

# Tournament Computations in GLU

R. Jagannathan and A.A. Faustini\*  
Computer Science Laboratory  
SRI International  
Menlo Park, California 94025,  
U.S.A.

## Abstract

A tournament computation is an approach in intensional programming for expressing data parallelism. We illustrate how this approach can be effectively utilized in GLU (an intensional language for large-grain parallel processing) for expressing massive parallelism in diverse applications. Additionally, we contrast the tournament computation approach with its equivalent in extensional programming, namely, the divide-and-conquer approach.

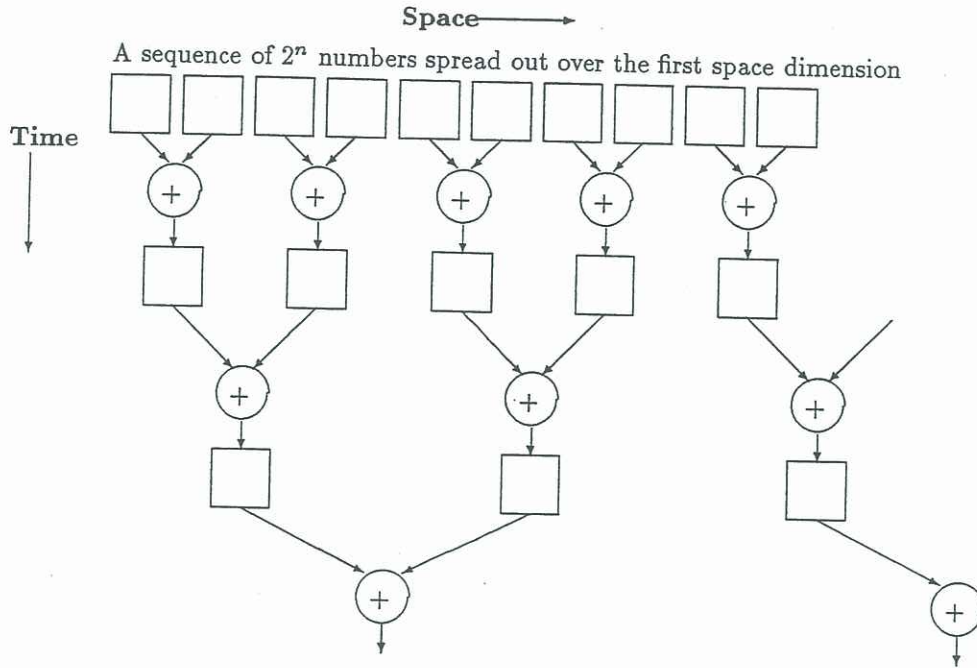
## 1 Introduction

The standard approach for expressing data (or SIMD) parallelism in functional (extensional) languages is by using the “divide-and-conquer” approach. In this approach, the solution to a problem is recursively expressed as a composition of solutions to subproblems. The approach has three stages: In the first stage, data (which defines an instance of the problem) is recursively divided; in the second stage, a function is applied on the kernel data; and in the third stage, the result of a problem is obtained by combining results obtained by solving subproblems.

Divide-and-conquer is the de facto approach for expressing data parallelism in one-dimensional Lucid. In this sense, one-dimensional Lucid, despite being an intensional language, relies upon certain “extensional” features such as recursive functions to express data parallelism. It was not until the evolution of multi-dimensional Lucid (in languages such as Field Lucid [3, 1], Indexical Lucid [5], and mLucid [4]) that the ability to intensionally express data parallelism was realized. Ashcroft (in 1987) proposed the “tournament computation” approach to express data parallelism by writing a simple program (in Field Lucid)

---

\*On sabbatical leave from Arizona State University



### Program

```
tsum( h, n ) = tree @t READY
  where
    h is current h;
    READY is current n;
    tree = h fby ( tree + succ tree ) @s (2*here);
  end
```

Figure 1: Tournament Summation in Field Lucid

to compute the sum of a sequence of numbers [2]. The program is shown in Figure 1.

The function `tsum` expresses an intensional “binary tree” (or tournament) summation without using recursion. It accepts a stream of vectors to be summed and a stream of their log sizes and produces a stream of the vector sums. The tournament computation associated with this function has implicit data parallelism in that pairs of adjacent numbers can be summed simultaneously. The crucial aspect of the intensional solution is that data is not treated as a monolithic structure; rather, it is available as a distributed collection of its constituent values. The structure of data is given by the context or tag associated with each constituent value.

In this paper, we are interested in showing the effectiveness of the tournament computation approach in expressing scientific, symbolic, and interactive applications in the intensional language GLU. In doing so, we contrast the ap-

proach with the divide-and-conquer approach that is espoused in extensional programming languages for expressing data parallelism.

## 2 Overview of GLU

GLU is a hybrid of the multidimensional dataflow programming model and the procedural programming model. The multidimensional dataflow programming model is based on the language Indexical Lucid [5] which, in turn, is an extension of Wadge and Ashcroft's Lucid [6]. The programming model is a natural extension of dataflow programming model as suggested by Lucid. The essential difference is that each edge denotes a multi-dimensional stream of data values rather than just a one-dimensional temporal stream<sup>1</sup>. The logical position of each data value of a multidimensional stream is an  $n$ -tuple where  $n$  is the number of dimensions and each component of the tuple identifies the logical position of a data value in a particular dimension. Multidimensional dataflow operations can operate on the multidimensional streams in any of the different dimensions. The programming model not only allows for implicit expression of function-level (or MIMD) parallelism but also data-level (or SIMD) parallelism.

The procedural paradigm allows for user-defined procedural functions, which enable programs to have "effects" such as I/O. It also allows for implementation of the language on a diverse set of currently-available "conventional" parallel systems. The use of procedural functions results in the granularity of parallelism to be large.

A GLU program is an expression. An expression can be as simple as a constant or a variable, or it can be a term (which is an operation applied to one or more expressions), or it can be an application of a user-defined function, or it can be the invocation of a subcomputation, or it can be an application of a procedural function. The expression may have an associated set of equations that define the variables of the expression. Each equation defines a variable (on its left-hand-side) in terms of an expression (on its right-hand-side.) The user-defined function is defined as an equation with the function symbol and the names of its parameters on the left-hand-side of the equation and an expression that may refer to the parameter variables on the right-hand-side. A subcomputation is defined using an indexed where-clause where the head of the where-clause is the output of the subcomputation and the body of the where-clause defines this output equationally.

In equational terms, a program is a mapping from a set of input streams of daton to an output stream. Furthermore, a term, a function invocation, a subcomputation invocation, or an expression are all mappings from their input streams of datons to an output stream of datons. A constant is a stream of

---

<sup>1</sup>A temporal stream refers to stream in which each data value's logical position corresponds to logical time.



a constant daton and each variable occurrence is the stream of datons that corresponds to that of its defining expression.

An implicit dimension called the **time-dimension** encloses every GLU program. In other words, each daton associated with an expression has implicit with it a **time-context**. The notion of time is a logical one and is not necessarily related to real time.

### 3 Example 1: Merge Sort

A classical application of the divide-and-conquer approach in extensional programming is the mergesort algorithm. In this algorithm, an unsorted list is divided into two sublists, the sublists are recursively sorted (using the same algorithm) and the sorted sublists are merged to produce the sorted list. This algorithm when realized in GLU is shown below.

The user-defined function **ms** implements the divide and conquer scheme by dividing the argument list if it is large enough and merging the sorts of the divided lists or simply invoking **sort** if the argument list is small enough. Function **in** is a procedural function which returns a list of the appropriate size.

```
display( ms( sequence ) )
where
  sequence = in( CHUNK_SIZE*NUM_CHUNKS );
  ms(seq) = if seq_size <= CHUNK_SIZE
            then sort( seq )
            else merge( ms( subseq( seq, 0, halfsize ) ),
                        ms( subseq( seq, 1, halfsize ) )
                      )
            fi
  where
    seq_size = seqsize(seq);
    halfsize = seq_size div 2;
  end;
end
```

The tournament computation approach views mergesort as follows: start off with a vector of  $n$  sublists at time 0 each of which are sorted independently and simultaneously (resulting in data parallelism); at each successive time, merge spatially adjacent sorted sublists; and at time  $\log n$ , the only remaining sublist will be the sorted list.

The GLU program that embodies the tournament computation approach is given below. The user-defined function **ms** merges the sorts of sublists of size **CHUNK\_SIZE** using **mergesort**. The sublists are obtained by applying function **subseq** on the unsorted list.

```

display( ms( sequence ) )
where
  sequence = in( CHUNK_SIZE*NUM_CHUNKS );
  ms(seq) = mergesort( sort( a ), NUM_CHUNKS )
  where
    index s;
    a = subseq( seq, s, CHUNK_SIZE );
    mergesort( sortsubseq, n ) = sorttree when.t ( size >= n )
    where
      index t;
      sorttree = sortsubseq fby.t (merge( lchild sorttree, rchild sorttree));
      size = 1 fby.t size+size;
    end;
  end;
end

```

The data parallelism that can be exploited is shown in Figure 2. The main difference between the divide-and-conquer approach and the tournament computation approach is the following. The divide-and-conquer approach views a data aggregate as a monolithic object which it divides into its constituent elements using recursion. The tournament computation approach views a data aggregate as a distributed collection of its elements and thus only needs iteration to combine these elements into the desired result.

From a parallel implementation standpoint, the tournament computation approach is superior than the divide-and-conquer approach for the reasons given below. First, realization of recursion requires centralized state information (such as display) which can severely limit the effectiveness of the implementation. Second, the divide-and-conquer approach possibly requires copying of data at each level of recursion both in the division and combining phases whereas the tournament computation approach only may require copying of data in the combine phase.

## 4 Example 2: Fractal Processing

This example illustrates how the tournament computation approach can be used in a graphics application that is rich in data parallelism. We consider a particular kind of fractal processing, namely, Mandelbrot set generation.

The problem can be stated as follows: For a constant  $C$ , the Mandelbrot set is the set of points  $(x, y)$  in a unit square such that

$$\lim_{i \rightarrow \infty} m(z_i) \leq 4$$

such that

$$z_0 = x + iy, \forall i > 0, z_i = z_{i-1}^2 + C, m(z_i) = a^2 + b^2, z_i = a + ib.$$

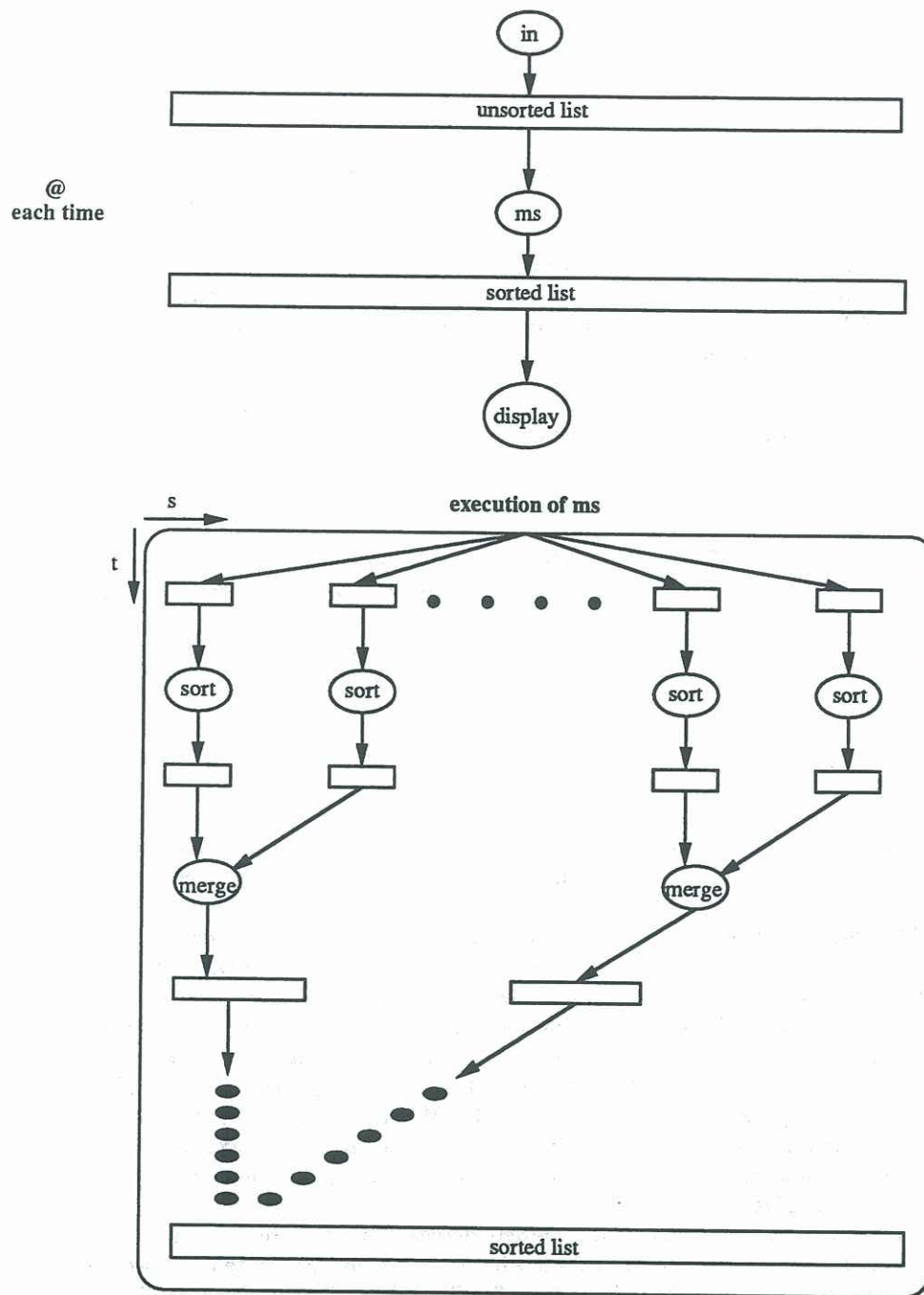


Figure 2: Data Parallelism in Mergesort



Associated with each point  $(x, y)$  is an integer  $c$  which is 0 if  $(x, y)$  is in the set and is the smallest  $i$  such that  $m(z_i) > 4$ . The integer associated with each point denotes a color with zero denoting black and positive integers denoting other colors. The computation of the color of each point can proceed independently of other points and typically requires several floating point operations.

The following GLU program uses the tournament computation approach to generating and displaying the Mandelbrot set.

```
q where
  q = pic asa.i size >= PICTURE_SIZE
    where
      index x,y,i;

      size = KERNEL_SIZE fby.i size + size;
      pic = display(x, y, size, generate(x, y, size))
        fby.i
        ( p + next.x p + next.y p + next.x next.y p
          where p = lchild.x lchild.y pic; end;
        );

    end;
end
```

The program computes  $q$  at time context 0 and all other contexts 0. This value itself is not important; what is important is the generation of subimages of the Mandelbrot set to be displayed.

The variable `size` is initially the `KERNEL_SIZE` and is doubled at each successive  $i$ -context. The value of term `pic asa.i size >= PICTURE_SIZE` at all contexts 0 is the value of `pic` at  $i$ -context where the value of `size` exceeds constant `PICTURE_SIZE` with all other contexts being 0. The definition of `pic` embodies the tournament computation approach. The variable `pic` at context  $(i, x, y)$  is defined as the sum of the variable at contexts  $(i-1, 2x, 2y)$ ,  $(i-1, 2x+1, 2y)$ ,  $(i-1, 2x+1, 2y+1)$ , and  $(i-1, 2x+1, 2y+1)$ . When  $i = 0$ , the variable `pic` is defined in terms of the procedural functions `display` and `generate` that actually generate and display a portion of the image (of size given by `size` starting at coordinates given by  $x, y$ .)

Essentially, variable `pic` when viewed across context  $i$  represents a two-dimensional tournament computation with image being generated in parallel and displayed at the the base level ( $i = 0$ ). The size of each subimage that is generated independently is the constant `KERNEL_SIZE`. The number of subimages and the depth of the tournament computation tree are inversely proportional to the size of this constant.

Figure 3 illustrates the data parallelism of the GLU program for Mandelbrot set generation.

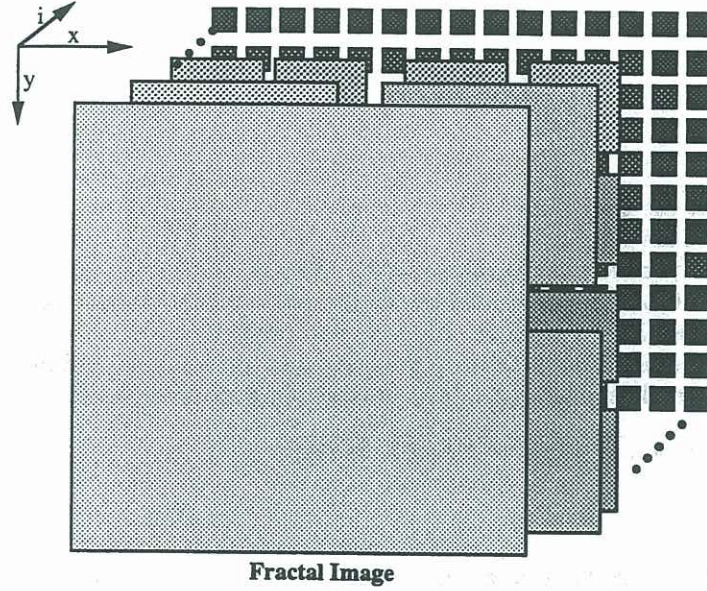


Figure 3: Data Parallelism in Mandelbrot Set Generation

## 5 Example 3: Matrix Multiplication

This example illustrates the use of tournament computation approach (in three dimensions) in matrix computations.

Given two matrices  $A$  of size  $p$  by  $q$  and  $B$  of size  $q$  by  $r$ , compute  $C = AB$  where size of  $C$  is  $p$  by  $r$  such that

$$C_{i,j} = \sum_{k=1}^q A_{i,k} B_{k,j}.$$

The standard algorithm to compute matrix product has a complexity of  $O(n^3)$  where  $n$  is the number of multiplications. (While Strassen's algorithm improves the complexity to  $O(n^{2.81})$ , we will not consider it here.)

The standard algorithm can be expressed as follows. Let  $A$  and  $B$  be two  $n$  by  $n$  matrices where  $n$  is a power of two (without loss of generality). We can divide each of  $A$  and  $B$  into four  $n/2$  by  $n/2$  matrices and express the product of  $A$  and  $B$  in terms of these  $n/2$  by  $n/2$  matrices.

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$$

where

$$C_{11} = A_{11}B_{11} + A_{12}B_{21},$$



$$C_{12} = A_{11}B_{12} + A_{12}B_{22},$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21},$$

$$C_{22} = A_{21}B_{12} + A_{22}B_{22}.$$

Observe that the divide-and-conquer approach can be applied recursively to multiplication of the smaller matrices. Further observe that the computation of each of the eight products of the  $n/2$  by  $n/2$  matrices can occur simultaneously.

We present a GLU program based on the tournament computation approach that completely avoids division through recursion (and the associated overhead cost). Instead, it simply builds the product matrices using addition and juxtaposition from the smallest possible products obtained by procedural multiplication.

```
display( p ) asa.t ( size >= TARGET_ORDER )
where
  index t,i,j,k;

  size = MINIMUM_ORDER fby.t size+size;
  p = mult(a @.j k,b @.i k) fby.t lchild.i lchild.j lchild.k conquer
  where
    conquer = combine( add(p,next.k p),
                      add(next.j p,next.k next.j p),
                      add(next.i p, next.k next.i p),
                      add(next.i next.j p, next.k next.i next.j p)
                    );
    a = in(i,j,MINIMUM_ORDER);
    b = in(i,j,MINIMUM_ORDER);
  end;
end
```

The procedural functions `display`, `combine`, `add`, `mult`, and `in` are not shown here in the interests of clarity.

The program views matrix multiplication as follows: At  $t$ -context 0, the variable  $p$  at  $(i,j,k)$ -context of  $(i,j,k)$  is the multiplication of the  $k^{th}$  value of the  $i^{th}$  row of  $a$  with the  $k^{th}$  value of the  $j^{th}$  column of  $b$ . (The "value" is a matrix of `MINIMUM_ORDER` and the procedural function `mult` performs the multiplication of matrices of that order.)

At successive  $t$ -contexts, the variable  $p$  at the  $(i,j,k)$ -context is the juxtaposition of four matrix sums from the previous  $t$ -context. The four matrix sums are of the matrices at  $k$ -contexts  $2k$  and  $2k+1$  for each of the following  $(i,j)$ -contexts:  $(2i, 2j)$ ,  $(2i, 2j+1)$ ,  $(2i+1, 2j)$ , and  $(2i+1, 2j+1)$ . Note that the size of the matrices being computed quadruples (or the order doubles) with each successive  $t$ .

Eventually, when the size of the matrix is that of the final product, the value (or matrix) corresponding to  $p$  at  $(i,j,k)$ -context  $(0,0,0)$  is the desired result.

This program implicitly expresses a substantial degree of data parallelism – when  $t = 0$ , the multiplication of each rows of matrix **a** (whose elements are of order `MINIMUM_ORDER`) with each column of **b** (whose elements are also of order `MINIMUM_ORDER`) can be performed in parallel. The “tournament-style” addition and juxtaposition of smaller matrices to form bigger matrices with successive  $t$ -contexts also has considerable data-level parallelism which diminishes with increasing  $t$ .

Figure 4 illustrates the parallelism of the GLU program for matrix multiplication.

## 6 Discussion

We have shown through examples that the tournament computation approach can be applied for diverse applications to express implicit data parallelism in GLU. While the divide-and-conquer approach is an alternate approach, it suffers from drawbacks in expressiveness and efficiency. Specifically, when the “tournament” is multi-dimensional, this approach tends to flatten the multi-dimensional structure into a one-dimensional structure that recursion offers. This makes programs less expressive. Efficiency of the divide-and-conquer approach in a parallel implementation is limited by the possible need for centralized information (for globals) and the overhead of copying argument data at each recursive invocation.

## 7 Conclusion

We have shown that large-grain data parallelism in diverse applications can be effectively expressed in the language GLU by using the tournament computation approach. Unlike the divide-and-conquer approach, this approach is both expressive and appears to be efficiently implementable.

## References

- [1] E.A. Ashcroft. Ferds - massive parallelism in Lucid. In *Proceedings of the Fourth Annual IEEE Phoenix Conference on Computers and Communications*, pages 16–21, 1985.
- [2] E.A. Ashcroft. Tournament computations. In *Third International Symposium on Lucid and Intensional Programming*, Queen’s University, Kingston, Ontario, Canada, 1990.
- [3] E.A. Ashcroft, A.A. Faustini, and R. Jagannathan. Intensional parallel programming. In B.K. Szymanski, editor, *Parallel Functional Programming Languages and Environments*. ACM Press, 1991.

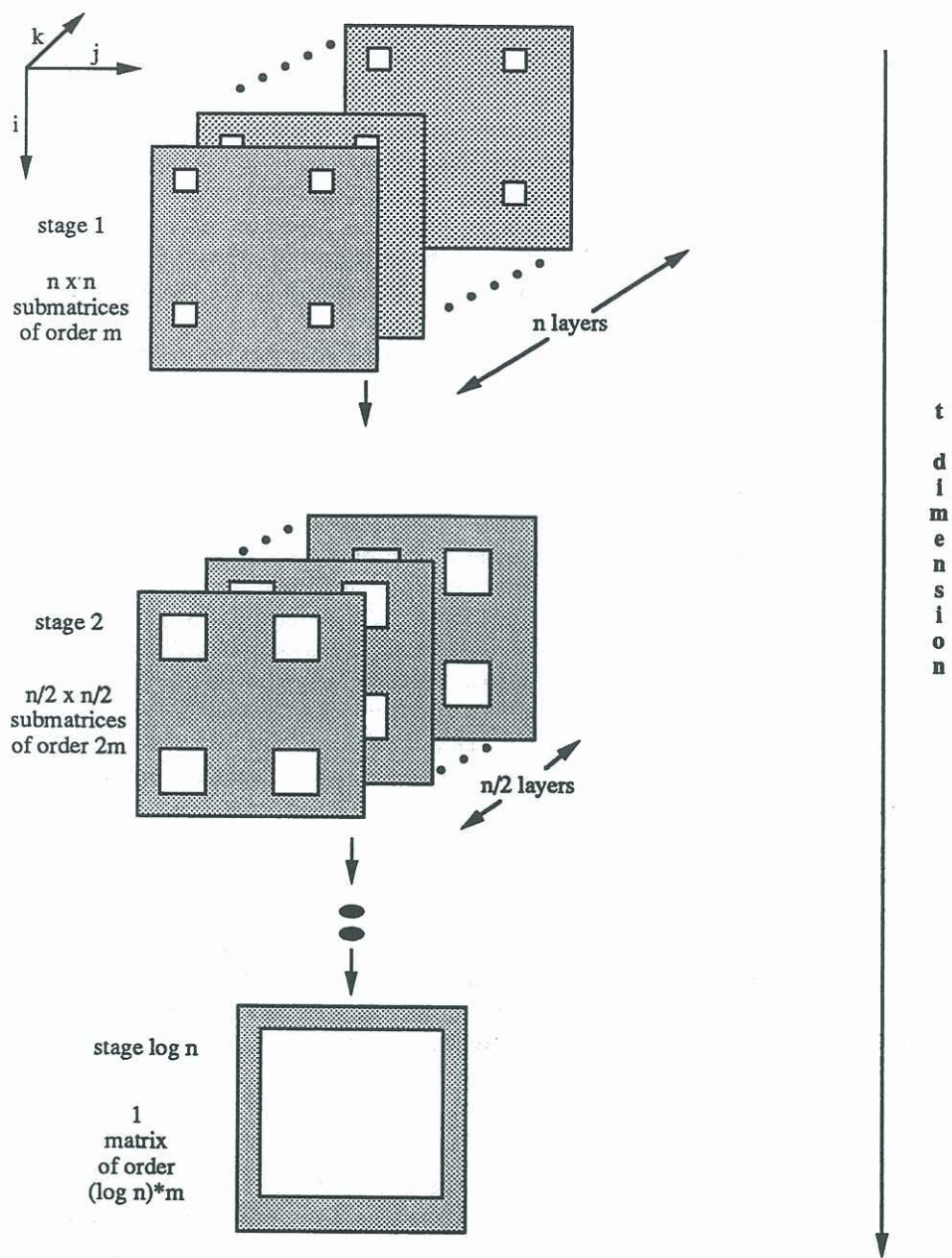


Figure 4: Data Parallelism in Matrix Multiplication



- [4] W. Du. Two indexical parallel programming techniques. In *Fourth International Symposium on Lucid and Intensional Programming*, Computer Science Laboratory, SRI International, Menlo Park, California 94025, 1991.
- [5] A.A. Faustini and R. Jagannathan. Indexical Lucid. In *Fourth International Symposium on Lucid and Intensional Programming*, Computer Science Laboratory, SRI International, Menlo Park, California 94025, 1991.
- [6] W.W. Wadge and E.A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.