ソフトウェア基礎論 38−10
プログラミング言語 28−10
（1991. 3. 8）

並列分散 TRS シミュレータの実現方法について

山本晋一郎　直井徹　坂部俊樹　稲垣康善
名古屋大学工学部

並列分散 TRS シミュレータの実現のために、一定個数のセルの集まりとその管理、および書換え等の操作を抽象化したクラス *BOB* (*Bundle of Branches*) を提案する。*BOB* は汎用のメモリ管理単位であるページを、参照の局所性の維持と並列性を意識して項書換え系に特化した機構であり、*BOB* を用いた項書換え系の実現は、並列分散処理と動的な負荷分散を容易にする。また、*BOB* を用いた実現が参照の局所性を維持することを計算機実験によって定量的に確かめた。

An Implementation of Distributed Term Rewriting

Shinichirou YAMAMOTO　Tohru NAOI　Toshiki SAKABE　Yasuyoshi INAGAKI
Faculty of Engineering, Nagoya University

In this paper, we propose a new methodology of data management for distributed term rewriting. Our unit of parallel execution, called *BOB*, is described as a class in the terminology of object oriented paradigm. *BOB* rewrites redexes independently of other *BOB*s with careful access to cells in critical region. Moreover, it can achieve high locality of references and experiments show that our method has between two and five times locality than conventional one.

1 はじめに

近年の著しいハードウェア技術の進歩によって、ネットワーク上に数台から数百台のワークステーション (以下 WS) が接続され、実メモリの総計が Gbyte に達する環境も珍しくない。しかし、現在の利用形態は、計算機の個別利用、ファイルの共有、あるいは計算結果を他の計算機に表示する程度に限定され、ネットワーク上の複数の計算機を有機的に結合して一つの計算を行なうことは一般的ではない。特に、異なったアーキテクチャの計算機が混在する環境では、オブジェクトファイルの形式が異なるので、計算タスクの分割や委託あるいは動的な負荷分散などは困難であり、これらは計算機工学に課せられた課題でもある。しかし、メモリに対する副作用に立脚した既存の手続き型言語を用いる限り、これらを素直に実現することは難しいと思われる。

一方、関数型プログラムは本質的に並列性を内在しているため、当然のことながら並列分散処理と馴染みが良いと考えられるが、その並列性を十分に引き出している処理系は数少なく、残念ながら実現方法も確立していない。

本稿では関数型プログラムの理論モデルの一つである項書換え系を取り上げ、ネットワーク上に緩く結合された計算機上で動作する並列分散 TRS シミュレータの実現方法について報告する。

項書換え系を実装する方法として、リダクションマシンをチップレベルから設計して製作する方法もある。このアプローチは、技術的にも興味深く、性能的にも有利であるが、Lisp マシン、Pascal エンジンなどの例をみるまでもなくこのような専用マシンの普及にはかなりの時間がかかると考えられる。一方、汎用 WS の性能向上は著しく、この進歩がしばらく持続すると思われることより、ソフトウェアによる実現という本稿のアプローチもかなりの期間有用であり、この方法で得られた成果をリダクションマシンに生かすためにも必要な実験であると考えられる。

本研究の目標を以下に示す。

1. 項書換え系あるいは関数型言語処理系の並列分散実装技術の確立。
2. ガベージコレクション (以下 GC) の実現方法と参照の局所性の維持の関係の調査。
3. 分散 OS に対応した記号処理系の実装実験。

具体的には、一定個数のセルの配列とその上の書換えと GC に代表される機能を備えたクラス *BOB* (*Bundle of Branches*) を提案する。*BOB* は汎用のメモリ管理単位であるページを、参照の局所性の維持と並列性を意識して、項書換え系に特化した機構であり、*BOB* を用いた項書換え系の実現は、並列分散処理と動的な負荷分散を容易にする。

一般に、Lisp に代表される記号処理言語はポインタによる参照を多用する。そのような複雑なデータ構造の管理に関する研究に [3,4] などがある。これらの研究は一般的なリストデータを対象に新しい記憶管理の方式を提案しているが、本稿では対象を項書換え系に限定し、並列分散処理への応用を指向した。

以下，2 章で項書換え系の基本概念，3 章で *BOB* の基本アイデアと実現方法についてそれぞれ説明する。また、4 章で計算機実験による評価について報告する。なお、本稿で述べるネットワークとは ethernet のことであり、計算機の台数としては数台から数百台程度を考える。

2 項書換え系

2.1 項書換え系

項書換え系は左辺，右辺と呼ばれる項の対の集合で，各々の項の対は書換え規則と呼ばれる。項 s の変数記号に適当な代入を施した項が，項 r と一致するとき，s は r にマッチするといい，そのときの代入をマッチング代入という。ある書換え規則の左辺の項にマッチング代入を施した項が，項 s に一致するとき，項 s をリデックスという。一般に一つの項は複数のリデックスを部分項として持つ。特に，他のリデックスの部分項でないリデックスを最外リデックスといい，反対に，他のリデックスを部分項として含まないリデックスを最内リデックスという。さ

らに、ある項の全てのリデックスからなる集合をその項のリデックス集合という。また，項 s と項 r にそれぞれ適当な代入を施した結果の項が一致するとき，s と r は単一化可能であるという。

項書換え系における計算は、書換えと呼ばれる次のような操作である。与えられた入力項のリデックス集合から、ある基準にしたがって、実際に書換える部分項を選択し、書換え規則の左辺とマッチするその部分項を右辺にマッチング代入を施して得られる項と置き換える。また、この基準を戦略と呼ぶ。

本稿では，次の 2 つの制約を満たす項書換え系を対象とする。

1. 無曖昧性: 任意の 2 つの書換え規則に対して，一方の左辺が他方の左辺の変数以外の部分項と単一化可能でない。ただし，自分自身との自明な単一化を除く。

2. 左線形性: 書換え規則の左辺に同一の変数が 2 回以上出現しない。

項書換え系が上の 1 と 2 を満たすとき，その系は合流性を持ち，各々の項の正規形は (存在するならば) 一意である [1]。すなわち、無曖昧かつ左線形な項書換え系には、バックトラックが不必要で、並列処理が可能である。

2.2 実現方法

この節では、項書換え系の一般的で単純な実現方法とその問題点について議論する。一般に項は頂点に関数記号が付加された順序木と同一視することができる。以下ではこれをさらに詳しく説明する。各頂点はセルと呼ばれる n+1 項組 ⟨ 関数記号, 子供へのポインタ 1 ,..., 子供へのポインタ n⟩ を持つ。ただし、関数記号の arity を n とする。しかし、この実現では関数記号の arity によってセルの大きさが異なり、セル管理が繁雑になる。よって本稿では大きさの固定された一種類のセルを頂点とする二分木を用いて一般の木を模方する。この場合のセルは、⟨ 関数記号, 子供へのポインタ, 兄弟へのポインタ ⟩ となる。例として項 $f(g(a), h(b, b))$ の n 分木による表現を図 1に、二分木による表現を図 2に示す。これらの図において、頂点は円によって、ポインタは矢印によって表される。また、矢印の先に頂点が存在しないポインタは空ポインタを表す。 項書換え系では、書換え規則の右辺に同一の変数が複数現れる場合がある。木構造を用いると、そのような変数に対応する部分項全体のコピーを複数作る必要があり、この作業は処理系にとって大きな負担であることが知られている。この場合、変数の部分はポインタによって共有し実際にコピーは行なわない。このような実現方法を採用すると、木構造で表された項は、一般に書換えが進むにつれて共通の部分項を共有した DAG(Directed Acyclic Graph) になる。

項書換え系の実現は、 DAG で表現された項に対して、書換え規則の左辺集合とマッチングを行ない、リデックスを探索し、発見したリデックスを書換えることである。リデックスの探索も項書換え系の処理系にとって重要な問題であるが、以下では *BOB* の実現に深く関係する書換えとセル管理について説明する。リデックスの書換えは次の二つの段階に分けられる。

1. 発見されたリデックスに対応する右辺を構成する。

2. リデックス内の書換え規則の左辺に対応するセルを解放する。

一般的に、セルはグローバルなフリーリストによっ

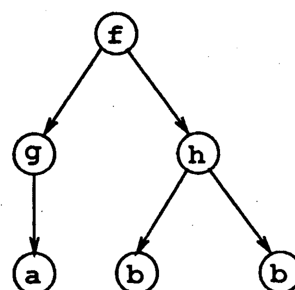

図 1: n 分木による表現

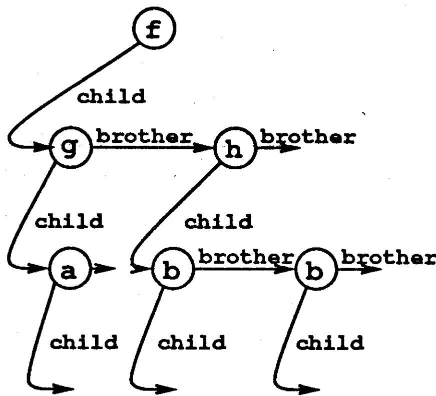

図 2: 二分木による表現

て管理される。すなわち、初期状態では、入力項を構成するセルを除く全てのセルはフリーリストにつながれている。そして、書換えの第1段階では、右辺の構成に必要なセルをフリーリストから順番に消費する。また、第2段階では、他のセルから参照されていない解放されたセルをフリーリストに追加される。

2.3 参照の局所性と並列性

項書換え系では項の形が動的に激しく変化するため、一般的なセル管理方式では、書換えの進行に伴って参照の局所性が著しく減少する。局所性の減少はスラッシングと呼ばれる現象の原因となり効率の低下をまねく。よって、局所性を保ちつつ項の書換えをおこなう必要がある。また、一般的な記号処理系で用いられているグローバルなフリーリストを用いるセルの管理方法も、参照の局所性を考慮していない。例えば、リデックスの近くにフリーなセルが存在することを仮定すると、右辺の構成に近傍のセルを用いることが望まれるが、グローバルなフリーリストを用いる実現では、このことが保証されない。

さらに、項書換え系の中に自然に内在する並列性を引き出すことが我々の目標の一つであるが、前述の一般的な実現では、相互排除すべき領域、すなわち保護領域が明確にされていない。よって、分散OSにおいてスレッドを用いてのマルチスレッド化などが容易ではない。

2.4 セル参照

一般的な (仮想) メモリ空間とは、アドレスを定義域、bit 列を値域とする関数と見ることができる。さらに、対象を項書換え系や Lisp に限定すると、セルのアドレスからセルの属性への関数と捉えることができる。そこで以下では、セルアドレスからそのセルの属性を求めることをセルに対する参照 (以下、セル参照) という。一台の計算機上での逐次的な実現を考えると、セルのアドレスとメモリ空間のアドレスを同一視することが可能である。一方、複数の計算機を用いる場合、あるいは複数のプロセスを用いる場合は、計算機の id、プロセスの id、プロセス内のセルのアドレスの3項組を用いて、メモリ空間の全てのセルに一意な番号を定める必要がある。ここでは、この3項組をセルアドレスと呼ぶ。

メモリ空間内において、一つのセルを頂点とする部分 DAG と書換え規則の左辺集合の各要素のマッチングを行なうためには、セル参照を、計算機やプロセスに対して透過的に実現する必要がある。このような透過的なセル参照の単純な実現法として、全てのセル参照を特定の関数 (セルデーモンと呼ぶ) を通して行なう方法が考えられる。セルデーモンは、与えられたセルアドレスからそのセルの存在する計算機を求め、必要ならばプロセス間通信によって該当するセルにアクセスする。このセルデーモンは MACH における外部ページャと同様な機構である。直観的には、外部ページャは他の計算機のメモリ資源を2次記憶装置として使用するための機構である。

しかし、外部ページャは巨大なメモリ空間をユーザに提供するだけで、並列性に関係する機構と局所性の維持の機構は別に実現する必要がある。

3 BOB - Bundle of Branches

前章までの考察より、局所性と並列性を意識した項書換え系専用の機構の実現が望ましい。この章では、一定個数のセルの集まりとその管理、および書換え等の操作を抽象化したクラス *BOB* (*Bundle of Branches*) を提案する。*BOB* は汎用のメモリ管理単位であるページを、参照の局所性の維持と並列性を意識して、項書換え系に特化したものであり、同一 *BOB* 内のセル参照と *BOB* 外に位置するセルの参照を明確に区別する。

BOB 内のセル参照は、それらのセルが同一のプロセス内に存在することが保証されていることから、通常のポインタによる間接参照とみなすことができる。一方、*BOB* 外に位置するセルの参照は、プロセス間通信の機構を用いて参照を行う必要があるが、本稿では、プロセス間通信実現の詳細にはふれない。

以下では、最初に *BOB* 間にまたがったセル参照のための機構 Port を、次に *BOB* の全体像を説明する。

3.1 Port

この節では *BOB* 間のセル参照について述べる。DAG の構成には片方向のポインタで十分であるが、片方向ポインタによる実現では、*BOB* を計算機間で転送することによる、*BOB* を単位とした動的な負荷分散は不可能である。したがって、*BOB* 間のセル参照は双方向のポインタによって実現する必要がある。

一般に、任意の二つの *BOB* は別々のプロセスに存在するので、通常のポインタによって *BOB* にまたがった参照を実現することはできない。そこで、Port と呼ぶセルアドレスを表す 3 項組 ⟨ 計算機 id、プロセス id、プロセス内のセルのアドレス ⟩ を仮想的なポインタとみる。ここで、計算機 id はネットワーク全体で一意な id、プロセス id は同一計算機内で一意な id とする。Port には Source と Sink の 2 種類の型があり、Source と Sink の組によって *BOB* 間のセル参照を構成している。すなわち、Source は他の *BOB* の Sink と双方向のポインタで結合されている。

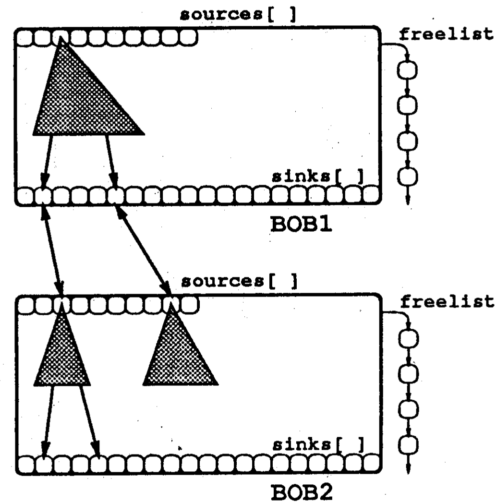

図 3: *BOB* の概念図

よって、ある Sink は他の *BOB* の Source を参照し、逆に Source はある Sink から参照されている。言い替えると、Source は *BOB* 内の部分項の頂点を保持し、文字通り *BOB* の入口である。Sink は *BOB* 内のセルから他の *BOB* セルへの参照を保持し、*BOB* の出口である。以下では、この Source と Sink の組によるセル参照をポートリンクと呼ぶ。

3.2 BOB

部分木 (正確には枝) をひとかたまりに束ねたものが、*BOB* の直観的なイメージである。そのイメージを図 3に示す。クラス *BOB* は以下のメンバとメンバ関数を持つ。ただし、BobId は全体で一意な id である。

- `heap[ ]`: セルの配列。
- `sources[ ]`: Source の配列。
- `sinks[ ]`: Sink の配列
- `freeListTop`: *BOB* 内のフリーリストを保持する変数。

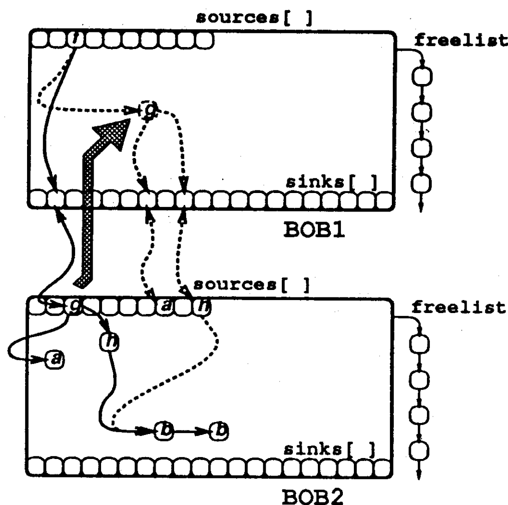

図 4: throw() の動作

- rewrite(): *BOB* 内のリデックスを書換えを行なう関数。ただし、*BOB* 間にまたがるリデックスの書換えはしない。

- throw(Port source, BobId bob_to):
引数 source で指定された Source を bob_to で指定された *BOB* に移動する。

- catch(Port source, BobId bob_from):
引数 source で指定された Source を bob_from で指定された *BOB* から移動する。

- migrate(MachineId machine_to):
BOB を引数 machine_to で指定された計算機に移動する関数。

throw() の動作を図 4に示す。この図は、項 $f(g(a), h(b, b))$ が BOB1 と BOB2 にまたがって配置されている状況を実線で表している。その後、BOB2 のソースを BOB1 に throw した状況が点線によって表されている。

Port の内容の変更を伴う操作は保護領域として扱う必要があるが、Port の値を変更しない *BOB* 内の書換えは、他の *BOB* の書換えと全く独立に行なうことができるこれは本方式の大きな特徴である。すなわち、各々の *BOB* を異なる計算機上のプロセスで処理することや、同一プロセス内の異なるスレッドで処理することが可能となる。また、OS によるページングのページサイズと *BOB* のサイズを整合させて主記憶と 2 次記憶間の転送回数を減少させることにより、いわゆるスラッシングを避けることもできる。

3.3 Port の操作

BOB 内の書換えが並列に実行可能であることと対照的にこれらの操作は保護領域として実現される。よって、*BOB* の状態は内部の書換えを行なう状態と Port の操作を行なう状態のいづれかである。以下の操作は Port の内容を変更する操作である。

- catch() と throw(): 書換えの途中で *BOB* 内のフリーなセルの個数がある基準以下になったときに、使われているセルの一部を他の *BOB* 移動してフリーセルの個数を確保するために用いられる。また、*BOB* 内の書換えのみを行なうと *BOB* 間にまたがったリデックスが書換えられないまま残されてしまう。catch() と throw() は、このような *BOB* にまたがるリデックスを他の *BOB* に移動させて書換えを継続させるためにも用いられる。

- GC の伝播: ある *BOB* の中で発生した GC を他の *BOB* に伝達する。GC によって Sink が回収された場合、Sink が参照している Source も回収される必要がある。これを GC の伝播という。

- migrate(): 特定の計算機に負荷が集中した時に、*BOB* 単位で処理を他の計算機に委託する。

これらの操作はデッドロックを防止するために、対応する *BOB* をロックして、内部の書換えの修了を待ち、逐次的に処理される。

3.4 BOBの特徴

前節までの議論から明らかなように、*BOB* の特徴として以下の項目を挙げることができる。

- *BOB* ごとにフリーリストを持つため、セル参照の局所性が比較的良く維持される。
- *BOB* 内の書換えは他の *BOB* と独立に行なうことができるため、自然な並列性を引き出すことができる。
- *BOB* の移動による動的な負荷分散実現の可能性がある。

一方、*BOB* が持つセル、Source、Sink 等の個数が固定されているために、メモリの有効利用ができないことと、*BOB* 間にまたがるリデックスの書換えが遅れることが欠点である。

4 実験

この章では、項書換え系の実現に *BOB* を用いることによって、参照の局所性が比較的維持されることを計算機実験によって定量的に示す。

4.1 準備

以下ではセル c のセルアドレスを address(c) によって表す。最初にセル参照の距離を定義する。

【定義 4.1.1】 セル c においてセルアドレス a で表現されるセル b を参照した場合の距離を

$$distance(c,a) = \begin{cases} 0 & a = 0 \text{ のとき} \\ 0 & \text{a と b が異なる } BOB \text{ に位置するとき} \\ |address(c) - a| & \text{それ以外} \end{cases}$$

とする。□

【定義 4.1.2】 項 t の距離 $distance(t)$ を t を構成する全てのセル参照の距離の合計とする。□

【定義 4.1.3】 書換え $t_0 \rightarrow t_1 \rightarrow \ldots \rightarrow t_n$ の距離を

$$\sum_{i=0}^{n} distance(t_i)$$

によって定義する。□

4.2 実験結果

計算機実験を行なうために、*BOB* を用いた項書換え系のシミュレータを作成した。ただし、このシミュレータは単独のプロセス上で逐次的に動作する。シミュレータの作成には C と C++ 言語を用い、プログラムの規模は lex と yacc のソースを含めて 5000 行程度である。

以下では、階乗を計算する項書換え系を対象とした実験について報告する。サンプルとして用いた項書換え系を以下に示す。

```
fact(0) ▷ s(0)
fact(s(X)) ▷ mult(s(X), fact(X))
mult(0, Y) ▷ 0
mult(s(X), Y) ▷ add(Y, mult(X, Y))
add(0, Y) ▷ Y
add(s(X), Y) ▷ s(add(X, Y))
```

実験は一つの *BOB* 内のセルの個数をパラメータとして行なった。なぜなら、入力項の大きさ、および書換えの途中の項の大きさと比較して *BOB* 内のセルの個数が十分に豊富な場合、*BOB* を用いた実現はグローバルなフリーリストを用いた一般的な実現と相違がないと考えられるためである。

実験ではパラメータの変化に伴って、入力項からその正規形に至る書換えの距離がどのように変化するかを計測した。また、同時に正規形が得られるまでのステップ数と書換えの途中で用いたポインタとポートリンクの総数も計測した。

書換えの戦略には各 *BOB* 内の最外リデックスを全て同時に書換える戦略を採用した。*BOB* 間にまたがったリデックスを無視すれば、全体としては最外リデックス集合を包含したリデックス集合を書換える戦略とみなすことができる。ただし、*BOB* 間

にまたがったリデックスの書換えは、後回しにされることがあるので、正確には並列最外戦略と一致しない。実験結果の表を付録1に示す。ここで、表の distance/pointer は一つのポインタあたりの平均的な参照の距離である。

4.3 評価

付録1より以下の傾向を読みとることが可能である。書換えの途中の項の大きさと比較して相対的に *BOB* 内のセルの個数が減少するにつれて、

- ステップ数が若干増加する。
- ポインタあたりの参照の距離が著しく減少する。
- 使用されたポートリンクの個数が増加する。

ここで、ステップ数の増加は *BOB* 間にまたがるリデックスが単独の *BOB* 内に移動するためのコストによると考えられる。また、ポートリンクの個数の増加は処理系にとって負担となる。*BOB* 間のセルの転送戦略を改善して、これらの負担を低減することは今後の課題である。

この実験結果より *BOB* の持つ参照の局所性を維持する機能を確認することができた。

5 まとめと今後の課題

並列分散 TRS シミュレータの実現のために、汎用のメモリ管理単位であるページを、参照の局所性の維持と並列性を意識して項書換え系に特化した *BOB* (*Bundle of Branches*) を提案した。*BOB* は、一定個数のセルの配列とその上の書換えと GC に代表される機能を備えたクラスであり、*BOB* を用いた項書換え系の実現は、並列分散処理と動的な負荷分散を容易にする。また、*BOB* を用いた実現が参照の局所性を維持することを計算機実験を通して定量的に確かめた。

実験は単独の計算機上で逐次的に動作する項書換え系シミュレータを用いて行なった。現在、プロセス間通信とスレッドを用いた処理系を作成中である。並列版の完成と *BOB* を用いた動的な負荷分散の実験と評価を行なうことは今後の課題として残されている。

謝辞

御討論頂いた平田富夫助教授、外山勝彦中京大講師、杉野花津江助手、酒井正彦助手、結縁祥治助手ならびに研究室の皆様に感謝致します。

参考文献

[1] HUET,G.: "Confluent Reductions: Abstract properties and applications to term rewriting systems", JACM,Vol.27, No.4(1980).

[2] Huet,G. and Lévy,J.-J.: "Call by need computations in non-ambiguous linear term rewriting systems", Rapport INRIA nr.359 (1979).

[3] 前川博俊 他:"Pointed-Linked Data における仮想記憶管理の一手法", 情報処理研究会資料, SYM50-1 (1989).

[4] 實藤隆則 他:"Linked Data Structures のための記憶管理とその動特性", 情報処理研究会資料, ARC85-10 (1990).

付録1

実験結果		fact(3)	fact(4)	fact(5)
cell 16	step	33	-	-
	distance/pointer	266	-	-
	pointer	734	-	-
	port link	133	-	-
cell 32	step	26	68	-
	distance/pointer	312	366	-
	pointer	534	4016	-
	port link	38	607	-
cell 64	step	25	61	-
	distance/pointer	400	645	-
	pointer	596	3620	-
	port link	0	137	-
cell 128	step	25	60	200
	distance/pointer	400	853	1153
	pointer	596	3913	40054
	port link	0	92	1888
cell 256	step	25	58	193
	distance/pointer	400	1406	2473
	pointer	596	4817	42250
	port link	0	0	615
cell 512	step	25	58	190
	distance/pointer	400	1406	3206
	pointer	596	4817	49853
	port link	0	0	331
cell 1024	step	25	58	189
	distance/pointer	400	1406	5956
	pointer	596	4817	67744
	port link	0	0	0