processes and path expressions are dual approaches to specifying the traces created in a computation; the processes generate sequences of actions which are constrained by path expressions to achieve particular synchronization constraints associated with the operations on abstract data types.

Summary

In practice, synchronization of parallel processes remains a difficult problem that becomes ever more complex with increased concurrency in architecture. Path expressions provide a separate specification of synchronization from the code of processes that, under some circumstances, can simplify parallel programming. Although not widely adopted, their declarative approach to specifying synchronization has been found useful in various parallel applications. Path expressions have been used to synchronize parallel operations on objects in parallel languages, real-time languages, event systems, VSLI, workflow, and debugging.

Bibliographic Notes and Further Reading

The semantics of parallel programs may be described using trace-based semantics. Path expression implementations restrict or recognize the permitted traces of parallel programs as described by Lauer and Campbell [6]. A semantics for path expressions is given by Dinning and Mishra using partially ordered multisets [8] and the authors provide a fully parallel implementation for a path expression language on MIMD shared memory architectures. Path Pascal and open path expressions were used as pedagogical tools for teaching parallel programs [9, 13].

Bibliography

1. Anantharaman TS, Clarke EM, Mishra B (1986) Compiling path expressions into VLSI circuits. Distrib Comput 1:150–166
2. Andler S (1979) Predicate path expressions. In: POPL'79 proceedings of the 6th ACM SIGACT-SIGPLAN symposium on principles of programming languages. ACM Press, New York, pp 226–236
3. Bruegge B, Hibbard P (1983) Generalized path expressions: a high-level debugging mechanism. J Syst Softw 3:265–276 (Elsevier Science Publishing)
4. Campbell RH (1976) Path expressions: a technique for specifying process synchronization. PhD. Thesis, The University of Newcastle Upon Tyne
5. Campbell RH, Habermann AN (1974) The specification of process synchronization by path expressions. In: Gelenbe E, Kaiser C (eds) Operating systems. Lecture notes in computer science, vol 16. Springer, Berlin, pp 89–102
6. Lauer PE, Campbell RH (1979) Formal semantics of a class of high-level primitives for coordinating concurrent processes, Acta Informatica, 5(4):297–332
7. Comte D, Durrieu G, Gelly O, Plas A, Syre JC (1978) Parallelism, control and synchronization expression in a single assignment language. ACM SIGPLAN Not 13(1): 25–33
8. Dinning A, Mishra B (1990) A fully parallel algorithm for implementing path expressions. J Parallel Distrib Comput 10:205–221
9. Dowsing RD, Elliott R (1986) Programming a bounded buffer using the object and path expression constructs of path pascal. Comput J 29(5):423–429
10. Heinlein C (2000) Workflow and process synchronization with interaction expressions and graphs. Ph. D. Thesis (in German), Fakultät für Informatik, Universität Ulm
11. Hoepner P (1992) Synchronizing the presentation of multimedia objects. Comput Commun 15(9):557–564
12. Kidd M-EC (1994) Ensuring critical event sequences in high integrity software by applying path expressions. Sandia Labs, Albuquerque
13. Kolstad RB, Campbell RH (1980) Path Pascal user manual. SIGPLAN Not 15(9):15–24
14. Laure E (1999) ParBlocks – a new methodology for specifying concurrent method executions in opus. In: Amestoy P, Berger P, Dayde M, Ruiz D, Duff I, Fraysse V, Giraud L (eds) Euro-Par'99. Lecture notes in computer science, vol 1685. Springer, Berlin, pp 925–929
15. Preiss O, Shah AP, Wegmann A (2003) Generating synchronization contracts for web services. In: Khosrow-Pour M (ed) Information technology and organizations: trends, issues, challenges & solutions, vol 1. Idea Group Publishing, Hershey, pp 593–596
16. Rees O (1993) Using path expressions as concurrency guards. Technical report, ANSA
17. Schoute AL, Luursema JJ (1990) Realtime system control by means of path expressions. In: Proceedings Euromicro '90 Workshop on Real Time, Horsholm, Denmark, pp 79–86
18. Shaw AC (1978) Software description with flow expressions. IEEE Trans Softw Eng SE-4(3):242–254

PaToH (Partitioning Tool for Hypergraphs)

Ümit Çatalyürek[1], Cevdet Aykanat[2]
[1]The Ohio State University, Columbus, OH, USA
[2]Bilkent University, Ankara, Turkey

Synonyms

Partitioning tool for hypergraphs (PaToH)

Definition

PaToH is a sequential, multilevel, hypergraph partitioning tool that can be used to solve various combinatorial scientific computing problems that could be modeled as hypergraph partitioning problem, including sparse matrix partitioning, ordering, and load balancing for parallel processing.

Discussion

Introduction

Hypergraph partitioning has been an important problem widely encountered in VLSI layout design [22]. Recent works since the late 1990s have introduced new application areas, including one-dimensional and two-dimensional partitioning of sparse matrices for parallel sparse-matrix vector multiplication [6–8, 12], sparse matrix reordering [6, 11], permuting sparse rectangular matrices into singly bordered block-diagonal form for parallel solution of LP problems [3], and static and dynamic load balancing for parallel processing [5]. PaToH [9] has been developed to provide fast and high-quality solutions for these motivating applications.

In simple terms, the hypergraph partitioning problem can be defined as the task of dividing a hypergraph into two or more roughly equal sized parts such that a cost function on the hyperedges connecting vertices in different parts is minimized. The hypergraph partitioning problem is known to be NP-hard [22], therefore a wide variety of heuristic algorithms have been developed in the literature to solve this complex problem [1, 15, 21, 23, 25]. Following the success of multilevel partitioning schemes in ordering and graph partitioning [4, 16, 18], PaToH [9] has been developed as one of the first multilevel hypergraph partitioning tools.

Preliminaries

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{N})$ is defined as a set of vertices (also called cells) $\mathcal{V}$ and a set of nets (hyperedges) $\mathcal{N}$ among those vertices. Every net $n \in \mathcal{N}$ is a subset of vertices, that is, $n \subseteq \mathcal{V}$. The vertices in a net n are called its *pins* in PaToH. The *size* of a net, $s[n]$, is equal to the number of its pins. The *degree* of a vertex is equal to the number of nets it is connected to. Graph is a special instance of hypergraph such that each net has exactly two pins. Vertices and nets of a hypergraph can be associated with weights. For simplicity in the presentation, net weights are refered as *cost* here and denoted with $c[.]$, whereas $w[.]$ will be used for vertex weights.

$\Pi = \{\mathcal{V}_1, \mathcal{V}_2, \ldots, \mathcal{V}_K\}$ is a *K-way partition* of $\mathcal{H}$ if the following conditions hold:

- Each part $\mathcal{V}_k$ is a nonempty subset of $\mathcal{V}$, that is, $\mathcal{V}_k \subseteq \mathcal{V}$ and $\mathcal{V}_k \neq \emptyset$ for $1 \leq k \leq K$.
- Parts are pairwise disjoint, that is, $\mathcal{V}_k \cap \mathcal{V}_\ell = \emptyset$ for all $1 \leq k < \ell \leq K$.
- Union of K parts is equal to $\mathcal{V}$, that is, $\bigcup_{k=1}^{K} \mathcal{V}_k = \mathcal{V}$.

In a partition Π of $\mathcal{H}$, a net that has at least one pin (vertex) in a part is said to *connect* that part. *Connectivity* λ_n of a net n denotes the number of parts connected by n. A net n is said to be *cut* (*external*) if it connects more than one part (i.e., $\lambda_n > 1$), and *uncut* (*internal*) otherwise (i.e., $\lambda_n = 1$). In a partition Π of $\mathcal{H}$, a vertex is said to be a *boundary* vertex if it is incident to a cut net. A K-way partition is also called a *multiway* partition if $K > 2$ and a *bipartition* if $K = 2$. A partition is said to be balanced if each part $\mathcal{V}_k$ satisfies the *balance criterion*:

$$W_k \leq W_{avg}(1+\varepsilon), \quad \text{for } k = 1, 2, \ldots, K. \tag{1}$$

In (1), weight W_k of a part $\mathcal{V}_k$ is defined as the sum of the weights of the vertices in that part (i.e., $W_k = \sum_{v \in \mathcal{V}_k} w[v]$), W_{avg} denotes the weight of each part under the perfect load balance condition (i.e., $W_{avg} = (\sum_{v \in \mathcal{V}} w[v])/K$), and ε represents the predetermined maximum imbalance ratio allowed.

The set of external nets of a partition Π is denoted as $\mathcal{N}_E$. There are various [13, 27] *cutsize* definitions for representing the cost $\chi(\Pi)$ of a partition Π. Two relevant definitions are:

$$\begin{aligned} &(a) \quad \chi(\Pi) = \sum_{n \in \mathcal{N}_E} c[n] \quad \text{and} \\ &(b) \quad \chi(\Pi) = \sum_{n \in \mathcal{N}_E} c[n](\lambda_n - 1). \end{aligned} \tag{2}$$

In (2a), the cutsize is equal to the sum of the costs of the cut nets. In (2b), each cut net n contributes $c[n](\lambda_n - 1)$ to the cutsize. The cutsize metrics given in (2a) and (2b) will be referred to here as *cut-net* and *connectivity* metrics, respectively. The hypergraph partitioning problem can be defined as the task of dividing a hypergraph into two or more parts such that the cutsize is minimized, while a given balance criterion (1) among part weights is maintained.

A recent variant of the above problem is the *multi-constraint hypergraph* partitioning [2, 6, 10, 19, 24] in which each vertex has a vector of weights associated

with it. The partitioning objective is the same as above, and the partitioning constraint is to satisfy a balancing constraint associated with each weight. Let $w[v,i]$ denote the C weights of a vertex v for $i = 1,\ldots,C$. Then balance criterion (1) can be rewritten as:

$$W_{k,i} \leq W_{avg,i}\,(1+\varepsilon) \text{ for } k = 1,\ldots,K \text{ and } i = 1,\ldots,C\,, \tag{3}$$

where the ith weight $W_{k,i}$ of a part $\mathcal{V}_k$ is defined as the sum of the ith weights of the vertices in that part (i.e., $W_{k,i} = \sum_{v\in\mathcal{V}_k} w[v,i]$), and $W_{avg,i}$ is the average part weight for the ith weight (i.e., $W_{avg,i} = (\sum_{v\in\mathcal{V}} w[v,i])/K$), and ε again represents allowed imbalance ratio.

Another variant is the *hypergraph partitioning with fixed vertices*, in which some of the vertices are fixed in some parts before partitioning. In other words, in this problem, a *fixed-part* function is provided as an input to the problem. A vertex is said to be *free* if it is allowed to be in any part in the final partition, and it is said to be fixed in part k if it is required to be in $\mathcal{V}_k$ in the final partition Π.

Using PaToH

PaToH provides a set of functions to read, write, and partition a given hypergraph, and evaluate the quality of a given partition. In terms of partitioning, PaToH provides a user customizable hypergraph partitioning via multilevel partitioning scheme. In addition, PaToH provides hypergraph partitioning with fixed cells and multi-constraint hypergraph partitioning.

Application developers who would like to use PaToH can either directly use PaToH through a simple, easy-to-use C library interface in their applications, or they can use stand-alone executable.

PaToH Library Interface

PaToH library interface consists of two files: a header file `patoh.h` which contains constants, structure definitions, and functions proto-types, and a library file `libpatoh.a`.

Before starting to discuss the details, it is instructive to have a look at a simple C program that partitions an input hypergraph using PaToH functions. The program is displayed in Fig. 1. The first statement is a function call to read the input hypergraph file which is given by the first command line argument. PaToH partition functions are customizable through a set of parameters. Although the application user can set each of these parameters one by one, it is a good habit to call PaToH function `PaToH_Initialize_Parameters` to set all parameters to one of the three preset default values by specifying `PATOH_SUGPARAM_<preset>`, where `<preset>` is `DEFAULT`, `SPEED`, or `QUALITY`. After this call, the user may prefer to modify the parameters according to his/her need before calling `PaToH_Alloc`. All memory that will be used by PaToH partitioning functions is allocated by `PaToH_Alloc` function, that is, there will be no more dynamic memory allocation inside the partitioning functions. Now, everything is set to partition the hypergraph using PaToH's multilevel hypergraph partitioning functions. A call to `PaToH_Partition` (or `PaToH_MultiConst_Partition`) will partition the hypergraph, and the resulting partition vector, part weights, and cutsize will be returned in the parameters. Here, variable `cut` will hold the cutsize of the computed partition according to cutsize definition (2b) since, this metric is specified by initializing the parameters with constant `PATOH_CONPART`. The user may call partitioning functions as many times as he/she wants before calling function `PaToH_Free`. There is no need to reallocate the memory before each partitioning call, unless either the hypergraph or the desired customization (like changing coarsening algorithm, or number of parts) is changed.

A hypergraph and its representation can be seen in Fig. 2. In the figure, large circles are cells (vertices) of the hypergraph, and small circles are nets. `xpins` and `pins` arrays store the beginning index of pins (cells) connected to each net, and IDs of the pins, respectively. Hence, `xpins` is an array of size equal to the number of nets plus one (11 in this example), and `pins` is an array of size equal to the number of pins in the hypergraph (31 in this example). Cells connected to net n_j are stored in `pins[xpins[j]]` through `pins[xpins[j+1]-1]`.

Stand-Alone Program

Distribution includes a stand-alone program, called `patoh`, for single constraint partitioning (this executable will not work with multiple vertex weights; for multi-constraint partitioning there is an interface and some sample source codes). The program `patoh` gets

its parameters from command line arguments. PaToH can be run from command line as follows:

```
> patoh <hypergraph-file>
  <number-of-parts> [[parameter1]
  [parameter2] ....].
```

Partitioning can be customized by using the optional `[parameter]` arguments. The syntax of these optional parameters is as follows: two-letter abbreviation of a parameter is followed by an equal sign and a value. For example, if the user wishes to change refinement algorithm (abbreviated as "RA") to "Kernighan–Lin with dynamic locking" (sixth algorithm out of 12 implemented in PaToH), the user should specify "RA=6." For a complete example, consider the sample hypergraph displayed in Fig. 2. In order to partition this hypegraph into three parts by using the Kernighan–Lin refinement algorithm with cut-net metric (the default is connectivity metric (Equation (2b)), one has to issue the following command whose output is shown next:

```
> patoh sample.u 3 RA=6 UM=U

+++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++ PaToH v3 (c) Nov 1999-, by  Umit V. Catalyurek
+++ Build # 872 Date: Fri, 09 Oct 2009
+++++++++++++++++++++++++++++++++++++++++++++++++++++++

*****************************************************************************
Hypergraph : sample.u   #Cells :    12   #Nets :    11   #Pins :    31
*****************************************************************************
 3-way partitioning results of PaToH:

 Cut Cost:     2
 Part Weights   : Min=          4 (0.000) Max=          4 (0.000)
-----------------------------------------------------------------------------
I/O          :       0.000 sec
I.Perm/Cons.H:       0.000 sec  ( 2.9%)
Coarsening   :       0.000 sec  ( 1.1%)
Partitioning :       0.000 sec  (75.8%)
Uncoarsening :       0.000 sec  ( 3.7%)
Total        :       0.001 sec
Total (w I/O):       0.001 sec
-----------------------------------------------------------------------------
```

This output shows that the cutsize (cut cost) according to cut-net metric is 2. Final imbalance ratios (in parentheses) for the least loaded and the most loaded parts are 0% (perfect balance with four vertices in each part), and partitioning only took about 1 ms. The input hypergraph and resulting partition is displayed in Fig. 3. A quick summary of the input file format (the details are provided in the PaToH manual [9]) is as follows: the first non-comment line of the file is a header containing the index base (0 or 1) and the size of the hypergraph, and information for each net (only pins in this case) and cells (none in this example) follows.

All of the PaToH customization parameters that are available through library interface are also available as command line options. PaToH manual [9] contains details of each of those customization parameters.

Customizing PaToH's Hypergraph Partitioning

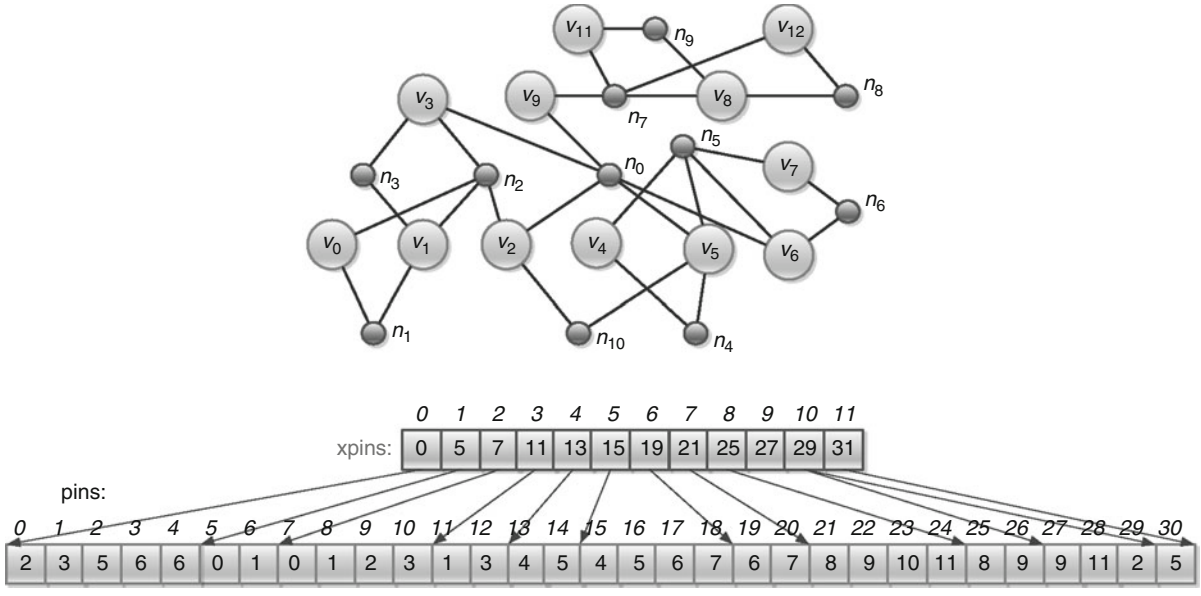
PaToH achieves K-way hypergraph partitioning through recursive bisection (two-way partition), and at each bisection step it uses a multilevel hypergraph bisection

```
#include <stdio.h>
#include "patoh.h"

int main(int argc,char *argv[])
{
PaToH_Parameters args;
int              _c, _n, _nconst, *cwghts, *nwghts,
                 *xpins, *pins, *partvec, cut, *partweights;


PaToH_Read_Hypergraph(argv[1], &_c, &n, &_nconst, &cwghts, &nwghts,
                      &xpins, &pins);

printf("Hypergraph %10s -- #Cells=%6d #Nets=%6d #Pins=%8d #Const=%2d\n",
       argv[1], _c, _n, xpins[_n], _nconst);

PaToH_Initialize_Parameters(&args, PATOH_CONPART, PATOH_SUGPARAM_DEFAULT);

args._k = atoi(argv[2]);
partvec = (int *) malloc(_c*sizeof(int));
partweights = (int *) malloc(args._k*sizeof(int));

PaToH_Alloc(&args, _c, _n, _nconst, cwghts, nwghts, xpins, pins);

if (_nconst==1)
    PaToH_Partition(&args, _c, _n, cwghts, nwghts,
                    xpins, pins, partvec, partweights, &cut);
else
    PaToH_MultiConst_Partition(&args, _c, _n, _nconst, cwghts,
                               xpins, pins, partvec, partweights, &cut);

printf("%d-way cutsize is: %d\n", args._k, cut);

free(cwghts);      free(nwghts);
free(xpins);       free(pins);
free(partweights); free(partvec);

PaToH_Free();
return 0;
}
```

PaToH (Partitioning Tool for Hypergraphs). Fig. 1 A simple C program that partitions an input hypergraph using PaToH functions

algorithm. In the recursive bisection, first a bisection of $\mathcal{H}$ is obtained, and then each part of this bipartition is further partitioned recursively. After $\lg_2 K$ steps, hypergraph $\mathcal{H}$ is partitioned into K parts. Please note that, K is not restricted to be a power of 2. For any $K > 1$, one can achieve K-way hypergraph partitioning through recursive bisection by first partitioning $\mathcal{H}$ into two parts with a load ratio of $\lfloor K/2 \rfloor$ to $(K - \lfloor K/2 \rfloor)$, and then recursively partitioning those parts into $\lfloor K/2 \rfloor$ and $(K - \lfloor K/2 \rfloor)$ parts, respectively, using the same approach.

A pseudo-code of the multilevel hypergraph bisection algorithm used in PaToH is displayed in Algorithm 1. Mainly, the algorithm has three phases: *coarsening*, *initial partitioning*, and *uncoarsening*. In the first phase, a bottom-up multilevel clustering is successively applied starting from the original hypergraph until either the number of vertices in the coarsened hypergraph reduces below a predetermined threshold value or clustering fails to reduce the size of the hypergraph significantly. In the second phase, the coarsest

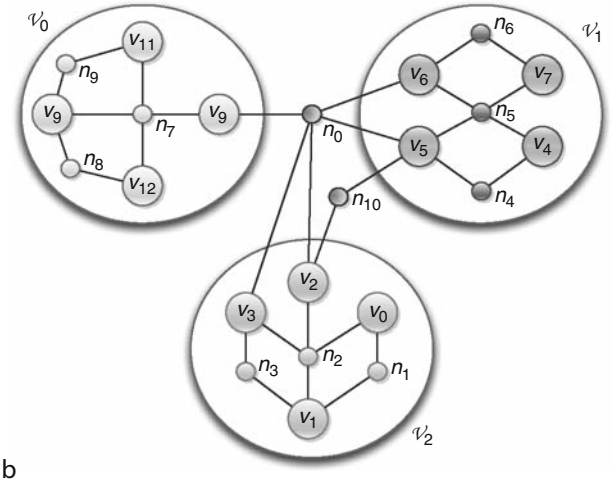


PaToH (Partitioning Tool for Hypergraphs). Fig. 2 A sample hypergraph and its representation

a

```
% base:(0/1) #cells #nets #pins
0 12 11 31
% pins of each net in the hypergraph
2 3 5 6 9
0 1
0 1 2 3
1 3
4 5
4 5 6 7
6 7
8 9 10 11
8 10
8 11
2 5
```

b



PaToH (Partitioning Tool for Hypergraphs). Fig. 3 Text file representation of the sample hypergraph in Fig. 2 and illustration of a partition found by PaToH

hypergraph is bipartitioned using one of the 12 initial partitioning techniques. In the third phase, the partition found in the second phase is successively projected back towards the original hypergraph while it is being improved by one of the iterative refinement heuristics. These three phases are summarized below.

1. Coarsening Phase: In this phase, the given hypergraph $\mathcal{H} = \mathcal{H}_0 = (\mathcal{V}_0, \mathcal{N}_0)$ is coarsened into a sequence of smaller hypergraphs $\mathcal{H}_1 = (\mathcal{V}_1, \mathcal{N}_1)$, $\mathcal{H}_2 = (\mathcal{V}_2, \mathcal{N}_2)$, ..., $\mathcal{H}_\ell = (\mathcal{V}_\ell, \mathcal{N}_\ell)$ satisfying $|\mathcal{V}_0| > |\mathcal{V}_1| > |\mathcal{V}_2| > \ldots > |\mathcal{V}_\ell|$. This coarsening is achieved by coalescing disjoint subsets of vertices of hypergraph $\mathcal{H}_i$ into *clusters* such that each cluster in $\mathcal{H}_i$ forms a single vertex of $\mathcal{H}_{i+1}$. The weight of each vertex of $\mathcal{H}_{i+1}$ becomes equal to the sum of its constituent vertices of the respective cluster in $\mathcal{H}_i$. The net set of each vertex of $\mathcal{H}_{i+1}$ becomes equal to the union of the net sets of the constituent vertices of the respective cluster in $\mathcal{H}_i$. Here, multiple pins of a net $n \in \mathcal{N}_i$ in a cluster of $\mathcal{H}_i$ are contracted to a single pin of the respective net $n' \in \mathcal{N}_{i+1}$ of $\mathcal{H}_{i+1}$. Furthermore, the single-pin

Algorithm 1 Multilevel Bisection.

```
function PaToHMLevelPartition(H = (V,N))
    H_0 ← H
    ℓ ← 0
    /* Coarsening Phase: */
    while |V_ℓ| > CoarseTo do
        find a clustering, C_ℓ, using one of the coarsening
          algorithms
        construct H_{ℓ+1} using C_ℓ
        if (|V_{ℓ+1}| − |V_ℓ|)/|V_ℓ| < CoarsePercent then
            break
        else
            ℓ ← ℓ + 1
        end if
    end while
    /* Initial Partitioning Phase: */
    find an initial partitioning Π_ℓ of H_ℓ
    /* Uncoarsening Phase: */
    while ℓ > 0 do
        refine Π_ℓ using one of the refinement algorithms
        if ℓ > 0 then
            project Π_ℓ to Π_{ℓ−1}
        end if
        ℓ ← ℓ − 1
    end while
    return Π_0
end function
```

nets obtained during this contraction are discarded. The coarsening phase terminates when the number of vertices in the coarsened hypergraph reduces below the predetermined number or clustering fails to reduce the size of the hypergraph significantly.

In PaToH, two types of clusterings are implemented, *matching-based*, where each cluster contains at most of two vertices; and *agglomerative-based*, where clusters can have more than two vertices. The former is simply called *matching* in PaToH, and the latter is called *clustering*.

The matching-based clustering works as follows. Vertices of $\mathcal{H}_i$ are visited in a user-specified order (could be random, degree sorted, etc.). If a vertex $u \in \mathcal{V}_i$ has not been matched yet, one of its unmatched *adjacent* vertices is selected according to a criterion. If such a vertex v exists, the matched pair u and v are merged into a cluster. If there is no unmatched adjacent vertex of u, then vertex u remains unmatched, that is, u remains as a singleton cluster. Here, two vertices u and v are said to be adjacent if they share at least one net, that is, $nets[u] \cap nets[v] \neq \emptyset$.

In the agglomerative clustering schemes, each vertex u is assumed to constitute a singleton cluster $C_u = \{u\}$ at the beginning of each coarsening level. Then, vertices are again visited in a user specified order. If a vertex u has already been clustered (i.e., $|C_u|>1$) it is not considered for being the source of a new clustering. However, an unclustered vertex u can choose to join a multi-vertex cluster as well as a singleton cluster. That is, all adjacent vertices of an unclustered vertex u are considered for selection according to a criterion. The selection of a vertex v adjacent to u corresponds to including vertex u to cluster C_v to grow a new multi-vertex cluster $C_u = C_v = C_v \cup \{u\}$.

PaToH includes a total of 17 coarsening algorithms: eight matchings and nine clustering algorithms, and the default method is a clustering algorithm that uses *absorption* metric. In this method, when selecting the adjacent vertex v to cluster with vertex u, vertex v is selected to maximize $\sum_{n \in nets[u] \cap nets[C_v]} \frac{|C_v \cap n|}{s[n]-1}$, where $nets[C_v] = \cup_{w \in C_v} nets[w]$.

2. Initial Partitioning Phase: The goal in this phase is to find a bipartition on the coarsest hypergraph $\mathcal{H}_\ell$. PaToH includes various random partitioning methods as well as variations of *Greedy Hypergraph Growing (GHG)* algorithm for bisecting $\mathcal{H}_\ell$. In GHG, a cluster is grown around a randomly selected vertex. During the coarse of the algorithm, the selected and unselected vertices induce a bipartition on $\mathcal{H}_\ell$. The unselected vertices connected to the growing cluster are inserted into a priority queue according to their *move-gain* [15], where the gain of an unselected vertex corresponds to the decrease in the cutsize of the current bipartition if the vertex moves to the growing cluster. Then, a vertex with the highest gain is selected from the priority queue. After a vertex moves to the growing cluster, the gains of its unselected adjacent vertices that are currently in the priority queue are updated and those not in the priority queue are inserted. This cluster growing operation continues until a predetermined bipartition balance criterion is reached. The quality of this algorithm is sensitive to the choice of the initial random vertex. Since the coarsest hypergraph $\mathcal{H}_\ell$ is small, initial partitioning heuristics can be run multiple times and select the best bipartition for refinement during the uncoarsening

phase. By default, PaToH runs 11 different initial partitioning algorithms and selects the bipartition with lowest cost.

3. Uncoarsening Phase: At each level i (for $i = \ell, \ell-1, \ldots, 1$), bipartition Π_i found on $\mathcal{H}_i$ is projected back to a bipartition Π_{i-1} on $\mathcal{H}_{i-1}$. The constituent vertices of each cluster in $\mathcal{H}_{i-1}$ is assigned to the part of the respective vertex in $\mathcal{H}_i$. Obviously, Π_{i-1} of $\mathcal{H}_{i-1}$ has the same cutsize with Π_i of $\mathcal{H}_i$. Then, this bipartition is refined by running a KL/FM-based iterative improvement heuristics on $\mathcal{H}_{i-1}$ starting from initial bipartition Π_{i-1}. PaToH provides 12 refinement algorithms that are based on the well-known Kernighan–Lin (KL) [20] and Fiduccia–Mattheyses (FM) [15] algorithms. These iterative algorithms try to improve the given partition by either swapping vertices between parts or moving vertices from one part to other, while not violating the balance criteria. They also provide heuristic mechanisms to avoid local minima. These algorithms operate on passes. In each pass, a sequence of unmoved/unswapped vertices with the highest *gains* are selected for move/swap, one by one. At the end of a pass, the maximum prefix subsequence of moves/swaps with the maximum prefix sum that incurs the maximum decrease in the cutsize is constructed, allowing the method to jump over local minima. The permanent realization of the moves/swaps in this maximum prefix subsequence is efficiently achieved by rolling back the remaining moves at the end of the overall sequence. The overall refinement process in a level terminates if the maximum prefix sum of a pass is not positive.

PaToH includes original KL and FM implementations, hybrid versions, like one pass FM followed by one pass KL, as well as improvements like *multilevel-gain* concept of Krishnamurthy [21] that adds a look-ahead ability, or *dynamic locking* of Hoffman [17], and Dasdan and Aykanat [14] that relaxes vertex moves allowing a vertex to be moved multiple times in the same pass. PaToH also provides heuristic trade-offs, like *early-termination* in a pass of KL/FM algorithms, or *boundary KL/FM*, which only considers vertices that are in the boundary, to speed up the refinement. The default refinement scheme is boundary FM+KL.

Related Entries

►Chaco
►Data Distribution
►Graph Algorithms
►Graph Partitioning
►Hypergraph Partitioning
►Linear Algebra, Numerical
►Preconditioners for Sparse Iterative Methods

Bibliographic Notes and Further Reading

Latest PaToH binary distributions, including recently developed MATLAB interface [26], and related papers can be found on the Web site listed in [9]. The "Hypergraph Partitioning" entry contains some use cases of hypergraph partitioning.

Bibliography

1. Alpert CJ, Kahng AB (1995) Recent directions in netlist partitioning: a survey. VLSI J 19(1–2):1–81
2. Aykanat C, Cambazoglu BB, Uçar B (May 2008) Multi-level direct k-way hypergraphy partitioning with multiple constraints and fixed vertices. J Parallel Distrib Comput 68(5):609–625
3. Aykanat C, Pinar A, Çatalyürek UV (2004) Permuting sparse rectangular matrices into block-diagonal form. SIAM J Sci Comput 26(6):1860–1879
4. Bui TN, Jones C (1993) A heuristic for reducing fill-in sparse matrix factorization. In: Proceedings of the 6th SIAM conference on parallel processing for scientific computing, Norfolk, Virginia, pp 445–452
5. Catalyurek U, Boman E, Devine K, Bozdag D, Heaphy R, Riesen L (Aug 2009) A repartitioning hypergraphy model for dynamic load balancing. J Parallel Distrib Comput 69(8):711–724
6. Çatalyürek UV (1999) Hypergraph models for sparse matrix partitioning and reordering. Ph.D. thesis, Bilkent University, Computer Engineering and Information Science, Nov 1999. http://www.cs.bilkent.edu.tr/tech-reports/1999/ABSTRACTS.1999.html.
7. Çatalyürek UV, Aykanat C (Dec 1995) A hypergraph model for mapping repeated sparse matrix vector product computations onto multicomputers. In: Proceedings of international conference on high performance computing
8. Çatalyürek UV, Aykanat C (1999) Hypergraph-partitioning based decomposition for parallel sparse-matrix vector multiplication. IEEE Trans Parallel Distrib Syst 10(7):673–693
9. Çatalyürek UV, Aykanat C (1999) PaToH: a multilevel hypergraph partitioning tool, version 3.0. Bilkent University, Department of Computer Engineering, Ankara, 06533 Turkey. PaToH. http://bmi.osu.edu/~umit/software.html, 1999 (accessed on November 26, 2010)
10. Çatalyürek UV, Aykanat C (2001) A hypergraph-partitioning approach for coarse-grain decomposition. In: ACM/EEE SC2001, Denver, CO, November 2001
11. Çatalyürek UV, Aykanat C, Kayaaslan E (2009) Hypergraph partitioning-based_ll-reducing ordering. Technical Report

OSUBMI-TR-2009-n02 and BU-CE-0904, The Ohio State University, Department of Biomedical Informatics and Bilkent University, Computer Engineering Department, 2009. submitted for publication
12. Çatalyürek UV, Aykanat C, Ucar B (2010) On two-dimensional sparse matrix partitioning: models, methods, and a recipe. SIAM J Sci Comput 32(2):656–683
13. Cheng C-K, Wei Y-C (1991) An improved two-way partitioning algorithm with stable performance. IEEE Trans Comput Aided Des 10(12):1502–1511
14. Dasdan A, Aykanat C (February 1997) Two novel multiway circuit partitioning algorithms using relaxed locking. IEEE Trans Comput Aided Des 16(2):169–178
15. Fiduccia CM, Mattheyses RM (1982) A linear-time heuristic for improving network partitions. In: Proceedings of the 19th ACM/IEEE design automation conference, pp 175–181
16. Hendrickson B, Leland R (1993) A multilevel algorithm for partitioning graphs. Technical reports, Sandia National Laboratories
17. Hoffmann A (1994) Dynamic locking heuristic – a new graph partitioning algorithm. In: Proceedings of IEEE international symposium on circuits and systems, pp 173–176
18. Karypis G, Kumar V (1998) A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput 20(1):359–392
19. Karypis G, Kumar V (1998) Multilevel algorithms for multi-constraint graph partitioning. Technical Report 98-019, University of Minnesota, Department of Computer Science/Army HPC Research Center, Minneapolis, MN 55455, May 1998
20. Kernighan BW, Lin S (Feb 1970) An efficient heuristic procedure for partitioning graphs. Bell SystTech J 49(2):291–307
21. Krishnamurthy B (May 1984) An improved min-cut algorithm for partitioning VLSI networks. IEEE Trans Comput 33(5):438–446
22. Lengauer T (1990) Combinatorial algorithms for integrated circuit layout. Willey–Teubner, Chichester, UK
23. Sanchis LA (Jan 1989) Multiple-way network partitioning. IEEE Trans Comput 38(1):62–81
24. Schloegel K, Karypis G, Kumar V (2000) Parallel multilevel algorithms for multi-constraint graph partitioning. In: Euro-Par, pp 296–310
25. Schweikert DG, Kernighan BW (1972) A proper model for the partitioning of electrical circuits. In: Proceedings of the 9th ACM/IEEE design automation conference, pp 57–62
26. Uçar B, Çatalyürek ÜV, Aykanat C (2010) A matrix partitioning interface to PaToH in MATLAB. Parallel Comput 36(5–6):254–272
27. WeiY-C,ChengC-K(July1991)Ratiocutpartitioningforhierarchical designs. IEEE Trans Comput Aided Des 10(7):91–921

Partitioning Tool for Hypergraphs (PaToH)

►PaToH (Partitioning Tool for Hypergraphs)

PC Clusters

►Clusters

PCI Express

Jasmin Ajanovic
Intel Corporation, Portland, OR, USA

Synonyms

3GIO; PCI-Express; PCIe; PCI-E

Definition

PCI (Peripheral Component Interconnect) Express is a highly scalable interconnect technology that is the most widely adopted IO interface standard used in the computer and communication industry [2]. By providing scalable speed/width, extendable protocol capabilities, a common configuration/software model, and various mechanical form-factors, PCI Express supports a broad range of applications. It allows implementation of flexible connectivity between a processor/memory complex and an IO subsystems, including peripheral controllers, such as graphics, networking, storage, etc. PCI Express technology development is managed by PCI-SIG (PCI Special Interest Group), an industry association comprising of over 800 member companies.

Discussion

Introduction – A Brief History of PCIe

PCI Express has his roots in Peripheral Component Interconnect (PCI), an open standard specification that was developed by the computing industry in 1992. PCI was a replacement for the ISA bus which was a mainstream PC architecture IO expansion standard at the time. Although there were several alternative solutions, such as MicroChannel, EISA, and VL-bus, that were aiming to replace/supplement ISA, none of them fully addressed the needs of an evolving PC industry. The PCI specification covered both the hardware and software interfaces between PC's CPU/memory complex and add-in cards, such as graphics, network, and disk controllers. One of the most important aspects of PCI was support for the so called "plug-and-play" mechanisms