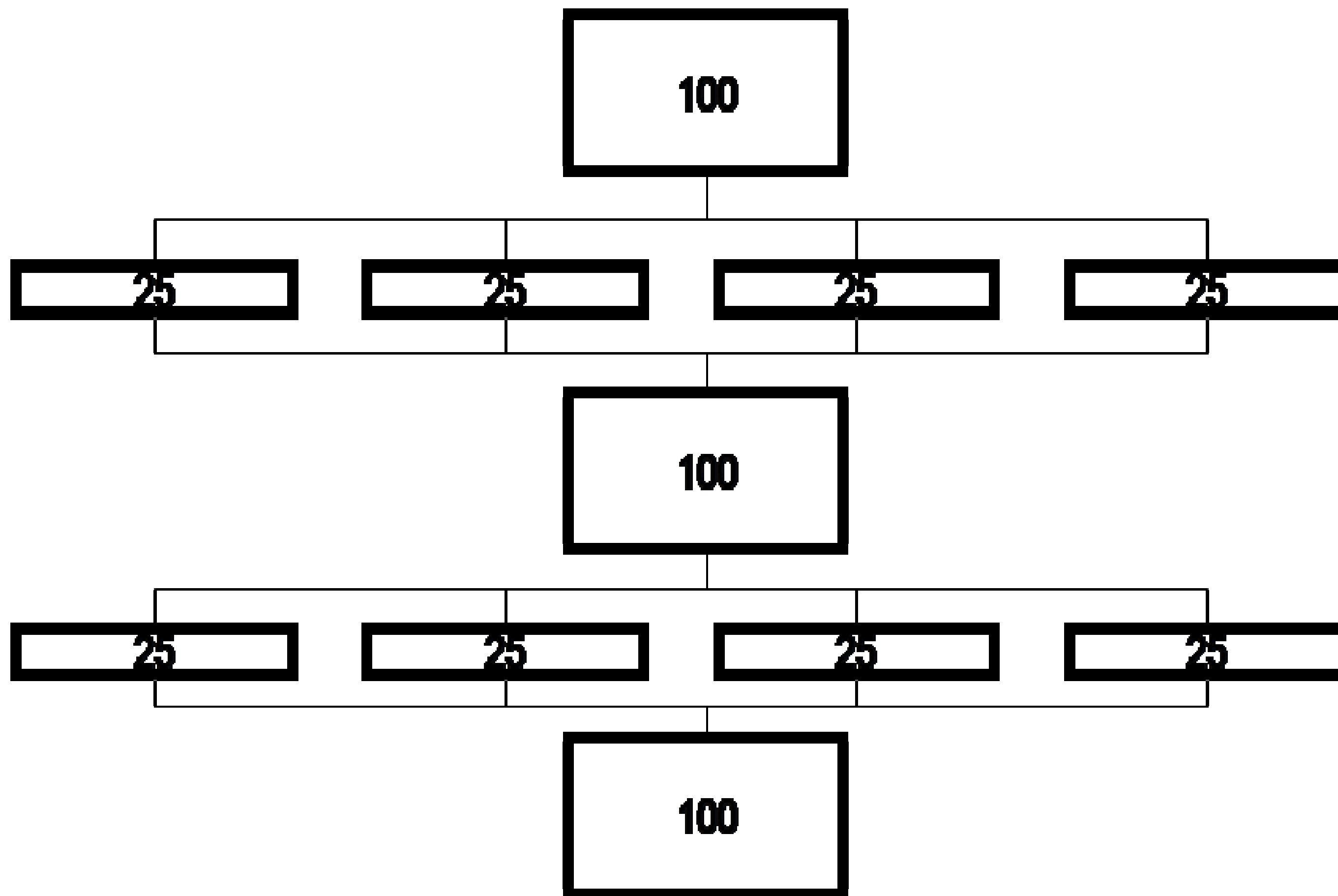
どれくらい並列化できるか？

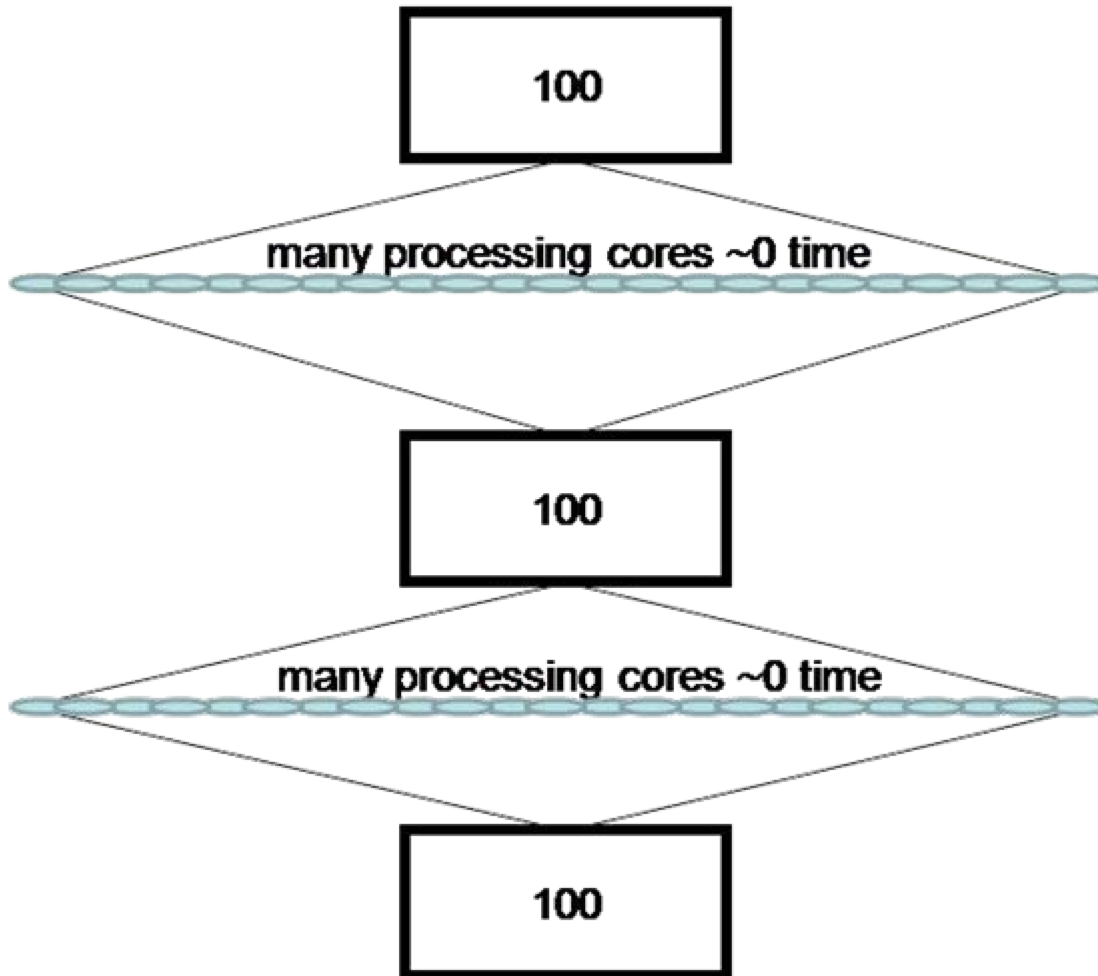
- Amdahl’s Law
- Gustafson’s observations on Amdahl’s Law

- アムダールの法則
- アムダールの法則対するグスタフソンの見解


100
100
100
100
100
Work 500 Time 500
Speedup 1X


100
50
50
100
50
50
100
Work 500 Time 400
Speedup 1.25X


100
25
25
25
25
100
25
25
25
25
100
Work 500 Time 350
Speedup 1.4X


100
many processing cores ~0 time
100
many processing cores ~0 time
100
Work 500 Time 300
Speedup 1.7X

アムダールの法則

"...the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude."

– Amdahl, 1967

"...ほぼ同じ大きさの直列処理の速度が改善されないのであれば、並列処理の速度を改善しようとする労力は無駄になる"

– Amdahl, 1967

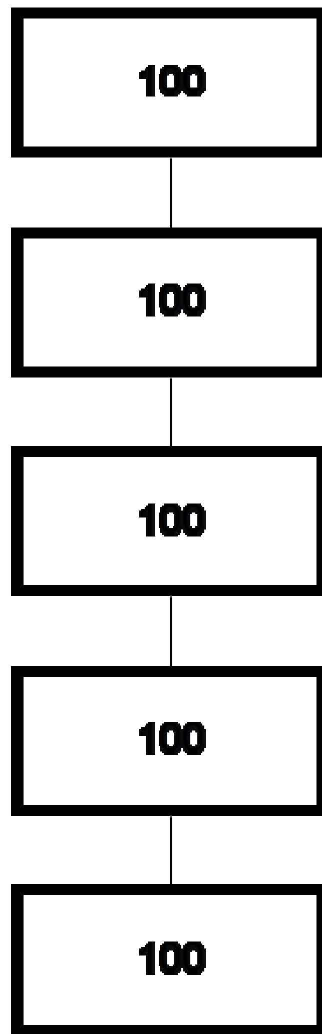
アムダールの法則 – 1つの見解

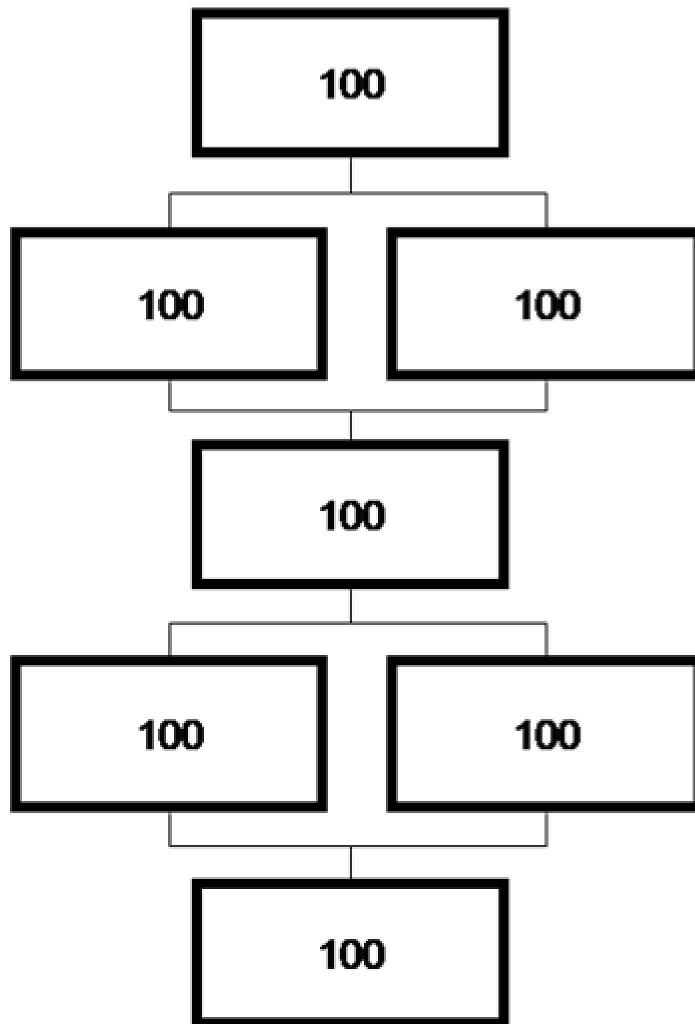
"…speedup should be measured by scaling the problem to the number of processors, not by fixing the problem size."

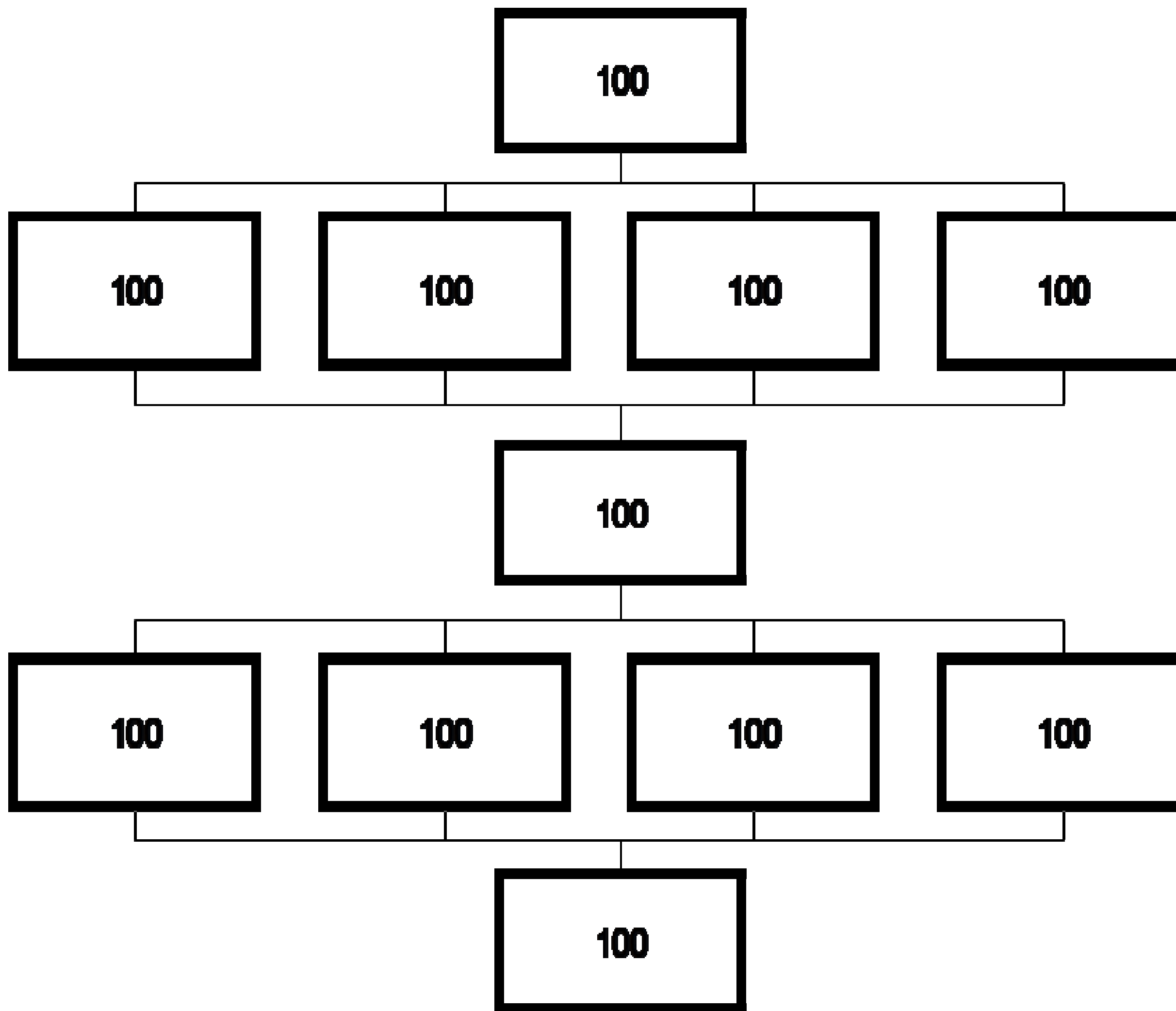
– Gustafson, 1988

"…スピードアップは処理サイズを固定するのではなくプロセッサーの数に処理をスケーリングして測定すべきである"

– Gustafson, 1988

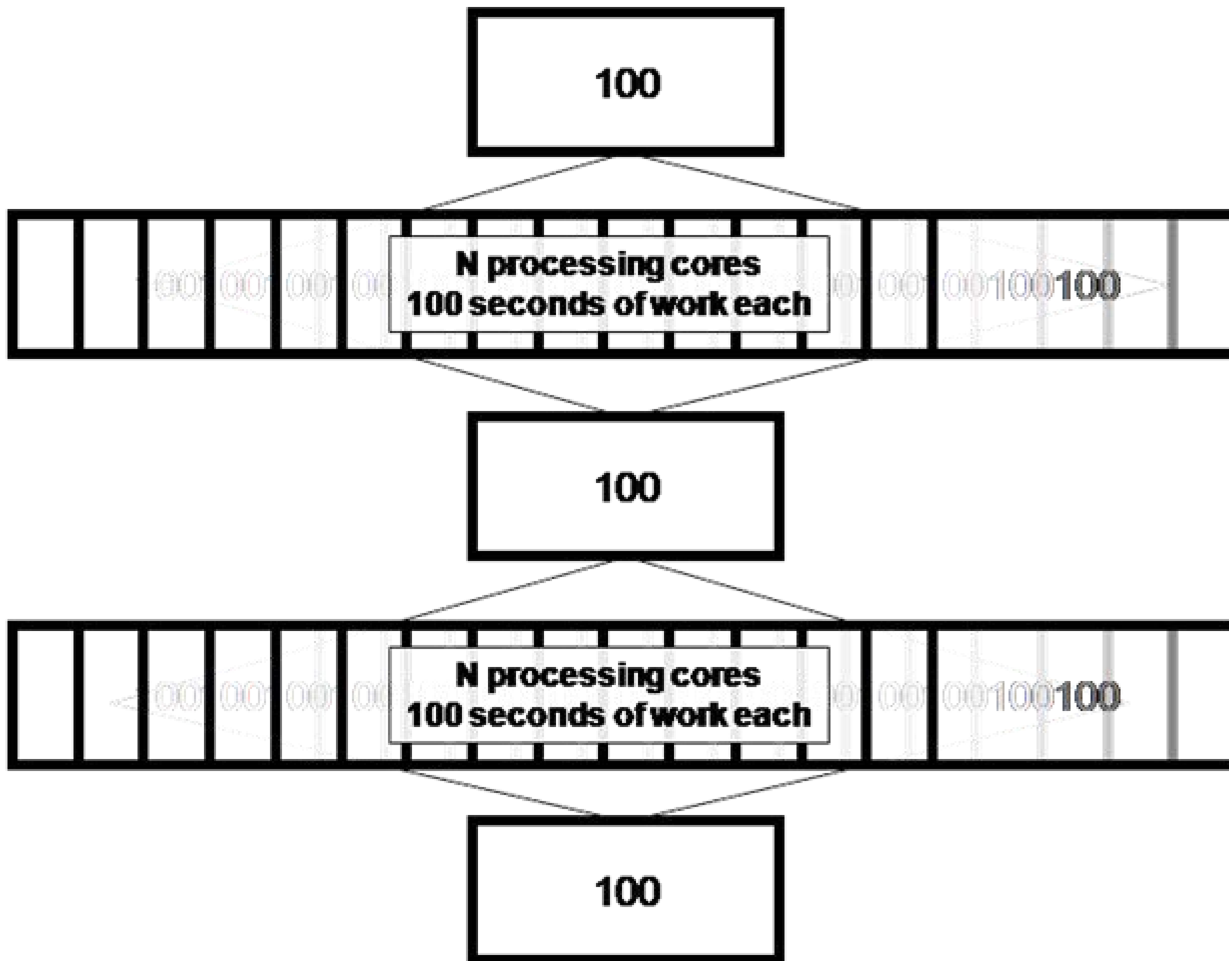

100
100
100
100
100
Work 500 Time 500
Speedup 1X


100
100
100
100
100
100
100
Work 700 Time 500
Speedup 1.4X



Work 1100 Time 500

Speedup 2.2X



Work 2*N*100+300 Time 500

Speedup O(N)

ワークロードを増やすことでスケーリングを計画する

The value of parallelism is easier to prove if you are looking forward than if you assume the world is not changing.

並列処理の価値は、世界は変わっていないと考えるよりも、前に進んでいると考えるほうがより簡単に証明できます

経験にもとづく手法

1. Think parallel.
2. Program using abstraction.
3. Program in tasks (chores), not threads (cores).
4. Design with the option to turn concurrency off.
5. Avoid using locks.
6. Use tools and libraries designed to help with concurrency.
7. Use scalable memory allocators.
8. Design to scale through increased workloads.

I hope you can see, in these tips for software developers, the implications for hardware design too.

1. 並列思考をする
2. 抽象化を使用したプログラム
3. スレッドではなくタスクをプログラムする
4. 並列性を無効にできるオプションを設計
5. ロックの使用を避ける
6. 並列化作業を補助するようなツールやライブラリーを利用する
7. スケーラブル・メモリー・アロケーターを利用する
8. ワークロードを増やしてスケールするように設計する

ソフトウェア開発者のためのこれらのヒントがハードウェアの設計にも密接な関係が見出されることを期待します

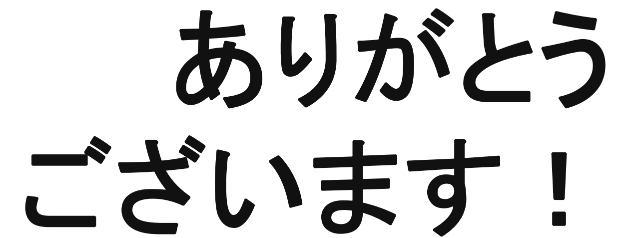
最後に：前方の道は曲がっている

1. Parallelism is intuitive
2. Heterogeneous processors are coming
3. Very high core counts are coming
4. Ease of programming and ease of use are more important than efficiency.
5. Parallelism can increase performance much faster than ever before
6. Parallelism offers new possibilities
7. All systems are affected – supercomputers, laptops, embedded systems.

1. 並列性は直感的
2. ヘテロジニアスなプロセッサーは現実のものに
3. 多くのコアを搭載するプロセッサーは現実のものに
4. 簡単なプログラミングと使いやすさは効率よりも重要
5. 並列化はこれまでよりもパフォーマンスをより高速化することが可能
6. 並列化は新たな可能性を生み出す
7. スーパーコンピューター、ラップトップ、組み込みシステムなどすべてのシステムが影響を受ける

Learn **Parallel**　並列化について学ぶ

Think **Parallel**　並列思考を行う



ジェイムス・レインダース
インテル

私が参加するプロジェクト…

http://go-parallel.com
http://threadingbuildingblocks.org
http://Whatif.intel.com
http://intel.com/technology/itj
http://intel.com/software/products