Chapter 1 Motivation, Background, and Key Concepts

> *"'If you build it, they will come.' And so we built them. Multiprocessor workstations, massively parallel supercomputers, a cluster in every department ... and they haven't come. ... The computer industry is ready to flood the market with hardware that will only run at full speed with parallel programs. But who will write these programs?"* Mattson et al, 2004 [1]

> *"We stand at the threshold of a many core world. The hardware community is ready to cross this threshold. The parallel software community is not."* Mattson, 2011 [2]

1.1 Introduction

Humans are an insatiable species. Fast is never fast enough, and enough is never enough. We want to do things faster and faster, and we want to accomplish more and more. People invented computers to solve problems faster than they could be solved by hand, and people have made the individual processors faster and faster, trying all the while to reduce their power consumption, their cost, and their size. In 1965, Gordon Moore predicted that the number of transistors on an integrated circuit would double every eighteen to twenty-four months, and so it has [3].

Be that as it may, these individual processors are not capable of solving the most significant computational problems, nor will they ever be, because of their inherent complexity. It is only natural then, that someone came along with the idea of putting multiple processors to work to solve a single problem, and thus was born the idea of parallel computing.

The idea of doing things in parallel on a computer made its first major appearance in the early supercomputers of the 1970s. The Cray-1 supercomputer, designed by Seymour Cray and installed in the *Los Alamos National Laboratory* in 1976, harnessed parallelism within the processor itself, using pipelining and vector operations. But supercomputers were not really parallel computers back then. They were essentially just super-fast, single-CPU machines.

Using multiple processors together became feasible when the microprocessor was invented, as this made it possible to create machines with many cheap, small processors working together. But connecting multiple processors to each other by itself does little good unless we know how to design algorithms that can use them correctly and efficiently, and while this may seem like it is easy enough to do, it is not.

The purpose of these notes is to provide an introduction to parallel algorithms, parallel programming, and their analysis. They are intended to accompany the very fine textbook written by Michael J. Quinn, *Parallel Programming in C with MPI and OpenMPI* [4]. They begin with an overview of the basic concepts, followed by a bit about parallel computer architectures. They then cover parallel design methodology, and use that methodology in the design of various parallel algorithms to solve problems such as matrix-vector multiplication, Monte Carlo methods, solving linear systems, classifying documents, sorting, and finite difference methods.

1.2 Basic Concepts

Definition 1. ***Parallel computing*** is the use of parallel computers to reduce the time needed to solve a single computational problem.

This begs the question, of course – what is a parallel computer? We will define them as follows.

Definition 2. A ***parallel computer*** is a computer containing more than one processor. Parallel computers can be categorized as either multicomputers or centralized multiprocessors.

Definition 3. A ***multicomputer*** is a computer that contains two or more computers connected by an interconnection network.

Definition 4. A ***centralized multiprocessor***, also known as a ***symmetrical multiprocessor*** (***SMP***), is a computer in which the processors share access to a single, global memory. A ***multi-core processor*** is a particular type of multiprocessor in which the individual processors (called "***cores***") are in a single integrated circuit.

Definition 5. ***Parallel programming*** is programming in a language that allows one to explicitly indicate how the different parts of the computation can be executed concurrently by different processors.

1.3 Scientific Problems and Methodology: Need for Parallel Computing

The nature of scientific exploration and discovery has changed dramatically since the dawn of the age of computing. Computers are now used to *discover* new scientific results, rather than just to validate or refine existing ones. In fact, in 2005, the *President's Information Technology Advisory Committee* (*PITAC*) issued a report in which they stated [5],

> "Together with theory and experimentation, computational science now constitutes the "third pillar" of scientific inquiry, enabling researchers to build and test models of complex phenomena – such as multi-century climate shifts, multidimensional flight stresses on aircraft, and stellar explosions – that cannot be replicated in the laboratory, and to manage huge volumes of data rapidly and economically."

Parallel computing makes it possible to speed up computations a hundred, a thousand, or even tens of thousands times. There are those who will argue that this does not make it possible to solve problems that are untractable, i.e., NP-hard problems, as it only allows solving negligibly larger instances of such problems. While this is theoretically true, such statements ignore the practical aspect of these problems, as is best characterized by the following statement from the PITAC report [5]:

> "The practical difference between obtaining results in hours, rather than weeks or years, is substantial – it qualitatively changes the range of studies one can conduct. For example, climate change studies, which simulate thousands of Earth years, are feasible only if the time to simulate a year of climate is a few hours."

This is just one of many examples of how the ability to solve a problem in hours or days rather than years can make a qualitative difference in scientific accomplishment. The Human Genome Project, started in 1990 and completed in 2001, was the first example, and it made scientists realize that computational science could be used to a far greater extent than had been imagined before.

The Human Genome Project was one of many ***grand challenge problems*** identified back then. In 1989, over one hundred scientists from major universities, national laboratories, and industrial research centers gathered on the Hawaiian island of Molokai to discuss and identify the need for greater computational power to solve what they called the "grand challenge" problems of science [6]. These were categorized as

- Quantum chemistry, statistical mechanics, and relativistic physics
- Cosmology and astrophysics
- Computational fluid dynamics and turbulence
- Materials design and superconductivity

- Biology, pharmacology, genome sequencing, genetic engineering, protein folding, enzyme activity, and cell modeling
- Medicine, and modeling of human organs and bones, and
- Global weather and environmental modeling.

Some of these projects were incorporated into the 1992 "Blue Book", a publication of the Office of Science and Technology Policy, an agency of the U.S. government, describing a strategic initiative to develop the computational resources to solve these problems in the long term [7]. Since then many more grand challenges have been identified, and they all require massive amounts of computation to solve. Without parallel computing, this is not possible.

1.4 The Landscape of Parallel Computers

Parallel computers have existed since the 1970's. Their cost was greatly diminished as the result of *Very Large Scale Integration* (*VLSI*) technology, which made it possible to reduce the chip count, thereby making reliable parallel computers within the financial reach of many more customers. Many of the parallel computers and the companies that built them no longer exist. Many experimental machines were invented and developed, many ideas tried, and many abandoned. Government projects partly spurred their development, among them the most important being the U.S. Department of Energy's ***Accelerated Strategic Computing Initiative*** (*ASCI*)[1], which resulted in the installation of a series of extremely powerful parallel computers named *ASCI Red*, *Blue Pacific*, *Blue Mountain*, *White*, *Q*, and *Purple*. These massively powerful computers were built primarily for the simulation of nuclear testing.

Commercial development has led to a vast array of parallel computers, from those that sit on the average person's desktop, containing anywhere from two to eight cores, to those installed at high performance computing centers around the world, ranging from small centers such as the *CUNY High Performance Computing Center*, which has a collection of five different parallel computing systems, ranging from 24 up to 1284 cores [8], to more powerful centers, such as the *Texas Advanced Computing Center at the University of Texas at Austin*, which has systems with over 100,000 cores [9]. In addition, commercial development has led to the creation of a class of parallel computers that harness large arrays of graphical processing units (GPUs), mostly developed by *NVidia Inc.* These GPU-based parallel computers range in size from just a few dozen GPU cores, to many thousands of GPU cores.

There are too many commercial parallel computers to list them all. One can find a long list of centralized multiprocessors at the Wikipedia page `http://en.wikipedia.org/wiki/Multi-core_processor`. To get a sense of the current state of massively parallel computing, one can visit the ***TOP500*** project website (`http://www.top500.org`.) The *TOP500* project ranks and details the 500 most powerful non-distributed computer systems in the world. The project publishes an updated list of these supercomputers twice a year, in June and November. These are generally parallel computers of various architectures. The November 2014 list can be found at `http://www.top500.org/list/2014/11/`.

To give you an idea of the scale of these computers and their processing capabilities, this is a short list:

- *Tianhe-2* is a 33.86 ***petaflops***[2] supercomputer located in *Sun Yat-sen University*, Guangzhou, China. In 2014 it was the world's fastest supercomputer according to the *TOP500* list. It has 16,000 computer nodes, each comprising two *Intel Ivy Bridge Xeon* processors and three *Xeon Phi* chips, for a total of 3,120,000 cores.
- *Cielo* is a supercomputer located at *Los Alamos National Laboratory* in New Mexico, USA built by *Cray Inc.* It has over 143,104 cores and a theoretical peak performance of 1,374 teraflops (1.374 petaflops.)
- *Titan* is a supercomputer built by *Cray* at *Oak Ridge National Laboratory* for use in a variety of science projects. *Titan* uses graphics processing units in addition to conventional central processing units. It

[1] Subsequently renamed the *Advanced Simulation and Computing Program* (*ASCP*).
[2] One petaflop is 1,000 teraflops, or 10^{15} floating-point operations per second.

is the first such hybrid to perform over 10 petaflops. *Titan* has 18,688 CPUs paired with an equal number of GPUs to perform at a theoretical peak of 27 petaflops.

- *IBM Sequoia* is a petascale *Blue Gene/Q* supercomputer constructed by *IBM* for the *National Nuclear Security Administration* as part of the *Advanced Simulation and Computing Program*. It contains 1,572,864 processor cores and has achieved a performance of 16.32 petaflops on some benchmarks.

1.5 Identifying Parallelism

Designing a parallel program to solve a given problem is only possible if the parallelism in the problem's solution can be identified. There are a few different ways to identify this parallelism.

1.5.1 Data Dependence Graphs

One way to discover the parallelism in any activity, not just a computation, is by its ***data dependence graph***. A data dependence graph is a ***directed graph*** $G =< V, E >$ in which each vertex represents a task to be completed, and an edge from vertex s to vertex t exists if and only if task s must be completed before task t can sbe started. When a graph has an edge (s, t), we say that t depends on s. (For those unfamiliar with directed graphs, see *Appendix D.*)

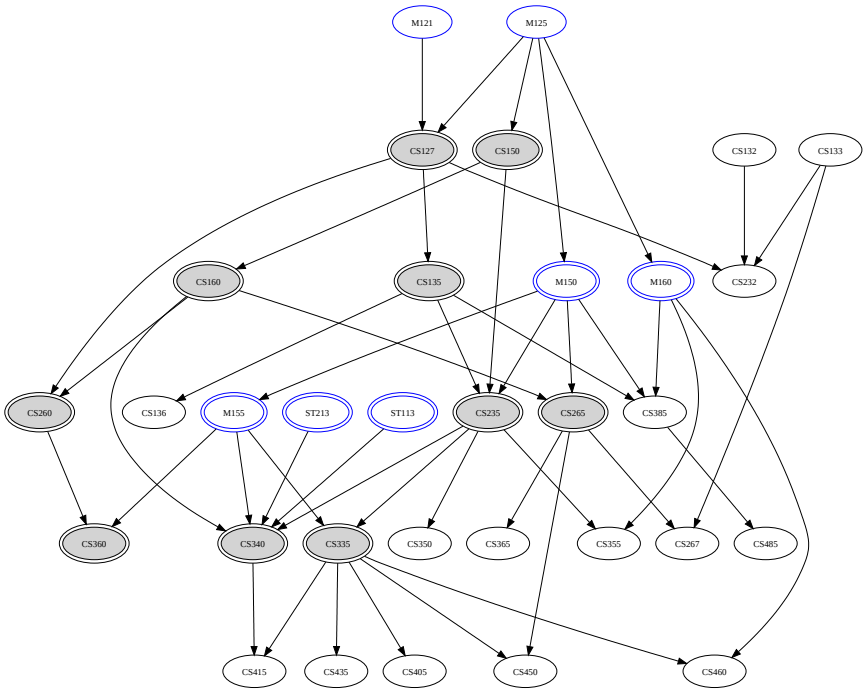


Figure 1.1: A data dependence graph. In this graph, the vertices represent courses offered in the Computer Science Department at Hunter College. Vertices with double-lines are required. In this context, data dependence indicates which course is a prerequisite to which other course.

An example of a data dependence graph is given in Figure 1.5. The overall mission is to complete a degree in computer science at Hunter College, and the graph shows the dependencies among the courses offered in the department, as well as those outside of the department that are required. Highlighted courses are required courses in the department; those with doubled circumferences are required to graduate. The tasks, a.k.a. courses, are aligned in horizontal rows to show which should be taken earlier than others in order to complete the mission in the least time. As such, the graph helps identify which tasks (courses to be completed) can

be done in parallel. It also shows the minimum amount of time necessary to complete the mission, if there were no constraints limiting how many tasks could be carried out in parallel (such as time to sleep, or pay the bills), which in this case is six semesters.

1.5.2 Data Parallelism

A data dependence graph exhibits ***data parallelism*** when it contains instances of a task that applies the same sequence of operations to different elements of a data set. The program whose data dependence graph is displayed in Figure 1.2 exhibits ***coarse-grained data parallelism***, because a single function, `sort()`, is applied to different data items in parallel. Data parallelism is coarse-grained when the parallel task consists of a many of instructions.

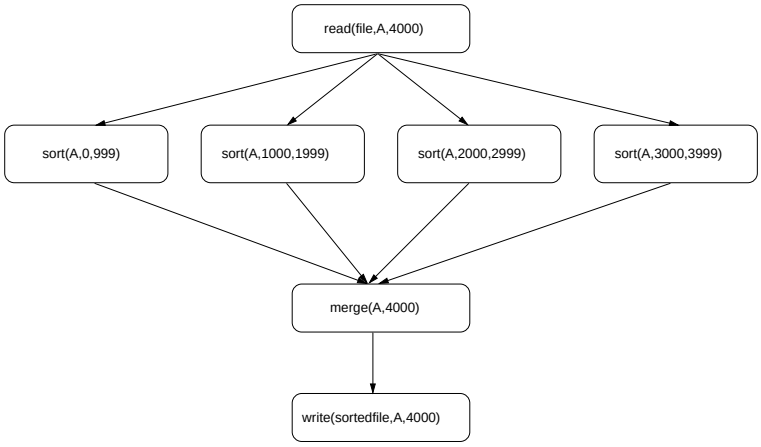


Figure 1.2: An example of data parallelism in a data dependence graph. An array is read from a file, then a single sort function is applied to different parts of it, after which the sorted sub-arrays are merged and output. The four instances of the sort function are examples of the same task applied to different data elements.

Fine-grained data parallelism exists when the operation consists of at most a few instructions, as in the following code fragment

```
for ( i = 0; i < 4000; i++ )
    A[i]++;
```

Here, the increment operation is being carried out on 4000 array elements. Its data dependence graph, in Figure 1.3, has no edges. It is just a collection of vertices (nodes), none of which depend upon any other.

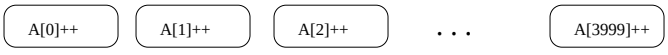
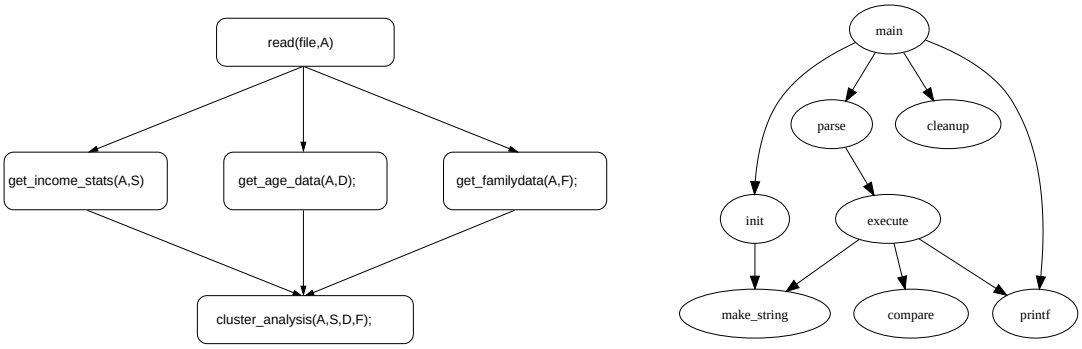


Figure 1.3: Fine-grained data parallelism.

1.5.3 Functional Parallelism

A data dependence graph exhibits ***functional parallelism*** if it contains independent tasks that apply different operations to possibly different data elements. Figure 1.4 illustrates two common types of problems in which functional parallelism can arise. In 1.4a, an array of data items, such as census data, is read from a file and distributed to three different tasks, which get statistics on income, age, and family data respectively, after which a cluster analysis is performed on the data. The three tasks can be performed in parallel because

they do not modify the input data and do not depend on each other's results. In 1.4b, there are several instances of functional parallelism:the tasks labeled `make_string`, `compare`, and `printf` can be executed in parallel, as can `parse` and `cleanup`, and `init` and `execute`.



(a) Census analyzer. (b) Software subsystem.

Figure 1.4: Two examples of functional parallelism in a data dependence graph.

1.5.4 Pipelining

When a data dependence graph is nothing but a simple path, with no branching at all, as in Figure 1.5, then there is no inherent parallelism in it, given that it only needs to be applied to a single problem instance, because no two tasks can be performed at the same time. However, if the operations or tasks in the graph are applied to multiple instances of the problem, then it is possible to overlap the sequences of tasks in parallel with each other on the different problem instances. This is known as a ***pipelined computation***.

If you are familiar with how a factory assembly line works, then you already know how pipelining works. Imagine that some factory that makes widgets has a very long conveyor belt, perhaps one hundred meters long. Along the belt are ten workstations where different parts are attached to a widget. A widget frame is placed at the start of the belt, and each time it reaches a new workstation, a machine or a person there adds a new part to it. By the time it rolls off of the conveyor belt, after workstation #10, it is complete and ready to be inspected. Suppose that each step takes an equal amount of time, say 5 minutes. Then it takes 50 minutes to assemble one widget, but if 1000 widgets are assembled each day, then once the assembly line is fully loaded, a widget is produced every 5 minutes, so the 1000 widgets are assembled in 5045 minutes, for an average of 5.045 minutes per widget. (Assembly of the k^{th} widget starts at time $5(k-1)$ and ends at time $5(k-1)+50 = 5k+45$.)

Pipelining can be applied to software as well. A loop such as

```
sum = 0;
for ( i = 0; i < 4; i++ )
    sum = sum + a[i];
```

that computes the sum of the elements in an array, has no data parallelism in it because of the dependence of the value of the sum in iteration k on its value in iteration $k-1$. However, if many arrays will be supplied to this loop, perhaps because it is nested in an outer loop, then the loop can be ***unrolled***. Unrolling a loop

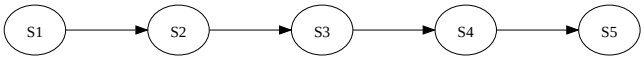


Figure 1.5: A data dependence graph exhibiting no parallelism. The tasks must be performed one after the other, from left to right.

means creating a sequence of instructions whose length is the number of loop iterations, using temporary variables to store interim results. This is bext explained by the example. The above loop is unrolled into the four instructions:

```
sum0 = a[0];
sum1 = sum0 + a[1];
sum2 = sum1 + a[2];
sum3 = sum2 + a[3];
```

Each instruction depends on the previous one, but the sequence can be turned into a pipeline, as shown in Figure 1.6. We can think of each operator as a task with two inputs and a single output. The output in the Figure is shown as being fed into the next stage of the pipeline as well as being available for access (the upward arrows) in case we want to read them outside of the pipeline.

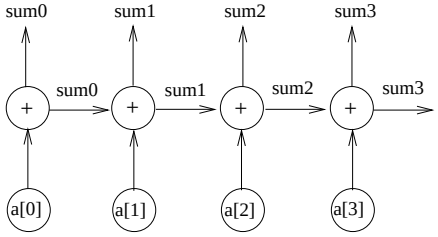


Figure 1.6: Summation pipeline for an unrolled loop.

1.6 Example: Data Clustering in Parallel

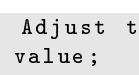
Data clustering is the classification of patterns such as observations, data items, or feature vectors into groups called ***clusters***. It is useful in several exploratory pattern-analysis, grouping, decision-making, and machine-learning contexts, including data mining, document retrieval, image segmentation, and pattern classification. Since the advent of the World Wide Web, vast amounts of data are being collected in an amazing assortment of subjects, across all disciplines, and discovering information in that data can only be done using computers. Clustering is one of the first steps in exploratory data analysis. However, clustering is a difficult problem combinatorially, and it is a good problem for parallelization.

We will use clustering as a simple example to demonstrate the discovery of data and functional parallelism. Given is a collection of N text documents. Each document is analyzed to determine how well it covers D different topics, and each document is placed into exactly one of K possible clusters. Each cluster consists of documents that are similar with respect to all of the D topics.

We use Quinn's sequential data clustering outline as the starting point [4]. It is a k-means partitional algorithm. It starts with a random initial partition and keeps reassigning the patterns to clusters based on the similarity between the pattern and the cluster centers until a convergence criterion is met:

Listing 1.1: Clustering Algorithm

```
1 Input N documents;
2 For each of the N documents generate a D-dimensional vector indicating how well it
      covers the D different topics;
3 Choose the K initial cluster centers using a random sample;
4 Repeat the following steps for I iterations or until the performance function
    converges, whichever comes first:
5       (a) For each of the N documents, find the closest center and compute its
            contribution to the performance function;
```



```
6       (b) Adjust the K cluster centers to try to improve the performance function
            value;
7   Output the K centers;
```

Our goal is not to understand clustering but to use it to demonstrate the formation and application of the data dependence graph. In order to expose the data parallelism in an algorithm, sometimes it is necessary to expand the data on which it acts into individual elements, as we showed in Figure 1.3 above. That is what Quinn does in this example, and we do the same. From the above description, you can see that there are six tasks, because we must separate steps 4a and 4b. (Step 4b can only be performed after 4a has been completed for all of the documents.) The data dependence graph in Figure 1.7 has a rectangle for each of steps 1, 2, 3, and 4a, and separate nodes for tasks 4b and 5. The central node in the graph, labeled "dummy task", is essentially representing the control expression in the loop of Step 4.

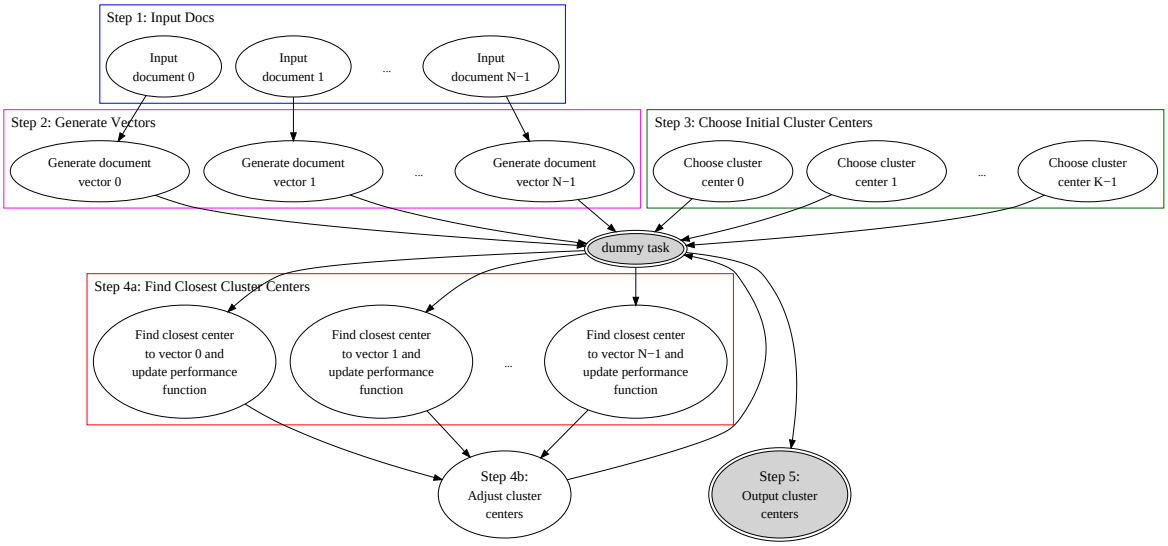


Figure 1.7: Data dependence graph for clustering algorithm from Listing 1.1.

The rectangular groups are different functional groups. Within each functional group we see that the same action is applied to multiple data elements. Thus, within each rectangle there is data parallelism. Tasks that can be executed in parallel include Steps 1 and 3, or Step 2 and 3. All of these must be completed before Step 4, but there is no dependence between Step 3 and either of Steps 1 or 2.

1.7 Paths towards Parallel Programming

In 1988, McGraw and Axelrod [10] identified four different approaches towards the creation of parallel software:

1. extending a compiler to translate a sequential program into parallel code;
2. extending a sequential language with new instructions that allow the expression of parallelism;
3. creating a new parallel language layer "on top of" an existing sequential language;
4. defining a new parallel language and compiler system.

We briefly consider these different strategies.

1.7.1 Extending a Compiler

The basic idea is that a compiler can be modified to detect the inherent parallelism in a program and output parallel code when it finds it. Compilers that do this are called ***parallelizing compilers***. The major advantage of this approach is that programmer would not have to do anything to create parallel programs, and existing sequential programs could be converted automatically into parallel code that could take advantage of parallel computers. As there is an enormous code base of existing scientific software written mostly in versions of *Fortran*, and *Fortran* does not have many of the features of later languages that make the problem difficult[3], most of the research effort has been directed towards the development of parallelizing *Fortran* compilers.

Studies have shown that 90% of the execution time of most programs is spent in 10% of the code, mostly in loops. Therefore, much of the effort is in detecting the parallelism in loops. Unfortunately there are many difficulties with this, such as that loops have an unknown number of iterations, and dependence analysis is hard for code that uses pointers and recursion. Hatcher and Quinn [11] also point out that this approach leads to a scenario in which the programmer inadvertently hides the inherent parallelism in sequential loops and other control structures, and the compiler has to seek it out.

1.7.2 Extending a Sequential Language

One relatively easy path towards parallelism is to add functions and compiler directives to an existing sequential programming language so that users can specify the parallelism explicitly. The additional features are incorporated into run-time libraries and header files, and good system provide high-quality documentation to make the programmer's transition as painless as possible. The *MPI* standard is probably the most widely-adopted system that falls into this category. ***MPI***, the ***Message Passing Interface***, is a standardized API used primarily for parallel and/or distributed computing. Language bindings have been written for *C*, *C++*, *Fortran 77* , *Fortran 90*, *Python*, and a few other languages. *MPI* is intended as a system to use when there is no shared address space among the processors, although it also works when there is a shared address space.

OpenMP is another type of extension to a sequential language, but *OpenMP* primarily consists of compiler directives that the programmer embeds in sequential code to direct the compiler about how to parallelize the program. *OpenMP* requires the use of compilers that have been modified to incorporate the *OpenMP* standard. Unlike *MPI*, it does not provide message passing facilities and can only be used when there is a shared address space. These concepts will be explained in Chapter 2.

This path towards parallelism is the one we adopt in this course. It is the easiest for the programmer. There is no need to learn a new language, and both *MPI* and *OpenMP* are so widely adopted that the parallel programs will be very portable. There is also a very large code base and excellent documentation for both systems.

1.7.3 Adding a Parallel Layer to an Existing Sequential Language

Another approach was to add a layer on top of a sequential programming language. The sequential language would be used for the sequentially executed code and the new layer would be a way of expressing parallelism. This approach was tried in the 1990's but the projects have all been discontinued.

1.7.4 Defining a Parallel Language

There are several parallel programming languages that have been either written from scratch or based upon existing sequential languages. The most popular parallel programming language that was written from scratch is *occam*, originally written in 1983 and subsequently revised several times. *Occam* was based upon a theoretical concurrent programming language proposed by Tony Hoare named *Communicating Sequential Processes* (CSP). While there are many others, some of the most prevalent include *Ada*, *Parlog*, and

[3] Pointers, aliasing, and dynamic memory allocation are the primary culprits.

Smalltalk. Languages built upon existing sequential programming languages include *High Performance Fortran* (*HPF*), *Fortran 90*, *C**, *Unified Parallel C*, and *Concurrent Haskell*. These languages come in a wide variety of paradigms. Some are imperative languages, like the *Fortran* and C based ones, some functional, like *Concurrent Haskell*, some logic-based, some based on the actor model (*Smalltalk*), and so on. *Occam* is based on the lambda-calculus, like *Lisp*. Therefore, there is little that all of these languages have in common.

Languages like *HPF* are relatively easy for the programmer to learn because the parallelism is data parallelism. For example, it has special for-loops that mean "execute the loop body in parallel" and high level matrix operations that can be executed in parallel.

Creating a parallel programming language from scratch frees the language designer from the constraints of using an existing language standard, but it also means creating a new compiler from scratch and trying to get users to adopt the new language. Its success also depends on the hardware vendors to develop good compilers for their parallel systems. Thus there are many risks involved from the perspective of the language developer.

For a user to develop code in a new parallel language, there is a portability issue. If there are too few systems that have produced compilers for it or that have not developed the run-time supporting libraries for it, then the code will not be portable. If a standard has not been adopted, then even if the systems support it, there will be compatibility problems.

References

[1] Timothy Mattson, Beverly Sanders, and Berna Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[2] Tim Mattson. The quest for general purpose parallel programming. In *Fourth Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-2011)*, January 2011.

[3] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[4] M.J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Higher Education. McGraw-Hill Higher Education, 2004.

[5] President's Information Technology Advisory Committee. Computational science: Ensuring america's competitiveness. Report to the President, June 2005.

[6] E. Levin. Grand challenges to computation science. *Communications of the ACM*, 32(12):1456–1457, December 1989.

[7] Mathematical Committee on Physical and Engineering Sciences. Grand challenges: High performance computing and communications. Report to the President, 1992.

[8] College of Staten Island CUNY High Performance Computing Center. Hpc systems at the cuny hpcc, 2013. at http://www.csi.cuny.edu/cunyhpc/HPC_Systems.html.

[9] Texas Advanced Computing Center. Stampede dell poweredge c8220 cluster with intel xeon phi coprocessors, 2013. at https://www.tacc.utexas.edu/resources/hpc/stampede-technical.

[10] James R. McGraw and Timothy S. Axelrod. Programming parallel processors. chapter Exploiting Multiprocessors: Issues and Options, pages 7–25. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[11] P.J. Hatcher and M.J. Quinn. *Data-parallel programming on MIMD computers*. MIT Press), address=Cambridge,MA,, 1991.

Subject Index

Accelerated Strategic Computing Initiative, 3

centralized multiprocessor, 2
coarse-grained data parallelism, 5

data clustering, 7
data dependence graph, 4
data parallelism, 5

fine-grained data parallelism, 5
functional parallelism, 5

grand challenge problems, 2

Message Passing Interface, 9
multi-core processor, 2
multicomputer, 2

parallel computer, 2
parallel computing, 1
parallel programming, 2
parallelizing compiler, 9
petaflops, 3
pipelined computation, 6

symmetrical multiprocessor, 2

TOP500, 3

unrolling loops, 6