TAIWANESE JOURNAL OF MATHEMATICS
Vol. 18, No. 1, pp. 93-114, February 2014
DOI: 10.11650/tjm.18.2014.3276
This paper is available online at http://journal.taiwanmathsoc.org.tw

MIXED VOLUME COMPUTATION IN PARALLEL

Tianran Chen, Tsung-Lin Lee[1] and Tien-Yien Li[2]

Abstract. Efficient algorithms for computing mixed volumes, via the computation of mixed cells, have been implemented in DEMiCs [18] and MixedVol-2.0 [13]. While the approaches in those two packages are somewhat different, they follow the same theme and are both highly serial. To fit the need for the parallel computing, a reformulation of the algorithms is inevitable. This article proposes a reformulation of the algorithm for the mixed volume computation rooted from algorithms in graph theory, making it much more fine-grained and scalable. The resulting parallel algorithm can be readily adapted to both distributed and shared memory computing systems. Illustrated by the numerical results on several different architectures, the speedups of our parallel algorithms for the mixed volume computation are remarkable.

1. Introduction

For $j = 1, \ldots, n$, let $\mathcal{S}_j$ be a finite subset of $\mathbb{N}_0^n \equiv (\mathbb{N} \cup \{0\})^n$, and $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$. Let $\mathcal{Q}_j$ be the convex hull of $\mathcal{S}_j$ for $j = 1, \ldots, n$. For positive numbers $\lambda_1, \ldots, \lambda_n$, the n-dimensional volume of the Minkowski sum

$$\lambda_1 \mathcal{Q}_1 + \cdots + \lambda_n \mathcal{Q}_n \equiv \{\lambda_1 \boldsymbol{q}_1 + \cdots + \lambda_n \boldsymbol{q}_n \mid \boldsymbol{q}_j \in \mathcal{Q}_j \ , \ j = 1, \ldots, n\}$$

is a homogeneous polynomial of degree n in the variables $\lambda_1, \ldots, \lambda_n$. The coefficient of the monomial $\lambda_1 \times \cdots \times \lambda_n$ in this polynomial is called the *mixed volume* of $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$, denoted by $\mathcal{M}(\mathcal{S})$.

Besides other applications, the mixed volume computation of $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$ plays a vitally important role in approximating all the isolated zeros of a polynomial

Received May 6, 2013, accepted June 28, 2013.
Communicated by Wen-Wei Lin.
2010 *Mathematics Subject Classification*: 52A39, 65H10, 65H20, 90C05.
Key words and phrases: Mixed volume, Polyhedral homotopy, Parallel computing, Multi-core, Cluster.
[1] Research supported in part by the National Science Council of R.O.C. under Grant NSC 100-2115-M-110-001-MY2.
[2] Research supported in part by NSF under Grant DMS-1115587.

system $P(\boldsymbol{x}) = (p_1(\boldsymbol{x}), \ldots, p_n(\boldsymbol{x}))$ by the homotopy continuation method [7, 12, 14, 15, 16], where, with $\boldsymbol{x} = (x_1, \ldots, x_n) \in \mathbb{C}^n$ and $\boldsymbol{a} = (a_1, \ldots, a_n) \in \mathbb{N}_0^n$,

$$p_j(\boldsymbol{x}) = \sum_{\boldsymbol{a} \in \mathcal{S}_j} c_{j,\boldsymbol{a}} \boldsymbol{x}^{\boldsymbol{a}} \ , \quad j = 1, \ldots, n$$

where $c_{j,\boldsymbol{a}} \in \mathbb{C}^* = \mathbb{C} \setminus \{0\}$ and $\boldsymbol{x}^{\boldsymbol{a}} = x_1^{a_1} \cdots x_n^{a_n}$. Here $\mathcal{S}_j$, a finite subset of $\mathbb{N}_0^n$, is called the *support* of $p_j(\boldsymbol{x})$, and $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$ is called the support of $P(\boldsymbol{x})$. We henceforth call all those $\mathcal{S}_i$'s in the definition of mixed volume given above *supports* in this article.

In 2005, a software package, MixedVol [10] (produced by T. Gao, T.Y. Li and M. Wu), emerged which led then existing codes [7, 9, 17] for mixed volume computation by a substantial margin. Soon after MixedVol was published, T. Mizutani, A. Takeda and M. Kojima [18] developed a more advanced mixed volume computation package, DEMiCs, which considerably overshadowed MixedVol in speed. Later, a revised version of MixedVol, MixedVol-2.0 [13], has reached the speed range of DEMiCs but with much accurate results in many situations in application [13]. In any event, while all those successfully developed algorithms in computing mixed volumes mentioned above have somewhat different approaches, they followed the same theme and are highly serial. In this article, we propose a reformulation of our algorithm in [13] for mixed volume computations rooted from algorithms in graph theory, making it much more fine-grained and scalable. It can be readily adapted to both distributed and shared memory computing systems. Remarkably, very high speed-ups were achieved in our numerical results, and we are now able to compute mixed volume of polynomial systems of very large scale, such as "VortexAC6" [1, 11] system with mixed-volume 27,550,213 and total degree 2^{30} (around 1 billion).

In section 2 we shall outline the basic procedures for serial mixed volume computation, and a parallel reformulation for the algorithm is proposed in section 3. We then present our detailed algorithms for different parallel computation architectures as well as the numerical results in section 4.

2. The Mixed Cell Computation

For each $j = 1, \ldots, n$, let $\mathcal{S}_j$ be a finite subset of $\mathbb{N}_0^n$, and $\omega_j : \mathcal{S}_j \to \mathbb{R}$ be a function with generically chosen images, called a generic *lifting* on $\mathcal{S}_j$. Write

$$\hat{\mathcal{S}}_j = \{\hat{\boldsymbol{a}} = (\boldsymbol{a}, \omega_j(\boldsymbol{a})) | \boldsymbol{a} \in \mathcal{S}_j\}.$$

A collection of pairs $(\{\boldsymbol{a}_1, \boldsymbol{a}_1'\}, \ldots, \{\boldsymbol{a}_n, \boldsymbol{a}_n'\})$ where $\{\boldsymbol{a}_1, \boldsymbol{a}_1'\} \subseteq \mathcal{S}_1, \ldots, \{\boldsymbol{a}_n, \boldsymbol{a}_n'\} \subseteq \mathcal{S}_n$ is called a *mixed cell* if there exists $\hat{\boldsymbol{\alpha}} = (\boldsymbol{\alpha}, 1) \in \mathbb{R}^{n+1}$ such that for each $j = 1, \ldots, n$

$$\langle \hat{\boldsymbol{a}}_j, \hat{\boldsymbol{\alpha}} \rangle = \langle \hat{\boldsymbol{a}}_j', \hat{\boldsymbol{\alpha}} \rangle < \langle \hat{\boldsymbol{a}}, \hat{\boldsymbol{\alpha}} \rangle \quad \text{for all } \hat{\boldsymbol{a}} \in \hat{\mathcal{S}}_j \backslash \{\hat{\boldsymbol{a}}_j, \hat{\boldsymbol{a}}_j'\}.$$

Here $\langle\ ,\ \rangle$ stands for the usual inner product in the Euclidean space. The volume of a mixed cell $(\{\boldsymbol{a}_1, \boldsymbol{a}_1'\}, \ldots, \{\boldsymbol{a}_n, \boldsymbol{a}_n'\})$ is defined to be the volume of the convex hull of the Minkowski sum

$$\{\boldsymbol{a}_1, \boldsymbol{a}_1'\} + \{\boldsymbol{a}_2, \boldsymbol{a}_2'\} + \cdots + \{\boldsymbol{a}_n, \boldsymbol{a}_n'\}\ ,$$

namely, the volume of mixed cell $(\{\boldsymbol{a}_1, \boldsymbol{a}_1'\}, \ldots, \{\boldsymbol{a}_n, \boldsymbol{a}_n'\})$ is $|\det(\boldsymbol{a}_1 - \boldsymbol{a}_1', \boldsymbol{a}_2 - \boldsymbol{a}_2', \ldots, \boldsymbol{a}_n - \boldsymbol{a}_n')|$. It is known [12] that the mixed volume of $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$ equals the sum of volumes of all such mixed cells, i.e.,

$$\mathcal{M}(\mathcal{S}) = \sum_{\boldsymbol{\alpha}} \left|\det(\boldsymbol{a}_1' - \boldsymbol{a}_1, \ldots, \boldsymbol{a}_n' - \boldsymbol{a}_n)\right|$$

where the summation runs through all the possible mixed cells. Thus the mixed volume computation is essentially a direct consequence if all those mixed cells can be found first. On the other hand, those mixed cells play a crucially important role in finding isolated zeros of polynomial systems numerically by the polyhedral homotopy [14, 15, 16]. They provide the starting points of the homotopy paths. We shall therefore concentrate ourselves in the computation of those mixed cells.

To find all the mixed cells induced by a given generic lifting $\boldsymbol{\omega} = (\omega_1, \ldots, \omega_n)$ on $\mathcal{S} = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$, we first construct the "Relation Tables" $T(i, j)$ for $1 \le i \le j \le n$ which display the relationship between elements of $\hat{\mathcal{S}}_i$ and $\hat{\mathcal{S}}_j$ in the following sense:

Given elements $\hat{\boldsymbol{a}}_l^{(i)} \in \hat{\mathcal{S}}_i$ and $\hat{\boldsymbol{a}}_m^{(j)} \in \hat{\mathcal{S}}_j$ does there exist an $\hat{\boldsymbol{\alpha}} = (\boldsymbol{\alpha}, 1) \in \mathbb{R}^{n+1}$ such that

$$\langle \hat{\boldsymbol{a}}_l^{(i)}, \hat{\boldsymbol{\alpha}} \rangle \le \langle \hat{\boldsymbol{a}}^{(i)}, \hat{\boldsymbol{\alpha}} \rangle \quad \text{for all} \quad \boldsymbol{a}^{(i)} \in \mathcal{S}_i \tag{1}$$

and

$$\langle \hat{\boldsymbol{a}}_m^{(j)}, \hat{\boldsymbol{\alpha}} \rangle \le \langle \hat{\boldsymbol{a}}^{(j)}, \hat{\boldsymbol{\alpha}} \rangle \quad \text{for all} \quad \boldsymbol{a}^{(j)} \in \mathcal{S}_j\ ?$$

Here, when $i = j$, then l and m must of course be different. Denote the entry on Table $T(i, j)$ at the intersection of the row containing $\hat{\boldsymbol{a}}_l^{(i)}$ and the column containing $\hat{\boldsymbol{a}}_m^{(j)}$ by $[\hat{\boldsymbol{a}}_l^{(i)}, \hat{\boldsymbol{a}}_m^{(j)}]$ and set $[\hat{\boldsymbol{a}}_l^{(i)}, \hat{\boldsymbol{a}}_m^{(j)}] = 1$ when the answer of Problem (1) is positive, $[\hat{\boldsymbol{a}}_l^{(i)}, \hat{\boldsymbol{a}}_m^{(j)}] = 0$ otherwise. An efficient algorithm to construct such tables was given in [9].

For $1 \le i \le n$, a pair $\hat{e} = \{\hat{\boldsymbol{a}}, \hat{\boldsymbol{a}}'\} \subset \hat{\mathcal{S}}_i$ is called a *lower edge* of $\hat{\mathcal{S}}_i$ if $[\hat{\boldsymbol{a}}, \hat{\boldsymbol{a}}'] = 1$ in the relation table $T(i, i)$. Denote the set of all lower edges of $\hat{\mathcal{S}}_i$ by $L(\hat{\mathcal{S}}_i)$. For k distinct integers $\{i_1, \ldots, i_k\} \subset \{1, \ldots, n\}$,

$$\begin{aligned} &\hat{E}_k := (\hat{e}_{i_1}, \ldots, \hat{e}_{i_k}), 1 \le k \le n, \qquad \text{where} \\ &\hat{e}_{i_j} = \{\hat{\boldsymbol{a}}_{i_j}, \hat{\boldsymbol{a}}_{i_j}'\} \in L(\hat{\mathcal{S}}_{i_j}) \quad j = 1, \ldots, k \end{aligned} \tag{2}$$

$\hat{\mathcal{S}}_i$

		$\hat{\boldsymbol{a}}_2^{(i)}$	$\hat{\boldsymbol{a}}_3^{(i)}$	$\cdots$	$\hat{\boldsymbol{a}}_{\lvert s_i\rvert-1}^{(i)}$	$\hat{\boldsymbol{a}}_{\lvert s_i\rvert}^{(i)}$
	$\hat{\boldsymbol{a}}_1^{(i)}$	$[\hat{\boldsymbol{a}}_1^{(i)}, \hat{\boldsymbol{a}}_2^{(i)}]$	$[\hat{\boldsymbol{a}}_1^{(i)}, \hat{\boldsymbol{a}}_3^{(i)}]$	$\cdots$	$[\hat{\boldsymbol{a}}_1^{(i)}, \hat{\boldsymbol{a}}_{\lvert s_i\rvert-1}^{(i)}]$	$[\hat{\boldsymbol{a}}_1^{(i)}, \hat{\boldsymbol{a}}_{\lvert s_i\rvert}^{(i)}]$
$\hat{\mathcal{S}}_i$	$\hat{\boldsymbol{a}}_2^{(i)}$		$[\hat{\boldsymbol{a}}_2^{(i)}, \hat{\boldsymbol{a}}_3^{(i)}]$	$\cdots$	$[\hat{\boldsymbol{a}}_2^{(i)}, \hat{\boldsymbol{a}}_{\lvert s_i\rvert-1}^{(i)}]$	$[\hat{\boldsymbol{a}}_2^{(i)}, \hat{\boldsymbol{a}}_{\lvert s_i\rvert}^{(i)}]$
				$\ddots$	$\vdots$	$\vdots$
	$\hat{\boldsymbol{a}}_{\lvert s_i\rvert-1}^{(i)}$					$[\hat{\boldsymbol{a}}_{\lvert s_i\rvert-1}^{(i)}, \hat{\boldsymbol{a}}_{\lvert s_i\rvert}^{(i)}]$

Table T(i,i)

$\hat{\mathcal{S}}_j$

		$\hat{\boldsymbol{a}}_1^{(j)}$	$\hat{\boldsymbol{a}}_2^{(j)}$	$\hat{\boldsymbol{a}}_3^{(j)}$	$\cdots$	$\hat{\boldsymbol{a}}_{\lvert s_j\rvert}^{(j)}$
	$\hat{\boldsymbol{a}}_1^{(i)}$	$[\hat{\boldsymbol{a}}_1^{(i)}, \hat{\boldsymbol{a}}_1^{(j)}]$	$[\hat{\boldsymbol{a}}_1^{(i)}, \hat{\boldsymbol{a}}_2^{(j)}]$	$[\hat{\boldsymbol{a}}_1^{(i)}, \hat{\boldsymbol{a}}_3^{(j)}]$	$\cdots$	$[\hat{\boldsymbol{a}}_1^{(i)}, \hat{\boldsymbol{a}}_{\lvert s_j\rvert}^{(j)}]$
$\hat{\mathcal{S}}_i$	$\hat{\boldsymbol{a}}_2^{(i)}$	$[\hat{\boldsymbol{a}}_2^{(i)}, \hat{\boldsymbol{a}}_1^{(j)}]$	$[\hat{\boldsymbol{a}}_2^{(i)}, \hat{\boldsymbol{a}}_2^{(j)}]$	$[\hat{\boldsymbol{a}}_2^{(i)}, \hat{\boldsymbol{a}}_3^{(j)}]$	$\cdots$	$[\hat{\boldsymbol{a}}_2^{(i)}, \hat{\boldsymbol{a}}_{\lvert s_j\rvert}^{(j)}]$
	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$
	$\hat{\boldsymbol{a}}_{\lvert s_i\rvert}^{(i)}$	$[\hat{\boldsymbol{a}}_{\lvert s_i\rvert}^{(i)}, \hat{\boldsymbol{a}}_1^{(j)}]$	$[\hat{\boldsymbol{a}}_{\lvert s_i\rvert}^{(i)}, \hat{\boldsymbol{a}}_2^{(j)}]$	$[\hat{\boldsymbol{a}}_{\lvert s_i\rvert}^{(i)}, \hat{\boldsymbol{a}}_3^{(j)}]$	$\cdots$	$[\hat{\boldsymbol{a}}_{\lvert s_i\rvert}^{(i)}, \hat{\boldsymbol{a}}_{\lvert s_j\rvert}^{(j)}]$

Table T(i,j)

is called a *level-k subface* of $\hat{\mathcal{S}} = (\hat{\mathcal{S}}_1, \ldots, \hat{\mathcal{S}}_n)$ (or simply "level-k subface" when no ambiguities exist) if there exists $\hat{\boldsymbol{\alpha}} = (\boldsymbol{\alpha}, 1) \in \mathbb{R}^{n+1}$ such that for each $j = 1, \ldots, k$

$$\langle \hat{\boldsymbol{a}}_{i_j}, \hat{\boldsymbol{\alpha}} \rangle \;=\; \langle \hat{\boldsymbol{a}}'_{i_j}, \hat{\boldsymbol{\alpha}} \rangle \;\leq\; \langle \hat{\boldsymbol{a}}, \hat{\boldsymbol{\alpha}} \rangle \quad \forall\, \boldsymbol{a} \in \mathcal{S}_{i_j} \backslash \{\boldsymbol{a}_{i_j}, \boldsymbol{a}'_{i_j}\} \;\cdot$$

For a level-k subface $\hat{E}_k = (\hat{e}_{i_1}, \ldots, \hat{e}_{i_k})$ of $\hat{\mathcal{S}} = (\hat{\mathcal{S}}_1, \ldots, \hat{\mathcal{S}}_n)$ with $1 \leq k < n$ and $\hat{e}_{i_j} = \{\hat{\boldsymbol{a}}_{i_j}, \hat{\boldsymbol{a}}'_{i_j}\} \in L(\hat{\mathcal{S}}_{i_j})$ for $j = 1, \ldots, k$, we say lower edge $\hat{e}_{i_{k+1}} = \{\hat{\boldsymbol{a}}_{i_{k+1}}, \hat{\boldsymbol{a}}'_{i_{k+1}}\} \in L(\hat{\mathcal{S}}_{i_{k+1}})$ for certain $i_{k+1} \in \{1, 2, \ldots, n\} \backslash \{i_1, \ldots, i_k\}$ can *extend* level-k subface $\hat{E}_k$ if $\hat{E}_{k+1} := (\hat{e}_{i_1}, \ldots, \hat{e}_{i_{k+1}})$ can be a level-$(k+1)$ subface of $\hat{\mathcal{S}} = (\hat{\mathcal{S}}_1, \ldots, \hat{\mathcal{S}}_n)$. We call $\hat{E}_k$ *extensible* in this situation and $\hat{E}_{k+1}$ is an *extension* of $\hat{E}_k$. Furthermore, we say $\hat{\boldsymbol{b}} \in \hat{\mathcal{S}}_l$ with $l \in \{1, 2, \ldots, n\} \backslash \{i_1, \ldots, i_k\}$ can *extend* $\hat{E}_k$ if there exists $\hat{\boldsymbol{\alpha}} = (\boldsymbol{\alpha}, 1) \in \mathbb{R}^{n+1}$ such that for $j = 1, \ldots, k$

$$\langle \hat{\boldsymbol{a}}_{i_j}, \hat{\boldsymbol{\alpha}} \rangle = \langle \hat{\boldsymbol{a}}'_{i_j}, \hat{\boldsymbol{\alpha}} \rangle \leq \langle \hat{\boldsymbol{a}}, \hat{\boldsymbol{\alpha}} \rangle \quad \text{for all } \boldsymbol{a} \in \mathcal{S}_{i_j} \backslash \{\boldsymbol{a}_{i_j}, \boldsymbol{a}'_{i_j}\} \tag{3}$$

and

$$\langle \hat{\boldsymbol{b}}, \hat{\boldsymbol{\alpha}} \rangle \;\leq\; \langle \hat{\boldsymbol{a}}, \hat{\boldsymbol{\alpha}} \rangle \quad \text{for all } \boldsymbol{a} \in \mathcal{S}_l. \tag{4}$$

Obviously, if $\hat{\boldsymbol{b}} \in \hat{\mathcal{S}}_l$ can not extend $\hat{E}_k$, *i.e.*, the system of inequalities in (3) and (4) is infeasible, then there exists no $\boldsymbol{b}' \in \mathcal{S}_l \backslash \{\boldsymbol{b}\}$ with $\{\hat{\boldsymbol{b}}, \hat{\boldsymbol{b}}'\} \in L(\hat{\mathcal{S}}_l)$ which can extend $\hat{E}_k$ to become a level-$(k+1)$ subface.

Notice that any mixed cell $(\{\boldsymbol{a}_1, \boldsymbol{a}_1'\}, \ldots, \{\boldsymbol{a}_n, \boldsymbol{a}_n'\})$ with $\{\boldsymbol{a}_1, \boldsymbol{a}_1'\} \subseteq \mathcal{S}_1, \ldots, \{\boldsymbol{a}_n, \boldsymbol{a}_n'\} \subseteq \mathcal{S}_n$ gives rise to a level-n subface of $\hat{\mathcal{S}} = (\hat{\mathcal{S}}_1, \ldots, \hat{\mathcal{S}}_n)$ and vise versa. Therefore a main strategy for finding mixed cells is the extension of the subfaces of $\hat{\mathcal{S}} = (\hat{\mathcal{S}}_1, \ldots, \hat{\mathcal{S}}_n)$ from level-1 to level-2, ..., etc., until level-n subfaces are reached. So, we pick an appropriate $\hat{\mathcal{S}}_{i_1}$ with $i_1 \in \{1, \ldots, n\}$ as our point of departure. From Table $T(i_1, i_1)$, those pairs $\{\boldsymbol{a}_{i_1},\ \boldsymbol{a}_{i_1}'\} \subseteq \mathcal{S}_{i_1}$ with $[\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}_{i_1}'] = 1$ are the only possible level-1 subfaces in $\hat{\mathcal{S}}_{i_1}$. Taking a fixed pair $\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}_{i_1}'\}$ among them as our level-1 subface, we search among $\{\hat{\mathcal{S}}_l : l \in \{1, \ldots, n\} \backslash \{i_1\}\}$ for the support which has minimal number of points that can possibly extend the level-1 subface $\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}_{i_1}'\}$. This support will be our $\hat{\mathcal{S}}_{i_2}$. To search for such support, points in each support $\hat{\mathcal{S}}_l$ with $l \neq i_1$ which can not extend $\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}_{i_1}'\}$ would be removed. The techniques for finding those removable points, which essentially check the feasibility of the inequalities in (3) and (4) systematically and skillfully, strongly dictate the efficiency of the searching process. The details can be found in [13, 18].

Suppose the selected $\hat{\mathcal{S}}_{i_2}$ contains the remaining points $\hat{\boldsymbol{b}}_1, \ldots, \hat{\boldsymbol{b}}_\ell$. The process of finding all the pairs among them that can extend $\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}_{i_1}'\}$ as in [9, 13] is outlined below:

For each $i = 1, \ldots, \ell$, consider the **One-Point test** on $\hat{\boldsymbol{b}}_i$:

$$\begin{aligned} &\text{Minimize} \quad \langle \hat{\boldsymbol{b}}_i, \hat{\boldsymbol{\alpha}} \rangle - \alpha_0 \\ (5) \qquad &\langle \hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{\alpha}} \rangle = \left\langle \hat{\boldsymbol{a}}_{i_1}', \hat{\boldsymbol{\alpha}} \right\rangle \leq \langle \hat{\boldsymbol{a}}, \hat{\boldsymbol{\alpha}} \rangle \quad \forall \boldsymbol{a} \in \mathcal{S}_{i_1} \\ &\alpha_0 \leq \langle \hat{\boldsymbol{b}}_k, \hat{\boldsymbol{\alpha}} \rangle \quad \forall k = 1, \ldots, \ell \end{aligned}$$

in the variables $\boldsymbol{\alpha} \in \mathbb{R}^n$ in $\hat{\boldsymbol{\alpha}} = (\boldsymbol{\alpha}, 1) \in \mathbb{R}^{n+1}$ and $\alpha_0 \in \mathbb{R}$. Apparently, when the optimal value of this **LP** (Linear Programming) problem is zero, the point $\hat{\boldsymbol{b}}_i$ is a vertex belonging to certain lower edges of $\hat{\mathcal{S}}_{i_2}$ and some of those lower edges may extend $\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}_{i_1}'\}$. Otherwise, $\hat{\boldsymbol{b}}_i$ would play no role in any pairs in $\hat{\mathcal{S}}_{i_2}$ that can extend $\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}_{i_1}'\}$ and therefore it can be safely removed. We shall use the notation $I((\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}_{i_1}'\}), \hat{\boldsymbol{b}}_i)$ to denote this One-Point test with $(\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}_{i_1}'\})$ being its *stem*, and set $I((\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}_{i_1}'\}), \hat{\boldsymbol{b}}_i)$ to be positive when the optimal value of this **LP** problem is zero. In short, those $\hat{\boldsymbol{b}}_i$'s with positive $I((\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}_{i_1}'\}), \hat{\boldsymbol{b}}_i)$'s are points in $\hat{\mathcal{S}}_{i_2}$ that can extend $\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}_{i_1}'\}$.

As explained in detail in [9], an important feature here is that one never needs to solve those corresponding **LP** problems for *all* $\hat{\boldsymbol{b}}_i$'s when the simplex method is used to solve (5), because the information generated by the simplicial pivoting in the simplex method already provides answers to many of the **LP** problems with respect to other $\hat{\boldsymbol{b}}_j$'s.

Let $\hat{\boldsymbol{b}}_{j_1}, \ldots, \hat{\boldsymbol{b}}_{j_\mu}$ be the remaining points in $\hat{\mathcal{S}}_{i_2}$ on which the above One-Point tests $I((\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}),$
$\hat{\boldsymbol{b}}_i)$'s are positive. We now fix $\hat{\boldsymbol{b}}_{j_1}$ first and for each $\ell = 2, \ldots, \mu$ consider the **One-Point test**

$$\text{(6)} \qquad \begin{aligned} &\text{Minimize} \quad \langle \hat{\boldsymbol{b}}_{j_\ell}, \hat{\boldsymbol{\alpha}} \rangle - \alpha_0 \\ &\langle \hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{\alpha}} \rangle = \left\langle \hat{\boldsymbol{a}}'_{i_1}, \hat{\boldsymbol{\alpha}} \right\rangle \le \langle \hat{\boldsymbol{a}}, \hat{\boldsymbol{\alpha}} \rangle \quad \forall \boldsymbol{a} \in \mathcal{S}_{i_1} \\ &\alpha_0 = \langle \hat{\boldsymbol{b}}_{j_1}, \hat{\boldsymbol{\alpha}} \rangle \le \langle \hat{\boldsymbol{b}}_{j_k}, \hat{\boldsymbol{\alpha}} \rangle \quad \forall k = 2, \ldots, \mu \end{aligned}$$

in the variables $\boldsymbol{\alpha} \in \mathbb{R}^n$ in $\hat{\boldsymbol{\alpha}} = (\boldsymbol{\alpha}, 1) \in \mathbb{R}^{n+1}$ and $\alpha_0 \in \mathbb{R}$. It is clear that only zero optimal value for this **LP** problem allows the pair $\{\hat{\boldsymbol{b}}_{j_1}, \hat{\boldsymbol{b}}_{j_\ell}\}$ to extend $\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}$ and $(\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}, \{\hat{\boldsymbol{b}}_{j_1}, \hat{\boldsymbol{b}}_{j_\ell}\})$ will become a level-2 subface of $\hat{\mathcal{S}} = (\hat{\mathcal{S}}_1, \ldots, \hat{\mathcal{S}}_n)$. We use $I((\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}, \{\hat{\boldsymbol{b}}_{j_1}\}), \hat{\boldsymbol{b}}_{j_\ell})$ to denote this One-Point test with $(\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}, \{\hat{\boldsymbol{b}}_{j_1}\})$ being its *stem*, and set $I((\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}, \{\hat{\boldsymbol{b}}_{j_1}\}), \hat{\boldsymbol{b}}_{j_\ell})$ to be positive when the optimal value of this **LP** problem is zero. In short, positive one point test $I((\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}, \{\hat{\boldsymbol{b}}_{j_1}\}), \hat{\boldsymbol{b}}_{j_\ell})$ permits $\{\hat{\boldsymbol{b}}_{j_1}, \hat{\boldsymbol{b}}_{j_\ell}\}$ to extend $\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_2}\}$. Again there is no need to solve all those **LP** problems with respect to all individual $\hat{\boldsymbol{b}}_{j_\ell}$ when the simplex method is used to solve (6) [9]. The same procedure may be repeated on fixing $\hat{\boldsymbol{b}}_{j_2}, \hat{\boldsymbol{b}}_{j_3}, \ldots$ etc., to find all the possible pairs among $\{\hat{\boldsymbol{b}}_{j_1}, \ldots, \hat{\boldsymbol{b}}_{j_\mu}\}$ that can extend $\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}$. Of course, if we fail to find any pairs in any $\hat{\mathcal{S}}_l$, $l \in \{1, \ldots, n\} \backslash \{i_1\}$ that can extend $\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}$, then the whole process will restart on different level-1 subfaces in $\hat{\mathcal{S}}_{i_1}$.

In general, to extend a level-k subface $\hat{E}_k = (\hat{e}_{i_1}, \ldots, \hat{e}_{i_k})$ of $\hat{\mathcal{S}} = (\hat{\mathcal{S}}_1, \ldots, \hat{\mathcal{S}}_n)$ with $1 \le k < n$ and $\hat{e}_{i_j} = \{\hat{\boldsymbol{a}}_{i_j}, \hat{\boldsymbol{a}}'_{i_j}\} \in L(\hat{\mathcal{S}}_{i_j})$ for $j = 1, \ldots, k$, we search among $\{\hat{\mathcal{S}}_l : l \in \{1, \ldots, n\} \backslash \{i_1, \ldots, i_k\}\}$ for the support having minimal number of points that can possibly extend $\hat{E}_k = (\hat{e}_{i_1}, \ldots, \hat{e}_{i_k})$. This support will be our $\hat{\mathcal{S}}_{i_{k+1}}$. (Note that for different level-k subface $\hat{E}_k = (\hat{e}_{i_1}, \ldots, \hat{e}_{i_k})$ we may find different $\hat{\mathcal{S}}_{i_{k+1}}$.) Let $\hat{\boldsymbol{c}}_1, \ldots, \hat{\boldsymbol{c}}_m$ be the points in $\hat{\mathcal{S}}_{i_{k+1}}$ that can extend $\hat{E}_k = (\hat{e}_{i_1}, \ldots, \hat{e}_{i_k})$. By consecutive one point tests $I((\hat{e}_{i_1}, \ldots, \hat{e}_{i_k}), \hat{\boldsymbol{c}}_i)$ and $I((\hat{e}_{i_1}, \ldots, \hat{e}_{i_k}), \{\hat{\boldsymbol{c}}_{j_1}\}), \hat{\boldsymbol{c}}_{j_\ell})$, defined similarly as in (5) and (6) respectively, we find all the pairs in $\hat{\mathcal{S}}_{i_{k+1}}$ that can extend level-k subface $\hat{E}_k = (\hat{e}_{i_1}, \ldots, \hat{e}_{i_k})$ to become a level-($k+1$) subface.

3. The Parallel Reformulation of Mixed Cell Computation

At present, the most efficient algorithms for computing mixed volumes, via computing mixed cells, have been implemented in DEMiCs [18] and MixedVol-2.0 [13]. While approaches in those two packages are somewhat different, they follow the same theme as described in the last section. As we can see, they are apparently very serial. For the need of parallel computing, a reformulation of the algorithm is inevitable.

The reformulation we proposed is rooted from algorithms in graph theory. First note that in the original algorithm for computing mixed cells while some of the One-Point tests are closely related, most of the other One-Point tests are nearly independent. We will group together all those One-Point tests having the same stem and call it a *task*. Namely, a task is a series of One-Point tests of the form (5) or (6). For example, all One-Point tests with stem $(\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}, \{\hat{\boldsymbol{b}}_{i_2}, \hat{\boldsymbol{b}}'_{i_2}\})$ will be grouped together to form a task, denoted by $I((\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}, \{\hat{\boldsymbol{b}}_{i_2}, \hat{\boldsymbol{b}}'_{i_2}\}), *)$. Such tasks will be our smallest units of computation around which the algorithm is designed.

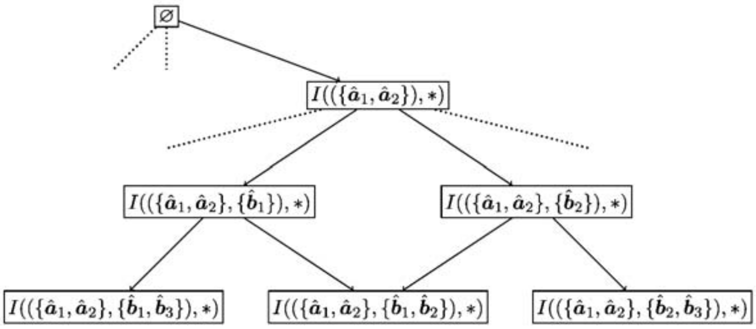

∅
$I((\{\hat{\boldsymbol{a}}_1, \hat{\boldsymbol{a}}_2\}), *)$
$I((\{\hat{\boldsymbol{a}}_1, \hat{\boldsymbol{a}}_2\}, \{\hat{\boldsymbol{b}}_1\}), *)$
$I((\{\hat{\boldsymbol{a}}_1, \hat{\boldsymbol{a}}_2\}, \{\hat{\boldsymbol{b}}_2\}), *)$
$I((\{\hat{\boldsymbol{a}}_1, \hat{\boldsymbol{a}}_2\}, \{\hat{\boldsymbol{b}}_1, \hat{\boldsymbol{b}}_3\}), *)$
$I((\{\hat{\boldsymbol{a}}_1, \hat{\boldsymbol{a}}_2\}, \{\hat{\boldsymbol{b}}_1, \hat{\boldsymbol{b}}_2\}), *)$
$I((\{\hat{\boldsymbol{a}}_1, \hat{\boldsymbol{a}}_2\}, \{\hat{\boldsymbol{b}}_2, \hat{\boldsymbol{b}}_3\}), *)$

Fig. 1. Graph of tasks.

Those tasks are interconnected in such a way that they form a *directed acyclic graph* or DAG, whose vertices are the tasks and edges between vertices are given by the natural *extension* relation between subfaces of the tasks as elaborated in the last section. In this way, we establish a graph representation of the totality of the One-Point tests. If we further associate the actual computation of each One-Point test with the act of visiting its corresponding vertex in such a task graph, then we can equate the mixed cell computation problem, i.e., the totality of all the One-Point tests, to the graph traversal problem which can usually be handled by graph traversal algorithms.

We are now guided by the following questions:

1. How does a generic graph traversal algorithm work?
2. How does the graph traversal algorithm map to our mixed cell computation problem?
3. How can the special structure of the problem allow us to reduce the total amount of computation?

3.1. Generic graph traversal algorithms

While this class of algorithms is well-known, we shall give a brief discussion of the algorithm below since it forms the basis of our parallel reformulation. Most graph traversal algorithms follow a “discover-explore” procedure with proper book keeping [24]. The key idea is to gradually explore the graph vertex by vertex through the connection between them. For a single vertex, such an algorithm is divided into *discover* and *explore* stages: a vertex is first discovered, and then its connections to other yet unknown vertices are explored. Clearly, each vertex only needs to be visited once. That is, one only needs to explore a *spanning tree* of the graph, a subgraph that contains all the vertices but is a tree in structure, so some mechanism must be used to prevent a vertex from being visited twice. To keep track of the vertices as they are being visited, each task is assigned a dynamic marker – its state. A vertex can be in one of the following three states:

undiscovered The initial status of every vertex. In this state, the existence of the vertex is completely unknown to us.

discovered The existence of the vertex is known, but its connections to other vertices are not yet explored.

completely-explored The existence of the vertex is known and its connections to other vertices have been fully explored.

Obviously, a vertex cannot be *completely-explored* before it is first discovered, so in the course of the algorithm, the state of vertices progresses from *undiscovered* to *discovered* to *completely-explored*. This point of view also reveals the parallelism in such algorithms: vertices on different branches of the spanning tree can be explored in parallel, while consecutive vertices on a single branch must be discovered and explored in order. To start the algorithm, an initial set of vertices are generated by some other means (bootstrapping). The algorithm then discovers other vertices through their edges. From these newly discovered vertices the algorithm can discover further more vertices. This will continue as a self-sustaining process until all connected vertices are visited. A complete algorithm also needs a data structure to keep track of the discovered but not yet completely explored vertices (bookkeeping). The detail of this class of algorithms can be found in standard textbooks such as [24].

3.2. Mapping generic graph traversal algorithms to mixed cell computation problem

The reformulation of our mixed cell computation algorithm will follow the standard scheme of graph traversal algorithms stated above with each task corresponds to a vertex in the graph. The states for vertices map, naturally, to that of tasks:

undiscovered The task in its initial state. In this state, the existence of the task is completely unknown to us. For example, before any tests are performed, for any pair $\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\} \in L(\hat{\mathcal{S}}_{i_1})$ the task $I(\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}, *)$ would be considered undiscovered, as their existence are completely unknown to us.

discovered The task after we have first encountered it and before exploring the possibility of extending its stem. For example, once we have found $\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}$ in $L(\hat{\mathcal{S}}_{i_1})$, the task $I((\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}), *)$ is considered to be discovered.

completely-explored The task after we have examined, via One-Point tests, all the possible ways in which its stem can be extended. For example, after all the One-Point tests on the task $T = I((\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}), *)$ have been performed and resulting newly discovered tasks are recorded, we will consider T to be completely-explored. For all practical purposes, we do not need to visit T again.

As in the generic graph traversal algorithm, the state of tasks progresses from *undiscovered* to *discovered* to *completely-explored*. For the bookkeeping part, it is clear that one simply does not need to keep track of tasks with unknown or explored status. Only the set of discovered tasks, i.e., tasks that are waiting to be explored, needs to be remembered. So the algorithm shall maintain a dynamic pool of *discovered* tasks: the *task pool*. A *priority queue* Q will be used as the abstract data structure for the task pool in which each element has a priority. It supports at least two primary operations:

enqueue(Q, x) Given an element x with implicitly defined priority, this operation inserts x into the priority queue Q.

dequeue(Q) As long as $Q \neq \varnothing$, this operation removes one element $x \in Q$ with the highest priority and returns x as the result to the caller of this operation. This operation is usually required about constant time, i.e., in $O(1)$.

The relative order of priority of tasks in a task pool determines the order in which they will be extracted by the *dequeue* operation. In the context of our algorithm, the priority determines the order in which tasks will be further explored. Two simplest choices are giving elements monotonically increasing or decreasing priority. When the increasing priority rule is used, our algorithm conducts the *depth first search* or DFS graph traversal algorithm [6, 24] in the sense that tasks are on a deeper level, i.e., those with longer stems will be explored first. On the other hand, when the decreasing priority is used, the algorithm conducts the behavior of the *breadth first search* or BFS graph traversal algorithm [24]. In the preliminary implementation of our parallel algorithm, we have found that while the monotone decreasing priority assignment tends to explore parallelism quickly, it often discovers tasks so quickly that the size of the task pool exceeds the capacity of the main memory of the computer. On the other hand, the monotone increasing priority assignment is generally helpful in keeping the size of the task pool relatively small. However, it is very frequently the case that the number of discovered tasks at a single point of time is less than the number of processors available, and as a result, some processors would idle. Therefore, a dynamic combination of the two priority assignment schemes is employed in our algorithm. The priority queue for the task pool always start with the monotone decreasing priority assignment. The algorithm keeps monitoring the size of the task pool, and when the size of the task pool exceeds a predetermined constant multiple of the number of threads, then the priority assignment for the priority queue is reversed, i.e., the monotone increasing order is used instead.

For the bootstrapping part, the relation-table computation produces useful by-products of the lower edges $L(\hat{\mathcal{S}}_{i_k})$ of each support. Once the index i_1 of the first support is determined, we consider tasks with stems in $L(\hat{\mathcal{S}}_{i_1})$ to be discovered.

3.3. The special structures of the mixed cell computation problem

For a large polynomial system where n is relatively big and supports $\mathcal{S}_i,\ i = 1,\dots,n$ contain many points, the induced graph can be unmanageably large. However, one key observation is that unlike the situation for a general graph traversing problem our graph contains many redundant vertices and there is no need to visit such vertices. For instance, if we already know the One-Point test $I(\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}, \{\hat{\boldsymbol{b}}_{i_2}\})$ is negative, it is clear from the systems of inequalities (5) and (6) that any tasks with $(\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}, \{\hat{\boldsymbol{b}}_{i_2}\})$ as part of its stem can be safely ignored, as the corresponding One-Point tests must also be negative. In the graph-theoretic language, once an One-Point test from a task is known to be negative, any subtree of the graph rooted from that One-Point test can be completely ignored, leaving only a small fraction of the tasks to be explored. Essentially important is to identify those vertices at $2n$-depth in the graph, which can be considered as mixed cells.

Another potential problem for very large systems is the computational cost for keeping track of the set of already discovered tasks could grows quickly. One solution is to apply extra constraints to the graph, so that it can be reduced to a tree structure. For instance, if we require the existence of the edge between $I((\{\hat{\boldsymbol{a}}_i\}), *)$ and $I((\{\hat{\boldsymbol{a}}_i, \hat{\boldsymbol{a}}'_j\}), *)$ implying $i < j$, then as shown in Figure 2 the edge from $I((\{\hat{\boldsymbol{a}}_1, \hat{\boldsymbol{a}}_2\}, \{\hat{\boldsymbol{b}}_2\}), *)$ to $I((\{\hat{\boldsymbol{a}}_1, \hat{\boldsymbol{a}}_2\}, \{\hat{\boldsymbol{b}}_1, \hat{\boldsymbol{b}}_2\}), *)$ in Figure 1 disappeared by this requirement. We can see that the set of tasks together with the now reduced set of edges actually form a tree structure instead of a more general graph structure. Since a vertex in a tree structure has at most one incoming edge, this reduced structure frees us from explicitly keeping track of the already discovered tasks. In our actual experiments, we found that this technique is generally beneficial for the parallel computation of very large systems.

3.4. The parallel algorithm

Initially, only tasks with stems in $L(\hat{\mathcal{S}}_{i_1})$ are considered to have been discovered. To completely explore a task, say $I((\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}), *)$, we shall perform One-Point test $I((\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}), \hat{\boldsymbol{b}}_j)$ for each $\hat{\boldsymbol{b}}_j \in \hat{\mathcal{S}}_{i_2}$ for the chosen i_2 that may lead to still *undiscovered* tasks, making them discovered. For each One-Point test as in (6) that gives a positive result, say $I((\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}), \hat{\boldsymbol{b}}_j)$, we considered the new task $I((\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}, \{\hat{\boldsymbol{b}}_j\}), *)$ as being *discovered*, and we shall placed this newly discovered task in the task pool to be explored later. After One-Point test is performed on each $\hat{\boldsymbol{b}}_j \in \hat{\mathcal{S}}_{i_2}$, we shall mark the original task $I(\{\hat{\boldsymbol{a}}_{i_1}, \hat{\boldsymbol{a}}'_{i_1}\}, *)$ as *completely-explored*.

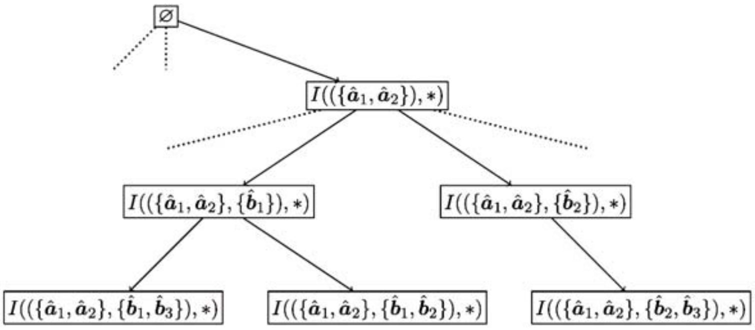

∅
$I((\{\hat{a}_1, \hat{a}_2\}), *)$
$I((\{\hat{a}_1, \hat{a}_2\}, \{\hat{b}_1\}), *)$
$I((\{\hat{a}_1, \hat{a}_2\}, \{\hat{b}_2\}), *)$
$I((\{\hat{a}_1, \hat{a}_2\}, \{\hat{b}_1, \hat{b}_3\}), *)$
$I((\{\hat{a}_1, \hat{a}_2\}, \{\hat{b}_1, \hat{b}_2\}), *)$
$I((\{\hat{a}_1, \hat{a}_2\}, \{\hat{b}_2, \hat{b}_3\}), *)$

Fig. 2. Tree of tasks

We continue the algorithm by repeatedly fetch a single already *discovered* task from the task pool and explore it by performing a series of One-Point tests. This *fetch-and-explore* procedure continues until the task pool is empty and there is no tasks that are currently being explored. At this point all the mixed cells should have been obtained, and the algorithm terminates. This part of the algorithm is described by the pseudo-code listed in Algorithm 1, in which eliminate$(L(\hat{\mathcal{S}}_l))$ refers to the process of eliminating edges using the techniques stated in [13, 18]. It will be the basic building block with which we construct all of our parallel algorithms. When multiple execution units (called *threads* in shared-memory model or *processes* in distributed-memory model) run simultaneously, each simply executes such a *fetch-and-explore* procedure repeatedly and thus they can all execute independently from one another as long as one can maintain the pool of *discovered* tasks in a *concurrent data structure*.

4. Mixed Cell Computation in Parallel

4.1. An overview of the parallel computing architectures

Different parallel computing systems nowadays form a wide spectrum of architectures. On one end of the spectrum, one can find super-scalar processors in which parallelism is exploited at an instruction level. On the other extreme side, there are distributed computing environments consisting of independent computers. It is virtually impossible to cater to the specific challenges of every possible architecture, nor is it likely that we can pay close attention to every characteristic of an architecture. Nonetheless two characteristics will have great influence on our design decisions. First, the memory organization decides how processing units can communicate with one another. Memory can be shared, in which case processing units can access the memory of one another, and writing as well as reading of the shared memory would be the main means of communication among them. When memory is not shared, each processing unit can only access its own local memory, and some other means of communication, such as message passing, must be used to communicate with other processing units. Second, the coupling among processing units will influence the way we organize different units. The word “coupling”, in this context, refers to the degree to which the processing units can interact with one another in a reliable way. On one hand, if the channel of communications among processing units is direct, fast, and reliable, we say they are tightly coupled. On the other hand, when the channel is indirect, slow or unreliable, we say they are loosely coupled.

Algorithm 1. ParallelExtend(Q)

```
Input: A priority queue $Q$ of tasks
Output: A set of mixed cells of $S = (S_1, \ldots, S_n)$
    $M \leftarrow \varnothing$                                  # the set M will contain the mixed cells
    while $Q \neq \varnothing$ do
      $\hat{E}_k = (e_{i_1}, \ldots, e_{i_k}) \leftarrow$ dequeue($Q$)   # where $\hat{e}_{i_j} \subset \hat{S}_{i_j}$
      $R_k \leftarrow \{1, \ldots, n\} \backslash \{i_1, \ldots, i_k\}$       # indices of the remaining supports
      for all $l \in R_k$ do
        $L_l \leftarrow$ eliminate($L(\hat{\mathcal{S}}_l)$)             # let $L_l$ be the remaining edges of $L(\hat{\mathcal{S}}_l)$
        $r_l \leftarrow |L_l|$                           # the number of remaining edges
      end for
      $m_{k+1} \leftarrow \min\{r_l\}_{l \in R_k}$                  # find the minimum number of remaining edges
      if $m_{k+1} > 0$ then
        $i_{k+1} \leftarrow \min\{l \mid r_l = m_{k+1}\}$          # pick next support to be one of that size
        for all $\hat{e}_{i_{k+1}} \in L_{i_{k+1}}$ do
          if $I(\hat{E}_k, \hat{e}_{i_{k+1}})$ is positive then
            if $k + 1 = n$ then
              $M \leftarrow M \cup \{(\hat{E}_k, \hat{e}_{i_{k+1}})\}$     # obtained a mixed cell
            else
                enqueue $(Q, (\hat{E}_k, \hat{e}_{i_{k+1}}))$      # obtained a mixed cell
            end if
            end if
        end for
     end if
    end while
    return $M$                                  # return the mixed cells
```

Based on these two criteria, we restrict our attention to three major classes of parallel architectures, and tailor our algorithms to these architectures. First, we deal with the shared-memory systems, the type of systems in which some form of shared memory is accessible to all processing units and the processors are tightly coupled in the sense that the channels of communication between them are direct, fast and very reliable. A typical example is the traditional SMP (symmetric multi-processor) architecture. We then face the distributed-memory single systems. They are computer systems in which memory are not shared among processing units, but processing units are connected by slightly slower but still relatively reliable networks so that, to a certain extend, they still form a single computer system. High performance computer clusters are examples of this class. Finally, we experiment on distributed environments in which memory are not shared among processing units, much like the previous class. However, processing units in such an environment are very loosely coupled, in the sense that the connection between them are very slow and even unreliable, and, more importantly, processing units may enter and exit the environment freely. For instance, a processing unit can become connected and ready to perform tasks in the middle of a computation process, or a unit may stop calculation or even fail suddenly, without any warning. To program in such an environment, we must be aware of the loose-coupling among processing units. The differences among the three major classes are summarized in the table below.

Class	Shared-Memory	Distributed-Memory single system	Distributed environment
Memory organization	shared	not shared	not shared
Coupling	tight	loose	very loose

These three classes are not intended to be all-inclusive nor mutually exclusive. Stream processors (used to construct GPUs), for example, are not included in the above classes. We are, however, excited to see the new trend of using GPUs in general computation, and we are looking forward to facing the challenge as they become more mature. That being said, majority of the parallel computers one may encounter belong to one of these classes. Although the tight coupling between different tasks seem to suggest that our parallel formulation is designed for shared-memory systems, in the following subsections we shall show that with careful implementation with respect to different hardware architectures, our algorithm can actually achieve remarkable performance on a wide range of architectures.

4.2. Shared-Memory systems

Typical examples of shared-memory systems include the symmetric multi-processing (SMP), the multi-core processors, and NUMA architectures. In an SMP architecture, all processors share a single connection to a common memory and access it at similar rates. Chip multiprocessors, also known as multi-core processors, involves more than one processor physically placed on a single chip package and often share some components. Multi-core processors can be considered as an extreme form of tightly-coupled SMP. Today, SMP and the multi-core processor architectures are very common, partly because they are efficient and easy to build. However they are not scalable to a large number of processing units, as the common connection between the shared memory and all the processors/cores may become a bottleneck. Non-Uniform Memory Access, or NUMA, architectures, are designed to overcome this very problem. The name, NUMA, refers to the fact that a processor can access memory residing on its own node much faster than it can access those residing on other nodes. Logically, programming a NUMA is no different than programming an SMP system. However, care must be taken when planning the data layout if one wishes to obtain the best performance on a NUMA system.

Despite the difference in organization, all shared-memory architectures allow fast and direct communication between processing units via writing and reading of the shared memory. As a result, in all shared-memory systems, a threading model is used in designing our parallel algorithm. In such a model, a *thread* is the smallest unit of execution, and multiple threads, each runs on an independent processing unit, can cooperate via the shared memory. With the shared memory efficiently accessible by all threads, in our algorithm a single and global task pool is placed in the shared memory and is used by all threads. Each thread simply executes Algorithm 1 stated in the last section. We shall show the parallel performance of our algorithm on typical multi-core and NUMA architectures. The algorithm is implemented using the standard Pthreads library. We have also used a more modern but less portable library, the Intel TBB (Thread Building Blocks, an open source threading library originally developed by Intel [23]), wherever it is available.

One of the biggest challenges one must face in shared-memory systems, which does not exist in the distributed-memory systems, is the need to handle *race conditions* caused by multiple threads accessing same memory locations at the same time. The task pool, for example, should, in principle, allow concurrent access from multiple threads, but such an access pattern will almost certainly cause race conditions. Thus some special mechanism must be in place to shield the task pool against such problems. In our Pthreads-based implementation, the standard *mutex* (or *futex* in Linux) is used to safeguard the shared task pool. But when Intel TBB is available on our target platform, we make use of the more efficient concurrent data structures provided by TBB itself. Not only does it free us from implementing our own concurrent task pool, it is also generally more efficient. So in the testings listed below, the TBB-based version is used.

4.2.1. Mixed volume computation on multi-core architecture

Our algorithm is applied in finding mixed volumes of a group of standard benchmark polynomial systems for numerical testing. Using TBB, nearly n-fold linear speedups scalable up to 12 processor cores have been achieved as shown in the table below. From the uniform speedups with respect to different number of processor cores, it is expected that the same range of speedups can be reached with a large amount of processor cores.

Here T_{serial} represents the absolute amount of time consumed to compute the mixed volumes using only one processor core, whereas the parallel speedup ratio using p

processor cores is given by the ratio T_{serial}/T_p in which T_p is the amount of time consumed using p threads.

Systems	Mixed Volume	T_{serial}	Parallel speedup ratio using p threads					
			$p=2$	$p=4$	$p=6$	$p=8$	$p=10$	$p=12$
cyclic15 [3]	35,243,520	8.4 hr	1.95	3.90	5.85	7.78	9.66	11.50
eco20 [19]	262,144	3.1 hr	1.97	3.94	5.90	7.86	9.81	11.75
katsura15 [4]	32,730	48 min	1.96	3.94	5.90	7.86	9.82	11.76
noon21 [20]	10,460,353,161	54 min	1.97	3.93	5.88	7.82	9.72	11.60
sonic8 [13]	6,533,120	1.3 hr	1.98	3.97	5.94	7.92	9.89	11.87

4.2.2. Mixed volume computation on NUMA architectures

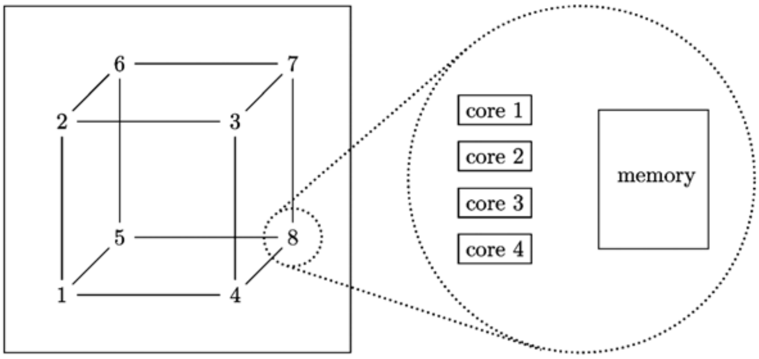


Fig. 3. An example of a NUMA node structure.

As a test problem, we have computed the mixed volume of the system known as the “Sonic8” (28 equations in 28 variables with total degree approximately 5.2×10^{10}) [13] on a NUMA system with 8 identical nodes configured as shown in Figure 3, in which each node has 4 independent processor cores and certain amount of local memory. Logically, the local memory of all nodes are accessible by all the nodes and hence form a global memory in which we can store the task pool and all other data. However from the point of view of efficiency, while processor cores in one node can access the memory on another node, the access time is roughly proportional to the length of the path between the two nodes in Figure 3. As a result, the best speedup ratio is achieved when one uses nodes that are closer together in Figure 3. With the best node placement, we were able to achieve close to n-fold linear speedup up to 16 processors. Beyond 16 processors, the speedup ratio drops slightly but maintains a relatively high efficiency. The speedup ratio as a function of the number of processors used is shown in Figure 4.

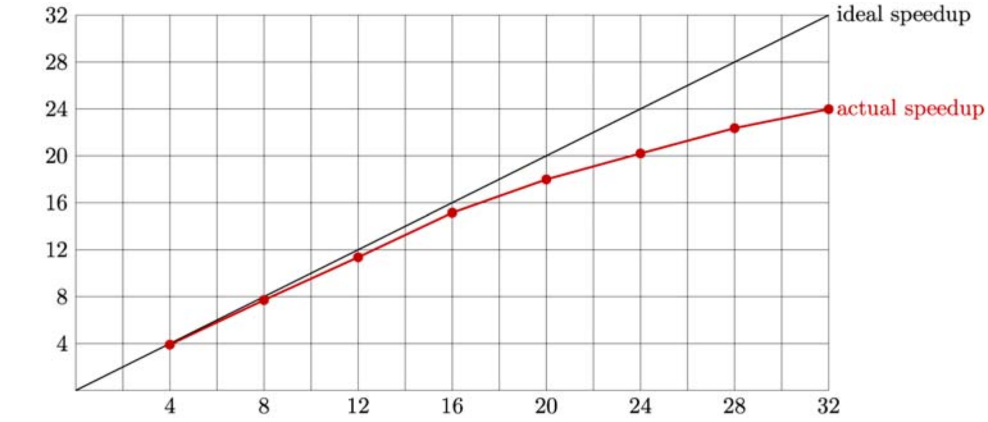
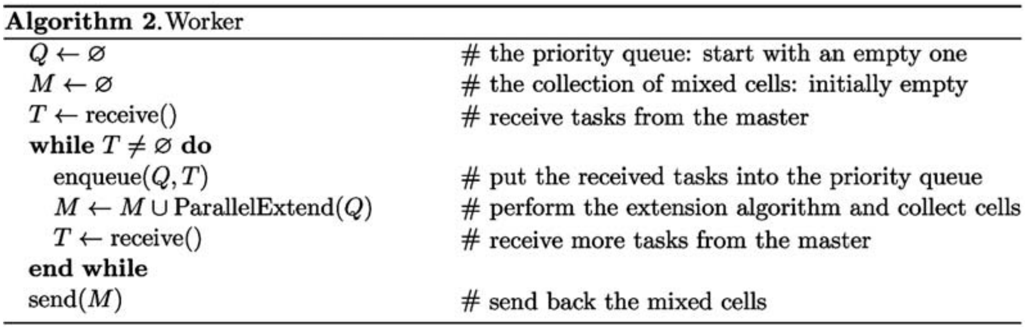


Fig. 4. Average parallel speedup on a NUMA architecture computer for the system "sonic-8".

4.3. Distributed-Memory single systems

In distributed-memory single systems, a master-worker model is used for our algorithm. In such models, the master populates the initial task pool, and then sends them to the workers to be further explored. Each worker, equipped with its own task pool for the newly discovered tasks by itself, continuously executes Algorithm 1 and requests more tasks from the master whenever it exhausted its own task pool. The overall algorithm for each worker is given by Algorithm 2 in which "send" and "receive" refer to sending to and receiving from the master.



Algorithm 2. Worker

```
$Q \leftarrow \varnothing$                              # the priority queue: start with an empty one
$M \leftarrow \varnothing$                              # the collection of mixed cells: initially empty
$T \leftarrow$ receive()                     # receive tasks from the master
while $T \neq \varnothing$ do
    enqueue($Q, T$)                     # put the received tasks into the priority queue
    $M \leftarrow M \cup$ ParallelExtend($Q$)     # perform the extension algorithm and collect cells
    $T \leftarrow$ receive()                 # receive more tasks from the master
end while
send($M$)                                # send back the mixed cells
```

The Message Passing Interface, or MPI, is an API (application programming interface) specification that allows computers to communicate with one another [21]. Even though it is not sanctioned by any major standards body, MPI has became a *de facto* standard for communication among processes of a parallel program running on a distributed memory system such as a computer cluster [25]. Currently, Algorithm 2 is implemented using MPI. The speedup effect using multiple nodes in a cluster on system "cyclic-15" [3] is shown in Figure 5. It is expected that the speedup ratio cannot get close to those achieved on a multi-core system, since MPI (at least MPI-1) only uses distributed memory model [22], and our mixed cell computation problem, not a pleasantly parallel problem, induces significant communication overhead between nodes. However, using MPI is possible to scale to more processors than using multi-core architecture. In particular, up to 96 processor cores can be used to compute mixed volume of the “cyclic15” [3] system with very good speedup ratio. With all 96 processors, the problem that will take a serial program almost 9 hours to compute can be finished within 10 minutes! It is expected that with larger and more advanced computer clusters, we can scale it to much larger number of processors.

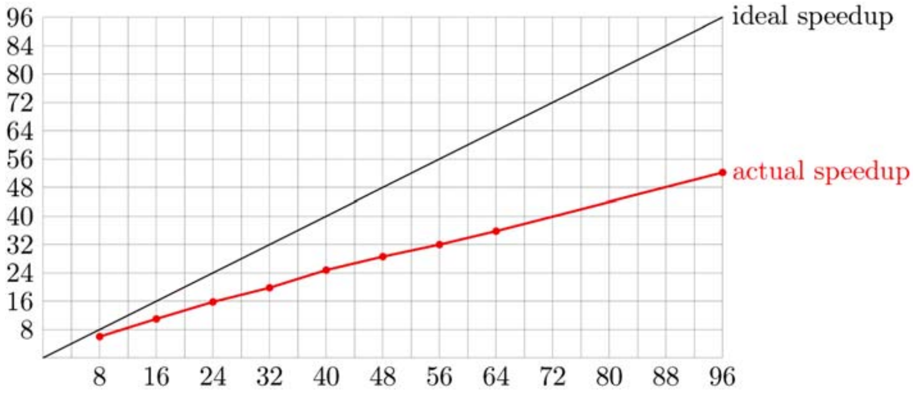
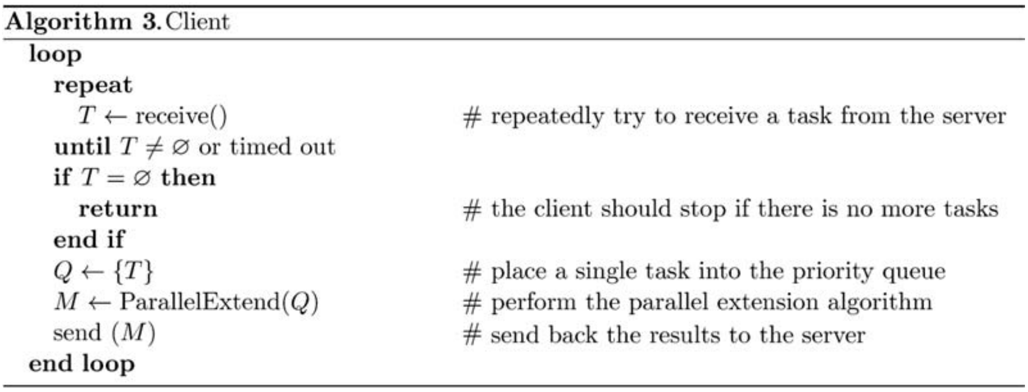


Fig. 5. Parallel speedup on clusters using MPI for the system “cyclic-15”.

4.4. Distributed environment

As one tries to solve larger and more challenging systems of polynomial equations, the computational power needed may very well exceed that of a single computer or cluster. Distributed computing technology has allowed a very loosely coupled set of computers, which may spread around the world, connected to each other via slower and less reliable networks, such as the Internet, to collaborate with each other and act as a single supercomputer.

To this end, an experimental version of our mixed volume computation algorithm was implemented using a client/server model based on TCP/IP protocol [26]. This implementation consists of a client side program and a server side program. One server is running at any time, and it is responsible for computing the relation table and dividing the problem into groups of tasks using the algorithm described above. To ensure the problem is being divided into enough number of groups of tasks to exploit the large scale parallelism allowed by the distributed environment, the server will perform the extension algorithm similar to the serial version until the task pool reaches a prescribed size min_Q, and then the server will wait for client requests and sends the groups to clients via the network. Multiple client programs can run simultaneously. A client can run on any computer that is connected to the network. Each client will continuously request tasks from the server via the network until the server can no longer reply with new tasks. For each task received, the client performs the extension algorithm described above and then sends the result back to the server. The client side algorithm is listed in Algorithm 3.



Algorithm 3. Client

```
loop
  repeat
    $T \leftarrow$ receive()            # repeatedly try to receive a task from the server
  until $T \neq \varnothing$ or timed out
  if $T = \varnothing$ then
    return                              # the client should stop if there is no more tasks
  end if
  $Q \leftarrow \{T\}$                  # place a single task into the priority queue
  $M \leftarrow$ ParallelExtend($Q$)    # perform the parallel extension algorithm
  send ($M$)                            # send back the results to the server
end loop
```

Comparing to the version for distributed-memory single systems, the main challenge here is the connection between a client and the server being generally slow and unreliable. So different from their counter parts in the distributed-memory single systems, the client must take into consideration the possibility of failing to connect to the server, while the server must take into consideration the possibility that the clients may stop computation before returning the results. Moreover, both sides need to consider the possibility of messages being corrupted. In particular, the server has to verify the correctness upon receiving the computation results from a client program: The server first performs standard check-sum procedures [5] on the results it received, then the results are checked using the systems of inequalities (3) and (4) which define subfaces. If the result fails either one of the two tests, the corresponding task will be sent out for recomputation. The same avenue applies if the server fails to hear back from the client at all. To implement this, the server is separated into two parts, the main part is responsible for performing the first stage of extension algorithm to generate the initial pool of tasks as well as sending them to the clients for further exploration. The other part, the receiving part, is responsible for listening for computation results from the clients. To make sure every task is explored, the server keeps a *time-stamp table* T as well as a waiting list W to keep track of the waiting time of each task that has been sent out for computation. In particular, upon being sent to a client, a task w is moved to the waiting list W, and the time-stamp is stored in the corresponding slot $T[w]$ of the table T, and upon receiving its result from the client, it is then removed from both W and T. The main part of the server continuously scans the time-stamp table T for tasks that has been on the waiting list for too long, and such tasks are moved back to the task pool Q to be recomputated. The two parts of the server-side algorithm is listed in Algorithms 4 and 5.

Algorithm 4. The receiving part of the server algorithm

```
repeat
  $(M', w) \leftarrow$ receive()          # receive the mixed cells from the client
  for all $m \in M'$ do
    check $m$                             # check the mixed cell for consistency
  end for
  $M = M \cup M'$                         # merge the lists of mixed cells
  $W = W \backslash \{w\}$                # remove $w$ from the waiting list
until $W = \varnothing$ or timed out
```

Algorithm 5. The main server algorithm

```
Input: Supports $S = (S_1, \ldots, S_n)$
Output: A set of mixed cells of $S = (S_1, \ldots, S_n)$
  Compute the relation table
  $M \leftarrow \varnothing$                                    # the set M will contain the mixed cells
  while $|Q| < min_Q$ do
    $\hat{E}_k = (e_{i_1}, \ldots, e_{i_k}) \leftarrow$ dequeue($Q$)   # where $\hat{e}_{i_j} \subset \hat{S}_{i_j}$
    $R_k \leftarrow \{1, \ldots, n\} \backslash \{i_1, \ldots, i_k\}$  # indices of the remaining supports
    for all $l \in R_k$ do
      $L_l \leftarrow$ eliminate($L(\hat{\mathcal{S}}_l)$)        # let $L_l$ be the remaining edges
      $r_l \leftarrow |L_l|$
    end for
    $m_{k+1} \leftarrow \min\{r_l\}_{l \in R_k}$                   # the minimum number of remaining edges
    if $m_{k+1} > 0$ then
      $i_{k+1} \leftarrow \min\{l \mid r_l = m_{k+1}\}$            # the index of the next support
      for all $\hat{e}_{i_{k+1}} \in L_{i_{k+1}}$ do
        if $I(\hat{E}_k, \hat{e}_{i_{k+1}})$ is positive then
          if $k + 1 = n$ then
            $M \leftarrow M \cup \{(\hat{E}_k, \hat{e}_{i_{k+1}})\}$   # obtained a mixed cell
          else
            enqueue($Q, (\hat{E}_k, \hat{e}_{i_{k+1}})$)           # obtained a new task to be explored
          end if
        end if
      end for
    end if
  end while
  $W \leftarrow \varnothing$                                    # tasks waiting for results
  repeat
    while $Q \neq \varnothing$ do
      $C \leftarrow$ wait-for-request()                          # wait for a request from a client
      $w =$ dequeue($Q$)                                         # get a task from the task pool
      $T[w] =$ now()                                             # mark the task sending time
      send($C, w$)                                               # send the requesting client a task
    end while
    for all $w \in W$ do
      if now() $- T[w] > T_{limit}$ then
        enqueue($Q, w$)                                          # put the task back to the task pool
        $W = W \backslash \{w\}$                                 # remove $w$ from the waiting list
      end if
    end for
  until $W = \varnothing$
  return $M$                                                     # return the mixed cells
```

With this implementation, the mixed volume of a very large system VortexAC6 (shown in Equation (7)) with 30 equations of 30 variables (total degree: 2^{30}) can be computed by using 145 individual processors from multiple clusters and workstations.

$$(7) \quad \begin{cases} \sum\limits_{k=1}^{6} y_{ik}\,(x_{jk} - x_{ik} - x_{ij}) + y_{jk}\,(x_{ik} - x_{jk} - x_{ij}) = 0 \\ 1 - x_{ij} - y_{ij} \cdot x_{ij} = 0 \end{cases} \qquad \text{for } 1 \leq i < j \leq 6$$

The total CPU hours involved exceeds eight months, but with the distributed model, we marvelously complete the computation within 2 days. It is worth noting that the number of processors used in this case is actually only limited to the hardware we have at our disposal. It appears that the same implementation can solve the same problem using 5,000 to 10,000 processors with similar efficiency. We must, however, emphasize that this ad hoc solution is only meant to be a proof-of-concept. To build a software package that is useful for general use, a more robust solution is necessary. The field of distributed computing is well-studied with active and exciting developments. Many mature toolkits are already available for helping programmers to build distributed computing software. We shall investigate the possibility of building robust distributed mixed cell computation package using toolkits such as Condor [27], Condor-G [8], and BOINC [2].

References

1. A. Albouy and A. Chenciner, Le probléme des n corps et les distances mutuelles, *Inv. Math.*, **131** (1998), 151-184.
2. D. P. Anderson, *BOINC: A System for Public-Resource Computing and Storage*, Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing, 2004, pp. 4-10.
3. G. Björk and R. Fröberg, A faster way to count the solutions of inhomogeneous systems of algebraic equations, *J. Symbolic Comput.*, **12(3)** (1991), 329-336.
4. W. Boege, R. Gebauer and H. Kredel, Some examples for solving systems of algebraic equations by calculating Groebner bases, *J. Symbolic Comput.*, **2** (1986), 83-98.
5. W. Cary Huffman and Vera Pless, *Fundamentals of Error Correcting Codes*, Cambridge University Press, New York, NY, 2003.
6. T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to Algorithms*, The MIT Press, Cambridge, MA, 2001.
7. I. Z. Emiris and J. F. Canny, Efficient incremental algorithms for the sparse resultant and the mixed volume, *J. Symbolic Comput.*, **20** (1995), 117-149.
8. J. Frey, T. Tannenbaum, M. Livny, I. Foster and S. Tuecke, Condor-G: A Computation Management Agent for Multi-Institutional Grids, *Cluster Computing*, **5(3)** (2002), 237-246.

9. T. Gao and T. Y. Li, Mixed volume computation for semi-mixed systems, *Discrete Comput. Geom.*, **29(2)** (2003), 257-277.

10. T. Gao, T. Y. Li and M. Wu, MixedVol: a software package for mixed volume computation, *ACM Trans. on Math. Soft.*, **31(4)** (2005), 555-560.

11. M. Hampton and R. Moeckel, Finiteness of stationary configurations of the four-vortex problem, *Trans. of Amer. Math. Soc.*, **361(3)** (2008), 1317-1332.

12. B. Huber and B. Sturmfels, A polyhedral method for solving sparse polynomial systems, *Math. Comp.*, **64** (1995), 1541-1555.

13. T. L. Lee and T. Y. Li, Mixed volume computation in solving polynomial systems, *Contemp. Math.*, **556** (2011), 97-112.

14. T. Y. Li, Numerical solution of multivariate polynomial systems by homotopy continuation methods, *ACTA Numerica,* (1997), 399-436.

15. T. Y. Li, Solving polynomial systems by polyhedral homotopies, *Taiwanese J. Math.*, **3** (1999), 251-279.

16. T. Y. Li, *Solving Polynomial Systems by the Homotopy Continuation Method*, Handbook of numerical analysis, Vol. XI, Edited by P. G. Ciarlet, North-Holland, Amsterdam, 2003.

17. T. Y. Li and X. Li, Finding mixed cells in the mixed volume computation, *Found. Comput. Math.*, **1** (2001), 161-181.

18. T. Mizutani, A. Takeda and M. Kojima, Dynamic enumeration of all mixed cells, *Discrete Comput. Geom.*, **37** (2007), 351-367.

19. A. Morgan, *Solving Polynomial Systems Using Continuation for Engineering and Scientific Problems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

20. V. W. Noonburg, A neural network modeled by an adaptive Lotka-Volterra system, *SIAM J. Appl. Math.*, **49** (1989), 1779-1792.

21. P. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann, Burlington, MA, 1996.

22. M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*, McGraw Hill, New York, NY, 2003.

23. J. Reinders, *Intel Threading Building Blocks: Outfitting C++ For Multi-core Processor Parallelism*, O'Reilly Media, Sebastopol, CA, 2007.

24. S. Skiena, *The Algorithm Design Manual*, Springer-Verlag, New York, 1998.

25. M. Snir, S. Otto, S. Hass-Lederman, D. Walker and J. Dongara, *M*PI: The Complete Reference, The MIT Press, Cambridge, MA, 1998.

26. W. R. Stevens, *T*CP/IP Illustrated, Vol. 1, The Protocols, Addison-Wesley, Boston, MA, 1994.

27. D. Thain, T. Tannenbaum and M. Livny, Distributed computing in practice: the condor experience, *Concurrency and Computation: Practice and Experience*, **17(2-4)** (2005), 323-356.

Tianran Chen
Department of Mathematics
Michigan State University
East Lansing, Michigan 48824
U.S.A.
E-mail: chentia1@msu.edu

Tsung-Lin Lee
Department of Applied Mathematics
National Sun Yat-sen University
Kaohsiung 80424, Taiwan
E-mail: leetsung@math.nsysu.edu.tw

Tien-Yien Li
Department of Mathematics
Michigan State University
East Lansing, MI 48824
U.S.A.
E-mail: li@math.msu.edu