On Using Volume Computation to Estimate the Work Distribution for Parallel Programs

Thomas Fahringer

Institute for Software Technology and Parallel Systems
University of Vienna
Bruenner Str. 72, A-1210 Vienna, Austria
tf@par.univie.ac.at

Hoon Hong

Research Institute for Symbolic Computation
Johannes Kepler University
University of Linz
A-4040 Linz, Austria
hhong@risc.uni-linz.ac.at

Abstract

In this paper we describe a performance parameter which models the work contained in a parallel program and the corresponding work distribution. The work distribution is modeled at the program level which carefully distinguishes between useful and redundant work. We achieve high accuracy due to aggressive exploitation of compiler knowledge such as loop iteration spaces, array access patterns and data distributions. The underlying algorithm is based on the intersection and volume computation of n-dimensional linear and convex polytopes.

The performance parameter has been implemented under the P^3T, which is a static parameter based performance prediction tool under the *Vienna Fortran Compilation System (VFCS)*.

1 Introduction

In order to parallelize scientific applications for distributed memory systems such as the iPSC/860 hypercube, Meiko CS-2, Intel Paragon, CM-5 and the Delta Touchstone the programmer commonly decomposes the physical domain of the application – represented by a set of large arrays – into a set of sub-domains. Each of these sub-domains is then mapped to one or several processors of a parallel machine. The sub-domain is local to the *owning* processors and remote to all others. This is known as *domain decomposition* [13]. The processors of a parallel system execute only those statement instantiations for which they own the corresponding sub-domain. This naturally specifies the amount of work to be done by each processor and consequently the overall work distribution of a parallel program. Therefore domain decomposition inherently implies a work distribution.

It is well known ([8, 5, 3, 28, 26, 17, 21, 6, 25, 18, 12]) that the work distribution has a strong influence on the cost/performance ratio of a parallel system. An uneven work distribution may lead to a significant reduction in a program's performance. Therefore providing both programmer and parallelizing compiler with a work distribution parameter for parallel programs is vital to steer the selection of a good domain decomposition.

The following code shows the Livermore Fortran Kernel-6 ([19]), which illustrates a general linear recurrence equation.

Example 1.1 *Livermore Fortran Kernel 6 (LFK-6)*

```
        PARAMETER (N=64)
        DOUBLE PRECISION W(1001), B(64,64)
        ....
        DO I=2,N
           DO J=1,I-1
S:            W(I) = W(I) + B(I,J) * W(I-J)
           ENDDO
        ENDDO
```

The LFK-6 traverses only part of a two dimensional array B as the loop variable J of the innermost loop depends on loop variable I of the outermost loop. The optimized target program – extended by appropriate Vienna Fortran processor and distribution statements (cf. Section 2) – as created by the *VFCS* is shown as follows:

Example 1.2 *Parallel LFK-6*

```
        PARAMETER (N=64)
        PROCESSORS P(8)
        DOUBLE PRECISION W(1001) DIST(BLOCK)
        DOUBLE PRECISION B(64,64) DIST(BLOCK,:)
        ...
        EXSR B(:,:) {[0/56,0/0]}
        DO I=2,N
           EXSR W(:) {[</>]}
           DO J=1,I-1
S:            OWNED(W(I))->W(I) = W(I) + B(I,J) * W(I-J)
           ENDDO
        ENDDO
```

Array W and B are respectively distributed block-wise and row-wise to 8 processors. The EXSR statements specify the communication implied by the chosen data distribution. The *VFCS* assigns 125 elements of W to $P(1), \ldots, P(4)$ and $P(6), \ldots, P(8)$ and 126 elements to processor $P(5)$. The first 125 elements are owned by $P(1)$. As the outermost loop of the above kernel iterates from 2 to 64, only $P(1)$ is actually executing the assignment statement S.

S is executed 2016 times for a single instantiation of the outermost loop. This is also the overall amount of work in this loop nest. The underlying data distribution strategy, however, assigns all 2016 iterations to the first processor. Therefore the work distribution of this example is far from being perfect. Optimal work distribution is attained if each processor would assign a new value to $W(I)$ for 252 (= 2016 / 8) distinct instantiations of S.

Much of previous research ([3, 21, 6, 25, 18]) concentrates on monitoring or estimating work distribution and close derivatives such as processor utilization, execution profile and average parallelism at the machine level. Traditional work distribution analysis distorts results in the following cases:

- Array replication ([32]) is a popular technique to optimize parallel programs. For instance, array replication can be used to decrease communication overhead. This technique commonly results in full processor utilization – traditional analysis reports perfect work distribution – although redundant (replicated) work is done. A processor performs redundant work, if some other processor is doing the same work. This is the case if program arrays or portions of it are assigned – replicated – to several processors. In all other cases the set of parallel processors perform useful work.

- A significant instrumentation overhead – deteriorating the measurement accuracy – is frequently induced, if software monitors (cf. [2]) are utilized to measure a parallel program's work distribution.

The method described in this paper presents a novel approach to estimate the *useful work distribution* of distributed memory parallel programs at the program level. A single profile run ([8]) of the original sequential code on a von Neumann architecture derives numerical values for program unknowns such as loop iteration and statement execution counts. All array assignment statements inside loops are examined for how many times they are executed by which processor. Replicated arrays are carefully distinguished from non-replicated arrays. The array subscript expressions are mapped into the corresponding loop iteration space based on the sub-domain owned by each processor. The loop iteration space correlates with the amount of work contained in a loop. The corresponding intersection of the array subscript functions with the loop iteration space defines a polytope, the volume of which serves as an approximate amount of work to be done by a specific processor. An analytical model is incorporated to describe the work distribution of the entire loop nest. This modeling technique is extended to procedures and the overall program.

Clearly, improving other performance parameters such as communication or memory accesses can be as important as a good work distribution. However, this is strongly architecture and application dependent. The described work distribution parameter is implemented as part of the P^3T ([11]) which is a static parameter based performance prediction tool. The P^3T automatically computes at compile time a set of parallel program parameters which report on the estimated communication overhead, data locality and the work distribution of a parallel program. This enables the programmer to evaluate the influence of different performance aspects of a parallel program. Previous work about the P^3T ([8, 11, 7, 9, 10]) describes the parallel program parameters, tradeoffs between these parameters, and demonstrated how to use the P^3T to guide the parallelization and optimization effort under a parallelizing compiler. The principal contribution of this paper is an in-depth evaluation of the *useful work distribution* parameter as an individual performance aspect.

The Lawrence Livermore Loops are used to validate the described approach. Results demonstrate that the estimates for the *useful work distribution* parameter are very accurate and consistently improves for increasing problem sizes.

The rest of this paper is organized as follows: The next section outlines the underlying compilation and programming model. The third section describes how to estimate the amount of work contained in a parallel program and the corresponding *useful work distribution*. The next section presents a detailed description of the algorithms incorporated for computing the work distribution parameter. In the fourth section experiments are shown to validate the accuracy and quality of the described parameter. In the fifth section related work is discussed. Finally some concluding remarks are made.

2 Compilation and Programming Model

The *useful work distribution* is implemented as a performance parameter of the P^3T. This tool is integrated in the *VFCS* ([4]), which is an automatic parallelization system for massively parallel distributed-memory systems (DMS) translating Fortran programs into explicitly parallel message passing programs. The described work distribution parameter is based on the underlying compilation and programming model of the *VFCS*, which is explained as follows:

The parallelization strategy of the *VFCS* is based on domain decomposition in conjunction with the Single-Program-Multiple-Data (SPMD) programming model. This model implies that each processor is executing the same program based on a different data domain. The input to *VFCS* are Vienna Fortran programs ([31]). Vienna Fortran is a machine-independent language extension to Fortran77, which provides annotations for the specification of data distributions. Note that for the single sequential profile run to eliminate unknown values in the program control structures, the Vienna Fortran language extensions are ignored. Thus the profile run is performed based on the original sequential program. The output of *VFCS* is a parallel Fortran program with explicit message passing. The performance prediction methods described in this paper are implemented for data parallel Fortran programs. However, most of the incorporated techniques can be used for other languages such as the C programming language as well.

Let P denote the set of processors available for the execution of the parallel program, and A an array with *index domain* I^A, as determined by the declaration of A. A *data distribution* for A with respect to P is a total function $\delta^A : I^A \to \mathcal{P}(P)$, which associates one or more processors with every element of A. If $p \in \delta^A(i)$ for some $i \in I^A$, then a processor p *owns* $A(i)$, or $A(i)$ is local to p. In this paper, only arbitrary block distributions, and total replication of arrays are considered.

The *work distribution* of a parallel program is determined – based on the data distribution – according to the *owner computes rule*. There are no other work distribution concepts considered such as those associated with FORALL loops in Vienna Fortran. The *VFCS* uses masked assignment statements to implement this rule. Let $S : v = \ldots$ denote an assignment statement. v is a subscripted variable of the form $A(exp_1, \ldots exp_m)$, and A is a properly distributed array, which is meant to be a non-replicated array. Then the associated *masked assignment statement* is written as $OWNED(v) \to S$. The execution of this statement in a process p results in the execution of S iff p owns v. $OWNED(v)$ is called the *mask* of S. For all other statements, the mask is constant TRUE.

Processors can only access local data. Non-local data referenced by a processor are buffered in so-called *overlap areas* that extend the memory allocated for the local segment. The updating of the overlap areas is automatically organized by the system via explicit message passing. This is the result of a series of optimization steps that determine the communication pattern, extract single communication statements from a loop, fuse different communication statements, and remove redundant communication. These optimizations are applied after the first step in the parallelization of the program has been performed. This first step results in a *defining parallel program* as described below: for each potentially non-local access to a variable, a communication statement *EXSR* (EXchange Send Receive) is automatically inserted. An *EXSR* statement ([15]) is syntactically described as $EXSR\ A(I_1, ..., I_n)\ [l_1/u_1, ..., l_n/u_n]$, where $v = A(I_1, ..., I_n)$ is the array element inducing communication and $ovp = [l_1/u_1, ..., l_n/u_n]$ is the overlap description for array A. For each i, l_i and u_i respectively specify the left and right extension of dimension i in the local segment. The dynamic behavior of an *EXSR* statement can be described as follows:

```
IF (executing processor p owns v) THEN
        send v to all processors p', such that
        (1) p' reads v, and
        (2) p' does not own v, and
        (3) v is in the overlap area of p'
ELSE IF (v is in the overlap area of p) THEN
         receive v
     ENDIF
ENDIF
```

In Vienna Fortran ([31]) the user may declare processor arrays by means of the *PROCESSORS* statement. For instance, *PROCESSORS P(4,8)* declares a two-dimensional processor array with 4x8 processors. Distribution annotations may be appended to array declarations to specify distributions of arrays to processors. For instance, *REAL A(N,N) DIST(BLOCK,BLOCK) TO P* distributes array A two dimensional block-wise to the processor array P.

3 Useful Work Distribution of a Parallel Program

In this section we develop a model to estimate the work contained in a parallel program and the corresponding *useful work distribution*.

Consider the following n-dimensional loop nest L with a masked statement S referencing a m-dimensional array A.

DO $I_1 = B_1, E_1$
 DO $I_2 = b_2(I_1), e_2(I_1)$
  DO $I_3 = b_3(I_1, I_2), e_3(I_1, I_2)$

```
  ...
  DO I_n = b_n(I_1,...,I_{n-1}), e_n(I_1,...,I_{n-1})
    ...
S:  OWNED(A(f_1(I_1,...,I_n),...,f_m(I_1,...,I_n))->A(f_1(I_1,...,I_n),...,f_m(I_1,...,I_n)) = ...
    ...
  ENDDO
 ENDDO
ENDDO
ENDDO
```

where the loops of L are consecutively numbered from 1 to n. The loop at level 1 is the outermost loop. The loop at level i is denoted by L_i. Loop L_i consists of a loop body and a loop header statement. Loops are normalized, so that the loop increment is equal to 1. L_l^u represents the set of loops between loop nest level l and u. I_i is the loop variable of L_i. I_l^u refers to the set of loop variables of L_l^u. f_i is a linear function of I_{i-1}^1 and $1 \leq i \leq m$. B_1, E_1 are constants, and b_j, e_j $(1 \leq j \leq n)$ are linear functions of I_{j-1}^1. In addition, L defines Υ_L, the *loop iteration space* of L, which can be described by a set of $2 * n$ inequalities:

$$\begin{array}{rcccl} B_1 & \leq & I_1 & \leq & E_1 \\ b_2(I_1) & \leq & I_2 & \leq & e_2(I_1) \\ & & \cdots & & \\ b_n(I_1,...,I_{n-1}) & \leq & I_n & \leq & e_n(I_1,...,I_{n-1}) \end{array}$$

Note that the mask of statement S is executed for every iteration of L for all processors. The assignment statement itself, however, is only executed by a processor p if the mask of S evaluates to TRUE for a specific instantiation of S.

Let S be an assignment statement inside of a nested loop L, A the left hand-side array of S, and P^A the set of all processors to which A is distributed.

Definition 3.1 Amount of work contained in an assignment statement

Let $work(S,p)$ denote a total function, which defines the number of times S is locally executed for a processor $p \in P^A$ during a single instantiation of L.

$work(S,P^A)$ denotes the overall amount of work contained in L with respect to S and is equal to the number of times S is executed in the sequential program version of L.

$work(S,P^A)$ refers to as the *useful work* contained in L with respect to S. This is in contrast to $\sum_{p \in P^A} work(S,p)$ which might contain redundant (replicated) work as demonstrated by the following corollary.

Corollary 3.1

$$\sum_{p \in P^A} work(S,p) \geq work(S,P^A)$$

Proof 3.1

$\sum_{p \in P^A} work(S,p) = work(S,P^A)$ *if A is a distributed array, and*
$\sum_{p \in P^A} work(S,p) > work(S,P^A)$ *if A is a replicated array.*

Υ_L, the loop iteration space of L, describes a n-dimensional convex and linear polytope PO in R^n ([8]). In order to find out how many times a mask evaluates to TRUE for a processor p, which is equal to $work(S,p)$, the *subscript functions*, which are associated with $A(f_1(I_1,\ldots,I_n),\ldots,f_m(I_1,\ldots,I_n))$ and based on the local array segment boundaries of p, are mapped into R^n. This mapping process can be described by a set of $2 * m$ *mapping inequalities*:

$$\begin{array}{ccccc} l_1^p & \leq & f_1(I_1, ..., I_n) & \leq & u_1^p \\ & & \cdots & & \\ l_m^p & \leq & f_m(I_1, ..., I_n) & \leq & u_m^p \end{array} \tag{1}$$

where l_i^p (u_i^p) – assumed to be known – is the lower (upper) segment boundary for the i-th dimension of array A associated with p.

Each of the above inequalities defines two n-dimensional half-spaces which may result in an intersection with PO. This assumes that $f_1, ...f_m$ are non-constant functions. A constant function is defined by a single numerical value not containing any variables. After $2 * m$ intersections – one for each inequality – a n-dimensional polytope PO_p is created, whose volume serves as an approximation for $work(S, p)$. To be exact, every integer-valued vector in PO_p represents a single evaluation to TRUE for the mask of S with respect to p. The overall number of such vectors is the precise amount of work to be done by processor p.

In the actual implementation, n-dimensional polytopes are represented by its set of geometrical objects: vertices, edges, faces, ..., n-1 dimensional facets. For an intersection of the polytope with a half-space, all geometrical objects are checked for being inside, partially inside, or outside of the half-space. Outside objects are deleted. Partially inside objects are cut at the boundary of the half-space (hyperplane) and closed by an appropriate facet. For instance, if a face is cut then it is closed by an edge. An edge is closed by a vertex, etc. Nothing has to be done for inside objects.

The volume of a n-dimensional polytope is computed by a triangulation of the corresponding polytope into a set of disjoint n-dimensional convex simplices. A n-dimensional convex simplex contains exactly $n + 1$ vertices. According to [24] the volume of a n-dimensional convex simplex is given by $\frac{1}{n!} Det(\vec{v}_2 - \vec{v}_1, \ldots, \vec{v}_{n+1} - \vec{v}_1)$, where $\{v_1, \ldots, v_{n+1}\}$ is the set of vertices of the corresponding simplex. The sum of the volumes across all simplices of PO_p defines the volume of PO_p. The intersection and volume algorithms are described and analyzed in detail in section 4. These algorithms are based on n-dimensional geometry, which made the corresponding implementation difficult for debugging and extensions. For this reason we want to investigate alternative algorithms in future work.

If the processor local segment boundaries of the above mapping inequalities are replaced by the array boundaries, then the resulting intersection with the loop iteration space and the array A subscript functions – based on the array boundaries – yields an n-dimensional convex polytope PO^A. The volume of PO^A is used as an approximation for $work(S, P^A)$.

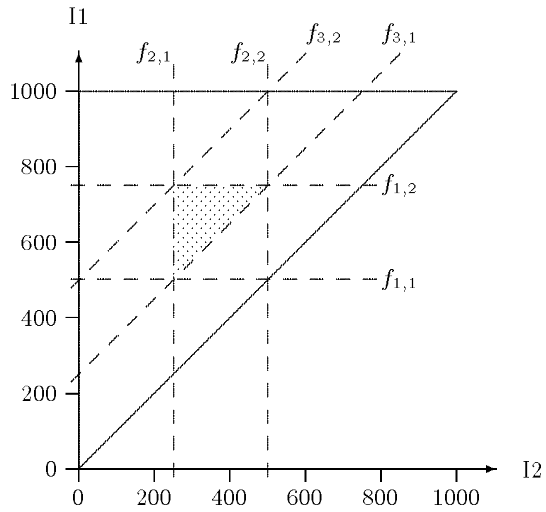
In the following example a 2-dimensional loop with a 3-dimensional array A on the left hand-side of an assignment statement is illustrated. The subscript functions associated with $A(I1, I2, I1 - I2 + 1)$ for processor $P(3, 2, 2)$, which owns $A(501 : 750, 251 : 500, 251 : 500)$, are mapped into R^2. The intersection between these functions and the loop iteration space represents the iteration space to be processed by $P(3, 2, 2)$ (shaded area in Figure 1).

Example 3.1 *3-dimensional array in a 2-dimensional loop*

```
PARAMETER (N=1000)
PROCESSORS P(4,4,4)
INTEGER B(N,N,N) DIST(BLOCK,BLOCK,BLOCK)

EXSR B(:,:,:) {[0/0,0/0,749/750]}
DO I1 = 1,N
   DO I2 = 1,I1
      OWNED(A(I1,I2,I1-I2+1))->A(I1,I2,I1-I2+1) = B(I1,I2,I2+1) + A(I1,I2,I1-I2+1)
```

```
    ENDDO
  ENDDO
```



Half-spaces for $P(3,2,2)$:

$f_{1,1}$: $I1 \geq 501$
$f_{1,2}$: $I1 \leq 750$
$f_{2,1}$: $I2 \geq 251$
$f_{2,2}$: $I2 \leq 500$
$f_{3,1}$: $I1 - I2 + 1 \geq 251$
$f_{3,2}$: $I1 - I2 + 1 \leq 500$

Figure 1: Loop iteration space intersection regarding $P(3,2,2)$

In the following the optimal amount of work to be done by each processor dedicated to an array assignment statement is defined.

Let S be an array assignment statement inside of a nested loop L, where A is the left hand-side array.

Definition 3.2 Optimal amount of work

The arithmetic mean: $owork(S) = work(S, P^A)/|P^A|$ *defines the optimal amount of work to be processed by every single processor in* P^A.

Based on the optimal amount of work a goodness function for the *useful work distribution* of an array assignment statement in a loop L is defined.

Definition 3.3 Useful work distribution goodness for an array assignment statement

The goodness of the useful work distribution with respect to an array assignment statement S *is defined by*

$$wd(S) = \frac{1}{owork(S)} \sqrt[2]{\frac{1}{|P^A|} \sum_{p \in P^A} \left(work(S,p) - owork(S)\right)^2}$$

The above formula is the standard deviation (σ) divided by the arithmetic mean ($owork(S)$), which is known as the variation coefficient in statistics (cf. [1]).

Note that there is an important difference between the *useful work distribution* as compared to traditional *work distribution* analysis. The latter one ignores whether or not redundant work is done by the processors, while the first one carefully models this effect. Traditional work distribution could be easily computed by simply replacing $work(S, P^A)$ in Definition 3.2 by $\sum_{p \in P^A} work(S,p)$. For the remainder of this paper the term work distribution refers to useful work distribution.

Note that a good work distribution does not necessarily imply a good overall performance of a parallel program. Besides the work distribution other performance aspects such as communication overhead ([8, 11]) and data locality ([7, 14, 30]) play an important role to obtain a well optimized program. P^3T fully adheres to this need by computing parallel program parameters for all of these performance aspects. Tradeoffs among parameters and parameter priorities have been discussed at length in [8, 11, 10].

4 Implementation

In this section we describe the most important mathematical methods and algorithms, which are used for the implementation of the parallel program parameters, in a very compact form. This includes two geometric operations. First, we present an algorithm to compute the intersection of a n-dimensional convex polytope with a half-space. Second, we show how to derive the volume of a n-dimensional convex polytope.

4.1 Intersection

In the following we describe the intersection between a n-dimensional polytope and a half-space in R^n, which results in a new n-dimensional polytope in R^n.

Definition 4.1 Hyperplane and Half-space

Let $f : R^n \rightarrow R$ be a linear non-constant function of $I_j \in R$ $(1 \leq j \leq n)$ and $c \in R$, then a hyperplane H in R^n is defined as follows:

$$f(I_1, ..., I_n) = c,$$

which in turn defines two half-spaces of the following form:

$$\begin{array}{c} f(I_1, ..., I_n) \leq c \\ f(I_1, ..., I_n) \geq c \end{array}$$

Definition 4.2 Convex polytope

Let $f_i : R^n \rightarrow R$ be a linear function $(1 \leq i \leq n)$ of $I_k \in R$ $(1 \leq k \leq n)$ and $l_j, u_j \in R$ $(1 \leq j \leq n)$, then a n-dimensional linear and convex polytope PO *in R^n is defined by a set of $2 * n$ inequalities:*

$$\begin{array}{ccccc} l_1 & \leq & f_1(I_1, ..., I_n) & \leq & u_1 \\ & & \cdots & & \\ l_n & \leq & f_n(I_1, ..., I_n) & \leq & u_n \end{array}$$

Note that for each of the above inequalities where $l_i = u_i$ $(1 \leq i \leq n)$ holds, the resulting polytope PO looses one dimension.

Inherently a convex n-dimensional polytope PO consists of a set of vertices, edges, faces, polyhedras, ..., n-1 dimensional polytopes, and the n-dimensional polytope itself. We will denote them the **geometric objects** of PO, where a vertex is a 0-dimensional, an edge a 1-dimensional, a face a 2-dimensional, etc. object of PO. The n-dimensional object of PO is PO itself. Every n-dimensional object B is closed at the boundary by a set of n-1 dimensional objects, which are denoted **facets** of B in the remaining part of this thesis.

In the following we define relationships among geometrical objects.

Definition 4.3 Geometric Relationship and Sub-Objects

Let PO *be a n-dimensional polytope in* R^n*, then the set of i-dimensional* $(0 \leq i \leq n)$ *sub-objects of* PO *is denoted by* OB_i*. The set of* j $(0 \leq j < i)$ *dimensional objects of* $B \in \mathrm{OB}_i$ *is defined by* $ldim(B, j)$*. The set of* k $(0 \leq i < k \leq n)$ *dimensional objects, of which* $B \in \mathrm{OB}_i$ *is part of, is defined by* $hdim(B, k)$*.*

Definition 4.4 Geometric Attribute

Let PO *be a n-dimensional polytope in* R^n*,* OB_0 *the set of vertices of* PO*,* Θ *a set of half-spaces in* R^n*,* $v \in \mathrm{OB}_0$ *a vertex described by a vector* $\vec{v} = (i_1, \ldots, i_n)$ *with* $i_j \in R$ $(1 \leq j \leq n)$*.* $\theta \in \Theta$ *is described by* $f(I_1, \ldots, I_n) \leq c$ $(f(I_1, \ldots, I_n) \geq c)$ *where* f *is a linear function of* $I_j \in R$ $(1 \leq j \leq n)$ *and* $c \in R$*. Then a vertex* $v \in \mathrm{OB}_0$ *is uniquely associated with one of the following geometric attributes:* $attr(v, \theta) = IN$ *for* $f(v) < c$ $(f(v) > c)$*,* $attr(v, \theta) = \mathrm{ON}$ *for* $f(v) = c$*, and* $attr(v, \theta) = \mathrm{OUT}$ *for* $f(v) > c$ $(f(v) < c)$*.*

$B \in \mathrm{OB_i}$ $(1 \leq i \leq n)$ *is associated with the geometric attribute IN/ON/OUT with respect to* $\theta \in \Theta$ *described by* $f(I_1, \ldots, I_n) \leq c$ $(f(I_1, \ldots, I_n) \geq c)$ $\Longleftrightarrow$ *for all vertices* $v \in ldim(B, 0)$*:* $f(v) < c$ / $f(v) = c$ / $f(v) > c$ $(f(v) > c$ / $f(v) = c$ / $f(v) < c)$ *respectively.*

In the remainder of this thesis a vertex v is associated with a unique vector description $\vec{v}$ according to the above definition.

Definition 4.5 Cut Object

Let PO *be a n-dimensional polytope in* R^n*, then* $B \in \mathrm{OB_i}$ $(i > 0)$ *is denoted a* **cut** *i-dimensional object of* PO *with respect to an arbitrary* $\theta \in \Theta$ $\Longleftrightarrow$ *there exist two vertices* $v_1, v_2 \in ldim(B, 0)$*, where* $attr(v_1, \theta) = IN$ *and* $attr(v_2, \theta) = OUT$*.*

If B *is a cut i-dimensional object with respect to* $\theta \in \Theta$*, then all objects in* $hdim(B, j)$ $(i < j \leq n)$ *are cut objects and* PO *is denoted a* **cut polytope***.*

Note that vertices cannot be cut objects.

In the following we describe an intersection algorithm which takes as an input a n-dimensional convex and linear polytope *PO1* $\in R^n$ and a half-space $\theta \in \Theta$. This algorithm results in an intersection polytope *PO2*, which is derived from the intersection between *PO1* and θ.

Algorithm 4.1 (Intersection between a polytope and a half-space in R^n**)**

INPUT:

- convex n-dimensional polytope *PO1* in R^n
- half-space θ in R^n described by $f(I_1, \ldots, I_n) \leq c$, $I_j \in R$ $(1 \leq j \leq n)$, $c \in R$, and f a linear function of $I_1, \ldots, I_n$

OUTPUT:

- *PO2*, the polytope resulting from the intersection between *PO1* and θ; $ldim(PO2, 0) = \phi$ iff there is no intersection between *PO1* and θ.

METHOD:

θ may induce an intersection with *PO1*. In that case the algorithm will cut a subset of the geometric objects in *PO1*. All cut objects are closed by newly created geometrical objects. The resulting *PO2* is a convex n-dimensional polytope in R^n. The algorithm incorporates two procedures: *intersect* and *intersect_dim*. The main procedure *intersect* is originally called by intersect(OB_n, n, θ), which in turn calls another procedure *intersect_dim* bottom up for every dimensions between 1 and n of *PO1*. OB_n is equivalent to *PO1*.

The algorithm is processed bottom up, beginning with edges (1-dimensional objects) and finishing with PO_n (n-dimensional object). It starts with the set of edges of *PO1*. If there is at least one cut edge, then there is an intersection. In that case intersect_dim is called for every dimension between 2 and n of *PO1*. Every time intersect_dim is called for a specific dimension i, all objects in OB_i are analyzed whether they are inside, outside or cut with respect to θ. If it turns out that an object is outside of θ then it is deleted. If it is inside of θ nothing is done, as it is part of the final intersection body *PO2*. If it is cut, then it has to be closed by an i-1 dimensional geometric object.

In the following we describe information associated with each geometric object:

- in(B, θ) is a boolean value indicating whether or not object B is inside of θ.
- on(B, θ) is a boolean value indicating whether or not object B is ON the hyperplane of θ.
- out(B, θ) is a boolean value indicating whether or not object B is outside of θ.
- cut(B, θ) is a boolean value indicating whether or not object B is a cut object.
- vertexl(E) and vertexr(E) denote the two vertices of an edge E in arbitrary order;

In the following we describe variables to be used by the described algorithm:

- object_on is a boolean variable indicating whether or not a specific object is on the hyperplane of a half-space θ currently being processed.
- object_in or object_out is a boolean variable indicating whether or not a specific object is respectively inside or outside of a half-space θ currently being processed.
- poly_intersect denotes whether or not a specific $\theta \in \Theta$, currently being processed by the described algorithm, actually cuts *PO1*.

In the following we describe auxiliary functions:

- inside(B, θ) and outside(B, θ) are functions which respectively evaluate whether or not object B is inside or outside a half-space θ.
- on_boundary(B, θ) is a function, which evaluates whether or not object B is ON the hyperplane of a half-space θ.
- $mark(B, IN/OUT/ON/CUT, \theta)$ is a function which marks an object B as IN/OUT/ON/CUT with respect to a half-space $\theta \in \Theta$ and its hyperplane;
- $mark\_higher\_dim(B, CUT, \theta)$ is a function which marks object B and all higher dimensional objects of which B is part of, as CUT objects with respect to $\theta \in \Theta$.
- create_new_vertex(E, θ) is a function which creates a new vertex v for a cut edge E based on an intersection with a half-space θ, such that edge E is inside of θ and v is ON the hyperplane of θ.
- create_new_object(i-1) is a function which creates a new i-1 dimensional object B and returns a reference to B.
- facet_of$(B_1$,$B_2)$ is a function which marks object B_2 as a facet of object B_1.

```
procedure intersect( OB_j, j, θ)
    OB_j : geometric object  PO1
    j      : dimensionality of  OB_j
    θ      : halfspace
begin
   /* initialization of variables and geometric objects */
   poly_intersect = FALSE;
   /* check whether or not there is an intersection between  OB_j and
      θ based on edges of  OB_j */
   intersect_dim(ldim( OB_j, 1), 1, θ,poly_intersect);
   if poly_intersect
      /* if there is an actual intersection then build  PO2
         starting with faces and finishing with  OB_j */
      for i=2 to n do
            intersect_dim(ldim( OB_j, i), i, θ,poly_intersect);
      endfor
   endif
   /*  PO2 is defined by  OB_j, which is transformed from  PO1 to  PO2
      by the described algorithm */
end intersect
procedure intersect_dim( OB_i,i,θ,poly_intersect)
   OB_i              : set of i-dimensional objects
   i                 : dimensionality of objects in  OB_i
   θ                 : halfspace
   poly_intersect : boolean
begin
   poly_intersect = FALSE;
   if i == 1   /* treat edges */
      for every edge E ∈  OB_1 do
         if outside(vertexl(E),θ)
               mark(vertexl(E),OUT,θ);
         elseif inside(vertexl(E),θ)
                  mark(vertexl(E),IN,θ);
         elseif on_boundary(vertexl(E),θ)
                  mark(vertexl(E),ON,θ);
         endif
         if outside(vertexr(E),θ)
               mark(vertexr(E),OUT,θ);
         elseif inside(vertexr(E),θ)
                  mark(vertexr(E),IN,θ);
         elseif on_boundary(vertexr(E),θ)
                  mark(vertexr(E),ON,θ);
         endif
         if (out(vertexr(E),θ) and in(vertexl(E),θ)) or
          (in(vertexr(E),θ) and out(vertexl(E),θ))           /* edge E is a cut edge */
            poly_intersect = TRUE;
            new_vertex = create_new_vertex(E,θ);
            mark(new_vertex,ON | CUT,θ);
            mark_higher_dim(E,CUT,θ);
         elseif out(vertexr(E),θ) && out(vertexl(E),θ)      /* edge is outside of θ */
                  mark(E,OUT,θ);
                   OB_1  =  OB_1 − {E};                      /* delete OUT edge */
         elseif in(vertexr(E),θ) and in(vertexl(E),θ)
                  mark(E,IN,θ);
         elseif on(vertexr(E),θ) and on(vertexl(E),θ)
                  mark(E,ON,θ);
         endif
     end for
   else          /* treat objects with dimension 2 or higher */
      for every object B_1 ∈  OB_i do
```

```
        object_out = TRUE;
        for every vertex v ∈ ldim(B1,0) do
            if out(v,θ) == FALSE
                object_out = FALSE;
                break;
            endif
        end for
        if object_out                          /* object B1 is outside of θ; delete B1 */
            OBi = OBi − {B1};
        elseif cut(B1,θ)                       /* object B1 is a CUT object */
                B2 = create_new_object(i-1);   /* create closing facet for B1 */
                facet_of(B1,B2);               /* B2 is a new object of B1 */
                mark(B2,ON,θ);                 /* B2 is on the hyperplane of θ */
                for every object B3 ∈ ldim(B1, i − 2) do
                    if cut(B3,θ) or on(B3,θ)
                        facet_of(B2,B3);
                    endif
                end for
        else        /* check whether or not object B1 is ON or inside of θ */
            object_on = TRUE; object_in = TRUE;
            for every vertex v ∈ ldim(B1,0) do
                if on(v,θ) == FALSE
                  object_on = FALSE;
                endif
                if in(v,θ) == FALSE
                  object_in = FALSE;
                endif
            end for
            if object_on
                mark(B1,ON,θ);
            elseif (object_in) mark(B1,IN,θ);
            endif
        endif
      end for
    endif
end intersect_dim
```

Figure 2 illustrates algorithm 4.1 by intersecting a triangular polytope *PO1* with a half-space θ. Figure 2.a shows the hyperplane (dotted line plane) of the corresponding half-space. The space below the hyperplane is inside of the half-space. *PO1* consists of $ldim(PO1,0) = \{v1, \ldots, v6\}$, $ldim(PO1,1) = \{e1, \ldots, e9\}$, and $ldim(PO1,2) = \{f1, \ldots, f5\}$. In the following we describe two phases of the algorithm:

- after procedure intersect_dim is processed for $i = 1$ (intersection regarding edges):
  - $ldim(PO1,1) = \{e1, \ldots, e8\}$
  - cut edges: $\{e5, \ldots, e8\}$
  - edges inside of θ: $\{e1, \ldots, e4\}$
  - edges outside of θ: $\{e9\}$
  - new vertices to close cut edges: $\{v7, \ldots, v10\}$
  - vertices outside of θ: $\{v5, v6\}$

- after procedure intersect_dim is processed for $i = 2$ (intersection regarding faces):
  - cut faces: $\{f1, \ldots, f4\}$
  - faces inside of θ: $\{f5\}$
  - new edges: $\{e10, \ldots, e13\}$
  - new (closing) faces: $\{f6\}$

The resulting polytope *PO2* of the intersection is shown Figure 2.b. *PO2* is visualized once for vertices, edges and faces.

The computational complexity of *procedure intersect* for $n = 2$ is $O(|OB_1|)$, otherwise it is $O(MAX_{1 \leq i \leq n-1}(|OB_i|) * |OB_0|)$. A special case of a n-dimensional polytope as defined by Def. 4.2 is a rectilinear polytope. The cardinality of OB_k $(0 \leq k \leq n)$ of a n-dimensional rectilinear polytope is $\binom{n}{k} 2^{n-k}$. For this case and for $n > 2$ the complexity of the above algorithm is exponential with respect to n. Considering that real-world programs containing arrays with more than 4 $(= n)$ dimensions are found rarely, this computational complexity seems very reasonably. A non-rectilinear loop iteration space Υ_L may induce polytopes with less sub-objects, for example, in case of triangular loop iteration spaces. Furthermore, successive intersections between *PO1* and several half-spaces may significantly decrease the number of remaining sub-objects. As a consequence the exponential complexity as mentioned above is a worst case scenario.

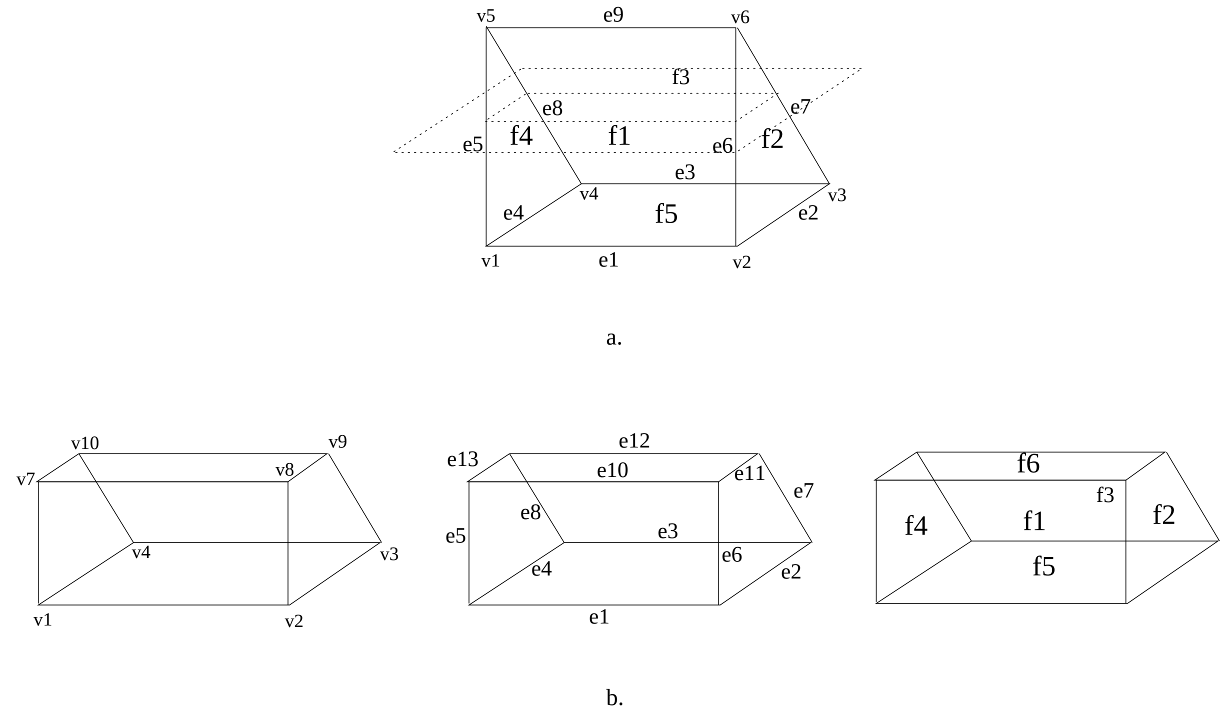


Figure 2: Intersection of a polytope with a hyperplane

4.2 Volume

In the following we describe how to derive the volume of a n-dimensional polytope in R^n.

Definition 4.6 Simplex

> *Let* OB_0 *be the set of vertices in a n-dimensional linear and convex polytope* PO *in* R^n*, then* PO *is a n-dimensional linear and convex simplex, iff* $|\mathrm{OB}_0| = n + 1$.

Theorem 4.1 (Volume of a n-dimensional convex and linear simplex)

Let PO *be a n-dimensional convex and linear simplex in* R^n *and* $\mathrm{OB}_0 = \{v_1, \ldots, v_{n+1}\}$ *the set of vertices of* PO*, then according to [24] the volume of* PO *is defined by a function* **volume** : $PO \to R_0^+$ *with*

$$volume(PO) = \frac{1}{n!} Det(\vec{v}_2 - \vec{v}_1, \ldots, \vec{v}_{n+1} - \vec{v}_1)$$

For the computation of the determinant of a matrix we use an algorithm with $O(n^3)$ complexity. This algorithm incorporates LU decomposition and is described in [22].

Definition 4.7 Center of Mass Vertex

Let OB_0 *be the set of vertices of a n-dimensional polytope* PO *in* R^n*, then*

$$\vec{v}_c := \frac{1}{|\mathrm{OB}_0|} \sum_{\vec{v} \in \mathrm{OB}_0} \vec{v}$$

is the center of mass vertex of PO.

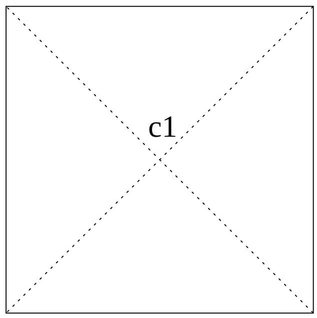


Figure 3: Triangularization of a 2-dimensional object

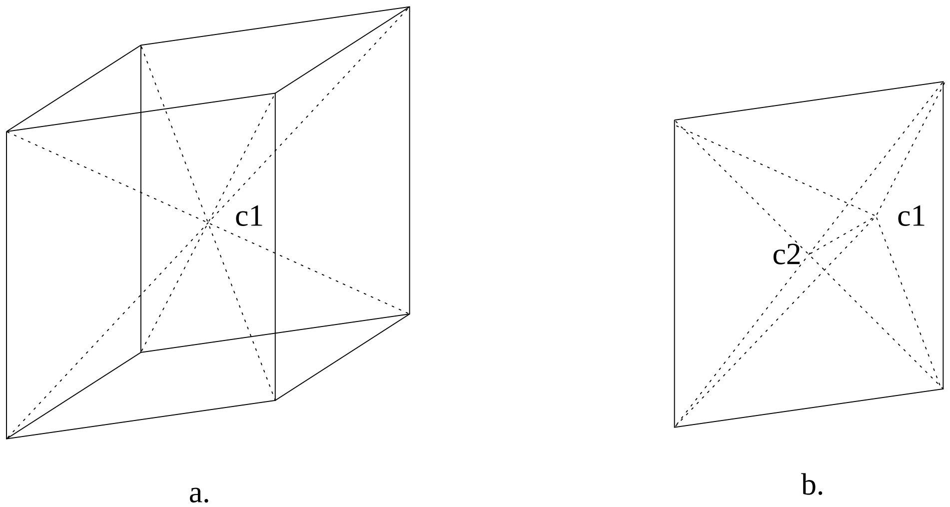


Figure 4: Triangularization of a 3-dimensional object

Algorithm 4.2 (Volume of a n-dimensional convex and linear polytope in R^n)

INPUT:

- n-dimensional convex and linear polytope PO in R^n described by OB_n

OUTPUT:

- volume of PO

METHOD:

The algorithm is based on triangularization which is implemented by a function volume_poly with two parameters: PO the geometric object for which the volume has to be computed and the dimensionality of PO. volume_poly is originally called by volume_poly(OB_n,n). If OB_n is an edge, then the volume is defined by the length of it. Otherwise a recursive algorithm *volume_poly_higher* splits OB_n into a set of n-dimensional simplices.

We start with OB_n. The center of mass vertex of OB_n is computed. For every facet F of OB_n we recursively call *volume_poly_lower*, which again computes the center of mass vertex of F and recursively calls *volume_poly_lower* for every facet of F. We proceed until we reach 1-dimensional facets (edges). At this point we have a set of n+1 vertices defining a specific simplex for which we know the volume based on Theorem 4.1. The sum of the volumes of all such simplices is the volume of OB_n.

The following variables are used by the described algorithm:

- M is a set of vectors defining a simplex where M_i denotes the i-th vector in M

In the following we describe auxiliary functions:

- $center\_of\_mass(PO, k)$ computes the center of mass vertex of a k-dimensional object PO based on Def. 4.7.

In the following we describe information associated with each edge E of a geometric object:

- vertexl(E) and vertexr(E) denote the two vertices of an edge E in arbitrary order.

```
function real volume_poly(PO,k)
 PO  : geometric object;
 k : dimensionality of PO;
begin
  if k == 1                  /* PO is an edge with 2 vertices v1 and v2 */
    v1 = vertexl(PO);
    v2 = vertexr(PO);
    return (|v1 - v2|);      /* The volume of an edge is the length of it */
  else                       /* object is of 2 or higher dimensionality */
    M = ∅;
    return (volume_poly_lower(PO,k,M));
  endif
end volume_poly
```

```
function real volume_poly_lower(PO,k,M)
  PO : geometric object;
  k : dimensionality of PO;
  M : Set of vectors;
begin
   float sum = 0.0;
   if k == 1   /* PO is an edge with vertices v1 and v2 */
     v1 = vertexl(PO);
     v2 = vertexr(PO);
     M = M + v⃗2;
     return (Det(M1 − v⃗1, ..., Mn − v⃗1)/n!);
   else
     v⃗c = center_of_mass(PO,k);
     M = M + v⃗c;
     for all facets F ∈ ldim(PO,k − 1) do
       sum = sum + volume_poly_lower(F,k-1,M);
     end for
     M = M − v⃗c;
     return (sum);
  endif
end volume_poly_lower
```

Function *volume_poly_lower* recursively calls itself for each facet of a given k-dimensional object. This is done for $1 \leq k \leq n$. If $k = 1$ then the determinant of a n x n Matrix M is computed. Therefore *Det(M)* is computed x $(=\sum_{o_1 \in \mathrm{OB}_{n-1}} \sum_{o_2 \in ldim(o_1, n-2)} \cdots \sum_{o_{n-1} \in ldim(o_{n-2}, 1)} 1)$ times. As a consequence the computational complexity of this procedure is given by $O(x * n^3)$, where n^3 is the complexity of the incorporated determinant algorithm (see above). If PO is a rectilinear polytope, which has $2*n$ sub-objects of dimension $n-1$, we can conclude the following: $|OB_{n-1}| = 2 * n$, $|ldim(o_1, n-2)| = 2 * (n-1)$,..., $|ldim(o_{n-2}, 1)| = 2*2 \Longrightarrow x = 2*n*2*(n-1)*2*(n-2)*\ldots*2*2 = 2^{n-2}*(n^2+n-2)$. For this case the computational complexity of the above algorithm is exponential with respect to n. We again assume that the described complexity is a worst case figure due to the same reason as indicated for Alg.4.1 on page 13.

Figure 3 shows a two dimensional geometric object comprised of 4 simplices which are separated by dotted lines; *c1* is the center of mass vertex. Figure 4.a displays a 3-dimensional object *PO* with the center of mass vertex *c1*. Figure 4.b illustrates the triangularization of a sub-object of PO with *c2* the center of mass vertex. *c2* corresponds to a distinct facet of *PO*. The sub-object contains four 3-dimensional simplices.

5 Experiments

In this section we present an experiment – using the Livermore Fortran Kernels 6 ([19]) – to demonstrate the estimation accuracy of the work distribution parameter.

The *VFCS* was used to create sequential and parallel program versions and the P^3T to automatically estimate the work distribution and work parameters for every specific parallel program. The parallel program versions were executed and measured on a iPSC/860 hypercube with 16 processors. Counter based profiling was used to obtain the exact work distribution figures for a comparison against the measured P^3T parameters.

We created four parallel program versions of the LFK-6 (cf. Example 1.1) using the *VFCS*. Program v_2, v_4, v_8 and v_{16} were derived by one dimensional block-wise distributing array W and row-wise distributing array B to 2, 4, 8 and 16 processors, respectively.

Table 1: Estimated versus measured work for v_2

	N=64			N=65			N=512			N=1024		
p	$work$	$\overline{work}$	diff	$work$	$\overline{work}$	diff	$work$	$\overline{work}$	diff	$work$	$\overline{work}$	diff
1	496	475	0.04	496	475	0.04	32640	32475	0.005	130816	130486	0.002
2	1520	1512	0.005	1584	1576	0.005	98176	98117	0.006	392960	392841	0.003

The relative difference *diff* between wd and $\overline{wd}$ is defined by:

$$diff = \begin{cases} \left|\frac{wd-\overline{wd}}{wd}\right| & : \text{ if } wd > 0 \\ |wd - \overline{wd}| & : \text{ if } wd = 0 \end{cases}$$

The exact values for the work distribution function wd with respect to the entire LFK-6 kernel are measured for all program versions for varying problem sizes $64 \leq N \leq 1024$. The P^3T was used to statically estimate wd which is indicated by $\overline{wd}$. In this experiment all figures for wd and $\overline{wd}$ are normalized by division of $|P^A - 1|$, which is the upper bound of $wd(S)$. A proof for this upper bound is given in [8]. Normalizing the work distribution figures allows us to compare different parallel program versions with respect to their work distribution goodness.

Figure 5 displays the measured (solid lines, v_2, v_4, v_8, v_{16}) and the estimated (dashed lines v_2', v_4', v_8', $v_{16'}$) work distribution functions.

It can be observed that the estimation accuracy for v_4 is almost 100 %. $\overline{wd}(v_2)$ deviates from $wd(v_2)$ in the worst case by about 3 %. For larger problem sizes ($N > 128$) the degree of accuracy is better than 99 %. The predictions for both v_8 and v_{16} are consistently lower than the measured results. Note that for small problem sizes the deviation is much larger than for higher problem sizes. This is because of the improving accuracy of the volume approximation for larger problem sizes. It can be clearly seen that the more processors are dedicated to a specific data distribution the better the associated work distribution.

Table 2: Work distribution accuracy

	v_2			v_4		
N	wd	$\overline{wd}$	*diff*	wd	$\overline{wd}$	*diff*
64	0.5079365	0.5249	0.032	0.18929	0.1894	0.0005
65	0.5230769	0.5401	0.032	0.190642	0.19063	0.0005
128	0.503937	0.5123	0.017	0.187803	0.1879	0.0005
256	0.50196	0.5061	0.008	0.18706	0.18713	0.0003
384	0.5013054	0.5041	0.005	0.18682	0.18686	0.0002
512	0.500978	0.50306	0.004	0.186703	0.18673	0.00014
768	0.500651	0.502	0.002	0.18658	0.186531	0.00009
1024	0.50048	0.50153	0.002	0.18652	0.18653	0.00006

In addition, Figure 5 reveals an interesting effect of the data distribution strategy of the underlying compilation system. While for problem sizes $N = \{64, 128, 256, 384, 512, 768, 1024\}$ the number of processors incorporated for every different program version evenly divides the array sizes. This is different for $N = 65$, where the *VFCS* assigns to a specific processor one data element more than to all other processors for every program version. Despite the small difference in assigned work, this has a clear impact on this experiment by suddenly deteriorating the work distribution of v_2 for $N = 65$. This effect diminishes with increasing number of processors incorporated, because $|P^A| - 1$ processors with the same local array segment size have a higher influence on the work distribution than a single processor with a slightly different array segment size. The described work distribution parameter accurately models this behavior.

Figure 5 displays the work distribution curves very coarse grain due to the strongly enlarged dimensionality of the wd-axis. Tables 2 and 3 show the exact difference between predicted and measured values in the *diff* columns. The worst case deviation is 11 %. For even small problem sizes, for instance, $N = 256$, *diff* is improving to a value of 3 %. For a problem size of $N = 1024$, *diff* is already less than 0.2 %.

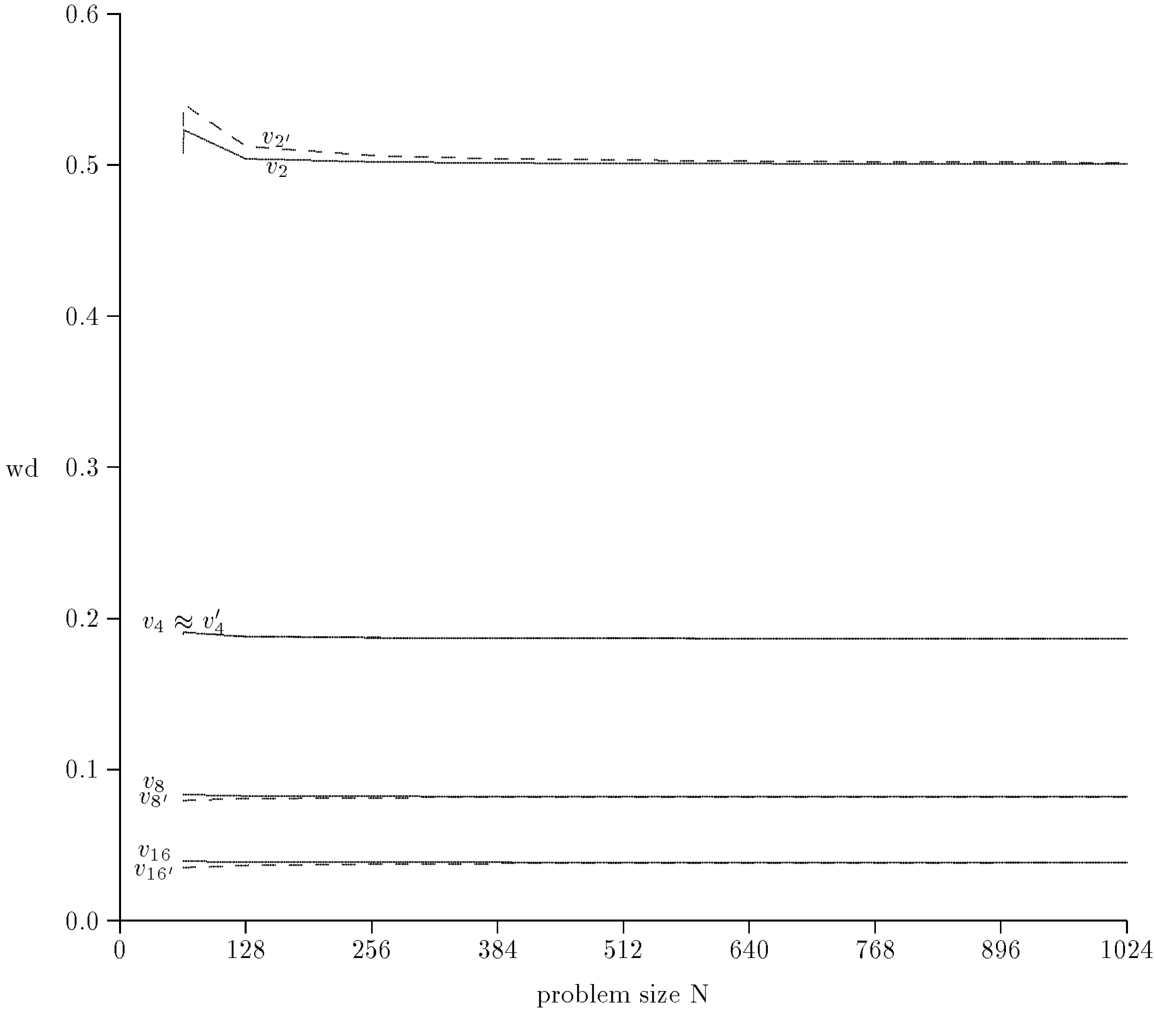


Figure 5: Estimated versus measured work distribution

Both experiments demonstrate that the described estimation for the overall amount of work contained in a program and the associated work distribution is very accurate for even small problem sizes and consistently improves for larger ones.

6 Related Work

B. Carlson et al. ([3]) analyze the execution profile of parallel programs, which reflects the number of busy processors as a function of time. This profile is measured at the machine level and does not consider whether useful or replicated work is done. They present a method to detect phases of homogeneous processor utilization. These phases can be correlated with the algorithms implemented in the underlying parallel program.

D. Eager, J. Zahorjan, and E. Lazowska ([6]) investigate the trade-off between speedup and efficiency, and how efficiently additional processors may be used. They obtain lower and upper bounds of these metrics and emphasize the average parallelism – the average number of busy processors through the life

Table 3: Work Distribution accuracy

	v_8			v_{16}		
N	wd	$\overline{wd}$	*diff*	wd	$\overline{wd}$	*diff*
64	0.08313	0.07941	0.045	0.03902	0.03488	0.11
65	0.083226	0.07951	0.044	0.039055	0.034952	0.10
128	0.082476	0.080585	0.023	0.03871	0.03638	0.06
256	0.082152	0.0812	0.012	0.03856	0.03734	0.032
384	0.08204	0.0814	0.008	0.038515	0.03768	0.022
512	0.081938	0.081511	0.005	0.038489	0.037866	0.016
768	0.081938	0.081614	0.004	0.03846	0.03804	0.011
1024	0.081911	0.081667	0.003	0.038452	0.038133	0.008

of an application.

M. Heath and J. Etheridge ([16]) describe a tool for visualizing performance of parallel programs. Among others they are concerned with the display of processor utilization, which shows the total number of processors in each of three states – busy, overhead, and idle – as a function of time. A processor is categorized as idle if it has suspended execution awaiting a message that has not yet arrived, overhead if it is communicating, and busy if it is executing some program portion other than communication. Several visualization choices are offered. However, none of them reflects redundant work due to array replication.

H. Mierendorf and R. Schwarzwald ([20]) predict the workload of a parallel system as a function of computational and transport work. They model both application program and architecture using a modeling language. This approach assumes the absence of dependencies among concurrently running processes.

K. Wang ([29]) relates to the work distribution of a parallel program by estimating processor busy and idle times. Two classes of processor idle times, namely data and control synchronization times are obtained. In the first case, a processor is idle while waiting for data to arrive. In the second case, two or more processors are waiting to be synchronized. This approach handles shared and distributed memory parallel programs.

N. Tawbi ([27]) estimates the execution time of loops for shared memory multiprocessors incorporating a loop iteration count prediction for all processors employed. For rational loop bounds she computes symbolic answers when feasible and evaluates average values otherwise. Her method does not describe how to deal with procedure calls inside of loop nests.

W. Pugh ([23]) developed a method for counting the number of integer solutions to selected free variables of a Presburger formula. The answer is given symbolically for free variables in the Presburger formula. This approach can be used to count the number of loop iterations. Pugh's method is more general at the cost of a rather high computational complexity to verify a Presburger formula. His algorithm to count loop iterations has not been implemented at the time the final version of this paper has been prepared. This made it difficult to compare it against the implementation and experiments described in this paper.

None of the above related work refers to useful versus redundant work as described in this paper. Most related approaches do not model procedure calls.

7 Conclusion

One of the most difficult and crucial tasks to obtain highly optimized parallel programs for distributed memory computers is to find performance efficient data distributions. The data distribution strongly correlates with the work distribution of a parallel program. Hence, in order to provide the programmer with performance information on a parallel program, a work distribution goodness function is inevitable.

This paper describes the *useful work distribution* performance parameter. In contrast to traditional work distribution analysis, the *useful work distribution* carefully distinguishes between useful and redundant work to be performed by the processors dedicated to a computation. Replicating arrays – which is a popular method to optimize parallel programs – implies redundant work. Traditional work distribution analysis commonly implies optimal work distribution for such cases.

The *useful work distribution* is statically estimated by the P^3T at the program level and is primarily used to guide the search for an efficient data distribution strategy of parallel programs under the Vienna Fortran Compilation System. It is based on mapping the subscript expression functions of arrays on the left hand-side of assignment statements into the loop iteration space of an enclosing loop nest. Subscript expression functions and loop bounds are linear functions of all loop variables of enclosing loops. The intersection between loop iteration space and subscript expression functions of an array based on the local array segment boundaries of a processor p results in a polytope, whose volume serves as an approximation for the work to be processed by p. Based on this parameter the *useful work distribution* is estimated.

For future work we will investigate alternative solutions to the volume and intersection algorithms incorporated for the described work distribution parameter. Furthermore, we plan to extend this parameter by symbolic analysis. So far the work distribution figures are numerical values. Providing parameterized performance functions which are dependent on the number of processors and certain problem sizes should alleviate the interpretation of performance effects significantly.

References

[1] J. Bleymüller, G. Gehlert, and H. Gülicher. *Statistik für Wirtschaftswissenschaftler.* Verlag Vahlen, München, 1985. WiSt Studienkurs.

[2] H. Burkhart and R. Millen. Performance-Measurement Tools in a Multiprocessor Environment. *IEEE Transactions on Computers*, 38(5):725 – 737, May 1989.

[3] B. Carlson, T. Wagner, L. Dowdy, and P. Worley. Speedup properties of phases in the execution profile of distributed parallel programs. In *Computer Performance Evaluation '92: Modeling Techniques and Tools*, pages 83–95, 1992. (Ed.) R. Pooley and J. Hillston.

[4] B. Chapman, S. Benkner, R. Blasko, P. Brezany, M. Egg, T. Fahringer, H.M. Gerndt, J. Hulman, B. Knaus, P. Kutschera, H. Moritsch, A. Schwald, V. Sipkova, and H.P. Zima. *VIENNA FORTRAN Compilation System - Version 1.0 - User's Guide*, January 1993.

[5] B. Chapman, T. Fahringer, and H. Zima. *Automatic Support for Data Distribution*, pages 184–199. Springer Verlag, Portland, Oregon, Aug. 1993.

[6] D.L. Eager, J. Zahorjan, and E.D. Lazowska. Speedup versus Efficiency in Parallel Systems. *IEEE Transactions on Computers*, 38(3):408 – 423, March 1989.

[7] T. Fahringer. Automatic Cache Performance Prediction in a Parallelizing Compiler. In *Proc. of the AICA'93 - International Section*, Lecce, Italy, September 1993.

[8] T. Fahringer. *Automatic Performance Prediction for Parallel Programs on Massively Parallel Computers.* PhD thesis, University of Vienna, Department of Software Technology and Parallel Systems, October 1993.

[9] T. Fahringer. Automatically Estimating Network Contention of Parallel Programs. In *Proc. of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, Vienna, Austria*, May 1994.

[10] T. Fahringer. Using the P^3T to Guide the Parallelization and Optimization Effort under the Vienna Fortran Compilation System. In *IEEE Proc. of the Scalable High Performance Computing Conference 1994, Knoxville, TN*, May 1994.

[11] T. Fahringer and H. Zima. A Static Parameter based Performance Prediction Tool for Parallel Programs. In *Invited Paper, In Proc. of the 7th ACM International Conference on Supercomputing 1993*, Tokyo, Japan, July 1993.

[12] D. Ferrari. *Computer Systems Performance Evaluation.* Prentice Hall, 1978.

[13] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Processors*, volume 1. and 2. Prentice Hall, Englewood Cliffs, NY, 1988.

[14] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. Journal of Parallel and Distributed Computing, October 1988.

[15] H.M. Gerndt. *Parallelization for Distributed-Memory Multiprocessing Systems.* PhD thesis, University of Bonn, December 1989.

[16] M.T. Heath and J. A. Etheridge. Visualizing performance of parallel programs . Technical Report ORNL/TM-11813, Oak Ridge National Laboratory, Oak Ridge, TN, May 1991.

[17] R. Jain. *The Art of Computer Systems Performance Analysis.* Wiley Professional Computing, 1991.

[18] M. Kumar. Measuring parallelism in computation intensive scientific/engineering applications . *IEEE Transactions on Computers*, 37(9):1088–1098, 1988.

[19] F.H. McMahon. *The Livermore Fortran Kernels Test of the Numerical Performance Range*, pages 143–186. Elsevier Science B.V.,North Holland, Amsterdam, 1988.

[20] H. Mierendorff and R. Schwarzwald. LAPAS: A Performance Evaluation Tool for Large Parallel Systems. In *Tagungsband*, München, März 1990. 11. ITG/GI Fachtagung.

[21] K.-H. Park. *Dynamic Processor Partitioning for Multiprogrammed Multiprocessor Systems.* PhD thesis, Vanderbilt University, Nashville, TN, Aug 1990.

[22] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in C; The Art of Scientific Computing.* Cambridge University Press, 1988.

[23] William Pugh. Counting Solutions to Presburger Formulas: How and Why. UMIACS-TR-94-27 and CS-TR-3234, Univ. of Maryland, Dept. of Computer Science, 1994.

[24] Boris Abramovic Rosenfeld. *Mutli-dimensional Spaces (in russian)*, chapter 5, pages 169–173. Nauka, Moscow, 1966.

[25] K.C. Sevcik. Characterization of parallelism in applications and their use in scheduling . *Performance Evaluation Review*, 17(1):171–180, 1989.

[26] Chatterjee Siddhartha. *Compiling data-parallel programs for efficient execution on shared-memory multiprocessors.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, October 1991.

[27] N. Tawbi and P. Feautrier. Processor Allocation and Loop Scheduling on Multiprocessor Computers. In *ACM International Conference on Supercomputing 1992*, pages 63–72, Washington D.C., July 1992.

[28] Sesh Venugopal and Vijay K. Naik. SHAPE: a parallelization tool for sparse matrix computations. Research report rc 17899, IBM Research Division, T.J. Watson Research Center, Yorktwon Heights, NY 10598, July 1992.

[29] K.Y. Wang. A Performance Prediction Model for Parallel Compilers . Technical report, Computer Science Dept., Purdue University, November 1990. Technical Report CSD-TR-1041, CAPO Report CER-90-43.

[30] M.E. Wolfe and M. Lam. A data locality optimizing algorithm. In *In Proceedings of the SIGPLAN 91 Conference on Program Language Design and Implementation*, Toronto, Canada, June 1991.

[31] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran - a language specification. Technical report, ICASE, Hampton,VA, 1992. ICASE Internal Report 21.

[32] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press Frontier Series. Addison-Wesley, 1990.