Loop Parallelism

1 Problem

How can a serial program containing a set of computationally intensive loops be transformed into an efficient parallel program?

2 Context

This pattern discusses transformations that can be made to loops and loop nests in order to achieve efficient parallel execution. These transformations enable both memory-system optimizations as well as mapping to multi-core and SIMD execution resources.

Many programs in scientific and engineering applications are expressed as operations over large datastructures. These may be simple element-wise operations on arrays or matrices, more complex loop nests with loop-carried dependencies, they may iterate over more complex data structures (e.g. edges of a graph). In many cases, the code can be re-structured so that loop iterations can be executed efficiently and in parallel on modern parallel platforms. The loop-structure in many programs written in higher-level languages may be hidden: for example, most operators in Matlab codes are implicitly element-wise, and could equivalently be written as loops over the elements of a matrix.

When an application is already implemented and functions correctly, it is usually desirable to minimize the amount of code restructuring needed to achieve acceptable speedups. Even when an application is developed from-scratch, it is usually easiest to develop a sequential version first and optimize once the application is debugged and tested. Focusing on loops in which a program performs most of its computation frequently allows for relatively localized changes to the source base, and simultaneously enables incremental parallelization: the programmer can optimize the code one loop (or loop nest) at a time, applying a small number of code transformations at a time. This approach to optimization eases testing and debugging, as the program is still complete after each incremental code change. The goal is to “evolve” a sequential program into a parallel program.

3 Forces

- Incremental parallelism: it is desirable for optimizations to be applied one-by-one, rather than all at once. This eases testing and debugging of individual optimizations.

- Amdahl’s Law: enough code must be optimized for the desired speedups to be achieved. However, it is important to select the set of loops most amenable to optimization, as
less code re-structuring will be necessary to achieve the desired speedup.

- **Memory System Performance**: simply exploiting parallelism is not usually enough. It is necessary to consider the data-movement and data-use necessitated by parallelization.

## 4 Solution

The main steps in the solution are:

- Identify the most important loops
- Analyze the loops’ dependencies and data usage
- Determine useful loop transformations
- Incrementally apply transformations, ensuring serial equivalence at each step.

Each of these topics could easily fill an entire textbook on compiler construction and code analysis. We have selected a few major points from each topic as particularly relevant to multi-core and many-core code optimization, although no such list can be complete. Interested readers should follow-up in the referenced works for a complete treatment.

Note that in code examples, arrays are represented by upper case symbols, and functions are represented by lower-case symbols.

### 4.1 Identify Important Loops

Important loops can most easily be identified via execution profiling: profiling tools (e.g. Intel VTune or the open-source gprof) record what fraction of a program’s runtime is spent in various portions of the code. gprof reports this data at function-level granularity, although more sophisticated tools may provide more data. Thus some manual analysis may be necessary to identify the most computationally intense loops. If profiling is insufficient, it is possible to manually instrument the code to provide more precise information. For example, one can capture a timestamp before and after a given loop is executed, along with the number of iterations of the loop. Additionally, the depth of nesting is important: code residing in very deeply nested loops will tend to be executed frequently. This information is sufficient to measure a loop’s computational importance: loops which execute for many (e.g. thousands or millions) of iterations and run for very long times are likely candidates for parallelization.

Given that one has collected the runtimes of all interesting loops, one should consider Amdahl’s law: it is necessary to consider enough of the code that the desired speedup is possible. For example, suppose a programmer desires a 10X overall program speedup, and expects that she can achieve a 100X speedup on parallelized loops. She then must target 90.9% of the runtime of the overall program for speedup. This is simultaneously encouraging and discouraging: 90.9% of the runtime may represent a large fraction of the source base, however it is certainly not the entire source base. Some loops, in particular those which are not parallel or otherwise difficult to optimize, may be ignored. It is important not to waste effort unnecessarily optimizing difficult code.
4.2 Analyze Dependencies and Data Usage

Loop-carried dependencies can prevent some loops from being parallelized. For example, consider the following loops: which repeatedly compute the mixed partial derivative of a 2-Dimensional signal:

for \( t = 0 \ldots T - 1 \)
\[
\text{for } i = 0 \ldots N \\
\quad \text{for } j = 0 \ldots N \\
\quad \quad A(t+1, i, j) = A(t,i+1,j+1) + A(t,i-1,j-1) - A(t,i-1,j+1) - A(t,i+1,j-1)
\]

That is, it weights the up-left and bottom-right neighbors by 1, and the up-right and bottom-left neighbors by \(-1\). In this example, there are loop-carried dependencies in the \( t \) loop: that is, data created in iteration \( t \) is used in iteration \( t + 1 \), so the two loop iterations cannot be computed in parallel. However, neither the \( i \) nor \( j \) loop carries any dependencies, and both are parallelizable.

Note that not all loop-carried dependencies prevent parallelization. In particular, reduction loops are frequently very parallel. Consider:

for \( i = 0 \ldots N \)
\[
\quad \text{sum } += f(i)
\]

This loop is a sum-reduction. The Map-Reduce pattern discusses how to implement efficient parallel reductions.

Locality and Data Re-Use is an extraordinarily important characteristic. As the rate of many computations quickly become bound by memory bandwidth, it is crucial to consider data movement. For example, consider the following loops:

for \( i = 0 \ldots N \)
\[
\quad C(i) = f(A(i), B(i))
\]
for \( i = 0 \ldots N \)
\[
\quad D(i) = f(C(i), A(i+1))
\]

In the first loop, the \( i^{th} \) element of \( A \) and \( B \) are used to compute the \( i^{th} \) element of \( C \). The second loop uses \( C(i) \) and \( A(i + 1) \) to compute \( D(i) \). Thus there is both locality (the use of near-by elements of \( A \) in the two loops) and re-use (output of the first loop is the input to the second loop). As the section 4.3 discusses, this would be a candidate for loop fusion.

Re-use is easily confused with loop-dependencies, however. For example, of the second loop from above were changed to:

for \( i = 0 \ldots N \)
\[
\quad D(i) = f(C(i+1), A(i+1))
\]
The loop fusion operation would require peeling to ensure that $C(i + 1)$ is computed before $D(i)$.

Locality and re-use can also exist within a single loop nest. For example, in the partial derivative example:

\[
\text{for } i = 0 \ldots N \\
\quad \text{for } j = 0 \ldots N \\
\quad D_{ij}(i, j) = A(i+1,j+1) + A(i-1,j-1) - A(i-1,j+1) - A(i+1,j-1)
\]

$A(i_0, j_0)$ is used in four iterations of the loop, at $(i, j) = \{(i_0 + 1, j_0 + 1), (i_0 - 1, j_0 - 1), (i_0 - 1, j_0 + 1), (i_0 + 1, j_0 - 1)\}$. Loop tiling, discussed in section 4.3 can be used to ensure that iterations are executed in a way that respects locality.

**Arithmetic Intensity** is the ratio of operations performed to bytes of data transferred. Frequently, “operations” are defined as floating-point arithmetic operations, although in some codes another definition may be appropriate. In the mixed partial derivative example:

\[
\text{for } i = 0 \ldots N \\
\quad \text{for } j = 0 \ldots N \\
\quad D_{ij}(i, j) = A(i+1,j+1) + A(i-1,j-1) - A(i-1,j+1) - A(i+1,j-1)
\]

Each iteration must read the four values of $A$, perform the addition and two subtractions, and write one value to $D_{ij}$. In double-precision floating point, where each value is an eight-byte value, this is 32 bytes of data read, 8 bytes of data written. Thus the arithmetic intensity is $\frac{3}{30}$, assuming that caches exploit no re-use. As most parallel processors can sustain at least $10 \times$ as much instruction throughput as DRAM data traffic, it is crucial to transform loops to maximize arithmetic intensity. Tiling and fusion are most often used to achieve this.

### 4.3 Determine Transformations

In no particular order, here is a list of some common transformations. In many cases, the transformations change the order in which values are computed, and the programmer must ensure that this does not change the semantics of the program. For lack of space, we will not discuss the dependence-analysis issue here. As loop optimizations can usually be applied incrementally, it is possible to test each transformation individually.

**Parallelization** is the distribution of loop iterations across different processor cores, or potentially threads within a simultaneously multi-threaded (SMT) processor core. The coding details differ substantially between target platforms, but on multi-core CPU targets, OpenMP provides a powerful set of code annotations to enable parallelization. For example, the following C code:

```c
for (i = 0; i < N; i++)
        A[i] = alpha + B[i]*C[i];
```

can be parallelized with the OpenMP parallel for directive:
#pragma omp parallel for private(i,j)
for (i = 0; i < N; i++)
{
    A[i] = alpha + B[i]*C[j];
    j = j + 100;
}

And the OpenMP compiler and runtime will distribute the iterations of the for loop across processor cores. The private(i,j) clause instructs the compiler to generate a copy of the variables i and j local to each thread.

**Stripmining** is a transformation that enables the use of vector or SIMD instructions. Given a loop of the form:

for i = 0...N
    A(i) = f(i)

The loop is transformed into a 2-deep loop nest, where the inner loop has a constant loop bound. Usually, this inner loop bound is chosen to be the vector length (or SIMD width) of the target machine:

for i = 0...N by 32
    for k = 0...32
        A(i) = f(i)

It is then possible to map the inner loop directly into vector or SIMD instructions, assuming that there are no loop dependencies that prevent it.

**Interchange** is a transformation applied to nested loops in which the order of the loops is changed. For example, a row-major computation of an array:

for i = 0...N
    for j = 0...N
        A(i,j) = f(i,j)

an interchange of the i and j loops produces:

for j = 0...N
    for i = 0...N
        A(i,j) = f(i,j)

In the interchanged code, the computation proceeds in column-major order, which may be preferable if, for example, A is stored in column-major order.

**Tiling** is a loop transformation on which many cache blocking algorithms are built, and consists of a combination of stripmining and interchange in multiply-nested loops. Frequently, a cache blocking strategy, will stripmine several nested loops, and perform interchanges to bring the strip-loops inward. Consider the multiply-nested loops of a matrix multiplication:
for $i = 0 \ldots M$
  for $j = 0 \ldots N$
    for $k = 0 \ldots K$
      \[ C(i,j) += A(i,k) * B(k,j) \]

Tiling all three loops produces:

for $ii = 0 \ldots M$ by $B$
  for $jj = 0 \ldots N$ by $B$
    for $kk = 0 \ldots K$ by $B$
      for $i = ii \ldots ii + B$
        for $j = jj \ldots jj + B$
          for $k = kk \ldots kk + B$
            \[ C(i,j) += A(i,k) * B(k,j) \]

**Fusion** multiple loops with similar or identical loop bounds can be fused into a single loop. This can reduce the instruction overhead due to iteration, as well as enable re-use of data produced in one loop and used in the second. One must be careful not to violate dependencies, however. For example, the following two loops:

for $i = 0 \ldots N$
  \[ B(i) = f(i) \]  
for $i = 0 \ldots N$
  \[ C(i) = g(B(i)) \]

can be fused into a single loop:

for $i = 0 \ldots N$
  \[ B(i) = f(i) \]
  \[ C(i) = g(B(i)) \]

If $B(i)$ is not used after this code executes, it need not be explicitly stored and memory bandwidth can be saved.

**Fission** refers to splitting a single loop into two or more loops. This transformation may enable parallelization by breaking inter-loop dependencies. For example, the following loop nest is not parallelizable due to the reference to $A(i,j-1)$:

for $i = 0 \ldots N$
  for $j = 0 \ldots N$
    \[ A(i,j) = B(i,j) + C(i,j) \]
    \[ D(i,j) = A(i,j-1) * 2; \]

However, splitting the $j$ loop produces two inner loops which are each parallelizable, albeit at a potentially fine granularity:
for $i = 0 \ldots N$
  for $j = 0 \ldots N$
    $A(i,j) = B(i,j) + C(i,j)$
  for $j = 0 \ldots N$
    $D(i,j) = A(i,j-1) * 2;$

**Peeling** is removing “special case” iterations from a loop to enable the loop code to be optimized more completely. The loop body is replicated for each special cases, and the code left inside the loop is simplified so as not to deal with corner cases any more.

**Unrolling** Consists of replicating the body of a loop multiple times to reduce loop iteration overhead:

for $i = 0 \ldots N$
  $A(i) = B(i) + C(i)$

Unrolled 4 times, becomes:

for $i = 0 \ldots N$ by 4
  $A(i) = B(i) + C(i)$
  $A(i+1) = B(i+1) + C(i+1)$
  $A(i+2) = B(i+2) + C(i+2)$
  $A(i+3) = B(i+3) + C(i+3)$

**Unroll-And-Jam** unrolls an outer loop of a loop-nest, and fuses the copies of the inner loop. This can frequently produce increased data re-use with the inner loop:

for $i = 0 \ldots N$
  for $j = 0 \ldots M$
    $A(i) = A(i) + B(j)$

Unrolled-and-Jammed 2X, exploits re-use of $B(j)$:

for $i = 0 \ldots N$ by 2
  for $j = 0 \ldots M$
    $A(i) = A(i) + B(j)$
    $A(i+1) = A(i+1) + B(j)$

4.4 **Incrementally Optimize**

The application of many transformations to many loops may be necessary before efficient performance is achieved. However, these transformations preserve the functionality of the code, and do not change the output of any individual code module. Thus they can be applied independently, one-at-a-time, and independently tested and debugged.
5 Examples

5.1 2-Dimensional Convolution

Convolution is an important kernel in many application fields, and computes the response of an input signal (below, $X(i,j)$) to a filter function (below, $f(i,j)$). Frequently, the size of the filter is much smaller than the size of the input signal – for example, the signal may be a $1024 \times 768$ pixel digital image, while typical sizes of the filter may be $11 \times 11$. In such a case, it is usually inefficient to compute the filter response via Fourier Transforms, and direct convolution is preferable.

Na"ively expressed, the convolution of an $N \times N$ image with a $2r + 1 \times 2r + 1$ filter is:

$$
\text{for (i = 0; i < N; i++)} \\
\quad \text{for (j = 0; j < N; j++)} \\
\quad \quad Y_{i,j} = 0; \\
\quad \quad \text{for (di = -r; di <= r; di++)} \\
\quad \quad \quad \text{for (dj = -r; dj <= r; dj++)} \\
\quad \quad \quad \quad Y_{i,j} = Y_{i,j} + f(di,dj) \times X[i+di + N*(j+dj)]; \\
\quad \quad \quad Y[i + N*j] = Y_{i,j};
$$

Suppose that we are targeting a multi-core CPU with 16-way SIMD instruction set extensions. Note that our arrays are column-major. Thus we want to exploit parallelism at two different levels. To expose the SIMD parallelism, we want to stripmine the $i$ loop (since it is the unit-stride direction), and to exploit multi-core parallelism, we want to parallelize either the $j$ or the $i$ loop. We did not choose to exploit parallelism in the $di$ an $dj$ loops, as those are a reduction of very small size. In general, reductions are only worth parallelizing at large scale.

The transformations necessary are as follows: We wish to strip-mine the $i$ loop to expose vector parallelism, and to parallelize only the $j$ loop. This is equivalent to a geometric decomposition of the $X$ and $Y$ arrays by columns. We interchange the strip-mined $i$ loop inside of the $di$ and $dj$ loops, and then translate it into SIMD intrinsics. Finally, we must interchange the $i$ and $j$ loops to enable parallelization of the entire computation. If the $j$ loop is left inside the $i$ loop, there will be an unnecessary implicit synchronization after the computation of each row. The variable $Y_{i,j}$ must also be previously declared as a SIMD variable. The resulting C code is:

```c
#pragma omp parallel for private(ii,i,j,di,dj,Y_{i,j})
for (j = 0; j < N; j++) \\
\quad for (i = 0; i < N; i = i + 16) \\
\quad \quad simd_set_16(Y_{i,j}, 0); \\
\quad for (di = -r; di <= r; di++) \\
\quad \quad for (dj = -r; dj <= r; dj++) \\
\quad \quad \quad X_{idi,jdj} = simd_load(X[i+di + N*(j+dj)]); \\
\quad \quad \quad Y_{i,j} = simd_add (Y_{i,j}, simd_mul_scalar (X_{idi,jdj}, f(di,dj)); \\
\quad \quad simd_store(Y[i + N*j] = Y_{i,j};
```
6 Known Uses

- Rivera Tiling for stencil codes.
- Cache blocking for BLAS-3 operations.
- Auto-vectorization SIMD instruction sets.

7 References

Allen and Kennedy: Optimizing Compilers for Modern Architectures (textbook)
OpenMP: openmp.org

8 Author

Mark Murphy