Optimizing for Parallelism and Locality

MATERIALS FROM CHAPTER 11 OF THE NEW DRAGON BOOK

MICHAEL WOLLOWSKI

Introduction

Many scientific applications process information stored in matrices:

- Weather prediction
- Climate research
- Protein folding
- Neural network training

Challenges

Obvious: Ensure validity of data

How to identify and divide units of computation that can run on separate processors

Ensure speedup by minimizing inter-processor communication

Data locality (processor often accesses data used recently) aids in parallelism

Parallelism and locality are treated together because they go hand in hand

To be specific, we want to optimize data for locality to use parallelism

Very simple Example

Consider:

- `for (i = 0; i < 10, i++) z[i] = 0;`

Iterations of the loop write to different locations.

Different processors can execute different locations concurrently.

If there were to be an additional statement in the mix, such as `z[j] = 1`, then we need to determine whether i and j could be the same.

Focus

We will focus on data in arrays that is accessed with simple, regular patterns

Affine array access:

- If *i* and *j* are indices, then *Z*[*i*][*j*] and *Z*[*i*][*j+1*] are affine accesses
- A function with one of more variables, x_1, x_2, …, x_n is *affine* if it can be expressed as a sum of a constant plus constant multiples of the variable.
- Example: $c_0 + c_1x_1 + c_2x_2 + \ldots + c_nx_n$ where $c_0 \ldots c_n$ are constants

Symmetric Multiprocessors

Symmetric multiprocessors share the same address space.

To communicate, a processor can write to a memory location which is then read by another processor.

Several processors are allowed to keep copies of the same cache line at the same time, provided they only read from it.

If a processors writes to the cache, copies from all other caches are removed.

Recall that memory access is expensive.

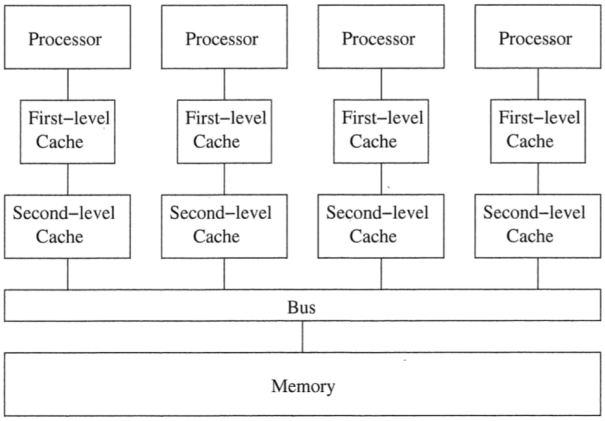


Figure 11.1: The symmetric multi-processor architecture

Distributed Memory Machines

Shared memory did not scale well beyond 10s of processors.

Bus could not keep up.

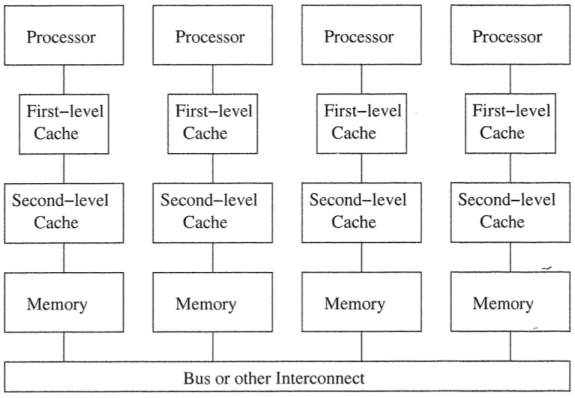


Figure 11.2: Distributed memory machines

Loop-level Parallelism

Consider:

```
for (i = 0; i < n; i++) {
  z[i] = x[i] – y[i];
  z[i] = z[i] * z[i];
}
```

Parallelizable, because each iteration accesses a different set of data.

Convert to:

```
b = celi(n/M);
for (i = b*p; i < min(n,b*(p+1)); i++) {
   z[i] = x[i] – y[i];
   z[i] = z[i] * z[i];
}
```

and divide the iterations evenly among he processors.

The *p*th processor is given the *p*th swath of iterations to execute.

Data Locality

Consider:

```
for (i = 0; i < n; i++) z[i] = x[i] – y[i];
for (i = 0; i < n; i++) z[i] = z[i] * z[i];
```

These two loops are not as advantageous as the prior, single loop.
This is because the prior loop may take advantage of data locality.
In other words, in the prior loop the data used for squaring is still in the registers.
There is also just one write.

Introduction to Affine Transform Theory

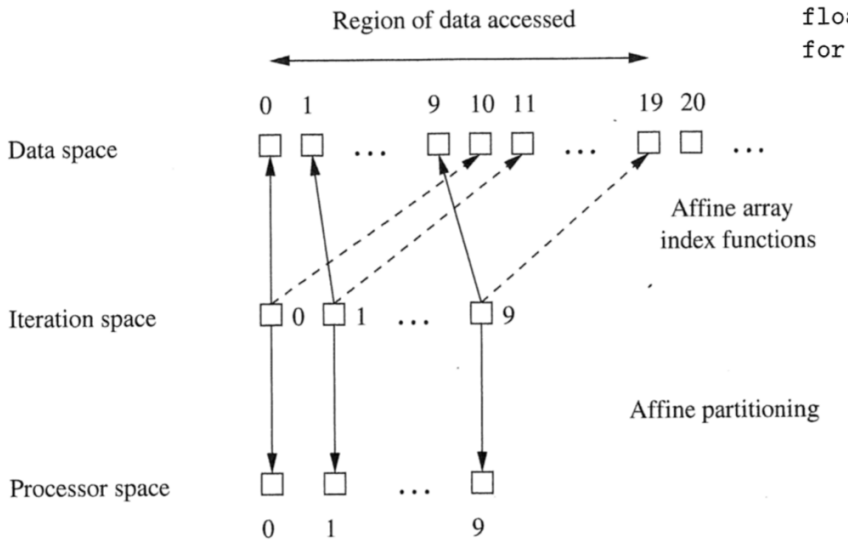
Data space. The set of array elements accessed.

Iteration Space. The set of combinations of values taken on by the loop index.

Processor space. The set of processors in the system.

Consider:

```
float z[100];
for (i = 0; i < 10; i++) z[i+10] = z[i];
```




```
float Z[100];
for (i = 0; i < 10; i++)
    Z[i+10] = Z[i];
```

Figure 11.4: Iteration, data, and processor space for Example 11.4

Matrix Multiply: An In-depth Example

Consider:

```
for (i = 0; i < n; i++)
   for (j = 0; j < n; j++) {
      Z[i,j] = 0.0;
      for (k = 0; k < n; k++)
         Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
   }
```

Figure 11.5: The basic matrix-multiplication algorithm

Serial Execution of Matrix Multiplication

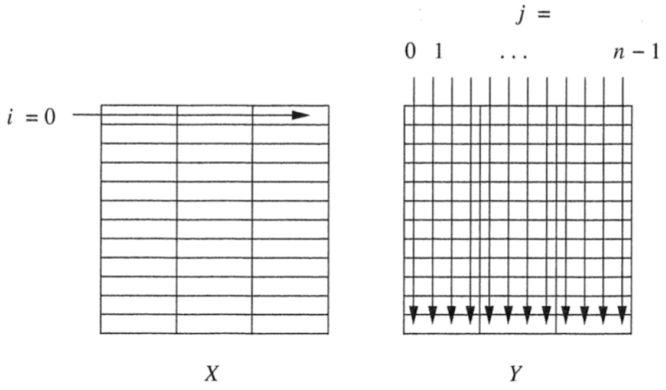


Figure 11.6: The data access pattern in matrix multiply

Optimizations

Assign different rows to different processors.

Decreases computation time

However, increases communication time, i.e. time to transfer data

Places a bottleneck on the bus

Can slow down computation rather than speed it up

Optimizations to Support Data Locality

Objective, reuse data while it is still in the cache.

The matrix multiplication algorithm has poor data locality and as such many cache misses

Change data layout.

- One could change the layout of matrices to either row or column major order.
- This would help since arrays are stored in pages and entire pages are loaded into the cache.
- However, this does not always work as matrices may be accessed in different orders during the same algorithm.

Optimizations to Support Data Locality

Blocking.

- Instead of computing the result a row or a column at a time, we divide the matrix into submatrices, or blocks.
- Reorder operations so an entire block is used over a short period of time.
- Objective: decrease cache misses.
- Ensure B is chosen so that all of the data fits into the cache.

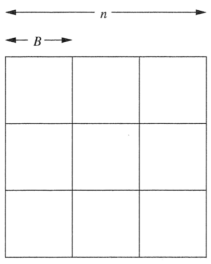


Figure 11.7: A matrix divided into blocks of side B

Optimization

```
1)  for (ii = 0; ii < n; ii = ii+B)
2)      for (jj = 0; jj < n; jj = jj+B)
3)          for (kk = 0; kk < n; kk = kk+B)
4)              for (i = ii; i < ii+B; i++)
5)                  for (j = jj; j < jj+B; j++)
6)                      for (k = kk; k < kk+B; k++)
7)                          Z[i,j] = Z[i,j] + X[i,k]*Y[k,j];
```

Figure 11.8: Matrix multiplication with blocking

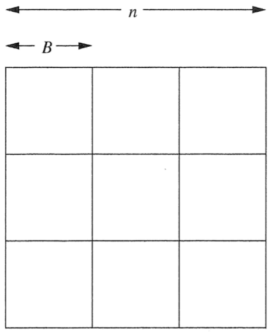


Figure 11.7: A matrix divided into blocks of side B