Parallel Gain-Bandwidth Characteristics Calculations for Thin Avalanche Photodiodes on an SGI Origin 2000 Supercomputer

Yi Pan
Department of Computer Science
Georgia State University
Atlanta, GA 30303, USA
email: pan@cs.gsu.edu

Constantinos S. Ierotheou
Parallel Processing Research Group
University of Greenwich
London SE10 9LS, UK
Email: C.Ierotheou@gre.ac.uk

Majeed M. Hayat
Department of Electrical & Computer Engineering
The University of New Mexico
Albuquerque, NM 87131-1356, USA
Email: hayat@eece.unm.edu

ABSTRACT

An important factor for high speed optical communication is the availability of ultrafast and low-noise photodetectors. Among the semiconductor photodetectors that are commonly used in today's long-haul and metro-area fiber-optic systems, avalanche photodiodes (APDs) are often preferred over *p-i-n* photodiodes due to their internal gain, which significantly improves the receiver sensitivity and alleviates the need for optical pre-amplification. Unfortunately, the random nature of the very process of carrier impact ionization, which generates the gain, is inherently noisy and results in fluctuations not only in the gain but also in the time response. Recently, a theory characterizing the autocorrelation function (or the power spectral density) of APDs has been developed by us which incorporates the dead-space effect. The research extends the time-domain analysis of the dead-space multiplication model to compute the autocorrelation function of the APD impulse response. However, the computation requires a large amount of memory space and is very time consuming. In this research, we describe our experiences in parallelizing the code using both MPI and OpenMP. Several array partitioning schemes and scheduling policies are implemented and tested. Our results show that the code is scalable up to 64 processors on an SGI Origin 2000 machine and has small average errors.

1 Introduction

High-speed optical communication has been used widely in networks. One key factor for high speed optical communication is the availability of ultrafast and low-noise photodetectors [1,2,3,5]. Among the semiconductor photodetectors that are commonly used in today's long-haul and metro-area fiber-optic systems, avalanche photodiodes (APDs) are often preferred over *p-i-n* photodiodes due to their internal gain, which significantly improves the receiver sensitivity and alleviates the need for optical pre-amplification. Unfortunately, the random nature of the very process of carrier impact ionization, which generates the gain, is inherently noisy and results in fluctuations not only in the gain but also in the time response [1,2,3,5].

Just as accounting for dead space is essential in the correct prediction of the excess noise factor in thin APDs, accurately predicting the bandwidth characteristics of thin APDs necessitates having a time-response analysis of the avalanche multiplication that includes the effect of dead space. This is particularly important if we were to push the performance limits of thin APDs to meet the needs of next-generation 40-Gbps lightwave systems [3].

Recently, a theory characterizing the autocorrelation function (or the power spectral density) of APDs has been developed which incorporates the dead-space effect [3]. The research extends the time-domain analysis of the dead-space multiplication model reported in [5] to compute the autocorrelation function of the APD impulse response. This extension involves developing six recurrence equations, which are derived according to the same renewal-theory rationale used in [3]. To solve these equations, a program called NP3 was developed. It deals with the calculation of the autocorrelation function of the APD's impulse response. We get the power spectral density by simply taking the Fourier transform of the autocorrelation function. The gain is not calculated in this code, and was calculated using another serial code, since it is much simpler. Numerical analysis indicates that the software gives accurate predictions. In general, the agreement with the experimental bandwidth vs. gain plots is good in the range of high gains (>15) with an approximate average error of 10%.

The computation of the autocorrelation functions in the code involves 12 huge three-dimensional arrays and hence requires a large amount of memory space. For example, for a suitable ($200 \times 500 \times 500$) array size, we estimate that we need at least 2.4 GBytes memory (assume that 32 bits are used for each floating-point numbers). Personal computers or workstations rarely exceed 2 GBytes of memory space. Thus, handling suitable mesh size for accurate calculation is currently problematic with serial programming due to limited memory space available on a single processor machine. Big execution time on a sequential machine is also a problem for repeated calculations. For example, for a problem with small array size such as $100 \times 100 \times 100$, the computation time using one processor on an SGI Origin 2000 takes about two and half hours. For larger problem sizes, the time will increase rapidly and the problem soon becomes

impractical within a reasonable time on a single processor system. Thus, solving these equations on a high performance computing (HPC) system is essential to solve both the time and memory constraints existing on a single processor system.

In this research, we describe our experimental results of parallelizing the NP3 code using both Message Passing Interface (MPI) [9] and OpenMP [4]. Our results show that the code can be parallelized efficiently and the code is also scalable up to at least 64 processors on an SGI Origin 2000 machine [8]. The rest of the paper is organized as follows.

The numerical formulation and basic structure of the corresponding sequential code will be discussed in section 2. MPI parallelization of the code is presented in section 3. Section 4 will cover the parallelization process using OpenMP. Experimental results, observation, and comparisons will be given in section 5. We conclude our paper in section 6.

2 Numerical Formulations and Structure of Sequential Code

To describe the computations involved in obtaining the power spectral density of APDs, we first provide a brief description of the mathematical model involved, drawing freely from the formulation developed in [3]. We begin by recalling key definitions involved in the dead-space multiplication theory developed in [3,5]. We will then recall the basic equations developed in [3], which characterise the first and second-order statistics of the APD's impulse response function. The parallel computing technique reported in this paper is developed precisely for the purpose of solving these integral equations.

2.1. The dead-space multiplication model (DSMT)

Consider an electron-injected APD with a multiplication region of width w. Let $Z_e(t,x)$ be the total number of *electrons* resulting from an initial parent electron born at location x, t units of time after its birth. Similarly, let $Z_h(t,x)$ be the total number of holes resulting from an initial parent electron, at location x, t units of time after its birth. The random impulse response, which is a stochastic process, can be related to the functions Z_e and Z_h through the relation $I(t) = (q/w)\,[v_e Z_e(t,0) + v_h\, Z_h(t,0)]$, where $v_{e\ and}\ v_e$, are, respectively, the electron and hole saturation velocities in the APD's depletion region. Our goal is to mathematically characterize the first and second-order moments of $I(t)$, which is accomplished when the statistics of $Z_e(t,0)$ and $v_e Z_h(t,0)$ are determined.

As discussed in [3], it turns out that it is necessary to first characterize the statistics of $Z_e(t,x)$ and $Z_h(t,x)$ for all x and then specialize the results to x=0. We also need to introduce auxiliary quantities representing cases when a hole initiates the multiplication. In particular, let $Y_e(t,x)$ be the total number of *electrons* resulting from a parent *hole* born

at location x, t units of time after its birth, and let $Y_h(t,x)$ be defined similarly to $Y_e(t,x)$ but with the number of generated electrons replaced with the number of generated holes. Using the above definitions, recurrence equations (integral equations) characterizing the mean of $Z_e(t,x)$, $Z_h(t,x)$, $Y_e(t,x)$ and $Y_h(t,x)$ have been derived in [5]. For example, if we define the mean quantities $z_e(t,x)$, $z_h(t,x)$, $y_e(t,x)$ and $y_h(t,x)$, then the functions $z_e(t,x)$ and $y_e(t,x)$ are related by the following integral equation:

$$\begin{aligned} z_e(t,x) &= u\ ([(w-x)/v_e]\ -\ t)\ [1-H_e(v_e t)] \\ &+ \int_{x}^{\min(x+v_e t,w)} [2z_e(t-(s-x)/v_e,s) + y_e(t_2-(s-x)/v_e,s)]h_e(s-x)ds \end{aligned} \tag{1.1}$$

where $H_e(x)$ is the indefinite integral of $h_e(x)$, which is a known probability density function whose form is given in [3], and $u(x)$ is the unit step function. A similar integral equation exits for $y_e(t,x)$ (also involving $y_e(t,x)$ and $z_e(t,x)$). Hence, to determine the mean quantities $z_e(t,x)$ and $y_e(t,x)$, we must solve two coupled integral equations of the type shown in (1.1). Similarly, two more coupled integral equations are available and must be solved to compute $y_h(t,x)$ and $z_h(t,x)$. This completes the description of computing the first-order statistics of the impulse response.

We now state the equations that characterize the autocorrelation function of the stochastic process $I(t)$, defined by $R_I(t_1,t_2) = E[I(t_1)\ I(t_2)]$. Following [3], the autocorrelation can be expressed in terms of certain count auto and cross correlations as follows:

$$R_I(t_1,t_2) = (q/w)^2\ [v_e^2\ C_{Ze}(t_1,t_2,0) + v_h^2\ C_{Zh}(t_1,t_2,0) + v_e\ v_h\ C_z(t_1,t_2,0) + v_e\ v_h\ C_z(t_2,t_1,0)],$$

where the count autocorrelations are defined as: $C_{Ze}(t_1,t_2,x) = E[Z_e(t_1,x)Z_e(t_2,x)]$ and $C_{Zh}(t_1,t_2,x) = E[Z_h(t_1,x)Z_h(t_2,x)]$, and the count cross correlation is defined by $C_z(t_1,t_2,x) = E[Z_e(t_1,x)Z_h(t_2,x)]$. In [3], it is shown that these auto and cross correlations satisfy certain linear and pairwise-coupled (integral) equations. For example, $C_{Ze}(t_1,t_2,x)$ and $C_{Ye}(t_1,t_2,x)$ satisfy the following equation:

$$\begin{aligned} C_{Z_e}(t_1,t_2,x) &= u\ ([(w-x)/v_e]\ -\ t_2)\ [1-H_e(v_e t_2)] \\ &+ \int_{\min(x+v_e t_1,w)}^{\min(x+v_e t_2,w)} [2z_e(t_2-\Delta_1,s) + y_e(t_2-\Delta_1,s)]h_e(s-x)ds \\ &+ \int_{x}^{\min(x+v_e t_1,w)} [2C_{Z_e}(t_2-\Delta_1,s) + C_{Y_e}(t_2-\Delta_1,s)]h_e(s-x)ds \\ &+ \int_{x}^{\min(x+v_e t_1,w)} 2z_e(t_1-\Delta_1,s)[z_e(t_2-\Delta_1,s) + y_e(t_2-\Delta_1,s)]h_e(s-x)ds \\ &\int_{x}^{\min(x+v_e t_1,w)} y_e(t_1-\Delta_1,s)[2z_e(t_2-\Delta_1,s) + y_e(t_2-\Delta_1,s)]h_e(s-x)ds \end{aligned} \tag{1.2}$$

A similar equation exists for $C_{Ye}(t_1,t_2,x)$, also in terms of $C_{Ze}(t_1,t_2,x)$ and $C_{Ye}(t_1,t_2,x)$, resulting in a pair of coupled equations. The two coupled equations must be solved to

yield $C_{Ze}(t_1, t_2, x)$ and $C_{Ye}(t_1, t_2, x)$. Note that in the above equation, the first-order quantities $y_e(t,x)$ and $z_e(t,x)$ are assumed known and must be solved using the equations described earlier in this Section. Similarly, two coupled integral equations are also available characterizing $C_{Zh}(t_1, t_2, x)$ and $C_{Yh}(t_1, t_2, x)$, and finally, two more are available for $C_Z(t_1, t_2, x)$ and $C_Y(t_1, t_2, x)$. In summary, to compute the autocorrelation function $R_I(t_1, t_2)$, three pairs of pairwise coupled integral equations (characterizing the second-order statistics) and two pairwise coupled integral equations (characterizing the first-order statistics) must be solved.

The above mentioned equations are solved numerically using a simple iteration technique. For example, for each pair of coupled integral equations (in two unknown functions), the unknown functions (e.g., $C_{Ze}(t_1, t_2, x)$ and $C_{Ye}(t_1, t_2, x)$) are initially assumed to be identically zero. The initial values are then substituted in the integral equations to yield the first-order iterates, and so on. The iteration process is terminated when the relative change from one iteration to the other drops below a prescribed level (10^{-8} in the calculations in [3]). The iteration procedure was encoded with FORTRAN.

2.2. Key subroutines used in the computations

Subroutine `mean_ze` numerically implements the integral equation given by (1.1). It consists of three nested loops: two loops to exhaust the variables t and x, and a loop that implements the integration. (The functions H_e and h_e are computed outside the subroutine and are passed to the subroutine whenever it is called.) The t and x variables are discretized using a mesh size n_t by n_s. Moreover, for each t and x, equation (1.1) is carried out using the same mesh size used for x. The loops in the subroutine have the following general structure:

```
do i=1,ns
  do j=1,nt
    "compute the first term on the right-hand side of (1.1)"
    do k=j,ns
      "compute and update the integrand in (1.1)"
```

Similarly-structured subroutines exist to implement the remaining three integral equations for the first-order statistics $y_e(t,x)$, $y_h(t,x)$, and $z_h(t,x)$: these subroutines are named `mean_ye, mean_yh, mean_zh,` respectively.

The subroutines used to compute the second-order statistics have an added loop to handle the extra time variable t_2. For example, subroutine `auto_Cze` numerically implements the integral equation given by (1.2). In addition to the three loops handling t_1, t_2 and x, there is a loop that carries out the integration. Again, the t_1, t_2 and x variables are discretized using a mesh size n_t by n_t by n_s, respectively. The loops in the subroutine have the following general structure:

```
do i=1,ns
```

```
        do j=1,nt
          do k=1,nt
            "compute the first term on the right-hand side of (1.2)"
            do m=j,ns
              "compute and update the integrand in (1.2)"
```

Similarly structured subroutines exist to implement the remaining five integral equations for the first-order statistics $C_{Ye}(t_1, t_2, x)$, $C_{Zh}(t_1, t_2, x)$ and $C_{Yh}(t_1, t_2, x)$, $C_Z(t_1, t_2, x)$ and $C_Y(t_1, t_2, x)$,: these are named `auto_Cye, auto_Czh, auto_Cyh, cross_Cz ,` and `cross_Cy`, respectively.

With the above subroutines defined, the sequential program structure is shown below.

```
        Program np3
        . . .
        do 10 kk=1, 300 (maximum allowed number of iterations)
          call mean_ye
          call mean_ze
          call mean_yh
          call mean_zh
          "check condition for terminating the iterations"

          . . .
10      continue
        . . .
        do 11 kk=1,300
          call cross_Cz
          call cross_Cy
          "check condition for terminating the iterations"

          . . .
11      continue
        . . .
        do 101 kk=1, 300
          call auto_Cye
          call auto_Czy
          call auto_Cyh
          call auto_Czh
          "check condition for terminating the iterations"
          . . .
101     continue
        . . .
```

As we can see from the serial code, the major work is done in the subroutines `mean_ye, mean_ze, mean_yh, mean_zh, cross_Cz, cross_Cy, auto_Cye, auto_Czy, auto_Cyh` and `auto_Czh`. Recall that all of these subroutines involve nested loops (three or four). In particular, the correlation subroutines are extremely memory and time intensive, since they involve three dimensional arrays and four nested loops. Clearly, if we can parallelize these loops efficiently, then we can reduce the computation time drastically. In the following sections, we will describe the parallelization process in more detail.

3 MPI Parallelization

MPI is a library specification for a message-passing scheme, proposed as a standard by a broadly based committee of vendors, implementers, and users [9]. The main advantages of establishing a message-passing standard are portability and ease-of-use. In a distributed memory communication environment in which the higher level routines and/or abstractions are built upon lower level message passing routines, the benefits of standardization are particularly apparent. Furthermore, the definition of a message passing standard provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability.

An important decision for an MPI implementation is to decide how to partition arrays in a distributed memory environment. An inspection of the original NP3 serial code did not appear to exhibit the characteristics of a code that would yield a favourable level of performance when executed using a distributed memory parallel system. For example, although there were a number of multi-dimensional arrays and nested loops, there appeared to be a high communication cost that would be associated with data movement due to the typical way in which the data was being accessed. As an illustration Figure 1 below shows a fragment of the NP3 code.

```
      Program np3
      . . .
      call mean_ye
      call mean_ze
      call mean_yh
      call mean_zh
      . . .


      subroutine mean_yh
      . . .
      do 21 k=1,nt+1
        do 22 n=2,ns+1
          sumy=0.
          do 23 i=max1(1.,n-(((k-1.)/lh)+1)),n-1
            s=1+int(k-lh*(n-i))
            sumy=sumy+(2*b(s,i)+a(s,i))*(gh(n-i+1))
23        continue
          c(k,n)=hh(k,n)+(sumy*dx)
22      continue
21    continue
      . . .

      subroutine mean_zh
      . . .
      do 31 k=1,nt+1
        do 32 n=1,ns
          sumz=0.
          do 33 i=n+1,min0(ns+1,n+int(((k-1.)/le)+1))
```

```
            s=1+int(k-le*(i-n))
            sumz=sumz+(2*a(s,i)+c(s,i))*(ge(i-n+1))
33        continue
          d(k,n)=sumz*dx
32      continue
31    continue
      . . .
```

FIGURE 1. Typical data accesses for arrays in mean-based routines

From this case alone there are at least two different scenarios that can be explored.

1. The arrays `a`, `b` and `c` are not partitioned. This in turn causes the routines to be executed in serial as each processor will compute information for all iterations of all loops. Although there are very few changes required for some of the routines, this is not ideal and will have a significant impact on the performance of the parallel version of the code.

2. Arrays `a`, `b` and `c` could ideally be partitioned in index 2 (or using the `n` loop index). However, this has two undesirable effects

    i. arrays `a` and `b` in routine `mean_yh` are accessed in index 2 using the `i` index (this is the innermost loop of the triple nest of loops). This conflicts with the requirement to use the `n` loop to define the masked statements in the parallel implementation. As a result the `a` and `b` arrays need to be broadcast to all processors prior to their use in routine `mean_yh`.

    ii. for similar reasons, array c will also require to be broadcast prior to its usage in routine `mean_zh`. This conflict of data accesses is prevalent in much of the NP3 code affecting many two and three dimensional arrays.

Therefore at a first glance, one would not expect to obtain a good quality parallelization. Scenario (2) has the greater scope for improvement if one can re-structure the existing code such that the data accessing of the arrays better reflects their alignment with their defined distribution [10]. One possible solution is to attempt the separation of the computation in routine `mean_yh` so that both the `i` index and the `n` index can be used to exploit the distribution in index 2. Figure 2 shows how this can be achieved for routine `mean_yh` at the expense of an increase in the program memory requirement.

```
      subroutine mean_yh
      . . .
      do 21 k=1,nt+1
        do 22 n=2,ns+1
          sumy(n,k)=0.
          do 23 i=max1(1.,n-(((k-1.)/lh)+1)),n-1
            s=1+int(k-lh*(n-i))
            sumy(n,k)=sumy(n,k)+(2*b(s,i)+a(s,i))*(gh(n-i+1))
23        continue
```

```
22      continue
21    continue

      do k=1,nt+1
        do n=2,ns+1
          c(k,n)=hh(k,n)+(sumy(n,k)*dx)
        enddo
      enddo
      . . .
```

FIGURE 2. Loop split to exploit parallelism in serial code

The loop split transformation [10] is a standard modification to loop structures that can only be applied if there is no violation in the order in which the computation is performed. In this case it can only be applied if the scalar `sumy` is expanded to a two dimensional array, thereby removing the data dependence for `sumy` between iterations of the `i` loop. All distributed accumulations of `sumy` are made in the first triple nest of loops, this is followed by a double nested loop that uses the array `sumy` to update the array `c`. The parallelism exploited here is now both at the `i` loop in the first nest and also at the `n` loop in the second nest. For correct parallel execution it is also necessary to complete the reduction operation by accumulating all local contributions into a single global summation. This would require communicating data of the order `ns+1` instead of the broadcast cost of `(nt+1)*(ns+1)` for each individual array.

The Computer Aided Parallelization toolkit [6] was used to attempt to perform the parallelization using the strategy described above and to generate a Single Program Multiple Data (SPMD) version of the NP3 code. There are a number of stages that the user needs to go through with the tools to generate the parallel code. These are shown schematically in Figure 3.

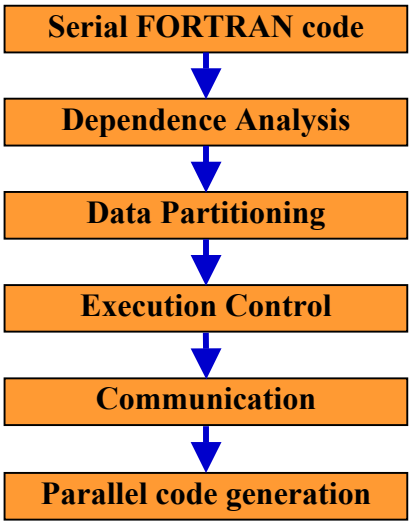

FIGURE 3. Overview of the message passing based parallelization stages performed by the tools

Serial Fortran code: The serial FORTRAN version of the code is parsed and stored in an internal form by the toolkit.
Dependence Analysis: the toolkit performs a detailed interprocedural, symbolic, value-based, dependence analysis. The dependence analysis defines the core of the toolkit and helps to identify the potential parallelism in the code. The user can then use the available transformation tools to re-structure all the necessary routines by performing a loop split as described in Figure 2. As part of this transformation process, the toolkit also check the legality of any transformation to ensure that the transformed code is valid. In addition, the user is given the opportunity to preview any transformed code and makes a decision to either accept or reject the suggested code changes.

Data Partitioning: The data partitioning of all relevant arrays is then also carried out by the toolkit. This process requires the user to suggest an initial starting point to the partitioner, for example, the user can specify index `2` of array `a` in routine `mean_yh`. The partitioner then uses this information and identifies all other arrays that can be defined to have similar data distributions throughout the entire code. The strategy uses a 1D domain decomposition. Due to the data dependency in the other loops a 2D decomposition is not feasible in this case.

Execution Control Masking: The re-structuring of the serial code to execute in parallel using an SPMD paradigm begins with attempting to identify and place execution control masks for all relevant statements. These masks define which processor(s) at run-time will execute any given statement. Ideally, one would like a uniform set of masks that are applied to as many statements as possible in the code. The use of masks that reflect the processor "ownership" or assignment area of the arrays is also desirable. So for example, if each processor at run-time has defined `low` and `high` assignment range limits then it would be more efficient to generate masks for routine `mean_yh` as shown in Figure 4. These masks exploit the parallelism at both the `i` loop and also the `n` loop by executing the statements (in italics) in parallel.

```
      subroutine mean_yh
      . . .
      do 21 k=1,nt+1
        do 22 n=2,ns+1
          sumy(n,k)=0.
          do 23 i=max(max1(1.,n-(((k-1.)/lh)+1)),low),min(n-1,high)
            s=1+int(k-lh*(n-i))
            sumy(n,k)=sumy(n,k)+(2*b(s,i)+a(s,i))*(gh(n-i+1))
23        continue
22      continue
21    continue

      do k=1,nt+1
        do n=max(2,low),min(ns+1,high)
          c(k,n)=hh(k,n)+(sumy(n,k)*dx)
        enddo
      enddo
      . . .
```

FIGURE 4. Execution control masks to define parallel execution

Communication Generation: In order to ensure parallel execution similar to that for the serial code, the final step in the parallelization process is to identify and place communication calls into the modified code. The aim is to try and identify a minimum set of communication requirements to reflect the changes already made to the code. There are many requests for data to be communicated based on the distribution of the data across the processors. The toolkit identifies these requests and then attempts to migrate them higher up in the call graph. Further movement of the communication requests is prevented when they encounter a barrier and this is usually an assignment of the variable requested for communication. At this point an attempt is made to merge any similar requests for the same variable, finally culminating in a communication call to a message passing library routine. In this code most of the communication calls were based on reduction operations that were generated as a result of the loop split shown in Figure 2. The final parallel version of routine `mean_yh` is shown in Figure 5.

```
      subroutine mean_yh
      . . .
      do 21 k=1,nt+1
        do 22 n=2,ns+1
          sumy(n,k)=0.
          do 23 i=max(max1(1.,n-(((k-1.)/lh)+1)),low),min(n-1,high)
            s=1+int(k-lh*(n-i))
            sumy(n,k)=sumy(n,k)+(2*b(s,i)+a(s,i))*(gh(n-i+1))
23        continue
22      continue
21    continue
      call cap_mcommutative(sumy(1,1),(nt+1)*(ns+1),2,cap_mradd)
      do k=1,nt+1
        do n=max(2,low),min(ns+1,high)
          c(k,n)=hh(k,n)+(sumy(n,k)*dx)
        enddo
      enddo
      . . .
```

FIGURE 5. High level communication call representing an array global summation

Parallel code generation: The final code generation to a file (or files) can be defined in one of two ways depending on the user's requirements. Currently, the two options are:

1. To generate parallel code that still retains the original array declarations. Therefore, every processor will contain a full copy of the all the arrays in the code.

2. To generate parallel code that re-defines the array declarations to be a function of the minimum number of processors used during program execution (generally this must be greater than 1). This will take into account whenever possible, the reduced memory requirement for each processor as a result of the distribution of the arrays. This approach generally has a better scalability property than (1) and will allow larger problem sizes to be solved. This was the selected option for the experiments conducted below.

4 OpenMP Parallelization

OpenMP's programming model uses fork-join parallelism where master thread spawns a team of threads as needed [4]. Parallelism can be added incrementally i.e., the sequential program evolves into a parallel program. Hence, we do not have to parallelize the whole program at once. A user finds the most time consuming loops in the code, and for each loop, the iterations are divided up amongst the available threads. In this section we will give some simple examples to demonstrate the major features of OpenMP.

When parallelizing a loop in OpenMP, we may also use the *schedule* clause to perform different scheduling policies to effect how loop iterations are mapped onto threads. There are four scheduling policies available in the OpenMP specification. The *static* scheduling method deals out blocks of iterations of size "chunk" to each thread. In the *dynamic* scheduling method, each thread grabs "chunk" iterations off a queue until all iterations have been handled. In the *guided* scheduling policy, threads dynamically grab blocks of iterations (the size of the block starts large and shrinks down to size "chunk" as the calculation proceeds). This helps to achieve a good load balance amongst the processors. Finally, in the runtime scheduling method, schedule and chunk size can either be set using the OMP_SCHEDULE environment variable or can be defined in the code for each loop. In our study we condsidered both static and dynamic scheduling approaches with varying chunk sizes.

The toolkit can also be used to generate OpenMP directive code for shared memory machines [7] and was used here to parallelize the NP3 code. As with the message passing parallelization, there are a number of stages that the user needs to go through with the tools to generate the parallel code, but these are fewer (and somewhat easier) to perform. These are shown schematically in Figure 6.

The serial FORTRAN code and Dependence analysis stages are the same as those described above for the message passing based process.

Directives Generation:
This involves the structured examination of the loops within the code. The classification of loop types makes it easier to identify critical loops and also loops that can be potentially made parallel. The GUI directives browser allows the user to see at a glance and to inspect the different types of serial and parallel loops that have been identified. In conjunction with the other tools browsers such as the dependence graph, variable definition and transformation browsers, the user is able to iteratively refine the identification and placement of OpenMP directives. The generation of the OpenMP code is then carried out automatically. These steps are illustrated in Figure 6.

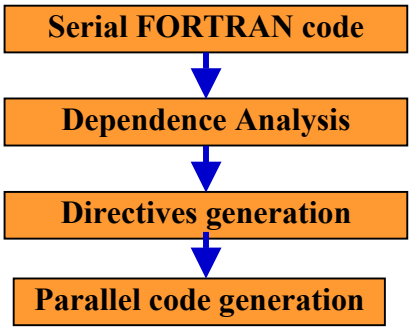

FIGURE 6. Overview of the directive based parallelization stages performed by the tools

The code segment shown in Figure 7 displays the corresponding subroutine `mean_yh` after inserting OpenMP directives for loop distribution. Clearly, in the code we would like to parallelize the `k` loop (the outmost loop) so that each thread has enough work to do, in doing so make the loop scheduling overhead less significant. Here, we do not use loop split since it may coarsen the granularity of the computation and result in poorer performance.

```
      subroutine mean_yh
      . . .
!$OMP DO
      do 21 k=1,nt+1
        do 22 n=2,ns+1
          sumy=0.
          do 23 i=max1(1.,n-(((k-1.)/lh)+1)),n-1
            s=1+int(k-lh*(n-i))
            sumy=sumy+(2*b(s,i)+a(s,i))*(gh(n-i+1))
23        continue
          c(k,n)=hh(k,n)+(sumy*dx)
22    continue
21    continue
!$OMP END DO
```

FIGURE 7. OpenMP Code Segment with Loop Distribution Directives.

Figure 8 shows that the placement of directives, in particular the setting up of a PARALLEL region can be very significant to the overall parallel performance. This can lead to a smaller overhead in the setting up of regions if a single one can be used instead of a number of smaller regions each with its own start up and synchronization cost. In the example shown, a single region is used to cover the exploitation of parallelism in all `mean_` routines by defining the region outside the calls rather than four individual ones – one inside each routine.

```
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(jj,j)
```

```
!$OMP&  SHARED(delta)
REDUCTION(MAX:error_zh,error_yh,error_ze,error_ye)
      call mean_ye
      call mean_ze
      call mean_yh
      call mean_zh
      . . .
!$OMP END PARALLEL
```

FIGURE 8. OpenMP Code Segment with careful placement of Directives.

5 Experimental Results

Two test sizes for a given test case were tried. Case 1 was defined by a 110x110x110 size problem and case 2 was defined by a 200x200x200 size problem. Results were run on an Origin 2000 populated with 64 300MHz processors and a total of 64Gb of memory. Each processor has a primary data cache of 32Kb, a primary instruction cache of 32Kb and a secondary unified data/instruction cache of 8Mb.

Table 1 shows the execution times for problem size 110x110x110 with varying chunk size using a *dynamic* scheduling policy. The results shows that the default chunk size (of one iteration per thread) for the scheduling approach yields the most efficient solution. The reason is probably that since we parallelize the outer loop using OpenMP, each chunk has a large amount of work to do. Increasing the chunk size will make the load more unbalanced across the threads. The corresponding speedups are shown in Figure 9.

# Threads	Default Chunk Size	Chunk Size = 2	Chunk Size =3
1	9698	9698	9698
2	5160	5102	5147
4	2687	2717	2768
8	1443	1478	1553
16	672	845	935
32	460	475	630
64	274	287	395

Table 1. Execution Times Using Dynamic Scheduling with Problem Size 110×110×110

Table 2 shows the execution times for problem size 110x110x110 with varying chunk sizes using a *static* scheduling policy. The results shows again that the default chunk size for the scheduling approach yields the most efficient solution when the number of threads is large. When the number of threads used is small (2-8), then using a small chunk size outperforms default chunk size. However, as the number of thread increases the default chunk size yields a better performance. The corresponding speedups are shown in Figure 10.

# Threads	Default Size	Size = 1	Size = 2	Size =3
1	9732	9732	9732	9732
2	5219	5041	5102	5147
4	2781	2753	2717	2768
8	1511	1478	1478	1553
16	796	796	845	935
32	481	490	499	630
64	291	298	310	395

Table 2. Execution Times Using Static Scheduling with Problem Size 110x110x110

Table 3 summarizes the execution times of the OpenMP code using *static* or *dynamic* scheduling policies and the MPI code. There is little to choose between the 'best' dynamic and static scheduling approaches – the dynamic approach was slightly more efficient (speedup = 35 when the number of processors used is 64). Figure 11 shows the corresponding speedups of the best performing OpenMP code using dynamic scheduling and the MPI code.

Results for the message passing parallelization for this test case shows an interesting variation as the number of processors are increased. Between 2-16 processors the better cache usage and relatively small communications give exceptional performance over the serial run. From about 20 processors onwards the communication cost becomes more significant and begins to outweigh the computation being carried out. The majority of communications are reduction operations. The cost of a reduction operation (implemented as a hypercube) is significant as the number of processors is increased. This cost starts to outweigh the volume of computation and the cache benefits (better with a small number of processors) being performed in the MPI parallelization.

Also notice that the MPI code outperforms the OpenMP code when the number of processors used is between 4 and 16. In fact, the MPI code displays superlinear speedups when the number of processors used is between 4 and 16. This is probably caused by much reduced cache misses in the code due to much less memory requirement on each processor when we use multiple number of processors instead of a single processor. On the other hand, the OpenMP code performs better than the MPI code when the number of processors used is really large (e.g., 64). Even when the number of processors is large, each processor still has enough work to do. On the other hand, the MPI code's communication time dominates the total time since each processor has less work to do as the number of processors are increased. Also the need for dynamic load balancing seems to hit the MPI implementation harder than the OpenMP. OpenMP has more flexibility in its use with dynamic scheduling. Due to all the above reasons, the MPI code is less efficient than the OpenMP code when using larger numbers of processors.

# Threads	Default Dynamic	Default Static	MPI
1	9698	9732	9811
2	5160	5219	6368
4	2687	2781	1626
8	1443	1511	678
16	672	796	522
32	460	481	499

64	274	291	590

Table 3. Comparisons of OpenMP Executions and MPI Executions with Problem Size 110x110x110

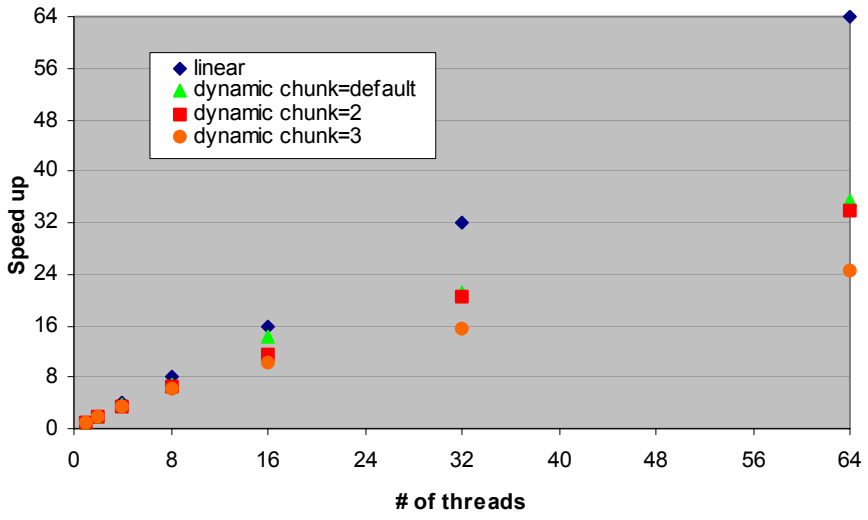

Figure 9. Speedups Using Dynamic Scheduling with Problem Size 110x110x110

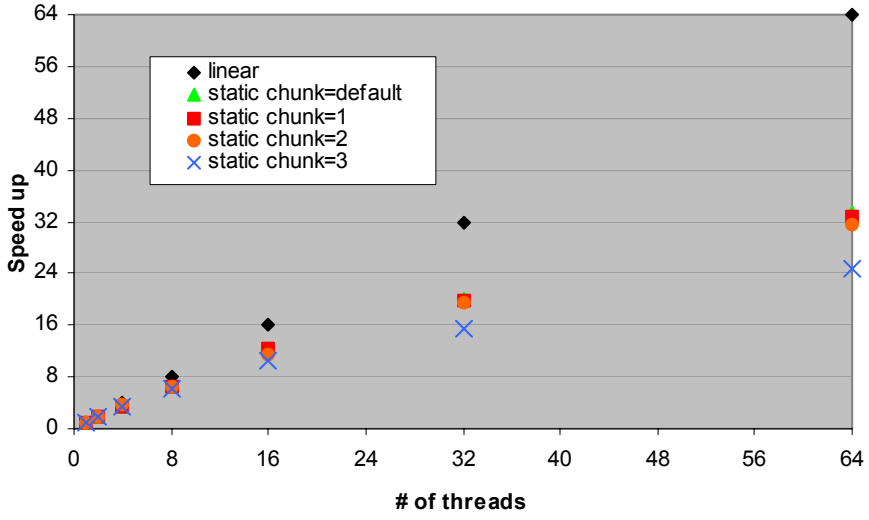

Figure 10. Speedups Using Static Scheduling with Problem Size 110x110x110

Table 4 summarizes the execution times of the OpenMP code using the dynamic scheduling policy and the MPI code with problem size 200x200x200. The speedups of the corresponding runs are also shown in Figure 12.

# Threads	Default Dynamic	MPI
1	114541	114911
2	59280	79681
4	30765	38966
8	16104	8368
16	7701	3982
32	4541	3290
64	2822	3305

Table 4. Comparisons of OpenMP Executions and MPI Executions with Problem Size 200x200x200

For this case, the conclusions are very similar to that of the previous case. For the message passing parallelization, the trend is shifted to a higher number of processors and the performance of the code on 64 processors is now much closer to the OpenMP implementation. The speedup is improved from 17 (case 1) to 35 (case 2). Performance from the shared memory parallelization indicates a speedup of 41 when the number of processors used is 64. Clearly, the speedup is better in this case (case 2) than in the previous case (case 1) where the speedup is only 35 when using 64 processors. We expect that as the problem size increases, the scalability of both codes will become even better due to the change of ratio in computation workload on each processor vs. the communication overheads.

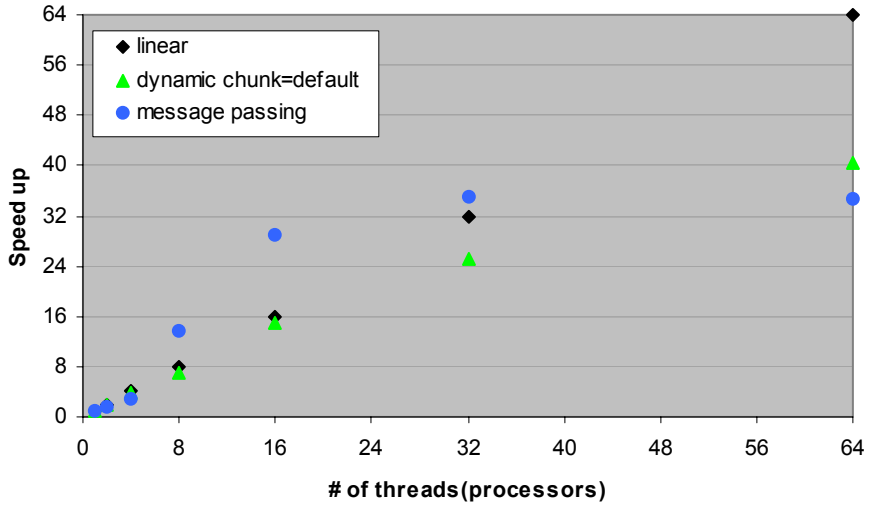

Figure 11. Speedups for Problem Size 110x110x110.

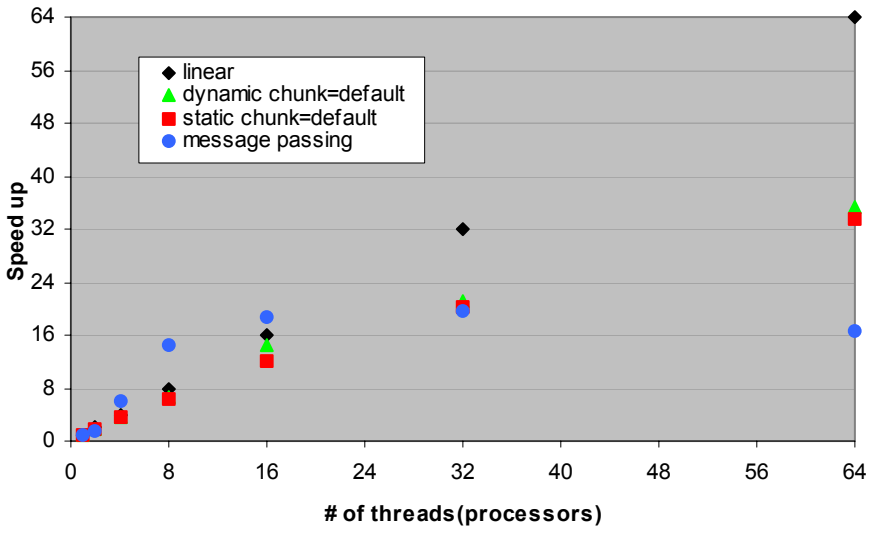

Figure 12. Speedups for Problem Size 200x200x200.

6 Conclusions

We have parallelized a sequential Fortran code, which is the major program for calculating gain-bandwidth characteristics for thin avalanche photodiodes, using both MPI and OpenMP. The code is parallelized with the aid of a toolkit which is capable of accurately analysing dependencies in serial codes and generating portable parallel source codes in a semi-automatic and interactive way. Using this approach, many designs can be implemented quickly, and decisions can be made efficiently. Despite the apparent lack of parallelism present when performing a distributed memory parallelisation, running the executable on an SGI Origin 2000 supercomputer indicates that both OpenMp and MPI codes are scalable up to 64 processors on the SGI machine - the OpenMP code is more efficient than the MPI code as the number of processors is increased. We predict that the OpenMP code will be more scalable and efficient when a bigger array size such as 500×500×500 is used. Our study further shows that it is possible to parallelize many engineering programs efficiently and economically with much reduced porting costs when using such a toolkit.

7 Acknowledgement

This research was supported in part by the National Science Foundation under Grant ECS-0196569. The authors have had many useful discussions with Prof. Joe C. Campbell of the University of Texas at Austin.

8 References

1. J. C. Campbell, W. S. Holden, G. J. Qua, and A. G. Dentai, ``Frequency response InP/InGaAs APD's with separate absorption grading and multiplication regions," IEEE J. Quantum Electronics, vol. QE-21, pp. 1743--1749, 1985.

2. J. C. Campbell, B. C. Johnson, G. J. Qua, and W. T. Tsang, ``Frequency response InP/InGaAsP/InGaAs APD's," J. Lightwave Technology, vol. 7, pp. 778--784, 1989.

3. M.M. Hayat, O.-H. Kwon, Yi Pan, P. Sotirelis, J.C. Campbell, B.E.A. Saleh, and M.C. Teich, ``Gain-Bandwidth Characteristics of Thin Avalanche Photodiodes,'' IEEE Trans. on Electron Devices, vol. 49, no. 5, pp. 770-781, May 2002.

4. R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, Parallel Programming in OpenMP, Morgan Kaufmann Publishers, 2000.

5. M. M. Hayat, and B. E. A. Saleh, ``Statistical properties of the impulse response function of double-carrier multiplication avalanche photodiodes including the effect of dead space," Journal of Lightwave Technology, vol.10, pp.1415--1425, 1992.

6. C.S. Ierotheou, S.P. Johnson, M. Cross, and P.F. Leggett, “Computer aided parallelization tools (CAPTools) - Conceptual Overview and Performance on the Parallelization of Structured Mesh Codes”, Parallel Computing, vol. 22, pp.163-195, 1996.

7. H. Jin, M. Frumkin, and J. Yan, “Automatic Generation of OpenMP Directives and Its Application to Computational Fluid Dynamics Codes,” International Symposium on High Performance Computing, Tokyo, Japan, October 16-18, 2000, in *Lecture Notes in Computer Science*, Vol. 1940, pp. 440-456.

8. J. Laudon and D. Lenoski, ``The SGI Origin: A ccNUMA Highly Scalable Server," The 1997 International Symposium on Computer Architectures, Denver, CO, pp. 241--251.

9. M. Snir, et al. MPI: the complete reference. MIT Press, Cambridge, Mass., 1996.

10. M. Wolfe, High Performance Compilers for Parallel Computing. Addison-Wesley Publishing Company, 1996.