APPROACHES TO PARALLEL PROGRAMMING

Techniques for programming parallel computers can be divided into three rough categories: parallelizing compilers, parallel programming languages, and parallel libraries. This section considers each approach in turn.

1. Parallelizing Compilers:

The concept of a parallelizing compiler is an attractive one. The idea is that programmers will write their programs using a traditional language such as C or Fortran, and the compiler will be responsible for managing the parallel programming challenges described in the previous section. Such a tool is ideal because it allows programmers to express code in a familiar, traditional manner, leaving the challenges related to parallelism to the compiler. Examples of parallelizing compilers include SUIF, KAP, and the Cray MTA compiler [HAA 96, KLS94, Ter99]

Listing 1.1: Sequential C Matrix Multiplication

```
for (i=0; i<m; i++) {
  for (k=0; k<o; k++) {
    C[i][k] = 0;
  }
}
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    for (k=0; k<o; k++) {
      C[i][k] += A[i][j] * B[j][k];
    }
  }
}
```

The primary challenge to automatic parallelization is that converting sequential programs to parallel ones is an entirely non-trivial task. As motivation, let us return to the example of matrix multiplication. Written in C, a sequential version of this computation might appear as in Listing 1.1.

Many parallelizing compilers, including those named above, take an intermediate approach in which programmers add directives to their codes to provide the compiler with information to aid it in the task of parallelizing the code. The more of these that need to be relied upon, the more this approach resembles a new programming language rather than a parallelizing compiler, so further discussion of this approach is deferred to the next section.

2 Parallel Programming Languages:

A second approach to parallel programming is the design and implementation of parallel programming languages. These are languages designed to support parallel computing better than sequential languages, though many of them are based on traditional languages in the hope that existing code bases can be reused. This dissertation categorizes parallel languages as being either global-view or local-view.

Global-view Languages:

Global-view languages are those in which the programmer specifies the behavior of their algorithm as a whole, largely ignoring the fact that multiple processors will be used to implement the program. The compiler is therefore responsible for managing all of the parallel implementation details, including data distribution and communication. Many global-view languages are rather unique, providing language-level concepts that are tailored specifically for parallel computing. The ZPL language and its regions form one such example. Other global-view languages include the directive-based variations of traditional programming languages used by parallelizing compilers, since the annotated sequential programs are global descriptions of the algorithm with no reference to individual processors. As a simple example of a directive-based global-view language, consider the pseudocode implementation of the SUMMA algorithm in Listing 1.2. This is essentially a sequential description of the SUMMA algorithm with some comments (directives) that indicate how each array should be distributed between processors

The primary advantage to global-view languages is that they allow the programmer to focus on the algorithm at hand rather than the details of the parallel implementation. For example, in the code above, the programmer writes the loops using the array's global bounds. The task of transforming them into loops that will cause each processor to iterate over its local data is left to the compiler.

This convenience can also be a liability for global-view languages. If a language or compiler does not provide sufficient feedback about how programs will be implemented, knowledgeable programmers may be unable to achieve the parallel implementations that they desire. For example, in the SUMMA code of Listing 1.2, programmers might like to be assured that an efficient broadcast mechanism will be used to implement the assignments to ColA and RowB, so that the assignment to C will be completely local. Whether or not they have such assurance depends on the definition of the global language being used.

Local-view Languages :

Local-view languages are those in which the implementor is responsible for specifying the program's behavior on a per-processor basis. Thus, details such as communication, data distribution, and load balancing must be handled explicitly by the programmer. A localview implementation of the SUMMA algorithm might appear as shown in Listing 1.3. The chief advantage of local-view languages is that users have complete control over the parallel

implementation of their programs, allowing them to implement any parallel algorithm that they can imagine. The drawback to these approaches is that managing the details of a parallel program can become a painstaking venture very quickly. This contrast can be seen even in short programs such as the implementation of SUMMA in Listing 1.3, especially considering that the implementations of its Broadcast...(), IOwn...(), and GlobToLoc...() routines have been omitted for brevity. The magnitude of these details are such that they tend to make programs written in local-view languages much more difficult to maintain and debug.

3 Parallel Libraries:

Parallel libraries are the third approach to parallel computing considered here. These are simply libraries designed to ease the task of utilizing a parallel computer. Once again, we categorize these as global-view or local-view approaches.

Global-view Libraries:

Global-view libraries, like their language counterparts, are those in which the programmer is largely kept blissfully unaware of the fact that multiple processors are involved. As a result, the vast majority of these libraries tend to support high-level numerical operations such as matrix multiplications or solving linear equations. The number of these libraries is overwhelming, but a few notable examplesinclude the NAG Parallel Library, ScaLAPACK, and PLAPACK [NAG00, BCC 97, vdG97]. The advantage to using a global-view library is that the supported routines are typically well-tuned to take full advantage of a parallel machine's processing power. To achieve similar performance using a parallel language tends to require more effort than most programmers are willing to make. The disadvantages to global-view libraries are standard ones for any library-based approach to computation. Libraries support a fixed interface, limiting their generality as compared to programming languages. Libraries can either be small and extremely specialpurpose or they can be wide, either in terms of the number of routines exported or the number of parameters passed to each routine [GL00]. For these reasons, libraries are a useful tool, but often not as satisfying for expressing general computation as a programming language.

Local-view Libraries:

Like languages, libraries may also be local-view. For our purposes, local-view libraries are those that aid in the support of processor-level operations such as communication between processors. Local-view libraries can be evaluated much like local-view languages: they give the programmer a great deal of explicit low-level control over a parallel machine, but by nature this requires the explicit management of many painstaking details. Notable examples include the MPI and SHMEM libraries [Mes94, BK94].