LANai

► Myrinet

Languages

► Programming Languages

LAPACK

Jack Dongarra, Piotr Luszczek
University of Tennessee, Knoxville, TN, USA

Definition

LAPACK is a library of Fortran 77 subroutines for solving most commonly occurring problems in dense matrix computations. It has been designed to be efficient on a wide range of modern high-performance computers. The name LAPACK is an acronym for Linear Algebra PACKage. LAPACK can solve systems of linear equations, linear least squares problems, eigenvalue problems, and singular value problems. LAPACK can also handle many associated computations such as matrix factorizations or estimating condition numbers.

Discussion

LAPACK contains driver routines for solving standard types of problems, computational routines to perform a distinct computational task, and auxiliary routines to perform a certain subtask or common low-level computation. Each driver routine typically calls a sequence of computational routines. Taken as a whole, the computational routines can perform a wider range of tasks than are covered by the driver routines. Many of the auxiliary routines may be of use to numerical analysts or software developers, so the Fortran source for these routines has been documented with the same level of detail used for the LAPACK routines and driver routines.

LAPACK is designed to give high efficiency performance on vector processors, high-performance “super-scalar” workstations, and shared memory multiprocessors. It can also be used satisfactorily on all types of scalar machines (PCs, workstations, and mainframes). A distributed memory version of LAPACK, ScaLAPACK [1], has been developed for other types of parallel architectures (e.g., massively parallel SIMD machines, or distributed memory machines).

LAPACK has been designed to supersede LINPACK [2] and EISPACK [3, 4] principally by restructuring the software to achieve much greater efficiency, where possible, on modern high-performance computers, and also by adding extra functionality, by using some new or improved algorithms, and by integrating the two sets of algorithms into a unified package.

LAPACK routines are written so that as much as possible of the computation is performed by calls to the Basic Linear Algebra Subprograms (BLAS) [5–7]. Highly efficient machine-specific implementations of the BLAS are available for many modern high-performance computers. The BLAS enable LAPACK routines to achieve high performance with portable code.

The complete LAPACK package or individual routines from LAPACK are freely available on Netlib [8] and can be obtained via the World Wide Web or anonymous ftp. The LAPACK homepage can be accessed via the following URL address: http://www.netlib.org/lapack/.

LAPACK90 is a Fortran 90 interface to the Fortran 77 LAPACK library. LAPACK++ is an object-oriented C++ extension to the LAPACK library. The CLAPACK library was built using a Fortran to C conversion utility called f2c [9]. The JLAPACK project provides the LAPACK and BLAS numerical subroutines translated...
from their Fortran 77 source into class files, executable by the Java Virtual Machine (JVM) and suitable for use by Java programmers. The ScaLAPACK (or Scalable LAPACK) [1] library includes a subset of LAPACK routines redesigned for distributed memory message-passing MIMD computers and networks of workstations supporting PVM and/or MPI.

Two types of driver routines are provided for solving systems of linear equations: simple and expert. Different driver routines are provided to take advantage of special properties or storage schemes of the system matrix. Supported matrix types and storage schemes include: general, general band, general tridiagonal, symmetric/Hermitian, positive definite, symmetric/Hermitian, positive definite (packed storage), positive definite band, positive definite tridiagonal, indefinite, complex symmetric, and indefinite (packed storage).

**Related Entries**
- **LINPACK Benchmark**
- **ScaLAPACK**

**Bibliography**


---

**Large-Scale Analytics**

- **Massive-Scale Analytics**

**Latency Hiding**

Latency hiding improves machine utilization by enabling the execution of useful operations by a device while it is waiting for a communication operation or memory access to complete. Prefetching, context switching, and instruction-level parallelism are mechanisms for latency hiding.

**Related Entries**
- **Denelcor HEP**
- **Multi-Threaded Processors**
- **Superscalar Processors**

**Law of Diminishing Returns**

- **Amdahl’s Law**

**Laws**

- **Amdahl’s Law**
- **Gustafson’s Law**
- **Little’s Law**
- **Moore’s Law**

**Layout, Array**

**Paul Feautrier**

Ecole Normale Supérieure de Lyon, Lyon, France

**Definitions**

A high-performance architecture needs a fast processor, but a fast processor is useless if a memory subsystem does not provide data at the rate of several words per clock cycle. Run-of-the-mill memory chips in today technology have a latency of the order of ten to a hundred processor cycles, far more than the necessary performance. The usual method for increasing the memory
bandwidth as seen by the processor is to implement a cache, i.e., a small but fast memory which is geared to hold frequently used data. Caches work best when used by programs with almost random but nonuniform addressing patterns. However, high-performance applications, like linear algebra or signal processing, have a tendency to use very regular addressing patterns, which degrade cache performance. In linear algebra codes, and also in stream processing, one finds long sequences of accesses to regularly increasing addresses. In image processing, a template moves regularly across a pixel array.

To take advantage of these regularities, in fine-grain parallel architectures, like SIMD or vector processors, the memory is divided into several independent banks. Adresses are evenly scattered among the banks. If \( B \) is the number of banks, then word \( x \) is located in bank \( x \mod B \) at displacement \( x \div B \). The low-order bits of \( x \) (the byte displacement) do not participate in the computation. In this way, words in consecutive addresses are located in distinct banks, and can, with the proper interface hardware, be accessed in parallel. A problem arise when the requested words are not at consecutive addresses. Array layouts were invented to allow efficient access to various templates.

**Discussion**

**Parallel Memory Access**

There are many ways of taking advantage of such a memory architecture. One possibility is for the processors to access in one memory cycle the \( B \) words which are necessary for a piece of computation, and then to cooperate in computing the result. Consider, for instance, the problem of summing the elements of a vector. One reads \( B \) words, adds them to an accumulator, and proceeds to the next \( B \) words.

Another organization is possible if the program operations can be executed in parallel, and if the data for each operation are regularly located in memory. Consider, for instance, the problem of computing the weighted sum of three rows in a TV image (downsampling). One possibility is to read three words in the same column and do the computation. But since it is likely that \( B \) is much larger than three, the memory subsystem will be under-utilized. The other possibility is to read \( B \) words in the first row, \( B \) words in the second row, and so on, do the summation, and store \( B \) words of the result. This method is more efficient, but places more constraints on the target application.

**Templates and Template Size**

In both situations, the memory is addressed through templates, i.e., finite sets of cells which can move across an array. Consider for instance a vector:

\[
\text{float A[100].}
\]

The template \((0,1,2,3)\) represents four consecutive words. If located (or anchored) at position \( i \), it covers elements \( A[i], A[i+1], A[i+2] \) and \( A[i+3] \) of vector \( A \). But the most interesting case is that of a two-dimensional array, which may be a matrix or an image. The template \(((0,0),(0,1),(1,0),(1,1))\) represents a two-by-two square block. If anchored at position \((i,j)\) in a matrix \( M \), it covers elements \( M[i][j], M[i][j+1], M[i+1][j], \) and \( M[i+1][j+1] \). The first template is one dimensional, and the second one is two dimensional. One may consider templates of arbitrary dimensions, but these two are the most important cases.

The basic problem is to distribute arrays among the available memory banks in such a way that access to all elements of the selected template(s) takes only one memory cycle. It follows that the size of a template is at most equal to the number of banks, \( B \). As has been seen before, memory usage is optimized when templates have exactly \( B \) cells; this is the only choice that is discussed here.

Observe also that, as the previous example has shown, there may be many template selections for a given program. The problem of selecting the best one will not be discussed here.

**Layouts**

A layout is an assignment of bank numbers to array cells. A template can be viewed as a stencil that moves accross an array and shows bank numbers through its openings. A layout is valid for a template position if all shown bank numbers are distinct. In Fig. 1, the layout is valid for all positions of the solid template. In Fig. 2, the layout is valid for all positions of the solid template, and valid for no position of the dashed template.

The reader may have noticed that the information given by array layouts such as Fig. 1 or 2 is not complete. One must also know at which address each cell is located in its bank. This information can always be retrieved
from the layout. Simply order the array cells and allocate each cell to the first free word in its bank. For Fig. 2, the bank address is found equal to the row number. This scheme imposes two constraints on a layout:

- Array cells must be evenly distributed among banks.
- The correspondence between cells and bank addresses cannot be too complex, since each processor must know and use it.

Depending on the application, one may want that a layout be valid at some positions in the array, or at every possible position. For instance, in linear algebra, it is enough to move the template in such a way that each entry is covered once and only once. One says that the template tesselates or tiles the array domain. Once a tiling has been found one simply numbers each square of the template from 0 to \( B - 1 \), and reproduces this assignment (perhaps after a permutation) at all positions of the tile. This insures that the array cells are evenly distributed among the banks. In other applications, like image processing, the template must be moved at every position in the array. Imagine for instance that a smoothing operator must be applied at all pixels of an image. In fact, the two situations are equivalent: If a template tiles an array and if block numbers are assigned as explained above, then the layout is valid for every position. For a proof, see [4].

### Skewing Schemes: Vector Processing

Consider the following example:

\[
\text{for}(i=0; \ i<\ n; \ i=i+1) \\
X[i] = Y[i] + Z[i],
\]

to be run on a SIMD architecture with \( B \) processing elements. If the layout of \( Y \) is such that \( Y[i] \) is stored in bank \( i \mod B \) at address \( i \div B \), \( B \) elements of \( Y \) can be fetched in one memory cycle. After another fetch of \( B \) words of \( Z \), the SIMD processor can execute \( B \) addition in parallel. Another memory cycle is needed to store the \( B \) results. Note that this scheme has the added advantage that all groups of \( B \) words are at the same address in each bank, thus fitting nicely in the SIMD paradigm.

Suppose now that the above loop is altered to read:

\[
\text{for}(i=0; \ i<\ n; \ i=i+1) \\
X[d*i] = Y[d*i] + Z[d*i],
\]

where \( d \) is a constant (the step). The block number \( d.i \mod B \) is equal to \( d.i - k.B \) for some \( k \), and hence is a multiple of the greatest common divisor (gcd) \( g \) of \( d \) and \( B \). It follows that only \( B/g \) words can be fetched in one cycle, and the performance is divided by \( g \). Since \( B \) is usually a power of 2 in order to simplify the computation of \( x \mod B \) and \( x \div B \), a loss of performance will be incurred whenever \( d \) is even. One can always invent layouts that support non-unit step accesses. For instance, if \( d = 2 \), assign cell \( i \) to bank \( (i \div 2) \mod B \).

However, in many algorithms that use non-unit steps, like the Fast Fourier Transform, \( d \) is a variable. It is not possible to accommodate all its possible values without costly remapping.

### Skewing Schemes: Matrix Processing

In C, the canonical method for storing a matrix is to concatenate its rows in the order of their subscripts. If the matrix is of size \( M \times N \), the displacement of element \((i,j)\) is \( N.i + j \). In Fortran, the convention is reversed, but the conclusions are the same, \textit{mutatis mutandis}.

One may assume that the row length \( N \) is a multiple of \( B \), by padding if necessary; hence, all matrices may be assumed to have size \( M \times B \). As a consequence, element \((i,j)\) is allocated to bank \( j \mod B \).
or are regularly permuted from tile to tile. Then the
layout will obviously be valid for each tile. The striking
fact is that the layout will also be valid when the template
is offset from a tile position. Observe that the previous
result on matrix layouts corresponds to the many ways
of tiling the plane with $B$-sized linear templates.

A tiling is defined by a set of vectors $v_1, \ldots, v_n$ (trans-
lations). If a template instance has been anchored at
position $(i,j)$, then there are other instances at posi-
tions $(i,j) + (i,j) + \ldots + v_n$. The translations $v_1, \ldots, v_n$
must be selected in such a way that the translates do not
overlap.

If an algorithm needs several templates, the first
condition is that each of them tiles the plane. But each
tiling defines a layout, and these layouts must be com-
patible. One can show that the compatibility condition
is that the translation vectors are the same for all tem-
plates. Since a template may tile the plane in several dif-
ferent ways, finding the right tiling may prove a difficult
combinatorial problem.

**Pipelined Processors and GPU**

Consider the loop:

\[
\text{for } (i=0; i<n; i=i+1) \\
\quad s = s + A[i],
\]

to be run on a pipelined processor which executes one
addition per cycle. If the memory latency is equal to $B$
processor cycles, the memory banks can be activated in
succession, in such a way that, after a prelude of $B$ cycles,
the processor receives an element of $A$ at each cycle. As
for parallel processors, if the loop has a step of $d$, the
performance will be reduced by a factor of $\gcd(d,B)$.

While today vector processors are confined to
niche applications, the same problem is reemerging for
Graphical Processing Units (GPU), which can access
$B$ words in parallel under the name “Global Memory
Access Coalescing.”

**Reordering**

With nonstandard layouts, the results of a parallel access
may not be returned in the natural order. Consider, for
instance, the problem of having parallel access both for
rows and for the main diagonal of a $4 \times 4$ matrix on a $4$
banks memory. The skewing factor $S$ must be such that
$S + 1$ is relatively prime to 4: $S = 2$ is a valid choice. The
corresponding layout is shown on Fig. 4. The integer in
row $i$ and column $j$ is the number of the bank which
holds element $(i,j)$ of the matrix. One can see that
bank 0 holds element $(0,0)$, bank 1 holds element $(3,3)$,
and so on. If the banks are connected to processors in

---

**Skewing Schemes: More Templates**

It is clear that a more general theory is needed when the
subject algorithm needs to use several templates at the
same time. As seen above, a template has a valid layout if
and only if it tiles the two-dimensional plane. This result
seems natural if the template is required to cover once
and only once each cell of the underlying array. If bank
numbers are assigned in the same order in each tile [4]
or are regularly permuted from tile to tile [3] then the
layout will obviously be valid for each tile. The striking
fact is that the layout will also be valid when the template
is offset from a tile position. Observe that the previous

### Table Example

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

The effect of skewing

- **Layout, Array. Fig. 3** The effect of skewing

If the algorithm accesses a matrix by rows, as in:

\[
\text{for } (j=0; j<N, j++) \\
\quad \ldots \ X[i][j] \ldots
\]

performance will be maximal. However, when accessing
columns, consecutive items will be separated by $B$ rows,
and be located all in the same bank. All parallelism will
be lost.

A better solution is skewing (Fig. 3). The cell $(i,j)$ is
now allocated to bank $(S_i + j) \mod B$. $S$ is the skewing
factor. Here, two consecutive cells in a row are still at a
distance of 1; hence, a row can still be accessed in one
cycle. But two consecutive cells in a column, $(i,j)$ and
$(i+1,j)$ are at a distance of $S$. If $S$ is selected to be rela-
tively prime to $B$ (for instance, $S = 1$ if $B$ is even Fig. 3
(b)) access to a column will also take one cycle.

This raises the question of whether other patterns
can be accessed in parallel. Consider the problem of
accessing the main diagonal of a matrix. In the above
skewing scheme, the distance from cell $(i,i)$ to $(i+1,j+1)$
is $S+1$, and $S$ and $S+1$ both cannot be odd. Hence, access-
ing both columns and diagonals (or both rows and
diagonals) in parallel with the same skewing scheme is
impossible when $B$ is even.
the natural order, the elements of the main diagonal are returned in the order \((0,0),(3,3),(2,2),(1,1)\). This might be unimportant in some cases, for instance, if the elements of the main diagonal are to be summed (the trace computation), since addition is associative and commutative.

In other cases, returned values must be reordered. This can be done by inserting a \(B\) words fast memory between the main memory subsystem and the processors. One can also use a permutation network which may route a datum from any bank to any processor.

The Number of Banks
Remember that in vector processing, when the step \(d\) is not equal to one, performance is degraded by a factor \(\text{gcd}(d,B)\). It is tempting to use a prime number for \(B\), since then the gcd will always be one except when \(d\) is a multiple of \(B\). However, having \(B\) a power of two considerably simplifies the addressing hardware, since computing \(x \mod B\) and \(x \div B\) is done just by routing wires. This has lead to the search for prime numbers with easy division, which are one of the forms \(2^p \pm 1\). See [2] for a discussion of this point.

Bibliography

"LBNL Climate Computer"

"Green Flash: Climate Machine (LBNL)"

"libflame"

FIELD G. VAN ZEE\(^1\), ERNIE CHAN\(^2\)
ROBERT A. VAN DE GEIJN\(^1\)
\(^1\)The University of Texas at Austin, Austin, TX, USA
\(^2\)NVIDIA Corporation, Santa Clara, CA, USA

Synonyms
FLAME

Definition
libflame is a software library for dense matrix computations.

Discussion
Introduction
The libflame library is a modern, high-performance dense linear algebra library that is extensible, easy to use, and available under an open source license. It is both a framework for developing dense linear algebra solutions and a ready-made library that strives to be user-friendly while offering competitive (and in many cases superior) real-world performance when compared to the more traditional Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) libraries. It was developed as part of the FLAME project.

The FLAME Project
A solution based on fundamental computer science. The FLAME project advocates a more stylized notation for expressing loop-based linear algebra algorithms. This notation, illustrated in Fig. 1, closely resembles how matrix algorithms are illustrated with pictures. The notation facilitates formal derivation of algorithms so that the algorithms are developed hand-in-hand with their proof of correctness. For a given operation this yields a family of algorithms so that the best option for a given situation (e.g., problem size or architecture) can be chosen.
Algorithm: $A \leftarrow \text{CHOL\_BLK}(A)$

Partition $A \rightarrow \begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix}$

where $A_{TL}$ is $0 \times 0$
while $m(A_{TL}) < m(A)$ do
  Determine block size $b$
  Repartition
  $\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix}$
  where $A_{11}$ is $b \times b$
  $A_{11} := A_{11} - A_{10} A_{11}^T$ (SYRK)
  $A_{21} := A_{21} - A_{20} A_{11}^T$ (GEMM)
  $A_{11} := \text{CHOL}(A_{11})$ (CHOL)
  $A_{21} := A_{21} A_{11}^T$ (TRSM)

Continue with
  $\begin{pmatrix} A_{TL} & A_{TR} \\ A_{BL} & A_{BR} \end{pmatrix} \rightarrow \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix}$

endwhile

libflame. Fig. 1 Blocked Cholesky factorization expressed with FLAME notation. Subproblems annotated as SYRK, GEMM, and TRSM correspond to level-3 BLAS operations while CHOL performs a Cholesky factorization of the submatrix $A_{11}$

Object-based abstractions and API. The libflame library is built around opaque structures that hide implementation details of matrices, such as data layout, and exports object-based programming interfaces to operate upon these structures. FLAME algorithms are expressed (and coded) in terms of smaller operations on sub-partitions of the matrix operands. This abstraction facilitates programming without array or loop indices, which allows the user to avoid index-related programming errors altogether, as illustrated in Fig. 2.

Educational value. The clean abstractions afforded by the notation and API, coupled with the systematic methodology for deriving algorithms, make FLAME well suited for instruction of high-performance linear algebra courses at the undergraduate and graduate level.

The libflame Library
The techniques developed as part of the FLAME project have been instantiated in the libflame library.

A complete dense linear algebra framework. The libflame library provides ready-made implementations of common linear algebra operations. However, libflame differs from traditional libraries in two important ways. First, it provides families of algorithms for each operation so that the best can be chosen for a given circumstance. Second, it provides a framework for building complete custom linear algebra codes. This makes it a flexible and useful environment for prototyping new linear algebra solutions.

High performance. The libflame library dispels the myth that user- and programmer-friendly linear algebra codes cannot yield high performance. Implementations of operations such as Cholesky factorization and triangular matrix inversion often outperform the corresponding implementations currently available in traditional libraries. For sequential codes, this is often simply the result of implementing the operations with algorithmic variants that cast the bulk of their computation in terms of higher-performing matrix operations such as general matrix rank-k updates.

Support for nontraditional storage formats. The libflame library supports both column-major and row-major order (“flat”) storage of matrices. Indeed, it allows for general row and column strides to be specified. In addition, the library supports storing matrices hierarchically, by blocks, which can yield performance gains through improved spatial locality. Previous efforts [2] at implementing storage-by-blocks focused on solutions that required intricate indexing. libflame simplifies the implementation by allowing the elements within matrix objects to be references to smaller submatrices. Furthermore, the library provides user-level APIs which allow application programmers to read and modify individual elements and submatrices while simultaneously abstracting away the details of the storage scheme.

Supported operations. As of this writing, the following operations have been implemented within libflame to operate on both “flat” and hierarchical matrix objects:

- **Factorizations.** Cholesky; LU with partial pivoting; QR and LQ.
- **Inversions.** Triangular; Symmetric/Hermitian Positive Definite.
libflame Fig. 2 The algorithm shown in Fig. 1 implemented with the FLAME/C API

- **Solvers.** Linear systems via any of the factorizations above; Solution of the Triangular Sylvester equation.

- **Level-3 BLAS.** General, Triangular, Symmetric, and Hermitian matrix-matrix multiplies; Symmetric and Hermitian rank-k and rank-k updates; Triangular solve with multiple right-hand sides.

- **Miscellaneous.** Triangular-transpose matrix multiply; Matrix-matrix `AXPY` and `COPY`; and several other supporting operations.

**Cross-platform support.** The libflame distribution includes build systems for both GNU/Linux and Microsoft Windows. The GNU/Linux build system provides a configuration script via GNU `autoconf` and employs GNU `make` for compilation and installation, and thus adheres to the well-established "configure; make; make install" sequence of build commands.

**Compatibility with LAPACK.** A set of compatibility routines is provided that maps conventional LAPACK invocations to their corresponding implementations within `libflame`. This compatibility layer is optional and may be disabled at configure-time in order to allow the application programmer to continue to...
use legacy LAPACK implementations alongside native libflame functions.

**Support for Parallel Computation**

**Multithreaded BLAS.** A simple technique for exploiting thread-level parallelism is to link the sequential algorithms for flat matrices in libflame to multithreaded BLAS libraries where each call to a BLAS routine is parallelized individually. Inherent synchronization points occur between successive calls to BLAS routines and thus restrict opportunities for parallelization between subproblems. Routines are available within libflame to adjust the block size of these sequential implementations in order to attain the best performance from the multithreaded BLAS libraries.

**SuperMatrix.** The fundamental bottleneck to exploiting parallelism directly within many linear algebra codes is the web of data dependencies that frequently exists between subproblems. As part of libflame, the SuperMatrix runtime system has been developed to detect and analyze dependencies found within algorithms-by-blocks (algorithms whose subproblems operate only on block operands). Once data dependencies are known, the system schedules sub-operations to independent threads of execution. This system is completely abstracted from the algorithm that is being parallelized and requires virtually no change to the algorithm code while exposing abundant high-level parallelism.

The directed acyclic graph for the Cholesky factorization, given a $4 \times 4$ matrix of blocks (a hierarchical
matrix stored with one level of blocking), is shown in Fig. 3. The nodes of the graph represent tasks that operate over different submatrix blocks, and the edges represent data dependencies between tasks. The subscripts in the names of each task denote the order in which the tasks would typically be executed in a sequential algorithm-by-blocks. Many opportunities for parallelism exist within the directed acyclic graph such as TRSM1, TRSM2, and TRSM3 being able to be executed in parallel.

PLAPACK. FLAME and libflame have their roots in the distributed-memory parallel PLAPACK library. Integrating PLAPACK into libflame is a current effort.

Related Entries
- BLAS (Basic Linear Algebra Subprograms)
- LAPACK
- PLAPACK

Bibliographic Notes and Further Reading

The FLAME notation and APIs have their root in the Parallel Linear Algebra Package (PLAPACK). Eventually the insights gained for implementing distributed-memory libraries found their way back to the sequential library libflame.

The first papers that outlined the vision that became the FLAME project were published in 2001 [3, 4]. A book has been published that describes in detail the formal derivation of dense matrix algorithms using the FLAME methodology [5]. How scheduling of tasks is supported was first published in [1] and discussed in more detail in [5]. An overview of libflame can be found in [7] and an extensive reference guide has been published [8].

Bibliography

Linda is not concerned with other programming language issues such as control structures (testing, looping), file I/O, variable declaration, etc. Thus, it is orthogonal to issues of traditional programming languages, and is normally embedded in a sequential language such as C, Fortran, Java, etc., resulting in a Linda dialect of that language (C-Linda, Fortran-Linda, Java-Linda, etc). It is important to distinguish between Linda as a conceptual model for coordination and a concrete X-Linda implementation. The latter necessarily imposes restrictions and limitations on the former.

**Tuples and Tuple Spaces**

Tuples (too-pulls) are the fundamental data objects in Linda. A tuple is an ordered, typed collection of any number of elements. These elements can be whatever values or objects are supported by the host language. Elements can also be typed place holders, or *formals*, which will act as wild cards. Here are some example tuples:

- ("counter", 3)
- ("table", 4, 6, 5.67)
- ("table, i, j, ?a")

Tuples exist in tuple spaces. Processes communicate and synchronize with one another by creating and consuming tuples from a common tuple space. Tuple spaces exist, at least abstractly, apart from the processes using them. Tuple space is sometimes referred to as an associative memory. A tuple establishes associations between the values of its constituent elements. These associations are exploited by specifying some subset of the values to reference the tuple as a whole.

**Tuple Space Operations**

There are two operations that create tuples: `out` and `eval`. `Out` simply evaluates the expressions that yield its elements in an unspecified order and then places the resulting tuple in tuple space:

````
out("counter", 5)
```

A tuple space is a multiset: if the above operation is repeated *N* times (and no other operations are done), then the tuple space will contain *N* identical ("counter", 5) tuples.

`Eval` is identical to `out`, except that the tuple is placed into tuple space without completing the evaluation of its elements. Instead, new execution entities are created to evaluate the elements of the tuple (depending on the implementation, these entities might be threads or processes). Once all elements have been evaluated, the tuple is indistinguishable from one created via `out`. `Eval` is typically used with tuples that contain function calls, and is an idiomatic and portable way to create parallelism in Linda.

For example, a process might create a new worker by doing:

````
eval("new worker", worker(i))
```

`Eval` will immediately return control to the calling program, but in the background a new execution entity will begin computing `worker(i)`. When that subroutine returns, its value will be the second element of a “new worker” tuple.

There are two operations that can extract tuples from tuple space: `in` and `rd` (Traditionally read was spelled *rd* to avoid collisions with read from the C library. The exact verbs used have varied somewhat, with `store` and `write` common variants of `out`, and `take` a common variant of `in`). These operations take a template and perform a match of its arguments against the tuples currently in tuple space. A template is similar to a tuple, consisting of some number of actual and formal fields.

If a match is found, the tuple is returned to the process performing the `in` or `rd`. In the case of `in`, the tuple is deleted from tuple space after matching, whereas `rd` leaves the tuple in tuple space. If multiple matches are available, one is chosen arbitrarily. If no matches are available, the requesting process blocks until a match is available. Tuples are always created and consumed atomically.

In order to match, the tuple and template must:

1. Agree on the number of fields
2. Agree on the types of corresponding fields
3. Agree in the values of corresponding actual fields
4. Have no corresponding formals (this is unusual)

`?` indicates that the argument is a place-holder, or formal argument. In this case, only the types must match, and as a side effect, the formal variable in the template will be assigned the value from the tuple. Formals normally appear only in templates, but in some limited circumstances can appear in tuples as well (it is
not uncommon for a particular implementation to omit support for formals in tuples). Some work was also proposed to allow a “ticking” type modifier to enable specification of elements still under evaluation. Absent this, the matching above implicitly enforces matches on tuples whose elements have all completed evaluation.

For example, the template:

\[ \text{in(“counter”, 5)} \]

would match the above tuple, as would

\[ \text{in(“counter”, ?i) also setting i to 5} \]
\[ \text{in(“counter”, i) assuming i was bound to 5} \]

But these templates would not match:

\[ \text{in(“counter”, 5, 5) rule 1} \]
\[ \text{in(“counter”, 5.0) rule 2} \]
\[ \text{in(“counter”, 6) rule 3} \]

In addition to in and rd, there are non-blocking variants inp and rdp, which return FALSE rather than blocking. These variants can be useful in certain circumstances, but their use has generally been discouraged since it introduces timing dependencies.

In the above examples, note that the first field is always a string that serves to name the tuple. This is not required, but is a common useful idiom. It also provides a useful hint to the Linda implementation that can improve performance, as discussed below.

**Linda Examples**

Linda programmers are encouraged to think of coordination in terms of data structure manipulations rather than message passing, and a number of useful idioms have been developed along these lines. Indeed, Linda can be viewed as a means to generalize Wirth’s “Algorithms + Data Structures = Programs”: “Algorithms + Distributed Data Structures = Parallel Programs” [2]. That is, tuples can be used to create rich, complex distributed data structures in tuple space. Distributed data structures can be shared and manipulated by cooperating processes that transform them from an initial configuration to a final result. In this sense, the development of a parallel program using Linda remains conceptually close to the development of a sequential (ordinary) program. Two very simple, yet common, examples are a FIFO queue and a master/worker pattern.

**FIFO Queue**

In the FIFO queue (Fig. 1), each queue is represented by a number of tuples: one for each element currently in the queue, plus two counter tuples that indicate the current position of the head and tail of the queue. Pushing a value onto the tail of the queue is accomplished by assigning it the current value of the tail counter, and then incrementing the tail, and similarly for popping. Important to note is that no additional synchronization is needed to allow multiple processes to push and pop from the queue “simultaneously.” The head and tail tuples act as semaphores, negotiating access to each position of the queue automatically.

**Master/Worker Pattern**

In the master/worker example (Fig. 2), the master initially creates some number of worker processes via eval. It then creates task tuples, one per task, and waits for the results. Once all the results are received, a special “poison” task is created that will cause all the workers to shut down.

The workers simply sit in a loop, waiting to get a tuple that describes a task. When they do, if it is a normal task, they process it into a result tuple that will be consumed by the master. In the case of a poison task, they immediately use its elements to create a new task tuple to replace the one they consumed, poisoning another worker, and then finish.

```linda
init(qname) {  
  out(qname, ‘head’, 0)  
  out(qname, ‘tail’, 0)  
}

push(qname, elem) {  
  in(qname, ‘tail’, ?p)  
  out(qname, ‘elem’, p, elem)  
  out(qname, ‘tail’, p+1)  
}

pop(qname) {  
  in(qname, ‘head’, ?p)  
  in(qname, ‘elem’, p, ?elem)  
  out(qname, ‘head’, p+1)  
  return elem  
}
```

Linda. Fig. 1 A FIFO queue written in Linda
This is the simplest sort of master/worker pattern one could imagine, yet it still presents a number of interesting features. Process startup and shutdown is represented. Load balancing occurs automatically, since slow workers or large tasks will simply take longer, and as long as there are significantly more tasks than workers the processing time will tend to balance.

The code shown can be modified in a number of simple but interesting ways:

1. If there are a very large number of tasks, rather than having the master create all the task tuples at once (and thereby possibly filling memory) it can create an initial pool. Then, each time it receives a result it can create another task, thus maintaining a reasonable number of tasks at all times.

2. If the master needs to process the results in task order, rather than as arbitrarily received, the in statement that retrieves the result can be modified to make taskid an actual, i.e., in("result", i, ?result).

3. The program can be made fault-tolerant by wrapping the worker's inner loop in a transaction (for Linda implementations that support transactions) [3]. If a worker dies during the processing of a task, the original task tuple will automatically be returned when the transaction fails, and another worker will pick up the task.

4. The program can be easily generalized to have new task tuples generated by the workers (in a sense, the propagation of the poison task is an example of this). This could, for example, be used to implement a dynamic divide-and-conquer decomposition.

Implementations

Linda has been implemented on virtually every kind of parallel hardware ever created, including shared-memory multiprocessors, distributed-memory multi processors, non-uniform memory access (NUMA) multiprocessors, clusters, and custom hardware.

Linda is an extremely minimalist language design, and naive, inefficient implementations are deceptively simple to build. Most take the approach of building a runtime library that simply implements tuple space as a list of tuples and performs tuple matching by exhaustively searching the list. These sorts of implementations led to a suspicion that Linda is not suitable for serious computation. Many critics admired the language's elegance, but insisted that it could not be implemented efficiently enough to compete with other approaches, such as shared memory with locking, or native message passing. In response, considerable work was done at Yale and elsewhere during the 1980s and 1990s to create efficient implementations that could compete with lower-level approaches. The efforts focused on two main directions:

1. Compile-time optimizations. Carriero [4] recognized that tuple matching overhead could be greatly reduced by analyzing tuple usage at compile time and translating it to simpler operations. Tuple operations could be segregated into distinct, non-interacting partitions based on type information available at compile time. Each partition was then classified into one of a number of usage patterns and reduced to an efficient data structure such as a counter, hash table or queue, which could then be searched and updated very efficiently. This work demonstrated that Linda on shared-memory multiprocessors could be as efficient as handwritten codes using low level locks. An important part of the demonstration showed for a variety of
representative problems that on average fewer than two tuples would need to be searched before a match was found.

2. Run-time optimizations. For distributed memory implementations, optimization of tuple storage was key to reducing the overall number of messages required by tuple space operations. Each tuple had a designated node on which it would be stored, usually chosen via hashing. Naively, each out/in pair would require as many as three distinct messages, since out would require a single message sending the tuple to its storage location, and in would require two messages, a request to the storage location and a reply containing the matched tuple. Bjornson [5] built upon Carriero’s compile-time optimizations by analyzing tuple traffic at runtime, recognizing common communication patterns such as broadcast (one-to-many), point-to-point, and many-to-one, and rearranging tuple space storage to minimize the number of messages required. This work demonstrated that most communication in Linda required only one or two sends, just as with message passing.

Overall, this work demonstrated that while in the abstract Linda programs could make unrestrained use of expensive operations such as content addressability and extensive search, in practice any given program was likely to confine itself to well-understood patterns that could be reduced through analysis to much simpler and more efficient low-level operations.

Implementations of Linda closely followed hardware developments in parallel computing. The first implementation was on a custom, bus-based architecture called the S/Net [4], followed by implementations on shared memory multiprocessors (Sequent Balance, Encore Multimax), distributed memory multiprocessors (Intel iPSC/1 and 2 hypercubes, Ncube, IBM SP1), collections of workstations, and true cluster architectures.

In terms of host languages, Linda has been embedded in many of the most commonly used computer languages. Initial imbedding was done in C, but important embeddings were built in Fortran, Lisp, and Prolog. In the 1990s, Java became an important host language for Linda, including the Jini, JavaSpaces [6], and Gigaspaces [7] implementations. More recently, Scientific Computing Associates (SCAI) created a variant of Linda called NetWorkSpaces that is designed for interactive languages such as Perl, Python, R (see below).

Open Linda

Initial compile-time Linda systems can be termed “closed,” in the sense that the Linda compiler needed to see all Linda operations that would ever occur in the program, in order to perform the analysis and optimization. In addition, only a single global tuple space was provided, which was created when the program started and destroyed when it completed. Linda programs tended to be collections of closely related processes that stemmed from the same code, and started, ran for a finite period, and stopped together.

During the 1990s, interest in distributed computing increased dramatically, particularly computing on collections of workstations. These applications consisted of heterogeneous collections of processes that derived from unrelated code, started and stopped independently, and might run indefinitely. The components could be written by different users and using different languages. In 1995, SCAI developed an Open Linda (Open referred to the fact that the universe of tuples remained flexible and open at runtime, rather than fixed and closed at compile time), which was later trademarked as “Paradise.” With Open Linda, tuple spaces became first-class objects. They could be created and destroyed as desired, or persist beyond the lifetime of their creator. Other Open Lindas that were created include JavaSpaces [6], TSpaces [8], and Gigaspaces [7].

In 2005 SCAI created a simplified Open Linda called NetWorkSpaces (NWS) [9], designed specifically for high productivity languages such as Perl, R, Python, Matlab, etc. In NetWorkSpaces, rather than communicating via tuples in tuple spaces, processes use shared variables stored in network spaces. Rather than use associative lookup via matching, in NetWorkSpaces shared data is represented as a name/value binding. Processes can create, consult, and remove bindings in shared network spaces.

Figure 3 contains an excerpt from a NWS code for performing a maximization search, using the Python API. In NWS, store, find, and fetch correspond to Linda’s out, rd, and in, respectively. In the code, a single shared variable “max” is used to hold the current best value found by any worker. The call NetWorkSpace ("my
def init():
    ws=NetWorkSpace('my space')
    ws.store('max', START)

def worker():
    ws=NetWorkSpace('my space')
    for x in MyCandidateList:
        currentMax = ws.find('max')
        y = f(x)
        if y > currentMax:
            currentMax = ws.fetch('max')
            if y > currentMax: currentMax = y
            ws.store('max', currentMax)

Linda. Fig. 3 Excerpt of NetWorkSpaces code for performing a maximization search

space") connects to the space of that name, creating it if necessary, and returning a handle to it.

Key Characteristics of Linda, and Comparison to Other Approaches
Linda presents a number of interesting characteristics that contrast with other approaches to parallel computing:

Portability: A Linda program can run on any parallel computer, regardless of the underlying parallel infrastructure and interconnection, whether it be distributed- or shared-memory or something in between. Most competing parallel approaches are strongly biased toward either shared- or distributed-memory.

Generative Communication: In Linda, when processes communicate, they create tuples, which exist apart from the process that created them. This means that the communication can exist long after the creating process has disappeared, and can be used by processes unaware of that process' existence. This aids debugging, since tuples are objects that can be inspected using tools like TupleScope [10].

Distribution in space and time: Processes collaborating via tuple spaces can be widely separated over both these dimensions. Because Linda forms a virtual shared memory, processes need not share any physical memory. Because tuples exist apart from processes, those processes can communicate with one another even if they don't exist simultaneously.

Content Addressability: Processes can select information based on its content, rather than just a tag or the identity of the process that sent it. For example, one can build a distributed hash table using tuples.

Anonymity: Processes can communicate with one another without knowing one another's location or identity. This allows for a very flexible style of parallel program. For example, in a system called Piranha [11], idle compute cycles are harvested by creating and destroying workers on the fly during a single parallel run. The set of processes participating at any given time changes constantly. Linda allows these dynamic workers to communicate with the master, and even with one another, because processes need not know details about their communication partners, so long as they agree on the tuples they use to communicate.

Simplicity: Linda consists of only four operations, in, out, rd, eval, and two variants, inp and rdp. No additional synchronization functions are needed, nor any administrative functions.

When Linda was invented in the early 1980s, there were two main techniques used for parallel programming: shared memory with low-level locking, and message passing. Each parallel machine vendor chose one of these two styles and built their own, idiosyncratic API. For example, the first distributed-memory parallel computers (Intel iPSC, nCube, and IBM SP1) each had their own proprietary message passing library, which was incompatible with all others. Similarly, the early shared-memory parallel computers (Sequent Balance, Encore Multimax) each had their own way of allocating shared pages and creating synchronization locks.

Linda presented a higher level approach to parallelism that combined the strengths of both. Like shared memory, Linda permits direct access to shared data (no need to map such data onto an "owner" process, as is the case for message passing). Like message passing, Linda self-synchronizes (no need to worry about accessing uninitialized shared data, as is the case for hardware shared memory). Once the corresponding Linda implementation had been ported to a given parallel computer, any Linda program could be simply recompiled and run correctly on that computer.

At the time of this writing, it is fair to say that message passing, as embodied by Message Passing Interface (MPI), has become the dominant paradigm for writing parallel programs. MPI addressed the portability
problem in message passing by creating a uniform library interface for sending and receiving messages. Because of the importance of MPI to parallel programming, it is worth contrasting Linda and MPI.

**Compiler Support:** Linda's compiler support performs both type and syntax checking. Since the Linda compiler understands the types of the data being passed through tuple space, marshalling can be done automatically. The compiler can also detect outs that have no matching in, or vice versa, and issue a warning. In contrast, MPI is a runtime library, and the burdens of type checking and data marshalling fall largely to the programmer.

**Decoupled/anonymous communication:** In Linda, communicating processes are largely decoupled. In MPI, processes must know one another's location and exist simultaneously. In addition, receivers must anticipate sends. If, on the hand, the coordination needed by an application lends itself to a message passing style solution, this is still easy to express in Linda using associative matching. In this case, the compile and runtime optimization techniques discussed will often result in wire traffic similar, if not identical, to that that would have obtained if message passing was used.

**Simplicity:** MPI is a large, complex API. Debugging usually requires the use of a sophisticated parallel debugger. Even simple commands such as send() exist in a number of subtly differently variants. In the spirit of Alan Perlis’s observation “If you have a procedure with 10 parameters, you probably missed some,” the MPI standard grew substantially from MPI 1 to MPI 2. Acknowledging the lack of a simple way to share data, MPI 2 introduced one-sided communication. This was not entirely successful [12]. In contrast, Linda consists of four simple commands.

**Future**

In 2010, parallel computing stands at an interesting crossroads. The largest supercomputer in the world contains 224,000 cpus [13]. Meanwhile, multicore systems are ubiquitous; modern high performance computers typically contain 8–32 cpu cores, with hundreds envisioned in the near future. These two use cases represent two very different extrema on the spectrum of parallel programming. The largest supercomputers require extremely efficient parallel codes in order to scale to tens of thousands of cpus, implying expert assistance by highly skilled programmers, careful coding, optimization and verification of programs, and are well suited to large-scale development efforts in languages such as C and tools such as MPI.

In contrast, nearly everyone with a compute-bound program will have access to a non-trivial parallel computer containing tens or hundreds of cpus, and will need a way to make effective use of those cpus.

An interesting characteristic of many of these users is that, rather than using traditional high performance languages such as C, Fortran, or even Java, they use scripting languages such as Perl, Python or Ruby, or high-productivity environments such as Matlab or R. These languages dramatically improve their productivity in terms of writing and debugging code, and therefore in terms of their overall goals of exploring data or testing theories. That is, they have explicitly opted for simplicity and expressivity in their tool of choice for sequential programming. However, these environments pose major challenges in terms of performance and memory usage. Parallelism on a relatively modest scale can permit these users to compute effectively while remaining comfortably within an environment they find highly productive. But, to be consistent with the choice made when selecting these sequential computing environments, the parallelism tool must also emphasize simplicity and expressivity. High level coordination languages like Linda can provide them with simple tools for accessing that parallelism.

**Influences**

Some of the work influencing Linda includes: Sussman and Steele’s language, Scheme, particularly continuations [14], Dijkstra’s semaphores [15], Halstead’s futures [16], C.A.R. Hoare’s CSP [17], and Shapiro’s Concurrent Prolog [18], particularly the concept of unification.

**History**

Linda was initially conceived by David Gelernter while a graduate student at SUNY Stony Brook in the early 1980s. After taking a faculty position at Yale University in 1982, Gelernter, along with a number of graduate students, most notably Carriero, built a number of Linda implementations as well as high level applications that used Linda.
In the early 1990s Scientific Computing Associates (SCAI), founded by Yale professors Stan Eisenstat and Martin Schultz, began commercial development of Linda. Linda systems created by SCAI were sold in a variety of industries, most notably finance and energy. Large-scale systems, running on hundreds of computers, were built and used in production environments for years at a time, performing tasks such as portfolio pricing, risk analysis, and seismic image processing.

Around 2000, interest grew in more flexible languages and approaches. Java became the rage, particularly in financial organizations, and a number of commercial Java Linda systems were built. Although they varied in detail, all shared many common features: an object-oriented design, open tuple spaces, and support for transactions.

Related Entries
- CSP (Communicating Sequential Processes)
- Logic Languages
- Multilisp
- Top500

Bibliographic Notes and Further Reading
The following papers are good general introductions to Linda: [19, 20]. Carriero and Gelernter used Linda in their introductory parallel programming book [21]. There have been conferences organized around the ideas embodied in Linda [22, 23].

Bibliography
13. Top 500 Supercomputing sites. [cited; Available from: www.top500.org]

Linear Algebra Software

JACK DONGARRA, PIOTR LUSZCZEK
University of Tennessee, Knoxville, TN, USA

Definition
Linear algebra software is software that performs numerical calculations aimed at solving a system of linear equations or related problems such as eigenvalue,
singular value, or condition number computations. Linear algebra software operates within the confines of finite precision floating-point arithmetic and is characterized by its computational complexity with respect to the sizes of matrix and vectors involved. Equally important metric is that of numerical robustness: The linear algebra methods aim to deliver a solution that is as close as possible (in a numerical sense by taking into account the accuracy of the input data) to the true solution (obtained in sufficiently extended floating-point precision) if such a solution exists.

**Discussion**

The increasing availability of high-performance computers has a significant effect on all spheres of scientific computation, including algorithm research and software development in numerical linear algebra. Linear algebra— in particular, the solution of linear systems of equations— lies at the heart of most calculations in scientific computing. There are two broad classes of algorithms: those for dense, and those for sparse matrices. A matrix is called sparse if it has a substantial number of zero elements, making specialized storage and algorithms necessary.

Much of the work in developing linear algebra software for high-performance computers is motivated by the need to solve large problems on the fastest computers available. For the past 15 years or so, there has been a great deal of activity in the area of algorithms and software for solving linear algebra problems. The goal of achieving high performance on codes that are portable across platforms has largely been realized by the identification of linear algebra kernels, the Basic Linear Algebra Subprograms (BLAS). See discussion of the EISPACK, LINPACK, LAPACK, and ScalAPACK libraries, which are expressed in successive levels of the BLAS.

The key insight to designing linear algebra algorithms for high-performance computers is that the frequency with which data are moved between different levels of the memory hierarchy must be minimized in order to attain high performance. Thus, the main algorithmic approach for exploiting both vectorization and parallelism in various implementations is the use of block-partitioned algorithms, particularly in conjunction with highly tuned kernels for performing matrix–vector and matrix–matrix operations (the Level 2 and 3 BLAS).

Algorithms for sparse matrices differ drastically because they have to take full advantage of sparsity and its particular structure. Optimization techniques differ accordingly due to changes in the memory access patterns and data structures. Commonly, then, the sparse methods are discussed separately.

**Dense Linear Algebra Algorithms**

**Origins of Dense Linear Systems**

A major source of large dense linear systems is problems involving the solution of boundary integral equations [1]. These are integral equations defined on the boundary of a region of interest. All examples of practical interest compute some intermediate quantity on a two-dimensional boundary and then use this information to compute the final desired quantity in three-dimensional space. The price one pays for replacing three dimensions with two is that what started as a sparse problem in $O(n^2)$ variables is replaced by a dense problem in $O(n^3)$.

Dense systems of linear equations are found in numerous applications, including:

- Airplane wing design
- Radar cross-section studies
- Flow around ships and other off-shore constructions
- Diffusion of solid bodies in a liquid
- Noise reduction
- Diffusion of light through small particles

The electromagnetics community is a major user of dense linear systems solvers. Of particular interest to this community is the solution of the so-called radar cross-section problem. In this problem, a signal of fixed frequency bounces off an object; the goal is to determine the intensity of the reflected signal in all possible directions. The underlying differential equation may vary, depending on the specific problem. In the design of stealth aircraft, the principal equation is the Helmholtz equation. To solve this equation, researchers use the method of moments [2, 3]. In the case of fluid flow, the problem often involves solving the Laplace or Poisson equation. Here, the boundary integral solution is known as the panel methods [4, 5], so named from the quadrilaterals that discretize and approximate a
structure such as an airplane. Generally, these methods are called \textit{boundary element methods}.

Use of these methods produces a dense linear system of size $O(N)$ by $O(N)$, where $N$ is the number of boundary points (or panels) being used. It is not unusual to see size $3N$ by $3N$, because of three physical quantities of interest at every boundary element.

A typical approach to solving such systems is to use LU factorization. Each entry of the matrix is computed as an interaction of two boundary elements. Often, many integrals must be computed. In many instances, the time required to compute the matrix is considerably larger than the time for solution.

The builders of stealth technology who are interested in radar cross sections are using direct Gaussian elimination methods for solving dense linear systems. These systems are always symmetric and complex, but not Hermitian.

\section*{Basic Elements in Dense Linear Algebra Methods}

Common operations involving dense matrices are the solution of linear systems $Ax = b$, the least squares solution of over- or underdetermined systems $\min_x \|Ax - b\|_2$, and the computation of eigenvalues and vectors $Ax = \lambda x$. Although these problems are formulated as matrix–vector equations, their solution involves a definite matrix–matrix component. For instance, in order to solve a linear system, the coefficient matrix is first factored as $A = LU$ (or $A = U^T U$ in the case of symmetry) where $L$ and $U$ are lower and upper triangular matrices, respectively. It is a common feature of these matrix–matrix operations that they take, on a matrix of size $n$ by $n$, a number of operations proportional to $n^3$, a factor $n$ more than the number of data elements involved.

It is possible to identify three levels of linear algebra operations:

\begin{itemize}
  \item Level 1: vector–vector operations such as the update $y \leftarrow y + ax \times y$ and the inner product $a = x^T y$. These operations involve (for vectors of length $n$) $O(n)$ data and $O(n)$ operations.
  \item Level 2: matrix–vector operations such as the matrix–vector product. These involve $O(n^2)$ operations on $O(n^2)$ data.
  \item Level 3: matrix–matrix operations such as the matrix–matrix product $C = AB$. These involve $O(n^3)$ operations on $O(n^3)$ data.
\end{itemize}

These three levels of operations have been realized in a software standard known as the Basic Linear Algebra Subprograms (BLAS) [6–8]. Although BLAS routines are freely available on the net, many computer vendors supply a tuned, often assembly-coded, BLAS library optimized for their particular architecture. The relation between the number of operations and the amount of data is crucial for the performance of the algorithm.

\section*{Optimized Implementations}

Due to its established position, BLAS form the foundation for optimized dense linear algebra operation on homogeneous as well as heterogeneous multicore architectures. Similarly, LAPACK API serves as the de facto standard for accessing various decompositional techniques. Currently active optimized implementations include ACML from AMD, cuBLAS from NVIDIA, LibSci from Cray, MKL from Intel, and open source ATLAS. In the distributed memory regime, Basic Linear Algebra Communication Subroutines (BLACS), PBLAS, and ScaLAPACK serve the corresponding role of defining an API and are accompanied by vendor-optimized implementations.

\section*{Sparse Linear Algebra Methods}

\section*{Origin of Sparse Linear Systems}

The most common source of sparse linear systems is the numerical solution of partial differential equations. Many physical problems, such as fluid flow or elasticity, can be described by partial differential equations (PDE). These are implicit descriptions of a physical model, describing some internal relation such as stress forces. In order to arrive at an explicit description of the shape of the object or the temperature distribution, the PDE needs to be solved, and for this, numerical methods are required.

\section*{Discretized Partial Differential Equations}

Several methods for the numerical solution of PDEs exist, the most common ones being the methods of finite elements, finite differences, and finite volumes. A common feature of these is that they identify discrete
points in the physical object, and give a set of equations relating these points.

Typically, only points that are physically close together are related to each other in this way. This gives a matrix structure with very few nonzero elements per row, and the nonzeros are often confined to a “band” in the matrix.

**Sparse Matrix Structure**

Matrices from discretized partial differential equations contain so many zero elements that it pays to find a storage structure that avoids storing these zeros. The resulting memory savings, however, are offset by an increase in programming complexity, and by decreased efficiency of even simple operations such as the matrix–vector product.

More complicated operations, such as solving a linear system, with such a sparse matrix present a next level of complication, as both the inverse and the $LU$ factorization of a sparse matrix are not as sparse, thus needing considerably more storage. Specifically, the inverse of a band sparse matrix is a full matrix, and factoring such a sparse matrix fills in the band completely.

**Basic Elements in Sparse Linear Algebra Methods**

Methods for sparse systems use, like those for dense systems, vector–vector, matrix–vector, and matrix–matrix operations. However, there are some important differences.

For iterative methods, there are almost no matrix–matrix operations. Since most modern architectures prefer these Level 3 operations, the performance of iterative methods will be limited from the outset.

An even more serious objection is that the sparsity of the matrix implies that indirect addressing is used for retrieving elements. For example, in the popular row-compressed matrix storage format, the matrix–vector multiplication looks like

\[
\text{for } i=1 \ldots n \\
\quad p\leftarrow \text{pointer to row } i \\
\quad \text{for } j=1 \ldots n_i \\
\quad \quad y_i \leftarrow y_i + a(p+j)^* (c(p+j))
\]

where $n_i$ is the number of nonzeros in row $i$, and $p(.)$ is an array of column indices. A number of such algorithms for several sparse data formats are given in [9].

Direct methods can have a BLAS 3 component if they are a type of dissection method. However, in a given sparse problem, the denser the matrices are, the smaller they are on average. They are also not general full matrices, but only banded. Thus, very high performance should not be expected from such methods either.

**Direct Sparse Solution Methods**

For the solution of a linear system, one needs to factor the coefficient matrix. Any direct method is a variant of Gaussian elimination. As remarked above, for a sparse matrix, this fills in the band in which the nonzero elements are contained. In order to minimize the storage needed for the factorization, research has focused on finding suitable orderings of the matrix. Re-ordering the equations by a symmetric permutation of the matrix does not change the numerical properties of the system in many cases, and it can potentially give large savings in storage. In general, direct methods do not make use of the numerical properties of the linear system, and thus, their execution time is affected in a major way by the structural properties of the input matrix.

**Iterative Solution Methods**

Direct methods, as sketched above, have some pleasant properties. Foremost is the fact that their time to solution is predictable, either a priori, or after determining the matrix ordering. This is due to the fact that the method does not rely on numerical properties of the coefficient matrix, but only on its structure. On the other hand, the amount of fill can be substantial, and with it the execution time. For large-scale applications, the storage requirements for a realistic size problem can simply be prohibitive.

Iterative methods have far lower storage demands. Typically, the storage, and the cost per iteration with it, is of the order of the matrix storage. However, the number of iterations strongly depends on properties of the linear system, and is at best known up to an order estimate; for difficult problems, the methods may not even converge due to accumulated round-off errors.

**Basic Iteration Procedure**

In its most informal sense, an iterative method in each iteration locates an approximation to the solution of the problem, measures the error between the approximation and the true solution, and based on the
error measurement improves on the approximation by constructing a next iterate. This process repeats until the error measurement is deemed small enough.

**Stationary Iterative Methods**
The simplest iterative methods are the “stationary iterative methods.” They are based on finding a matrix $M$ that is, in some sense, “close” to the coefficient matrix $A$. Instead of solving $Ax = b$, which is deemed computationally infeasible, $Mx_1 = b$ is solved. The true measure of how well $x_1$ approximates $x$ is the error $e_1 = x_1 - x$, but, since the true solution $x$ is not known, this quantity is not computable. Instead, the “residual”: $r_1 = Ae_1 = Ax_1 - b$ is monitored since it is a computable quantity. It is easy to see that the true solution satisfies $x = A^{-1}b = x_1 - A^{-1}r_1$, so, replacing $A^{-1}$ with $M^{-1}$ in this relation, $x_2 = x_1 - M^{-1}r_1$ can be defined.

Stationary methods are easily analyzed: It can be proven that $r_i \to 0$ if all eigenvalues $\lambda = \lambda(I - AM^{-1})$ satisfy $|\lambda| < 1$. For certain classes of $A$ and $M$, this inequality is automatically satisfied [10, 11].

**Krylov Space Methods**
The most popular class of iterative methods nowadays is that of “Krylov space methods.” The basic idea there is to construct the residuals such that the n-th residual $r_n$ is obtained from the first by multiplication by some polynomial in the coefficient matrix $A$, that is, $r_n = P^{n-1}(A)r_1$. The properties of the method then follow from the properties of the actual polynomial [12–14].

Most often, these iteration polynomials are chosen such that the residuals are orthogonal under some inner product. From this, one usually obtains some minimization property, though not necessarily a minimization of the error.

Since the iteration polynomials are of increasing degree, it is easy to see that the main operation in each iteration is one matrix–vector multiplication. Additionally, some vector operations, including inner products in the orthogonalization step, are needed.

**Optimized Implementations**
Sparse BLAS used to be available from some of the vendors. The more common approach is to have the dedicated sparse packages available publicly and have the vendors tune them for their software offerings. Examples of direct sparse solvers include MUMPS, SuperLU, UMFPACK, and Paradiso. The iterative solvers in wide use are PETSc and Trilinos. Also, autotuning efforts such as OSKI aim to provide basic building blocks for use with more complete iterative method frameworks.

**Available Software**
You will find a comprehensive list of open source software that is freely available for solving problems in numerical linear algebra, specifically dense, sparse direct and iterative systems and sparse iterative eigenvalue problems at: [http://www.netlib.org/utk/people/JackDongarra/la-sw.html](http://www.netlib.org/utk/people/JackDongarra/la-sw.html)

**Related Entries**
- LAPACK
- LINPACK Benchmark
- Linear Algebra, Numerical
- ScaLAPACK

**Bibliography**
Linear Algebra, Numerical

BERNARD PHILIPPE¹, AHMED SAMEH²
¹Campus de Beaulieu, Rennes, France
²Purdue University, West Lafayette, IN, USA

Synonyms
Matrix computations

Definition
Numerical linear algebra (or matrix computations) is the design and analysis of matrix operations and algorithms. Matrix computations plays a vital role in enabling the majority of computational science and engineering applications. Examples of such applications include: computational fluid dynamics, structural mechanics, fluid-structure interaction, computational electromagnetics, image processing, web search (PageRank), and information retrieval, just to list a few. Matrix computations can be divided into two classes: (a) dense matrix computations, and (b) sparse matrix computations, with the former being rich in data locality and hence can readily achieve high performance on modern architectures. The basic standard problems in numerical linear algebra are: (i) solving linear systems of equations, (ii) solving linear least squares problems with or without constraints, (iii) solving standard and generalized eigenvalue problems, and (iv) obtaining singular values or singular triplets of a matrix.

Discussion
Sparse Matrix Computations
The overwhelming majority of science and engineering applications give rise to sparse matrix computations. A sparse matrix is one that is populated mainly with zeros, and hence the need for storing only the nonzero elements especially for large problems. In the context of sparse matrix computations, one usually refers to sparse matrices with irregular sparsity structure. Designing algorithms that realize high performance for structured sparse matrix computations is much easier than the case of irregular sparsity. Irregular sparsity leads to lack of data locality and expensive memory references across the memory hierarchies of modern parallel architectures. While the creation of a standard set of Sparse Basic Linear Algebra Subroutines (SBLAS) was not as successful as the classical dense BLAS the most important kernels in sparse matrix computations are: (i) sparse matrix-vector multiplication and (ii) sparse matrix-tall dense matrix multiplication. Next to generating and applying preconditioners, these two kernels have a substantial impact on the realizable performance of Krylov subspace iterative schemes on these architectures.

One should also state that in Finite Element Analysis it has been the tradition to resort to matrix-free computations where the sparse matrices involved are not stored explicitly but algorithms are provided to compute their action, or the action of functions of such matrices, on a vector or a block of vectors. Such matrix-free approach makes it rather difficult to use effective preconditioners for solving large sparse linear systems with the needed iterative methods. Thus, matrix-free formulations may not be the optimal choice when using the emerging petascale parallel architectures with their abundant memories.

Dense Matrix Computations
Unlike sparse matrix computations, dense matrix algorithms benefit from the regular storage of the dense matrices involved to achieve high data locality and hence high performance through the fine-tuned three-level dense BLAS on parallel architectures. The third level of this set BLAS-3 includes matrix–matrix multiplications, which are the procedures that reach the highest rate of computation on such architectures. Even when dealing with sparse matrix computations, the goal is often to transform algorithms that are originally vector-oriented into their block versions so as to involve dense matrix multiplications. Moreover, the very popular Krylov methods for solving sparse linear systems or eigenvalue problems consist of projecting the
initial sparse problem onto some subspace of reduced dimension, leading to a similar dense matrix problem.

**Bibliographic Notes and Further Reading**

For examples of parallel dense matrix computations, see [1, 3, 4], and for examples of parallel sparse matrix computations, see [2].

**Bibliography**


---

**Linear Least Squares and Orthogonal Factorization**

**Definition**

A linear least squares problem is given by

\[
\minimize{x \in \mathbb{R}^n} \| b - Ax \|	ag{1}
\]

obtain \( x \in \mathbb{R}^n \), such that the two-norm \( \| b - Ax \| \) is minimized,

where \( A \in \mathbb{R}^{m \times n} \) and \( b \in \mathbb{R}^m \). The vector \( x \) is a solution if and only if the residual \( r = b - Ax \) is orthogonal to the range of \( A \): \( A^T r = 0 \). When \( A \) is of full column rank \( n \), there exists a unique solution. When \( \text{rank}(A) < n \), the solution of choice is that one with the smallest two-norm.

**Discussion**

Note: The number of the floating-point operations (additions, multiplications, divisions, square roots) of the algorithms are simply called operations in what follows.

**Full Rank Standard Problem**

Consider first the case in which \( n \leq m \) and \( A \) is of full column rank \( n \).

**General scheme.** Since the two-norm is invariant under orthogonal transformations, the most common approach for solving this least squares problem consists of computing the orthogonal factorization of \( A \). The procedure amounts to premultiplying \( A \) and \( b \) by a finite number – say \( p \) – of orthogonal transformations \( Q_1, \ldots, Q_p \), such that

\[
Q_p \cdots Q_2 Q_1 A = \begin{pmatrix} R \\ 0 \end{pmatrix},
\]

where \( R \in \mathbb{R}^{n \times n} \) is a nonsingular upper triangular or upper trapezoidal matrix. Therefore,

\[
\| b - Ax \| = \| Q_p \cdots Q_1 (b - Ax) \| = \| \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} - \begin{pmatrix} R \\ 0 \end{pmatrix} x \|
\]

where \( c_1 \in \mathbb{R}^n \), and the unique solution of (1) is given by solving \( Rx = c_1 \). In what follows, the orthogonal reduction to the triangular form will only be considered. For solving the resulting triangular system, please refer to [BLAS 12.c.ii.1.a, linear solvers 12.c.ii.2.a].
Householder Reductions

The sequential algorithm. Given a vector \( \mathbf{a} \in \mathbb{R}^s \), a vector \( \mathbf{v} \in \mathbb{R}^s \) can be determined such that the transformation \( \mathbf{H} = H(\mathbf{v}) = \mathbf{I} - \beta \mathbf{v} \mathbf{v}^T \), where \( \beta = (2 / \mathbf{v}^T \mathbf{v}) \), yields \( \mathbf{H} \mathbf{a} = \pm \mathbf{a}_1 \) \( \mathbf{e}_1 \) where \( \mathbf{e}_1 \) is the first canonical vector of \( \mathbb{R}^s \) (there are two such transformations from which one is selected for numerical stability [16]). \( \mathbf{H} \) is an elementary reflector, i.e., \( \mathbf{H} \) is orthogonal and \( \mathbf{H}^2 = \mathbf{I} \). Application of \( \mathbf{H} \) to a given vector \( \mathbf{x} \in \mathbb{R}^s \) is given by \( \mathbf{H}(\mathbf{v})\mathbf{x} = \mathbf{x} - \beta (\mathbf{v}^T \mathbf{x}) \mathbf{v} \) and computed using two Level 1 BLAS routines [19], namely, DDOT and DAXPY, involving only \( 4s \) operations and not \( 2s^2 \) as the case in matrix-vector multiplication.

The triangularization of \( \mathbf{A} \) is obtained by successively applying such orthogonal transformations to \( \mathbf{A} : \mathbf{A}_0 = \mathbf{A} \) and for \( k = 1 : \min(n, m-1) \), \( \mathbf{A}_{k+1} = \mathbf{H}_k \mathbf{A}_k \) where

\[
\mathbf{H}_k = \begin{pmatrix}
I_{k-1} & 0 \\
0 & 0
\end{pmatrix}
\]

with \( v_k \) chosen such that all but the first entry of \( H(v_k)A_k(k : m, k) \) are zero. At step \( k \), the multiplication \( H_k \mathbf{A}_k \) is obtained from \( H(v_k)A_k(k : m, k : n) \) which involves \( 4(n - k)(m - k) + O(m - k) \) operations. It can be implemented by two Level 2 BLAS routines [9], namely, the matrix-vector multiplication DGEMV and the rank-one update DGER. The procedure stores the matrix \( A_{k+1} \) in place of \( A_k \).

The total procedure involves \( 2n^2(m - n)/3 + O(mn) \) operations for obtaining \( \mathbf{R} \). To apply simultaneously the transformations to the right-hand-side \( \mathbf{b} \) of problem (1), one may append that vector as the \( n + 1 \)st column of \( \mathbf{A} \). When necessary for subsequent calculations, the sequence of the vectors \( (v_k)_{k=1}^n \) can be stored for instance in the lower part of the transformed matrix \( \mathbf{A}_k \). This is the case when \( n < m \) and an orthogonal basis \( \tilde{Q} = [q_1, \ldots, q_n] \in \mathbb{R}^{m \times n} \) of the range of \( \mathbf{A} \) is sought: the basis is obtained by premultiplying the matrix

\[
\begin{pmatrix}
I_n \\
0
\end{pmatrix}
\]

successively by \( H_1, H_{n-1}, \ldots, H_n \). This process involves \( 4(m^2n - mn^2 + n^3/3) + O(mn) \) operations.

Block versions of the procedure have been introduced by Bischof and Van Loan [4] (the \( \mathbf{WY} \) form) and by Schreiber and Parlett [25] (the \( \mathbf{GG}^T \) form). The generation and application of the \( \mathbf{WY} \) form are implemented in the routines DLARFT and DLARFB of LAPACK [1]. The latter involves Level 3 BLAS primitives [9]. The \( \mathbf{WY} \) form consists of considering a narrow window in which the single-vector algorithm is used before applying the corresponding transformations to the remaining part of \( \mathbf{A} \): if \( s \) is the window width, \( s \) steps are accumulated in the form \( \mathbf{H}_{k+1} \cdots \mathbf{H}_k = \mathbf{I} + \mathbf{WY}^T \) where \( \mathbf{W}, \mathbf{Y} \in \mathbb{R}^{ms \times s} \). This expression allows the use of Level 3 BLAS in updating the remaining part of \( \mathbf{A} \). In LAPACK, the driver routine which implements the resolution of the least square problem (1) is DORGQR.

The parallel algorithm. Theoretically, by using \( O(mn) \) processors, the computation may be performed in \( O(n \log m) \) time steps. This is a direct extension to rectangular matrices of Sameh’s result [22]: in each of the \( n \) update phases, each scalar product can be performed in \( O(\log m) \) time steps on \( m \) processors and there are at most \( n \) columns to be processed in parallel.

Since, except for very special computing architectures (e.g., systolic arrays), independent tasks cannot be reduced to one operation; a plausible approach is to wrap the columns of \( \mathbf{A} \) across a ring of processors as shown in [14]. The block version of the algorithm above is the preferred parallel scheme which is implemented in ScaLAPACK [7] via the routines PDGEQRF and PDGELS where \( \mathbf{A} \) and \( \mathbf{b} \) are distributed on a two-dimensional grid of processes according to the block cyclic scheme [7] (see [4. Parallel programming / i. Data distribution]). The block size is chosen large enough to allow Level 3 BLAS routines to achieve maximum performances on each involved uniprocessor (at least 90% as recommended by the authors of ScaLAPACK [8]). While increasing the block size improves the granularity of the computation, it may negatively affect concurrency. An optimal tradeoff therefore must be found depending on the architecture of the computing platform.

Givens Rotations

The sequential algorithm. In the standard Givens’ process, the orthogonal matrices in (2) consist of \( \mathbf{p} = mn - n(n + 1)/2 \) plane rotations extended to the entire space \( \mathbb{R}^m \). The triangularization of \( \mathbf{A} \) is obtained by successively applying plane rotations \( \mathbf{R}_k \) so that \( A_{k+1} = R_k A_k \) for \( k = 1 : mn - n(n + 1)/2 \) with \( A_0 = A \). Each of these rotations will be chosen to introduce a single zero below the diagonal of \( A \) at step \( k \), eliminating the entry \( a_{i\ell}^{(k)} \) of \( A_k \) where \( 1 \leq \ell \leq n \) and \( 1 < i \leq m \) can be done by the rotation
where \( 1 \leq j \leq n \), and \( c^2 + s^2 = 1 \) are such that

\[
\begin{pmatrix}
c & s \\
-s & c
\end{pmatrix}
\begin{pmatrix}
a_{(k)}^{(j)} \\
a_{(k+1)}^{(j)}
\end{pmatrix} =
\begin{pmatrix}
a_{(k)}^{(j)} \\
0
\end{pmatrix}.
\]

Two such rotations exist from which one is selected for numerical stability [16]. The construction of a rotation and its application to a couple of rows are implemented in the Level 1 BLAS [19] via the procedures DROTG and DROT, respectively.

The order of annihilation is quite flexible as long as applying a new rotation does not destroy a previously introduced zero. The simplest way is to organize columnwise eliminations. The total number of operations is \( 3mn^2 - n^3 + O(mn) \) which is roughly 1.5 times that required by the Householder’s reduction.

**Parallel algorithms.** One parallel algorithm consists of organizing the \( mn - n/(n+1)/2 \) rotations into sweeps which are sets of at most \( \lfloor m/2 \rfloor \) parallel rotations. For square matrices, Sameh and Kuck [24] and later, Cosnard and Robert [12], introduced such \( O(n) \) procedures using \( O(n^2) \) processors. In the general situation of rectangular matrices, Cosnard et al. in [11] proved the theoretical optimality of the greedy algorithm introduced by Modi and Clarke in [21] over any other algorithm. More details on the chronological appearance of the Givens algorithms and their comparison with the Householder reductions are given in [14].

**Hybrid Algorithms (\( m \gg n \))**

**Dense matrix.** This is a situation that arises in some computations like in the adjustment of geodetic networks [15], or in the parallel construction of a Krylov basis where often \( m \) is larger than a million and \( n \) is smaller than 100 [26].

The algorithm was introduced by Sameh [23]. It is depicted in Fig. 1 in the case where two processors are used. It consists of partitioning \( A \) into contiguous blocks of \( r \) rows where \( r \geq n \), with each block assigned to one processor. In the first step (Fig. 1a), each block is triangularized by Householder reduction. This step consumes the major part of the overall computation and is void of any communications. The second step is devoted to annihilating all the resulting triangular factors except the first one. There are several possibilities for achieving this goal. In [15], for annihilating the entries of the triangular matrix \( T_2 \), the rotations are chosen as depicted in Fig. 1b. The entry \((1,1)\) of the first row of \( T_2 \) is annihilated by rotating the first rows of \( T_1 \) and \( T_2 \). Then all the other entries of the first diagonal of \( T_2 \) can easily be annihilated by rotating rows of \( T_2 \). As soon as the entry \((2,2)\) of \( T_2 \) is annihilated, the entry \((1,2)\) of \( T_2 \) can be annihilated before the annihilation of the first diagonal of the matrix is completed. Rotations indicated with stars are the only ones which require synchronization.

**Parallel annihilation**
By following that strategy, Sidje [26] developed the software RODDEC which performs the QR factorization on a ring of processors.

**Block angular matrix.** In some applications, a domain decomposition provides a matrix such that under some appropriate numbering of the domains and corresponding interfaces, the matrix \( A \) is of the form:

\[
A = \begin{pmatrix}
B_1 & C_1 \\
B_2 & C_2 \\
\vdots & \vdots \\
B_r & C_r
\end{pmatrix}
\]

where \( A \) as well as each block \( B_i \), for \( i = 1, \cdots, r \) is of full column rank. The special structure implies that the QR factorization of \( A \) yields an upper triangular matrix with the same block structure; this can be seen as shown by Golub et al. in [15] from the Cholesky factorization of \( A^T A \) which is a block arrowhead matrix. Therefore, similar to the first step of the algorithm previously described, for \( i = 1, \cdots, r \), processor \( i \) first performs the QR factorization of the block \( B_i \). The corresponding blocks of \( [B_i, C_i] \) are now transformed into

\[
\begin{pmatrix}
R_i \\
0
\end{pmatrix}, \begin{pmatrix}
D_{i1} \\
D_{i2}
\end{pmatrix}
\]

By virtually reordering the block rows, the resulting matrix can be expressed by:

\[
A_1 = \begin{pmatrix}
R_1 & D_{11} \\
R_2 & D_{21} \\
\vdots & \vdots \\
R_r & D_{r1} \\
D_{12} \\
D_{22} \\
\vdots \\
D_{r2}
\end{pmatrix}
\]

To complete the QR factorization of \( A \), it is therefore necessary to obtain the QR factorization of the sub-matrix defined by the sub-blocks \( D_{i2} \) for \( i = 1, \cdots, r \). This is exactly the situation where the procedure RODDEC can be used. Other approaches can also be considered as mentioned in [6]. For instance Golub et al. in [15] introduce a version of the algorithm adapted to the hierarchical definition of the subdomains. The algorithm takes advantage of the locality of the frontiers to minimize communications.

**Gram-Schmidt procedures.** The goal of these procedures is to obtain the orthogonal factorization \( A = QR \) of a full rank matrix \( A = [a_1, \cdots, a_n] \in \mathbb{R}^{m \times n} \) where \( Q = [q_1, \cdots, q_n] \in \mathbb{R}^{m \times n} \) has orthonormal columns, and \( R \in \mathbb{R}^{n \times n} \) is a nonsingular upper-triangular matrix with positive diagonal elements thus insuring the uniqueness of the factorization. This factorization is often referred to as the *skinny* QR factorization. Since orthogonality of the vectors \( q_i \) deteriorates rather quickly, reorthogonalization is often necessary. The general structure of the scheme is given by

**Algorithm :** Gram-Schmidt

\[
do k = 0 : n-1, \\
w = P_k(a_{k+1}) \\
q_{k+1} = w/\|w\| \\
end
\]

where \( P_k \) is the orthogonal projector onto the orthogonal complement of the subspace spanned by \( \{a_1, \cdots, a_k\} \) (\( P_0 \) is the identity). \( R = Q^T A \) can be built step by step during the process.

Two different versions of the algorithm are obtained by expressing \( P_k \) in distinct ways:

- **Classical_GS:** for the Classical Gram-Schmidt \( P_k = I - Q_k Q_k^T \), where \( Q_k = [q_1, \cdots, q_k] \). Unfortunately, this version has been proved by Bjorck [5] to be numerically unreliable except when it is applied two times which makes it numerically equivalent to **Modified_GS**.

- **Modified_GS:** for the Modified Gram-Schmidt \( P_k = (I - q_k q_k^T) \cdots (I - q_1 q_1^T) \). Numerically, the columns of \( Q \) are orthogonal up to a tolerance determined by the machine precision multiplied by the condition number of \( A \) [5]. By applying the algorithm a second time (i.e., with complete reorthogonalization), the columns of \( Q \) become orthogonal up to the tolerance determined by the machine precision parameter similar to the Householder or Givens reductions.
The basic procedures involve \( mn^2 + O(mn) \) operations. Classical GS is based on Level 2 BLAS procedures but it must be applied two times while Modified GS is based on Level 1 BLAS routines. By inverting the two loops of Modified GS, the procedure can proceed with a Level 2 BLAS routine. This is only possible when \( A \) is available explicitly (for instance, this is not possible with the Arnoldi procedure [Krylov methods 12.c.ii.3.f]).

In order to proceed with Level 3 BLAS routines, a block version Block GS is obtained by considering blocks of vectors instead of single vectors in Modified GS and by replacing the normalizing step by an application of Modified GS on the individual blocks. Jalby and Philippe [17] proved that it is necessary to apply the normalizing step twice to reach the numerical accuracy of Modified GS. However, with \( q \) blocks of \( r \) vectors (\( n = qr \)), the Block2 GS procedure only involves \((1/q)(mn^2) + O(mn)\) additional operations which makes Block2 GS faster than Modified GS as soon as \( q \) is not too small.

Parallel implementations of the different forms of Gram-Schmidt orthogonalization are similar to those of Gauss elimination schemes in which there are two nested loops of vector operations. To orthogonalize the columns of \( A \) on a cluster, the procedure Modified GS can be expressed by:

**Algorithm**: Parallel Modified GS

```
    do j = 1 : n,
        a_k = a_j / |a_j|;
    doall k = j+1 : n,
        a_k = a_k - (a_j a_k) a_j;
    end
```

By inserting a synchronizing mechanism to ensure that normalizing a vector happens only after it is completely updated, it is even possible to transform the sequential outer loop into a doallcross loop.

On distributed memory platforms, this algorithm may also be used, but it might be more efficient to use a block partitioning similar to that of ScalAPACK for either Modified GS or Block2 GS.

**Other Least Squares Problems**

**Rank-deficient problems.** When \( A \) is rank \( q < n \), there is an infinite set of solutions for (1): for any solution \( x \), the set of all solutions is expressed by \( S = x + N(A) \) where \( N(A) \) is the \((n - q)\)-dimensional null space of \( A \). The Singular Value Decomposition (SVD) of \( A \) is given by

\[
A = U \begin{pmatrix} \Sigma & 0_{q,n-q} \\ 0_{m-q,q} & 0_{m-q,n-q} \end{pmatrix} V^T,
\]

where the diagonal entries of the matrix \( \Sigma = \text{diag}(\sigma_1, \cdots, \sigma_q) \) are positive, \( U \in \mathbb{R}^{m \times m} \), \( V \in \mathbb{R}^{n \times n} \) are orthogonal matrices. Let \( c = U^T b = \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} \) where \( c_1 \in \mathbb{R}^q \). Therefore, \( x \in S \) if and only if there exists \( y \in \mathbb{R}^{n-q} \) such that \( x = V \begin{pmatrix} \Sigma^{-1} c_1 \\ y \end{pmatrix} \) and the corresponding minimum residual norm is equal to \(|c_2|\) which is null when \( m = q \). In \( S \), the classically selected solution is the one with the smallest norm, i.e., when \( y = 0 \).

**Householder QR with column pivoting.** In order to avoid computing the singular-value decomposition which is quite expensive, the factorization (6) is often replaced by a similar factorization where \( \Sigma \) is not assumed to be diagonal. Such factorizations are called complete orthogonal factorizations. One way to compute such factorization relies on the procedure introduced by Businger and Golub [10]. It computes the QR factorization by Householder reductions with column pivoting. Factorization (2) is now replaced by

\[
Q_q \cdots Q_2 Q_1 A P_1 P_2 \cdots P_q = \begin{pmatrix} T \\ 0 \end{pmatrix},
\]

where matrices \( P_1, \cdots, P_q \) are permutations and \( T \in \mathbb{R}^{q \times q} \) is upper-trapezoidal. The permutations are determined so as to insure that the absolute values of the diagonal entries of \( T \) are nonincreasing. To obtain a complete orthogonal factorization of \( A \) the process performs first the factorization (7) and then computes an LQ factorization of \( T \) (equivalent to a QR factorization of \( T^T \)). The process ends up with a factorization similar to (6) with \( \Sigma \) being a nonsingular lower-triangular matrix.

In LAPACK, Householder QR with column pivoting is implemented in the routine DGEQPF which only involves Level 1 and Level 2 BLAS routines. The new
Parallel implementation. In ScaLAPACK, Householder QR with column pivoting and complete orthogonal factorization are both implemented via the routines DGGEQPF and PDTZRFZ, respectively.

Bischof introduced a method that avoids global pivoting to achieve a better parallel efficiency [2]. It consists of a local pivoting strategy limited to each block allocated to a single processor. This strategy is numerically acceptable as long as the condition number of a block remains below some threshold. In order to control it, the author uses an Incremental Condition Estimator (ICE) studied in [3]. Achieving adequate local pivoting strategies is a topic that is yet to be completely settled.

Generalized least squares and total least squares problems. In many applications, the right-hand-side $b$ of problem (1) is known only with some uncertainty: its error is known to follow a statistical law given by its covariance matrix $\Omega$. In general, $\Omega$ is positive definite and its Cholesky factorization $\Omega = LL^T$ does exist. The Generalized Least Squares problem consists of computing $x \in \mathbb{R}^n$ such that $\| b - Ax \|_{\Omega^{-1}}$ is minimized. It can therefore be recast into the regular least squares problem: compute $x \in \mathbb{R}^n$ such that $\| L^{-1} (b - Ax) \|$ is minimized. This approach is acceptable when (1) $\Omega$ is not ill-conditioned and (2) the factor $L$ can be computed.

When the matrix $A$ suffers from uncertainty as well, the problem becomes one of Total Least Squares. The basic problem is then given by

$$
\text{minimize } \| [A; b] - [\hat{A}; \hat{b}] \|_F, \quad \text{subject to } \hat{b} \in R(\hat{A}),
$$

where $\| \cdot \|_F$ denotes the Frobenius norm and where $R(\hat{A})$ is the range of $\hat{A}$. Any $x \in \mathbb{R}^n$ for which $\hat{A}x = \hat{b}$ is a Total Least Squares solution (TLS). By denoting, respectively, $\sigma_n$ and $\tilde{\sigma}_{n+1}$ the smallest singular values of $A$ and $[A; b]$, assuming that $\sigma_n > \tilde{\sigma}_{n+1}$, the only TLS solution is $\hat{x} = (A^TA - \tilde{\sigma}_{n+1}^2I)^{-1}A^Tb$. Total Least Squares theory, problems, as well as solution schemes are given by Van Huffel and Vandewalle in [27].

Parallel algorithms for the Generalized or Total Least Squares problems are based on Linear Least Squares, Cholesky factorization, and Singular Value Decomposition. Kontoghiorghes considers the Seemingly Unrelated Regression Equations Models (SURE) models as Generalized Least Squares problems [18]. In subsequent work with his co-authors, he considers updating (adding equations to the model), down-dating (deleting equations) as well as parallel resolution of the models. Gang Lou and Shih-Ping Han introduced in [20] a general class of parallel methods for solving other Least Squares problems with constraints.

Related Entries
- BLAS (Basic Linear Algebra Subprograms)
- Dense Linear System Solvers
- LAPACK
- Linear Algebra, Numerical
- PLAPACK
- ScaLAPACK

Bibliographic Notes and Further Reading

Bibliography

Linear Regression

- Linear Least Squares and Orthogonal Factorization

LINPACK Benchmark

Jack Dongarra, Piotr Luszczek
University of Tennessee, Knoxville, TN, USA

Definition

The LINPACK benchmark is a computer benchmark that reports the performance for solving a system of linear equations with a general dense matrix of size 100, 1,000, and also of arbitrary size. The matrices, the calculations, and the solution must use 64-bit floating point arithmetic, and partial pivoting needs to be used for numerical stability. The allowed parallelism modes include automatic parallelization done by the compiler as well as manual parallelization that uses hardware-assisted shared memory or explicit message passing on a distributed memory machine.

Discussion

The original LINPACK Benchmark is, in some sense, an accident. It was originally designed to assist users of the LINPACK package [1] by providing information on execution times required to solve a system of linear equations. The first “LINPACK Benchmark” report appeared as an appendix in the LINPACK Users’ Guide in 1979. The appendix comprised data for one commonly used path in the LINPACK software package. Results were provided for a matrix problem of size 100, on a collection of widely used computers (23 computers in all). This was done so users could estimate the time required to solve their matrix problem by extrapolation.
Over the years, additional performance data was added, more as a hobby than anything else, and today the collection includes over 1,300 different computer systems. In addition to the increasing number of computers, the scope of the benchmark has also expanded. The benchmark report describes the performance for solving a general dense matrix problem $Ax = b$ at three levels of problem size and optimization opportunity: 100 by 100 problem (inner loop optimization), 1,000 by 1,000 problem (three loop optimization – the whole program), and a scalable parallel problem.

The LINPACK benchmark features two routines: DGEFA and DGESL (these are the double precision versions, usually 64-bit floating point arithmetic, SGEFA and SGESL are the single-precision counter parts, usually 32 bit floating point arithmetic); DGEFA performs the decomposition with partial pivoting and DGESL uses that decomposition to solve the given system of linear equations. Most of the execution time – $O(n^2)$ floating-point operations – is spent in DGEFA. Once the matrix has been decomposed, DGESL is used to find the solution; this requires $O(n^2)$ floating-point operations.

DGEFA and DGESL in turn call three BLAS routines: DAXPY, IDAMAX, and DSCAL. By far the major portion of time – over 90% at order 100 – is spent in DAXPY. DAXPY is used to multiply a scalar, $ff$, times a vector, $x$, and add the results to another vector, $y$. It is called approximately $n^2/2$ times by DGEFA and $2n$ times by DGESL with vectors of varying length. The statement $y_i ← y_i + ff x_i$, which forms an element of the DAXPY operation, is executed approximately $n^3/3 + n^2$ times, which gives rise to roughly $2/3n^3$ floating-point operations in the solution. Thus, the $n = 100$ benchmark requires roughly $2/3$ million floating-point operations.

The statement $y_i ← y_i + ff x_i$, besides the floating-point addition and floating-point multiplication, involves a few one-dimensional index operations and storage references. While the LINPACK routines DGEFA and DGESL involve two-dimensional array references, the BLAS refer to one-dimensional arrays. The LINPACK routines in general have been organized to access two-dimensional arrays by column. In DGEFA, the call to DAXPY passes an address into the two-dimensional array $A$, which is then treated as a one-dimensional reference within DAXPY. Since the indexing is down a column of the two-dimensional array, the references to the one-dimensional array are sequential with unit stride. This is a performance enhancement over, say, addressing across the column of a two-dimensional array. Since Fortran dictates that two-dimensional arrays be stored by column in memory, accesses to consecutive elements of a column lead to simple index calculations. References to consecutive elements differ by one word instead of by the leading dimension of the two-dimensional array.

**Detailed Operation Counts**

The results reflect only one problem area: solving dense systems of equations using the LINPACK programs in a Fortran environment. Since most of the time is spent in DAXPY, the benchmark is really measuring the performance of DAXPY. The average vector length for the algorithm used to compute LU decomposition with partial pivoting is $2/3n$. Thus in the benchmark with $n = 100$, the average vector length is 66.

In order to solve this matrix problem, it is necessary to perform almost 700,000 floating point operations. The time required to solve the problem is divided into this number to determine the megaflops rate.

The routines DGEFA calls IDAMAX, DSCAL and DAXPY. Routine IDAMAX, which computes the index of a vector with largest modulus, is called 99 times, with vector lengths running from 2 to 100. Each call to IDAMAX gives rise to $n$ double precision absolute value computation and $n – 1$ double precision comparisons. The total number of operations is 5,364 double precision absolute values and 4,950 double precision comparisons.

DSCAL is called 99 times, with vector lengths running from 1 to 99. Each call to DSCAL performs $n$ double precision multiplies, for a total of 4,950 multiplies.

DAXPY does the bulk of the work. It is called $n$ times, where $n$ varies from 1 to 99. Each call to DAXPY gives rise to one double precision comparison with zero, $n$ double precision additions, and $n$ double precision multiplications. This leads to 4,950 comparisons against 0, 328,350 additions and 328,350 multiplications. In addition, DGEFA itself does 99 double precision comparisons against 0, and 99 double precision reciprocals. The total operation count for DGEFA is given in Table 1.
**LINPACK Benchmark. Table 1** Double precision operations counts for LINPACK 100’s DGEFA routine

<table>
<thead>
<tr>
<th>Operation type</th>
<th>Operation count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>328,350</td>
</tr>
<tr>
<td>Multiply</td>
<td>333,300</td>
</tr>
<tr>
<td>Reciprocal</td>
<td>99</td>
</tr>
<tr>
<td>Absolute value</td>
<td>5,364</td>
</tr>
<tr>
<td>(\leq)</td>
<td>4,950</td>
</tr>
<tr>
<td>(\neq 0)</td>
<td>5,247</td>
</tr>
</tbody>
</table>

**LINPACK Benchmark. Table 2** Double precision operations counts for LINPACK 100’s DGESL routine

<table>
<thead>
<tr>
<th>Operation type</th>
<th>Operation count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>9,900</td>
</tr>
<tr>
<td>Multiply</td>
<td>9,900</td>
</tr>
<tr>
<td>Divide</td>
<td>100</td>
</tr>
<tr>
<td>Negate</td>
<td>100</td>
</tr>
<tr>
<td>(\leq)</td>
<td>100</td>
</tr>
<tr>
<td>(\neq 0)</td>
<td>199</td>
</tr>
</tbody>
</table>

Routine DGESL, which is used to solve a system of equations based on the factorization from DGEFA, does much more modest amount of floating point operations. In DGESL, DAXPY is called in two places, once with vector lengths running from 1 to 99 and once with vector lengths running from 0 to 99. This leads to a total of 9,900 double precision additions, the same number of double precision multiplications, and 199 compares against 0. DGESL does 100 divisions and 100 negations as well. The total operation count for DGESL is given in Table 2.

This leads to a total operation count for the LINPACK benchmark given in Table 3 or a grand total of 697,509 floating point operations. (The LINPACK uses approximately \(2/3n^3 + 2n^2\) operations, which for \(n = 100\) has the value of 686,667.)

It is instructive to look just at the contribution due to DAXPY. Of these floating point operations, the calls to DAXPY from DGEFA and DGESL account for a total of 338,160 additions, 338,160 multiplications, and 5,147 comparisons with zero. This gives a total of 681,467 operations, or over 97% of all the floating point operations that are executed. The total time is taken up with more than arithmetic operations. In particular, there is quite a lot of time spent loading and storing the operands of the floating point operations. We can estimate the number of loads and stores by assuming that all operands must be loaded into registers, but also assuming that the compiler will do a reasonable job of promoting loop-invariant quantities out of loops, so that they need not be loaded each time within the loop. Then DAXPY accounts for 681,468 double precision loads and 338,160 double precision stores. IDAMAX accounts for 4,950 loads, DSCAL for 5,049 loads and 4,950 stores, DGEFA outside of the loads and stores in the BLAS does 9,990 loads and 9,694 stores, and DGESL for 492 loads and 193 stores. Thus, the total number of loads is 701,949 and stores is 352,997. Here again DAXPY dominates the statistics. The other overhead that must be accounted for is the load indexing, address arithmetic, and the overhead of argument passing and calls.

**Precision**

In discussions of scientific computing, one normally assumes that floating-point computations will be carried out to full precision or 64-bit floating point arithmetic. Note that this is not an issue of single or double precision as some systems have 64-bit floating point arithmetic as single precision. It is a function of the arithmetic used.

**Related Entries**

- **Benchmarks**
- **HPC Challenge Benchmark**
- **LINPACK Benchmark**
- **Livermore Loops**
- **TOP500**
Definition  
*Lisp (pronounced "star-lisp") is a data-parallel dialect of Lisp developed around 1984 for Connection Machine supercomputers manufactured by Thinking Machines Corporation. It consists of Common Lisp (or, in its earliest implementations, Symbolics Zetalisp) augmented with a package of relatively low-level data-parallel operations. The language design draws a sharp distinction between data residing in the front-end Common Lisp processor and data residing in the massively parallel Connection Machine coprocessor. While pointers to any Lisp data could reside in the Connection Machine processors, parallel processing was most effective on arrays of numeric and character data.

Discussion  
Of the four programming languages (*Lisp, C*, CM Fortran, and CM-Lisp) provided by Thinking Machines Corporation for Connection Machine Systems, *Lisp was the first to be implemented and the one most closely aligned to details of the Connection Machine architecture; essentially anything that could be done with the Connection Machine hardware could be expressed in *Lisp, but the programmer was freed from much drudgery because expression evaluation made automatic use of a stack for parallel data structures and provided automatic conversions among data types and sizes as necessary.

To quote from the *Connection Machine Model CM-2 Technical Summary*, 1987:

> The parallel primitives of *Lisp support a model of the Connection Machine in which each processor executes a subset of Common Lisp, with a single thread of control residing on the front-end computer. For most Common Lisp functions, *Lisp provides a corresponding parallel function that can operate on all processors, or some selected subsets, simultaneously. In addition, the language provides Lisp-level operators for communicating between processors, both through pointers and in regular patterns. Sequential Common Lisp code, running on the front end, can be freely intermixed with the parallel code executed on the Connection Machine. Most *Lisp functionality corresponds directly to underlying Paris [Connection Machine PARallel Instruction Set] instructions... As a result... direct calls to Paris instructions and special purpose microcode blend in naturally with *Lisp code [1, p. 46], [2, p. 223].

By 1989 the phrase “most Common Lisp functions” had been amended to “many Common Lisp functions” [3, p. 27].

A *Lisp pvar (parallel variable) was, in effect, an array indexed by processor number, so that each (virtual) Connection Machine processor contained one array element. A unary parallel operation would perform the same scalar operation on each element simultaneously; a binary parallel operation would operate on corresponding elements of two pvars. While, in general, a pvar could hold pointers to any Common Lisp objects, not all operations on Common Lisp objects were supported in parallel form. In particular, classic list operations such as CAR and CDR and ASSOC had no parallel versions (“*Lisp does not implement list pvars” [4, p. 155]). On the other hand, most atomic Common Lisp types were eventually supported, including integers and floating-point numbers of various fixed sizes, complex numbers, and characters; and many of the Common Lisp sequence functions were provided in parallel form for operating on many arrays simultaneously. “The eight basic pvar types are boolean, integer, floating-point, complex, character, structure, and front-end value” [5, p. 80]. The operation of indexing into a pvar to fetch or store a specified element was supported in both scalar and parallel forms; in this way either the front-end computer or the entire set of Connection Machine processors could freely fetch or store any element of a pvar.

*Lisp was useful because it provided all the program-building tools of Common Lisp [6, 7],
including user-defined macros, and because it provided convenient access to all the facilities of the Connection Machine hardware, including (on the model CM-2) the multidimensional NEWS network, “backwards routing” in the hypercube network, the thousands of red LEDs that were a distinctive feature of the Connection Machine cabinetry, and operations on integers and floating-point numbers that could be of any specified length, such as 23-bit integers or floating-point numbers with 9-bit exponents and 43-bit significands (a flexibility of the model CM-1 that few users exploited in practice once the hardware floating-point accelerators came into use on the model CM-2).

By 1990, *Lisp had grown to include over 400 functions and macros, listed in a *Lisp Dictionary of over 1,100 pages [8]. Compared with other dialects and extensions of Lisp, *Lisp had a distinct visual style because of its consistent use of either a leading “*” or a trailing “!!” in names:

- The names of functions and macros that return pvars as their values end in “!!”. This suffix, pronounced bang-bang, is meant to look like two parallel lines.
- All *Lisp functions that perform parallel computation and do not end in “!!” begin with “*” (pronounced star); hence, the name *Lisp [9, p. 5].

A later document further explains:

- The characters “!!” are meant to resemble the mathematical symbol ||, which means parallel [5, p. 82].

(It should be noted that the lexical syntax of Common Lisp [6, 7] would have made it awkward to use two vertical-bar symbols “| |” rather than two exclamation points.)

Figure 1 shows an early (1987) example of a *Lisp program that identifies prime numbers less than 100,000 by the method of the Sieve of Eratosthenes, taken (with minor corrections) from [1]. A set of virtual processors of size 100,000 is assumed to have been established before the function find-primes is called. The special form *all enables all virtual processors for activity. The special form *let is analogous to Common Lisp let in that it binds local variables, but *let binds local parallel variables, allocating them on a stack within the Connection Machine (actually, one stack per virtual processor). In this case, the local parallel variable prime is initially nil (false) within each processor, and the local parallel variable candidate is initially t (true) within each processor. The conditional form *when is analogous to Common Lisp when, but operates by (a) evaluating a parallel test form; (b) temporarily disabling for execution any processors that computed nil for the test form in step (a); (c) executing the form(s) in the body of the *when construct; and finally (d) re-enabling any processors disabled in step (b). The function self-address!! returns the virtual processor address, which is a different integer in each virtual processor (starting from zero); the function !! broadcasts a scalar value from the front-end processor to all Connection Machine processors, so that (!! 2) is a parallel quantity with value 2 in every virtual processor; and function <!! compares two parallel quantities, producing a parallel Boolean (t or nil) result. The functions *or and *min are reduction operations, producing the logical OR or the arithmetic minimum of one value taken from every active virtual processor; thus (*or candidate) will be nil only if every virtual

(*defun find-primes ()
  (*all
    (*let ((prime (!! nil) (candidate (!! t))))
      (*when (<!! (self-address!!) (!! 2))
        (*set candidate (!! nil)))
    (do () ((*or candidate))
      (*when candidate
        (let ((next-prime (*min (self-address!!))))
          (setf (pref prime next-prime) t)
            (*when (zerop!! (mod!! (self-address!!) (!! next-prime)))
              (*set candidate (!! nil))))))
  prime)))

*Lisp. Fig. 1 Example *Lisp program for identifying prime numbers
processor has the value nil for candidate, and
('min (self-address!!)), because it occurs within ('when candidate ...), will return the
smallest integer for which candidate is true. The
function pref is analogous to Common Lisp aref;

(The parallel equivalent, not used in this example, is
pref!!, which allows every Connection Machine pro-
cessor to fetch a value by indexing into a parallel value;
this is one way of performing interprocessor commu-
ication in *Lisp.) The functions zerop!! and mod
are parallel arithmetic operations that are analogous to
Common Lisp zerop and mod.

Related Entries
► C*
► Connection Machine
► Connection Machine Fortran
► Connection Machine Lisp

Bibliography
1. Thinking Machines Corporation. Connection Machine model
   CM-2 technical summary, technical report HA87-4. Cambridge,
   MA, April 1987
2. Michael Hord R (1990) Parallel supercomputing in SIMD architec-
tures. CRC Press, Boca Raton
3. Thinking Machines Corporation. Connection Machine technical
   summary, version 5.1. Cambridge, MA, May 1989
4. Thinking Machines Corporation. Supplement to the *Lisp refer-
ce manual, version 5.0. Cambridge, MA, September 1988
5. Thinking Machines Corporation. Connection Machine CM-5
technical summary, 3rd edn. Cambridge, MA, November 1993
   Common Lisp: The language. Digital Press, Burlington
7. Steele GL Jr., Fahlman SE, Gabriel RP, Moon DA, Weinreb DL,
   Bobrow DG, DeMichiel LG, Keene SE, Kiczales G, Perdue C,
   language, 2nd edn. Digital Press, Bedford
   Cambridge, MA, February 1990
   5.0. Cambridge, MA, September 1988

Little’s Law

John L. Gustafson
Intel Labs, Intel Corporation, Santa Clara, CA, USA

Synonyms
Little’s lemma; Little’s principle; Little’s result; Little’s theorem

Definition
Little’s Law says that in the long-term, steady state of a
production system, the average number of items \( L \) in the
system is the product of the average arrival rate \( \lambda \) and
the average time \( W \) that an item spends in the system,
that is,

\[
L = \lambda W.
\]

Discussion
At first glance, Little’s Law looks like common sense. If
items arrive faster than the system can process them, the
system will overflow. Perhaps the earliest mention of the
relation is by A. Cobham in 1954, and he states it as a fact
without proof [2]. However, the law is insightful in two
ways. First, it does not depend on the probability dis-
tributions of any of the variables. Second, since arrival
rates are generally less than maximum processing rates,
its says what the capacity of the system must be to handle
the queue in a system design.

Figure 1 shows a visualization of a queue in steady
state that explains Little’s Law:

Proofs of Little’s Law use the Law of Large Numbers,
which is implicit in the phrase “steady state” of the sys-
tem. The number of items in any time period \( T \) is the
integral of the arrival rate over the period, so if there is
a steady state average rate \( \lambda \), then \( L = \lambda W \) is the area of
the rectangle under the dotted line for time period \( W \).
A more formal proof appears in [3] or [4].

Notice that the units in the expression are like that of
a speed equation: work equals rate times time. Here,
“work” is items that complete processing, “rate” is the
arrival rate, and “time” is the time spent in the system.

The result is from queueing theory, and the meaning
of \( L \), \( \lambda \), and \( W \) varies by application. In manufacturing,
\( L \) might mean inventory, \( \lambda \) the rate of arrival of mate-
rials, and \( W \) the amount of time the materials spend
Little’s Law. Fig. 1 Visualization of Little’s law

in inventory. In retail applications, $L$ might mean the average number of customers in a store, $\lambda$ the rate they arrive at the store, and $W$ the average time a customer spends in the store.

Example

Suppose a fast food restaurant has an average business of one customer every two minutes, and the average customer takes 40 minutes to eat. For purposes of planning seating to accommodate the customers, Little’s Law says how many seats in the restaurant will be in use, on average. Here, $\lambda = 0.5$ customers per minute, and $W = 40$ minutes; hence, $L = 0.5 \times 40 = 20$ occupied seats.

If the business doubles to $\lambda = 1.0$ customers per minute and the restaurant does not have enough seating area, then Little’s Law says there are two solutions: double the seating area, or halve the average amount of time customers spend eating to 20 minutes, say, by making the seats less comfortable.

Compound Queues

It is often useful to apply Little’s Law in compound ways. For example, a store has a rate at which customers arrive and are in the process of browsing, and after selecting an item, they enter the checkout queue within the store to pay for the item. Thus, the law predicts the length of the checkout line and how crowded the store will be. Alternatively, it can predict how many checkout tellers suffice to prevent the checkout lines from averaging more than a certain goal in length.

Example

Suppose the same fast food business assesses the trade-off between the cost of personnel and the effect waiting in line has on customer satisfaction. It then sets a goal that the average amount of time a customer wishing to order food should have to wait in line is 6 minutes. Suppose also that there are two queues for ordering. How long will the line be to order food, on the average?

Since the two queues must accommodate the total arriving customer rate of $\lambda = 1.0$ customers per minute, the rate for each food-ordering line is 0.5 customers per minute. Each line will then have an average length of

$$L = \lambda W = 0.5 \times 6 = 3 \text{ customers.}$$

Implications for Computer Design

Little’s Law is of obvious value for computer design, especially for decoupled architectures. Computer architect Burton Smith first noted the following form of Little’s Law in 1995:

Concurrency = Latency × Bandwidth.

Bandwidth is a rate of arrival of data into a processing queue. Latency is the time spent in the queue, and concurrency (or degree of parallelism) is the number of simultaneous data items to process. Some refer to this as Little’s Law of High Performance Computing [1].

Processing elements require holding areas for data (buffers), and Little’s Law provides guidance for how to match the processing rate, storage size of a buffer, and time a datum spends in the buffer. Just as more checkout tellers can reduce the average size of the checkout queue, parallel computing elements can reduce the amount of buffer space and the time spent in the buffer.

Example

Suppose the processing units in a parallel computer can collectively process 1 trillion ($10^{12}$) items of data in main memory per second, and the latency between the
main memory and the processors is 50 ns. Then Little's Law says that the number of parallel elements for processing memory must be

\[ \text{Concurrency} = 10^{12} \times (50 \times 10^{-9}) = 50,000 \text{ processors}. \]

The parallel elements could be in the form of pipeline stages, data parallel units, or any other way that permits concurrent activity at the level of 50,000 items of data processed at once. Thus, if the system consists of 10,000 servers organized as a networked cluster, then each server must have an internal parallelism of a factor of five to accommodate the demands of the queue.

Even within a single processor, Little's Law is quite useful for answering architectural questions. For example, if the latency to the main memory is 60 ns for one core of a multicore processor, and the core operates at 3.0 GHz, then the memory system must be able to accommodate

\[ (3.0 \times 10^9) \times (60 \times 10^{-9}) = 180 \text{ outstanding memory references}, \]

if each core is to make full use of its capability (that is, if the average arrival rate is the same as the maximum possible arrival rate).

Another important computer application of Little's Law is in assuring the fairness of benchmarks, especially those related to transaction processing. When measuring a system, if the benchmarking engineer asks it to process so many transactions (arrival rate \( \lambda \)) that the storage (number of items \( L \) in the system) is larger than the main memory and thus spills into mass storage, the performance will drop catastrophically.

**Perspective**

Operations Research (OR) originated in the 1950s, and courses on the subject rely heavily on queuing theory. Little was teaching formal queuing theory at Case Western Reserve University and noticed that the \( L = \lambda W \) behavior was independent of the probability distribution of the arrival times. At the urging of one of his students (Sid Hess), Little published the proof of the generality of the law. Little's 1961 theorem was one of the simplest yet most useful results from that early era. While formal queuing theory requires an understanding of concepts like Poisson distributions and a somewhat advanced level of statistics and mathematics, the simple formula \( L = \lambda W \) is one that requires no advanced training to understand and apply. There was no thought in those early years of OR about answering questions of computer design. The traditional textbook examples are for inventory, manufacturing, order entry, and so on. Computer designs of that era tended to be tightly coupled, and so deterministic that the only statistical queuing behavior that would enter into their use would be that of users submitting jobs to the batch queue and waiting for computed results.

Burton Smith (Tera Computer) and David Bailey (NASA Ames Research) deserve credit for showing, in the late 1990s, the applicability of Little's Law to many issues of computing design. It seems obvious in retrospect that a law originally intended for use in the arena of business operations with human time scales could apply to computing design. Here the use is a subtle variation on the “average” behavior, in that the law assesses the optimum behavior. If the processing speed is to be some idealized maximum, and the latency is as small as possible, then the law predicts a lower bound on the capacity or parallelism required for near-peak performance.

Little's Law for High Performance Computing provides perhaps the simplest way to explain the necessity for the parallel computing approach. Latency tends to be difficult to reduce because of the laws of physics; concurrency is the product of latency and bandwidth (processing rate), so increasing bandwidth forces the need for more concurrency.

**Related Entries**

- Bandwidth-Latency Models (BSP, LogP)
- Dependences
- Dependence Abstractions
- Network Obliviousness
- NUMA Caches
- Pipelining

**Bibliographic Notes and Further Reading**

Little's original 1961 paper is of historic interest, as is a less general precedent text (1958) by OR pioneer Philip Morse: *Queues, Inventories and Maintenance* [6]. A very readable account with many examples drawn from everyday situations is in [5]. The latter work also gives perspective on the impact of the law over four
decades, and provides examples from managing email, traffic created by toll booths, and the housing market.


Bibliography

Livermore Loops

JACK DONGARRA, PIOTR LUSZCZEK
University of Tennessee, Knoxville, TN, USA

Definition
Livermore Loops are a set of 24 Fortran DO-loops (The Livermore Fortran Kernels, LFK) extracted from operational codes used at the Lawrence Livermore National Laboratory [1, 2]. They have been used since the early 1970s to assess the arithmetic performance of computers and their compilers. They are a mixture of vectorizable and nonvectorizable loops and test rather fully the computational capabilities of the hardware as well as the skill of the software in compiling and vectorization of efficient code. The main value of the benchmark is the range of performance that it demonstrates, and in this respect it complements the limited range of loops tested in the LINPACK benchmark.

Discussion
As a benchmark, the Livermore Loops provide the individual performance of each loop, together with various averages (arithmetic, geometric, harmonic) and the quartiles of the distribution. However, it is difficult to give a clear meaning to these averages, and the value of the benchmark is more in the distribution itself. In particular, the maximum and minimum give the range of likely performance in full applications. The ratio of maximum to minimum performance has been called the instability or the specialty [3], and is a measure of how difficult it is to obtain good performance from the computer, and therefore how specialized it is. The minimum or worst performance obtained on these loops is of special value, because there is much truth in the saying that “the best computer to choose is that with the best worst-performance.”

The LFK set began in the early 1970s with only 12 kernels and was timed on the mainframes of the era with CPU-times of microsecond accuracy. By the early 1980s the number of kernels in the set grew to 24 and, during the decade, statistical features of the results reporting code were enhanced and adaptations were made to accommodate crude UNIX timers of only millisecond resolution. A port of LFK to C happened in the middle
of 1980s. By the beginning of the 1990s, the compilers were able to perform code-hoisting which necessitated changes to the code like introducing functions to embed the critical kernels inside it and hide it from the optimization. The 1990s brought new hardware from HP, SGI, and SUN which exhibited vast performance improvements but still suffered with low timer resolution which required increase in the repetition count to assure 1% accuracy of the reported timings.

The origins or functions of each of the LFK kernels are the following:

1. Fragment of hydrodynamics code
2. ICCG (Incomplete Cholesky Conjugate Gradient) excerpt
3. Inner product calculation
4. Banded linear equations
5. Tri-diagonal elimination below the diagonal
6. General linear recurrence equations
7. Fragment of equation of state code
8. ADI (Alternating Direction Implicit) integration scheme for differential equations
9. Integrate predictor kernel
10. Difference predictor kernel
11. Summation with storage of partial sums
12. First difference scheme
13. 2D PIC (Particle In Cell) code
14. 1D PIC
15. A challenging (for the compiler) ordering of scalar operations in FORTRAN
16. Monte Carlo search loop
17. Implicit conditional computation
18. 2D explicit hydrodynamics fragment
19. General linear recurrence equations
20. Discrete ordinates transport: recurrence
21. Matrix–matrix multiplication
22. Planckian probability distribution
23. 2D implicit hydrodynamics fragment
24. A loop that finds location of first minimum in an array

In the LFK collection, scalar tests are numbered 5, 11, 17, 19, and 20 ([1], page 14). In the late 1980s, the scalar features of the contemporary machines were important for supercomputing at Los Alamos (LANL) – as much as 30% of the code executed at the laboratory might have been scalar [4]. The LFK scalar codes are small, so their indicated performance is generally higher than that measured with larger benchmarks. Nonetheless, a good correlation exists among the LFK scalar tests and others [5].

Vectorization at the compiler level was turned on and off in the LFK kernels through special comments that were recognized as compiler directives. Similarly, code directives were used to render selected kernels scalar and force their execution onto the scalar units of the processor. The scalar execution was used as the definition of necessary computation and the appropriated number of the performed floating point operations. The actual number of floating point operations for vectorized execution could have been larger. For the purposes of the calculation of the performance results, each type of floating operation was weighted. The weight represented the execution time relative to floating point division. For example, the weight of floating point division was 4 which meant that it takes four times as many cycles to complete a division than an addition of two floating point numbers.

The authors of LFK suggested to use the results by weighing their performance in proportion to the actual usage of that category of computation in the total workload. For example, for a hydrodynamics code, kernels 1, 18, and 23 would be the most relevant and consequently they should be given the greatest weight.

Another important aspect of performance benchmarking stressed by the LFK report was the treatment of the results' data as statistical quantity. A single number quoted as the performance rate was considered insufficient. Statistical sampling was encouraged and each reported number was to be accompanied by minimum and maximum values obtained as well as the equi-weighted harmonic, geometric, and arithmetic means. In addition, the first, second, third, and fourth quartiles were used to perform sensitivity analysis of harmonic mean rate with respect to various weight distributions.

At the time of release of the LFK set, the available hardware included machines from Cray, DEC, IBM, and NEC. The clock frequencies were of order 100 MHz and the achieved performance varied between single digit MFlop/s to man hundreds of MFlop/s depending on the kernel and the type of processor (vector or RISC).

Related Entries
- Benchmarks
- HPC Challenge Benchmark
Load Balancing

Definition

Load balancing in distributed memory systems is the process of redistributing work between hardware resources to improve performance, typically by moving work from overloaded resources to underloaded resources.

Discussion

In a parallel application, when work is distributed unevenly so that underloaded processors are forced to wait for overloaded processors, the application is suffering from load imbalance. Load imbalance is one of the key impediments in achieving high performance on large parallel machines, especially when solving highly dynamic and irregular problems. Load balancing is a technique that performs the task of distributing computation and communication load across the hardware resources of a parallel machine so that no single processor is overloaded. It can reduce processor idle time across the machine while also reducing communication costs by colocating related work on the same processor.

Balancing an application’s load involves making decisions about where to place newly created computational tasks on processors, or where to migrate existing work among processors. Orthogonally, some applications only require static load balancing: They have consistent behaviors over their lifetimes, and once they are balanced they remain balanced. Other applications exhibit dynamic changes in behavior and require periodic rebalancing. Typically it is far too expensive to compute an optimal distribution of work in any realistic application, but a wide variety of heuristic algorithms have been developed that are very effective in practice across a wide variety of application domains.

The load associated with an application can be conceptualized as a graph, where the nodes are discrete units of work and an edge exists between two nodes if there is a communication between them. To solve the load balancing problem one must split this graph into parts, with each part representing the work to be associated with a particular hardware resource. This partitioning process is at the heart of load balancing.

Given the heuristic nature of load balancing, which makes perfect load balance an unrealistic goal, it is important to remember that the goal of load balancing is not so much to equalize the amount of work on each processor as it is to minimize the load of the most heavily loaded processor in the system. The most overloaded processor is the bottleneck that determines time to completion. A heuristic that leaves some processors idle is preferable to one that reduces the variance in load, as long it reduces the load on the most loaded processor. Load balancing heuristics can also take factors beyond time to completion into account, for example by trying to minimize power usage over an application’s lifetime.

Modern HPC systems include clusters of multi-core nodes. The general load balancing strategies described
in this entry are still applicable at the level of nodes (i.e., work is assigned to nodes). Finer grained load balancing within a node (work assigned to cores) can be performed independently. In the simplest case, it can be done by applying the same load balancing strategies that are used for node-level load balancing, except that the cost of communication among cores within a node is much smaller than the internode communication, and the number of cores involved within a node is relatively small.

**Periodic Load Balancing**

In a periodic load balancing scheme, computational tasks are persistent; load is balanced only as needed, and the balancing consists of migrating existing tasks and their associated data. With periodic load balancing, expensive load balancing decision making and data migration are not continuous processes. They occur only distinct point in the application when a decision to do balancing has been made. Periodic load balancing schemes are suitable for iterative scientific applications such as molecular dynamics, adaptive finite element simulation, and climate simulation, where the computation typically consists of a large number of time steps, iterations (as in iterative linear system solvers), or a combination of both. A computational task in these applications is executed for a long period of time, and tends to be persistent. During execution, at periodic intervals, partially executed tasks may be moved to different processors to achieve global load balance.

**The Load Balancing Process**

In an application which rebalances its load periodically, there are four distinct steps in the load balancing process. The first step is load estimation, which tries to gauge what the load on each processor will be in the near future if no rebalancing is done. This may involve the use of a model that predicts future performance based on current conditions, or it may be based directly on past measured load, with the assumption that the near future will closely resemble the near past. The second step is making a decision of when to execute load balancing. This is a trade-off between the cost of load balancing itself (i.e., the cost of determining a new mapping of application tasks to hardware and the cost of migration) and the savings one can expect from a better load distribution. The nature of the application being balanced and the cost of the load balancing method to be used factor heavily into this decision. The third step is determining how the load will be rebalanced. There are many algorithms devoted to computing a good mapping of work onto processors, each with its own strengths and weaknesses. This section describes some of the most widely used methods. The general problem of creating a mapping of work onto processors which minimizes time to completion is NP-hard, so all strategies we discuss are heuristic. The final step in the load balancing process is migration, the process by which work is actually moved. This may simply be a matter of relocating application data, but it can also encompass thread and process migration.

**Initial Balancing**

For some classes of application, there is very little change in load balance over time once the application reaches a steady state. Thus, once good load balance is achieved, no further balancing need be done. However, load balancing is often still an important part of such applications. For example, consider the case of molecular dynamics, where one must distribute simulated particles onto processors. These applications do not experience significant dynamic load imbalance, but determining a good initial mapping may be difficult because of the difficulty of estimating the load of multiple computational tasks accurately a priori.

In cases like this, one can simply start the application with an unoptimized distribution of work, run the application long enough to accurately measure the load, rebalance based on those measurements, and then continue without the need for any further balancing.

**Classifying Load Balancers**

There are many families of load balancing approaches that can be classified according to how they answer the fundamental questions of load balancing: How do we estimate the load, at what granularity should one balance the load, and where should the load balancing decisions be made.

Load estimation underlies all load balancing algorithms. Load balancing is fundamentally a forward-looking task which aims to improve the future performance of the application. To do this effectively, the load balancer must have some model of what the
future performance of the application will be in different scenarios. There are two common approaches to estimating future load. The first is to measure the current load associated with each piece of work in the application and to assume that this load will remain the same after load balancing. This assumption is based on the principle of persistence, which posits that, for certain classes of scientific and engineering applications, computational loads and communication patterns tend to persist over time, even in dynamically evolving computations. This approach has several advantages: It can be applied to any application, it accurately and automatically accounts for the particular characteristics of the machine on which the application is running, and it removes some of the burden of load estimation from the application developer.

The alternative is to build some model of application performance which will provide a performance estimate for any given distribution of work. These models can be sophisticated enough to take into account dynamic changes in application performance that a measurement-based scheme will not account for, but they must be created specifically to match the actual characteristics of the application they are used for. If the model does not match reality then poor load balancing may result.

Load balancing may take place at any of several levels of granularity. The greatest flexibility in mapping work onto hardware resources can be achieved by exposing the smallest possible meaningful units of data to the load balancer. For example, these may be nodes in a finite element application or individual particles in a molecular simulation. Alternatively, one may group this data into larger chunks and only expose those chunks to the load balancing algorithm. This makes the load balancing problem smaller while guaranteeing respect for locality within the chunks. For example, in a molecular simulation the load balancer might only try to balance contiguous regions which may contain a large number of particles without being directly aware of the particles. It is also possible to balance load by migrating entire processes, avoiding the need for application developers to write code dedicated to migrating their data during the load balancing process.

Load balancers may be further categorized according to where decisions are made: locally, globally, or according to some hierarchical scheme. Global load balancers collect global information about load across the entire machine and can use this information to make decisions that take the entire state of the application into account. The advantage of these schemes is that load balancing decisions can take a global view of the application, and all decisions about which objects should migrate to which processors can be made without further coordination during the load balancing process. However, these schemes inherently lack scalability. As problem sizes increase, the object communication graph may not even fit in memory on a single node, making global algorithms infeasible. Even if the load balancing process is not constrained by memory, the time required to compute an assignment for very large problems may preclude the use of a global strategy. However, with coarse granularity, it is possible to use global strategies up to several thousand processors, especially if load balancing is relatively infrequent.

Parallelized versions of global load balancing schemes are also possible. These use the same principles for partitioning as a serial global scheme, but to improve scalability the task graph is spread across many processors. This introduces some overhead in the partitioning process, but allows much larger problems to be solved than a purely serial scheme. One example of a parallel global solver is ParMETIS, the parallel version of the METIS graph partitioner.

Distributed load balancing schemes lie at the opposite end of the scale from global schemes. Distributed schemes use purely local information in their decision-making process, typically looking to offload work from overloaded processors to their less-loaded immediate neighbors in some virtualized topology. This leads to a sort of diffusion of work through the system as objects gradually move away from overloaded regions into underloaded regions. These schemes are very cheap computationally, and do not require global synchronization. However, they have the disadvantage of redistributing work slowly compared to global schemes, often requiring many load balancing phases before achieving system-wide load balance, and they typically perform more migrations before load balance is achieved.

Hybrid load balancing schemes represent a compromise between global and distributed schemes. In a hybrid load balancer, the problem domain is split
into a hierarchy of sub-domains. At the bottom of the hierarchy, global load balancing algorithms are applied to the small sub-domains, redistributing load within each sub-domain. At higher levels of the hierarchy, the load balancing strategies operate on the sub-domains as indivisible units, redistributing these larger units of work across the machine. This keeps the size of the partitioning problem to be solved low without giving up some global decision-making capabilities.

**Algorithms**

An application’s load balancing needs can vary widely depending on its computational characteristics. Accordingly, a wide variety of load balancing algorithms have been developed. Some do careful analysis of an application’s communication patterns in an effort to reduce internode communication; others only try to minimize the maximum load on a node. Some try to do a reasonably good job as quickly as possible; others take longer to provide a higher quality solution. Some are even tailored to meet the needs of particular application domains. Despite the wide variety of algorithms available, there are a few in wide use that demonstrate the range of available techniques.

**Parallel Prefix**

In some cases, the pattern of communication between objects is unimportant, and we care only about keeping an equal amount of work on each processor. In this scenario, load may be effectively distributed using a simple parallel prefix strategy (also known as scan or prefix sum).

The input to the parallel prefix algorithm is simply a local array of work unit weights on each processor reflecting the amount of work that each unit represents. The classic parallel prefix algorithm computes the sum of the first $i$ units for each of the $n$ unit in the list. Once the prefix sum is computed, all work can be equally distributed across processors by sending each unit to the processor number given by its prefix sum value divided by the total work divided by the number of processors.

The primary advantage of the parallel prefix scheme is its fast, inexpensive nature. The computation can be shared across $p$ processors using only $\log p$ communications, and efficient prefix algorithms are widely available in the form of library functions like `MPI_Scan`. An assignment of work units to processors based on parallel prefix can be computed virtually instantaneously even for extremely large processor counts, regardless of variations in work unit weights.

The corresponding disadvantage of using the parallel prefix algorithm for load balancing is its simplicity. It does not account for the structure of communication between processors and will not attempt to minimize the communication volume of the resulting partition. It also does not attempt to minimize the amount of migration needed.

**Recursive Bisection**

Recursive bisection is a divide-and-conquer technique that reduces the partitioning problem to a series of bisection operations. The strategy of recursive bisection is to split the object graph in two approximately equal parts while minimizing communication between the parts, then proceeding to subdivide each half recursively until the required number of partitions is achieved. This recursive process introduces parallelism into the partitioning process itself. After the top-level split is completed, the two child splits are independent and can be computed in parallel, and after $n$ levels of bisection there are $2^n$ independent splitting operations to perform. In addition, geometric bisection operations can themselves be parallelized by constructing a histogram of the nodes to be split.

There are many variations of the recursive bisection algorithm based on the algorithm used to split the graph. Orthogonal recursive bisection (ORB), shown in Fig. 1, is a geometric approach in which each object is associated with coordinates in some spatial domain. The bisection process splits the domain in two using an axis-aligned cutting plane. This can be a fast way of creating a good bisection if most communication between objects is local. However, this method does not guarantee that it will produce geometrically connected subdomains, and it may also produce sub-domains with high aspect ratios. Further variations exist which allow cutting planes that are not axis-aligned.

An alternate approach is to use spectral methods to do the bisection. This involves constructing a sparse matrix from the communication graph, finding an eigenvector of that matrix, and using it as a separator field to do the bisection. This method can produce superior results compared with ORB, but at a substantially higher computational cost.
Load Balancing, Distributed Memory. Fig. 1 Orthogonal recursive bisection (ORB) partitions by finding a cutting plane that splits the work into two approximately equal pieces and recursing until the resulting pieces are small enough [8]

For many problems, recursive bisection is a fast way of computing good partitions, with the added benefit that the recursive procedure naturally exposes parallelism in the partitioning process itself. The advantages and disadvantages of recursive bisection depend greatly on the nature of the bisection algorithm used, which can greatly affect partition quality and computational cost.

Space-Filling Curve
A space-filling curve is a mathematical function that maps a line onto the entire unit square (in two dimensions, or the entire unit $N$-cube in $N$ dimensions). There are many such curves, and they are typically defined as the limit of sequence of simple curves, so that closer and closer approximations to the space-filling curve can be constructed using an iterative process. The value of space-filling curves for load balancing is that they can be used to convert n-dimensional spatial data into one-dimensional data. Once the higher-dimensional data has been linearized, it can be easily partitioned by splitting the line into pieces with equal numbers of elements.

Consider the case where each object in the application has a two-dimensional coordinate associated with it. Starting with a coarse approximation to the space-filling curve, a recursive refinement process can be used to create closer and closer approximations until each object is associated with its own segment of the approximated curve. This creates a linear ordering of the objects, as depicted in Fig. 2, which can then be split into even partitions.

Load Balancing, Distributed Memory. Fig. 2 To partition data using a space-filling curve, a very coarse approximation of the curve is replaced with finer and finer approximations until each unit of work is in its own segment. The curve is then split into pieces, with each piece containing an equal number of work units. This figure shows a Hilbert curve [8]

A good choice of space-filling curve will tend to keep objects which are close together in the higher-dimensional space. This property is necessary so that the partitions that result respect the locality of the data. Many different curves have been used for load balancing purposes, including the Peano curve, the Morton curve, and the Hilbert curve. The Hilbert curve is a common choice of space-filling curve for partitioning applications because it provides good locality.

Graph Partitioning
The pattern of communication in any parallel program can be represented as a graph, with nodes representing discrete units of work, and weighted edges representing communication. Graph partitioning is a general-purpose technique for splitting such a graph into pieces, typically with the goal of creating one piece for each processor. There are two conflicting goals in graph partitioning: achieving equal-size partitions, and minimizing the total amount of communication across partition boundaries, known as the edge cut. Finding an optimal solution to the graph partitioning problem is NP-Hard, so heuristic algorithms are required.

Because it is computationally infeasible to attempt to partition the whole input graph at once, the graph partitioning algorithm proceeds by constructing a series of...
coarser representations of the input graph by combining distinct nodes in the input graph into a single node in the coarser graph. By retaining information about which nodes in the original graph have been combined into each node of the coarse graph, the algorithm maintains basic information about the graph’s structure while reducing its size enough that computing a partitioning is very simple.

Once the coarsest graph has been partitioned, the coarsening process is reversed, breaking combined nodes apart. At each step of this uncoarsening process, a refinement algorithm such as the Kernighan–Lin algorithm can be used to improve the quality of the partition by finding advantageous swaps of nodes across partition boundaries. Once the uncoarsening process is complete, the user is left with a partitioning for the original input mesh.

This technique can be further generalized to operate on hypergraphs. Whereas in a graph an edge connects two nodes, in a hypergraph a hyperedge can connect any number of nodes. This allows for the explicit representation of relationships that link several nodes together as one edge rather than the standard graph representation of pairwise edges between each of the nodes. For example, in VLSI simulations hypergraphs can be used to more accurately reflect circuit structure, leading to higher quality partitions. Hypergraphs can be partitioned using a variation of the basic graph partitioning algorithm which coarsens the initial hypergraph until it can be easily partitioned, then refining the partitioned coarse hypergraph to obtain a partitioning for the full hypergraph.

The primary advantages of load balancing schemes based on graph partitioning are their flexibility and generality. An object communication graph can be constructed for any problem without depending on any particular features of the problem domain. The resulting partition can be made to account for varying amounts of work per object by weighting the nodes of the graph and to account for varying amounts of communication between objects by weighting the edges. This flexibility makes graph partitioners a powerful tool for load balancing.

The primary disadvantage of these schemes is their cost. Compared to the other methods discussed here, graph partitioning may be computationally intensive, particularly when a very large number of partitions is needed. The construction of an explicit communication graph may also be unnecessary for problems where communication may be inferred from domain-specific information such as geometric data associated with each object.

Rebalancing with Refinement Versus Total Reassignment

There are two ways to approach the problem of rebalancing the computational load of an application. The first approach is to take an existing data distribution as a baseline and attempt to incrementally improve load balance by migrating small amounts of data between partitions while leaving the majority of data in place. The second approach is to perform a total repartitioning, without taking the existing data distribution into account. Incremental load balancing approaches are desirable when the application is already nearly balanced, because they impose smaller costs in terms of data motion and communication. Incremental approaches are more amenable to asynchronous implementations that operate using information from only a small set of processors.

Software Frameworks

Many parallel dynamic load balancing libraries and frameworks have been developed before for specialized domains. These libraries are often particularly useful because they allow application developers to specify the structure of their application once and then easily try a number of different load balancing strategies to see which is most effective in practice without needing to reformulate the load measurement and task graph construction process for each new algorithm.

The Zoltan toolkit [1] provides a suite of dynamic load balancing and parallel repartitioning algorithms, including geometric, hypergraph, and graph methods. It provides a simple interface to switch between algorithms, allowing straightforward comparisons of algorithms in applications. The application developers provide an explicit cost function and communication graph for the Zoltan algorithms to use.

Charm++ [2] adopts a migratable object-based load balancing model. As such, Charm programs have a natural grain size determined by their objects, and can be load balanced by redistributing their objects among processors. Charm provides facilities for automatically
measuring load, and provides a variety of associated measurement-based load balancing strategies that use the recent past as a guideline for the near future. This avoids the need for developers to explicitly specify any cost functions or information about the application’s communication structure. Charm++ includes a suite of global, distributed, and hierarchical load balancing schemes.

DRAMA [3] is a library for parallel dynamic load balancing of finite element applications. The application must provide the current distributed mesh, including information about its computation and communication requirements. DRAMA then provides it with all necessary information to reallocate the application data. The library computes a new partitioning, either via direct mesh migration or via parallel graph re-partitioning, by interfacing to the ParMetis or Jostle graph partitioning libraries. This project is no longer under active development.

The Chombo [4] package was developed by Lawrence Berkeley National Lab. It provides a set of tools including load balancing for implementing finite difference methods for the solution of partial differential equations on block-structured adaptively refined rectangular grids. It requires users to provide input indicating the computational workload for each box, or mesh partition.

**Task Scheduling Methods**

Some applications are characterized by the continuous production of tasks rather than by iterative computations on a collection of work units. These tasks, which are continually being created and completed as the application runs, form the basic unit of work for load balancing. For these applications, load balancing is essentially a task scheduling or task allocation problem. This task pool abstraction captures the execution style of many applications such as master/worker and state-space search computations. Such applications are typically non-iterative.

Load balancing strategies in this category can be classified as centralized, fully distributed, or hierarchical. Hierarchical strategies are hybrids that aim to combine the benefits of centralized and distributed methods. In centralized strategies, a dedicated “central” processor gathers global information about the state of the entire machine and uses it to make global load balancing decisions. On the other hand, in a fully distributed strategy, each processor exchanges state information only with other processors in its neighborhood.

Fully distributed load balancing received significant attention from the early days in parallel computing. One way to categorize them is based on which processor initiates movement of tasks.

Sender-initiated schemes assign newly created tasks to some processor, chosen randomly or from one of the neighbors in a physical or virtual topology. The decision to assign it to another processor, instead of retaining the task locally may also be taken randomly, or based on a load metric such as a queue size. Random assignment has some good statistical properties but suffers from high communication costs, by requiring that most tasks be sent to remote processors. With neighborhood strategies, global balancing is achieved as tasks are moved from heavily loaded neighborhood diffuse onto lightly loaded processors. In the adaptive contraction within neighborhood (ACWN) scheme, tasks always travel to topologically adjacent neighbors with the least load, but only if the difference in loads is more than a predefined threshold. In addition, ACWN does saturation control by classifying the system as being either lightly, moderately, or heavily loaded.

In receiver-initiated schemes, the underloaded processors request load from heavily loaded processors. You may chose the victim to request work from randomly, or via a round-robin policy, or from among “neighboring” processors. Randomized work stealing is yet another distributed dynamic load balancing technique, which is used in some runtime systems such as Cilk.

The distinction between sender or receiver initiation is blurred when processors exchange load information, typically with neighbors in a virtual topology. Although the decision to send work is taken by the sender, it is taken in response to load information from the receiver. For example, in neighborhood averaging schemes, after periodically exchanging load information with neighbors in a virtual (and typically, low-diameter) topology, each processor that is overloaded compared with its neighbors, sends equalizing work to its lower-loaded neighbors. Such policies tend to be proactive compared with work stealing, trading better load balance for extra communication.
Another strategy in this category, and one of the oldest ones, is the gradient model. Here each processor participates in a continuous fixed-point computation with its neighbors to identify the neighbor that is closest to an idle processor. Overloaded processors then send work toward the idle processors via that neighbor.

The gradient model is a demand-driven approach. In gradient schemes, underloaded processors inform other processors of their state, and overloaded processors respond by sending a portion of their load to the nearest lightly loaded processor. The resulting effect is a form of a gradient map that guides the migration of tasks from overloaded to underloaded processors.

The dimensional exchange method is a distributed strategy in which balancing is performed in an iterative fashion by “folding” an $P$ processor system into $\log P$ dimensions and balancing one dimension at a time. That is, in phase $i$, each processor exchanges load with its neighbor in the $i$th dimension so as to equalize load among the two. After $\log P$ phases, the load is balanced across all the processors. This scheme is conceptually designed for a hypercube system but may be applied to other topologies with some modification.

Several hierarchical schemes have been proposed that avoid the bottleneck of a centralized strategies, while retaining some of their ability to achieve global balance quickly. Typically, processors are organized in a two-level hierarchy. Managers at the lower level behave like masters in centralized strategies for their domain of processors, and interact with other managers as processors in a distributed strategies. Alternatively, load balancing is initiated at the lowest levels in the hierarchy, and global balancing is achieved by ascending the tree and balancing the load between adjacent domains at each level in the hierarchy.

Often, priorities are associated with tasks, especially for applications such as branch-and-bound or searching for one solution in a state-space search. Task balancing for these scenarios is complicated because of the need to balance load while ensuring that high priority work does not get delayed by low priority work. Sender-initiated random assignment as well as some hierarchical strategies have shown good performance in this context.

### Related Entries
- Chaco
- Graph Partitioning
- Hypergraph Partitioning
- Reduce and Scan
- Scan for Distributed Memory, Message-Passing Systems
- Space-filling Curves
- Task Graph Scheduling
- Topology Aware Task Mapping

### Bibliographic Notes and Further Reading
There is a rich history of load balancing literature that spans decades. This entry is only able to cite a very small amount of this literature, and attempts to include modern papers with broad scope and great impact. Kumar et al. [5] describe the scalability and performance characteristics of several task scheduling schemes, and Devine et al. [6] gives a good overview of the range of techniques used for periodic balancing in modern scientific applications and the load balancing challenges faced by these applications. Xu and Lau [7] give an in-depth treatment of distributed load balancing articles, covering both mathematical theory and actual implementations.

For practical information on integrating load balancing frameworks into real applications, literature describing principles and practical use of such systems as DRAMA [3], Charm++ [2, 9], Chombo [4], and Zoltan [1] is a crucial resource.

More information is available on the implementation of task scheduling methods, whether they are centralized [10, 11], distributed [12–14], or hierarchical [14]. Work stealing is described in detail by Kumar et al. [5], while associated work on Cilk is described in Frigo et al. [15]. Dinan et al. [16] extended work stealing to run on thousands of processors using ARMCi.

### Bibliography
Locality of Reference and Parallel Processing

KESHAV PINGALI
The University of Texas at Austin, Austin, TX, USA

Definition

The term *locality of reference* refers to the fact that there is usually some predictability in the sequence of memory addresses accessed by a program during its execution. Programs are said to exhibit *temporal locality* if they access the same memory location several times within a small window of time. Programs that access nearby memory locations within a small window of time are said to exhibit *spatial locality*. Many parallel processors are nonuniform memory access (NUMA) machines in which the time required by a given processor to access a memory location may depend on the address of that location. Parallel programs that exploit this nonuniformity are said to exhibit *network locality*. Exploiting all types of locality is critical in parallel processing because the performance of most programs is limited by the latency of memory accesses. Hardware mechanisms for exploiting locality include caches and pre-fetching. Software mechanisms include program and data transformations, software pre-fetching, and topology-aware mappings of computations and data. Multithreading can reduce the impact of memory latency on programs that have a lot of parallelism but little locality of reference.

Discussion

Introduction

The sequence of memory addresses accessed by a program or a thread during its execution is called its memory trace. Memory traces are not random number sequences since there is some predictability to the addresses in most traces, which arises from fundamental features of the von Neumann model of computation. For example, instructions at address i is executed and it is not a branch
Locality of Reference and Parallel Processing

Locality of Reference

Locality of Reference and Parallel Processing

Instruction, the instruction at address \( i + 1 \) must be executed next. This kind of predictability in address traces is called locality of reference, and exploiting it is key to obtaining good performance on sequential and parallel machines.

There are three kinds of locality of reference: temporal, spatial, and network.

Temporal Locality

A program execution exhibits temporal locality if the occurrences of a given memory address in its memory trace occur close to each other. A program is said to exhibit temporal locality if its executions for most inputs of interest exhibit temporal locality.

Temporal locality in instruction addresses arises mainly from the execution of loops (in particular, innermost loops in loop nests). For example, when multiplying two \( N \times N \) matrices using the standard algorithm shown below, the body of the innermost loop is executed \( N^3 \) times back to back, resulting in temporal locality in instruction addresses. Temporal locality in instruction addresses may also arise from the execution of “hot” methods that are called repeatedly during program execution.

\[
\text{for (i = 0; i < N; i++)}
\text{for (j = 0; j < N; j++)}
\text{for (k = 0; k < N; k++)}
\text{C[i][j] = C[i][j] + A[i][k] * B[k][j];}
\]

Temporal locality in data addresses may arise from accesses to constants and memory locations accessed repeatedly within loops. For example, in the matrix multiplication code shown above, every iteration of the innermost loop for given outer loop indices \( (i, j) \) accesses the same array location \( C[i][j] \), giving rise to temporal locality. Some compilers can exploit this temporal locality by loading \( C[i][j] \) into a register just before the innermost loop begins execution, and performing the additions to this register rather than to the memory location \( C[i][j] \) (the value in the register is written back to memory when loop finishes execution). In contrast, there is little temporal locality in the references to the \( B \) array since successive accesses to a given element of \( B \) are separated by \( O(N^2) \) other addresses in the trace.

Spatial Locality

A program execution exhibits spatial locality if the occurrences in its trace of nearby memory addresses occur close together in that trace. A program is said to exhibit spatial locality if its executions for most inputs of interest exhibit spatial locality.

Spatial locality in instruction accesses arises because instructions between branches are executed sequentially, so if an instruction at address \( i \) is executed and it is not a branch, the instruction at address \( i + 1 \) must be executed next. Instruction accesses within innermost loops therefore exhibit both temporal and spatial locality. Programs that perform operations on vectors exhibit spatial locality if successive vector elements are accessed during the computation, such as the following program which adds the elements of two vectors:

\[
\text{for (i = 0; i < N; i++)}
\text{C[i] = A[i] + B[i];}
\]

Notice that in this program, there are no fewer than four sources of spatial locality: the accesses to the three vectors, and the instructions in the loop body. Programs that access vector elements with some small stride also exhibit spatial locality.

Programs that access arrays of two or more dimensions will exhibit spatial locality if array elements are accessed successively (or with some small non-unit stride) in the order in which they are stored in memory. For example, in the C programming language, arrays are stored in row-major order, so the elements of the matrix are stored row by row in memory. If a program accesses such arrays row by row, these accesses will exhibit spatial locality. In the matrix multiplication code shown above, the accesses to arrays \( A \) and \( C \) have spatial locality of reference, since these arrays are accessed row by row, but the accesses to array \( B \) do not since this array is accessed by columns.

Network Locality

In a parallel computer in which memory addresses are distributed across the processors, a processor may have faster access to memory locations mapped locally than to memory locations mapped to other processors. There may also be differences in the access times to memory locations mapped to different processors because most communication networks for parallel computers are multi-stage networks in which communication packets from a given processor may go through different numbers of stages to get to different processors, and each
stage adds some latency to the communication. Parallel computers that exhibit these kinds of nonuniform memory access times are called Non-Uniform Memory Access (NUMA) parallel computers. A parallel program that attempts to exploit features of NUMA parallel computers is said to exhibit *network locality*.

### Exploiting Locality: Caches

Cache memories exploit temporal and spatial locality of reference to provide programmers with the performance characteristics of a large and fast memory system but at reasonable cost. The key is a hierarchical organization of the storage, known usually as the memory hierarchy and shown pictorially below. On modern processors, there may be many memory hierarchy levels such as the registers, Level 1 (L1) cache, Level 2 (L2) cache, Level 3 (L3) cache, and main memory. The virtual memory system can be considered to be another level in the memory hierarchy. L1 cache memory is small, fast, and usually implemented in SRAM, while main memory is large, relatively slow, and implemented in DRAM; the other levels of cache fall somewhere in between these two extremes. For example, on the Intel Nehalem multicore processor, each core has 32 KB L1 instruction and data caches, and a 256 KB L2 cache; all the cores share a 8 MB L3 cache. The size of the main memory may be many GB. On most processors, accesses to the L1 cache typically take around 1–3 cycles, and the latency of memory accesses increases by roughly an order of magnitude in going from one level to the next (Fig. 1).

![Memory hierarchy of a typical processor](image.png)

By providing a faster and smaller cache for frequently accessed data, caches reduce the effective latency of memory accesses. Given such a memory hierarchy, the effective latency of memory accesses as observed by the processor will be considerably less than DRAM access time provided most of the memory accesses are satisfied by the faster cache levels. One way to accomplish this is to store the addresses and contents of the most recently accessed memory locations in the cache memories. When a processor issues a memory request, the fastest cache level in which that address is cached responds to the request, so the long-latency trip to memory is required only if the address is not cached anywhere. Programs that have good temporal locality obviously benefit from this caching strategy. When a memory request goes all the way to main memory, most memory systems will fetch not just the contents of the desired address but the contents of a block of addresses, one of which is the desired address. Because the latency of fetching a cache block from memory is almost the same as the latency of fetching the contents of a single address, programs with spatial locality benefit from this caching policy.

The design of caches becomes more complex in parallel machines because of the *cache-coherence* problem. A major advantage of caches is that they are transparent to the programmer in the sense that although caches may confer performance benefits, they do not change the output of the program. Ensuring this transparency in a shared-memory multiprocessor is complicated by the fact that a memory location may be cached in the local caches of several cores/processors, so it is necessary to ensure that updates to that address made by different processors are coordinated in some way to preserve the transparency of caching to the programmer. This is referred to as the cache-coherence problem, and many solutions to this problem have been explored in the literature.

### Program Transformations for Exploiting Caches

While caches are transparent to the programmer, obtaining the performance benefits of caching may require the programmer to be aware of the cache hierarchy and to transform programs to become more “cache-friendly” by enhancing temporal and spatial locality. For the most part, these transformations can be divided...
into two classes: (1) rescheduling of operations and (2) reordering of data structures.

The goal of rescheduling operations is to ensure that operations that access the same memory address (or more generally, the same cache line) are executed more or less contemporaneously, thereby improving locality. In dense matrix computations, loop permutation and loop tiling are the most important transformations for enhancing locality. For example, consider the matrix multiplication program discussed above. As is well known, all six permutations of the three loops produce the same output; however, their locality characteristics are very different. If the \( j \) loop is permuted into the innermost position in the loop nest, the references to arrays \( C \) and \( B \) enjoy good spatial locality since the inner loop accesses these arrays row by row, and the reference to array \( A \) enjoys good temporal locality since all iterations of the inner loop for given outer loop indices access the same element of \( A \). Conversely, permuting the \( i \) loop into the innermost position is bad for exploiting spatial locality since none of the matrices will be accessed by row in the innermost loop.

While loop permutation is useful, the most important transformation for enhancing locality is loop tiling. A tiled version of matrix multiplication is shown below. The effect of this transformation is to produce block matrix programs, in which the inner loops can be viewed as operating on blocks of the original matrices. In the tiled code shown below, \( \text{BLOCK\_SIZE} \) is a parameter that determines the size of the blocks multiplied in the inner three loops. The value of this parameter must be chosen carefully so that the working set of the three innermost loops fits in cache. If there are multiple cache levels, it may be necessary to tile for some or all these levels.

Choosing an optimal tile size is a difficult problem in general, and it may be necessary to execute the program for different tile sizes and determine the best one by measurement. This approach is called empirical optimization and it is used by the ATLAS system, which generates high-performance Basic Linear Algebra Subroutines (BLAS). It is often possible to use simple machine models to reduce the search space.

```c
for (ii = 0; ii < N; ii += BLOCK_SIZE)
  for (kk = 0; kk < N; kk += BLOCK_SIZE)
    for (jj = 0; jj < N; jj += BLOCK_SIZE)
      for (i = ii; i < ii + BLOCK\_SIZE && i < N; i++)
        for (k = kk; k < kk + BLOCK\_SIZE && k < N; k++)
          for (j = jj; j < jj + BLOCK\_SIZE && j < N; j++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

Spatial locality can also be improved by changing the layout of data structures in memory. For example, in the \((i, j, k)\) order of the three nested loop version of matrix multiplication, changing the layout of \( B \) to column-major order will improve spatial locality for the references to array \( B \). For data transformations to be useful, the overhead of transforming the data layout must be amortized by the benefits of improved spatial locality in computing with that data. For example, when performing the tiled matrix multiplication shown above, high performance implementations will usually copy blocks of the matrices into contiguous memory locations because the overhead of copying the data is amortized by the benefits of improved spatial locality when performing the block matrix multiplication.

Many techniques for changing the layouts of other kinds of data structures have been explored in the literature. For example, records and structures that span multiple cache lines can be allocated so that fields that are often accessed contemporaneously are packed together into the same cache line, if possible, rather than being allocated in different cache lines. In managed languages like Java and C#, the garbage collector can improve locality by moving data items that are accessed contemporaneously into the same page of virtual memory.

**Cache-Oblivious Programs**

Divide-and-conquer algorithms for many problems are naturally cache-friendly because they repeatedly generate sub-problems of smaller size until they reach a base case, and then assemble the solution to the problem from the solutions to these sub-problems. If the base case of the divide-and-conquer algorithm is chosen so that the working set of that sub-problem fits in the cache, and the data movement required for assembling the solution from the solutions to the sub-problems is small, the program may have excellent locality. Divide-and-conquer algorithms for problems like matrix multiplication, FFT, and Cholesky factorization enjoy this
property and they are known as cache-oblivious programs because they have good temporal locality even though they are not explicitly optimized for any particular cache size or structure; this is in contrast to iterative algorithms for these problems, which require tiling to be made cache-friendly. Spatial locality in cache-oblivious matrix multiplication programs can be enhanced by using space-filling curves to order the storage of matrix elements.

Prefetching
Caches can be classified as latency-avoidance mechanisms since the goal is to avoid long latency memory operations by exploiting locality of reference. The performance impact of long latency memory operations can also be reduced by latency-tolerance techniques such as prefetching. If the memory accesses of a program can be predicted well in advance of when the values are actually required in the computation, the processor might be able to prefetch these values from memory into the cache so that they are available immediately when they are needed for the computation. This is an instance of a more general optimization technique in parallel programming called overlapping communication with computation: the key idea is to perform data movement in parallel with computation to reduce the effective overhead of the data movement. Prefetching is particularly useful for streaming programs since these programs touch large volumes of data and have predictable data accesses but have little temporal locality, so caches are not effective for reducing the effective memory latency.

In some processors, prefetching is under the control of the programmer. For example, the iA32 SSE instruction set has prefetch instructions for prefetching data into some or all cache levels. The programmer or compiler is responsible for inserting prefetch instructions into the appropriate places in the program. In other processors, prefetching is implemented in hardware. Usually, the prefetch unit monitors the stream of memory addresses from the processor and attempts to detect sequences of fixed stride memory accesses. If it finds such a sequence, it uses that sequence to determine memory addresses for prefetching. To avoid unnecessary page faults, prefetching usually does not cross page boundaries.

Exploiting Network Locality
For techniques to exploit network locality, see the Encyclopedia entry on "Topology Aware Task Mapping."

Using Parallelism to Tolerate Memory Latency
When programs have little locality of reference and also make unpredictable memory accesses, neither caching nor prefetching is effective in reducing the performance impact of long latency memory accesses. Many irregular programs such as graph computations fall in this category. For such programs, it is possible to tolerate latency by exploiting parallelism, provided the processor can switch rapidly between parallel threads of execution. There are many implementations of this scheme, but a simple approach is the following. The processor issues instructions from a thread until that thread becomes idle waiting for a memory load to complete. At that point, the processor switches to a different thread that is ready to execute and begins issuing instructions from that thread, and so on. As long as there is enough parallelism in the program and the processor can switch quickly between threads, the performance impact of long latency loads from memory is minimized. In other designs, the processor switches between ready threads at every cycle.

This approach requires specially designed processors since the processor needs to be able to switch between threads very rapidly. Tera’s MTA processor is an example. Each processor can support 128 simultaneous threads. Simultaneous multi-threading (SMT) is an implementation of this idea in general-purpose processors. The same principle also underlies the design of dataflow computers, although threads in that case are very fine-grain, comprising of a single instruction.

Related Entries
► Numerical Linear Algebra
► Code Generation
► Loop Nest Parallelization
► Nonuniform Memory Access (NUMA) Machines
► Parallelization, Automatic
► Parallelization, Basic Block
► Scheduling Algorithms
► Topology Aware Task Mapping
► Unimodular Transformations
Bibliographic Notes and Further Reading

Although the importance of exploiting locality of reference in programs was recognized from the very start of computing by pioneers like von Neumann, the first machine to incorporate cache memory was the IBM 360 Model 85 [8]. Work on transforming programs to optimize them for parallelism and locality of reference began at the University of Illinois, Urbana-Champaign in the 1960s [2, 12]. The automation of loop and data transformations was enabled by the development of powerful integer linear programming tools in the 1990s [5, 6, 13]. For the most part, these techniques are restricted to dense matrix programs. Data-shackling is a more general data-centric approach to reasoning about locality [11]. Data transformations for non-array data structures are described in [4].

Determining optimal values for parameters such as tile sizes is an active topic of research. The empirical optimization approach is incorporated in the popular ATLAS system, which produces automatically tuned Basic Linear Algebra Subroutines (BLAS) [1]. An approach using simple performance models and a detailed comparison with the empirical optimization approach can be found in [14]. Cache-oblivious algorithms [7] are based on the notion of I/O complexity [9].

Latency tolerance through multithreading was implemented in the Tera computer [3]. Network locality and network-oblivious algorithms are explored in [4].

Bibliography


Lock-Free Algorithms

» Non-Blocking Algorithms

Locks

» Synchronization

» Transactional Memory

Logarithmic-Depth Sorting Network

» AKS Network
Logic Languages

Manuel Carro\textsuperscript{1}, Manuel Hermenegildo\textsuperscript{1,2}
\textsuperscript{1}Universidad Politécnica de Madrid, Madrid, Spain
\textsuperscript{2}IMDEA Software Institute, Madrid, Spain

Synonyms
Concurrent logic languages; Distributed logic languages; Prolog

Definition
The common name of Parallel Logic Languages groups those languages which are based on logic programming and which, while respecting as much as possible the declarative semantics, have an operational semantics which exploits parallelism or concurrency, either explicitly or implicitly, to gain in efficiency or expressiveness.

Discussion

Logic Programming
The application of logic and mechanized proofs to express problems and their solutions is at the origins of computer science \cite{6}. The basis of this approach is to express the knowledge on some problem (e.g., how a sorted tree is organized) as a consistent theory in some logic and to model the desired goal (e.g., storing an item on the tree) as a formula. If the formula is true in the theory capturing the problem conditions, then the objective expressed by such formula is achievable (i.e., the item can be stored). The set of formulas which are true in the theory are those which state the set of objectives achievable (computable) by the program. This constitutes the declarative semantics of logic programs.

A usual proof strategy is to add to the initial theory a formula stating the negation of the desired goal and to prove the inconsistency of the resulting theory by generating a counterexample. The valuation(s) of the free variable(s) in the counterexample constitute a solution to the initial problem.

As an example, Fig. 1 presents a set $\mathcal{T}$ of formulas describing insertion in a sorted tree, where constants, predicate names, and term names are written in lowercase, and variables start with uppercase letters, in conformance with logic programming languages. Arithmetic predicates are assumed to be available. $\text{void}$ represents the empty tree. An argument of the form tree($I$, $L$, $R$) represents a tree node with item $I$ and left and right children $L$ and $R$, respectively. A fact insert($T_1$, $I$, $T_2$) can be inferred from this theory if $T_2$ is the resulting tree after inserting item $I$ in tree $T_1$. Given the term $T_1$ representing a sorted tree and an item $I$ to be stored, the inconsistency of $T \land \neg \exists T_2. \text{insert}(T_1, I, T_2)$ can be proved with a counterexample for $T_2$ which will be, precisely, the tree which results from inserting $I$ into $T_1$.

From the procedural point of view, the method used to generate a counterexample is basically a search, where at each step one of the inference rules in the logic is applied, maybe leaving other rules pending to be applied. Making this search as efficient as possible is paramount in order to make logic a practical programming tool. This can be achieved by reducing the set of inference rules to be applied and the number of places where they can be applied.

Concrete logic programming languages specify the deduction method (operational semantics) further as well as the syntax and (possibly) subsets of the logic. The most successful combination is resolution (only one inference rule) on Horn clauses (conjunctions of non-negated literals implying a non-negated literal, as in Fig. 1). In particular, the usual operational semantics (SLD resolution) for the logic programming language Prolog uses the resolution principle with two rules to decide which literals the inference rule is applied to. SLD resolution in Prolog:

- Explores conjunctions in every implication from left to right, as written in the program text (computation rule).
- Tries to match and evaluate clauses from top to bottom (search rule).

From a goal like insert(tree(void, 3, void), 4, $T_2$), SLD resolution selects one of the implications with a matching head, renames apart its parameters (i.e., variables with new names which do not clash with previous ones are created), and expands the right-hand side. The
Logic Languages. Fig. 1 Insertion in a sorted tree

first matching clause is (2), and expanding its body results in:

\[ T_2 = \text{tree}(\text{II}, 3, \text{void}) \land 4 < 3 \land \text{insert}(\text{void}, 4, \text{II}) \]

This conjunction of goals is called a resolvent. The leftmost one is a unification \([12]\) (i.e., an equation which expresses syntactical equality between terms) which, when successful, generates bindings for the variables in it. In this case, as \( T_2 \) is a variable, it just succeeds by assigning the term \( \text{tree}(\text{II}, 3, \text{void}) \) to it. That leaves the resolvent:

\[ 4 < 3 \land \text{insert}(\text{void}, 4, \text{II}) \]

The next goal to be solved is \( 4 < 3 \), which is false, and therefore the whole conjunction is false and another alternative for the last selection is to be taken. Bindings made since the last selection are undone, and clause (3) is selected:

\[ T_2 = \text{tree}(\text{void}, 3, \text{Ir}) \land 3 < 4 \land \text{insert}(\text{void}, 4, \text{Ir}) \]

In this case, the arithmetic predicate succeeds and the resolvent is reduced to \( \text{insert}(\text{void}, 4, \text{Ir}) \). This matches the head of the first clause and succeeds with the binding \( \text{Ir} = \text{tree}(\text{void}, 4, \text{void}) \). The initial goal is then proved with the output binding:

\[ T_2 = \text{tree}(\text{void}, 3, \text{tree}(\text{void}, 4, \text{void})) \]

which represents a tree with a 3 at its root and a 4 in its (unique) right subtree.

Logic programming languages include additional nonlogical constructs to perform I/O, guide the execution, and provide other facilities. They also offer a compact, programming-oriented syntax (e.g., avoiding as much as possible quantifiers; compare the rest of the programs in this text with the mathematical representation of Fig. 1).

Operational View and Comparison with Other Programming Paradigms

Due to the fixed computation and search rules, SLD resolution can be viewed as a relatively simple operational semantics that has significant similarities to that of other programming paradigms. This provides a highly operational view of resolution-based theorem proving.

In fact, in the procedural view, SLD is akin to a traditional execution that traverses clauses sequentially from left to right, in the same way execution traverses procedure bodies in traditional languages. Predicates take the role of procedures, clause head arguments are procedure arguments, clauses are similar to case statements, etc. Predicate invocation takes the role of procedure calling and unification amounts to parameter passing (both for input and output). Logical variables are similar to (declarative) pointers that can be used to pass data or pointers inside data structures. Thus, pure deterministic logic programs can be seen as traditional programs with pointers and dynamic memory allocation but without destructive update. In addition, in non-deterministic programs clause selection offers a built-in backtracking-based search mechanism not present in other paradigms, which allows exploring the possible ways to accomplish a goal. Finally, unification provides a rich, bidirectional pattern matching-style parameter passing and data access mechanism. Similarly, logic programming (specially variants that support higher order) can be seen as a generalization of functional programming where pattern matching is bidirectional, logic variables are allowed, and more than one definition is possible for each function. See, for example, [8] for a more detailed discussion of the correspondence between logic programming and other programming paradigms.
Toward Parallelism and Concurrency

Parallel logic programming languages aim at retaining the observable semantics while speeding up the execution w.r.t. a sequential implementation. Concurrent logic languages (► Sect. Concurrency in Logic Programming) focus more on expressiveness and often adopt an alternative semantics. Opportunities for parallel and concurrent execution come from relaxing the left-to-right, top-to-bottom order of the computation and search rules: it is, in principle, possible to select several different literals in a logical conjunction to resolve against simultaneously (and-parallelism), or explore several different clauses at the same time (or-parallelism) [4, 7].

The distinction between parallel and concurrent logic languages is not as clear-cut as implied so far. In fact, parallel execution models have been proposed for concurrent logic languages in which independent parts of the execution of concurrent processes are executed in different processors simultaneously. Conversely, the language constructs developed for parallelism have also been used for concurrency and distributed execution. The current trend is toward designing languages that combine in useful ways parallelism, concurrency, and search capabilities.

Most parallel logic programming languages have been implemented for shared-memory multicomputers. There is no a priori reason not to execute a logic program in a distributed fashion, as demonstrated by distributed implementations [11]. However, the need to maintain a consistent binding store seen by all the agents involved in an execution imposes an overhead which does not appear in the shared memory case. The focus will, therefore, be on linguistic constructions which, although not necessarily requiring shared-memory computers, have been mainly exploited in this kind of architectures.

Exploiting Parallelism in Logic Programming

Independently from whether parallelism is exploited among conjunctions of literals or among different clauses, it can be detected in several ways:

Implicit parallelism: The compiler and run-time system decide what can be executed in parallel with no programmer intervention, analyzing, for example, data dependencies. One disadvantage of this approach is that it may impose considerable run-time overhead, and the programmer cannot give clues as to what can (or should) be executed in parallel.

Explicit parallelism: The compiler and run-time system know beforehand which clauses/goals are to be executed in parallel typically via programmer annotations. This has the disadvantage that it puts the responsibility of parallelization (a hard task) solely in the hands of the programmer.

Combinations: This is realized in practice as a combination of handwritten annotations, static analysis, and run-time checks.

The last option is arguably the one with the most potential, as it aims at both reducing run-time effort and simplifying the task of the programmer, while still allowing annotating programs for parallel execution. In the following sections, variants of these sources of parallelism will be revised briefly, assuming that annotations to express such parallelism at the source language level are available and the programs have been annotated with them. Methods for introducing these annotations automatically (briefly mentioned in ► Sect. Automatic Detection of Parallelism) are the subject of another entry (► Parallelization, Automatic).

And-Parallelism

And-parallelism stems from simultaneously selecting several literals in a resolvent (i.e., several steps in a procedure body). While the semantics of logic programming makes it possible to safely select any goal from the resolvent, practical issues make it necessary to carefully choose which goals can be executed in parallel. For example, the evaluation of arithmetic expressions in classical Prolog needs variables to be adequately instantiated, and therefore the sequential execution order should be kept in some fragments; input/output also needs to follow a fixed execution order; and tasks which are too small may not benefit from parallel execution [8].

Whether a sequential execution order has to be respected (either for efficiency or correctness) can be decided based on goal independence, a general notion which captures whether two goals can proceed without mutual interference. Deciding whether the amount of
work inside a goal is enough to benefit from parallel execution can be done with the use of (statically inferred) cost (complexity) functions. With them, the expected sequential and parallel execution time can be compared at compile time or at run time (but before executing the parallel goals).

An Illustrating Example

The code in Fig. 2 implements a parallel matrix by matrix multiplication. Commas denote sequential conjunctions (i.e., A, B means “execute goal A first, and when successfully finished, execute goal B”), while goals separated by an ampersand (A & B) can be safely executed in parallel.

Matrices are represented using nested lists, and matrix multiplication is decomposed into loops performing vector by matrix and vector by vector multiplication, which are in turn expressed by means of recursion. Every iteration step can be performed independently of the others in the same loop and therefore parallelism can be expressed by stating that predicate calls in the body of the recursive clauses (e.g., mat_vec_multiply in mat_mat_multiply) can be executed in parallel with the corresponding recursive call (mat_mat_multiply). Note that the arithmetic operation R is Prod + NewRes is performed after executing the other goals in the clause, because it needs NewRes to be computed.

\[
\begin{align*}
\text{mat_mat_multiply}([],_&,[]). \\
\text{mat_mat_multiply}([V0|\text{Rest}], V1,[R|\text{Os}]):- \\
\quad \text{mat_vec_multiply}(V1,V0,R) & \\
\quad \text{mat_mat_multiply}(\text{Rest}, V1, \text{Os}).
\end{align*}
\]

\[
\begin{align*}
\text{mat_vec_multiply}([],_&,[]). \\
\text{mat_vec_multiply}([V0|\text{Rest}], V1,[R|\text{Os}]):- \\
\quad \text{vec_vec_multiply}(V0,V1,R) & \\
\quad \text{mat_vec_multiply}(\text{Rest}, V1, \text{Os}).
\end{align*}
\]

\[
\begin{align*}
\text{vec_vec_multiply}([],[],0). \\
\text{vec_vec_multiply}([H1|T1],[H2|T2], R):- \\
\quad \text{Prod is } H1*H2 & \\
\quad \text{vec_vec_multiply}(T1,T2, \text{NewRes}), \\
\quad R \text{ is } \text{Prod} + \text{NewRes}.
\end{align*}
\]

The \& construct is similar to a FORK-JOIN, where the existence of A & B implicitly marks a FORK for the goals A and B and a JOIN after these two goals have finished their execution. Such parallel execution can be nested to arbitrary depths. The strict fork-join structure is also not compulsory and other, more flexible parallelism primitives are also available.

A relevant difference with procedural languages is the possible presence of search, that is, a goal can yield several solutions on backtracking, corresponding to the selection of different clauses, or even fail without any solution. Parallel execution should cope with this: when A & B can yield several solutions, an adequate operational semantics (and underlying execution mechanism) has to be adopted. Several approaches exist depending on the relationship between A and B. Perhaps, the simplest one is to execute in parallel and in separate environments A and B, gather all the solutions, and then build the cross product of the solutions. However, even if A and B can be safely executed in parallel, this approach is often not satisfactory for performance reasons.

Independent And-Parallelism

Independent and-parallelism (IAP) only allows the parallel execution of goals which do not interfere through, for example, bindings to variables, input/output, assertions onto the shared database, etc. In absence of side effects, independence is customarily expressed in terms of allowing only compatible accesses to shared variables.

Independent and-parallelism can be strict or non-strict. The strict variant (SIAP) only allows parallel execution between goals which do not share variables at run time. The non-strict variant (NSIAP) relaxes this requirement to allow parallel execution of goals sharing variables as long as at most one of the parallel goals attempts to bind/check them or they are bound to compatible values. The matrix multiplication (Fig. 2) features SIAP if it is assumed that the input data (the first two matrices) does not contain any variable at all (i.e., these matrices are ground) and therefore independence is ensured as no variables can be shared.

As a further example, Fig. 3, left, is an implementation of the QuickSort algorithm, where it is assumed that a ground list of elements is given in order to be sorted. In this case, partition splits the input list into two lists which are recursively sorted and joined.
qsort([], []). qsort([X|Xs], Sorted):-
  partition(Xs, X, Large, Small),
  qsort(Small, SmallSorted),
  qsort(Large, LargeSorted),
  append(Small, [X|Large], Sorted).

% Partition is common to all
% QuickSort versions
partition([], _P, [], []). partition([X|Xs], P, [X|Ls], Ss):-
  X > P, partition(Xs, P, Ls, Ss).
partition([X|Xs], P, Ls, [X|Ss]):-
  X =< P, partition(Xs, P, Ls, Ss).

Logic Languages. Fig. 3 QuickSort: sequential version (left) and SIAP version (right)

qsort(Unsrt, Sorted):-
  qs(Unsrt, Sorted, []).
qsort([], S, S).
qsort([X|Xs], SortedSoFar, Tail):-
  partition(Xs, X, Large, Small),
  qsort(Small, SortedSoFar, Rest),
  qsort(Large, Rest, Tail).
qsort(Unsrt, Sorted):-
  qs(Unsrt, Sorted, []).
qsort([], S, S).
qsort([X|Xs], SortedSoFar, Tail):-
  partition(Xs, X, Large, Small),
  qsort(Small, SortedSoFar, Rest) &
  qsort(Large, Rest, Tail).

Logic Languages. Fig. 4 QuickSort, using difference lists and annotated for non-strict and-parallelism

afterward. The code on the right is the SIAP parallelization, correct under these assumptions, where the two calls to QuickSort are scheduled for parallel execution.

A perhaps more interesting example is the implementation of QuickSort in Fig. 4. The sequential version on the left uses a technique called difference lists. The sorted list is, in this case, not ended with a nil, but with a free variable (a “logical pointer”) which is carried around. When this variable is instantiated, the tail of the difference list is instantiated. This makes it possible to append two lists in constant time. A procedural counterpart would carry around a pointer to the last element of a list, so that it can be modified directly. The two calls to Qsort share a variable and cannot be executed in parallel following the SIAP scheme. However, since only the second call will attempt to bind it they can be scheduled for NSIAP execution.

A last example of IAP (this time including some search) appears in Fig. 5. It is a solution to the problem of placing N queens on an \( N \times N \) chessboard so that no two queens attack each other. It places the queens column by column picking (with select/3) a position from a list of candidates. not_attack/3 checks that this candidate does not attack already placed queens. Search is triggered by the multiple solutions of select/3, and the program can return all the solutions to the queens problem via backtracking.

Checking that queens are attacked is independent from trying to place the rest of the queens, and so the two goals can be run in parallel, performing some speculative work. Unlike the previous examples, the not_attack/3 check can fail and force the failure of a series of parallel goals which should be immediately stopped.

Applying the execution scheme sketched in Sect. An Illustrating Example (collecting solutions and making a cross product) is not appropriate: if a candidate queen cannot be placed on the chessboard, the work invested in trying to build the rest of the solutions is
queens(N, Qs) :-
  range(1, N, Ns),
  queens(Ns, [], Qs).
queens([], Qs, Qs).
queens(UnplacedQs, SafeQs, Qs) :-
  select(UnplacedQs, UnplacedQs1, Q),
  not_attack(SafeQs, Q, 1) &
  queens(UnplacedQs1, [Q|SafeQs], Qs).
not_attack([], _, _).
not_attack([Y|Ys], X, N) :-
  X =\= Y+N, X =\= Y-N,
  N1 is N+1,
  not_attack(Ys, X, N1).
select([X|Xs], Xs, X).
select([Y|Zs], [Y|Zs], X).

Logic Languages. Fig. 5  N-Queens annotated for and-parallelism

wasted. If only one solution is needed and the candidate queen is correctly placed, the work spent in finding all the solutions is wasted.

One workaround is to adapt the sequential operational semantics to the parallel case and perform recomputation. Assuming the conjunction $A \land B \land C$, forward execution is done as in a FORK-JOIN scheme, and backtracking is performed from right to left. New solutions are first sought for $C$ and combined with the existing ones for $A$ and $B$. When the solutions for $C$ are exhausted, $B$ is restarted to look for another solution while $C$ is relaunched from the beginning. When the goal $A$ finally fails, the whole parallel conjunction fails. This generates the solutions in the same order as the sequential execution, which may be interesting for some applications.

Additionally, in the case of IAP, if one of the goals (say, $B$) fails without producing any solution, the parallel conjunction can immediately fail. This is correct because $B$ does not see bindings from $A$, and therefore no solution for $A$ can cause $B$ not to fail. This semi-intelligent backtracking (combined with some constraints on scheduling) guarantees the no-slowdown property: an IAP execution will never take longer than a sequential one (modulo overheads due to scheduling and goal launching).

**Dependent And-Parallelism**

IAP’s independence restriction has its roots in technical difficulties: a correct, efficient implementation of a language with logical variables and backtracking and in which multiple processing units are concurrently trying to bind these variables is challenging – that is the scenario in Dependent And-Parallelism (DAP). On one hand, every access to (potentially) shared variables and their associated internal data structures has to be protected. Additionally, when two processes disagree on the value given to a shared variable, one of them has to backtrack to generate another binding. Parallel goals sharing the variable whose binding is to be undone may have to backtrack if the binding was used to select which clause to take.

However, since DAP subsumes IAP, it offers a priori more opportunities for parallelism: in any of the QuickSort examples, the lists generated by partition can be directly fed to the qsort goals, which could run in parallel between them and with the partition predicate. When a new item is generated, the corresponding qsort call will wake up and advance the sorting process.

If the goals to be executed in parallel do not bind shared variables differently (i.e., they collaborate rather than compete), the problem of ensuring binding coherence disappears (it is, in fact, NSIAP). Moreover, if they are deterministic, there is no need to handle backtracking, and the parallel goals act as if they were producers and consumers. This is called (deterministic) stream and-parallelism, and can be (efficiently) exploited. The QuickSort program is an example when it is called with a free variable as second argument, because this ensures that sorting will not fail and will produce a single solution. On the other hand, an initial goal such as ?- qsort([1,2], [2,1]) will fail, and cannot be executed using stream parallelism.

Despite its complexity, proposals which fully deal with DAP exist. One approach to avoid considering all variables as potentially shared is to mark possibly shared variables at compile time, so that only these are tested at
run time [16] (note that in the example above X and Y are aliased after the execution of q):

\[
p(X, Y) :-
q(X, Y),
\text{dep}([X, Y]) \Rightarrow r(X) \& s(Y) \& t(Y).
\]

\[
q(A, A).
\]

Shared variables are given a special representation to, for example, make it possible to lock them. In order to control conflicts in the assignment to variables, goals are dynamically classified as producers or consumers of variables as follows: for a given variable X, the leftmost goal which has access to X is its producer, and can bind it. The rest of the goals are consumers of the variable, and they suspend should they try to bind the variable. In the previous code, r would initially be the producer of X (which is aliased to Y by q/2) and s and t its consumers. When r finishes, s becomes the producer of Y. The overhead of handling this special variable representation is large and impacts sequential execution: experiments show a reduction of 50% in sequential speed.

Goal-Level and Binding-Level Independence
While IAP and DAP differ in the kind of problems they attack and the techniques needed to implement them, there are unifying concepts behind them: both IAP and DAP need, at some level, operations which are executed atomically and without interference. In the case of DAP, binding a shared variable and testing that this binding is consistent with those made by other processes is an atomic operation. Stream and-parallelism avoids testing for consistency by requiring that bindings by different processes do not interfere; thereby it exhibits independence at the level of individual bindings, while in IAP noninterference appears at the level of complete goals. The requirement that variables are not concurrently written to makes variable locking unneeded for IAP. Other proposals (Sec. Eventual and Atomic Tells) have a notion of independence at the level of groups of unifications explicitly marked in the program source.

Or-Parallelism
Or-Parallelism originates from executing in parallel the different clauses which can be used to solve a goal in the resolvent. Multiple clauses, as well as the continuation of the calling goal, can thus be simultaneously tried. Or-parallelism allows searching for solutions to a query faster, by exploring in parallel the search space generated by the program. For example, in the queens program of Fig. 6 the two clauses of the select/3 predicate can proceed in parallel as well as the calls to not_attack/3 and queens/3 which correspond to every solution of select/3: branches continue in parallel after the predicate which forked the execution has finished and into the continuations of the clauses calling such predicate. This means that each possible selection of a queen will be explored in parallel.

In principle, all the branches of the search tree of a program are independent and thus or-parallelism can apparently always be exploited. In practice, dependencies appear due to side effects and solution order-dependent control structures (such as the “cut”), and must be taken into account. Control constructs that are more appropriate for an or-parallel context have also

:- parallel select/3.
queens(N, Qs) :-
range(1, N, Ns),
queens(Ns, [], Qs).
queens([], Qs, Qs).
queens(UnplacedQs, SafeQs, Qs) :-
select(UnplacedQs, UnplacedQs1, Q),
not_attack(SafeQs, Q, 1),
queens(UnplacedQs1, [Q|SafeQs], Qs).

not_attack([], _, _).
not_attack([Y|Ys], X, N) :-
X =\= Y+N, X =\= Y-N,
N1 is N+1,
not_attack(Ys, X, N1).
select([X|Xs], Xs, X).
select([Y|Ys], [Y|Zs], X) :-
select(Ys, Zs, X).
been proposed. Much work has also been devoted to developing efficient task scheduling algorithms and, in general, or-parallel Prolog systems [1,7,13].

Or-parallelism frequently arises in applications that explore a large search space via backtracking such as in expert systems, optimization and relaxation problems, certain types of parsing, natural language processing, scheduling, or in deductive database systems.

Data Parallelism and Unification Parallelism
Most work in parallel logic languages assumed an underlying MIMD execution model of parallel programs which were initially written for sequential execution. However, several approaches to exploiting SIMD parallelism exist.

And-data-parallelism, as in [2], tries to detect loops which apply a given operation to all the elements of a data structure in order to unroll them and apply the operation in parallel to every data item in the structure. In order to access quickly the different elements in the data structure, parallel unification algorithms were developed. This kind of data parallelism can be seen as a highly specialized version of independent and parallelism [9]. Or-data-parallelism, as in [17], changes the standard head unification to keep several environments simultaneously. Unifications of several heads can then be executed in parallel.

Combinations of Or- and And-Parallelism
Combining or- and and-parallelism is possible, and in principle it should be able to exploit more opportunities for parallel execution. As an example, Fig. 7 shows the Queens program annotated to exploit and+or-parallelism, by combining the annotations in Figs. 5 and 6. The efficient implementation of and+or-parallelism is difficult due to the antagonistic requirements of both cases: while and-parallelism needs variables shared between parallel goals to be accessible by independent threads, or-parallelism needs shared variables to have distinct instantiations in different threads. Restricted versions of and+or-parallelism have been implemented: for example, in [5], teams of processors execute deterministic dependent and-parallel goals. When a nondeterministic goal is to be executed, the team moves to a phase which executes the goal using or-parallelism. Every branch of the goal is deterministic, from the point of view of the agent executing it, so deterministic dependent and-parallelism can be used again.

Automatic Detection of Parallelism
A number of parallelizing compilers have been developed for (constraint) logic programming systems [3,10,14]. They exemplify a combined approach to exploiting parallelism (Sect. Exploiting Parallelism in Logic Programming): the compiler aims at automatically uncovering parallelism in the program by producing a version annotated with parallel operators, which is then passed to another stage of the compiler to produce lower-level parallel code. At the same time, programmers can introduce parallel annotations in the source program that the parallelizer will respect (and check for correctness, possibly introducing run-time checks for independence). The final parallelized version could always have been directly written by hand, but of course the possibility of being able to automatically parallelize (parts of)

:- parallel select/3.
queens(N, Qs) :-
  range(1, N, Ns),
  queens(Ns, [ ], Qs).
queens([ ], Qs, Qs).
queens(UnplacedQs, SafeQs, Qs) :-
  select(UnplacedQs, UnplacedQs1, Q),
  not_attack(SafeQs, Q, 1) &
  queens(UnplacedQs1, [Q|SafeQs], Qs).

not_attack([], _, _).
not_attack([Y|Ys], X, N) :-
  X =\= Y+N, X =\= Y-N,
  N1 is N+1,
  not_attack(Ys, X, N1).
select([X|Xs], Xs, X).
select([Y|Ys], [Y|Zs], X) :-
  select(Ys, Zs, X).

Logic Languages. Fig. 7 N-Queens annotated for and- and or-parallelism
programs initially written for sequential execution is very appealing. The parallelization can leave some independence checks for run time, also combining static and dynamic techniques. Experimental results have shown that this approach can uncover significant amounts of parallelism and achieve good speedups in practice [3, 14] with minimal human intervention. See the entry Parallelization, Automatic for more details on this topic.

Extensions to Constraint Logic Programming
Constraint logic programming extends logic programming with constraint solving capabilities by generalizing unification while preserving the basic syntax and resolution-based semantics of traditional logic programming. These extensions open a large class of new applications and also bring improved semantics for certain aspects of traditional logic programming, such as fully relational support for arithmetic operations. Constraint logic programs also explore a search space and Or-parallelism is directly applicable in this context. Parallelism can also be exploited within the constraint solving process itself in a similar way to unification parallelism. Finally, and-parallelism can also be exploited, and the constraints point of view brings new, interesting generalizations of the concepts of independence.

Concurrency in Logic Programming
Concurrent logic programming languages [15, 18] do not aim primarily at improving execution speed but rather at increasing language expressiveness. As a consequence, their concurrency capabilities remain untouched in single-processor implementations. A strong point of these languages is the capability to express reactive systems. This affects their view of non-determinism: sequential logic programs adopt don’t know non-determinism, where predicate clauses express different ways to achieve a goal, but it is not known which one of these clauses leads to a solution for a given problem instance before their execution successfully finishes. Unsuccessful branches can be discarded (i.e., backtracked over) at any moment and their intermediate results are undone, except for side effects such as input/output or changes to the internal database. These computations are not reactive, as their only “real” result is the substitutions produced after success.

Don’t care non-determinism reflects the fact that, among the clauses that are eligible to generate a solution, any of them will do the job, and it does not matter which one is selected. Once a clause is committed to, the computation it performs is visible by the rest of the processes collaborating in the computation – hence the reactivity of the language – and backtracking cannot make this effect disappear.

Syntax and (Intuitive) Semantics
A general, simplified syntax which is adopted in several proposals for concurrent logic languages will be used here. A concurrent logic program is a collection of clauses of the form

\[ \text{Head} : \text{Ask} : \text{Tell} | \text{Body}. \]

where variables have been abstracted. Their meaning is:

- **Head** is a literal whose formal arguments are distinct variables.
- **Ask**, the clause guard, is a conjunction of constraints (equality, disequality, simple tests on variables, and arithmetic predicates in the simplest case). These constraints are used to check the state of the head variables and do not introduce new bindings. If the **Ask** constraints succeed, the clause can be selected. Otherwise, the clause suspends until it becomes selectable or until another clause of the same predicate is selected to be executed (only one clause of a predicate can be selected to execute a given goal). If all clauses for all pending goals are suspended, the computation is said to deadlock.
- **Tell** is a conjunction of constraints which, upon clause selection, are to be executed to create new bindings, which are seen inside and outside the clause.
- **Body** is a conjunction of goals which can be true (in which case it can be omitted) to express the end of a computation, a single literal to express the transformation of a process into another process, or a conjunction of literals to denote spawning of a series of processes interconnected by means of shared variables.

Fig. 8 (taken from [15]), left, shows an example. A nondeterministic merger accepts inputs from two lists of elements and generates an output list taking elements from both lists while respecting the relative order in them. Clauses are selected first depending on the
merge(In1, In2, Out):-
    In1 = [X|Ins] : Out = [X|Rest] |
    merge(Ins, In2, Rest).
merge(In1, In2, Out):-
    In2 = [X|Ins] : Out = [X|Rest] |
    merge(In1, Ins, Rest).
merge(In1, In2, Out):-
    In1 = [] : Out = In2 | true.
merge(In1, In2, Out):-
    In2 = [] : Out = In1 | true.

use_merge(L):-
    skip_starting(0, 2, Evens),
    skip_starting(1, 2, Odds),
    merge(Evens, Odds, L).

skip_starting(Base, Skip, List):-
    List = [Base|Rest] |
    NextBase is Base + Skip,
    skip_starting(NextBase, Skip, Rest).

Logic Languages. Fig. 8 Stream merger (left) and how to use it (right)

availability of elements in the input lists: when there is an element available in some input list (e.g., In1 = [X|Ins]), it is used to construct stepwisethe output list (Out = [X|Rest]), and the process is recursively called with new arguments.

The use of such a merger is shown in Fig. 8, right. Predicate use_merge is expected to return in L the list of natural numbers. Three processes are spawned: two of them generate a list of even numbers starting at 0 (resp. odd and starting at 1) and the third one merges these lists. The shared variables Evens and Odds act as communicating channels which are read by merge as they are instantiated.

A note on syntax: Other concurrent languages have adopted a different syntax to distinguish Tell from Ask constraints and, in some cases, to specify input and output modes. For example, the original syntax by Ueda [19] uses head unification for Ask constraints, and the first clause of merge would therefore be:

merge([X|Ins], In2, Out):-
    Out = [X|Rest] |
    merge(Ins, In2, Rest).

Other languages (e.g., Parlog) include mode declarations to specify which formal arguments are going to be read from or written to, and require the programmer to provide them to the compiler.

Another example of the expressiveness of concurrent logic languages (which is in some sense related to that of lazy functional languages) appears in Fig. 9, which shows a solution [15] to the generation of the Hamming sequence: the ordered stream of numbers of the form $2^i3^j5^k$, where $i,j,k$ are natural numbers, without repetition. The standard procedural solution keeps a pool of candidates to be the next number and outputs the smallest one when it is sure that no smallest candidate can be produced.

Fig. 9 assumes that head unifications are Ask guards. The concurrent logic solution starts with a seed for $i = j = k = 0$ and spawns three multiply processes which generate, from an existing partial stream, streams $X2$, $X3$, and $X5$ with a last element depending on the previous last. These streams are merged, filtering duplicates, by two ord_merge predicates. The result of the last ord_merge predicate is fed back to the three multiply processes which can take then a step to generate the next candidate item in the answer stream every time $Xs$ is further instantiated.

Variations on Semantics

There are some variability points on the semantics of concurrent logic languages which, in addition to slightly different syntax, cause large differences among the languages. These semantic differences also have an important impact on the associated implementation techniques. It is remarkable that the history of concurrent logic languages was marked by a “simplification” of the semantics, which brought about less expressiveness of the language (such as, for example, eliminating the possibility of generating multiple solutions) and, accordingly, less involved implementations [18]. This has been partly attributed to such very expressive mechanisms not being needed for the tasks these languages were mainly used for, and in part to the complication involved in their implementation. Other research aimed at including concurrency without giving up multiple solutions includes some concurrent constraint languages such as Oz and AKL, as well as more traditional
logic programming systems which include extensions for distributed and concurrent execution (e.g., Ciao, SWI, or Yap).

**Deep and Flat Guards**

Flat guard languages restrict the Ask guards to simple primitive operations (i.e., unifications, arithmetic predicates, etc.). Deep guards, on the other hand, can invoke arbitrary user predicates. Any bindings generated by deep guards have to be isolated from the rest of the environment (e.g., the caller) in order not to influence other guards which may be suspended or under evaluation.

Deep guards are more expressive than flat guards but the implementation techniques are much more complex. The technology necessary is related to that of the implementation of or-parallel systems, as bindings to variables in deep guards need to be kept in separate environments, just as in the or-parallel execution of logic programs.

**Fairness**

The question of whether processes which can progress will effectively do so is decomposed into two different concepts, or-fairness and and-fairness, respectively, related to which clause (among those satisfying their guards) and process (among the runnable ones) is selected to execute.

Or-fairness can be exemplified with the merger in Fig. 8. For infinite stream producers, it does not guarantee that items from any of the input streams will eventually be written onto its output stream. Whether this happens will depend on the concrete semantics under which it is being executed. If this is guaranteed to eventually happen, then the language has or-fairness.

And-fairness concerns the selection of which process can progress among the runnable ones. In Fig. 8, right, the two instances of skip_starting do not need to suspend waiting for any condition. Therefore, it would be possible that one of them runs without interruption, generating a stream of numbers which are consumed by merge, which would forward them to its output list. Had ord_merge been used instead of merge, the stream producers would alternatively suspend and resume.

**Eventual and Atomic Tells**

Tell guards can be executed at once before the goals in the body, or be separately scheduled. In the former case, Tells are said to be atomic because all the bindings in the Tell are generated at once. In the latter case, they are said to be eventual, as they are handled according to the and-fairness rules of the language. Atomic Tell makes it possible to write some algorithms more elegantly than eventual Tell. However, eventual Tells make it possible to treat unifications and constraints more homogeneously, and are more adequate to reason on the actual properties of distributed implementations of concurrent logic languages.
Related Entries

- Functional Languages
- Parallelization, Automatic
- Prolog Machines

Bibliographic Notes and Further Reading

Pointers to a brief selection of the related literature have been provided within the text and are included in the references. For a much more detailed coverage of this very large topic the reader is referred to the comprehensive surveys in [7, 15, 18].

Bibliography


LogP Bandwidth-Latency Model

- Bandwidth-Latency Models (BSP, LogP)

Loop Blocking

- Tiling

Loop Nest Parallelization

Utpal Banerjee
University of California at Irvine, Irvine, CA, USA

Synonyms
Parallelization

Definition

Loop Nest Parallelization refers to the problem of finding parallelism in a perfect nest of sequential loops. It typically deals with transformations that change the execution order of iterations of such a nest L by creating a different nest L', such that L' is equivalent to L and some of its loops can run in parallel.
Discussion

Introduction

For a better understanding of the current essay, the reader should first go through the essay Unimodular Transformations in this encyclopedia.

The program model is a perfect nest \( L \) of \( m \) sequential loops. An iteration of \( L \) is an instance of the body of \( L \). The program consists of a certain set of iterations that are to be executed in a certain sequential order. This execution order imposes a dependence structure on the set of iterations, based on how they access different memory locations. A loop in \( L \) carries a dependence if the dependence between two iterations is due to the unfolding of that loop. A loop can run in parallel if it carries no dependence.

In this essay, one considers transformations that change \( L \) into a perfect nest \( L' \) with the same set of iterations executing in a different sequential order. (The number of loops may differ from \( L \) to \( L' \).) The new nest \( L' \) is equivalent to the old nest \( L \) (i.e., they give the same results), if whenever an iteration depends on another iteration in \( L \), the first iteration is executed after the second in \( L' \). If a loop in \( L' \) carries no dependence, then that loop can run in parallel. The goal is to transform \( L \) into an equivalent nest \( L' \), such that some inner and/or outer loops of \( L' \) can run in parallel.

Two major loop nest transformations are discussed here: unimodular and echelon. Unimodular transformations have already been introduced in this encyclopedia in the essay under that name. There it is shown by examples how to achieve inner and outer loop parallelization via unimodular transformations. Detailed algorithms for achieving those goals are given in this essay. Echelon transformations are explained after that, and it is pointed out how a combination of both transformations can maximize loop parallelization.

Mathematical Preliminaries

Lexicographic order and Fourier’s method of elimination, as explained in the essay Unimodular Transformations, will be used in the following. In this section, some of the elementary definitions and algorithms of matrix theory are stated, customized for integer matrices. From now on, a matrix is an integer matrix, unless explicitly designated to be otherwise.

The transpose of a matrix \( A \) is denoted by \( A' \). The \( m \times m \) unit matrix is denoted by \( I_m \). A submatrix of a given matrix \( A \) is the matrix obtained by deleting some rows and columns of \( A \). If \( A \) is an \( m \times p \) matrix and \( B \) an \( m \times q \) matrix, then the column-augmented matrix \((A;B)\) is the \( m \times (p+q) \) matrix whose first \( p \) columns are the columns of \( A \) and the last \( q \) columns are the columns of \( B \). Thus,

\[
A = \begin{pmatrix} 1 & 2 \\ 0 & 0 \\ 0 & 5 \end{pmatrix} \quad \text{and} \quad B = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \\
\begin{pmatrix} 0 & 1 \end{pmatrix} = (A;B). 
\]

A row-augmented matrix is defined similarly.

The (main) diagonal of an \( m \times n \) matrix \((a_{ij})\) is the vector \((a_{11}, a_{22}, \ldots, a_{pp})\) where \( p = \min(m,n) \). The rank of a matrix \( A \), denoted by \( \text{rank}(A) \), is the maximum number of linearly independent rows (or columns) of \( A \). For a given \( m \times n \) matrix \( A \), let \( \ell_i \) denote the column number of the leading (first nonzero) element of row \( i \). (For a zero row, \( \ell_i \) is undefined.) Then \( A \) is an echelon matrix, if for some integer \( \rho \) in \( 0 \leq \rho \leq m \), the following holds:

1. The rows 1 through \( \rho \) are nonzero rows.
2. The rows \( \rho + 1 \) through \( m \) are zero rows.
3. For \( 1 \leq i \leq \rho \), each element in column \( \ell_i \) below row \( i \) is zero.
4. \( \ell_1 < \ell_2 < \cdots < \ell_\rho \).

If there is such a \( \rho \), then \( \rho = \text{rank}(A) \). A zero matrix is an echelon matrix for which \( \rho = 0 \). Three nonzero echelon matrices with the values \( \rho = 4, 3, 2 \), respectively, are given below:

\[
\begin{pmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 6 \\ 0 & 0 & 0 \end{pmatrix}, \quad \begin{pmatrix} 5 & 2 & 3 \\ 0 & -1 & 0 \\ 0 & 0 & 3 \\ 0 & 0 & -1 \end{pmatrix}, \quad \begin{pmatrix} 0 & 5 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.
\]
For any given matrix, there are three types of elementary row operations:

1. **Reversal**: multiply a row by \(-1\).
2. **Interchange**: interchange two rows.
3. **Skewing**: add an integer multiple of one row to another row.

The elementary column operations are defined similarly.

An elementary matrix is any matrix obtained from a unit matrix by one elementary row operation. Thus, there are three types of elementary matrices. The elementary matrices can also be derived by column operations. For example, the same matrix is obtained from the \(3 \times 3\) unit matrix if we interchange rows 1 and 3, or interchange columns 1 and 3.

A unimodular matrix is a square integer matrix with determinant 1 or \(-1\). Each elementary matrix is unimodular, and each unimodular matrix can be expressed as the product of a finite number of elementary matrices (Lemma 2.3 in [2]). For a given matrix \(A\), performing an elementary row operation is equivalent to forming a product of the form \(EA\), where \(E\) is the corresponding elementary matrix. Applying a finite sequence of row operations to \(A\) is the same as forming a product of the form \(UA\), where \(U\) is a unimodular matrix.

Reducing an \(m \times n\) matrix \(A\) to echelon form means finding an \(m \times m\) unimodular matrix \(U\) and an \(m \times n\) echelon matrix \(S\), such that \(UA = S\). Figure 1 gives an algorithm that reduces any given matrix \(A\) to echelon form. Note that applying the same row operation to the two matrices \(U\) and \(S\) separately is equivalent to applying that operation to the column-augmented matrix \((U; S)\).

For a given matrix \(A\), one may need to find a unimodular matrix \(V\) and an echelon matrix \(S\) such that \(A = VS\). This \(V\) could be the inverse of the unimodular matrix \(U\) of Algorithm 1. However, if \(U\) is not needed, Algorithm 1 can be modified to yield directly a unimodular \(V\) and an echelon \(S\) such that \(A = VS\). This is the algorithm of Fig. 2. In Algorithm 2, whenever a row operation is applied to \(S\), it is followed by a column operation applied to \(V\). The row operation is equivalent to forming a product of the form \(ES\), where \(E\) is an elementary matrix. The column operation that follows is equivalent to forming the product \(VE^{-1}\). Hence, the relation \(A = VS\) remains unchanged since \(VS = (VE^{-1})(ES)\).

Each algorithm can be modified by using extra reversal operations, as needed, so that any given rows of the computed echelon matrix are nonnegative.

The program model for all transformations is a perfect nest of loops \(L = (L_1, L_2, \ldots, L_m)\) shown in Fig. 3. The limits \(p_r\) and \(q_r\) are integer-valued linear functions of \(I_1, I_2, \ldots, I_{r-1}\). All distance vectors are assumed to be uniform. If there is no distance vector \(d\) with \(d \succ \mathbf{0}\), then the loop \(L_r\) carries no dependence and is able to run in parallel.

**Unimodular Transformations**

Let \(U\) denote an \(m \times m\) unimodular matrix. The unimodular transformation of the loop nest \(L\) induced by \(U\) is

---

**Algorithm 1**

Given an \(m \times n\) integer matrix \(A\), this algorithm finds an \(m \times m\) unimodular matrix \(U\) and an \(m \times n\) echelon matrix \(S = (s_{ij})\), such that \(UA = S\). It starts with \(U = I_m\) and \(S = A\), and works on the matrix \((U; S)\). Let \(i_0\) denote the row number in which the last processed column of \(S\) had a nonzero element.

**Algorithm 1**

```plaintext
set \(U \leftarrow I_m\), \(S \leftarrow A\), \(i_0 \leftarrow 0\)
do \(j = 1, n, 1\)
  if there is at least one nonzero \(s_{ij}\) with \(i_0 < i \leq m\)
    then
      set \(i_0 \leftarrow i_0 + 1\)
      do \(i = m, i_0 + 1, 1\)
        do while \(s_{ij} \neq 0\)
          set \(\sigma \leftarrow \text{sgn}(s_{i-1,j} s_{i,j})\)
          \(z \leftarrow ||s_{i-1,j}|/|s_{ij}|\|
          subtract \(\sigma z\) times row \(i\) from row \((i - 1)\) in \((U; S)\)
o interchange rows \(i\) and \((i - 1)\) in \((U; S)\)
  enddo
enddo
```

---

Loop Nest Parallelization. Fig. 1 Echelon Reduction Algorithm
Algorithm 2 Given an $m \times n$ integer matrix $A$, this algorithm finds an $m \times m$ unimodular matrix $V$ and an $m \times n$ echelon matrix $S = (s_{ij})$, such that $A = VS$. It starts with $V = I_m$ and $S = A$, and works on the matrices $V$ and $S$. Let $i_0$ denote the row number in which the last processed column of $S$ had a nonzero element.

set $V \leftarrow I_m$, $S \leftarrow A$, $i_0 \leftarrow 0$

for $j = 1, n, 1$

if there is at least one nonzero $s_{ij}$ with $i_0 < i \leq m$ then

set $i_0 \leftarrow i_0 + 1$
do $i = m$, $i_0 + 1, -1$
do while $s_{ij} \neq 0$

set $\sigma \leftarrow \text{sgn}(s_{i-1,j} s_{ij})$

$z \leftarrow |s_{i-1,j}/|s_{ij}|$

subtract $\sigma z$ times row $i$ from row $(i-1)$ in $S$

add $\sigma z$ times column $(i-1)$ to column $i$ in $V$

interchange rows $i$ and $(i-1)$ in $S$

interchange columns $i$ and $(i-1)$ in $V$

end do

end if

end for

Loop Nest Parallelization. Fig. 2 Modified Echelon Reduction Algorithm

```
L_1 : do l_1 = p_1, q_1
L_2 : do l_2 = p_2, q_2
...               ...
L_m : do l_m = p_m, q_m

H(I)
```

Loop Nest Parallelization. Fig. 3 Loop Nest L

```
L'_1 : do k_1 = a_1, b_1
L'_2 : do k_2 = a_2, b_2
...               ...
L'_m : do k_m = a_m, b_m

H'(K)
```

Loop Nest Parallelization. Fig. 4 Loop Nest L'

A loop nest $L'$ of the form shown in Fig. 4, where $K = IU$ and $H'(K) = H(KU^{-1})$. The transformed program has the same set of iterations as the original program, executing in a different sequential order. The transformation $L \Rightarrow L'$ is Valid if and only if $dU > 0$ for each distance vector $d$ of $L$. In that case, the vectors $dU$ are precisely the distance vectors of the new nest $L'$.

Inner Loop Parallelization

Let $D$ denote the set of distance vectors of $L$. For the transformation $L \Rightarrow L'$ to be valid, the matrix $U$ must be so chosen that $dU > 0$ for each $d \in D$. After a valid transformation by $U$, the set of distance vectors of $L'$ is \{ $dU : d \in D$ \}. A loop $L'_i$ of $L'$ can run in parallel if there is no $d \in D$ satisfying $dU >_\alpha 0$. Thus, a unimodular matrix $U$ will transform the loop nest $L$ into an equivalent loop nest $L'$ where the loops $L'_1, L'_2, \ldots, L'_m$ can all run in parallel, if and only if $U$ satisfies the condition

$$dU >_\alpha 0 \quad (d \in D).$$

(1)

The following informal algorithm shows in detail that a unimodular matrix with this property can always be constructed.

Algorithm 3 Given a perfect nest $L$ of $m$ loops, as shown in Fig. 3, with a set of distance vectors $D$, this algorithm finds an $m \times m$ unimodular matrix $U = (u_{rt})$ that transforms $L$ into an equivalent loop nest $L'$, such that the inner loops $L'_1, L'_2, \ldots, L'_m$ of $L'$ can all run in parallel.

1. If $D = \emptyset$, then all loops of $L$ can execute in parallel; take for $U$ the $m \times m$ unit matrix and exit. Otherwise, go to step 2.
2. Condition (1) is equivalent to the following system of inequalities

$$d_1 u_1 + d_2 u_2 + \cdots + d_m u_m \geq 1 \quad (d \in D),$$

(2)

where $d = (d_1, d_2, \ldots, d_m)$ and the fixed second subscript of $u_{rt}$ has been dropped for convenience.

For each $r$ in $1 \leq r \leq m$, define the set

$$D_r = \{ d \in D : d >_\alpha 0 \}.$$
At least one of these sets is nonempty. Since \( d_1 = d_2 = \cdots = d_{r-1} = 0 \) for each \( d \in D_r \), one can break up the system (2) into a sequence of subsystems of the following type:

\[
\begin{align*}
    d_m u_m & \geq 1 & (d \in D_m) \\
    d_{m-1} u_{m-1} + d_m u_m & \geq 1 & (d \in D_{m-1}) \\
    \vdots & & \\
    d_1 u_1 + d_2 u_2 + \cdots + d_{m-1} u_{m-1} + d_m u_m & \geq 1 & (d \in D_1).
\end{align*}
\]

Since \( d_r > 0 \) for \( d \in D_r \), this sequence can be rewritten as

\[
\begin{align*}
    u_m & \geq 1/d_m & (d \in D_m) \\
    u_{m-1} & \geq (1 - d_m u_m)/d_{m-1} & (d \in D_{m-1}) \\
    \vdots & & \\
    u_1 & \geq (1 - d_2 u_2 - \cdots - d_m u_m)/d_1 & (d \in D_1).
\end{align*}
\]

3. If \( D_m = \emptyset \), then take \( u_m = 0 \). Otherwise, take \( u_m = 1 \). Suppose \( u_m, u_{m-1}, \ldots, u_{r-1} \) have been chosen, where \( 1 \leq r < m \). If \( D_r = \emptyset \), then take \( u_r = 0 \). Otherwise, \( u_r \) has a set of lower bounds of the form:

\[
u_r \geq (1 - d_{r+1} u_{r+1} - d_{r+2} u_{r+2} - \cdots - d_m u_m)/d_r \quad (d \in D_r).
\]

Take for \( u_r \) the smallest nonnegative integer that would work:

\[
u_r = \max\{ (1 - d_{r+1} u_{r+1} - d_{r+2} u_{r+2} - \cdots - d_m u_m)/d_r \}^+.
\]

4. Find \( s \) in \( 1 \leq s \leq m \) such that \( u_s \) is the first nonzero element in the sequence: \( u_m, u_{m-1}, \ldots, u_1 \). Note that \( u_s = 1 \). Set

\[
U = \begin{pmatrix}
    u_1 & 1 & 0 & 0 & \cdots & 0 \\
    \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
    u_{s-1} & 0 & 1 & 0 & \cdots & 0 \\
    1 & 0 & 0 & 0 & \cdots & 0 \\
    0 & 0 & 0 & 1 & \cdots & 0 \\
    \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\
    0 & 0 & 0 & 0 & \cdots & 1
\end{pmatrix},
\]

such that row \( s \) of \( U \) is \((1, 0, 0, \ldots, 0)\), its first column is \((u_1, u_2, \ldots, u_m)\), and the submatrix obtained by deleting row \( s \) and the first column of \( U \) is the \((m - 1) \times (m - 1)\) unit matrix. Since \( \det U = 1 \), \( U \) is unimodular.

5. Note that the inverse of \( U \) is given by

\[
U^{-1} = \begin{pmatrix}
    0 & \cdots & 0 & 1 & 0 & \cdots & 0 \\
    1 & 0 & \cdots & -u_1 & 0 & \cdots & 0 \\
    \vdots & \ddots & \ddots & \vdots & \ddots & \ddots & \vdots \\
    0 & 0 & \cdots & 0 & 1 & \cdots & 0 \\
    \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots \\
    0 & 0 & \cdots & 0 & 0 & \cdots & 1
\end{pmatrix}
\]

such that column \( s \) is \((1, -u_1, \ldots, -u_{s-1}, 0, \ldots, 0)\), row 1 is \((0, \ldots, 0, 1, 0, \ldots, 0)\), and the submatrix obtained by deleting column \( s \) and the first row is the \((m - 1) \times (m - 1)\) unit matrix. From the equation \((I_1, I_2, \ldots, I_m) = (K_1, K_2, \ldots, K_m)U^{-1}\), it follows that

\[
\begin{align*}
I_1 & = K_2 \\
\vdots & \quad \vdots \\
I_{s-1} & = K_s \\
I_s & = K_1 - u_1 K_2 - \cdots - u_{s-1} K_s \\
I_{s+1} & = K_{s+1} \\
\vdots & \quad \vdots \\
I_m & = K_m.
\end{align*}
\]

In the system of inequalities

\[p_r(I_1, \ldots, I_{r-1}) \leq I_r \leq q_r(I_1, \ldots, I_{r-1}) \quad (1 \leq r \leq m),\]

substitute for each \( I_r \) the corresponding expression in \( K_1, K_2, \ldots, K_m \). Get the loop limits \( \alpha_r(K_1, \ldots, K_{r-1}) \) and \( \beta_r(K_1, \ldots, K_{r-1}) \) of Fig. 4 by
6. Now each distance vector $d$ of $L$ satisfies the condition $dU >_1 0$. Hence, the transformed nest $L'$ of Fig. 4 created by this matrix $U$ is equivalent to the original nest $L$, and the inner loops of $L'$ can all execute in parallel.

**Remark 1** There exist infinitely many vectors $u = (u_1, u_2, \ldots, u_m)$ satisfying (3) and $\gcd(u_1, u_2, \ldots, u_m) = 1$. For each such vector $u$, there exist infinitely many unimodular matrices with $u$ as the first column. (See Sect. 3.3 in [2].) Any such matrix will satisfy the goal of Algorithm 3. Among all possible choices for $u$, an ideal vector is one that minimizes the number of iterations $(\beta_1 - \alpha_1 + 1)$ of the outermost loop $L'_1$ that must run sequentially. This poses an optimization problem; see Chap. 3 in [3] for details.

Since $K = IU$ and $(u_1, u_2, \ldots, u_m)$ is the first column of $U$, one has $K_1 = u_1I_1 + u_2I_2 + \cdots + u_mI_m$. An equation of the form $K_1 = c$, where $c$ is a constant, represents a hyperplane in the vector space $\mathbb{R}^m$ with coordinate axes $I_1, I_2, \ldots, I_m$. The integral values of $K_1$ from $\alpha_1$ to $\beta_1$ then represent a packet of parallel hyperplanes through the index space of the loop nest $L$, perpendicular to the vector $u$. Geometrically speaking, Algorithm 3 finds a sequence of parallel hyperplanes, such that the planes are taken sequentially, and iterations for all index points on each plane are executed in parallel. This algorithm is derived from Lamport's 1974 CACM paper [8]. The name Hyperplane Method, often used to describe it, also comes from [8].

**Example 1** To better understand how Algorithm 3 creates the matrix $U$ from a given set of distance vectors $D$, consider a loop nest $L = (L_1, L_2, L_3, L_4)$ with the set:

$$D = \{(0, 0, 0, 2), (0, 3, 1, -2), (0, 4, -6, 0), (1, -5, 3, 1), (2, 1, 0, 0), (3, 0, -2, 1)\}.$$

The only loop that carries no dependence is $L_3$. Hence, only $L_3$ can run in parallel. The goal is to find a unimodular matrix $U$ that will transform $L$ into an equivalent loop nest $L'$, where all the inner loops can execute in parallel. The first column $(u_1, u_2, u_3, u_4)$ of $U$ must satisfy the inequality

$$d_1u_1 + d_2u_2 + d_3u_3 + d_4u_4 \geq 1$$

for each $(d_1, d_2, d_3, d_4) \in D$. After simplification, one gets a set of six inequalities:

$$
\begin{align*}
U_4 & \geq 1/2 \\
U_2 & \geq (1 - u_3 + 2u_4)/3 \\
U_2 & \geq (1 + 6u_4)/4 \\
U_1 & \geq 1 + 5u_2 - 3u_3 - u_4 \\
U_1 & \geq (1 - u_2)/2 \\
U_1 & \geq (1 + 2u_3 - u_4)/3.
\end{align*}
$$

For each $u$, select the smallest possible nonnegative integral value that will work. So, take $u_4 = 1$. Since $D_3 = \emptyset$ (i.e., there is no distance vector $d$ with $d >_3 0$), take $u_3 = 0$. The constraints on $u_2$ are then $u_2 \geq 1$ and $u_2 \geq 1/4$. Take $u_2 = 1$. Finally, the lower bounds on $u_1$ are given by $u_1 \geq 5$, $u_1 \geq 0$, and $u_1 \geq 0$, so that one takes $u_1 = 5$. The unimodular matrix $U$ as constructed in Step 4 of Algorithm 3 is

$$U = \begin{pmatrix}
5 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0
\end{pmatrix}.$$

Here $s = 4$. Row 4 of $U$ is $(1, 0, 0, 0)$, its first column is $(u_1, u_2, u_3, u_4) = (5, 1, 0, 1)$, and the submatrix obtained by deleting the fourth row and the first column is the $3 \times 3$ unit matrix. The set of distance vectors of the equivalent loop nest $L' = (L'_1, L'_2, L'_3, L'_4)$ is the set of vectors $dU$ for $d \in D$, that is, the set

$$\{(2, 0, 0, 0), (1, 0, 3, 1), (4, 0, 4, -6), (1, 1, -5, 3), (11, 2, 1, 0), (16, 3, 0, -2)\}.$$

None of the inner loops $L'_2, L'_3, L'_4$ carries a dependence, and hence they can all run in parallel.

**Outer Loop Parallelization**

For a given $m \times n$ matrix $A = (a_{ij})$, the rows are denoted by $a_1, a_2, \ldots, a_m$ and the columns by $a^1, a^2, \ldots, a^n$. Note that the proof of the following theorem uses both the set $D$ of distance vectors and the distance matrix $D$ of $L$, while the previous algorithm used only $D$. 

Theorem 1 Consider a nest L of m loops. Let D denote the distance matrix of L and r the rank of D. Then there exists a valid unimodular transformation L → L' such that the outermost (m−r) loops and the innermost (r−1) loops of L' can execute in parallel.

Proof Let D denote the set of distance vectors of L. A unimodular matrix U will induce a valid transformation L → L' if and only if dDU > 0 for each d ∈ D. After a valid transformation by U, the distance vectors of L' are the vectors dU, where d ∈ D. The outermost (m−r) loops of D and the innermost (r−1) loops of L' can execute in parallel, if and only if dU >, 0 is false for each d ∈ D when r ≠ m−r+1. Let n = m−r. Then each d ∈ D must satisfy dU ≻, n+1 0.

The transpose D' of D has m rows and its columns are the distance vectors of L. By Algorithm 1, find an m × m unimodular matrix V and an echelon matrix S such that VD' = S. (S and D' have the same size.) The number of nonzero rows of S is the rank r of D, and the number of zero rows is n = m−r. Since the lowest n rows of S are zero vectors, it follows that each of the lowest n rows of V is orthogonal to each d ∈ D.

Next, find an m-vector u by Algorithm 3 such that du > 0 for each d ∈ D. Let U denote the m × (n+1) matrix where the columns a¹, a², ..., aⁿ are the lowest n rows of V, and aⁿ⁺¹ = u. Then each d ∈ D satisfies the equations:

\[ da¹ = 0, \ da² = 0, \ldots, daⁿ = 0, \ daⁿ⁺¹ > 0. \] (4)

By Algorithm 2, find an m × m unimodular U and an m × (n+1) echelon matrix

\[ T = \begin{pmatrix} t_{11} & t_{12} & t_{13} & \cdots & t_{1n} & t_{1,n+1} \\ 0 & t_{22} & t_{23} & \cdots & t_{2n} & t_{2,n+1} \\ 0 & 0 & t_{33} & \cdots & t_{3n} & t_{3,n+1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & t_{nn} & t_{n,n+1} \\ 0 & 0 & 0 & \cdots & 0 & t_{n+1,n+1} \\ 0 & 0 & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 0 & 0 \end{pmatrix} \]

such that A = UT and the diagonal element t_{n+1,n+1} is nonnegative. The unimodular transformation L → L' induced by U satisfies the theorem.

After writing the relation A = UT in the form

\[ (a¹, a², \ldots, aⁿ⁺¹) = (u¹, u², \ldots, uⁿ, uⁿ⁺¹, uⁿ⁺², \ldots, uᵐ⁺¹) \cdot T, \]

it becomes clear that

\[ \begin{align*}
  a¹ &= t_{11} u¹ \\
  a² &= t_{12} u¹ + t_{22} u² \\
  &\vdots \\
  aⁿ &= t_{1n} u¹ + t_{2n} u² + \cdots + t_{nn} uⁿ \\
  aⁿ⁺¹ &= t_{1,n+1} u¹ + t_{2,n+1} u² + \cdots + t_{n,n+1} uⁿ + t_{n+1,n+1} uⁿ⁺¹.
\end{align*} \] (5)

Since a¹, a², ..., aⁿ form distinct rows of a unimodular matrix V, they are linearly independent. Hence, the diagonal elements t₁₁, t₂₂,..., tₙₙ of T must be nonzero. Multiply the equations of (5) by any d ∈ D and use (4) to get

\[ \begin{align*}
  t_{11} (du¹) &= 0 \\
  t_{12} (du¹) + t_{22} (du²) &= 0 \\
  &\vdots \\
  t_{1n} (du¹) + t_{2n} (du²) + \cdots + t_{nn} (duⁿ) &= 0 \\
  t_{1,n+1} (du³) + t_{2,n+1} (du⁴) + \cdots + t_{n,n+1} (duⁿ) + t_{n+1,n+1} (duⁿ⁺¹) &= 0.
\end{align*} \]

Since t₁₁, t₂₂,..., tₙₙ are all nonzero and tₙ₊₁,ₙ₊₁ ≥ 0, this implies

\[ du¹ = 0, \ du² = 0, \ldots, duⁿ = 0, \ duⁿ⁺¹ > 0, \]

that is, dU >ₙ₊₁ 0, for each d ∈ D. This completes the proof.

Example 2 Consider a loop nest L = (L₁, L₂, L₃) whose distance matrix D and its transpose D' are given by

\[ D = \begin{pmatrix} 6 & 4 & 2 \\ 0 & 1 & -1 \\ 1 & 0 & 1 \end{pmatrix} \quad \text{and} \quad D' = \begin{pmatrix} 6 & 0 & 1 \\ 4 & 1 & 0 \\ 2 & -1 & 1 \end{pmatrix}. \]
Here \( m = 3 \). By Algorithm 1, find two matrices
\[
V = \begin{pmatrix}
0 & 0 & 1 \\
0 & 1 & -2 \\
1 & -1 & -1
\end{pmatrix}
\quad \text{and} \quad
S = \begin{pmatrix}
2 & -1 & 1 \\
0 & 3 & -2 \\
0 & 0 & 0
\end{pmatrix},
\]
such that \( V \) is unimodular, \( S \) is echelon, and \( VD' = S \). Then \( \rho = 2 \) and \( n = 1 \). Since the bottom row of \( S \) is zero, it follows that the bottom row \((1, -1, -1)\) of \( V \) is orthogonal to each column of \( D' \), that is, to each distance vector \( d \) of \( L \). This will be the first column of a \( 2 \times 3 \) matrix \( A \) that is being constructed.

Next, one needs a vector \( u \) such that \( du > 0 \), or equivalently, \( du \geq 1 \) for each distance vector \( d \). The set of inequalities to be satisfied is:
\[
\begin{align*}
6u_1 & + 4u_2 + 2u_3 \geq 1 \\
u_2 & - u_3 \geq 1 \\
u_1 & + u_3 \geq 1
\end{align*}
\]
or
\[
\begin{align*}
u_2 & \geq 1 + u_3 \\
u_1 & \geq (1 - 4u_2 - 2u_3)/6 \\
u_1 & \geq 1 - u_3.
\end{align*}
\]
Algorithm 3 returns the vector \( u = (1, 1, 0) \). This will be the second column of the matrix \( A \). Thus,
\[
A = \begin{pmatrix}
1 & 1 \\
-1 & 1 \\
-1 & 0
\end{pmatrix}.
\]

By Algorithm 2, find two matrices
\[
U = \begin{pmatrix}
-1 & 1 & 1 \\
1 & 1 & 0 \\
1 & 0 & 0
\end{pmatrix}
\quad \text{and} \quad
T = \begin{pmatrix}
-1 & 0 \\
0 & 1 \\
0 & 0
\end{pmatrix},
\]
such that \( U \) is unimodular, \( T \) is echelon (with a nonnegative diagonal element on second row), and \( A = UT \). Since
\[
(6, 4, 2)U = (0, 10, 6), \quad (0, 1, -1)U = (0, 1, 0),
\]
and \( (1, 0, 1)U = (0, 1, 1) \)
are all positive vectors, the transformation \((L_1, L_2, L_3) \Rightarrow (L_1', L_2', L_3')\) induced by \( U \) is valid. The distance vectors of \((L_1', L_2', L_3')\) are \((0, 10, 6), \quad (0, 1, 0)\) and \((0, 1, 1)\). Since the loops \( L_1' \) and \( L_2' \) do not carry a dependence, they can run in parallel.

**Echelon Transformation**

Some background needs to be prepared before echelon transformations can be defined. Let \( N \) denote the number of distance vectors of the loop nest \( L \) of Fig. 3, and \( D \) its distance matrix. Apply Algorithm 2 to the \( N \times m \) matrix \( D \) to find an \( N \times \rho \) unimodular matrix \( V \) and an \( N \times m \) echelon matrix \( S = (s_{t\nu}) \) with nonnegative rows, such that \( D = VS \). Let \( \rho \) denote the number of positive rows of \( S \), so that \( \rho = \text{rank}(D) \).

The top \( \rho \) rows of \( S \) are positive rows and the bottom \((N - \rho)\) rows are zero rows. Let \( \hat{S} \) denote the \( \rho \times m \) submatrix of \( S \) consisting of the positive rows, and \( \hat{V} \) the \( N \times \rho \) submatrix of \( V \) consisting of the leftmost \( \rho \) columns. Then, \( D \) can be written in the form:
\[
D = \hat{V}\hat{S}.
\]

**Lemma 1** The rows of \( \hat{V} \) are positive vectors.

**Proof** Let \( v \) denote any row of \( \hat{V} \). Then \( \hat{V} = d \), where \( d \) is a distance vector. Since \( d \) must be positive, \( v \) cannot be the zero vector. Hence, it has the form \( v = (0, \ldots, 0, v_t, \ldots, v_{\ell}) \), where \( 1 \leq t \leq \rho \) and \( v_t \neq 0 \). Let \( \ell = \ell_t \) denote the column number of the leading (first nonzero) element on row \( t \) of \( \hat{S} \). Then, \( s_{t\ell} > 0 \), column \( \ell \) of \( \hat{S} \) has the form \((s_{t\ell}, s_{2\ell}, \ldots, s_{e\ell}, 0, \ldots, 0)\), and each column \( j \) for \( 1 \leq j < \ell \) has the form \((s_{tj}, s_{2j}, \ldots, s_{(\ell-1)j}, 0, \ldots, 0)\). Hence, the product \( v\hat{S} \) has the form \((0, \ldots, 0, v_ts_{t\ell}, \ldots)\). Now, \( v_t s_{t\ell} \neq 0 \) since \( v_t \neq 0 \) and \( s_{t\ell} > 0 \). However, being the leading element of a distance vector, \( v_ts_{t\ell} \) must be positive. This means \( v_t > 0 \), since \( s_{t\ell} > 0 \). Thus, \( v \) is a positive vector. \( \Box \)

**Lemma 2** For \( 1 \leq t \leq \rho \), let \( \ell_t \) denote the column number of the leading element on row \( t \) of \( \hat{S} \). For each \( i \in Z^\rho \), there exists a unique \( Y \in Z^\rho \) and a unique \( K \in Z^\rho \), such that
\[
0 \leq Y_{\ell_t} \leq s_{t\ell_t} - 1 \quad (1 \leq t \leq \rho)
\]
and
\[
I = Y + K\hat{S}.
\]
Proof  Note that the leading elements $s_{i\ell_i}$ are all positive by construction. Let $L = (I_1, I_2, \ldots, I_m) \in \mathbb{Z}^m$. Define $K = (K_1, K_2, \ldots, K_p) \in \mathbb{Z}^p$ by

\[
\begin{align*}
K_1 &= I_{\ell_1}/s_{\ell_1} \\
K_2 &= (I_{\ell_1} - s_{\ell_1}K_1)/s_{\ell_2} \\
& \vdots \\
K_p &= (I_{\ell_p} - s_{\ell_p}K_1 - s_{\ell_p}K_2 - \cdots - s_{p-1,\ell_{p-1}}K_{p-1})/s_{p\ell_{p}}.
\end{align*}
\]

After defining $K$, define $Y \in \mathbb{Z}^m$ by $Y = I - KS$. Then $Y$ and $K$ are well defined and (8) holds.

For any real number $x$, one has $0 \leq x - [x] < 1$. If $I$ and $s$ are integers with $s > 0$, then $0 \leq I/s - [I/s] < 1$ implies $0 \leq I - s[I/s] \leq s - 1$. Hence, if $I = Y + s[I/s]$, then $0 \leq Y \leq s - 1$. Since (8) implies

\[
I_{\ell_1} = Y_{\ell_1} + s_{\ell_1}K_1 \\
I_{\ell_2} - s_{\ell_1}K_1 = Y_{\ell_2} + s_{\ell_2}K_2 \\
& \vdots \\
I_{\ell_p} - s_{\ell_p}K_1 - s_{\ell_p}K_2 - \cdots - s_{p-1,\ell_{p-1}}K_{p-1} = Y_{\ell_p} + s_{p\ell_{p}}K_p,
\]

it follows from the definition of $(K_1, K_2, \ldots, K_p)$ that $Y_{\ell_1}, Y_{\ell_2}, \ldots, Y_{\ell_p}$ satisfy (7). \hfill \square

The index variables $I_1, I_2, \ldots, I_m$ of the loop nest of Fig. 3 satisfy the constraints:

\[p_r(I_1, I_2, \ldots, I_{r-1}) \leq I_r \leq q_r(I_1, I_2, \ldots, I_{r-1}) \quad (1 \leq r \leq m).\]

For $I_1, I_2, \ldots, I_m$ in these inequalities, substitute the corresponding expressions in $Y_1, Y_2, \ldots, Y_m, K_1, K_2, \ldots, K_p$ from (8). Add (7) to that system to get a system of inequalities in $Y_1, Y_2, \ldots, Y_m, K_1, K_2, \ldots, K_p$. Apply Fourier’s method to the combined system and eliminate the variables $K_p, K_{p-1}, \ldots, K_1, Y_m, Y_{m-1}, \ldots, Y_1$ (in this order) to get bounds of the form:

\[
\begin{align*}
\alpha_1 &\leq Y_1 \leq \beta_1 \\
\alpha_2(Y_1) &\leq Y_2 \leq \beta_2(Y_1) \\
& \vdots \\
\alpha_m(Y_1, Y_2, \ldots, Y_{m-1}) &\leq Y_m \leq \beta_m(Y_1, Y_2, \ldots, Y_{m-1}) \\
\alpha_{m+1}(Y_1, K_1) &\leq K_1 \leq \beta_{m+1}(Y) \\
\alpha_{m+2}(Y, K_1) &\leq K_2 \leq \beta_{m+2}(Y, K_1) \\
& \vdots \\
\alpha_{m+p}(Y, K_1, K_2, \ldots, K_{p-1}) &\leq K_p \leq \beta_{m+p}(Y, K_1, K_2, \ldots, K_{p-1}),
\end{align*}
\]

where $Y = (Y_1, Y_2, \ldots, Y_m)$. For each index point $I$ of $L$, there is a unique $(m+p)$-vector $(Y, K)$ satisfying these constraints, and conversely. The nest $L'$ of $(m+p)$ loops shown in Fig. 5, where $I = Y + KS$ and $H'(Y, K) = H(I)$, has the same set of iterations as $L$, but the order of execution is different. The transformation $L \Rightarrow L'$ is called the echelon transformation of the loop nest $L$.

Two iterations $H(i)$ and $H(j)$ in $L$ become iterations $H'(y, k)$ and $H'(z, l)$, respectively, in $L'$, where $i = y + kS$ and $j = z + lS$. The transformed nest $L'$ is equivalent to the original nest $L$, if whenever $H(j)$ depends on $H(i)$ in $L$, $H'(y, k)$ precedes $H'(z, l)$ in $L'$, that is, $(y, k) < (z, l)$.

**Theorem 2**  The loop nest $L'$ obtained from a nest $L$ by echelon transformation is equivalent to $L$, and the $m$ outermost loops of $L'$ can run in parallel.

**Proof**  To prove the equivalence of $L'$ to $L$, consider two iterations $H(i)$ and $H(j)$ of $L$, such that $H(j)$ depends on $H(i)$ in $L$. Let $(y, k)$ denote the value of $(Y, K)$ corresponding to the value $i$ of $I$, and $(z, l)$ the value corresponding to $j$. Since $d = j - i$ is a distance vector of $L$, it follows from (6) that $d = vS$ for some row $v$ of $S$.
Then
\[ j = i + d = y + k\overline{s} + v\overline{s} = y + (k + v)\overline{s}. \]

Since \( y \) satisfies (7), \((y, k + v)\) is the image of \( j \) under the mapping \( I \mapsto (Y, K) \). But the image of \( j \) is also \((z, I)\) by assumption. Since this mapping is well defined, it follows that \( z = y \) and \( I = k + v \). Hence,
\[ (z, I) - (y, k) = (z - y, I - k) = (0, v). \]

Since \( v \) is positive by Lemma 2, this implies \((y, k) < (z, I)\). Hence, \( L' \) is equivalent to \( L \) by definition.

It is clear from the above discussion that the distance vectors of the nest \( L' \) are of the form \((0, v)\), where \( v \) is a row of \( V \). This means the \( m \) outermost \( Y \)-loops carry no dependence, and hence they can run in parallel. \( \square \)

**Corollary 1**  In \( L' \), the distance matrix of the innermost nest of \( K \)-loops is \( \overline{V} \).

**Example 3**  Consider the loop nest \( L \):

\[
\begin{align*}
L_1 &: \quad \text{do } i_1 = 1, 100 \\
L_2 &: \quad \text{do } i_2 = 1, 200 \\
L_3 &: \quad \text{do } i_3 = i_1, 300 \\
& \quad X(i_1, i_2, i_3) = X(i_1 - 1, i_2 - 3, i_3) + \]
\[
X(i_1 - 2, i_2 - 10, i_3) + X(i_1 - 1, i_2 + 1, i_3)
\]
\end{align*}
\]

whose distance matrix is
\[
D = \begin{pmatrix}
1 & 3 & 0 \\
2 & 10 & 0 \\
1 & -1 & 0
\end{pmatrix}
\]

By Algorithm 2, find two matrices
\[
V = \begin{pmatrix}
1 & 1 & 0 \\
2 & 3 & 1 \\
1 & 0 & 0
\end{pmatrix}
\quad \text{and} \quad
S = \begin{pmatrix}
1 & -1 & 0 \\
0 & 4 & 0 \\
0 & 0 & 0
\end{pmatrix}
\]

such that \( V \) is unimodular, \( S \) is echelon, and \( D = VS \). The leading elements of nonzero rows of \( S \) are positive. It is clear that \( p = \text{rank}(D) = \text{rank}(S) = 2 \). The submatrix \( \overline{S} \) of \( S \) consisting of its nonzero rows is given by
\[
\overline{S} = \begin{pmatrix}
1 & -1 & 0 \\
0 & 4 & 0
\end{pmatrix}
\]

Here \( s_{1i_1} = 1 \) and \( s_{2i_1} = 4 \). Define a mapping \((I_1, I_2, I_3) \mapsto (Y_1, Y_2, Y_3, K_1, K_2)\) of the index space of \( L \) into \( \mathbb{Z}^5 \) by the equation
\[
(I_1, I_2, I_3) = (Y_1, Y_2, Y_3) + (K_1, K_2) \cdot \begin{pmatrix}
1 & -1 & 0 \\
0 & 4 & 0
\end{pmatrix},
\]

and the constraints
\[
0 \leq Y_1 \leq 0 \quad \text{and} \quad 0 \leq Y_2 \leq 3.
\]

**Equation (9)** is equivalent to the system:
\[
\begin{cases}
I_1 &= Y_1 + K_1 \\
I_2 &= Y_2 - K_1 + 4K_2 \\
I_3 &= Y_3.
\end{cases}
\]

Substituting for \( I_1, I_2, I_3 \) from (11) into the constraints defining the loop limits of \( L \), one gets the following set of inequalities:
\[
\begin{cases}
1 & \leq Y_1 + K_1 \leq 100 \\
1 & \leq Y_2 - K_1 + 4K_2 \leq 200 \\
Y_1 + K_1 & \leq Y_3 \leq 300.
\end{cases}
\]

Eliminate the variables \( K_2, K_1, Y_3, Y_2, Y_1 \) from (10) and (12) by Fourier's method:
\[
\begin{cases}
[(1 - Y_2 + K_1)/4] & \leq K_2 \leq [(200 - Y_2 + K_1)/4] \\
1 - Y_1 & \leq K_1 \leq \min(100 - Y_1, Y_3 - Y_1) \\
1 & \leq Y_3 \leq 300 \\
0 & \leq Y_2 \leq 3 \\
0 & \leq Y_1 \leq 0.
\end{cases}
\]

Using the fact that \( Y_1 = 0 \), simplify expressions and get

the following loop nest \( L' \) equivalent to \( L \):

\[
\begin{align*}
L_2 &: \quad \text{do } Y_2 = 0, 3 \\
L_3 &: \quad \text{do } Y_3 = 1, 300 \\
L_4 &: \quad \text{do } K_1 = 1, \min(100, Y_3) \\
L_5 &: \quad \text{do } K_2 = \left[(1 - Y_2 + K_1)/4, \right. \\
& \left. [(200 - Y_2 + K_1)/4] \right. \\
& \left. X(K_1, Y_2 - K_1 + 4K_2, Y_3) = \right. \\
& \left. X(K_1 - 1, Y_2 - K_1 + 4K_2 - 3, Y_3) + \right. \\
& \left. X(K_1 - 2, Y_2 - K_1 + 4K_2 - 10, Y_3) + \right. \\
& \left. X(K_1 - 3, Y_2 - K_1 + 4K_2 + 1, Y_3) \right.
\end{align*}
\]
In $L'$, the $Y_2$ and the $Y_3$ loops can run in parallel. (The $Y_1$-loop with a single iteration has been omitted.)

The distance matrix of the nest of $K$-loops is the sub-matrix consisting of the two leftmost columns of $V$, that is, the matrix:

\[
\hat{V} = \begin{pmatrix}
1 & 1 \\
2 & 3 \\
1 & 0 
\end{pmatrix}.
\]

Since the $K_2$-loop carries no dependence, it can run in parallel.

**Remark 2** When $\rho = m$, no valid unimodular transformation can give a parallel outer loop (see Corollary 1 to Theorem 3.8 in [3]). A simple example would be a nest of two loops with two distance vectors $(2, 0)$ and $(0, 3)$, for which $\rho = 2 = m$. However, echelon transformation of this nest will yield a parallel $Y_1$-loop with 2 iterations and a parallel $Y_2$-loop with 3 iterations.

If both transformations are available, the best strategy would be to apply the echelon transformation first to get a nest of $m$ parallel outermost $Y$-loops and \( \rho \) sequential innermost $K$-loops. Then apply Algorithm 3 to the nest of $K$-loops to get one sequential outermost loop followed by $(\rho - 1)$ parallel inner loops.

**Bibliographic Notes and Further Reading**

**Unimodular Transformation.** Research on this topic goes back to Leslie Lamport’s paper in 1974 [8], although he did not use this particular term. This essay is based on the theory of unimodular transformations as developed in the author’s books on loop transformations [2, 3]. The general theory grew out of the theory of unimodular transformations of double loops described in [1]. (Watch for some notational differences between these references and the current essay.) The paper by Michael Wolf and Monica Lam [13] covers many useful aspects of unimodular transformations. See also Michael Dowling [5], Erik H. D’Hollander [4], and François Irigoin and Rémi Triolet [6, 7].

Consider a sequential loop nest where each iteration (other than the first) depends on the previous one. The Hyperplane method would still transform it into a nest where the outermost loop is sequential and all inner loops are parallel. There is no great surprise here. It simply means that each inner loop in the new nest will have a single iteration. (Such a loop can run in parallel according to the definition given.) See [13] for the case when there are too many distance vectors.

**Echelon Transformation.** In his PhD thesis [9], David Padua developed the greatest common divisor method for finding a partition of the index space. Peir and Cytron [10] described a partition based on minimum dependence distances. Shang and Fortes [12] offered an algorithm for finding a maximal independent partition. D’Hollander’s paper [4] builds on and incorporates these approaches. The general theory of echelon transformations presented here is influenced by these works. See also Polychronopoulos [11].

See [3] for details on unimodular, echelon, and other transformations used in loop nest parallelization, and more references.

**Related Entries**

- Code Generation
- Parallelism Detection in Nested Loops, Optimal
- Parallelization, Automatic
- Unimodular Transformations

**Bibliography**

Loops, Parallel

Roland Wismüller
University of Siegen, Siegen, Germany

Synonyms
Doall loops; Forall loops

Definition
Parallel loops are one of the most widely used concepts to express parallelism in parallel languages and libraries. In general, a parallel loop is a loop whose iterations are executed at least partially concurrently by several threads or processes. There are several different kinds of parallel loops with different semantics, which will be discussed below. The most prominent kind is the Doall loop, where the iterations are completely independent.

Discussion
Introduction
In a task parallel program, where the execution of different pieces of code is distributed to parallel processors, there are two principal ways of specifying parallel activities. The first one is to specify several different code regions, i.e., tasks, which should be executed in parallel. This construct is typically called parallel regions or parallel case. It offers only a very limited scalability, since the maximum degree of parallelism is determined by the number of regions. The second way is to use parallel loops. In a parallel loop, the parallel processors execute the same code region, namely, the loop body, but with different data. Thus, parallel loops are a special kind of SPMD programming. Typically, parallel loops are used within a shared memory programming model, for example, OpenMP and Intel’s Threading Building Blocks. However, they have also been included in some distributed memory programming models, for example, Occam.

In a sequential loop, the loop body is executed for each element in the loop’s index domain in a fixed order implied by the domain. For example, in

```c
for (i=0; i<N; i++)
    s[i] = sin(PI*i/N);
```
the assignments to `s` occur in the order `s[0]`, `s[1]`, ..., `s[N]`, although there is obviously no particular reason which mandates this order of the loop iterations. In a parallel loop, the restrictions on the ordering of the loop iterations are relaxed, which allows the iterations to execute – at least partially – in parallel. In the example, all iterations can execute in parallel, since they are completely independent of each other.

Types of Parallel Loops
Today’s parallel programming interfaces offer different constructs for parallel loops, which typically fall into one of the following classes which are described by Polychronopoulos [7] and Wolfe [10]: First, there are parallel loops which can be viewed as sequential loops extended by additional properties. A *Doall loop* assures that its iterations can be executed completely independently, while a *Doacross loop* may contain forward dependences. These loops can also be executed sequentially without changing their semantics. In addition, some parallel languages offer a *Forall loop*; however, the term is used with a nonuniform meaning. For example, the Forall loop in Vienna Fortran [3] is actually a Doall loop, while in High Performance Fortran (HPF)
Loops, Parallel

and Fortran95, it has a special nonsequential execution semantics, which is described below.

**Doall Loops**

In a Doall loop, every iteration can be executed independently from any other iteration. This means that the iterations can be executed in any sequential order or concurrently.

More formally, a Doall loop is not allowed to contain loop-carried dependences. This means that when \( W(i) \) denotes the set of variables the loop’s body writes to in some iteration \( i \) and \( R(i) \) denotes the set of variables read in iteration \( i \) the following conditions, often called Bernstein’s conditions, must hold for any two different iterations \( i_1 \) and \( i_2 \) of the loop:

\[
W(i_1) \cap W(i_2) = \emptyset \quad \text{and} \quad R(i_1) \cap W(i_2) = \emptyset \quad \text{and} \quad W(i_1) \cap R(i_2) = \emptyset
\]

In this context, the term variable means either a scalar variable or an element of an array or some other structured data type. As an example, consider the following loops:

```plaintext
for (i=0; i<N; i++) {
    b[i] = sin(PI*i/N);
    a[i] = a[i] + b[i];
}

for (i=0; i<N; i++) {
    b[i] = sin(PI*i/N);
    a[i] = a[i-1] + b[i];
}
```

Here, the first loop is a Doall loop, while the second one is not. For the first loop, \( W(i) = \{a[i], b[i]\} \) and \( R(i) = \{a[i], b[i], i\} \). Thus, for different iterations \( i_1 \) and \( i_2 \), the write and read sets never overlap. On the other hand, for the second loop, \( W(i) = \{a[i], b[i]\} \) and \( R(i) = \{a[i-1], b[i], i\} \) which leads to \( W(i_1) \cap R(i_2) = \{a[i_1]\} \) for \( i_2 = i_1 + 1 \).

As an additional requirement, the iteration domain of a Doall loop must be fixed when the loop is entered, that is, the loop body is neither allowed to exit the loop prematurely nor to change the loop variable or its upper bound.

When a Doall loop is to be executed by a fixed number of processors (or threads), the iterations can be arbitrarily scheduled to the processors without changing the loop’s computational results. However, the scheduling can have a large impact on the performance. With today’s cache-based memory systems, it is beneficial if each processor executes a contiguous block of loop iterations since this will maximize the locality of the memory accesses on each processor. For example, when a loop iterating from 1 to 100 is executed by four processors, the first one would execute iterations 1–25, the second one 26–50, and so on. This situation is visualized in Fig. 1.

The blockwise scheduling works well, however, only when all iteration have approximately the same execution time. When the execution time differs largely between iterations, blockwise scheduling can lead to a poor performance. A typical example for this situation is a triangular loop nest, where the outer loop is parallel:

```plaintext
for (i=1; i<=100; i++)
    for (j=1; j<=i; j++)
        a[i][j] = some_computation(i,j);
```

With a blockwise scheduling, processor 4 gets the largest amount of work. Since there is typically a global synchronization (barrier) between all processors at the end of a parallel loop, the other three processors will have to wait, which results in a load imbalance. In such cases, a cyclic schedule, where single iterations are assigned to processors in a round-robin fashion offers a better load balancing, as it is shown in Fig. 2. However, the cache performance of such an execution may be poor. As a compromise between cache performance and load balancing, blocks of iterations may be assigned to the processors in a cyclic manner.

<table>
<thead>
<tr>
<th>Processor 1:</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor 2:</td>
<td>26</td>
<td>27</td>
<td>...</td>
<td>50</td>
</tr>
<tr>
<td>Processor 3:</td>
<td>51</td>
<td>52</td>
<td>...</td>
<td>75</td>
</tr>
<tr>
<td>Processor 4:</td>
<td>76</td>
<td>77</td>
<td>...</td>
<td>100</td>
</tr>
</tbody>
</table>

**Loops, Parallel. Fig. 1** Blockwise schedule for a simple parallel loop
In addition to the static scheduling discussed above, dynamic scheduling may be used when the iterations’ execution times vary irregularly. In this case, each processor fetches an iteration from a global work pool, executes it, and then fetches the next one, until the work pool is empty. Dynamic scheduling achieves very good load balancing at the price of the additional overhead for the synchronized access to the global work pool. Like in the static case, processors may fetch contiguous blocks of iterations from the pool in order to increase the cache performance and to decrease the overhead.

Doacross Loops
A Doacross loop is a parallel loop, where individual iterations are scheduled to the processors in a round-robin fashion, either dynamically or statically. The iterations are started in the same order as in a sequential loop and then execute overlappingly in a pipelined fashion. This allows Doacross loops to possess forward dependences between iterations.

The following example shows a best case for a Doacross loop:

```c
for (i=1; i<N; i++) {
    a[i] = f(i);
    b[i] = a[i] - a[i-1];
}
```

Although this loop contains a flow dependence, since the value of a[i] written in iteration i is read by a[i-1] in the following iteration i + 1, it can be executed in parallel with optimal scalability. Figure 3 shows an execution under the assumption that the first statement of the loop body always needs two time units to execute, while the second one needs just one. Of course, in practice, some synchronization statements must be added to the loop in order to ensure the correct temporal order of the dependent statements:

```c
for (i=1; i<N; i++) {
    a[i] = f(i);
    signal(i);
    wait(i-1);
    b[i] = a[i] - a[i-1];
}
```

Here, the operation wait(i-1) waits until the corresponding signal(i-1) has been executed, which indicates that the assignment to a[i-1] has finished.

There are also loops, called **serial loops**, where without code restructuring dependences in the loop body can completely inhibit any parallelism, for example,

```c
for (i=1; i<N; i++) {
    a[i] = b[i-1];
    b[i] = f(i);
}
```

As the schedule in Fig. 3 shows, the loop iterations cannot overlap due to the dependence, since the first statement of iteration i + 1 depends on the last statement of iteration i. However, in this special case, the two statements could be swapped, enabling nearly perfect parallelism. This kind of reorganization is typically performed by interactive restructuring tools or autoparalleling compilers.

Forall Loops
The Fortran D language [5] introduced a parallel loop, called **FORALL**, which deals with dependences in a special way: A read operation on a variable will return a new value only if that value was assigned in the current loop iteration, otherwise, it returns the old value of the variable at the time the loop was entered. Thus, conceptually each iteration works on its own copy of the data space.

In High Performance Fortran (HPF) [4], this construct was inherited with some modifications under the
Loops, Parallel

Loops, Parallel. Fig. 3 Best case execution of a Doacross loop: full overlap

Loops, Parallel. Fig. 4 Worst case execution of a Doacross loop: no overlap

Variable in Parallel Loops

In a shared memory environment, variables in a parallel loop may either be shared between all threads executing the loop or they may be private, which means that each thread has its own copy of the variable. Typically, the data the loop is working on is shared, while the loop variables of the parallel loop and all its inner loops must be private. Private variables can also be used to compute thread local intermediate results inside a loop, which are combined only after the loop terminated.

A special case of this situation is a reduction variable. A reduction variable is a variable $v$ such that each assignment in the loop has the form $v = v \oplus \cdots$, with $\oplus$ being an associative operator, and $v$ is not otherwise read in the loop. A typical example is the computation of the integral of a function:

```plaintext
double s = 0;
for (i=0; i<N; i++)
    s += f((i+0.5)/N) / N;
```

Strictly, this loop can only be executed in parallel as a Doall loop, where the exact ordering of the summation is preserved. This is shown in Fig. 5a, where it is assumed that the execution time of $f()$ shows some variation. If the reduction operation is associative and commutative, which is true for integer arithmetic, but due to rounding errors typically not for floating point arithmetic, the order of the summation may be changed arbitrarily without changing the result. In this case, the loop may be executed as a Doall loop, provided that the concurrent updates to the shared variable $s$ are protected by a proper synchronisation for mutual exclusion. Fig. 5b shows that although the loop performs better than the Doacross loop, the degree of parallelism is still rather limited. A nearly optimal parallelization can be achieved using a private reduction variable. As indicated in Fig. 5c, each thread first computes a partial sum in its private instance of the variable. At the end of the loop, the partial sums are added to the global sum within a small critical section. Parallel programming interfaces typically allow to mark reduction variables in

name **FORALL construct**, which was later included into the Fortran 95 language standard. The statements inside a FORALL construct are executed in their sequential order; however, each statement is executed for all elements of the index domain by first performing all read operations and only afterward doing the write operations. For example, in the construct:

```plaintext
FORALL ( i=2:n-1 )
    a(i) = a(i-1) + a(i+1)
    b(i) = a(i)
END FORALL
```

first the expression $a(i-1) + a(i+1)$ will be evaluated for all values of $i$, and only afterward all the assignments to $a(i)$ will occur. Then, the second statement is executed in a similar way to copy $a$ into $b$. Thus, the FORALL construct in HPF and Fortran 95 actually is rather a convenient way to express a sequence of vectorizable and/or parallelizable array assignments than a (parallel) loop.
parallel loops and transparently introduce a local reduction variable as outlined above. A code example using OpenMP is given in the next section.

**Parallel Loops in OpenMP**

OpenMP is a parallel programming model for shared memory computers, which extends traditional sequential languages with directives controlling the parallel execution. The main construct is the parallel region, which indicates that the region’s code should be executed by a prespecified number of threads. Inside such a parallel region, loops may be designated as parallel loops, resulting in the loop iterations being distributed across the available threads. In C and C++, this is done using the directive

```c
#pragma omp for
```

when the loop is already contained in a parallel region, or the combined directive

```c
#pragma omp parallel for
```

which additionally creates a parallel region around the loop. These directives must be placed immediately before the loop. OpenMP puts a couple of restrictions on a parallel loop: it must be a for-loop with an integer loop variable, constant increment, and a simple termination test. The loop must not exit prematurely, for example, via a break or return statement or an exception. In addition, the loop body must neither modify the loop variable nor the loop bounds.

The detailed behavior of the loop can be controlled by a number of optional clauses, which may be appended to the directive:

- The **schedule** clause controls how loop iterations are scheduled to threads. The default behavior is a static blockwise schedule as shown in Fig. 1. Using the clause `schedule(static, size)`, a fixed size for the blocks may be specified. The blocks then are assigned to threads in a round-robin fashion. Thus, for example, `schedule(static, 1)` results in a cyclic schedule like in Fig. 2b. It is also possible to specify a dynamic scheduling, again with an optional block size. The schedule type `guided` uses dynamic scheduling with exponentially decreasing block sizes, which can help to improve the load balancing while keeping the overhead reasonably. Finally, the definition of the schedule type can be postponed to the runtime, in order to quickly experiment with different schedules.
- Several other clauses concern the variables used inside the loop. The `shared` and `private` clauses explicitly define variables as being shared between threads or being thread private. The loop variable and all local variables declared inside the loop body are always private. Variables declared outside the loop are shared by default, although this behavior can be changed by using the `default` clause.

  The value of a private variable is undefined at the beginning of the loop and is lost at its end. The clauses `firstprivate` and `lastprivate` allow the initialization and finalization of the value, respectively.

  The `reduction` clause is used to specify a reduction variable together with the operation used in the reduction. The reduction will then be implemented as shown in Fig. 5c.

- Finally, two clauses control the synchronisation. By default, at the end of each parallel loop, all threads synchronize using a barrier. The `nowait` clause instructs OpenMP to omit this synchronization.

  The `ordered` clause indicates that the loop contains an ordered directive, which is discussed below.

OpenMP executes parallel loops as Doall loops. However, OpenMP does not check whether the iterations are independent, thus, parallel loops in OpenMP may actually contain dependences. It is the responsibility of the programmer to ensure that the results of the parallel execution are correct. A means to achieve this is the use of synchronization constructs inside the loop. For mutually exclusive access to shared variables, OpenMP provides the `omp critical` and `omp atomic` directives, where the latter is an optimized form for statements like `x += expr`. Another synchronization directive is `omp ordered`, which ensures that the statement(s) controlled by the directive are executed in the same order as in the sequential loop. This directive can only be used in loops having an `ordered` clause and roughly results in the loop behaving like a Doacross loop. Such a loop should always use a `(static, 1)` schedule to achieve a reasonable performance.

In order to illustrate the usage of the mentioned directives, the following code examples show how the parallel reduction outlined before can be implemented using OpenMP.

(a) As Doacross loop:

```c
double s = 0;
#pragma omp parallel for \ 
ordered schedule(static, 1)
for (i=0; i<N; i++) {
    double h = f((i+0.5)/N) / N;
#pragma omp ordered
    s += h;
}
```

(b) As Doall loop:

```c
double s = 0;
#pragma omp parallel for
for (i=0; i<N; i++) {
    double h = f((i+0.5)/N) / N;
#pragma omp critical
    s += h;
}
```

(c) Local summation using the reduction clause:

```c
double s = 0;
#pragma omp parallel for \ 
reduction(+: s)
for (i=0; i<N; i++)
    s += f((i+0.5)/N) / N;
```

(d) Explicit local summation:

```c
double s = 0;
#pragma omp parallel
{
    double ls = 0;
#pragma omp for
    for (i=0; i<N; i++)
        ls += f((i+0.5)/N) / N;
#pragma omp atomic
    s += ls;
}
```

In the code examples (a) and (b), a private auxiliary variable is used in order to allow the calls to `f()` being executed in parallel. In example (d), `ls` is a private variable, since it is declared inside the parallel region. The execution behavior of these loops is exactly as outlined in Fig. 5.
Parallel Loops in Intel(R) Threading Building Blocks

The Threading Building Blocks (TBB) are a template library for ISO C++, which includes a variety of constructs supporting the parallel programming with shared memory. Among these constructs are two different kinds of Doall loops, as well as a reduction operation. Since TBB is just a library and does not require any compiler support, parallel loops must be expressed by calling a library function which receives the loop body in the form of a functor, that is, a function object.

For example, the body of the Doall loop shown in the beginning of this entry can be expressed in a class like this:

```cpp
class Body1 {
  double *a;
  double *b;
public:
  void operator()(const tbb::blocked_range<int>& r) const {
    for (int i=r.begin(); i<r.end(); i++) {
      b[i] = sin(PI*i/N);
      a[i] = a[i] + b[i];
    }
  }
  Body1(double *arg_a, double *arg_b) { a = arg_a; b = arg_b; }
};
```

The class must define an ()-operator, which must execute the loop body for the index range specified by its parameter. In addition to the class blocked_range for a contiguous one-dimensional interval, TBB also provides classes for two- and three-dimensional ranges. A parallel for-loop with the specified body is then written as a function call, for example,

```cpp
tbb::parallel_for(tbb::blocked_range<int>(0,N), Body1(a,b));
```

Here, a and b are the two array variables, which are shared between the threads executing the loop. Since the loop body is instantiated as an object, pointers to the shared data must be passed to the constructor and stored in attributes of the body class. TBB executes the loop by recursively splitting the index range into a set of subranges, for which the ()-operator will be invoked. The execution of each subrange forms a task, which is dynamically scheduled to one of the available worker threads. By default, their number is one less than the number of CPU cores, thus reserving one core for the manager thread. The way how the index range is partitioned can be controlled by passing a partitioner as an optional argument to the function. The default partitioner uses a heuristic to determine a suitable size of the subranges. It can be fine-tuned using an optional parameter in the constructor of blocked_range, which allows to specify a minimum size for the subranges. In order to achieve a good speedup, it should be chosen such that the ()-operator executes at least some 10,000 instructions. For parallel loops which are executed repeatedly, TBB provides an additional partitioner that tries to improve cache affinity.

A second kind of parallel loop construct available in TBB implements loops based on C++ iterators. Iterators are extensively used in C++ class libraries in order to iterate over all elements of a collection, for example, a linked list. Although the parallel_do loop must usually fetch the work items serially, since most iterators only support a strictly sequential traversal of a collection, the processing of these work items in the loop body may be initiated in an arbitrary order. Thus, also parallel_do implements a Doall loop where all iterations must be independent. However, the body of a parallel_do is allowed to extend the iterations space by inserting items into the collection. This is illustrated in the following example:

```cpp
class Body2 {
  public:
    void operator()(Item& item, tbb::parallel_do_feeder<Item>& feed) const {
      // may generate a new item
      Item res = processItem(item);
      if (res != NULL) {
        // add item to collection
        feed.add(res);
      }
    }
  ...

  std::list<Item> l;
  ... // initialize list
```
tbb::parallel_do(l.begin(), l.end(), Body2());

The loop iterates over all elements of a linked list and invokes the function `processItem` on each of these elements. The processing of an item may result in a new item being generated, which is added to the collection via a `parallel_do_feeder` object passed as an optional argument to the ()-operator. Since TBB makes extensive use of templates, the items may have an arbitrary data type.

Besides independent loops, TBB allows to implement parallel loops with reductions by using the `parallel_reduce` function. Reduction loops are executed by recursively splitting the index range into subranges, computing the local values for the leaf subranges, and then recursively combining these local results into the global one. Since the initialization of the local result and the combining operation depend on the concrete loop, the class defining the loop’s body must provide an additional constructor and a join operation. The following code implements the integration used as an example before:

class Sum {
public:
  double sum;
  
  void operator()(const tbb::blocked_range<int>& r) {
    for (int i=r.begin(); i<r.end(); i++)
      sum += f((i+0.5)/N) / N;
  }

  // splitting constructor
  Sum(const Sum& s, tbb::split) {
    sum = 0;
  }

  // combining operation
  void join(const Sum& s) {
    sum += s.sum;
  }

  Sum() { sum = 0; }
};

...  
Sum s;
  tbb::parallel_reduce(tbb::blocked_range<int>(0,N), s);

The dummy argument of type `tbb::split` is used to distinguish the splitting constructor from a copy constructor.

**Related Entries**
- Array Languages
- Dependence Analysis
- HPF (High Performance Fortran)
- Intel® Threading Building Blocks (TBB)
- Loop Nest Parallelization
- Metrics
- OpenMP
- Parallelization, Automatic
- Reduce and Scan
- SPMD Computational Model

**Bibliographic Notes and Further Reading**

In the 1980s, many parallel and vector supercomputers, for example, Cray X-MP or Alliant FX/8, started to offer parallel loops as a programming construct, which was implemented by their proprietary compilers. Karp [6] provides an overview of the state of the art of parallel programming at that time, including a discussion of parallel loops.

Since parallel loops are typically supported by a compiler, either by the use of directives like in OpenMP, or by autoparallelization, in-depth discussions can be found in books on parallelizing compilers. Wolfe presents a very good overview on data dependences and techniques for loop parallelization in [10]. In [7], Pacheco discusses different methods for scheduling parallel loops in detail.

Since Fortran is the predominant language in the area of numerical computations, there have been many research efforts to integrate parallel loops into this language, most of them addressing distributed memory parallel computers. The group around Ken Kennedy at Rice University developed Fortran D [5], while in parallel Hans Zima and his coworkers devised Vienna Fortran [3]. Both languages influenced the High Performance Fortran language, whose first specification appeared as a special issue of the Scientific Computing journal [4].

Programming with OpenMP is discussed in ample detail in several books [1, 2, 9]. There is also a textbook [8] about the Intel Threading Building Blocks, which
contains many examples. Additional information about OpenMP and TBB can be found in the related entries of this encyclopedia.

**Bibliography**


**LU Factorization**

- Dense Linear System Solvers
- Sparse Direct Methods