بسم الله الرحمن الرحیم

دانشکده ریاضی و علوم کامپیوتر

پایان‌نامه برای دریافت درجه کارشناسی ارشد در رشته علوم کامپیوتر

گرایش علوم تصمیم و دانش

نسخه موازی روش گرادیان مزدوج در محیط‌های توزیعی و اشتراکی

استاد راهنما

دکتر امین رفیعی

استاد مشاور

دکتر محمود امین طوسی

پژوهشگر:

زهرا اسعدی

مهر ۱۴۰۰

باسمه تعالی

فرم ارزشیابی و صورتجلسه دفاع از پایان‌نامه کارشناسی ارشد

فرم ۱۱۳-ت

جلسه دفاع از پایان‌نامه آقای /خانم **زهرا اسعدی** دانشجوی رشته علوم کامپیوتر گرایش علوم تصمیم و دانش به شماره دانشجویی ۹۷۱۳۱۸۵۳۱۶ با عنوان:

نسخه موازی روش گرادیان مزدوج در محیط‌های توزیعی و اشتراکی

در مورخه ..... در دانشکده ریاضی و علوم کامپیوتر تشکیل و توسط هیات داوران مورد ارزشیابی قرار گرفت و نمره ..... برابر درجه ..... برای آن تعیین گردید .

به‌این ترتیب از این تاریخ آقای/ خانم زهرا اسعدی به عنوان کارشناس ارشد در رشته مذکور شناخته می‌شود .

موارد ارزشیابی	موارد	حداکثر نمره	نمره کسب شده
۱- کیفیت نگارش	رعایت اصول نگارش انسجام در تنظیم بخشهای مختلف ،کیفیت تصاویر ، جداول و اشکال ،تنظیم فهرست ها ، منابع و ماخذ	٤	
۲- کیفیت علمی	بررسی تاریخچه و سابقه تجربی و نظری موضوع انسجام منطقی دربخش های مختلف پایان نامه ،ابتکار و نوآوری ،اهمیت و ارزش علمی پایان نامه ،استفاده از منابع معتبر و جدید ،کیفیت تجزیه و تحلیل یافته ها و نتیجه گیری ،روشن بودن روش کار ،هدف ها و فرضیه های تحقیق ،جدید بودن روش تحقیق	۱۰	
۳-کیفیت ارایه در جلسه دفاع	تسلط بر موضوع و بیان واضح وتفهیم آن ،توانایی در پاسخگویی به سوالات مطرح شده در جلسه ، رعایت زمان ارایه , روش ارائه	٤	
٤-ارزشیابی گزارشات	گزارش های دوره ای پیشرفت کار(حداقل ٤مورد)	۱	
٥- خروجی پایان نامه	مقاله مستخرج از پایان نامه: این نمره به صورت زیر اختصاص می یابد ۱) چکیده کنفرانسی هر مورد ۰/۲۵ نمره تا سقف۰/۵ نمره ۲) مقاله کامل در مجموع مقالات همایشهای معتبر یا مقاله در مجلات علمی-ترویجی معتبر پذیرفته شده یا چاپ شده هر مورد ۰/۵ نمره تا سقف ۱نمره ۳)مقاله پذیرفته شده یا چاپ شده در مجلات علمی پژوهشی معتبر ۱ نمره ٤)مقاله ارسال شدهبه مجلات علمی پژوهشی معتبرهر مورد ۲۵/. نمره تا سقف ۰/۵ نمره ٥) دستگاه ساخته شده دارای گواهی ثبت اختراع یا به سفارش سازمان ها تا سقف ۱ نمره ٦) دستگاه ساخته شده کاربردی که به تایید رئیس دانشکده رسیده باشد تا سقف ۰/۵ نمره	۱	
جمع			

درجه معادل کسب شده: (از ۱۹تا ۲۰ عالی☐ از۱۸تا ۱۸/۹۹بسیارخوب ☐ از ۱٦تا ۱۷/۹۹خوب ☐ از ۱٤ تا ۱۵/۹۹قابل قبول☐ کمتر از ۱٤ غیر قابل قبول☐)

مشخصات هیات دوران

ردیف	نام و نام خانوادگی	سمت	مرتبه علمی	محل کار	امضـــــــــــا
۱	دکتر امین رفیعی	استاد راهنما	دانشیار	دانشگاه حکیم سبزواری	
۲	دکتر محمود امین طوسی	استاد مشاور	استادیار	دانشگاه حکیم سبزواری	
۳		استاد داور	استادیار	دانشگاه حکیم سبزواری	
۴	دکتر غلامرضا مقدسی	نماینده تحصیلات تکمیلی	استادیار	دانشگاه حکیم سبزواری	

امضا

مدیر گروه

امضا

رئیس دانشکده

سوگند نامه دانش آموختگان دانشگاه حکیم سبزواری

به نام خداوند جان و خرد کزین برتر اندیشه بر نگذرد

اینک که به خواست آفریدگار پاک، کوشش خویش و بهره گیری از دانش استادان و سرمایه‌های مادی و معنوی این مرز و بوم، توشه‌ای از دانش و خرد گردآورده‌ام، در پیشگاه خداوند بزرگ سوگند یاد می‌کنم که در به کارگیری دانش خویش، همواره بر راه راست ودرست گام بردارم. خداوند بزرگ، شما شاهدان، دانشجویان و دیگر حاضران را به عنوان داورانی امین گواه می گیرم که ازهمه دانش و توان خود برای گسترش مرزهای دانش بهره گیرم و از هیچ کوششی برای تبدیل جهان به جایی بهتر برای زیستن، دریغ نورزم. پیمان می‌بندم که همواره کرامت انسانی را در نظر داشته باشم و همنوعان خود را در هر زمان و مکان تا سر حد امکان یاری دهم. سوگند می خورم که در به کارگیری دانش خویش به کاری که با راه و رسم انسانی، آیین پرهیزگاری، شرافت و اصول اخلاقی برخاسته از ادیان بزرگ الهی، به ویژه دین مبین اسلام، مباینت دارد دست نیازم. همچنین در سایه اصول جهان شمول انسانی و اسلامی، پیمان می‌بندم از هیچ کوششی برای آبادانی و سرافرازی میهن و هم میهنانم فروگذاری نکنم و خداوند بزرگ را به یاری طلبم تا همواره در پیشگاه او و در برابر وجدان بیدار خویش و ملت سرافراز، بر این پیمان تا ابد استوار بمانم.

نام و نام خانوادگی: زهرا اسعدی

تاریخ و امضا:

تأییدیه‌ی صحت و اصالت نتایج

باسمه تعالی

اینجانب زهرا اسعدی به شماره دانشجویی ۹۷۱۳۱۸۵۳۱۶ دانشجوی رشته علوم کامپیوتر مقطع تحصیلی کارشناسی ارشد تأیید می‌نمایم که کلیه‌ی نتایج این پایان‌نامه حاصل کار اینجانب و بدون هرگونه دخل و تصرف است و موارد نسخه برداری شده از آثار دیگران را با ذکر کامل مشخصات منبع ذکر کرده‌ام. درصورت اثبات خلاف مندرجات فوق، به تشخیص دانشگاه مطابق با ضوابط و مقررات حاکم (قانون حمایت از حقوق مؤلفان و مصنفان و قانون ترجمه و تکثیر کتب و نشریات و آثار صوتی، ضوابط و مقررات آموزشی، پژوهشی و انضباطی ...) با اینجانب رفتار خواهد شد و حق هرگونه اعتراض درخصوص احقاق حقوق مکتسب و تشخیص و تعیین تخلف و مجازات را از خویش سلب می‌نمایم. در ضمن، مسؤولیت هرگونه پاسخگویی به اشخاص اعم از حقیقی و حقوقی و مراجع ذی صلاح (اعم از اداری و قضایی) به عهده ی اینجانب خواهد بود و دانشگاه هیچ گونه مسؤولیتی در این خصوص نخواهد داشت.

نام و نام خانوادگی: زهرا اسعدی

تاریخ و امضا:

مجوز بهره برداری از پایان‌نامه

بهره برداری از این پایان‌نامه در چهارچوب مقررات کتابخانه و با توجه به محدودیتی که توسط استاد راهنما به شرح زیر تعیین می‌شود، بلامانع است:

☐ بهره برداری از این پایان‌نامه برای همگان بلامانع است.

☐ بهره برداری از این پایان‌نامه با اخذ مجوز از استاد راهنما، بلامانع است.

☐ بهره برداری از این پایان‌نامه تا تاریخ ..... ممنوع است.

استاد راهنما: دکتر امین رفیعی

تاریخ و امضا:

تقدیم به:

پدر و مادرم

سپاس خداوندگار حکیم را که با لطف بی کران خود، آدمی را زیور عقل آراست.
در آغاز وظیفه خود می دانم از زحمات بی دریغ استاد راهنمای خود، جناب آقای دکتر امین رفیعی، صمیمانه تشکر و قدردانی کنم که قطعاً بدون راهنمایی های ارزنده ایشان، این مجموعه به انجام نمی رسید.
از جناب آقای دکتر محمود امین طوسی که زحمت مطالعه و مشاوره این رساله را تقبل فرمودند و در آماده سازی این رساله، به نحو احسن اینجانب را مورد راهنمایی قرار دادند، کمال امتنان را دارم.
همچنین لازم می دانم از گروه پارسی‌لاتک در پاسخگویی به مشکلات کاربران کمال قدردانی را داشته باشم.
در پایان، بوسه می زنم بر دستان خداوندگاران مهر و مهربانی، پدر و مادر عزیزم و بعد از خدا، ستایش می‌کنم وجود مقدس شان را و تشکر می‌کنم از خانواده عزیزم به پاس عاطفه سرشار و گرمای امیدبخش وجودشان، که بهترین پشتیبان من بودند.

زهرا اسعدی

مهر ۱۴۰۰

فهرست مطالب

فهرست جداول ج

فهرست تصاویر د

چکیده ۱

پیش‌گفتار ۲

فصل ۱: آشنایی با مفاهیم اولیه ۳

۱-۱ مقدمه . . . ۳

۱-۲ پردازش موازی . . . ۳

۱-۳ طبقه‌بندی رایانه‌های موازی . . . ۴

۱-۴ معماری حافظه رایانه‌های موازی . . . ۶

۱-۵ برنامه‌نویسی موازی . . . ۶

۱-۶ حل دستگاه معادلات خطی . . . ۸

۱-۶-۱ روش گرادیان مزدوج . . . ۱۰

۱-۷ مثال عددی الگوریتم گرادیان مزدوج . . . ۱۲

۱-۸ ماتریس‌های تنک . . . ۱۴

۱-۹ نحوه اجرای برنامه موازی در محیط برنامه‌نویسی OpenMP . . . ۱۶

۱-۹-۱ اجرای برنامه موازی OpenMP در سیستم عامل ویندوز . . . ۱۷

۱-۹-۲ اجرای برنامه موازی OpenMP در سیستم عامل لینوکس . . . ۱۹

۱-۱۰ برخی از دستورات موجود در محیط برنامه‌نویسی OpenMP . . . ۲۰

۱-۱۰-۱ تعیین تعداد رشته‌ها و رتبه‌بندی آنان . . . ۲۰

۱-۱۰-۲ استفاده از حلقه for در محیط OpenMP . . . ۲۱

۱-۱۰-۳ ساختارهای هماهنگ‌سازی بین رشته‌ها . . . ۳۲

۱-۱۱ نحوه اجرای برنامه موازی در محیط برنامه‌نویسی MPI ۳۵
۱-۱۱-۱ اجرای برنامه موازی MPI در سیستم عامل ویندوز ۳۵
۱-۱۱-۲ اجرای برنامه موازی MPI در سیستم عامل لینوکس ۳۷
۱-۱۲ برخی از دستورات موجود در محیط برنامه‌نویسی MPI ۳۸
۱-۱۲-۱ تعیین تعداد پردازشگرها و رتبه‌بندی آنان ۳۸
۱-۱۲-۲ دستورات ارتباطی MPI ۳۹
۱-۱۲-۳ ارتباط بلوکی و غیربلوکی ۵۳

فصل ۲: جزئیات پیاده‌سازی الگوریتم گرادیان مزدوج در محیط OpenMP ۵۸
۲-۱ مقدمه ۵۸
۲-۲ زیربرنامه‌های موازی استفاده شده در الگوریتم گرادیان مزدوج ۵۹

فصل ۳: جزئیات پیاده‌سازی الگوریتم گرادیان مزدوج در محیط MPI ۶۵
۳-۱ مقدمه ۶۵
۳-۲ نحوه توزیع ماتریس و بردار بین پردازشگرها ۶۵
۳-۳ زیربرنامه‌های موازی استفاده شده در الگوریتم گرادیان مزدوج ۷۴

فصل ۴: پیاده‌سازی‌های عددی و نتایج آن ۸۰
۴-۱ مقدمه ۸۰
۴-۲ نتایج آزمایش‌های عددی در محیط OpenMP ۸۱
۴-۳ نتایج آزمایش‌های عددی در محیط MPI ۸۳
۴-۴ نتیجه‌گیری ۸۴

فهرست منابع ۸۵

پیوست آ: برنامه‌های استفاده شده در این پایان‌نامه ۸۶

واژه‌نامه فارسی به انگلیسی ۱۰۳

واژه‌نامه انگلیسی به فارسی ۱۰۴

فهرست جداول

۱-۱ دستورات کامپایل برنامه توسط کامپایلر GCC ۱۸
۱-۲ دستور کامپایل برنامه توسط کامپایلر Intel C++ در ویندوز ۱۹
۱-۳ دستورات کامپایل برنامه توسط کامپایلر Intel در سیستم عامل لینوکس ۲۰
۱-۴ دستورات کامپایل برنامه توسط کامپایلر OpenMPI در سیستم عامل لینوکس ۳۷
۱-۵ دستورات کامپایل برنامه MPI توسط کامپایلر Intel در سیستم عامل لینوکس ۳۸
۱-۶ انواع داده در MPI ۳۹
۱-۷ انواع عملگر در MPI ۵۲
۱-۸ دستورات بلوکی MPI و معادل غیربلوکی آن‌ها ۵۵

۴-۱ زمان اجرا و تعداد تکرارهای الگوریتم گرادیان مزدوج در محیط OpenMP ۸۱
۴-۲ زمان اجرا و تعداد تکرارهای الگوریتم گرادیان مزدوج در محیط MPI ۸۳

فهرست تصاویر

۱-۱ معماری ماشین‌هایSISD و نحوه اجرای دستورالعمل‌ها در آن ۴
۱-۲ معماری ماشین‌هایSIMD و نحوه اجرای دستورالعمل‌ها در آن ۴
۱-۳ معماری ماشین‌هایMISD و نحوه اجرای دستورالعمل‌ها در آن ۵
۱-۴ معماری ماشین‌هایMIMD و نحوه اجرای دستورالعمل‌ها در آن ۵
۱-۵ معماری حافظه رایانه‌های موازی ۶
۱-۶ الگوی انشعاب و اتصال رشته‌ها برای اجرای ناحیه موازی ۷
۱-۸ تنظیمات ویژوال استودیو برای پشتیبانی از OpenMP در ویندوز ۱۷
۱-۷ اجرای مثال ۱-۹-۱ در محیط ویندوز با استفاده از کامپایلر MSVC ۱۷
۱-۹ کامپایل و اجرای خط فرمانی مثال ۱-۹-۱ در ویندوز با استفاده از کامپایلر GNU ۱۸
۱-۱۰ کامپایل و اجرای خط فرمانی مثال ۱-۹-۱ با استفاده از کامپایلر اینتل در ویندوز ۱۹
۱-۱۱ کامپایل و اجرای خط فرمانی مثال ۱-۹-۱ با استفاده از کامپایلر GNU در لینوکس ۲۰
۱-۱۲ تنظیمات ویژوال استودیو برای اجرای برنامه MPI با استفاده از کامپایلر MS-MPI ۳۶
۱-۱۳ اجرای مثال ۱-۱۱-۱ با استفاده از کامپایلر MS-MPI در ویندوز ۳۶
۱-۱۴ تنظیمات ویژوال استودیو برای اجرای برنامه با استفاده از کامپایلر IntelMPI ۳۶
۱-۱۵ کامپایل و اجرای خط فرمانی مثال ۱-۱۱-۱ با کامپایلر OpenMPI در لینوکس ۳۷
۱-۱۶ نحوه توزیع داده‌ها در MPI_Bcast و MPI_Scatter ۴۳
۱-۱۷ نحوه جمع‌آوری داده‌ها در MPI_Gather ۴۴
۱-۱۸ نحوه جمع‌آوری داده‌ها در MPI_Allgather ۴۶

۲-۱ الگوریتم گرادیان مزدوج و بخش‌ها و زیربرنامه‌های موازی آن در محیط OPenMP ۵۹

۳-۱ توزیع ماتریس و بردار بین پردازشگرها در محیط MPI ۶۶
۳-۲ توزیع دو بردار b و nnzCounts بین پردازشگرها ۷۲
۳-۳ توزیع بردار IA بین پردازشگرها ۷۳
۳-۴ توزیع دو بردار AA و JA بین پردازشگرها ۷۴

۳-۵ الگوریتم گرادیان مزدوج و بخش‌ها و زیربرنامه‌های موازی آن در محیط MPI ۷۵
۳-۶ ضرب موازی ماتریس توزیعی A در بردار یکپارچه x در محیط MPI ۷۶
۳-۷ اجرای موازی جمع برداری در محیط MPI ۷۸
۳-۸ اجرای موازی ضرب داخلی دو بردار در محیط MPI ۷۹

۴-۱ مقایسه زمان اجرای الگوریتم سری با الگوریتم موازی گرادیان مزدوج در محیط OpenMP ۸۲
۴-۲ نمودار مقایسه زمان اجرا الگوریتم سری با الگوریتم موازی گرادیان مزدوج در محیط MPI ۸۴

آ-۱ خروجی فایل matrix_csr برای بعد ۵ . . . . . ۸۶
آ-۲ خروجی فایل omp_csr برای بعد ۱۰۰۰ . . . . . ۸۹

فرم چکیده ی پایان‌نامه ی دوره ی تحصیلات تکمیلی

مدیریت تحصیلات تکمیلی

دانشگاه حکیم سبزواری

نام خانوادگی دانشجو:اسعدی	نام:زهرا	ش. دانشجویی:۹۷۱۳۱۸۵۳۱۶
استاد راهنما: دکتر امین رفیعی		
استاد مشاور: دکتر محمود امین طوسی		
دانشکده ریاضی و علوم کامپیوتر	رشته: علوم کامپیوتر	گرایش: علوم تصمیم و دانش
مقطع: کارشناسی ارشد	تاریخ دفاع:مهر ۱۴۰۰	تعداد صفحات: ۱۰۵
عنوان پایان‌نامه: نسخه موازی روش گرادیان مزدوج در محیط‌های توزیعی و اشتراکی		
کلید واژه‌ها:گرادیان مزدوج، ماشین‌های موازی توزیعی و اشتراکی، زیرفضای کریلف		

چکیده:

امروزه یکی از مهم‌ترین مسائل در محاسبات علمی، حل دستگاه‌های معادلات خطی $Ax = b$ می‌باشد. برای حل این دستگاه‌ها می‌توان از روش‌های مستقیم یا تکراری استفاده نمود. روش گرادیان مزدوج از جمله روش‌های تکراری مبتنی بر زیرفضای کریلف می‌باشد که می‌تواند جهت یافتن تقریبی از جواب دستگاه مورد استفاده قرار گیرد. در مسائلی با ابعاد بالا، برای حل این دستگاه زمان و هزینه زیادی را متحمل خواهیم شد. از این رو مساله کاهش زمان محاسبات به منظور افزایش کارایی، در طول زمان همواره مورد توجه محققان بوده است. با ظهور معماری موازی در رایانه‌ها، برنامه‌نویسی موازی کمک قابل توجهی به کاهش زمان محاسبات داشته است.

در این پایان‌نامه به بررسی جزئیات پیاده‌سازی الگوریتم گرادیان مزدوج در هر یک از محیط‌های موازی OpenMP و MPI می‌پردازیم. در ادامه با استفاده از دستگاه‌های مصنوعی ساخته شده، زمان اجرای الگوریتم گرادیان مزدوج در هریک از این محیط‌ها را مورد تحلیل قرار داده و زمان اجرای الگوریتم را با تعداد هسته‌های متفاوت مقایسه می‌کنیم.

پیش‌گفتار

حل دستگاه‌های بزرگ خطی $Ax = b$ یکی از مباحث کلیدی محاسبات علمی می‌باشد که در سال‌های اخیر مورد تحقیقات گسترده‌ای قرار گرفته است. از آنجایی که این مسائل در بسیاری از برنامه‌های علمی مورد استفاده می‌گیرند، برای حل این دستگاه‌ها روش‌های متعددی پیشنهاد شده است که به دو دسته تقسیم می‌شوند: روش‌های مستقیم [۱] و تکراری [۲]. در مسائلی با ابعاد بالا، روش‌های تکراری نسبت به روش‌های مستقیم عملکرد بهتری خواهند داشت [۱]. روش زیرفضای کریلف [۳] یکی از مهمترین روش‌های تکراری برای حل دستگاه‌های معادلات خطی می‌باشد. اگر در دستگاه معادلات خطی، ماتریس ضرایب معین مثبت و متقارن باشد، جهت یافتن جواب دستگاه می‌توان از روش گرادیان مزدوج [۴] که از جمله روش‌های زیرفضای کریلف است استفاده نمود.

با افزایش حجم داده‌های ورودی، روند حل دستگاه با مشکل مواجه شده و با کاهش سرعت روبرو خواهیم شد. برای رفع این مساله و بهبود سرعت و افزایش کارایی، می‌توان از موازی‌سازی [۵] استفاده نمود. امروزه با افزایش توان پردازنده‌ها استفاده از تکنیک‌های موازی‌سازی نیز افزایش یافته است. هدف اصلی این پایان‌نامه استفاده از روش‌های موازی برای پیاده‌سازی الگوریتم گرادیان مزدوج و بهبود عملکرد و زمان اجرای این روش می‌باشد.

این پایان‌نامه شمال چهار فصل می‌باشد:

در فصل ۱، مفاهیم اصلی شامل محاسبات موازی، معماری ماشین‌های موازی و بستر مناسب برای برنامه‌نویسی موازی تعریف می‌شود. و همچنین به بررسی جزئیات الگوریتم گرادیان مزدوج می‌پردازیم.

در فصل ۲،جزئیات پیاده‌سازی الگوریتم گرادیان مزدوج در محیط موازی اشتراکی [۶] مطرح خواهد شد.

در فصل ۳، به نحوه تقسیم داده‌های ورودی بین پردازنده‌ها و جزئیات پیاده‌سازی الگوریتم در محیط برنامه‌نویسی توزیعی [۷] پرداخته می‌شوند.

و در انتها در فصل ۴، به بررسی نتایج پیاده‌سازی عددی الگوریتم گرادیان مزدوج در هریک از محیط‌های موازی می‌پردازیم. به این صورت که الگوریتم را بر روی ابعاد مختلفی از ماتریس‌ها اجرا کرده و زمان اجرای آن را در هریک از محیط‌های برنامه‌نویسی موازی مقایسه خواهیم کرد.

---

[1] directive [2] iterative [3] Krylov subspace methods [4] Conjugate Gradient [5] parallelize

[6] Shared [7] Distributed

فصل ۱

آشنایی با مفاهیم اولیه

۱-۱ مقدمه

در این فصل برخی از مفاهیم اولیه را بیان می‌کنیم. در ابتدا با توضیحاتی درباره محاسبات موازی و معماری ماشین‌های موازی شروع کرده و در ادامه استانداردهای برنامه‌نویسی در هریک از محیط‌های توزیعی و اشتراکی را معرفی می‌کنیم. سپس الگوریتم گرادیان مزدوج که یکی از روش‌های زیرفضای کریلف برای حل دستگاه معادلات خطی می‌باشد را بیان می‌کنیم. همچنین در مورد ماتریس‌های تنک [1] و فرمت‌های ذخیره‌سازی آن‌ها توضیحاتی می‌دهیم. در نهایت به نحوه اجرای یک برنامه موازی در محیط‌های اشتراکی و توزیعی پرداخته و برخی از توابع موجود در این دو محیط را معرفی می‌کنیم.

۱-۲ پردازش موازی

برای مدت‌های طولانی با افزایش سرعت سی‌پی‌یو[2]ها طراحان سخت‌افزار و نویسندگان کامپایلر سعی در افزایش کارایی برنامه‌نویسی سری داشتند، اما با ظهور معماری‌های پیچیده وضعیت تغییر کرد و صنعت کامپیوتر پذیرفت که به‌جای افزایش سرعت سی‌پی‌یوها تعداد آن‌ها بیشتر شود تا کارایی بیشتری داشته باشند. این تغییر به سمت پردازنده‌های چندهسته‌ای، برنامه‌نویسی را از فرم معمول سری [3] به سمت موازی‌سازی سوق داده است [۲].

محاسبات موازی استفاده از رایانه موازی برای کاهش زمان حل یک مساله محاسباتی می‌باشد [۳]. ایده اصلی در آن این است که یک مساله را به زیرمسائل کوچک‌تری تقسیم کرد تا با اجرای همزمان این زیرمسائل و هماهنگ‌کردن آن‌ها مساله اصلی در زمان کوتاه‌تری حل شود.

---

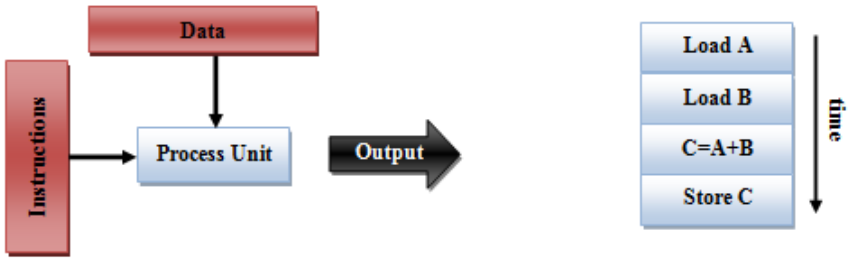
[1] sparse [2] CPU [3] Sequential

۱-۳ طبقه‌بندی رایانه‌های موازی

طبقه‌بندی فلین [۱] معروفترین دسته‌بندی برای رایانه‌های موازی است که در آن معماری رایانه براساس میزان تعامل بین جریان دستورالعمل‌ها [۲] و جریان داده‌ها [۳] به چهار دسته تقسیم می‌شوند [۴]:

- **یک دستورالعمل یک داده (SISD) [۴]:**

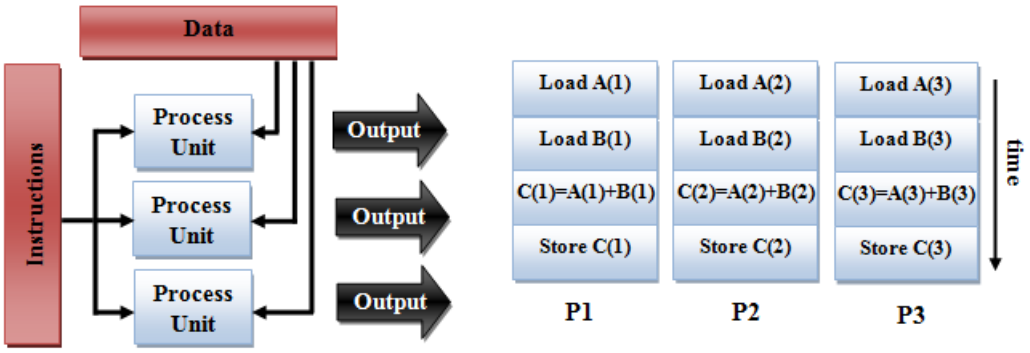
یک واحد پـردازش [۵] وجـود دارد که یک دستورالعمل واحد را از حافظه [۶] دریافت می‌کند و آن را بر روی یک جریان داده واحد اجرا می‌کند. در هر زمـان یک عمـل انجـام می‌شود. نمونه‌هایی از این معمـاری، رایانه‌های تک‌پردازنـده‌ای [۷] قدیمی هستند.



شکل ۱-۱: معماری ماشین‌هایSISD و نحوه اجرای دستورالعمل‌ها در آن

- **یک دستورالعمل چند داده (SIMD) [۸]:**

چندین واحد پردازش وجود دارد که هر واحد پردازش دستورالعمل یکسانی را در هر چرخه زمانی مشخص، بر روی یک عنصر داده متفاوت اجرا می‌کند. این معماری توسط بسیاری از رایانه‌های موازی امروزی مورد استفاده قرار می‌گیرد.



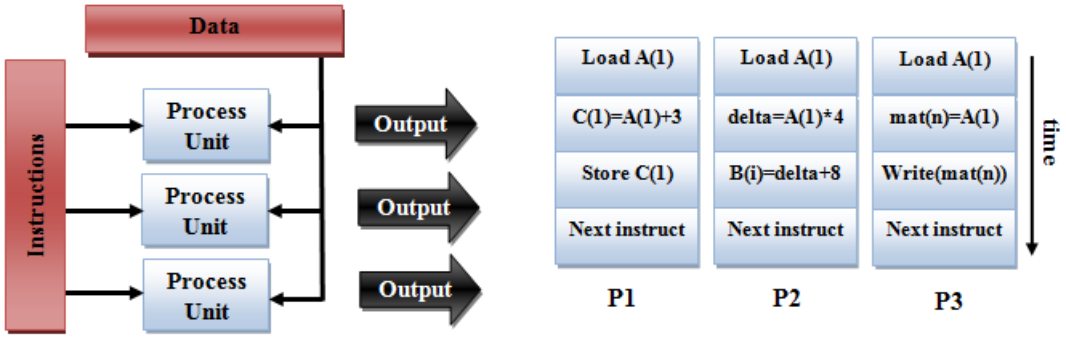
شکل ۱-۲: معماری ماشین‌هایSIMD و نحوه اجرای دستورالعمل‌ها در آن

---

[1]Flynn's taxonomy [2]instructions flow [3]Data flow [4]Single Instructions, Single Data
[5]Process Unit [6]memory [7]Uni-Processor [8]Single Instructions, Multiple Data

- **چند دستورالعمل یک داده (MISD)** [1]:

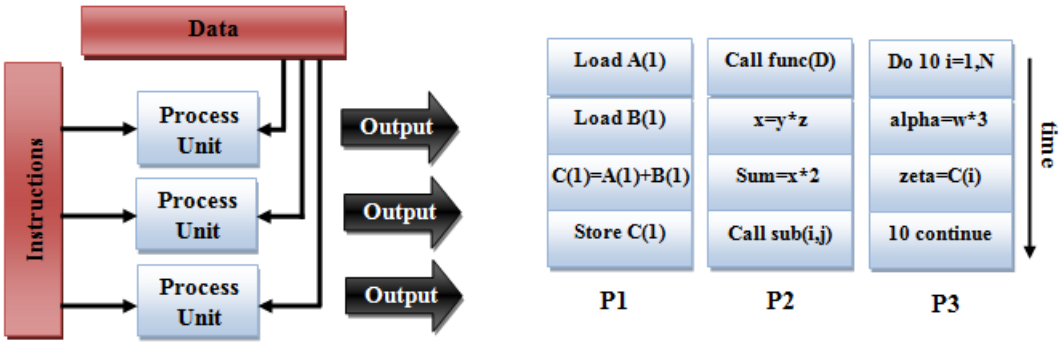
واحدهای پردازش متعددی وجود دارند که یک جریان داده واحد به آن‌ها داده می‌شود .هر واحد پردازشی بطور مستقل، جریان‌های دستورالعمل جداگانه را بر روی داده‌ها اعمال می‌کند. هیچ رایانه موازی تجاری از این نوع ساخته نشده است.



شکل ۱-۳: معماری ماشین‌هایMISD و نحوه اجرای دستورالعمل‌ها در آن

- **چند دستورالعمل چند داده (MIMD)** [2]:

چندین پردازنده مستقل به صورت همزمان دستورالعمل‌های مختلف را روی داده های مختلف اجرا می‌کنند. معماری SIMD را می‌توان زیر مجموعه‌ای از این معماری در نظر گرفت. پردازنده‌های چندهسته‌ای [3] و رایانه‌های خوشه‌ای [4] نمونه‌هایی از MIMD هستند.



شکل ۱-۴: معماری ماشین‌هایMIMD و نحوه اجرای دستورالعمل‌ها در آن

[1]Multiple Instructions, Single Data [2]Multiple Instructions, Multiple Data [3]Multi-Core Processors

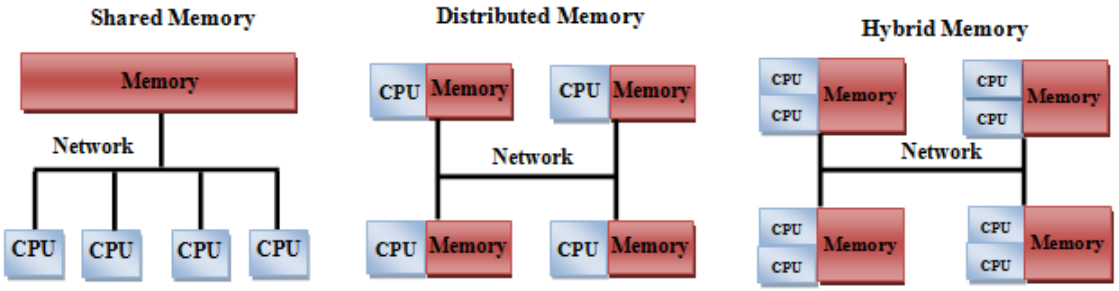
[4]Cluster Computer

۱-۴ معماری حافظه رایانه‌های موازی

براساس ساختار فیزیکی، حافظه رایانه‌های موازی را می‌توان به دو دسته تقسیم نمود: حافظه اشتراکی [۱] (چندپردازنده) و حافظه توزیعی [۲]. اما ساختار دیگری از حافظه نیز وجود دارد که ترکیبی از حافظه اشتراکی و توزیع‌شده می‌باشد [۳].

- **حافظه اشتراکی**: رایانه‌ها با حافظه اشتراکی فیزیکی، ماشین‌های حافظه اشتراکی [۴] (SMM) نامیده می‌شوند. این نوع رایانه‌ها شامل تعدادی پردازنده (هسته [۵])، حافظه فیزیکی مشترک (حافظه سراسری) و یک شبکه اتصال [۶] برای ارتباط پردازنده با حافظه هستند. شبکه ارتباطی در این نوع از رایانه‌ها، به صورت داخلی [۷] می‌باشد. هسته‌های واحد پردازش، دسترسی مشترکی به حافظه سراسری دارند یعنی به صورت مشترک از حافظه برای خواندن و نوشتن استفاده می‌کنند.

- **حافظه توزیعی**: رایانه‌ها با حافظه توزیع‌شده فیزیکی، که به نام ماشین‌های حافظه توزیع‌شده [۸] (DMM) نیز شناخته می‌شوند، شامل تعدادی عناصر پردازشی به نام گره [۹] و یک شبکه اتصال است که گره‌ها را به هم متصل می‌کند .هر گره یک واحد مستقل است که پردازنده و حافظه جداگانه و مخصوص به خود را دارد و تنها پردازشگر محلی می‌تواند مستقیما به حافظه محلی خود دسترسی داشته باشد. در رایانه‌هایی با حافظه توزیع‌شده، شبکه نقش اساسی دارد؛ زمانی که یک پردازنده برای محاسبات به داده‌ای از حافظه محلی گره‌های دیگر نیاز داشته باشد، باید انتقال پیام از طریق بستر شبکه انجام شود. در واقع ساختار توزیع‌شده متشکل از سیستم‌های اشتراکی است که از طریق شبکه به تبادل اطلاعات با یکدیگر می‌پردازند [۵].



شکل ۱-۵: معماری حافظه رایانه‌های موازی

۱-۵ برنامه‌نویسی موازی

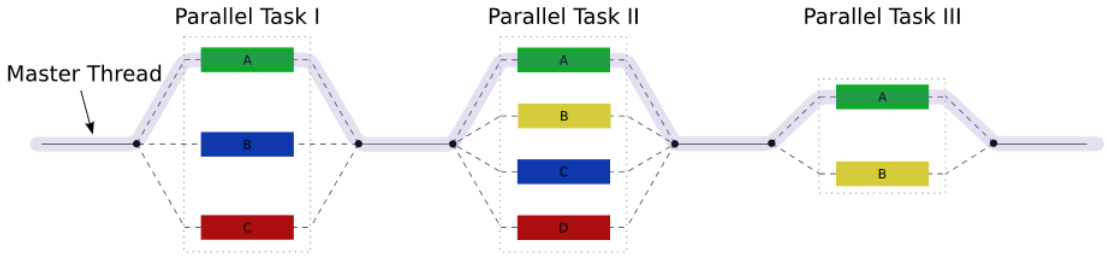
برنامه‌نویسی موازی، برنامه‌نویسی به زبانی است که به شما اجازه می‌دهد تا مشخص کنید چه بخش‌هایی از برنامه به طور همزمان توسط پردازنده‌های مختلف اجرا شود [۳]. برنامه‌های سری که برای سیستم‌های تک‌هسته‌ای نوشته شده‌اند،

[1]Shared Memory [2]Distributed Memory [3]Hybrid Memory [4]Shared Memory Machine [5]core
[6]Connection Network [7]on board [8]Distributed Memory Machine [9]node

در سیستم‌های چندهسته‌ای کارایی ندارند. برای حل این مساله می‌توانیم از دستورالعمل‌های کامپایلر و برنامه‌های مترجم [1] برای اجرای برنامه‌های سری استفاده کنیم، تا برنامه را به صورت خودکار و بدون دخالت برنامه‌نویس به یک زیربرنامه موازی تبدیل کنیم. یا اینکه می‌توانیم از استانداردهای مختلف برنامه‌نویسی موازی استفاده کنیم و برنامه‌های سری را به‌صورت موازی بنویسیم تا بتوانند از چندین هسته استفاده کنند. در ادامه دو استاندارد برنامه‌نویسی موازی را معرفی می‌کنیم [۶].

- **OpenMP** [2]:

در سال ۱۹۹۷ یک رابط برنامه‌نویسی کاربردی [3] (API) مقیاس‌پذیر [4] به طور مشترک توسط گروهی از فروشندگان بزرگ سخت‌افزار و نرم‌افزار طراحی شد. OpenMP یک زبان برنامه‌نویسی نیست، بلکه یک استاندارد ارتباطی برای برنامه‌نویسی موازی بر روی سیستم‌هایی با حافظه اشتراکی می‌باشد که با استفاده از آن می‌توان درهریک از سیستم‌عامل‌های ویندوز، لینوکس، مک‌اواس [5] و... به هریک از زبان‌های برنامه‌نویسی C، C++ یا Fortran برنامه نوشت تا به صورت موازی از منابع یک سیستم کامپیوتری استفاده‌کند [۷]. مدل برنامه‌نویسی OpenMP بر پایه همکاری رشته [6] هایی است که به‌طورهمزمان بر روی چند پردازنده یا هسته اجرا می‌شوند. این رشته‌ها در قالب یک الگوی انشعاب-اتصال [7] ایجاد و نابود می‌شوند. اجرای یک برنامه با یک رشته واحد (رشته اولیه) شروع می‌شود، که برنامه را به شکل سری اجرا می‌کند، تا زمانی که به اولین ناحیه موازی برخورد کند. در این هنگام عملیات انشعاب رخ می‌دهد، به این صورت که رشته اولیه مجموعه‌ای از رشته‌ها، شامل رشته‌های جدید و خود رشته(رشته اصلی) را ایجاد می‌کند. در پایان یک منطقه موازی، یک سد هماهنگ‌ساز [8] وجود دارد که باعث می‌شود تنها رشته اصلی پس از این منطقه اجرای خود را ادامه دهد (عملیات اتصال رشته‌ها). مناطق موازی می‌توانند تودرتو باشند و هر رشته تولید شده در مواجه با یک ناحیه موازی، می‌تواند خود یک مجموعه از رشته‌ها را ایجاد کند [۵].



شکل ۱-۶: الگوی انشعاب و اتصال رشته‌ها برای اجرای ناحیه موازی

[1] translation programs [2] Open Multi Processing [3] Application Programming Interface [4] scalable
[5] Macintosh Operating system(MACOS) [6] thread [7] fork/join [8] Barrier Synchronization

- **MPI**[1]:

یک کتابخانه استاندارد شده ارتباط پیام برای برنامه‌نویسی موازی بر روی سیستم‌های موازی توزیعی می‌باشد که نظیر OpenMP بر روی سیستم‌عامل‌های متفاوت و در زبان‌های برنامه‌نویسی متفاوت می‌توان برنامه‌های موازی را اجرا نمود. MPI شامل مجموعه‌ای از توابع ارتباطی است که باعث نقل و انتقالات داده‌ها می‌شود. ساده‌ترین تابع انتقال داده، باعث انتقال نقطه‌به‌نقطه [2] بین دو پردازشگر می‌شود که یکی از آن‌ها عملیات ارسال و دیگری عملیات دریافت را برعهده دارد. همچنین توابع پیشرفته‌تری نیز وجود دارد که در آن‌ها بیش از دو پردازشگر درگیر هستند. یک برنامه توزیعی توسط مجموعه‌ای از پردازنده‌ها اجرا می‌شود، که در آن هر پردازنده داده‌های محلی خودش را دارد. تعداد پردازنده‌ها اغلب هنگام شروع برنامه مشخص می‌شوند. هر پردازنده به داده‌های محلی خود دسترسی دارد و می‌تواند با ارسال و دریافت پیام‌ها، اطلاعات و داده‌ها را با دیگر پردازنده‌ها مبادله کند [۵].

۱-۶ حل دستگاه معادلات خطی

دستگاه‌های معادلات خطی یکی از مهم‌ترین مسائل در محاسبات علمی می‌باشد که به صورت زیر نمایش داده می‌شود:

$$Ax = b \tag{۱-۱}$$

که در آن $A \in \mathbb{R}^{n\times n}$ ماتریس ضرایب و $x, b \in \mathbb{R}^{n\times 1}$ به ترتیب بردار مجهولات و بردار سمت راست نامیده می‌شوند. روش‌های زیادی برای حل دستگاه فوق موجود است که به دو دسته مستقیم و تکراری تقسیم می‌شوند. در روش‌های مستقیم پس از اعمال برخی عملیات های محاسباتی بر روی ماتریس ضرایب، به جواب دقیق دستگاه خواهیم رسید. اما در مسائل واقعی که ماتریس ضرایب در آن بزرگ و با پراکندگی زیاد می‌باشد، به دلیل حجم بالای محاسبات و ذخیره‌سازی محدود در رایانه‌ها روش‌های مستقیم مناسب نبوده و از روش‌های تکراری استفاده می‌کنیم. روش‌های تکراری شامل روش‌هایی است که از یک بردار اولیه شروع کرده و پس از انجام چندین مرحله عملیات تکراری در صورت امکان به سمت جواب واقعی همگرا خواهند شد. روش‌های تکراری، خود شامل دو دسته ایستا و غیرایستا می‌شوند. مبنای روش‌های تکراری ایستا، شکافت (تجزیه) ماتریس ضرایب می‌باشند که نمونه‌ای از این روش‌ها ژاکوبی، گاوس-سایدل، SOR و... می‌باشد، جهت اطلاع از این روش‌ها می‌توانید به مرجع [۸] مراجعه کنید.

روش‌های زیرفضای کریلف جزو روش‌های تکراری غیرایستا هستند که در زمره روش‌های تصویری نیز می‌باشند. در این روش‌ها ابتدا با یک حدس x_0 اولیه شروع کرده و سپس جواب تقریبی دستگاه یعنی x_m را در زیرفضای $x_0 + \mathcal{K}_m$

[1] Message Passing Interface [2] point-to-point

جستجو می‌کنیم. $\mathcal{K}_m = \mathcal{K}_m(A, r_0)$ زیرفضای کریلف از بعد m می‌باشد که به صورت زیر تعریف می‌شود:

$$\mathcal{K}_m(A, r_0) = span\{r_0, Ar_0, \ldots, A^{m-1}r_0\} \qquad (۲-۱)$$

در رابطه (۱-۲)، $r_0 = b - Ax_0$ بردار مانده اولیه می‌باشد. جهت محاسبه جواب تقریبی x_m باید شرایط پتروف-گالرکین[1] برقرار باشد یا به عبارتی بردار مانده $b - Ax_m$ باید بر زیرفضای دیگری نظیر L_m که از بعد m است، عمود باشد. دو انتخاب به صورت زیر برای زیرفضای L_m وجود دارد:

۱. $L_m = A\mathcal{K}_m$

۲. $L_m = \mathcal{K}_m$

که انتخاب اول منجر به ارائه روش زیرفضای کریلف مانده مینیمال تعمیم یافته (GMRES) [2] می‌شود. و در حالتی که ماتریس A معین‌مثبت و متقارن باشد انتخاب دوم منجر به روش زیرفضای کریلف گرادیان مزدوج خواهد شد.

برای محاسبه پایه‌های یکامتعامد زیرفضای $\mathcal{K}_m(A, r_0)$ می‌توان از الگوریتم آرنولدی که به شکل زیر می‌باشد، استفاده کرد. چنانچه در این الگوریتم بردار اولیه $v_1 = \frac{r_0}{\|r_0\|_2}$ انتخاب شود آن‌گاه بردارهای $v_1, v_2, \ldots, v_m$ محاسبه شده از این الگوریتم، پایه‌های یکا متعامدی برای زیرفضای کریلف خواهند بود. مبنای الگوریتم آرنولدی روش گرام‌اشمیت [3] اصلاح شده می‌باشد.

الگوریتم ۱-۱ الگوریتم آرنولدی

Input: Choose a vector v_1 such that $\|v_1\|_2 = 1$.
Output: vectors $v_1, v_2, \ldots, v_m$.

for $j = 0, 1, ..., m$ **do**
  $w_j = Av_j$
  for $i = 0, 1, ..., j$ **do**
    $h_{i,j} = (w_j, v_j)$
    $w_j = w_j - h_{i,j}v_j$
  end for
  $h_{j+1,j} = \|w_j\|_2$
  if $h_{j+1,j} = 0$ **then**
    break
  end if
  $v_{j+1} = w_j / h_{j+1,j}$
end for

[1] Petrov–Galerkin [2] Generalized Minimal Residual [3] Gram–Schmidt

۱-۶-۱ روش گرادیان مزدوج

این روش نخستین بار در سال ۱۹۵۲ توسط هستنس [1] و استایفل [2] به صورت مستقل از یکدیگر ارائه گردیده است. اگر ماتـریس ضرایب معین‌مثبت و متقارن باشد، روش گرادیان مزدوج که از جمله روش‌های زیرفـضای کریلف است می‌تواند جهت یافتن جواب تقریبـی از جواب دستگاه (۱-۱) مورد استفـاده قرار گیرد. روش کار الگـوریتم گرادیان مزدوج در ادامه آورده شده است.

الگوریتم ۱-۲ الگوریتم گرادیان مزدوج

Input: Choose a vector x_0 and ε_0.
Output: approximate solution x_{j+1}.

$r_0 = b - Ax_0$, $p_0 = r_0$
for $j = 0, 1, ...$ **do**
  $w_j = Ap_j$
  $\alpha_j = (r_j, r_j)/(w_j, p_j)$
  $x_{j+1} = x_j + \alpha_j p_j$
  $r_{j+1} = r_j - \alpha_j w_j$
  if $\|r_{j+1}\|_2^2 \leq \varepsilon_0$ **then**
    break
  end if
  $\beta_j = (r_{j+1}, r_{j+1})/(r_j, r_j)$
  $p_{j+1} = r_{j+1} + \beta_j p_j$
end for

در الگوریتم فوق (r_j, r_j) به معنی ضرب داخلی دو بردار می‌باشد. در این پایان‌نامه برای نمایش ضرب داخلی از این نماد استفاده شده است. در ادامه چگونگی محاسبه اسکالرهای α_j و β_j که در خطوط ۵ و ۹ الگوریتم مطرح شده‌اند را به کمک قضیه (۱-۶-۱) به اختصار بیان خواهیم کرد.

قضیه ۱-۶-۱. فرض کنید الگوریتم (۱-۲) اجرا شده باشد، در این الگوریتم جواب‌های تقریبی x_{j+1} به شکل

$$x_{j+1} = x_j + \alpha_j p_j \qquad (۱-۳)$$

محاسبه می‌شوند که p_j بردارهای مسیر می‌باشند و با استفاده از رابطه

$$p_{j+1} = r_{j+1} + \beta_j p_j \qquad (۱-۴)$$

[1] Hestenes [2] Stiefel

به‌هنگام خواهند شد. در رابطه فوق r_{j+1} بردار مانده نظیر به x_{j+1} می‌باشد که به صورت

$$r_{j+1} = r_j - \alpha_j A p_j \tag{۵-۱}$$

محاسبه گردیده است. به ازای $m = 0, 1, \ldots$ بردارهای مانده $r_m = b - Ax_m$ و بردارهای مسیر p_m در شرایط زیر صدق می‌کنند:

۱. بردارهای مانده r_m بر هم عمود هستند یعنی: $(r_i, r_j) = 0, \quad i \neq j$

۲. بردارهای مسیر p_j، A-متعامد هستند یعنی: $(Ap_i, p_j) = 0, \quad i \neq j$

برای اثبات می‌توانید به مرجع [۸] مراجعه کنید.

نتیجه: رابطه (۱-۵) را در نظر بگیرید، از آنجایی که بردارهای r_j بر هم عمود می‌باشند پس $(r_j - \alpha_j A p_j, r_j) = 0$ در نتیجه داریم:

$$\alpha_j = \frac{(r_j, r_j)}{(Ap_j, r_j)} \tag{۶-۱}$$

از رابطه (۱-۴) و با توجه به اینکه بردارهای مسیر بر هم A-متعامد هستند، خواهیم داشت:

$$(Ap_j, r_j) = (Ap_j, p_j - \beta_{j-1} p_{j-1}) = (Ap_j, p_j)$$

پس رابطه (۱-۶) به شکل زیر تبدیل خواهد شد:

$$\alpha_j = \frac{(r_j, r_j)}{(Ap_j, p_j)}.$$

همانطور که گفته شد طبق رابطه (۱-۴) بردارهای مسیر به‌روز گردیدند. برای محاسبه‌ی β_j با استفاده از این خاصیت که بردارهای مسیر برهم A-متعامد هستند داریم:

$$\beta_j = -\frac{(r_{j+1}, Ap_j)}{(p_j, Ap_j)} \tag{۷-۱}$$

با توجه به رابطه (۱-۵)

$$Ap_j = -\frac{1}{\alpha_j}(r_{j+1} - r_j)$$

و در نهایت با جایگذاری آن در رابطه (۱-۷) خواهیم داشت:

$$\beta_j = \frac{۱}{\alpha_j}\frac{(r_{j+۱}, (r_{j+۱} - r_j))}{(Ap_j, p_j)} = \frac{r_{j+۱}, r_{j+۱}}{r_j, r_j}. \tag{۱-۸}$$

۱-۷ مثال عددی الگوریتم گرادیان مزدوج

در ادامه برای درک بهتر الگوریتم گرادیان مزدوج (الگوریتم۱-۲) ، چند گام از یک مثال عددی را شرح خواهیم داد:

مثال ۱-۷-۱. در این مثال از یک دستگاه مصنوعی مشخص استفاده کرده‌ایم که در آن ماتریس ضرایب یک ماتریس معین مثبت و متقارن است و جواب دقیق آن بردار تمام یک می‌باشد.

$$\underbrace{\begin{bmatrix} ۰/۶۱ & ۰ & ۰/۷۴ \\ ۰ & ۰/۹۹ & ۰ \\ ۰/۷۴ & ۰ & ۱/۲۴ \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} ۱ \\ ۱ \\ ۱ \end{bmatrix}}_{x} = \underbrace{\begin{bmatrix} ۱/۳۵ \\ ۰/۹۹ \\ ۱/۹۸ \end{bmatrix}}_{b}$$

حل: ابتدا بردار حدس اولیه $x_۰$ را یک بردار تمام صفر در نظر می‌گیریم و مطابق خط شماره ۱ الگوریتم داریم:

$$r_۰ = p_۰ = b - Ax_۰ = \begin{bmatrix} ۱/۳۵ \\ ۰/۹۹ \\ ۱/۹۸ \end{bmatrix}$$

گام اول: $(i = ۰)$

مطابق با خط شماره ۳ الگوریتم باید حاصلضرب ماتریس تنک A در بردار $p_۰$ محاسبه شود.

$$w_۰ = Ap_۰ = \begin{bmatrix} ۰/۶۱ & ۰ & ۰/۷۴ \\ ۰ & ۰/۹۹ & ۰ \\ ۰/۷۴ & ۰ & ۱/۲۴ \end{bmatrix} \begin{bmatrix} ۱/۳۵ \\ ۰/۹۹ \\ ۱/۹۸ \end{bmatrix} = \begin{bmatrix} ۲/۲۸۸۷ \\ ۰/۹۸۰۱ \\ ۳/۴۵۴۲ \end{bmatrix}$$

سپس با انجام عملیات ضرب داخلی در خط ۴ الگوریتم مقدار اسکالر α را به دست می‌آوریم.

$$(r_۰, r_۰) = \begin{bmatrix} ۱/۳۵ & ۰/۹۹ & ۱/۹۸ \end{bmatrix} \begin{bmatrix} ۱/۳۵ \\ ۰/۹۹ \\ ۱/۹۸ \end{bmatrix} = ۶/۷۲۳$$

$$(w_۰, p_۰) = \begin{bmatrix} ۲/۲۸۸۷ & ۰/۹۸۰۱ & ۳/۴۵۴۲ \end{bmatrix} \begin{bmatrix} ۱/۳۵ \\ ۰/۹۹ \\ ۱/۹۸ \end{bmatrix} = ۱۰/۸۹۹$$

$$\alpha_۰ = \frac{(r_۰, r_۰)}{(w_۰, p_۰)} = \frac{۶/۷۲۳}{۱۰/۸۹۹} = ۰/۶۱۶۸$$

حال در خط ۵ با استفاده از اسکالر α و بردار جهت p جواب دستگاه را به‌روز می‌کنیم.

$$x_۱ = x_۰ + \alpha_۰ p_۰ = \begin{bmatrix} ۰ \\ ۰ \\ ۰ \end{bmatrix} + ۰/۶۱۶۸ \begin{bmatrix} ۱/۳۵ \\ ۰/۹۹ \\ ۱/۹۸ \end{bmatrix} = \begin{bmatrix} ۰/۸۳۲۷ \\ ۰/۶۱۰۷ \\ ۱/۲۲۱۳ \end{bmatrix}$$

و بردار مانده r در خط ۶ الگوریتم به صورت زیر به‌روز خواهد شد.

$$r_۱ = r_۰ - \alpha_۰ w_۰ = \begin{bmatrix} ۱/۳۵ \\ ۰/۹۹ \\ ۱/۹۸ \end{bmatrix} - ۰/۶۱۶۸ \begin{bmatrix} ۲/۲۸۸۷ \\ ۰/۹۸۰۱ \\ ۳/۴۵۴۲ \end{bmatrix} = \begin{bmatrix} -۰/۰۶۱۷ \\ ۰/۳۸۵۴ \\ -۰/۱۵۰۶ \end{bmatrix}$$

پس از اینکه شرط همگرایی مطابق با خط ۷ الگوریتم ($\|r_{j+۱}\|_۲ \leq \varepsilon$) بررسی شود. مشاهده می‌کنیم که شرط همگرایی برقرار نشده است لذا اسکالر β به صورت زیر در خط ۱۰ محاسبه می‌گردد.

$$(r_۱, r_۱) = \begin{bmatrix} -۰/۰۶۱۷ & ۰/۳۸۵۴۵ & -۰/۱۵۰۶ \end{bmatrix} \begin{bmatrix} -۰/۰۶۱۷ \\ ۰/۳۸۵۴ \\ -۰/۱۵۰۶ \end{bmatrix} = ۰/۱۷۵۱$$

$$\beta_۰ = \frac{(r_۱, r_۱)}{(r_۰, r_۰)} = \frac{۰/۱۷۵۱}{۶/۷۲۳} = ۰/۰۲۶۰۴$$

و در انتها در خط ۱۱ الگوریتم بردار جهت با استفاده از اسکالر β به‌هنگام خواهد شد.

$$p_1 = r_1 + \beta_0 p_0 = \begin{bmatrix} -0/0617 \\ 0/3854 \\ -0/1506 \end{bmatrix} + 0/02604 \begin{bmatrix} 1/35 \\ 0/99 \\ 1/98 \end{bmatrix} = \begin{bmatrix} -0/0265 \\ 0/4112 \\ -0/0991 \end{bmatrix}$$

حال گام‌های بعدی از الگوریتم، مشابه گام اول انجام می‌شوند تا زمانی که شرط همگرایی الگوریتم در خط شماره ۷ الگوریتم برقرار شده و الگوریتم متوقف شود.

در این مثال برای حل دستگاه $Ax = b$ با استفاده از الگوریتم گرادیان مزدوج، ماتریس A یک ماتریس با تراکم بالا در نظر گرفته شده بود اما در بسیاری از مسائل دنیای واقعی ماتریس ضرایب یک ماتریس تنک می‌باشد. در این پایان‌نامه قصد داریم تا بر روی وضعیتی که ماتریس A تنک است، تمرکز کنیم. لذا در ادامه اشاره‌ای به ماتریس‌های تنک و نحوه ذخیره‌سازی آن‌ها خواهیم داشت.

۱-۸ ماتریس‌های تنک

ماتریس تنک ماتریسی است که بیشتر عناصر آن را درایه‌های صفر تشکیل می‌دهد. ماتریس‌ها در حافظه دسترسی تصادفی [1] کامپیوتر به صورت خطی و با استفاده از آدرس‌های مجاور به ترتیب سطر یا ستون ذخیره می‌شوند. بنابراین، برای ذخیره یک ماتریس دوبعدی، ابتدا باید فضای آدرس دوبعدی به فضای آدرس یک بعدی نگاشت شود. الگوریتم گرادیان مزدوج امروزه برای حل سیستم‌های خطی بزرگ و تنک مورد استفاده قرار می‌گیرد [۱]. ذخیره‌سازی ماتریس‌های تنک با ابعاد بزرگ حجم زیادی از حافظه را اشغال می‌کنند. به منظور افزایش کارایی و کاهش حجم موردنیاز برای ذخیره کردن ماتریس‌های تنک می‌توان از ذخیره عناصر صفر در ماتریس تنک جلوگیری کرد. روش‌های ذخیره‌سازی زیادی برای ماتریس‌های تنک مطرح شده که ما در اینجا به فرمت ذخیره‌سازی فشرده سطری [2] (CSR) و فشرده ستونی [3] (CSC) می‌پردازیم.

- **روش فشرده سطری:** در این فرم ذخیره‌سازی از سه آرایه استفاده می‌شود.

- **آرایه حقیقی AA:** در این آرایه، درایه‌های غیرصفر ماتریس به صورت سطر به سطر ذخیره می‌شوند با فرض اینکه تعداد عناصر غیرصفر ماتریس NNz [4] باشد طول آرایه AA، NNz خواهد بود.
- **آرایه صحیح JA:** طول این آرایه به اندازه NNz یا تعداد درایه‌های غیرصفر کل ماتریس خواهد بود. مقادیر ذخیره شده در آرایه شامل اندیس ستون عناصر غیرصفرسطرهای ماتریس می‌باشد.

---

[1] Random Access Memory(RAM) [2] Compress Sparse Row (CSR) [3] Compress Sparse Column (CSC) [4] Number of Non Zero

- **آرایه صحیح IA:** از عناصر این آرایه به عنوان اشاره‌گر استفاده خواهد شد. به عنوان مثال IA(i) به موقعیت شروع سطر i ام در آرایه‌های AA و JA اشاره می‌نماید. یعنی مکان AA[IA(i)] موقعیت شروع عناصر غیرصفر سطر i ام ماتریس A می‌باشد. به همین ترتیب JA[IA(i)] موقعیت شروع اندیس ستونی درایه‌های غیرصفر سطر i ام ماتریس A است. طول این آرایه به اندازه n+1 می‌باشد که n بعد ماتریس است. در خانه شماره n+1 ام این آرایه تعداد عناصر غیرصفر ماتریس یا NNz ذخیره خواهد شد.

به عنوان مثال برای ماتریس زیر فرمت ذخیره‌سازی سطری را نشان می‌دهیم:

$$A = \begin{bmatrix} ۱ & ۰ & ۰ & ۲ & ۰ \\ ۳ & ۴ & ۰ & ۵ & ۰ \\ ۶ & ۰ & ۷ & ۸ & ۹ \\ ۰ & ۰ & ۱۰ & ۱۱ & ۰ \\ ۰ & ۰ & ۰ & ۰ & ۱۲ \end{bmatrix}$$

AA=

۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰	۱۱	۱۲

JA=

۱	۴	۱	۲	۴	۱	۳	۴	۵	۳	۴	۵

IA=

۱	۳	۶	۱۰	۱۲	۱۳

از موقعیت IA(i) تا IA(i+1)-1 عناصر آرایه‌های AA و JA شامل مقادیر عددی و اندیس‌های ستونی درایه‌های غیرصفر سطر i ام ماتریس می‌باشند.

- **روش فشرده ستونی:** این فرمت ذخیره‌سازی نیز با استفاده از سه آرایه انجام خواهد گرفت.

- **آرایه حقیقی AA:** در این آرایه، درایه‌های غیرصفر ماتریس به صورت ستون به ستون ذخیره خواهند شد. طول آرایه AA، NNz می‌باشد.
- **آرایه صحیح JA:** طول این آرایه نیز به اندازه تعداد درایه‌های غیرصفر کل ماتریس خواهد بود. مقادیر ذخیره شده در آرایه JA شامل اندیس سطر عناصر غیرصفر ستون‌های ماتریس می‌باشد.
- **آرایه صحیح IA:** عناصر این آرایه اشاره‌گر می‌باشند. به عنوان مثال IA(i) به موقعیت شروع ستون i ام در آرایه‌های AA و JA اشاره می‌نماید. یعنی مکان AA[IA(i)] موقعیت شروع عناصر غیرصفر ستون i ام ماتریس A می‌باشد. به همین ترتیب JA[IA(i)] موقعیت شروع اندیس سطری درایه‌های غیرصفر ستون i ام ماتریس A است. طول این آرایه به اندازه n+1 می‌باشد. در خانه شماره n+1 ام این آرایه تعداد عناصر غیرصفر ماتریس ذخیره خواهد شد.

برای ماتریس فوق فرمت فشرده ستونی را به فرم زیر اعمال می‌کنیم:

AA=	۱	۳	۶	۴	۷	۱۰	۲	۵	۸	۱۱	۹	۱۲

JA=	۱	۲	۳	۲	۳	۴	۱	۲	۳	۴	۳	۵

IA=	۱	۴	۵	۷	۱۱	۱۳

از موقعیت IA(i) تا IA(i+1)-1 عناصر آرایه‌های AA و JA شامل مقادیر عددی و اندیس سطری درایه‌های غیرصفر ستون i ام ماتریس می‌باشند.

در اینجا برای درک بهتر مفهوم فشرده سازی سطری، الگوریتم زیر را ارائه خواهیم کرد که ضرب ماتریس در بردار را شرح خواهد داد. در این مثال حاصلضرب $y = Ax$ محاسبه خواهد شد که A یک ماتریس تنک می‌باشد که درایه‌های غیرصفر آن به فرمت سطری فشرده ذخیره گردیده است.

الگوریتم ۱-۳ ضرب ماتریس در بردار با فرمت CSR

Input: Choose a sparse matrix in CSR format and a vector x.
Output: vector y.

```
1: for i = 0 to n do
2:     y[i] = 0
3:     for j = IA[i] to IA[i + 1] − 1 do
4:         t = JA[j]
5:         y[i]+ = x[t] ∗ AA[j]
6:     end for
7: end for
```

۱-۹ نحوه اجرای برنامه موازی در محیط برنامه‌نویسی OpenMP

OpenMP به عنوان یک واسط برنامه‌نویسی مبتنی بر دستورالعمل، برای حافظه‌های اشتراکی شناخته می‌شود. فرض کنیم از کامپایلر C و C++ استفاده می‌کنیم که رابط برنامه‌نویسی OpenMP را پشتیبانی می‌کند. برای اینکه بخشی از برنامه به صورت موازی اجرا شود، باید در یک بخش تحت عنوان pragma کدنویسی شود. به این مفهوم که تمامی دستوراتی که درون این بخش قرار دارند، به صورت موازی اجرا خواهند شد. درون این بخش اجازه داریم تا توابع کتابخانه‌ای omp را صدا بزنیم. مثال (۱-۹-۱) ساده‌ترین برنامه در محیط OpenMP می‌باشد [۷].

مثال ۱-۹-۱. مثال ساده Hello World در محیط OpenMP

```
//OpenMP header                                                    ۱
#include <omp.h>                                                   ۲
#include <stdio.h>                                                 ۳
                                                                   ۴
int main(int argc, char* argv[])                                   ۵
{                                                                  ۶
  //Beginning of parallel region                                   ۷
  #pragma omp parallel                                             ۸
  {                                                                ۹
    printf("Hello World OpenMP...\n");                             ۱۰
  }                                                                ۱۱
  //Ending of parallel region                                      ۱۲
}                                                                  ۱۳
```

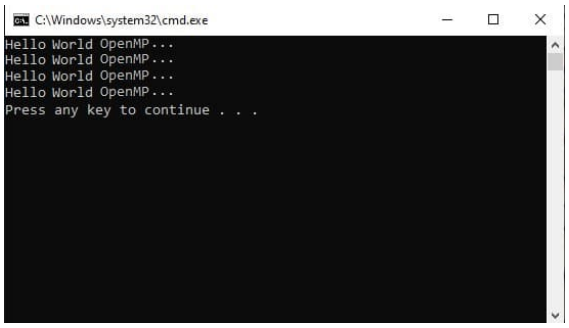
حال نحوه اجرای این برنامه در هریک از سیستم‌عامل‌های ویندوز و لینوکس را شرح می دهیم:

۱-۹-۱ اجرای برنامه موازی OpenMP در سیستم عامل ویندوز

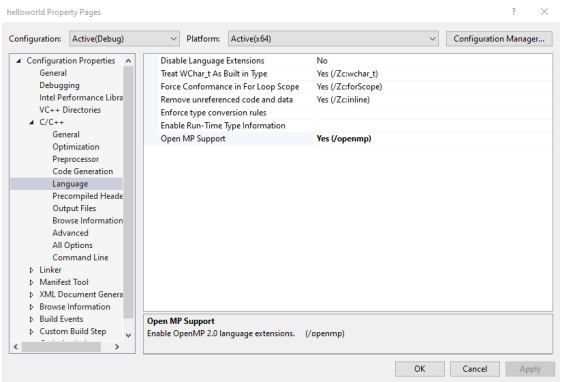
برای کامپایل[1] کردن یک برنامه OpenMP در ویندوز، از کامپایلرهای مختلفی می‌توان استفاده نمود که چند مورد از آن‌ها را نام برده و به توضیح آن‌ها می‌پردازیم.

- **کامپایلر Microsoft Visual C++(MSVC):** یک کامپایلر از مایکروسافت برای زبان‌های برنامه‌نویسی C و C++ است. MSVC یک نرم‌افزار اختصاصی است که به عنوان بخشی از مجموعه بزرگتر Microsoft Visual Studio می‌باشد. کامپایلر MSVC از استاندارد OpenMP 2.0 پشتیبانی می‌کند.

برای اجرای مثال (۱-۹-۱) در محیط ویژوال استودیو ابتدا بر روی برنامه راست کلیک کرده و گزینه properties را انتخاب می‌کنیم، پس از آن مطابق تصویر (۱-۸) وارد زبانه C/C++ > Language شده و OpenMP Support را فعال می‌کنیم. شکل (۱-۷) خروجی اجرای مثال (۱-۹-۱) را در محیط Visual Studio 2015 و در ماشینی با ۴ هسته نمایش می‌دهد.



شکل ۱-۷: اجرای مثال ۱-۹-۱ در محیط ویندوز با استفاده از کامپایلر MSVC



شکل ۱-۸: تنظیمات ویژوال استودیو برای پشتیبانی از OpenMP در ویندوز

[1] compile

- **کامپایلر GNU:** مجموعه کامپایلرهای گنو[1] (GCC) توسط ریچارد استالمن [2] در سال ۱۹۸۷ منتشر شد. در ابتدا تنها برای زبان C به کار می‌رفت، اما بعدها کامپایلر زبان‌های C++، Fortran، Java و... نیز به آن اضافه شد. GCC علاوه بر اینکه کامپایلر رسمی سیستم‌عامل GNU است، می‌تواند برنامه‌های نوشته شده درون هریک از سیستم‌عامل‌های ویندوز، لینوکس، اندروید، آی‌اواس و... را نیز کامپایل کند.

برای استفاده از این کامپایلر در سیستم‌عامل ویندوز، از ابزارهای مختلفی استفاده می‌شود که از این ابزارها می‌توان به MinGW[3] اشاره کرد. MinGW یک محیط توسعه برای برنامه‌های کاربردی مایکروسافت ویندوز می‌باشد که به صورت رایگان عرضه شده است [4]. پس از نصب MinGW به راحتی می‌توان با دستورات خط فرمانی در محیط cmd برنامه را اجرا کرد. در جدول زیر دستورات مربوط به هر زبان برای کامپایل برنامه توسط این کامپایلر آورده شده است:

جدول ۱-۱: دستورات کامپایل برنامه توسط کامپایلر GCC

Language	Command
C	gcc -fopenmp Sourse.c
C++	g++ -fopenmp Sourse.cpp

خروجی اجرای مثال (۱-۹-۱) در سیستم‌عامل ویندوز با استفاده از کامپایلر GCC و در ماشینی با ۴ هسته در تصویر (۱-۹) به نمایش درآمده است.

شکل ۱-۹: کامپایل و اجرای خط فرمانی مثال ۱-۹-۱ در ویندوز با استفاده از کامپایلر GNU

- **کامپایلر Intel C++:** گروهی از کامپایلرهای C و C++ از اینتل است که برای سیستم‌عامل‌های ویندوز، مک و لینوکس قابل استفاده می‌باشد. کامپایلرهای اینتل با Microsoft Visual C++ در ویندوز سازگار هستند و در مایکروسافت ویژوال استودیو با یکدیگر ادغام می‌شوند. در Linux و Mac، این کامپایلرها با مجموعه کامپایلرهای گنو (GCC) سازگاری دارند.

کامپایل کردن برنامه با استفاده از کامپایلر اینتل به دو صورت خط فرمانی و یا درون محیط برنامه ویژوال استودیو امکان‌پذیر است. برای کامپایل برنامه به صورت خط فرمانی با استفاده از کامپایلر اینتل از دستورات آورده شده در

[4] برای اطلاعات بیشتر می‌توانید به http://mingw-w64.org/doku.php مراجعه کنید

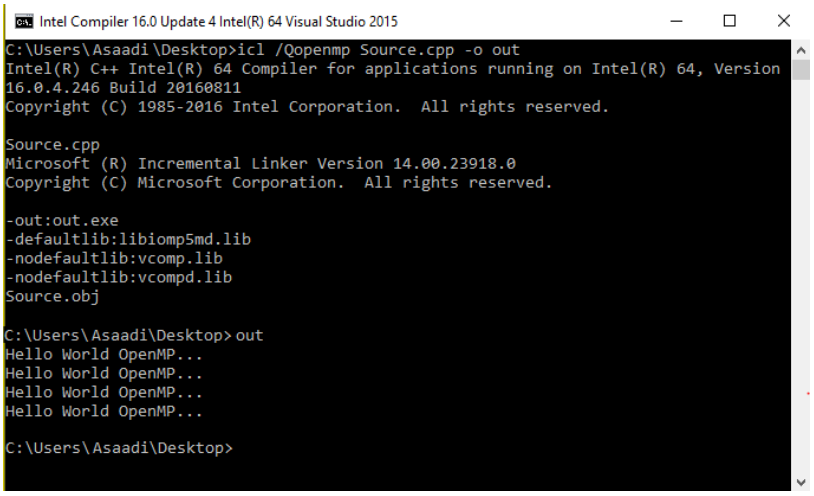
[1]GNU Compiler Collection [2]Richard Stallman [3]Minimalist GNU for Windows

جدول (۱-۲) استفاده می‌شود.

جدول ۱-۲: دستور کامپایل برنامه توسط کامپایلر Intel C++ در ویندوز

Language	Command
C/C++	icl /Qopenmp Source.c

خروجی حاصل از اجرای مثال ۱-۹-۱ با استفاده از کامپایلر اینتل در ماشینی با ۴ هسته در تصویر (۱-۱۰) دیده می‌شود.

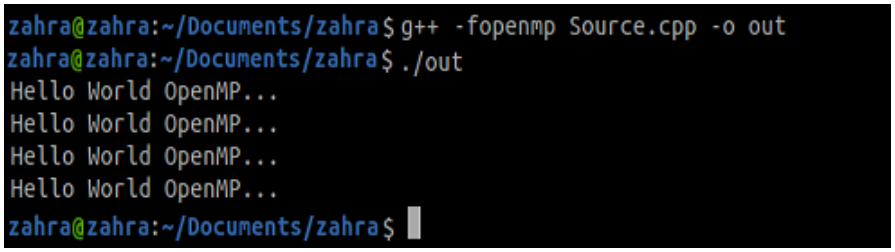


شکل ۱-۱۰: کامپایل و اجرای خط فرمانی مثال ۱-۹-۱ با استفاده از کامپایلر اینتل در ویندوز

۱-۹-۲ اجرای برنامه موازی OpenMP در سیستم عامل لینوکس

مشابه سیستم‌عامل ویندوز، در سیستم‌عامل لینوکس نیز می‌توان از کامپایلرهای مختلفی برای اجرای برنامه OpenMP استفاده نمود که در ادامه چند مورد از آن‌ها را شرح می‌دهیم.

- **کامپایلر GNU:** همانطور که در بخش قبل توضیح داده شد، این کامپایلر برای کامپایل و اجرای برنامه‌ها به زبان‌های C، C++، Fortran، Java در سیتم‌عامل‌های مختلف مورد استفاده قرار می‌گیرد. برای کامپایل برنامه OpenMP با استفاده از این کامپایلر در سیستم‌عامل لینوکس مشابه با ویندوز عمل کرده و از دستورات موجود در جدول (۱-۱) استفاده خواهد شد. در ادامه در شکل (۱-۱۱) خروجی مثال (۱-۹-۱) با استفاده از کامپایلر GNU در رایانه ۴ هسته‌ای با سیستم‌عامل لینوکس آورده شده است.
- **کامپایلر Intel C++:** با استفاده از دستورات آورده شده در جدول (۱-۳) می‌توانیم برنامه OpenMP را در سیستم‌عامل لینوکس کامپایل کنیم.



شکل ۱-۱۱: کامپایل و اجرای خط فرمانی مثال ۱-۹-۱ با استفاده از کامپایلر GNU در لینوکس

جدول ۱-۳: دستورات کامپایل برنامه توسط کامپایلر Intel در سیستم عامل لینوکس

Language	Command
C	icc -qopenmp Sourse.c -o out
C++	icpc -qopenmp Sourse.cpp -o out

۱-۱۰ برخی از دستورات موجود در محیط برنامه‌نویسی OpenMP

۱-۱۰-۱ تعیین تعداد رشته‌ها و رتبه‌بندی آنان

همان‌طور که قبلا گفته شد ناحیه موازی توسط چندین رشته اجرا می‌شود که می‌توان با استفاده از دستور omp_set_num_threads() تعداد رشته‌ها را تعیین کرد یا اینکه هنگام اجرا با دستور omp_get_num_threads() از کاربر دریافت نمود. تعداد پیش‌فرض این رشته‌ها معمولا به اندازه تعداد هسته‌های واحد پردازنده در نظر گرفته می‌شود. پس از ورود به ناحیه موازی و تشکیل مجموعه‌ای از رشته‌ها (عملیات انشعاب)، هر کدام از رشته‌ها با استفاده از دستورالعمل omp_get_thread_num() یک شناسه (Id) [1] مخصوص به خود را می‌گیرد که این شناسه می‌تواند از صفر (رشته اصلی) تا (۱-تعداد رشته‌ها) شماره‌گذاری شود.

مثال ۱-۱۰-۱. تعیین تعداد رشته‌های اجراکننده برنامه و رتبه‌بندی رشته‌ها در محیط OpenMP

```
#include <omp.h>                                                    ۱
#include <stdio.h>                                                  ۲
                                                                    ۳
int main (int argc, char *argv[])                                   ۴
{                                                                   ۵
    int nthreads, tid;                                              ۶
    #pragma omp parallel                                            ۷
    {                                                               ۸
        tid = omp_get_thread_num();                                 ۹
        nthreads = omp_get_num_threads();                           ۱۰
        printf("Hello World. I am Thread %d of %d Threads.\n", tid, ۱۱
            nthreads);
    }                                                               ۱۲
}                                                                   ۱۳
```

[1] Identifier(Id)

خروجی:

```
Hello World. I am Thread 0 of 4 Threads.
Hello World. I am Thread 1 of 4 Threads.
Hello World. I am Thread 2 of 4 Threads.
Hello World. I am Thread 3 of 4 Threads.
```

۱-۱۰-۲ استفاده از حلقه for در محیط OpenMP

در صورتی که حلقه for درون ناحیه موازی قرار گیرد به صورت موازی اجرا خواهد شد. هنگامی که رشته‌های تولید شده در اثر عملیات انشعاب به حلقه for می‌رسند، OpenMP به صورت خودکار تکرار[1]های حلقه را بین این رشته‌ها تقسیم می‌کند. هر رشته مسئولیت اجرای قسمتی از این تکرارها را به عهده می‌گیرد. نحوه تقسیم تکرارها بین رشته‌ها به شکل‌های متفاوتی انجام می‌شود که در ادامه به آن می‌پردازیم. به منظور سهولت در کدنویسی حلقه for را می‌توان با دستور ناحیه موازی به شکل زیر ترکیب نمود. در این حالت نیازی به دوباره‌نویسی pragma نخواهد بود.

```
#pragma omp parallel
{
  #pragma omp for [clause]
  // for loop to parallelize
}
```

```
#pragma omp parallel for[clause]
{
  // for loop to parallelize
}
```

بند[2]های مختلفی وجود دارد که می‌توان یک یا چند مورد از این بندها را به حلقه موازی اضافه نمود. نکته قابل ذکر این است که اگر متغیری درون یکی از این بندها تعریف شده باشد، دیگر نمی‌تواند همزمان درون بند دیگری قرار بگیرد. در ادامه به چند مورد از این بندها اشاره کرده و به توضیح آن‌ها می‌پردازیم.

- **بند مشترک (shared):** به طور کلی اگر یک متغیر در ناحیه موازی به صورت مشترک تعریف شده باشد، همه رشته‌ها به متغیر اصلی دسترسی دارند و تغییرات در همان متغیر انجام شده و متغیر به‌روز می‌شود. به منظور درک بیشتر این بند به مثال (۱-۱۰-۲) توجه کنید:

مثال ۱-۱۰-۲. استفاده از بند shared در محیط OpenMP

```
int main(void){                                                    ۱
  int x=10;                                                        ۲
  #pragma omp parallel for shared(x)                               ۳
  for(int i=0; i<4; i++){                                          ۴
    x = x+i;                                                       ۵
    printf("threadNumber:%d -> x:%d\n",omp_get_thread_num(),x);    ۶
  }                                                                ۷
  printf("x is:%d\n", x);                                          ۸
}                                                                  ۹
```

[1] iteratio [2] clause

در این مثال تعداد رشته‌ها مشخص نشده است، به همین خاطر تعداد رشته‌ها توسط OpenMP به صورت پیش فرض به اندازه‌ی هسته‌های واحد پردازش در نظر گرفته می‌شود. از آنجایی که این زیربرنامه در یک ماشین با پردازنده ۴ هسته‌ای اجرا شده، با ۴ رشته نیز شروع به کار خواهد کرد که از ۰ تا ۳ شماره گذاری می‌شوند. در خط شماره ۳ این مثال متغیر x با استفاده از بند shared به صورت مشترک تعریف شده است یعنی در دسترس تمامی رشته‌های موجود در حلقه موازی می‌باشد. هر رشته عملیات مربوط به خود را بر روی متغیر x اعمال کرده و این متغیر را به روز می‌کند. پس از هر بار اجرا، نتیجه اجرا متفاوت خواهد بود و هر رشته مقدار متفاوتی را نمایش خواهد داد.

threadNumber:2 → x:12
threadNumber:0 → x:10
threadNumber:1 → x:13
threadNumber:3 → x:16
x is:16

- **بند خصوصی (private):** هر رشته یک آدرس حافظه منحصربه‌فرد را دریافت می‌کند و یک نسخه کپی از متغیر اصلی را در آن ذخیره می‌کند. هنگامی که ناحیه موازی به پایان می‌رسد، حافظه آزاد شده و این متغیر دیگر وجود نخواهد داشت. در ادامه توضیحات بیشتر در مثال (۱-۱۰-۳) آورده شده است.

مثال ۱-۱۰-۳. استفاده از بند private در محیط OpenMP

```
int main(void){                                                    ۱
  int x=10;                                                        ۲
  #pragma omp parallel for private(x)                              ۳
  for(int i=0; i<5; i++){                                          ۴
    x = x+i;                                                       ۵
    printf("threadNumber:%d -> x:%d\n",omp_get_thread_num(),x);    ۶
  }                                                                ۷
  printf("x is:%d\n", x);                                          ۸
}                                                                  ۹
```

در خط شماره ۳ این مثال، متغیر x با استفاده از بند private به صورت خصوصی تعریف شده است. تمامی رشته‌های موجود در حلقه for پس از رسیدن به این دستور، یک نسخه کپی مخصوص به خود را از متغیر x دریافت می‌کنند. مقدار این متغیر در ابتدا به صورت پیش‌فرض صفر در نظر گرفته می‌شود. هر رشته تغییرات را بر روی نسخه x مخصوص به خود اعمال می‌کند و رشته‌ها هیچ گونه دخالتی بر کار یکدیگر ایجاد نمی‌کنند. اگر این زیر برنامه را در یک رایانه با ۴ هسته اجرا کنیم، خروجی زیر را خواهیم داشت و می‌بینیم که x همان مقدار قبل از ورود به حلقه موازی را دارد.

Thread number:1 → x:2
Thread number:3 → x:4
Thread number:2 → x:3
Thread number:0 → x:0
Thread number:0 → x:1
x is 10

- **بند پیش‌فرض(default):** این بند به کاربر این امکان را می‌دهد تا تکلیف تمام متغیرهایی که در حلقه for موازی استفاده می‌شود را مشخص کند. در زبان C و C++ دو شکل متفاوت از این بند استفاده می‌شود. نکته‌ای که باید به آن توجه داشته باشیم، این است که در هر حلقه موازی فقط یک شکل از بند پیش‌فرض را می‌توان به کار برد.

۱. **default(shared):** همه متغیرهای موجود در بلوک موازی را به صورت اشتراکی در نظر می‌گیرد، اگر بخواهیم بعضی از متغیرها را به‌صورت خصوصی تعریف کنیم باید آنها را در بند private اضافه کنیم. به مثال (۱-۱۰-۴) توجه کنید که در آن دو متغیر a، n و آرایه c به طول n داریم:

مثال ۱-۱۰-۴. استفاده از بند default(shared) در محیط OpenMP

```
int a=10, n=5;                                                    ۱
int c[n];                                                         ۲
#pragma omp parallel for default(shared) private(c)               ۳
for (int i=0; i<n; i++){                                          ۴
   c[i]= a+i;                                                     ۵
    printf("threadNumber:%d -> c[%d]=%d\n",omp_get_thread_num     ۶
       (),i,c[i]);
}                                                                 ۷
```

در خط سوم این زیربرنامه از بند default(shared) استفاده شده است که باعث می‌شود تمام متغیرهای استفاده شده در حلقه موازی به صورت مشترک درنظر گرفته می‌شود. اما از آنجایی که در این مثال می‌خواهیم هر رشته به صورت خصوصی به عناصر آرایه c دسترسی داشته باشد، پس باید با استفاده از بند private این آرایه را به صورت خصوصی تعریف می‌کنیم.

خروجی این زیربرنامه با ۴ رشته به صورت زیر می‌باشد:

threadNumber:2 → c[3]=13
threadNumber:3 → c[4]=14
threadNumber:1 → c[2]=12
threadNumber:0 → c[0]=10
threadNumber:0 → c[1]=11

۲. **default(none):** استفاده از آن مستلزم این است که برنامه‌نویس صراحتا دامنه اشتراک تمام متغیرها را تعیین کند. مثال (۱-۱۰-۵) را در نظر بگیرید:

مثال ۱-۱۰-۵. استفاده از بند default(none) در محیط OpenMP

```
int a=10, n=10;                                                   ۱
int vector[n];                                                    ۲
#pragma omp parallel for default(none) shared(n,vector)           ۳
for (int i=0; i<n; i++){                                          ۴
  vector[i] = i*a;                                                ۵
}                                                                 ۶
```

همانطور که مشاهده می‌کنید در خط سوم این زیر برنامه از بند default(none) استفاده شده است. به این معنا که باید تکلیف تمام متغیرهای موجود در حلقه مشخص شود. در این حلقه متغیرهای a، n و آرایه vector استفاده شده است که متغیر n و آرایه vector به صورت مشترک مشخص شده‌اند، اما در مورد متغیر a هیچ‌گونه تعیین تکلیفی نشده است. که باعث می‌شود پس از اجرا با خطای زیر مواجه شویم.

Error: 'a' not specified in enclosing parallel
vector[i] = i*a;

ممکن است این سوال برای متغیر i پیش بیاید که چرا نوع آن مشخص نشده است. برای پاسخ به آن باید بگوییم که OpenMP به صورت خودکار تعداد تکرارهای حلقه را بین رشته‌های موجود در حلقه تقسیم می‌کند. این تقسیم‌بندی به شکل‌های متفاوتی انجام می‌شود که در بند schedule به توضیح آن می‌پردازیم.

- **بند firstprivate:** همانند بند private یک نسخه کپی از متغیر اصلی را برای خود دارد، با این تفاوت که در بند private یک متغیر با همان نام و همان نوع ساخته می‌شود، اما در firstprivate برای نسخه کپی‌شده مقداردهی اولیه نیز انجام می‌شود.

مثال ۱-۱۰-۶. استفاده از بند firstprivate در محیط OpenMP

```
int main(void) {                                                    ۱
  int x=10;                                                         ۲
  #pragma omp parallel for firstprivate(x)                          ۳
  for(int i=0; i<5; i++) {                                          ۴
    x = x+i;                                                        ۵
    printf ("threadNumber:%d -> x:%d\n",omp_get_thread_num(),x);    ۶
  }                                                                 ۷
  printf("x is:%d\n", x);                                           ۸
}                                                                   ۹
```

مثال (۱-۱۰-۶) را در نظر بگیرید که در خط سوم آن متغیر x به صورت firstprivate تعریف شده است. زمانی که رشته‌های اجراکننده حلقه موازی به بند firstprivate می‌رسند، یک نسخه خصوصی از متغیر x را دریافت می‌کنند. این رشته‌ها تغییرات را بر روی متغیر خصوصی خود اعمال کرده و هیچ گونه دخالتی بر متغیر خصوصی رشته‌های دیگر ندارند. نکته قابل توجه در مورد بند firstprivate این است که در این بند برای متغیر خصوصی هر رشته، مقداردهی اولیه نیز انجام می‌شود. یعنی هر رشته یک متغیر خصوصی x با مقدار ۱۰ دریافت می‌کند.

threadNumber:1 → x:12
threadNumber:0 → x:10
threadNumber:0 → x:11
threadNumber:3 → x:14
threadNumber:2 → x:13
x is:10

- **بند lastprivate:** این بند نیز همانند بند خصوصی یک نسخه کپی از متغیر اصلی را برای خود دارد، اما پس از پایان ناحیه موازی مقدار آخرین تکرار حلقه بر روی متغیر اصلی قرار می‌گیرد.

مثال ۱-۱۰-۷. استفاده از بند lastprivate در محیط OpenMP

```
int main(void) {                                                    ۱
  int x=10;                                                         ۲
  #pragma omp parallel for lastprivate(x)                           ۳
  for(int i=0; i<5; i++) {                                          ۴
    x = x+i;                                                        ۵
    printf("threadNumber:%d -> x:%d\n",omp_get_thread_num(),x);     ۶
  }                                                                 ۷
  printf("x is:%d\n", x);                                           ۸
}                                                                   ۹
```

مثال (۱-۱۰-۷) را در نظر بگیرید که در آن هنگامی که رشته‌های اجرا کننده حلقه موازی به خط سوم این زیر برنامه و بند lastprivate می‌رسند، همانند بند private یک نسخه کپی مخصوص به خود را از متغیر x دریافت می‌کنند. در ابتدا مقدار این متغیر به صورت پیش‌فرض صفر در نظر گرفته می‌شود. هر کدام از این رشته‌ها عملیات مربوط به خود را بر روی متغیر خصوصی خود اعمال می‌کنند. بدون اینکه دخالتی بر متغیر خصوصی رشته‌های دیگر داشته باشند. پس از اجرای این زیربرنامه با ۴ رشته، مشاهده می‌کنیم که مقدار متغیر پس از پایان حلقه موازی، ۴ خواهد بود. به این معنی که هنگام استفاده از بند lastprivate پس از اینکه اجرای حلقه موازی پایان یافت، مقدار آخرین تکرار حلقه بر روی متغیر x قرار خواهد گرفت.

threadNumber: 3 → x:4
threadNumber: 1 → x:2
threadNumber: 0 → x:0
threadNumber: 0 → x:1
threadNumber: 2 → x:3
x is: 4

- **بند reduction:** این بند شامل دو پارامتر می‌باشد. پارامتر اول این بند، شامل یک عملگر منطقی یا ریاضی و پارامتر دوم آن یک متغیر است. و نحوه عملکرد آن به این صورت است که یک کپی از متغیر موجود برای هر رشته ساخته می‌شود و در پایان بلوک موازی مقادیر نهایی همه نسخه‌های خصوصی، با عملگر مشخص شده ترکیب می‌شوند و نتیجه در اختیار تمام رشته‌ها قرار می‌گیرد. در مثال (۱-۱۰-۸) جزئیات این روش آورده شده است.

مثال ۱-۱۰-۸. استفاده از بند reduction در محیط OpenMP

```
int sum=0;                                                  ۱
int vector[10];                                             ۲
#pragma omp parallel for shared(vector) reduction(+:sum)    ۳
for(int i=0; i<10; i++){                                    ۴
  sum += vector[i];                                         ۵
}                                                           ۶
```

همانطور که مشاهده می‌کنید در خط سوم این برنامه بند reduction استفاده شده است و پس از اینکه رشته‌های موجود در حلقه به این بند می‌رسند، یک نسخه محلی [1] از متغیر sum را دریافت میکنند. مقدار اولیه این متغیر در خط شماره ۱، صفر در نظر گرفته شده است. هنگامی که رشته‌های موجود به خط چهارم این زیربرنامه می‌رسند، ۱۰ تکرار حلقه بین آن‌ها تقسیم می‌شود. به این صورت که رشته شماره ۰ مسئول اجرای تکرارهای ۰،۱و۲ از این حلقه موازی می‌باشد. همچنین رشته شماره ۱ تکرارهای ۳،۴و۵، رشته شماره ۲ تکرارهای ۶ و ۷ و رشته شماره ۳ تکرارهای ۸ و ۹ را از این حلقه موازی دریافت خواهند کرد. پس از دریافت تکرارهای حلقه، هر رشته عملیات درون حلقه را برای متغیر محلی خود به شکل زیر انجام می‌دهد.

thread 0:
$$sum^{(0)} = vector[0] + vector[1] + vector[2]$$
thread 1:
$$sum^{(1)} = vector[3] + vector[4] + vector[5]$$
thread 2:
$$sum^{(2)} = vector[6] + vector[7]$$
thread 3:
$$sum^{(3)} = vector[8] + vector[9]$$

در نهایت پس از اتمام اجرای حلقه موازی، تمامی متغیرهای محلی sum توسط پارامتر اول بند reduction که در اینجا عملیات جمع می‌باشد، ترکیب می‌شود.

$$sum = sum^{(0)} + sum^{(1)} + sum^{(2)} + sum^{(3)}$$

- **بند schedule:** همان‌گونه که قبلا اشاره شد، هنگامی رشته‌های موجود در ناحیه موازی به حلقه for می‌رسند تکرارهای حلقه بین این رشته‌ها تقسیم می‌شود. نحوه تقسیم‌بندی این تکرارها بین رشته‌های موجود توسط بند schedule تعیین می‌شود که حاوی دو پارامتر می‌باشد. پارامتر اول نوع تقسیم‌بندی تکرارهای حلقه میان رشته‌ها را مشخص می‌کند و پارامتر دوم این بند شامل اندازه سهم هر رشته از تکرارهای حلقه (chunk-size) می‌باشد. در ادامه به پارامترهای static،dynamic،guided و auto اشاره کرده و به توضیح آن‌ها می‌پردازیم.

  - **پارامتر static:** در این روش از تقسیم‌بندی، تکرارهای حلقه به اندازه chunk-size تعیین شده و به صورت چرخشی بین رشته‌های اجراکننده حلقه توزیع می‌شود. اگر هنگام استفاده از این پارامتر chunk-size را تعیین نکرده باشیم، OpenMP به صورت پیش‌فرض آن را به اندازه تعداد تکرارهای حلقه تقسیم بر تعداد رشته‌ها در نظر می‌گیرد. استفاده از پارامتر static برای زمانی که همه تکرارهای حلقه هزینه محاسباتی یکسانی داشته باشند، مناسب‌تر است. هنگام استفاده از پارامتر static نتیجه تقسیم‌بندی پس از هر بار اجرا ثابت بوده و تغییری نخواهد داشت.

[1] local

مثال ۱-۱۰-۹. استفاده از بند schedule همراه با پارامتر static و chunk-size ۲

```
int main(void){                                                    ۱
  int i, n=15;                                                     ۲
  #pragma omp parallel for schedule(static,2)                      ۳
  for(i=0; i<=n; i++){                                             ۴
    printf("Thread number:%d received the iteration(%d)\n",        ۵
        omp_get_thread_num(),i);
  }                                                                ۶
}                                                                  ۷
```

همانطور که در مثال (۱-۱۰-۹) می‌بینید، در خط سوم این مثال از بند schedule همراه با پارامترهای static و ۲ استفاده شده است. به این معنی که هر کدام از رشته‌های اجرا کننده حلقه، تعداد ۲ تکرار از تکرارهای حلقه را دریافت می‌کند. پس از اینکه تمامی رشته‌های موجود در حلقه تعداد ۲ تکرار خود را دریافت کردند در صورتی که هنوز تکرارهای حلقه باقی مانده باشد، رشته‌ها مجددا تکرارهای حلقه را به اندازه chunk-size دریافت می‌کنند و این روند تا زمانی که تکرارهای حلقه تمام شوند ادامه خواهد یافت. خروجی زیربرنامه فوق بر روی ۴ رشته، در جدول زیر نمایش داده شده است.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Thread 0	*	*							*	*						
Thread 1			*	*							*	*				
Thread 2					*	*							*	*		
Thread 3							*	*							*	*

- **پارامتر dynamic**: در هنگام استفاده از این پارامتر هیچ ترتیبی وجود ندارد. هر رشته تکه‌ای از تکرارها را اجرا می‌کند و سپس تکه دیگری را درخواست می‌کند تا زمانی که دیگر تکه‌ای در دسترس نباشد. اگر برای این پارامتر، مقدار chunk-size را تعیین نکنیم به صورت پیش‌فرض برابر با یک در نظر گرفته می‌شود. استفاده از پارامتر dynamic برای زمانی که تکرارهای حلقه هزینه‌های محاسباتی متفاوتی دارند، مناسب‌تر است. از آنجایی که در هنگام استفاده از این پارامتر تمامی تکرارها به صورت پویا بین رشته‌های موجود توزیع می‌گردد، پس از هر بار اجرا نتیجه متفاوتی را شاهد خواهیم بود.

مثال ۱-۱۰-۱۰. استفاده از بند schedule همراه با پارامتر dynamic و chunk-size ۲

```
int main(void){                                                    ۱
  int i, n=15;                                                     ۲
  #pragma omp parallel for schedule(dynamic,2)                     ۳
  for(i=0; i<=n; i++){                                             ۴
    printf("Thread number:%d received the iteration(%d)\n",        ۵
        omp_get_thread_num(),i);
  }                                                                ۶
}                                                                  ۷
```

مثال (۱-۱۰-۱۰) را در نظر بگیرید که در آن از بند schedule همراه با پارامتر dynamic و chunk-size، ۲ استفاده شده است. هر رشته که به حلقه for می‌رسد تعداد ۲ تکرار از تکرارهای حلقه را دریافت می‌کند و پس از انجام عملیات درون حلقه، ۲ تکرار دیگر را درخواست می‌کند. این فرایند برای هر رشته تکرار می‌شود تا زمانی که دیگر تکراری از حلقه باقی نمانده باشد.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Thread 0					*	*									*	*
Thread 1							*	*			*	*				
Thread 2	*	*							*	*			*	*		
Thread 3			*	*												

- **پارامتر guided**: در این پارامتر نیز همانند بند private هیچگونه ترتیبی در اجرا وجود ندارد. هر بار سهم هر رشته از تکرارهای حلقه، به اندازه تعداد تکرارهای اختصاص داده نشده تقسیم بر تعداد رشته‌ها در نظر گرفته می‌شود. همچنین هیچ رشته‌ای نمی‌تواند گام‌های حلقه را کمتر از مقدار chunk-size دریافت کند. مگر اینکه هیچ کاری برای اجرا باقی نمانده باشد. اگر اندازه chunk-size را تعیین نکنیم به صورت پیش‌فرض برابر با یک در نظر گرفته می‌شود.

مثال ۱-۱۰-۱۱. استفاده از بند guided همراه با پارامتر static و chunk-size ۲

```
int main(void){                                                    ۱
  int i, n=15;                                                     ۲
  #pragma omp parallel for schedule(guided,2)                      ۳
  for(i=0; i<=n; i++){                                             ۴
    printf("Thread number:%d received the iteration(%d)\n",        ۵
        omp_get_thread_num(),i);
  }                                                                ۶
}                                                                  ۷
```

در خط شماره ۳ مثال (۱-۱۰-۱۱) از بند schedule همراه با دو پارامتر guided و ۲ استفاده شده است. در مثال فوق حلقه for دارای ۱۶ گام است و با ۴ رشته اجرا می‌شود. رشته شماره ۱ وارد حلقه موازی می‌شود. از آنجایی که هنوز هیچ تکراری از حلقه به رشته‌های موجود اختصاص داده نشده پس رشته ۱ تعداد $\frac{۱۶}{۴} = ۴$ گام از گام‌های حلقه را دریافت می‌کند. حال تعداد گام‌های اختصاص داده نشده به $۱۶ - ۴ = ۱۲$ تغییر می‌کند. پس از آن رشته شماره صفر $\frac{۱۲}{۴} = ۳$ گام بعدی حلقه موازی را دریافت خواهد کرد. رشته شماره ۳ به اجرای $\frac{۹}{۴} = ۲$ گام بعدی حلقه یعنی گام‌های ۷ و ۸ می‌پردازد. در ادامه سهم رشته ۲ از گام‌های حلقه، $\frac{۷}{۴} = ۱$ می‌باشد اما این تعداد از پارامتر دوم بند schedule کمتر است لذا مقدار آن برای این رشته و رشته‌هایی که پس از آن مسئولیت اجرای گام‌های حلقه را دارند باید به اندازه chunk-size باشد. زمانی که به اجرای آخرین گام حلقه می‌رسیم رشته شماره ۱ گام پایانی حلقه را دریافت کرده و به اجرای آن می‌پردازد.

اجرای برنامه به صورت جدول زیر می‌باشد. اگر یک بار دیگر این زیربرنامه اجرا شود، طرح و نقشه کلی تقسیم‌بندی همین است اما ترتیب آن تغییر خواهد کرد.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Thread 0					*	*	*					*	*			
Thread 1	*	*	*	*												*
Thread 2										*	*					
Thread 3								*	*					*	*	

- **پارامتر auto:** طرح کلی آن در قالب مثال (۱-۱۰-۱۲) مورد استفاده قرار می‌گیرد و به کامپایلر واگذار می‌کند که کدامیک از پارامترهای static، dynamic و یا guided را انتخاب کند. این پارامتر به تنهایی و بدون تعیین chunk-size استفاده می‌شود. همچنین ممکن است در حلقه‌های مختلف، انتخاب متفاوت باشد.

مثال ۱-۱۰-۱۲. استفاده از بند schedule همراه با پارامتر auto

```
int main(void){                                                    ۱
  int i, n=15;                                                     ۲
  #pragma omp parallel for schedule(auto)                          ۳
  for(i=0; i<=n; i++){                                             ۴
    printf("Thread number:%d received the iteration(%d)\n",        ۵
        omp_get_thread_num(),i);
  }                                                                ۶
}                                                                  ۷
```

- **زمانی که هیچ بند schedule برای حلقه for استفاده نشود:** این حالت معادل با استفاده از بند schedule با پارامتر static می‌باشد، اما chunk-size آن به صورت پیش فرض یعنی تعداد گام‌های حلقه تقسیم بر تعداد رشته‌ها است. به مثال (۱-۱۰-۱۳) توجه کنید. خطوط ۴ تا ۶ این مثال شامل حلقه for می‌باشد که ۱۶ گام دارد و با ۴ رشته اجرا خواهد شد.

مثال ۱-۱۰-۱۳. اجرای حلقه موازی بدون استفاده از هیچ بند schedule

```
int main(void){                                                    ۱
  int i, n=15;                                                     ۲
  #pragma omp parallel for                                         ۳
  for(i=0; i<=n; i++){                                             ۴
    printf("Thread number:%d received the iteration(%d)\n",        ۵
        omp_get_thread_num(),i);
  }                                                                ۶
}                                                                  ۷
```

خروجی آن در جدول زیر آورده شده است. اگر بار دیگر نیز این مثال اجرا شود خروجی آن به همین شکل خواهد بود.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Thread 0	*	*	*	*												
Thread 1					*	*	*	*								
Thread 2									*	*	*	*				
Thread 3													*	*	*	*

- **بند ordered:** این بند مشخص می‌کند که تکرارهای حلقه موازی با همان ترتیبی که در اجرای سریال داشتند، اجرا شوند. اگر تکرارهای قبلی هنوز کامل نشده باشند، رشته‌ها قبل از اجرای تکه‌های مربوط به خود باید منتظر بمانند.

مثال ۱-۱۰-۱۴. اجرای حلقه موازی هنگام استفاده از بند ordered

```
#pragma omp parallel                1
{                                   2
  #pragma omp for                   3
  for(int i=0; i<4; i++){           4
    b(i);                           5
    c(i);                           6
  }                                 7
}                                   8
```

خروجی:

Thread 0:	b(0)	c(0)
Thread 1:	b(1)	c(1)
Thread 2:	b(2)	c(2)
Thread 3:	b(3)	c(3)

```
#pragma omp parallel                1
{                                   2
  #pragma omp for ordered           3
  for(int i=0; i<4; i++){           4
    b(i);                           5
    #pragma omp ordered             6
    c(i);                           7
  }                                 8
}                                   9
```

خروجی:

Thread 0:	b(0)	c(0)
Thread 1:	b(1)	c(1)
Thread 2:	b(2)	c(2)
Thread 3:	b(3)	c(3)

به مثال (۱-۱۰-۱۴) توجه کنید. در خطوط ۳ تا ۷ کادر بالا، حلقه for موازی چهار گام دارد که با ۴ رشته اجرا شده است. از آنجایی که حلقه بدون هیچ بند schedule می‌باشد، هر رشته پس از ورود به حلقه تعداد ۱ گام از گام‌های حلقه را اجرا می‌کند. اگر به اجرای زیربرنامه در سمت راست دقت کنید، مشاهده می‌کنید که اجرای عملیات درون حلقه برای هر رشته به صورت همزمان اتفاق می‌افتد. حال به کادر پایین دقت کنید که در خط سوم آن از بند ordered استفاده شده است. مشابه زیربرنامه بالا هر رشته مسئولیت اجرای ۱ گام از گام‌های حلقه را خواهد داشت. با توجه به خروجی اجرای زیربرنامه در سمت راست، عملیات ()b توسط تمام رشته‌ها به صورت همزمان اتفاق می‌افتد. اما در خط ۶ این زیر برنامه از دستور #pragma omp ordered استفاده شده است. به این مفهوم که ترتیب اجرای عملیات ()c مشابه با اجرای سری خواهد بود یعنی رشته‌ها قبل از اجرای گام‌های مربوط به خود باید منتظر بمانند تا گام‌های قبلی از حلقه اجرا شوند .

- **بند collapse:** اگر در یک برنامه OpenMP حلقه‌های تو در تو داشته باشیم، ساختار OpenMP توان اجرای

موازی حلقه‌های تو در تو را نخواهد داشت و فقط حلقه بیرونی به صورت موازی اجرا شده و حلقه داخلی توسط هر رشته به صورت سری اجرا خواهد شد. اگر ساختار برنامه این امکان را به ما بدهد می‌توانیم با استفاده از دستور collapse() حلقه‌های تو در تو را با یکدیگر ترکیب نموده و یک حلقه بزرگتر بسازیم. پارامتری که در این بند قرار می‌گیرد نشان‌دهنده تعداد حلقه‌های تو در تویی است که باید با یکدیگر ترکیب شوند. این حلقه‌ها از بیرونی ترین حلقه شمارش می‌شوند. توجه داشته باشید که دستور collapse(1) معادل با این است که هیچ دستوری نوشته نشده باشد.

```
#pragma omp parallel for
    private(j)
for (int i=0; i<n; i++){
  for (int j=0; j<m; j++)
  ...
}
```

```
#pragma omp parallel for
    private(j) collapse(2)
for (int i=0; i<n; i++){
  for (int j=0; j<m; j++)
  ...
}
```

- **بند nowait:** در انتهای یک ناحیه موازی، سد هماهنگ‌سازی وجود دارد که باعث می‌شود هر رشته موجود در ناحیه موازی منتظر بماند تا همه رشته‌ها اجرای خود را به پایان برسانند. اما با استفاده از دستور nowait می‌توانیم این سد هماهنگ‌ساز را غیرفعال کرده و از انتظار بیهوده جلوگیری کنیم. در مثال زیر تا زمانی که تمام رشته‌ها تکرارهای حلقه را انجام ندهند، d() اجرا نخواهد شد. اما با دستور nowait می‌توانیم سد هماهنگ‌ساز را غیرفعال کرده و از انتظار بیهوده جلوگیری کنیم.

مثال ۱-۱۰-۱۵. اجرای حلقه موازی هنگام استفاده از بند nowait

```
a();                                  1
#pragma omp parallel                  2
{                                     3
  b();                                4
  #pragma omp for                     5
  for(int i=0; i<=5; i++){            6
    c(i);                             7
  }                                   8
  d();                                9
}                                     10
```

خروجی:

Thread 0:	a	b	c(0)	c(1)	d
Thread 1:		b	c(2)	c(3)	d
Thread 2:		b	c(4)		d
Thread 3:		b	c(5)		d

```
a();                                  1
#pragma omp parallel                  2
{                                     3
  b();                                4
  #pragma omp for nowait              5
  for(int i=0; i<=5; i++){            6
    c(i);                             7
  }                                   8
  d();                                9
}                                     10
```

خروجی:

Thread 0:	a	b	c(0)	c(1)	d
Thread 1:		b	c(2)	c(3)	d
Thread 2:		b	c(4)	d	
Thread 3:		b	c(5)	d	

مثال (۱-۱۰-۱۵) را در نظر بگیرید. در مثال کادر بالایی قبل از شروع بخش موازی، عملیات a() توسط رشته شماره صفر اجرا می‌شود. پس از آنکه رشته صفر به خط شماره ۲ می‌رسد، بخش موازی شروع شده و عملیات انشعاب انجام می‌شود. با فرض اینکه زیربرنامه توسط ماشین ۴ هسته‌ای انجام شده باشد، ۴ رشته تشکیل می‌شود که عملیات b() توسط هر چهار رشته اجرا خواهد شد. سپس رشته‌ها به حلقه موازی با ۶ گام می‌رسند. این حلقه بدون هیچ بند schedule نوشته شده است، پس مشابه بند schedule با پارامتر static و بدون chunk-size با آن رفتار می‌شود. گام‌های ۰ و ۱ حلقه توسط رشته شماره صفر و گام‌های ۲ و ۳ حلقه توسط رشته شماره ۱ اجرا خواهند شد. همچنین رشته‌های ۲ و ۳ هر کدام به ترتیب گام‌های ۴ و ۵ از حلقه را اجرا خواهند نمود. سد هماهنگ‌سازی که در پایان حلقه موازی وجود دارد باعث می‌شود عملیات d() تا زمانی که تمام رشته‌ها گام‌های حلقه را انجام ندهند، اجرا نشود. اما در کادر پایین استفاده از بند nowait باعث می‌شود تا سد هماهنگ‌ساز غیرفعال شده و از انتظار بیهوده رشته‌ها جلوگیری شود.

۱-۱۰-۳ ساختارهای هماهنگ‌سازی بین رشته‌ها

لزوم وجود هماهنگ‌سازی را با کمک یک مثال بیان می‌کنیم. مثال ساده زیر را در نظر بگیرید و فرض کنید که با کمک دو رشته این زیربرنامه را اجرا نماییم. اجرای آن به صورت زیر خواهد بود:

مثال ۱-۱۰-۱۶.

```
int  x=0;                                                          ۱
#pragma omp parallel shared(x)                                     ۲
{                                                                  ۳
  x = x+1;                                                         ۴
  printf("threadNumber:%d -> x:%d\n",omp_get_thread_num(),x);      ۵
}                                                                  ۶
```

رشته شماره ۰ متغیر x را برابر با صفر مقداردهی می‌کند. سپس وارد بخش موازی شده و مقدار ۱ را به x اضافه کرده و آن را چاپ می‌کند. از طرفی رشته شماره ۱ نیز دقیقا همین اقدامات را همزمان انجام می‌دهد. بنابراین هم رشته ۰ و هم رشته ۱ همزمان مقدار x را از صفر به ۱ تغییر می‌دهند. برای جلوگیری از چنین وضعیتی بین رشته‌ها، از دستور هماهنگ‌سازی استفاده خواهیم کرد تا اطمینان حاصل کنیم که هر رشته فقط یک کار را انجام دهد و تداخل کاری با یکدیگر نداشته باشند. OpenMP چندین ساختار هماهنگ‌سازی را ارائه کرده است که به بررسی چند مورد آن‌ها می‌پردازیم.

- **هماهنگ‌ساز critical:** این دستور منطقه‌ای از کدها را مشخص می‌کند که باید تنها توسط یک رشته در یک زمان اجرا شود. اگر یک رشته درون این منطقه اجرا شود و همزمان یک رشته دیگر به آن منطقه برسد و بخواهد آن را اجرا کند، تا زمانی که اولین رشته از آن منطقه خارج نشده باشد کار اجرا مسدود می‌شود.

مثال ۱-۱۰-۱۷. استفاده از هماهنگ‌ساز critical در محیط OpenMP

```
int  x=0;                                                          ۱
#pragma omp parallel shared(x)                                     ۲
{                                                                  ۳
  #pragma omp critical                                             ۴
  x = x+1;                                                         ۵
  printf("threadNumber:%d -> x:%d\n",omp_get_thread_num(),x);      ۶
}                                                                  ۷
```

به مثال (۱-۱۰-۱۷) توجه کنید. این مثال همان مثال (۱-۱۰-۱۶) می‌باشد با این تفاوت که در خط شماره ۴ آن از هماهنگ‌ساز critical استفاده شده است. با فرض اینکه این مثال نظیر مثال قبل با ۲ رشته اجرا شده باشد، اجرای این زیربرنامه به صورت زیر خواهد بود. این اجرا نشان می‌دهد در لحظه‌ای که رشته شماره صفر در حال اجرای خط شماره ۵ است، رشته شماره ۱ اجازه این کار را ندارد. پس از اینکه اجرای رشته شماره صفر تمام شود، رشته شماره ۱ به اجرای این ناحیه می‌پردازد و مقدار متغیر x را که ۱ می‌باشد، به ۲ آپدیت می‌کند.

threadNumber: 0 → x:1
threadNumber: 1 → x:2

- **هماهنگ‌ساز atomic:** این هماهنگ‌ساز، یک نسخه کوچک از هماهنگ‌ساز critical است و از امکان خواندن و نوشتن چندین رشته به طور همزمان جلوگیری می‌کند. از نظر کلیات شبیه به هماهنگ‌ساز critical می‌باشد اما بندها و جزئیات بیشتری نسبت به آن دارد. از بندهایی که به همراه این هماهنگ‌ساز به کار رفته می‌شود می‌توان به بند read، write، update و capture اشاره کرد.

  - **بند read:** به مثال (۱-۱۰-۱۸) توجه کنید. در خط شماره ۴ این مثال هماهنگ‌ساز atomic همراه با بند read استفاده شده است. بنابراین در خط شماره ۵ در هر لحظه فقط یک رشته اجازه خواندن متغیر x را از حافظه خواهد داشت و زمانی که آن رشته متغیر را از حافظه می‌خواند، بقیه رشته‌ها اجازه خواندن این متغیر را نخواهند داشت.

مثال ۱-۱۰-۱۸. استفاده از هماهنگ‌ساز atomic همراه با بند read در محیط OpenMP

```
int x=1, var;                                                      ۱
#pragma omp parallel                                               ۲
{                                                                  ۳
  #pragma omp atomic read                                          ۴
  var=x;                                                           ۵
}                                                                  ۶
```

  - **بند write:** مثال (۱-۱۰-۱۹) را در نظر بگیرید. در خط شماره ۴ این مثال هماهنگ‌ساز atomic همراه با بند write مورد استفاده قرار گرفته است. بنابراین در خط شماره ۵ در هر لحظه فقط یک رشته اجازه نوشتن متغیر expr درون متغیر x را خواهد داشت و بقیه رشته‌ها باید منتظر بمانند.

مثال ۱-۱۰-۱۹. استفاده از هماهنگ‌ساز atomic همراه با بند write در محیط OpenMP

```
int x=1, var;                                                    ۱
#pragma omp parallel                                             ۲
{                                                                ۳
  #pragma omp atomic write                                       ۴
  x=expr;                                                        ۵
}                                                                ۶
```

- **بند update:** نوشتن این بند در هماهنگ‌ساز atomic الزامی نیست. بنابراین زمانی که هماهنگ‌ساز atomic بدون هیچ بندی نوشته شود، به صورت پیش‌فرض بند update را مورد استفاده قرار می‌دهد. به مثال (۱-۱۰-۲۰) توجه کنید. در خط شماره ۴، این هماهنگ‌ساز به تنهایی استفاده شده است. به این معنی که در هر لحظه فقط یک رشته می‌تواند اجازه به‌هنگام سازی متغیر x را با یکی از اعمال داده شده داشته باشد.

مثال ۱-۱۰-۲۰. استفاده از هماهنگ‌ساز atomic همراه با بند update در محیط OpenMP

```
int x=1, var;                                                    ۱
#pragma omp parallel                                             ۲
{                                                                ۳
  #pragma omp atomic                                             ۴
  //one of the forms:                                            ۵
  x++; or x--; or ++x; or --x; or x binop expr;                  ۶
}                                                                ۷
```

- **بند capture:** بندهای خواندن و به‌روزرسانی را همزمان با هم ترکیب می‌کند. مثال (۱-۱۰-۲۱) را در نظر بگیرید. در خط شماره ۴ این مثال، هماهنگ ساز atomic همراه با بند capture مورد استفاده قرار گرفته است. به این مفهوم که هر رشته علاوه بر اینکه مقدار متغیر x می‌خواند، مقدار آن را نیز آپدیت می‌کند. در خط ۶ می‌تواند یکی از گزینه‌های آورده شده مورد استفاده قرار بگیرد. در لحظه‌ای که یک رشته این عملیات را انجام می‌دهد، بقیه رشته‌ها اجازه این کار را نخواهند داشت.

مثال ۱-۱۰-۲۱. استفاده از هماهنگ‌ساز atomic همراه با بند capture در محیط OpenMP

```
int x=1, var;                                                    ۱
#pragma omp parallel                                             ۲
{                                                                ۳
  #pragma omp atomic capture                                     ۴
  //one of the forms:                                            ۵
  v=x++; or v=x--; or v=++x; or v=--x; or v=x binop expr;         ۶
}                                                                ۷
```

۱-۱۱ نحوه اجرای برنامه موازی در محیط برنامه‌نویسی MPI

مثال (۱-۱۱-۱) را در نظر بگیرید که ساده‌ترین برنامه در محیط MPI می‌باشد.

مثال ۱-۱۱-۱. مثال ساده Hello World در محیط MPI

```
//MPI header                                                    ۱
#include <mpi.h>                                               ۲
#include <stdio.h>                                             ۳
                                                               ۴
int main(int argc, char** argv) {                              ۵
  // Initialize the MPI environment                            ۶
  MPI_Init(NULL, NULL);                                        ۷
                                                               ۸
  printf("Hello world MPI...\n");                              ۹
                                                               ۱۰
  // Finalize the MPI environment.                             ۱۱
  MPI_Finalize();                                              ۱۲
}                                                              ۱۳
```

شروع برنامه MPI با دستور MPI_Init() است که باعث می‌شود محیط MPI شروع به کار کردن کند. مهم نیست این دستور در کدام قسمت برنامه قرار بگیرد، تنها نکته مهم این است که قبل از هر دستور MPI دیگری فراخوانی شود. شکل کلی آن به صورت زیر می‌باشد:

MPI_Init (int** argc, char*** argv);

پارامترهای به کار رفته در آن به شرح زیر است.

argc: اشاره‌گری که به تعداد پارامترهای برنامه اشاره دارد. مقدار آن می‌تواند NULL باشد.

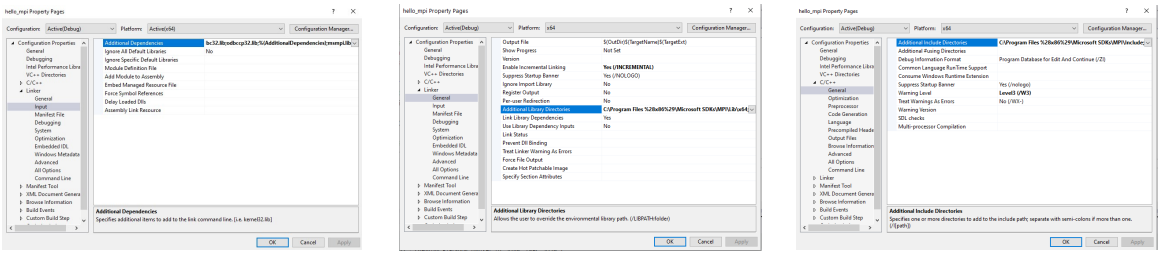
argv: اشاره‌گری که به لیستی از پارامترهای برنامه اشاره دارد و مقدار آن می‌تواند NULL باشد.

در انتهای برنامه، از دستور MPI_Finalize() استفاده شده است که محیط MPI را می‌بندد و منابعی که به آن اختصاص داده شده بود را آزاد می‌کند.

در ادامه نحوه اجرای برنامه فوق را در هریک از سیستم‌عامل‌های ویندوز و لینوکس توضیح می‌دهیم.

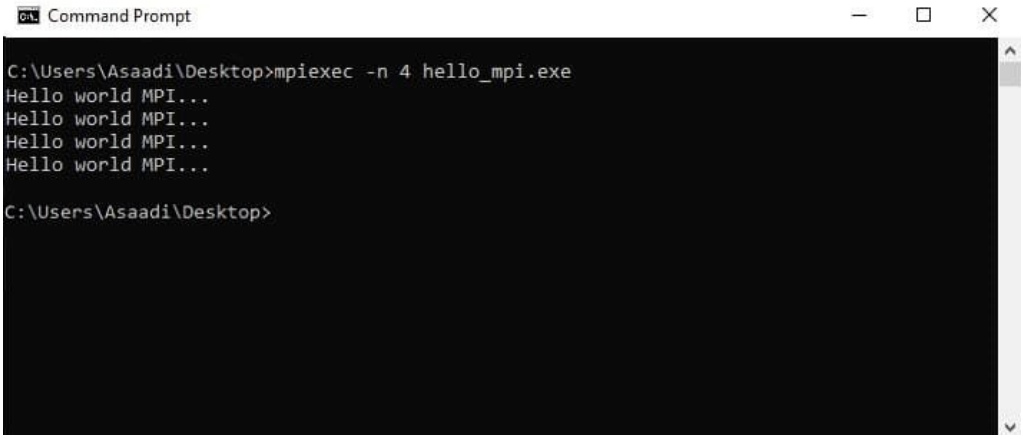
۱-۱۱-۱ اجرای برنامه موازی MPI در سیستم عامل ویندوز

- **Microsoft MPI(MS-MPI):** یک پیاده‌سازی مایکروسافت از استاندارد MPI است که برای اجرای برنامه‌های موازی بر روی محیط ویندوز مورد استفاده قرار می‌گیرد. کامپایل برنامه با استفاده از این کامپایلر در ویندوز به صورت گرافیکی و با استفاده از برنامه ویژوال استودیو خواهد بود. به این صورت که برای کامپایل برنامه بر روی پروژه راست کلیک کرده و گزینه properties را انتخاب می‌کنیم، پس از آن مطابق تصویر (۱-۱۲) تنظیمات را تغییر می‌دهیم. در این پایان‌نامه از برنامه Visual Studio 2015 استفاده شده است.



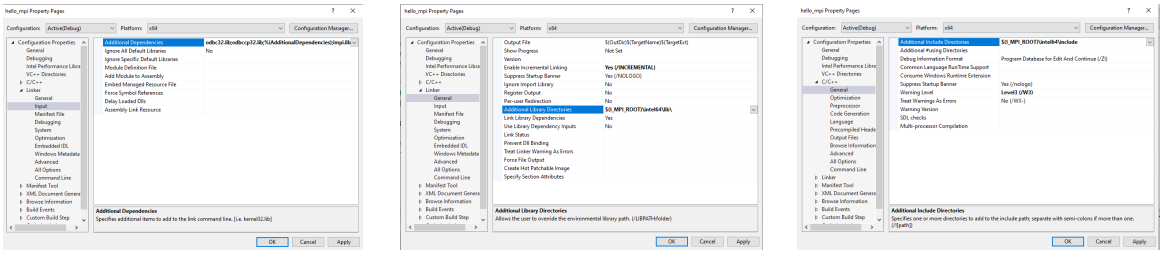
شکل ۱-۱۲: تنظیمات ویژوال استودیو برای اجرای برنامه MPI با استفاده از کامپایلر MS-MPI

اجرای برنامه MPI با استفاده از کامپایلر MS-MPI در سیستم‌عامل ویندوز به صورت خط فرمانی در محیط cmd انجام می‌پذیرد. تصویر (۱-۱۳) خروجی اجرای مثال (۱-۱۱-۱) را با استفاده از ۴ هسته رایانه و کامپایلر MS-MPI در سیستم‌عامل ویندوز نمایش می‌دهد.



شکل ۱-۱۳: اجرای مثال ۱-۱۱-۱ با استفاده از کامپایلر MS-MPI در ویندوز

- **IntelMPI:** از این کامپایلر برای کامپایل و اجرای برنامه‌های MPI در سیستم‌عامل‌های ویندوز و لینوکس استفاده می‌شود. برای کامپایل برنامه در ویندوز همانند کامپایلر MS-MPI، تنها اجرای برنامه به صورت خط فرمانی انجام می‌گیرد و کامپایل برنامه به صورت گرافیکی و درون محیط ویژوال استودیو خواهد بود. تصویر (۱-۱۴) نحوه تنظیم کتابخانه‌ها برای کامپایل مثال (۱-۱۱-۱) را نمایش می‌دهد.



شکل ۱-۱۴: تنظیمات ویژوال استودیو برای اجرای برنامه با استفاده از کامپایلر IntelMPI

۱-۱۱-۲ اجرای برنامه موازی MPI در سیستم عامل لینوکس

- **OpenMPI:** یک پروژه کتابخانه MPI است که توسط بسیاری از ابر رایانه‌های TOP500 [۱] استفاده می‌شود. پروژه OpenMPI یک پیاده‌سازی متن باز [۲] از MPI می‌باشد که توسط جمعی از شرکای دانشگاهی، تحقیقاتی و صنعتی توسعه و نگهداری می‌شود. Open MPI با استانداردهای MPI-3.1 انطباق کامل دارد [۳]. در جدول (۱-۴) دستورات مربوط به هر زبان برای کامپایل برنامه با استفاده از کامپایلر OpenMPI آورده شده است:

جدول ۱-۴: دستورات کامپایل برنامه توسط کامپایلر OpenMPI در سیستم عامل لینوکس

Language	Command
C	mpicc Sourse.c -o out
C++	mpicxx Sourse.cpp -o out

پس از کامپایل برنامه برای اجرای آن از دو دستور زیر می‌توان استفاده نمود.

mpirun -n <num-of-procs> out

mpexec -n <num-of-procs> out

هر دو دستور برای اجرای برنامه MPI استفاده می‌شود. دستور mpexec در استاندارد MPI تعریف شده است اما دستور mpirun فرمانی است که توسط بسیاری از پیاده سازی‌های MPI اجرا می‌شود و استاندارد نشده است. به همین دلیل هنگام اجرای برنامه دستور mpexec ترجیح داده می‌شود. خروجی حاصل از اجرای مثال (۱-۱۱-۱) در سیستم‌عامل لینوکس با استفاده از کامپایلر OpenMPI در تصویر (۱-۱۵) آورده شده است.

```
asaadi@DESKTOP-1J91ENU:/mnt/i/Parallel-CG/code $ mpicxx hello_mpi.cpp -o out
asaadi@DESKTOP-1J91ENU:/mnt/i/Parallel-CG/code $ mpiexec -n 4 ./out
Hello world from MPI...
Hello world from MPI...
Hello world from MPI...
Hello world from MPI...
asaadi@DESKTOP-1J91ENU:/mnt/i/Parallel-CG/code $
```

شکل ۱-۱۵: کامپایل و اجرای خط فرمانی مثال ۱-۱۱-۱ با کامپایلر OpenMPI در لینوکس

- **IntelMPI:** با استفاده از دستورات موجود در جدول (۱-۵) می‌توانیم برنامه MPI را در سیستم‌عامل لینوکس کامپایل کنیم.

---

[۱] پروژه TOP500 تعداد ۵۰۰ تا از قوی‌ترین سیستم‌های رایانه‌ای غیر توزیعی در جهان را رتبه‌بندی می‌کند. این پروژه از سال ۱۹۹۳ آغاز شده و در هر سال دو بار لیست ابر رایانه‌ها را منتشر می‌کند. برای اطلاعات بیشتر به سایت https://www.top500.org مراجعه کنید.

[۲] open-source

[۳] برای جزئیات بیشتر می‌توانید به سایت https://www.open-mpi.org مراجعه کنید.

جدول ۱-۵: دستورات کامپایل برنامه MPI توسط کامپایلر Intel در سیستم عامل لینوکس

Language	Command
C	mpiicc Sourse.c -o out
C++	mpiicpc Sourse.cpp -o out

۱-۱۲ برخی از دستورات موجود در محیط برنامه‌نویسی MPI

۱-۱۲-۱ تعیین تعداد پردازشگرها و رتبه‌بندی آنان

به مثال (۱-۱۲-۱) توجه کنید. در خط شماره ۵ محیط MPI شروع به کار خواهد کرد و در خط ۱۶ این برنامه محیط MPI به پایان رسیده و بسته خواهد شد. زمانی که MPI شروع به کار می‌کند، برای نقل و انتقال پیام(داده) در بین پردازشگرها به یک گروه ارتباطی نیاز است و باید یک گروه ارتباطی تعریف کنیم. به طور مثال گروه ارتباطی MPI_COMM_WORLD که به صورت پیش‌فرض از تمام پردازشگرها تشکیل شده است. البته امکان این نیز وجود دارد که برخی از پردازشگرهای گروه ارتباطی را استخراج کنیم و یک گروه ارتباطی دیگر را تشکیل بدهیم و از آن استفاده کنیم. اما در کل کارهایی که ما در این پایان‌نامه انجام می‌دهیم، از محیطی که MPI_COMM_WORLD برایمان فراهم می‌کند استفاده می‌کنیم. بعد از اینکه MPI_COMM_WORLD ساخته شده و محیط ارتباط موازی فراهم شد، نیاز است تا هر پردازشگر رتبه خود را دریافت کند. با اجرای دستور MPI_Comm_rank در خط شماره ۸ این برنامه هرکدام از تعداد p پردازشگرهایی که در گروه ارتباطی حضور دارند، رتبه بین ۰ تا p-1 را دریافت می‌کنند. همچنین در خط ۱۱ از دستور MPI_Comm_size استفاده شده است که این دستور تعداد کل پردازشگرها را در یک گروه ارتباطی مشخص می‌کند. این برنامه را در ماشین اشتراکی با ۴ هسته اجرا شده که هر هسته به عنوان یک پردازشگر در نظر گرفته شده است.

مثال ۱-۱۲-۱. تعیین تعداد پردازشگرها و رتبه‌بندی آنان در محیط MPI

```
#include <mpi.h>                                                    ۱
#include <stdio.h>                                                  ۲
int main(int argc, char** argv) {                                   ۳
  // Initialize the MPI environment                                 ۴
  MPI_Init(NULL, NULL);                                             ۵
  // Get the rank of the process                                    ۶
  int world_rank;                                                   ۷
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);                       ۸
  // Get the number of processes                                    ۹
  int world_size;                                                   ۱۰
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);                       ۱۱
  // Print a hello world message                                    ۱۲
  printf("Hello world from processor %d out of %d processors.\n",   ۱۳
  world_rank, world_size);                                          ۱۴
  // Finalize the MPI environment.                                  ۱۵
  MPI_Finalize();                                                   ۱۶
}                                                                   ۱۷
```

خروجی:

```
Hello world from processor 0 out of 4 processors.
Hello world from processor 1 out of 4 processors.
Hello world from processor 2 out of 4 processors.
Hello world from processor 3 out of 4 processors.
```

۱-۱۲-۲ دستورات ارتباطی MPI

در نقل و انتقال پیام(داده) ممکن است فقط دو پردازنده یا بیش از دو پردازنده مشارکت داشته باشند. اکنون به معرفی برخی از دستورات ارتباطی پرداخته و نحوه کار آن‌ها را شرح می‌دهیم [۶].

- **دستور MPI_Send:** برای ارسال پیام(داده) از یک پردازشگر به پردازشگر دیگر استفاده می‌شود و شکل آن به صورت زیر است:

```
MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

حال به توضیح هریک از متغیرهای به کار رفته در این دستور می‌پردازیم.

buf: اشاره‌گری از نوع ثابت و void که به آدرس بافر ارسالی اشاره می‌کند.

count: متغری از نوع صحیح که تعداد عناصر موجود در بافر ارسالی را نشان می‌دهد.

datatype: متغیری از نوع MPI_Datatype است که نوع داده هریک از عناصر بافر ارسالی را مشخص می‌کند. نوع داده‌ها در MPI از نوع MPI_Datatype تعریف می‌شود که در زبان C و Fortran به شکل‌های مختلف استفاده می‌شود. برخی از انواع داده که در C استفاده می‌شود، در جدول (۱-۶) آورده شده است.

جدول ۱-۶: انواع داده در MPI

انواع داده در MPI	معادل نوع داده در C,C++
MPI_INT	int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_CHAR	char
MPI_SHORT	short int
MPI_LONG	long int
MPI_BOOL	bool

dest: متغیری از نوع صحیح است که رتبه پردازشگری را مشخص می‌کند که باید پیام به آن ارسال شود.

tag: متغیری از نوع صحیح که برچسب یا تمبر چسبیده بر روی بافر ارسالی را نشان می‌دهد. این برچسب توسط پردازشگری که کار ارسال را انجام می‌دهد، بر روی بافر ارسالی چسبیده می‌شود.

comm: متغیری از نوع MPI_Comm می‌باشد که گروه ارتباطی پردازشگرها را مشخص می‌کند.

- **دستور MPI_Recv:** وقتی یک پیام(داده) از یک پردازشگر به پردازشگر دیگر ارسال می‌شود. برای دریافت این پیام در پردازشگر دوم از دستور MPI_Recv استفاده می‌شود و شکل آن به صورت زیر می‌باشد:

MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)

buf: یک اشاره‌گر ازنوع void می‌باشد که به آدرس بافر دریافتی اشاره می‌کند. این اشاره‌گر متغیر ثابت نیست زیرا ممکن است پس از دریافت داده، عملیاتی بر روی آن انجام شود.

count: متغیری از نوع صحیح که تعداد عناصر موجود در بافر دریافتی می‌باشد.

datatype: متغیری از نوع MPI_Datatype است که نوع داده عناصر بافر دریافتی را مشخص می‌کند. برای اطلاع از انواع داده‌ها در MPI به جدول (۱-۶) مراجعه کنید.

dest: رتبه پردازشگری که پیام از آن دریافت می‌شود.

tag: برچسبی که بر روی بافر پیام ارسالی چسبیده شده است.

comm: متغیری از نوع MPI_Comm می‌باشد که گروه ارتباطی پردازشگرها را مشخص می‌کند.

status: متغیری از نوع MPI_Status می‌باشد که حاوی اطلاعاتی در مورد وضعیت پیام دریافتی می‌باشد.

MPI_Status: یک ساختار است که در یکی از جاهایی که از آن استفاده می‌شود، تابع MPI_Recv می‌باشد. به این مفهوم که یک پیام ارسال شده و ما باید وضعیت دریافت پیام را بررسی می‌کنیم که آیا به درستی دریافت شده است یا خیر. البته این ساختار در توابع انتظار [1] و آزمایش [2] نیز استفاده می‌شود که در بخش (۱-۱۲-۳) به توضیح بیشتر این دو تابع خواهیم پرداخت. شکل این ساختار به صورت زیر می‌باشد:

```
Struct MPI_Status{
        int count, → مقدار ورودی دریافت شده
        int cancelled, → اشاره به لغو درخواست مربوطه
        MPI_SOURCE, → پردازشگری که پیام از آن دریافت شده
        int MPI_TAG, → برچسب پیام
        int MPI_ERROR → خطای مربوط به پیام
};
```

حال برای فهم بهتر توابع MPI_Send و MPI_Recv به توضیح مثال (۱-۱۲-۲) می‌پردازیم. در این مثال از

[1] MPI_Wait [2] MPI_Test

دستورات MPI_Send و MPI_Recv استفاده شده است و ساختار آن به صورتی است که فقط می‌تواند با دو پردازنده اجرا شود. در مورد دستورات MPI_Init و MPI_Finalize در خطوط ۴ و ۱۸، همچنین دستورات MPI_Comm_rank و MPI_Comm_size در خط ۶ و ۷ برنامه قبلا توضیح داده شده است. قالب این برنامه طوری طراحی شده است که پردازشگر شماره صفر داده را ارسال و پردازشگر شماره ۱ آن را دریافت خواهد کرد. خط شماره ۱۰ را در نظر بگیرید که در آن پردازشگر صفر مقدار ۱۲۳۴۵ را در قالب buffer_send با اندازه ۱ می‌فرستد. پیام ارسالی از نوع صحیح بوده و به پردازشگر شماره ۱ ارسال می‌شود. برچسب پیام ۱۰۰ بوده و از طریق گروه ارتباطی MPI_COMM_WORLD فرستاده شده که تمام پردازشگرهای محیط(دو پردازشگر) را درگیر کرده است. خط شماره ۱۳ مشخص می‌کند که عملیات دریافت تنها توسط پردازشگر شماره ۱ انجام می‌شود. در خط شماره ۱۵ عملیات دریافت به این صورت اتفاق می‌افتد که پردازشگر شماره ۱ بافر received را به اندازه ۱ و از نوع MPI_INT دریافت می‌کند. این بافر دریافتی از پردازشگر شماره صفر دریافت شده که برچسب آن ۱۰۰ می‌باشد. فضای کاری MPI_COMM_WORLD بوده و وضعیت دریافت پیام در متغیر status ذخیره خواهد شد.

مثال ۱-۱۲-۲. استفاده از دستورات MPI_Send و MPI_Recv در محیط MPI

```
//This application is meant to be run with 2 processes                    ۱
MPI_Status status;                                                        ۲
int main(int argc, char* argv[]){                                         ۳
  MPI_Init(&argc, &argv);                                                 ۴
  int my_rank, world_size;                                                ۵
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);                                ۶
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);                             ۷
  if(my_rank==0) {                                                        ۸
        int buffer_send = 12345;                                          ۹
        MPI_Send(&buffer_send,1,MPI_INT,1,100,MPI_COMM_WORLD);            ۱۰
        printf("MPI process %d sends value %d.\n", my_rank,               ۱۱
            buffer_send);
  }                                                                       ۱۲
  else if(my_rank==1) {                                                   ۱۳
        int received;                                                     ۱۴
        MPI_Recv(&received,1,MPI_INT,0,100,MPI_COMM_WORLD,                ۱۵
            &status);
        printf("MPI process %d received value %d.\n", my_rank,            ۱۶
            received);
  }                                                                       ۱۷
  MPI_Finalize();                                                         ۱۸
}                                                                         ۱۹
```

خروجی به ازای دو پردازشگر:

MPI process 0 sends value 12345.
MPI process 1 received value 12345.

- **دستور MPI_Bcact:** یکی از دستورات ارتباط پیام است که در آن بیش از دو پردازشگر مشارکت دارند. در این دستور، داده‌ای که در دسترس یکی از پردازشگرها می‌باشد، برای همه پردازشگرهای موجود در گروه ارتباطی

ارسال می‌شود. فرض کنید در یک برنامه موازی ورودی باید از کاربر دریافت شود. یکی از موارد استفاده از دستور MPI_Bcact به این صورت می‌باشد که یکی از پردازشگرها مسئول گرفتن ورودی از کاربر باشد و سپس با استفاده از دستور MPI_Bcact، ورودی خوانده شده بدون اینکه در آن دستکاری رخ دهد به صورت یکسان به تمام پردازشگرهای دیگر در گروه ارتباطی ارسال شود. شکل کلی این دستور به صورت زیر می‌باشد:

MPI_Bcast(void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)

حال به توضیح هریک از متغیرهای استفاده شده در این دستور می‌پردازیم:

buffer: اشاره‌گری از نوع void می‌باشد که به آدرس بافر ارسالی اشاره می‌کند. ممکن است بافر یک متغیر یا یک آرایه باشد.

count: متغیر از نوع صحیح می‌باشد که تعداد عناصر بافر ارسالی را نشان می‌دهد.

datatype: نوع داده بافر ارسالی می‌باشد که از نوع MPI_Datatype است و قبلا در جدول (۱-۶) به آن اشاره شده است.

root: رتبه پردازشگری که از طریق آن عملیات ارسال انجام می‌گیرد.

comm: محیط ارتباطی که از نوع MPI_Comm است.

مثال (۱-۱۲-۳) را در نظر بگیرید که در آن از دستور MPI_Bcast استفاده شده است. این برنامه بر روی ماشینی با ۴ هسته اجرا شده که محیط MPI هر کدام از این هسته‌ها را به عنوان یک پردازشگر مستقل در نظر می‌گیرد. در خط شماره ۱۲ متغیر buffer با مقدار ۱۲۳۴۵ که از نوع MPI_INT می‌باشد، در دسترس پردازشگر صفر (broadcast_root) است و با دستور MPI_Bcast این متغیر برای تمام پردازشگرهای گروه ارتباطی ارسال می‌شود. این نقل و انتقالات در فضای MPI_COMM_WORLD انجام می‌شود که در اینجا متشکل از ۴ هسته ماشین می‌باشد. در خط شماره ۱۴ تمام پردازنده‌هایی که رتبه غیر صفر دارند، متغیری که دریافت کرده‌اند را نمایش می‌دهند. خروجی برنامه در ادامه آورده شده که نشان دهنده این است که پردازشگر صفر متغیر ۱۲۳۴۵ را ارسال کرده و پردازشگرهای ۱ تا ۳ آن متغیر را دریافت کرده‌اند.

مثال ۱-۱۲-۳. استفاده از دستور MPI_Bcast در محیط MPI

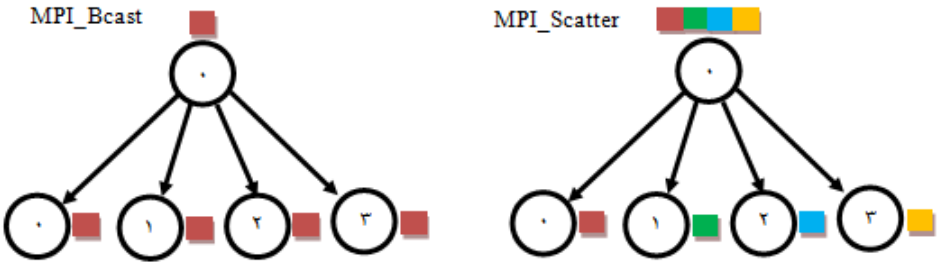
```
int main(int argc, char* argv[]){                                    ۱
  MPI_Init(&argc, &argv);                                            ۲
  int my_rank, world_size;                                           ۳
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);                           ۴
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);                        ۵
  int broadcast_root = 0;                                            ۶
  int buffer;                                                        ۷
  if (my_rank == broadcast_root) {                                   ۸
    buffer = 12345;                                                  ۹
    printf("(MPI process %d) I am broadcast root,and send value      ۱۰
       %d.\n",my_rank, buffer);
  }                                                                  ۱۱
  MPI_Bcast(&buffer,1,MPI_INT,broadcast_root,MPI_COMM_WORLD);        ۱۲
```

```
  if (my_rank != broadcast_root) {                                    ۱۳
    printf("(MPI process %d) I am a broadcast receiver,and            ۱۴
        obtained value %d.\n", my_rank, buffer);
  }                                                                   ۱۵
  MPI_Finalize();                                                     ۱۶
}                                                                     ۱۷
```

خروجی به ازای ۴ پردازشگر:

(MPI process 0) I am the broadcast root, and send value 12345.
(MPI process 1) I am a broadcast receiver, and obtained value 12345.
(MPI process 2] I am a broadcast receiver, and obtained value 12345.
(MPI process 3] I am a broadcast receiver, and obtained value 12345.

- **دستور MPI_Scatter:** فرض کنیم که یک آرایه که در دسترس یک پردازشگر می‌باشد. می‌خواهیم این آرایه را به تکه‌های مساوی بین بقیه پردازشگرهای گروه ارتباطی پخش کنیم. برای این کار می‌توانیم از دستور MPI_Scatter استفاده کنیم. تفاوت این دستور، با دستور MPI_Bcast در این است که MPI_Scatter تکه‌های مساوی از یک آرایه را به پردازشگرهای مختلف می‌فرستد. اما MPI_Bcast دقیقا همان داده را به پردازشگرهای گروه ارتباطی ارسال می‌کند.



شکل ۱-۱۶: نحوه توزیع داده‌ها در MPI_Bcast و MPI_Scatter

شکل کلی این دستور به صورت زیر می‌باشد:

MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

حال به توضیح هریک از متغیرهای استفاده شده در این دستور می‌پردازیم:

sendbuf: آدرس بافر ارسالی از نوع void می‌باشد و متغیر ثابت است.

sendcount: متغیر از نوع صحیح که تعداد عناصر بافر ارسالی به هر پردازشگر را نشان می‌دهد.

sendtype: نوع داده ارسالی به هر پردازشگر که از نوع MPI_Datatype می‌باشد. که جزئیات مربوط به آن در جدول (۱-۶) می‌باشد.

recvbuf: آدرس بافر دریافتی که از نوع void است.

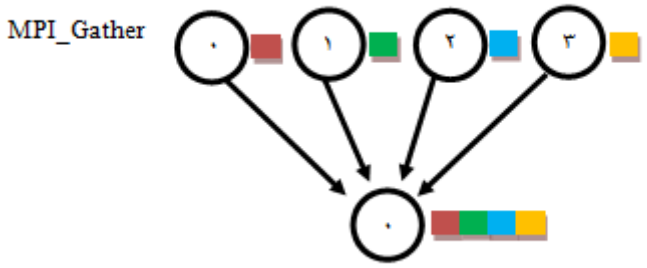
recvcount: تعداد عناصر موجود در بافر دریافتی که متغیر صحیح می‌باشد.

recvtype: متغیری از نوع MPI_Datatype که نوع داده بافر دریافتی را مشخص می‌کند.

root: متغیری از نوع صحیح که رتبه پردازشگر ارسال‌کننده آرایه را مشخص می‌کند.

comm: نشان دهنده گروه ارتباطی که در آن دستور MPI_Scatter انجام می‌شود.

- **دستور MPI_Gather:** این دستور، عکس دستور MPI_Scatter است. فرض کنیم تکه‌های مختلف داده با اندازه‌های مساوی در دسترس پردازشگرهای موجود در گروه ارتباطی باشند. با دستور MPI_Gather این تکه‌های داده از تمام پردازشگرهای گروه ارتباطی دریافت شده و به هم متصل می‌شوند و آرایه تشکیل شده بر روی یک پردازشگر مشخص قرار خواهد گرفت. این دستور به شکل زیر مورد استفاده قرار می‌گیرد:



شکل ۱-۱۷: نحوه جمع‌آوری داده‌ها در MPI_Gather

MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

حال به توضیح هریک از متغیرهای استفاده شده در این دستور می‌پردازیم:

sendbuf: آدرس بافر ارسالی از هر پردازشگر که از نوع void می‌باشد و متغیر ثابت است.

sendcount: متغیر از نوع صحیح که تعداد عناصر بافر ارسالی از هر پردازشگر را نشان می‌دهد.

sendtype: نوع داده ارسالی از هر پردازشگر که از نوع MPI_Datatype می‌باشد. در بخش MPI_Send جزئیات مربوط به آن، توضیح داده شده است.

recvbuf: آدرس بافر دریافتی که از نوع void است.

recvcount: تعداد عناصر موجود در بافر دریافتی که متغیر صحیح می‌باشد.

recvtype: متغیری از نوع MPI_Datatype که نوع داده بافر دریافتی را مشخص می‌کند.

root: متغیری از نوع صحیح که رتبه پردازشگر دریافت‌کننده آرایه را مشخص می‌کند.

comm: نشان دهنده گروه ارتباطی که در آن دستور MPI_Gather انجام می‌شود.

مثال (۱-۱۲-۴) را در نظر بگیرید که در آن دستورات MPI_Scatter و MPI_Gather مورد استفاده قرار گرفته

است. این برنامه بر روی ماشینی با ۴ هسته اجرا شده که هر هسته به عنوان یک پردازشگر در نظر گرفته می‌شود. در خط شماره ۱۰ تا ۱۳ این برنامه، آرایه buffer با طول ۴، توسط پردازشگر شماره صفر (root_rank) مقداردهی می‌شود. در خط شماره ۱۶ تعداد ۱ عنصر از این آرایه که از نوع MPI_INT می‌باشد، برای پردازشگرهای گروه ارتباطی ارسال می‌شود. هر پردازشگر ۱ داده دریافتی از نوع MPI_INT را در my_value ذخیره می‌کند. این نقل و انتقالات نیز در گروه ارتباطی MPI_COMM_WORLD انجام می‌شود. در خط شماره ۱۹ هر پردازشگر my_value دریافت شده را با شماره رتبه خود جمع کرده و مقدار آن را آپدیت می‌کند. در ادامه در خط ۲۰ برنامه، مقادیر my_value با اندازه ۱ و نوع MPI_INT از تمام پردازشگرها جمع‌آوری و به یکدیگر متصل می‌شوند. پس از آن داده‌های متصل شده، در آرایه collect_buffer در پردازشگر شماره صفر ذخیره خواهد شد. خروجی برنامه در ادامه آورده شده که پردازشگر شماره صفر تکه‌های مساوی از آرایه را بین پردازشگرهای موجود در گروه ارتباطی پخش کرده و هر یک از پردازشگرها، داده دریافت شده خود را نمایش می‌دهند. و پس از آن مجددا تکه‌های آرایه به‌روز شده، از پردازشگرها جمع‌آوری شده و در قالب یک آرایه متصل در دسترس پردازشگر صفر قرار می‌گیرد.

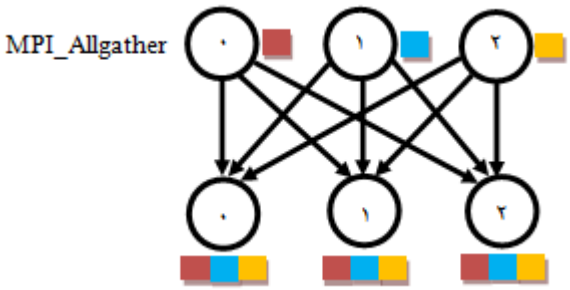
مثال ۱-۱۲-۴. استفاده از دستورات MPI_Scatter و MPI_Gather در محیط MPI

```
int main(int argc, char* argv[]){                                    ۱
  MPI_Init(&argc, &argv);                                            ۲
  int my_rank, world_size;                                           ۳
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);                           ۴
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);                        ۵
                                                                     ۶
  int root_rank = 0;                                                 ۷
  int buffer[4], my_value, collect_buffer[4];                        ۸
                                                                     ۹
  if (my_rank == root_rank){                                         ۱۰
    for(int i=0; i<4; i++){                                          ۱۱
      buffer[i] = i*100;                                             ۱۲
    }                                                                ۱۳
    printf("Process %d scatter the buffer: %d, %d, %d, %d.\n",       ۱۴
        my_rank,buffer[0],buffer[1],buffer[2],buffer[3]);
  }                                                                  ۱۵
  MPI_Scatter(buffer,1,MPI_INT,&my_value, 1,MPI_INT, root_rank,      ۱۶
     MPI_COMM_WORLD);
  printf("Process %d received value = %d.\n",my_rank,my_value);      ۱۷
                                                                     ۱۸
  my_value = my_value + my_rank;                                     ۱۹
  MPI_Gather(&my_value,1,MPI_INT,collect_buffer,1,MPI_INT,           ۲۰
     root_rank,MPI_COMM_WORLD);
                                                                     ۲۱
  if (my_rank == root_rank) {                                        ۲۲
    printf("Values collected on process %d: %d,%d,%d,%d.\n",         ۲۳
        my_rank,collect_buffer[0],collect_buffer[1],collect_buffer
        [2],collect_buffer[3]);
  }                                                                  ۲۴
  MPI_Finalize();                                                    ۲۵
}                                                                    ۲۶
```

خروجی به ازای ۴ پردازشگر:

Process 0 scatter the buffer: 0, 100, 200, 300.
Process 0 received value =0.
Process 1 received value =100.
Process 2 received value =200.
Process 3 received value =300.
Values collected on process 0: 0, 101, 202, 303.

- **دستور MPI_Allgather:** فرض کنیم یک آرایه به چندین بخش مساوی تقسیم‌شده و هر تکه از آرایه در دسترس یکی از پردازشگرهای گروه ارتباطی باشد. دستور MPI_Allgather این تکه‌ها را از پردازشگرهای مختلف گرفته و آن‌ها را به هم متصل می‌کند و یک آرایه واحد تشکیل می‌دهد. سپس این آرایه تشکیل شده را در اختیار تمام پردازشگرهای گروه ارتباطی قرار می‌دهد. در واقع دستور MPI_Allgather ترکیبی از دو دستور MPI_Gather و MPI_Bcast می‌باشد.



شکل ۱-۱۸: نحوه جمع‌آوری داده‌ها در MPI_Allgather

این دستور به شکل زیر استفاده می‌شود:

MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)

هریک از متغیرهای به کار رفته در آن به شرح زیر می‌باشد:

sendbuf: آدرس بافر ارسالی از نوع void می‌باشد و متغیر ثابت است.

sendcount: متغیر از نوع صحیح که تعداد عناصر بافر ارسالی از هر پردازشگر را نشان می‌دهد.

sendtype: نوع داده ارسالی از هر پردازشگر که از نوع MPI_Datatype می‌باشد. در بخش MPI_Send جزئیات مربوط به آن، توضیح داده شده است.

recvbuf: آدرس بافر دریافتی که از نوع void است.

recvcount: تعداد عناصر موجود در بافر دریافتی که متغیر صحیح می‌باشد.

recvtype: متغیری از نوع MPI_Datatype که نوع داده بافر دریافتی را مشخص می‌کند.

comm: نشان دهنده گروه ارتباطی که از نوع MPI_Comm می‌باشد.

مثال (۱-۱۲-۵) را در نظر بگیرید که با استفاده از ۳ هسته ماشین اجرا شده است. در خط شماره ۶ این برنامه هر پردازشگر درون گروه ارتباطی، پس از اینکه رتبه خود را با دستور MPI_Comm_rank دریافت کرد، مقدار آن را با عدد ۱۰۰ ضرب کرده و در متغیر my_value ذخیره می‌کند. در خط شماره ۹ با استفاده از دستور MPI_Allgather متغیر my_value با اندازه ۱ و نوع صحیح از تمام پردازشگرها جمع‌آوری شده و به هم متصل می‌شوند. پس از آن در آرایه buffer ذخیره می‌شود که در اختیار تمام پرازشگرهای گروه ارتباطی می‌باشد.

مثال ۱-۱۲-۵. استفاده از دستور MPI_Allgather در محیط MPI

```
int main(int argc, char* argv[]){                                    ۱
  MPI_Init(&argc, &argv);                                            ۲
  int my_rank, world_size;                                           ۳
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);                           ۴
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);                        ۵
  int my_value = my_rank *100;                                       ۶
  printf("Process %d, my value = %d.\n",my_rank,my_value);           ۷
  int buffer[world_size];                                            ۸
  MPI_Allgather(&my_value,1,MPI_INT,buffer,1,MPI_INT,                ۹
     MPI_COMM_WORLD);
  printf("Process %d received value = %d, %d, %d.\n",my_rank,        ۱۰
     buffer[0],buffer[1],buffer[2]);
  MPI_Finalize();                                                    ۱۱
}                                                                    ۱۲
```

خروجی به ازای ۳ پردازشگر:

```
Process 0, my value =0.
Process 1, my value =100.
Process 2, my value =200.
Process 0 received value = 0, 100, 200.
Process 1 received value = 0, 100, 200.
Process 2 received value = 0, 100, 200.
```

فرض کنیم که می‌خواهیم یک آرایه را با دستور MPI_Scatter بین پردازشگرها تقسیم کنیم، و یا اینکه با استفاده از دستور MPI_Gather یا MPI_Allgather تکه‌هایی از آرایه را از تمام پردازشگرهای گروه ارتباطی جمع‌آوری کنیم. در هنگام استفاده از این دستورات اندازه تکه‌های آرایه یکسان خواهد بود. اما اگر بخواهیم آرایه را به تکه‌هایی با اندازه‌های متفاوت در بین پردازشگرها تقسیم کنیم، از دستور MPI_Scatterv استفاده خواهیم کرد. همچنین برای جمع‌آوری تکه‌های آرایه با اندازه نامساوی از پردازشگرهای گروه ارتباطی، دستورات MPI_Gatherv و MPI_Allgatherv مورد استفاده قرار خواهند گرفت. در ادامه به توضیح هر یک از این دستورات می‌پردازیم.

- **دستور MPI_Scatterv:** فرض کنیم که یک آرایه که در دسترس یک پردازشگر می‌باشد. می‌خواهیم این آرایه مفروض را به تکه‌هایی با اندازه نامساوی بین پردازشگرهای گروه ارتباطی پخش کنیم. برای این کار از دستور MPI_Scatterv استفاده می‌کنیم. شکل آن به صورت زیر است:

MPI_Scatterv(const void *sendbuf, const int *sendcounts, const int *displs, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)

متغیرهایی که در آن استفاده شده است را شرح می‌دهیم:

sendbuf: آدرس بافر ارسالی که از نوع void می‌باشد و متغیری ثابت است.

sendcount: آرایه‌ای از اعداد صحیح که شامل تعداد عناصر بافر ارسالی به هر پردازشگر است. طول این آرایه به اندازه تعداد پردازشگرهای گروه ارتباطی می‌باشد.

displs: آرایه‌ای از اشاره‌گرها که به اندازه تعداد پردازشگرهای گروه ارتباطی است. عناصر این آرایه به موقعیت شروع تکه مربوط به هر پردازشگر، در بافر ارسالی اشاره می‌کنند.

sendtype: نوع داده بافر ارسالی به هر پردازنده که از نوع MPI_Datatype می‌باشد.

recvbuf: آدرس بافر دریافتی که از نوع void است.

recvcount: تعداد عناصر موجود در بافر دریافتی که متغیر صحیح می‌باشد.

recvtype: متغیری از نوع MPI_Datatype که نوع داده بافر دریافتی را مشخص می‌کند.

root: متغیری از نوع صحیح که رتبه پردازشگر ارسال‌کننده آرایه را مشخص می‌کند.

comm: گروه ارتباطی که از نوع MPI_Comm می‌باشد.

- **دستور MPI_Gatherv:** این دستور، عکس دستور MPI_Scatterv می‌باشد. فرض کنیم تکه‌های مختلف داده با اندازه‌های متفاوت در دسترس پردازشگرهای موجود در گروه ارتباطی باشند. با دستور MPI_Gatherv این تکه‌های داده از تمام پردازشگرهای گروه ارتباطی دریافت شده و در جایگاه تعیین شده قرار گرفته و آرایه واحدی را تشکیل می‌دهند. در ادامه این آرایه تشکیل شده، بر روی یک پردازشگر مشخص قرار خواهد گرفت. این دستور به صورت زیر استفاده می‌شود:

MPI_Gatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int *recvcounts, const int *displs, MPI_Datatype recvtype, int root, MPI_Comm comm)

حال به توضیح هریک از متغیرهای استفاده شده در این دستور می‌پردازیم:

sendbuf: آدرس بافر ارسالی از هر پردازشگر که از نوع void می‌باشد و متغیر ثابت است.

sendcount: متغیری از نوع صحیح که تعداد عناصر بافر ارسالی از هر پردازشگر را نشان می‌دهد.

sendtype: نوع داده ارسالی از هر پردازشگر که از نوع MPI_Datatype می‌باشد.

recvbuf: آدرس بافر دریافتی که از نوع void است.

recvcount: آرایه‌ای از اعداد صحیح می‌باشد که شامل تعداد عناصر بافر دریافتی از هر پردازشگر است. طول این

آرایه به اندازه تعداد پردازشگرهای گروه ارتباطی می‌باشد.

displs: آرایه‌ای از اشاره‌گرها می‌باشد که به اندازه تعداد پردازشگرهای گروه ارتباطی است. عناصر این آرایه به موقعیت قرار گرفتن تکه‌های مربوط به هر پردازشگر اشاره می‌کند.

recvtype: متغیری از نوع MPI_Datatype که نوع داده بافر دریافتی را مشخص می‌کند.

root: متغیری از نوع صحیح که رتبه پردازشگر دریافت‌کننده آرایه را مشخص می‌کند.

comm: گروه ارتباطی که از نوع MPI_Comm می‌باشد.

به مثال (۱-۱۲-۶) دقت کنید که در آن از دستورات MPI_Scatterv و MPI_Gather مورد استفاده قرار گرفته است. این برنامه با استفاده از ۳ هسته ماشین اجرا شده است که هر هسته به عنوان یک پردازشگر در نظر گرفته می‌شود. در خط شماره ۱۵ تا ۱۷ این برنامه، آرایه buffer با طول ۵، توسط پردازشگر شماره صفر (root_rank) مقداردهی می‌شود. در خط شماره ۲۰، آرایه buffer با توجه به counts و displacements که در خطوط ۸ و ۹ تعریف شده‌اند، بین پردازشگرهای گروه ارتباطی به اندازه chunk تقسیم می‌شود. به طور مثال پردازشگر شماره صفر تعداد ۲ عنصر از موقعیت صفر آرایه buffer را دریافت کرده و در my_values مربوط به خود ذخیره خواهد کرد. برای پردازشگرهای ۱ و۲ نیز به همین ترتیب خواهد بود. در خطوط ۲۳ تا ۲۵ برنامه هر پردازشگر عناصر my_values دریافتی را با شماره رتبه خود جمع کرده و مقدار آن‌ها را به‌روز می‌کند. پس از آن در خط شماره ۲۶ آرایه‌های my_values با اندازه chunk از تمام پردازشگرها جمع‌آوری شده و با استفاده از counts و displacements در جایگاه مخصوص به خود در آرایه collect_buffer که در اختیار پردازشگر صفر است، قرار خواهند گرفت.

مثال ۱-۱۲-۶. استفاده از دستورات MPI_Scatterv و MPI_Gatherv در محیط MPI

```
int main(int argc, char* argv[]){                                    ۱
  MPI_Init(&argc, &argv);                                            ۲
  int my_rank, world_size;                                           ۳
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);                           ۴
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);                        ۵
                                                                     ۶
  int root_rank = 0;                                                 ۷
  int counts[world_size] = {2, 2, 1};                                ۸
  int displacements[world_size] = {0, 2, 4};                         ۹
                                                                     ۱۰
  int chunk = counts[my_rank];                                       ۱۱
  int buffer[5], my_values[chunk], collect_buffer[5];                ۱۲
                                                                     ۱۳
  if (my_rank == root_rank) {                                        ۱۴
    for(int i=0; i<5; i++){                                          ۱۵
      buffer[i] = i*100;                                             ۱۶
    }                                                                ۱۷
    printf("Process %d scatter the buffer: %d, %d, %d, %d, %d.       ۱۸
        \n",my_rank,buffer[0],buffer[1],buffer[2],buffer[3],
        buffer[4]);
  }                                                                  ۱۹
```

```
  MPI_Scatterv(buffer,counts,displacements,MPI_INT,&my_values,    ۲۰
     chunk,MPI_INT,root_rank,MPI_COMM_WORLD);
  printf("Process %d received my_values.\n", my_rank);            ۲۱
                                                                  ۲۲
  for(int i= 0 ; i <chunk ; i++) {                                ۲۳
    my_values[i] = my_values[i] + my_rank;                        ۲۴
  }                                                               ۲۵
  MPI_Gatherv(&my_values,chunk,MPI_INT,collect_buffer,counts,     ۲۶
     displacements,MPI_INT,root_rank,MPI_COMM_WORLD);
                                                                  ۲۷
  if (my_rank == root_rank) {                                     ۲۸
    printf("Values collected on process %d: %d, %d, %d, %d, %d.  ۲۹
       \n",my_rank,collect_buffer[0],collect_buffer[1],
       collect_buffer[2],collect_buffer[3],collect_buffer[4]);
  }                                                               ۳۰
  MPI_Finalize();                                                 ۳۱
}                                                                 ۳۲
```

خروجی به ازای ۳ پردازنده:

Process 0 scatter the buffer: 0, 100, 200, 300, 400.
Process 0 received values.
Process 1 received values.
Process 2 received values.
Values collected on process 0: 0, 100, 201, 301, 402.

- **دستور MPI_Allgatherv:** در واقع این دستور ترکیبی از دو دستور MPI_Gatherv و MPI_Bcast می‌باشد. فرض کنیم که تکه‌های آرایه با اندازه‌های نامساوی، در دسترس هر یک از پردازشگرهای گروه ارتباطی باشند. دستور MPI_Allgatherv این تکه‌ها را از پردازشگرهای مختلف گرفته و آن‌ها را در جایگاه تعیین شده در آرایه واحد قرار می‌دهد. سپس این آرایه تشکیل شده را در اختیار تمام پردازشگرهای گروه ارتباطی قرار خواهد داد. شکل کلی این دستور به صورت زیر می‌باشد:

MPI_Allgatherv(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, const int *recvcounts, const int *displs, MPI_Datatype recvtype, MPI_Comm comm)

حال هریک از متغیرهای استفاده شده در این دستور را شرح می‌دهیم:

sendbuf: آدرس بافر ارسالی از هر پردازشگر که از نوع void می‌باشد و متغیر ثابت است.

sendcount: متغیری از نوع صحیح که تعداد عناصر بافر ارسالی از هر پردازشگر را نشان می‌دهد.

sendtype: نوع داده ارسالی از هر پردازشگر که از نوع MPI_Datatype می‌باشد.

recvbuf: آدرس بافر دریافتی که از نوع void است.

recvcount: آرایه‌ای از اعداد صحیح می‌باشد که شامل تعداد عناصر بافر دریافتی از هر پردازشگر است. طول این آرایه به اندازه تعداد پردازشگرهای گروه ارتباطی می‌باشد.

displs: آرایه‌ای از اشاره‌گرها می‌باشد که به اندازه تعداد پردازشگرهای گروه ارتباطی است. عناصر این آرایه به

موقعیت قرار گرفتن تکه‌های مربوط به هر پردازشگر اشاره می‌کند.

recvtype: متغیری از نوع MPI_Datatype که نوع داده بافر دریافتی را مشخص می‌کند.

comm: گروه ارتباطی که از نوع MPI_Comm می‌باشد.

مثال ۱-۱۲-۷. استفاده از دستور MPI_Allgatherv در محیط MPI

```
int main(int argc, char* argv[]){                                    ۱
  MPI_Init(&argc, &argv);                                            ۲
  int my_rank, world_size;                                           ۳
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);                           ۴
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);                        ۵
                                                                     ۶
  int counts[world_size] = {2, 2, 1};                                ۷
  int displacements[world_size] = {0, 2, 4};                         ۸
                                                                     ۹
  int start = displacements[my_rank];                                ۱۰
  int chunk = counts[my_rank];                                       ۱۱
                                                                     ۱۲
  int buffer[5], my_values[chunk];                                   ۱۳
                                                                     ۱۴
  int size=0;                                                        ۱۵
  for(int i=start; i<start+chunk; i++){                              ۱۶
        my_values[size++] = i*100;                                   ۱۷
  }                                                                  ۱۸
  MPI_Allgatherv(&my_values,chunk,MPI_INT,buffer,counts,             ۱۹
     displacements,MPI_INT,MPI_COMM_WORLD);
  printf("Process %d received value = %d, %d, %d, %d, %d.\n",        ۲۰
     my_rank,buffer[0],buffer[1],buffer[2],buffer[3],buffer[4]);
  MPI_Finalize();                                                    ۲۱
}                                                                    ۲۲
```

خروجی به ازای ۳ پردازشگر:

Process 0 received value = 0, 100, 200, 300, 400.
Process 1 received value = 0, 100, 200, 300, 400.
Process 2 received value = 0, 100, 200, 300, 400.

- **دستور MPI_Allreduce و MPI_Reduce:** در دستور MPI_Reduce، داده‌ها(آرایه‌ها) از تمام پردازشگرهای گروه ارتباطی جمع‌آوری می‌شود. سپس با استفاده از عملگری که تعریف شده، ترکیب می‌شوند و داده(آرایه) به دست آمده در یک پردازنده مشخص قرار خواهد گرفت. نحوه عملکرد این دستور مشابه به دستور MPI_Gather می‌باشد، با این تفاوت که در دستور MPI_Gather پس از دریافت داده‌ها از پردازشگرهای گروه ارتباطی، نتایج در یک آرایه واحد به یکدیگر متصل می‌شوند. فرض کنیم می‌خواهیم تا نتیجه حاصل از اجرای دستور MPI_Reduce، بر روی تمام پردازشگرهای موجود در گروه ارتباطی قرار بگیرد. با دستور MPI_Allreduce داده‌ها از تمام پردازشگرهای گروه ارتباطی دریافت شده و پس از اینکه با عملگر تعریف شده ترکیب شوند، نتیجه در دسترس تمام پردازشگرها قرار خواهد گرفت. در واقع این دستور، دو دستور MPI_Reduce و MPI_Bcast را اجرا می‌کند. شکل کلی این

دستورات مشابه به یکدیگر و به صورت زیر می‌باشد. تفاوت این دو دستور در این است که MPI_Allreduce متغیر root را ندارد.

MPI_Reduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

MPI_Allreduce(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

حال به توضیح هریک از متغیرهای استفاده شده در این دو دستور، می‌پردازیم:

sendbuf: آدرس بافر ارسالی از هر پردازشگر که از نوع void می‌باشد و متغیر ثابت است.

recvbuf: آدرس بافر دریافتی که از نوع void می‌باشد.

count: متغیری از نوع صحیح که تعداد عناصر بافر ارسالی را نمایش می‌دهد.

datatype: متغیری از نوع MPI_Datatype که نوع داده بافر ارسالی را تعیین می‌کند.

op: عملیاتی که باید بر روی بافر دریافتی انجام شود. این عملیات‌ها از پیش توسط MPI تعریف شده که برخی از آن‌ها در جدول (۱-۷) آورده شده است.

جدول ۱-۷: انواع عملگر در MPI

انواع عملگر در MPI	عملیات
MPI_MAX	ماکزیمم عنصر را برمی‌گرداند.
MPI_MIN	مینیمم عنصر را برمی‌گرداند.
MPI_SUM	عناصر را جمع می‌کند.
MPI_PROD	عناصر را ضرب می‌کند.
MPI_SLAND	یک عملگر منطقی AND را انجام می‌دهد.
MPI_LOR	یک عملگر منطقی OR را انجام می‌دهد.
MPI_MAXLOC	ماکزیمم مقدار و رتبه پردازنده متعلق به آن را برمی‌گرداند.
MPI_MINLOC	مینیمم مقدار و رتبه پردازنده متعلق به آن را برمی‌گرداند.

root: متغیری از نوع صحیح که رتبه پردازشگر دریافت کننده بافر را مشخص می‌کند. (این متغیر در دستور MPI_Allreduce وجود ندارد.)

comm: گروه ارتباطی که از نوع MPI_Comm می‌باشد.

به مثال (۱-۷) توجه کنید که در آن دو دستور MPI_Reduce و MPI_Allreduce مورد استفاده قرار گرفته شده است. در خط شماره ۱۰ این برنامه رتبه هر پردازشگر با نوع MPI_INT از تمام پردازشگرهای گروه ارتباطی دریافت شده و عملیات ضرب بین آن‌ها انجام می‌شود. نتیجه حاصل در متغیر Prod_resault ذخیره شده و در دسترس

پردازشگر صفر خواهد بود. در خط ۱۱ برنامه، رتبه پردازشگرهای گروه ارتباطی دریافت می‌شود. در ادامه، این رتبه‌های دریافت شده توسط عملیات جمع با یکدیگر ترکیب می‌شوند. نتیجه به دست آمده در متغیر Sum_resault ذخیره می‌شود که در اختیار تمام پردازشگرهای گروه ارتباطی قرار دارد.

مثال ۱-۱۲-۸. استفاده از دستورات MPI_Reduce و MPI_Allreduce در محیط MPI

```
int main(int argc, char* argv[]){                                    ۱
  MPI_Init(&argc, &argv);                                            ۲
  int my_rank, world_size;                                           ۳
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);                           ۴
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);                        ۵
  int Prod_resault, Sum_resault = 0;                                 ۶
  if(my_rank == 0){                                                  ۷
    Prod_resault = 1;                                                ۸
  }                                                                  ۹
  MPI_Reduce(&my_rank,&Prod_resault,1,MPI_INT,MPI_PROD,0,            ۱۰
     MPI_COMM_WORLD);
  MPI_Allreduce(&my_rank,&Sum_resault, 1,MPI_INT,MPI_SUM,            ۱۱
     MPI_COMM_WORLD);
  if(my_rank == 0){                                                  ۱۲
    printf("(MPI Process %d) The product of all ranks is %d.\n", ۱۳
       my_rank,Prod_resault);
  }                                                                  ۱۴
  printf("(MPI Process %d) The sum of all ranks is %d.\n",           ۱۵
     my_rank,Sum_resault);
  MPI_Finalize();                                                    ۱۶
}                                                                    ۱۷
```

خروجی به ازای ۴ پردازشگر:

```
(MPI Process 0) The product of all ranks is 0.
(MPI Process 0) The sum of all ranks is 6.
(MPI Process 1) The sum of all ranks is 6.
(MPI Process 2) The sum of all ranks is 6.
(MPI Process 3) The sum of all ranks is 6.
```

۱-۱۲-۳ ارتباط بلوکی [۱] و غیربلوکی [۲]

نقل و انتقالات پیام در محیط MPI به دو صورت بلوکی و غیربلوکی قابل انجام می‌باشد. در ارتباط بلوکی (مسدود) فرستنده پیام تا زمانی که عملیات ارسال به اتمام نرسیده باشد، مسدود می‌شود. به عبارت دیگر تنها زمانی بافر منتقل شده قابل استفاده مجدد است، که MPI آن را ذخیره کرده باشد و یا اینکه در مقصد پیام دریافت شده باشد. به طور مشابه در سمت گیرنده پیام نیز همین اتفاق می‌افتد یعنی گیرنده پیام مسدود می‌شود، تا زمانی که بافر دریافتی با داده‌های معتبر پر شود.

[1] blocking [2] nonblocking

از طرف دیگر در ارتباط غیربلوکی (بدون انسداد) پردازنده‌هایی که در فرآیند ارتباطی مشارکت دارند، بدون اینکه مسدود شوند به ادامه اجرا می‌پردازند حتی اگر انتقال پیام هنوز تمام نشده باشد. برای اطلاع از وضعیت پیام باید از دستورات MPI_Wait یا MPI_Test استفاده کنیم که در ادامه به توضیح هریک از آن‌ها می‌پردازیم.

- **دستور MPI_Wait:** هنگامی که در برنامه MPI از دستورات غیربلوکی استفاده شود، این دستور به کار گرفته می‌شود. به این مفهوم که اگر پردازشگری درخواستی را ارسال کرده باشد و به دستور MPI_Wait برسد باید تا زمانی که درخواستش کامل نشده، منتظر بماند(مسدود شود). شکل کلی این دستور به صورت زیر است:

MPI_Wait(MPI_Request *request, MPI_Status *status)

هریک از پارامترهای استفاده شده در آن به صورت زیر می‌باشد:

request: متغیری از نوع MPI_Request می‌باشد و نشان دهنده درخواستی است که در یک دستور غیربلوکی استفاده شده و باید منتظر آن بمانیم.

status: متغیری از نوع MPI_Status می‌باشد و وضعیت دستور غیربلوکی در آن ذخیره می‌شود. ساختار کلی و جزئیات آن در بخش (۱-۱۲-۲) ذکر شده است. اگر وضعیت برایمان مهم نباشد می‌توانیم از دستور MPI_STATUS_IGNORE استفاده کنیم.

- **دستور MPI_Test:** این دستور نیز مشابه دستور MPI_Wait در برنامه‌هایی که از دستورات غیر بلوکی استفاده می‌کنند، به کار می‌رود. دستور MPI_Test برخلاف دستور MPI_Wait منتظر نمی‌ماند تا دستور غیربلوکی انجام شود. بلکه بررسی می‌کند که آیا یک دستور غیربلوکی در یک زمان مشخص کامل شده است یا خیر. شکل کلی آن به صورت زیر می‌باشد:

MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

حال به توضیح هریک از پارامترهای استفاده شده در آن می‌پردازیم:

request: متغیری از نوع MPI_Request است که نشان دهنده درخواستی می‌باشد که در یک دستور غیربلوکی استفاده شده و باید مورد بررسی قرار بگیرد.

flag: متغیری از نوع منطقی که نتیجه بررسی در آن ذخیره می‌شود. اگر دستور غیربلوکی به درستی کامل شده باشد، مقدار آن true و در غیر اینصورت مقدار آن false خواهد بود.

status: متغیری از نوع MPI_Status می‌باشد و وضعیت دستور غیربلوکی در آن ذخیره می‌شود. اگر وضعیت برایمان مهم نباشد می‌توانیم از دستور MPI_STATUS_IGNORE استفاده کنیم.

هریک از دستورات بلوکی MPI دارای یک دستور معادل غیربلوکی می‌باشد که برخی از آن‌ها در جدول (۱-۸) آورده شده است:

جدول ۱-۸: دستورات بلوکی MPI و معادل غیربلوکی آن‌ها

دستورات بلوکی در MPI	دستورات معادل غیربلوکی
MPI_Send	MPI_Isend
MPI_Recv	MPI_Irecv
MPI_Scatter	MPI_Iscatter
MPI_Gather	MPI_Igather
MPI_Allgather	MPI_Iallgather
MPI_Reduce	MPI_Ireduce
MPI_Allreduce	MPI_Iallreduce
MPI_Bcast	MPI_Ibcast

در ادامه ابتدا پارامترهای دو دستور غیربلوکی MPI_Isend و MPI_Irecv را بررسی می‌کنیم. سپس برای اطلاع بیشتر از نحوه استفاده دستورات غیربلوکی دو مثال را آورده‌ایم که در مثال (۱-۱۲-۹) از دو دستور MPI_Isend و MPI_Irecv همراه با MPI_Wait استفاده شده است. و در مثال (۱-۱۲-۱۰) دو دستور MPI_Isend و MPI_Irecv همراه با MPI_Test عنوان گردیده است.

MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request *request)

MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request *request)

پارامترهای به کار رفته در این دو دستور دقیقا مشابه پارامترهای موجود در دستورات MPI_Send و MPI_Recv می‌باشد که در بخش (۱-۱۲-۲) به طور مفصل توضیح داده شده است. اما در این دو دستور پارامتر اضافه request وجود دارد که از نوع MPI_Request می‌باشد و نشان دهنده درخواستی است که در دستور غیربلوکی استفاده شده و باید وضعیت آن مورد بررسی قرار بگیرد.

مثال ۱-۱۲-۹. استفاده از دستورات MPI_Isend و MPI_Irecv با استفاده از MPI_Wait (این برنامه باید با دو پردازشگر اجرا شود)

```
//This application is meant to be run with 2 processes                    ۱
int main(int argc, char* argv[]){                                         ۲
  MPI_Init(&argc, &argv);                                                 ۳
  int my_rank, world_size;                                                ۴
  MPI_Status status;                                                      ۵
  MPI_Request request;                                                    ۶
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);                                ۷
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);                             ۸
  int buffer_send, received;                                              ۹
  if (my_rank == 0) {                                                     ۱۰
        buffer_send = 12345;                                              ۱۱
```

```
        MPI_Isend(&buffer_send,1,MPI_INT,1,100,MPI_COMM_WORLD,
            &request);
  }
  else if (my_rank == 1) {
        MPI_Irecv(&received,1,MPI_INT,0,100,MPI_COMM_WORLD,&request);
  }
  MPI_Wait(&request,&status);
  printf("Processor %d is waiting...\n",my_rank);
  if (my_rank == 0) {
        printf("Processor %d sent %d\n",my_rank,buffer_send);
  }
  else if (my_rank == 1) {
        printf("Processor %d received %d\n", my_rank,received);
  }
  MPI_Finalize();
}
```

خروجی اجرای موفقیت آمیز برنامه به ازای ۲ پردازشگر به صورت زیر خواهد بود:

```
Processor 0 is waiting ...
Process 0 send 12345
Processor 1 is waiting ...
Process 1 received 12345
```

مثال ۱-۱۲-۱۰. استفاده از دستورات MPI_Isend و MPI_Irecv با استفاده از MPI_Test (این برنامه باید با دو پردازشگر اجرا شود)

```
//This application is meant to be run with 2 processes
int main(int argc, char* argv[]){
  MPI_Init(&argc, &argv);
  int my_rank, world_size;
  MPI_Status status;
  MPI_Request request, request2;
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);
  int flag,test=0, buffer_send, received;
  if (my_rank == 0) {
        buffer_send = 12345;
        MPI_Isend(&buffer_send,1,MPI_INT,1,100,MPI_COMM_WORLD,
            &request);
  }
  else if (my_rank == 1) {
        MPI_Irecv(&received,1,MPI_INT,0,100,MPI_COMM_WORLD,&request);
  }
  MPI_Test(&request, &flag, &status);
  while (!flag && test<1e5){
        test ++;
        if(test == 1){
     printf("Processor %d is testing...\n",my_rank);
        }
    MPI_Test(&request, &flag, &status);
  }
  if (my_rank == 0) {
        printf("Processor %d sent %d\n",my_rank,buffer_send);
  }
```

```
  else if (my_rank == 1) {                                          ۲۸
        printf("Processor %d received %d\n", my_rank,received);     ۲۹
  }                                                                 ۳۰
  MPI_Finalize();                                                   ۳۱
}                                                                   ۳۲
```

اجرای موفقیت آمیز برنامه به ازای ۲ پردازشگر به صورت زیر می‌باشد:

```
Processor 0 sent 12345
Processor 1 is testing ...
Processor 1 received 12345
```

فصل ۲

جزئیات پیاده‌سازی الگوریتم گرادیان مزدوج در محیط OpenMP

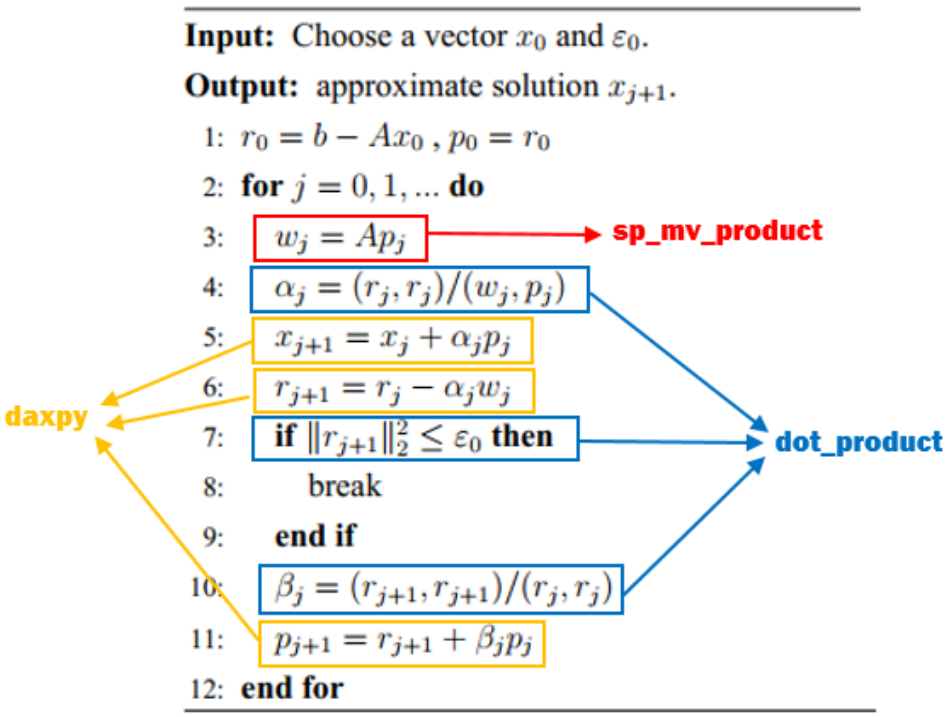
۲-۱ مقدمه

در بخش (۱-۶-۱) این پایان‌نامه، پیرامون الگوریتم گرادیان مزدوج و نحوه عملکرد آن بحث نمودیم. اولین مرحله برای موازی‌سازی این الگوریتم شناسایی عملیات‌های اصلی در آن می‌باشد. برای این منظور به شکل (۲-۱) توجه کنید که در آن الگوریتم گرادیان مزدوج به همراه بخش‌هایی از این الگوریتم که خاصیت موازی دارند، در تصویر آورده شده است. همچنین زیربرنامه‌های موازی که در محیط OpenMP برای پیاده‌سازی موازی این خطوط استفاده شده، مشخص گردیده است. در ادامه به این زیربرنامه‌ها اشاره می‌کنیم.

در خط ۳ این الگوریتم از زیربرنامه موازی به نام sp_mv_product استفاده شده است که حاصلضرب ماتریس تنک در بردار را محاسبه می‌کند و در آن ماتریس به صورت فشرده سطری ذخیره شده است. جزئیات پیاده‌سازی این زیربرنامه را در ادامه این فصل به طور مفصل شرح خواهیم داد.

در خطوط ۴، ۷ و ۱۰ الگوریتم، زیربرنامه موازی دیگری به نام dot_product مورد استفاده قرار گرفته که ضرب داخلی دو بردار را در محیط OpenMP محاسبه می‌کند. در ادامه این فصل جزئیات بیشتری از این زیربرنامه را بیان خواهیم نمود.

در انتها در خطوط ۵، ۶ و ۱۱ این الگوریتم نیز از زیربرنامه daxpy استفاده شده که دو بردار را به صورت موازی در محیط OpenMP جمع می‌کند. جزئیات این زیربرنامه در ادامه شرح داده خواهد شد.



شکل ۲-۱: الگوریتم گرادیان مزدوج و بخش‌ها و زیربرنامه‌های موازی آن در محیط OPenMP

۲-۲ زیربرنامه‌های موازی استفاده شده در الگوریتم گرادیان مزدوج

- **ضرب ماتریس در بردار:** در خط شماره ۳ الگوریتم شکل (۲-۱) برای محاسبه تکرار بعدی از جواب دستگاه، نیاز به محاسبه ضرب ماتریس در بردار خواهیم داشت. برای محاسبه این ضرب ماتریس در بردار زیربرنامه موازی sp_mv_product مورداستفاده قرار گرفته است. ماتریس ضرایب A، یک ماتریس تنک می‌باشد که با فرمت فشرده سطری ذخیره شده است. همانطور که در فصل قبل توضیح داده شد، فرمت ذخیره‌سازی فشرده سطری شامل ۳ آرایه می‌باشد. فرض کنید NNz تعداد کل عناصر غیرصفر موجود در ماتریس A باشد:

- **آرایه حقیقی AA:** یک آرایه با طول NNz می‌باشد که برای ذخیره‌سازی عناصر غیر صفر ماتریس مورد استفاده قرار گرفته می‌شود.
- **آرایه صحیح JA:** یک آرایه صحیح با طول NNz می‌باشد که برای ذخیره کردن اندیس ستون عناصر غیر صفر ماتریس استفاده می‌شود.
- **آرایه صحیح IA** آرایه با طول n+1 یک آرایه اشاره‌گر است که عنصر i ام آن به موقعیت شروع عناصر غیر صفر و اندیس ستونی سطر i ام ماتریس اشاره می‌کند.

قلب و هسته مرکزی زیربرنامه sp_mv_product یک برنامه سری می‌باشد که در الگوریتم (۲-۱) مطرح شده است. این الگوریتم حاصلضرب $y = Ax$ را به صورت سری محاسبه می‌کند. برای فهم راحت‌تر الگوریتم (۲-۱) ابتدا به شرح یک گام از آن می‌پردازیم.

اندیس i را در نظر بگیرید که اشاره به سطر شماره i در ماتریس A دارد. در خط ۲ و ۳ این الگوریتم K1 اشاره‌گری است که به اولین عنصر غیرصفر سطر i ام ماتریس A در آرایه AA اشاره می‌کند. و K2 اشاره به آخرین عنصر غیرصفر سطر i ام ماتریس A در آرایه AA دارد. خطوط ۴ تا ۶ این الگوریتم یک حلقه داخلی است که در آن عناصر غیرصفر سطر i ام ماتریس A که در $AA(k۱), AA(k۱+۱), ..., AA(k۲)$ ذخیره شده‌اند در مولفه‌های $x(JA(k۱)), x(JA(k۱+۱)), ..., x(JA(k۲))$ به صورت مولفه در مولفه ضرب می‌شوند و در نهایت حاصل این مقادیر محاسبه شده درون درایه i ام بردار y قرار می‌گیرند.

الگوریتم ۲-۱ ضرب ماتریس تنک با فرمت CSR در یک بردار

۴

Input: sparse matrix A in CSR format and a vector x.
Output: vector y.

1: **for** $i = 0$ to n **do**
2: $K1 = IA[i]$
3: $K2 = IA[i+1] - 1$
4: **for** $j = K1$ to $K2$ **do**
5: $y[i] = dotproduct(AA[j], x[JA[j]])$
6: **end for**
7: **end for**

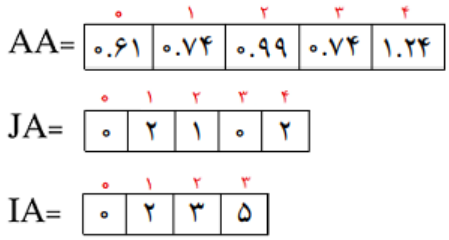
برای درک بیشتر این الگوریتم در ادامه یک مثال عددی را بیان می‌کنیم. در این مثال حاصلضرب ماتریس A در بردار x با استفاده از الگوریتم فوق محاسبه خواهد شد.

مثال ۲-۲-۱.

$$y = Ax = \begin{bmatrix} ۰/۶۱ & ۰ & ۰/۷۴ \\ ۰ & ۰/۹۹ & ۰ \\ ۰/۷۴ & ۰ & ۱/۲۴ \end{bmatrix} \begin{bmatrix} ۱ \\ ۱ \\ ۱ \end{bmatrix}$$

حل: فرمت فشرده سطری ماتریس A در ادامه آورده شده است. از آنجایی که پیاده‌سازی الگوریتم در زبان C++ می‌باشد، اندیس آرایه‌ها از صفر شروع می‌شوند. گام‌های حلقه الگوریتم (۲-۱) به صورت سری بر روی آرایه‌های فشرده سطری ماتریس و بردار تمام یک اعمال شده و نتیجه نهایی حاصلضرب در مولفه‌های

بـردار y قرار می‌گیرند.



هر گام از حلقه خارجی الگوریتم، حاصلضرب یک سطر ماتریس در بردار را محاسبه می‌کند.

گام اول: $(i = ۰)$

$K۱ = IA[۰] = ۰$

$K۲ = IA[۱] - ۱ = ۱$

$j = ۰ \rightarrow y[۰]^{(۰)} = AA[۰] * x[JA[۰]] = ۰/۶۱ * x[۰] = ۰/۶۱$

$j = ۱ \rightarrow y[۰]^{(۲)} = AA[۱] * x[JA[۱]] = ۰/۷۴ * x[۲] = ۰/۷۴$

مقدار $y[۰]$ به ازای مقادیر غیرصفر سطر شماره صفر ماتریس محاسبه شده و در انتها این مقادیر با یکدیگر جمع شده و مولفه اول بردار y را تشکیل خواهند داد.

$y[۰] = y[۰]^{(۰)} + y[۰]^{(۲)} = ۰/۶۱ + ۰/۷۴ = ۱/۳۵$

گام دوم: $(i = ۱)$

$K۱ = IA[۱] = ۲$

$K۲ = IA[۲] - ۱ = ۲$

$j = ۲ \rightarrow y[۱]^{(۱)} = AA[۲] * x[JA[۲]] = ۰/۹۹ * x[۱] = ۰/۹۹$

مقدار $y[۱]$ به ازای مقادیر غیرصفر سطر شماره یک ماتریس محاسبه می‌شود. از آنجایی که سطر دوم از ماتریس فقط دارای یک مقدار غیر صفر می‌باشد، پس مولفه شماره یک بردار y برابر با این مقدار محاسبه شده است.

$y[۱] = y[۱]^{(۱)} = ۰/۹۹$

گام سوم: $(i = ۲)$

$$K۱ = IA[۲] = ۳$$

$$K۲ = IA[۳] - ۱ = ۴$$

$$j = ۳ \rightarrow y[۲]^{(۰)} = AA[۳] * x[JA[۳]] = ۰/۷۴ * x[۰] = ۰/۷۴$$

$$j = ۴ \rightarrow y[۲]^{(۲)} = AA[۴] * x[JA[۴]] = ۱/۲۴ * x[۲] = ۱/۲۴$$

مقدار $y[۲]$ به ازای مقادیر غیرصفر سطر آخر ماتریس محاسبه شده و در انتها این مقادیر با یکدیگر جمع شده و مولفه آخر بردار y را تشکیل خواهند داد.

$$y[۲] = y[۲]^{(۰)} + y[۲]^{(۲)} = ۰/۷۴ + ۱/۲۴ = ۱/۹۸$$

برای پیاده‌سازی موازی این الگوریتم، حلقه خارجی آن را به صورت موازی درمی‌آوریم. برای این منظور از بندهای حلقه for موازی در محیط OPenMP استفاده می‌کنیم تا بتوانیم حلقه خارجی را موازی کنیم. ضرب سطرهای ماتریس A در بردار x به صورت مستقل از یکدیگر می‌باشند. پس بدون تداخل با یکدیگر می‌توانند به صورت موازی در یکدیگر ضرب شوند و در نهایت تمامی این حاصلضرب‌های محاسبه شده، در درایه‌های بردار y قرار می‌گیرند [۸]. این کار اصلی است که در زیربرنامه sp_mv_product اتفاق می‌افتد. حال به توضیح این زیربرنامه می‌پردازیم. زیربرنامه‌ای که در ادامه اورده شده است، طرح و نمای کلی الگوریتم sp_mv_product در محیط C++ می‌باشد.

برنامه ۲-۱: ضرب ماتریس تنک با فرمت CSR در یک بردار در محیط OpenMP

```
void sp_mv_product(double *y, double *AA, int *JA, int *IA,      ۱
    double *x, int n) {
                                                                  ۲
//! Compute the sparse matrix-vector product (CSR format)        ۳
int i,j;                                                          ۴
#ifdef OPENMP                                                     ۵
#pragma omp parallel for default(none) \                          ۶
private(i,j) shared(y,AA,JA,IA,x,n)                               ۷
#endif                                                            ۸
  for (i = 0; i < n; i++) {                                       ۹
    y[i] = 0.0;                                                   ۱۰
    for (j = IA[i]; j < IA[i+1]; j++) {                           ۱۱
      y[i] += AA[j]*x[JA[j]];                                     ۱۲
    }                                                             ۱۳
  }                                                               ۱۴
}                                                                 ۱۵
```

در خط ۶ و ۷ این زیربرنامه از حلقه for موازی استفاده شده که در این حلقه بند default(none) مورد استفاده قرار گرفته شده است. همان گونه که در فصل قبل در مورد این بند توضیح داده شد، باید تکلیف تمامی متغیرهای تابع از نظر اشتراکی یا خصوصی بودن مشخص گردد. حلقه خارجی در خطوط ۹ تا ۱۴ این الگوریتم به صورت موازی اجرا خواهد شد. که در آن هر سطر از ماتریس A در بردار x ضرب شده و مولفه‌های بردار y محاسبه خواهد

شد. در این حلقه موازی از هیچ بند schedule استفاده نشده است. لذا نحوه تقسیم‌بندی گام‌های حلقه به صورت static و تعداد chunk-size آن به صورت تعداد گام‌های حلقه تقسیم بر تعداد رشته‌های اجرا کننده حلقه در نظر گرفته خواهد شد.

- **ضرب داخلی دو بردار:** در خطوط شماره ۴ و ۷ و ۱۰ الگوریتم (۲-۱) به محاسبه ضرب داخلی دو بردار نیاز داریم. برای پیاده‌سازی موازی این خطوط از زیربرنامه dot_product استفاده شده است.

برنامه ۲-۲: انجام عملیات ضرب داخلی در محیط OpenMP

```
double dot_product(double *x, double *y, int n) {                    ۱
                                                                      ۲
//! Compute the dot product of the provided vectors                   ۳
double result = 0.0; int i;                                           ۴
                                                                      ۵
#ifdef OPENMP                                                         ۶
#pragma omp parallel for default(none) \                              ۷
private(i) shared(x,y,n) ordered reduction(+:result)                  ۸
#endif                                                                ۹
  for (i = 0; i < n; i++) {                                           ۱۰
    result += x[i]*y[i];                                              ۱۱
  }                                                                   ۱۲
  return result;                                                      ۱۳
}                                                                     ۱۴
```

در خط ۷ این زیربرنامه از بند default(none) استفاده شده است. بنابراین باید اشتراکی یا خصوصی بودن تمامی متغیرهای موجود مشخص شود. در خط شماره ۸ الگوریتم از بند ordered استفاده شده به این معنا که ترتیب اجرای عملیات درون حلقه مشابه با اجرای سری خواهد بود یعنی رشته‌های اجرا کننده حلقه قبل از اجرای گام های مربوط به خود باید منتظر بمانند تا گام های قبلی از حلقه اجرا شوند. برای جزئیات بیشتر در مورد بند ordered به مثال (۱-۱۰-۱۴) فصل ۱ مراجعه کنید. از آنجایی که در این زیربرنامه از هیچ بند schedule استفاده نشده است. پس تعداد chunk-sizeها برابر با تعداد گام‌های حلقه تقسیم بر تعداد رشته‌ها درنظر گرفته شده و در اختیار این رشته‌ها قرار گرفته می‌شوند. با توجه به اینکه برای انجام عملیات ضرب داخلی از بند reduction استفاده شده است. هر کدام از این رشته‌ها یک نسخه محلی از result مربوط به خود را دریافت خواهند کرد. از آنجا که بند reduction همراه با عملگر جمع استفاده شده است، هر رشته result محلی خود را با متغیر کلی جمع می‌کنند بدون اینکه تداخلی با یکدیگر داشته باشند. و در نهایت مقدار result محاسبه می‌شود.

- **به‌روزرسانی برداری:** در خطوط شماره ۵ و ۶ و ۱۱ الگوریتم (۲-۱) عملیاتی به شکل $z = x + \alpha y$ محاسبه می‌شود. که در آن x و y یک آرایه و α یک اسکالر می‌باشد که x و y از یک بعد هستند. زیربرنامه موازی که برای انجام این عملیات مورد استفاده قرار می‌گیرد، در ادامه عنوان شده است.

برنامه ۲-۳: انجام عملیات به‌روزرسانی برداری در محیط OpenMP

```
void daxpy(double *z, double alpha, double *x, double *y,          ۱
    int n) {
                                                                   ۲
//! Compute the addition of a vector with a vector times a         ۳
    constant
int i;                                                             ۴
#ifdef OPENMP                                                      ۵
#pragma omp parallel for default(none) \                           ۶
private(i) shared(z,alpha,x,y,n)                                   ۷
#endif                                                             ۸
  for (int i = 0; i < n; i++) {                                    ۹
    z[i] = alpha*x[i] + y[i];                                      ۱۰
  }                                                                ۱۱
}                                                                  ۱۲
```

در خطوط ۹ تا ۱۱ این زیربرنامه یک حلقه ساده داریم که این حلقه باید به صورت موازی اجرا شود. تنظیمات موازی این حلقه در خطوط ۶ و۷ انجام شده است. این حلقه هیچ بند schedule ندارد. به این مفهوم که تعداد chunk-sizeهایی که در اختیار هر رشته قرار می‌گیرند، برابر با تعداد تکرارهای حلقه تقسیم بر تعداد کل رشته‌ها است. متغیرهایی که به صورت اشتراکی تعریف شده‌اند و در اختیار تمام رشته‌ها قرار می‌گیرند، بردار z، پارامتر α و بردار x و y و عدد n که بعد آرایه می‌باشد. و متغیر i که گام‌های حلقه می‌باشد، به صورت خصوصی تعریف شده است.

فصل ۳

جزئیات پیاده‌سازی الگوریتم گرادیان مزدوج در محیط MPI

۳-۱ مقدمه

همانطور که در فصل یک اشاره شد، یک برنامه در محیط برنامه‌نویسی MPI توسط مجموعه‌ای از پردازشگرها اجرا می‌شود که در آن هر پردازشگر داده‌های محلی خودش را دارد و می‌تواند با ارسال و دریافت پیام، به تبادل داده با دیگر پردازشگرها بپردازد. قبل از شروع برنامه، ابتدا باید داده‌های مربوط به هر پردازشگر مشخص گردد. در ادامه این فصل به نحوه توزیع ماتریس و بردار بین پردازشگرها خواهیم پرداخت. سپس زیربرنامه‌های استفاده شده برای پیاده‌سازی موازی الگوریتم گرادیان مزدوج را در محیط MPI بررسی نموده و جزئیات آن‌ها شرح خواهیم داد.

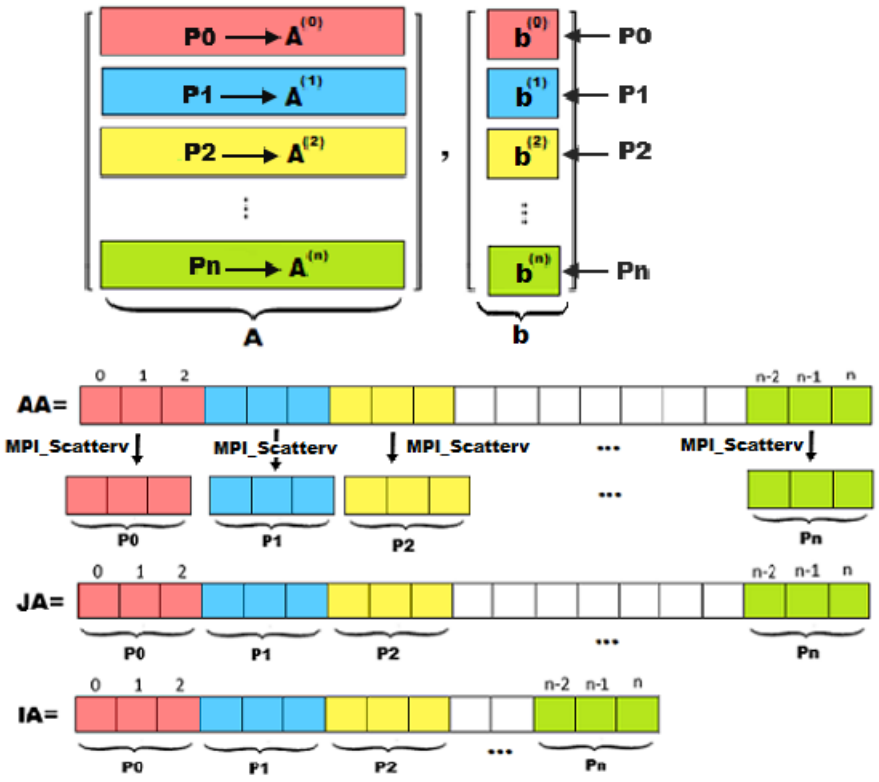
در این فصل نماد Pi به معنی Processor شماره i می‌باشد. همچنین در برخی قسمت‌ها در متن، از مفهوم Processor i برای اشاره به پردازشگر شماره i استفاده شده است.

۳-۲ نحوه توزیع ماتریس و بردار بین پردازشگرها

نکته اصلی برای اجرای یک برنامه به صورت موازی در محیط برنامه‌نویسی توزیعی، نحوه تقسیم‌بندی داده‌ها بین پردازشگرهای شرکت‌کننده در اجرای برنامه می‌باشد. در الگوریتم گرادیان مزدوج، این داده‌ها شامل ماتریس ضرایب و بردار سمت راست در دستگاه خطی $Ax = b$ هستند. با توجه به اینکه ماتریس ضرایب یک ماتریس تنک می‌باشد که به فرمت فشرده سطری ذخیره شده است، بنابراین به جای ماتریس ضرایب A سه آرایه AA، JA و IA بین پرازشگرها توزیع خواهد شد. در این پایان‌نامه از دستور MPI_scatterv برای توزیع بردارها بین پردازشگرها و دریافت سهم هر پردازشگر

استفاده شده است.

در شکل (۳-۱) طرح کلی توزیع ماتریس A و بردار b در بین پردازشگرهای برنامه نمایش داده می‌شود. با توجه به تصویر، $A^{(i)}$ و $b^{(i)}$ بخش‌هایی از ماتریس A و بردار b می‌باشند که در اختیار پردازشگر شماره i قرار گرفته‌اند.

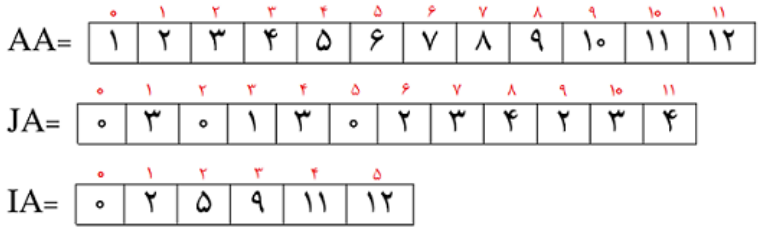


شکل ۳-۱: توزیع ماتریس و بردار بین پردازشگرها در محیط MPI

در ادامه سه زیربرنامه آورده شده که نحوه توزیع آرایه‌های ماتریس و بردار b را در محیط MPI نمایش خواهند داد. ابتدا عنوان هر زیربرنامه را شرح داده و به توضیح جزئیات پیاده‌سازی آن‌ها خواهیم پرداخت. فرض کنیم که ماتریس ضرایب و بردار سمت راست دستگاه را به صورت زیر داریم:

$$A = \begin{bmatrix} ۱ & ۰ & ۰ & ۲ & ۰ \\ ۳ & ۴ & ۰ & ۵ & ۰ \\ ۶ & ۰ & ۷ & ۸ & ۹ \\ ۰ & ۰ & ۱۰ & ۱۱ & ۰ \\ ۰ & ۰ & ۰ & ۰ & ۱۲ \end{bmatrix}, b = \begin{bmatrix} ۳ \\ ۱۲ \\ ۳۰ \\ ۲۱ \\ ۱۲ \end{bmatrix} \quad (۱-۳)$$

شکل فشرده سطری ماتریس فوق به صورت زیر تعریف می‌شود:



می‌خواهیم شکل فشرده سطری ماتریس و بردار b را در بین ۳ پردازشگر توزیع کنیم. برای این منظور ابتدا زیربرنامه (۳-۱) را در نظر بگیرید که در آن ۴ آرایه rowCounts، rowDispls، nnzCounts و nnzDispls مقداردهی خواهند شد. از این ۴ آرایه، دو آرایه rowCounts و rowDispls برای توزیع آرایه‌های IA و بردار سمت راست b استفاده می‌شود و همچنین دو آرایه nnzCounts و nnzDispls برای توزیع آرایه‌های AA و JA مورد استفاده قرار خواهند گرفت.

برنامه ۳-۱: ساخت آرایه‌های موردنیاز برای توزیع ماتریس و بردار سمت راست

```
MPI_Comm_rank(MPI_COMM_WORLD, &myid);                                    ۱
MPI_Comm_size(MPI_COMM_WORLD, &numprocs);                                ۲
                                                                         ۳
count = M / numprocs;                                                    ۴
remainder = M - count * numprocs;                                        ۵
                                                                         ۶
if(myid == 0){                                                           ۷
  rowCounts = new int[numprocs];                                         ۸
  rowDispls = new int[numprocs];                                         ۹
  nnzCounts = new int[numprocs];                                         ۱۰
  nnzDispls = new int[numprocs];                                         ۱۱
                                                                         ۱۲
  int prefixSum = 0;                                                     ۱۳
  for (int i = 0; i < numprocs; ++i) {                                   ۱۴
    int t1 = (i < remainder) ? count + 1 : count;                        ۱۵
    rowCounts[i] = t1;                                                   ۱۶
    int t2 = prefixSum;                                                  ۱۷
    rowDispls[i] = t2;                                                   ۱۸
    prefixSum += t1;                                                     ۱۹
    nnzCounts[i] = IA[t2+t1] - IA[t2];                                   ۲۰
    nnzDispls[i] = IA[t2];                                               ۲۱
  }                                                                      ۲۲
}                                                                        ۲۳
```

حال به توضیح جزئیات این زیربرنامه خواهیم پرداخت. در خط شماره ۱ این برنامه، هر پردازشگر رتبه خود را با استفاده از دستور MPI_Comm_rank دریافت نموده و در متغیر myid ذخیره می‌کند. و همچنین در خط ۲ این برنامه، تعداد کل پردازشگرهای اجراکننده برنامه با استفاده از دستور MPI_Comm_size درون متغیر numprocs دریافت خواهد شد. جزئیات بیشتر در مورد این دستورات در بخش (۱-۱۲-۱) فصل یک ذکر شده است.

خطوط ۷ تا ۲۲ از این برنامه توسط پردازشگری که مقدار myid آن برابر با صفر است اجرا خواهد شد. اجرای این برنامه در مورد ماتریس (۳-۱) که بعد آن برابر با M =۵ است، به صورت زیر می‌باشد:

count = ۵/۳ = ۱ ، remainder = ۵-(۳*۱) =۲

گام اول: $(i = ۰)$

prefixSum = ۰

t1 = (۰ < remainder) → t1 = ۱ + ۱ = ۲

rowCounts[۰] = ۲

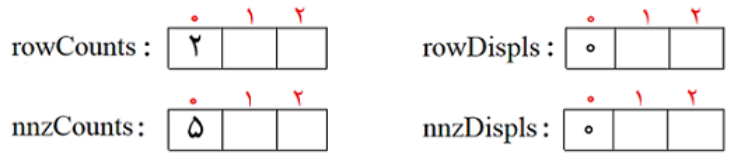
t2 = ۰

rowDispls[۰] = ۰

prefixSum = ۰ + ۲ = ۲

nnzCounts[۰] = IA[۰+۲] – IA[۰] = ۵

nnzDispls[۰] = IA[۰] = ۰



گام دوم: $(i = ۱)$

t1 = (۱ < remainder) → t1 =۱ + ۱ = ۲

rowCounts[۱] = ۲

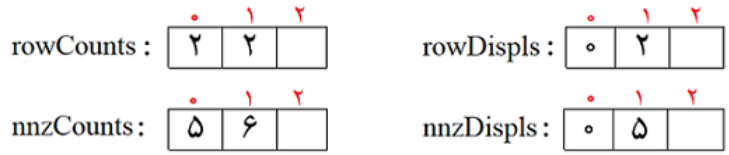
t2 = ۲

rowDispls[۱] = ۲

prefixSum = ۲ + ۲ = ۴

nnzCounts[۱] = IA[۲+۲] – IA[۲] = ۶

nnzDispls[۱] = IA[۲] = ۵



گام سوم: $(i = ۲)$

t1 = (۲ < remainder) → t1 =۱

rowCounts[۲] = ۱

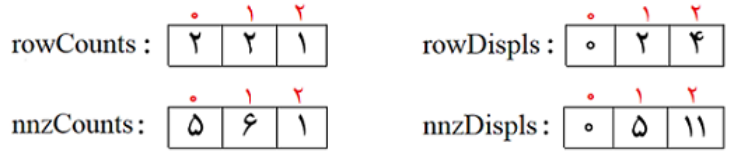
t2 = ۴

rowDispls[۲] = ۴

prefixSum = ۴ + ۱ = ۵

nnzCounts[۲] = IA[۴+۱] − IA[۴] = ۱

nnzDispls[۲] = IA[۴] = ۱۱



آرایه‌های rowCounts و rowDispls نشان دهنده این است که کدام درایه‌ها از ماتریس A و بردار b و چه تعداد از درایه‌های ماتریس A و بردار b در اختیار پردازشگرها قرار می‌گیرد.

- ازسطر شماره صفر (rowDispls[0]) به اندازه ۲ سطر (rowCounts[0]) از ماتریس A و بردار b در اختیار پردازشگر شماره صفر قرار خواهد گرفت.
- ازسطر شماره ۲ (rowDispls[1]) به اندازه ۲ سطر (rowCounts[1]) از ماتریس A و بردار b در اختیار پردازشگر شماره یک قرار خواهد گرفت.
- ازسطر شماره ۴ (rowDispls[2]) به اندازه ۱ سطر (rowCounts[2]) از ماتریس A و بردار b در اختیار پردازشگر شماره دو قرار خواهد گرفت.

به طور مشابه آرایه‌های nnzCounts و nnzDispls نشان دهنده این می‌باشند که کدام درایه‌ها از آرایه‌های AA و JA و چه تعداد از درایه‌های این آرایه‌ها در اختیار پردازشگرها قرار می‌گیرد.

- برای پردازشگر شماره صفر، از خانه شماره صفر (nnzDispls[0]) آرایه‌های AA و JA به تعداد ۵ عنصر (nnzCounts[0]) در اختیار پردازشگر شماره صفر قرار می‌گیرد که شامل تمام عناصر غیرصفر موجود در دو سطر اول ماتریس می‌باشند.
- از خانه شماره ۵ (nnzDispls[1]) آرایه‌های AA و JA به تعداد ۶ عنصر (nnzCounts[1]) در اختیار پردازشگر شماره یک قرار می‌گیرد که شامل تمام عناصر غیرصفر موجود در دو سطر شماره ۲ و ۳ ماتریس می‌باشند.

- از خانه شماره ۱۱ (nnzDispls[2]) آرایه‌های AA و JA به تعداد ۱ عنصر (nnzCounts[2]) در اختیار پردازشگر شماره دو قرار می‌گیرد که شامل تمام عناصر غیرصفر موجود در آخرین سطر ماتریس می‌باشند.

حال از آرایه‌های rowCounts و rowDispls استفاده می‌کنیم تا بردار IA و b را بین پردازشگرها توزیع کنیم. زیربرنامه‌ای که در ادامه آورده شده است این توزیع را نشان می‌دهد. خطوط ۸ تا ۱۷ برنامه برای توزیع بردار b، خطوط ۱۹ تا ۲۷ برنامه برای پخش بردار nnzCounts و خطوط ۳۰ تا ۳۹ برنامه برای پخش بردار IA به کار می‌روند. حال به توضیح هریک از آن‌ها می‌پردازیم.

برنامه ۳-۲: توزیع بردار b و IA با استفاده از دو آرایه rowCounts و rowDispls

```
// own size                                                          ۱
myRowsSize = myid < remainder ? count + 1 : count;                   ۲
                                                                     ۳
double *b_Part = new double[myRowsSize];                             ۴
int *my_ia = new int[myRowsSize+1];                                  ۵
                                                                     ۶
//! Scatter vector entries to each processor                         ۷
MPI_Scatterv(                                                        ۸
  b,                                                                 ۹
  rowCounts, //t1                                                    ۱۰
  rowDispls, //t2                                                    ۱۱
  MPI_DOUBLE,                                                        ۱۲
  b_Part,                                                            ۱۳
  myRowsSize,                                                        ۱۴
  MPI_DOUBLE,                                                        ۱۵
  0, //root rank                                                     ۱۶
  MPI_COMM_WORLD);                                                   ۱۷
                                                                     ۱۸
MPI_Scatter(                                                         ۱۹
  nnzCounts,                                                         ۲۰
  1,                                                                 ۲۱
  MPI_INT,                                                           ۲۲
  &nnz,                                                              ۲۳
  1,                                                                 ۲۴
  MPI_INT,                                                           ۲۵
  0, //root rank                                                     ۲۶
  MPI_COMM_WORLD);                                                   ۲۷
                                                                     ۲۸
//! Scatter partial Row pointers to each processor                   ۲۹
MPI_Scatterv(                                                        ۳۰
  IA,                                                                ۳۱
  rowCounts,                                                         ۳۲
  rowDispls,                                                         ۳۳
  MPI_INT,                                                           ۳۴
  my_ia,                                                             ۳۵
  myRowsSize,                                                        ۳۶
  MPI_INT,                                                           ۳۷
  0, //root rank                                                     ۳۸
  MPI_COMM_WORLD);                                                   ۳۹
                                                                     ۴۰
//! Update Row pointers in each processor                            ۴۱
int Offset = my_ia[0];                                               ۴۲
```

```
for (int i = 0; i < myRowsSize; i++){                                    ۴۳
  my_ia[i] = my_ia[i] - Offset;                                          ۴۴
}                                                                        ۴۵
my_ia[myRowsSize] = nnz;                                                 ۴۶
```

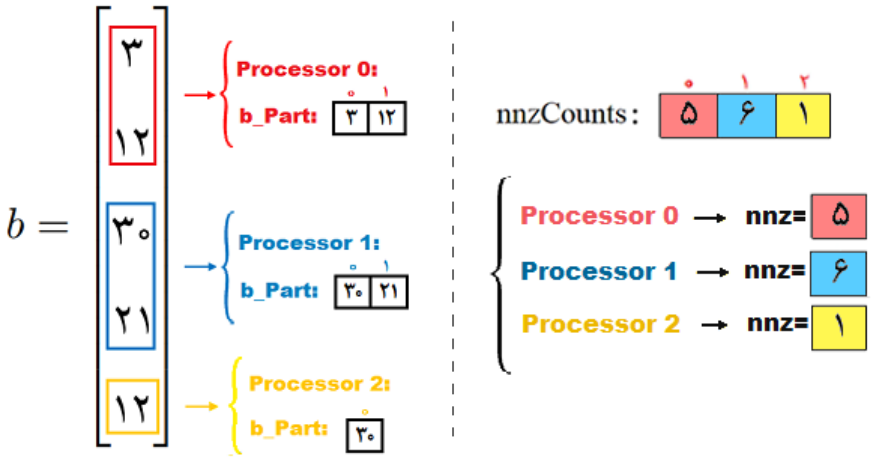
در خط شماره ۲ برنامه متغیر myRowsSize نشان‌دهنده تعداد سطرهای ماتریس A و تعداد مولفه‌هایی از بردار سمت راست b می‌باشد که در اختیار هریک از پردازشگرها قرار خواهند گرفت. به عنوان مثال myRowsSize برای پردازشگری که myid آن برابر با ۱ می‌باشد، مقدار ۲ خواهد بود. یعنی دو سطر از ماتریس A و دو درایه از بردار b در اختیار پردازشگر شماره یک قرار خواهد گرفت. در قسمت قبل به صورت مفصل توضیح داده شد که کدام سطرها و کدام درایه‌ها می‌باشند.

در خط شماره ۸ تا ۱۷ این برنامه با استفاده از دستور MPI_Scatterv درایه‌های b بین پردازشگرهای موجود به گونه‌ای توزیع می‌شود که هرکدام از پردازشگرها به اندازه myRowsSize از درایه‌های بردار b را در اختیار دارند و این درایه‌ها در یک آرایه محلی به نام b_Part که فقط بر روی همان پردازشگر تعریف شده، ذخیره خواهد شد. اینکه کدام درایه‌ها و چه تعدادی از درایه های بردار b در اختیار پردازشگرها قرار بگیرند، توسط دو آرایه rowCounts و rowDispls کنترل خواهد شد. به همین دلیل از دستور MPI_Scatterv استفاده شده است. کار توزیع بردار سمت راست در بین تمام پردازشگرها توسط پردازشگر شماره صفر (پردازشگری که myid=۰) انجام می‌شود. که در خط شماره ۱۶ الگوریتم اندیس صفر اشاره به این مطلب دارد. به عنوان مثال اگر به شکل (۳-۲) نگاه کنید، اجرای این بخش از برنامه برای پردازشگر شماره یک باعث می‌شود که درایه‌های ۳۰ و ۲۸ از بردار b توسط پردازشگر شماره صفر با استفاده از دستور MPI_Scatterv در اختیار پردازشگر شماره ۱ قرار بگیرد. و آرایه محلی b_Part به صورتی که در شکل نشان داده شده ، بر روی پردازشگر شماره ۱ پر خواهد شد.

در خطوط ۱۹ تا ۲۷ برنامه، عملیات MPI_Scatter توسط پردازشگر شماره صفر برای توزیع هر یک از مولفه‌های آرایه nnzCounts بین پردازشگرها استفاده‌خواهد شد. یادآوری می‌کنیم که دستور MPI_Scatter آرایه nnzCounts را به صورت مساوی بین پردازشگرها تقسیم می‌کند و عناصر دریافت شده در متغیر nnz ذخیره خواهد شد. لازم به ذکر است که عناصر آرایه nnzCounts شامل تعداد درایه‌های غیرصفر آرایه‌های AA و JA می‌باشد که در اختیار هریک از پردازشگرها قرار می‌گیرند. به عنوان نمونه اگر به شکل (۳-۲) دقت کنید، اجرای این بخش از برنامه بر روی پردازشگر شماره یک باعث می‌شود که پردازشگر شماره صفر، عدد ۶ را در اختیار پردازشگر شماره صفر قرار بدهد. که این عدد تعداد درایه‌های غیرصفر سطرهای شماره ۲ و ۳ ماتریس می‌باشد که در اختیار پردازشگر شماره یک قرار دارد.

تا به این قسمت برنامه اینکه هر پردازشگر چه سطرهایی از ماتریس A را در اختیار دارد، کاملا مشخص است. همچنین تعداد عناصر غیرصفر موجود در ماتریس A که در اختیار هر پردازشگر قرار دارد در متغیر nnz ذخیره شده است.

در خطوط ۳۰ تا ۳۹ این برنامه، بردار IA توسط پردازشگر شماره صفر با استفاده از دستور MPI_Scatterv بین پردازشگرهای مختلف توزیع خواهد شد. بخشی از آرایه IA که در اختیار هرکدام از پردازشگرها قرار می‌گیرد، در یک آرایه محلی به نام my_ia درون هر پردازشگر ذخیره می‌شود. بعد این آرایه به اندازه myRowsSize+1 می‌باشد که در خط ۵



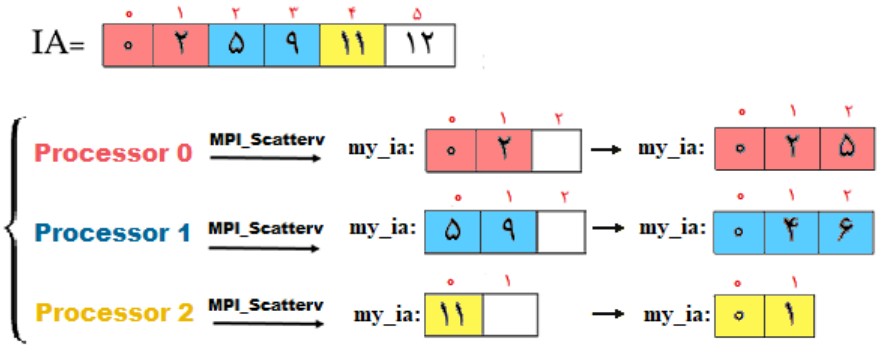
شکل ۳-۲: توزیع دو بردار b و nnzCounts بین پردازشگرها

برنامه مشخص شده است. اینکه کدام بخش از درایه‌های IA و به چه تعداد در اختیار هر پردازشگر قرار می‌گیرد، توسط دو آرایه کنترلی rowCounts و rowDispls کنترل خواهد شد. به عنوان مثال به شکل شماره (۳-۳) توجه کنید اجرای این بخش از برنامه بر روی پردازشگر شماره یک به این صورت است که از خانه شماره ۲ (rowDispls[1]) به تعداد ۲ درایه (rowCounts[1]) در اختیار پردازشگر شماره یک قرار خواهد گرفت.

آرایه IA یک آرایه اشاره‌گر می‌باشد که به مقادیر عددی و اندیس‌های ستونی عناصر غیرصفر ماتریس A در آرایه‌های AA و JA اشاره می‌کند. هنگامی که آرایه‌های AA و JA بین پردازشگرها توزیع شوند، هر پردازشگر آرایه‌های AA و JA محلی مربوط به خود را در اختیار دارد. پس آرایه my_ia باید به مقادیر آرایه‌های محلی AA و JA که بر روی هر پردازشگر متفاوت است، اشاره کند. برای این منظور درایه‌های آرایه my_ia در خطوط ۴۲ تا ۴۶ این زیربرنامه به‌روز می‌شوند. به این صورت که مقدار هر عنصر در آرایه my_ia از مقدار اولین عنصر آرایه کسر شده و نتیجه آن در آرایه آپدیت شده قرار خواهد گرفت. در انتها مقدار آخرین عنصر آرایه my_ia برابر با تعداد عناصر غیر صفر موجود در سهم هر پردازشگر از ماتریس A می‌باشد که با استفاده از دستور MPI_Scatter از آرایه nnzCounts دریافت شده است.

به عنوان مثال در شکل (۳-۳) برای پردازشگر شماره ۱ مقادیر ۵ و ۹ از آرایه IA دریافت شده و در آرایه محلی my_ia ذخیره خواهد شد. در اجرای خطوط ۴۲ تا ۴۶ برنامه برای پردازشگر شماره ۱، متغیر Offset برابر با ۵ می‌باشد. مقدار آپدیت شده به ازای اولین عنصر آرایه my_ia برابر با صفر (۰ = ۵-۵) و به ازای عنصر دوم آرایه، مقدار آن ۴ (۴ = ۵-۹) خواهد بود. در انتها مقدار آخرین عنصر آرایه برابر با nnz محلی پردازشگر شماره صفر، یا به عبارتی ۶ خواهد بود.

تا به این قسمت، با اجرای زیربرنامه (۳-۲) هرکدام از پردازشگرها می‌دانند که باید چه تعداد از درایه‌های AA و JA را از کدام قسمت این آرایه‌ها در اختیار بگیرند. اما هنوز به ازای این درایه‌ها هیچ‌گونه عملیات توزیعی انجام نشده است. حال می‌خواهیم این دو بردار را بین پردازشگرهای مختلف توزیع کنیم. زیربرنامه (۳-۳) کار توزیع بردارهای AA و JA بین پردازشگرهای متفاوت انجام خواهد داد. در این برنامه از دو آرایه nnzCounts و nnzDispls که مقادیر آن‌ها در زیربرنامه (۳-۱) پر شده‌اند استفاده می‌کنیم.



شکل ۳-۳: توزیع بردار IA بین پردازشگرها

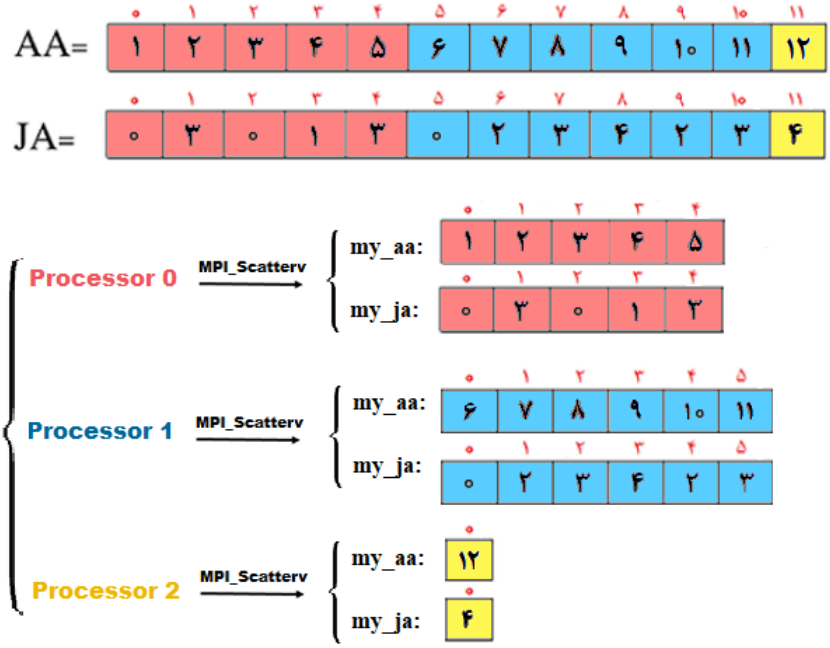
خطوط ۵ تا ۱۴ برنامه برای توزیع بردار AA و خطوط ۱۷ تا ۲۶ برنامه برای پخش بردار JA به کار می‌روند. در ادامه به توضیح هریک از آن‌ها می‌پردازیم.

برنامه ۳-۳: توزیع بردار AA و JA با استفاده از دو آرایه nnzCounts و nnzDispls

```
double *my_aa = new double[nnz];
int *my_ja = new int[nnz];

//! Scatter matrix entries to each processor
MPI_Scatterv(
  AA,
  nnzCounts,
  nnzDispls,
  MPI_DOUBLE,
  my_aa,
  nnz,
  MPI_DOUBLE,
  0, //root rank
  MPI_COMM_WORLD);

//! Scatter partial Column Index to each processor
MPI_Scatterv(
  JA,
  nnzCounts,
  nnzDispls,
  MPI_INT,
  my_ja,
  nnz,
  MPI_INT,
  0, //root rank
  MPI_COMM_WORLD);
```

در خط شماره ۵ تا ۲۶ این برنامه با استفاده از دستور MPI_Scatterv درایه‌های آرایه‌های AA و JA بین پردازشگرهای موجود به گونه‌ای توزیع می‌شود که هرکدام از پردازشگرها به اندازه nnz از درایه‌های AA و JA را در اختیار دارند و این درایه‌ها در یک آرایه محلی به نام my_ia و my_ja که فقط بر روی همان پردازشگر تعریف شده، ذخیره خواهد شد. اینکه کدام

درایه‌ها و چه تعدادی از درایه های AA و JA در اختیار پردازشگرها قرار بگیرند، توسط دو آرایه nnzCounts و nnzDispls کنترل خواهد شد. به همین دلیل از دستور MPI_Scatterv استفاده شده است. این کار توزیع در بین تمام پردازشگرها توسط پردازشگـر شماره صفر (پردازشگری که myid=۰) انجام می‌شود. که خطوط شماره ۱۳ و ۲۵ الگوریتم و اندیس صفر اشاره به این مطلب دارد. به عنوان مثال اگر به شکل (۳-۳) نگاه کنید، اجرای این بخش از برنامه برای پردازشگر شماره ۱ به این صورت است که از خانه شماره ۵ (nnzDispls[1]) به تعداد ۶ درایه (nnzCounts[1]) از آرایه‌های AA و JA توسط پردازشگر شماره صفر با استفاده از دستور MPI_Scatterv در اختیار پردازشگر شماره ۱ قرار می‌گیرد.



شکل ۳-۴: توزیع دو بردار AA و JA بین پردازشگرها

۳-۳ زیربرنامه‌های موازی استفاده شده در الگوریتم گرادیان مزدوج

ابتدا به تصویر (۳-۵) توجه کنید که در آن الگوریتم گرادیان مزدوج شرح داده شده است. بخش‌های موازی، زیربرنامه‌ها و دستورات MPI که برای پیاده‌سازی موازی این الگوریتم به‌کار گرفته شده‌اند، مشخص گردیده است. نکته قابل توجه این است که بردار سمت راست b و بردار اولیه x_0 که برای شروع این الگوریتم مورد استفاده قرار می‌گیرند بردارهای توزیعی و همچنین ماتریس A ماتریس توزیعی می‌باشد. یعنی بردارهای x_0 و b به صورت همسان بین پردازشگرهای متفاوت توزیع شده‌اند. به عنوان مثال اگر پردازشگر شماره صفر درایه‌های ۱ تا n1 بردار x_0 را در اختیار داشته باشد، همان درایه‌های ۱ تا n1 از بردار b در اختیار پردازشگر شماره صفر قرار خواهد داشت. این قرارداد تا آخر الگوریتم پابرجا خواهد بود. به طور مشابه ماتریس توزیعی ماتریسی است که سطرهای آن در اختیار پردازشگرهای متفاوت قرار دارد و به فرمت فشرده سطری بر روی

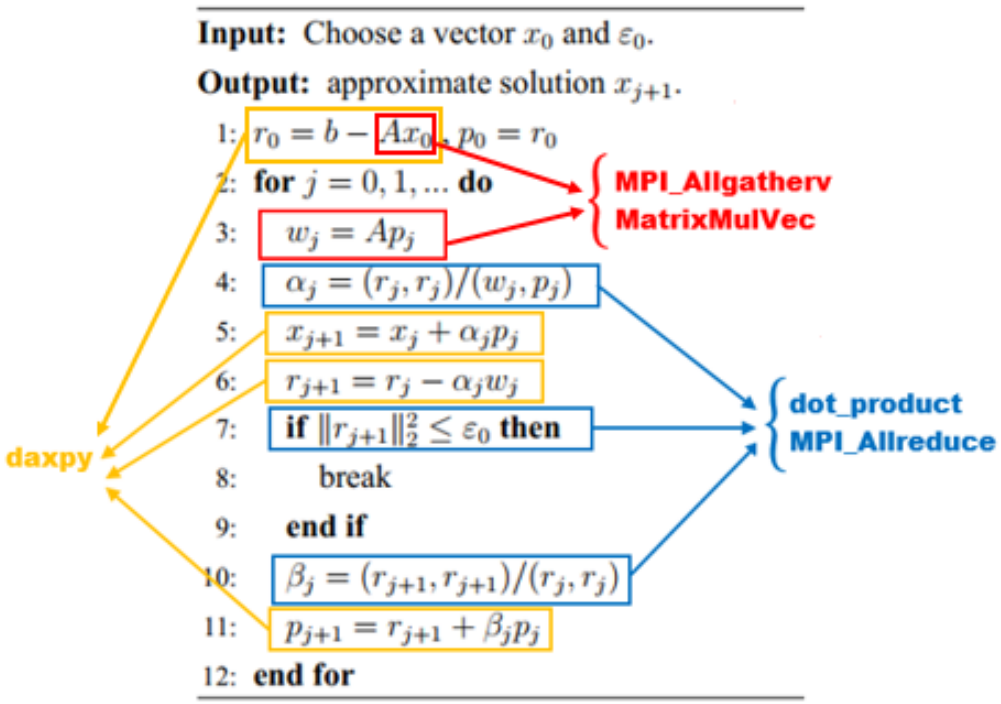
هر پردازشگر ذخیره شده است.

در خطوط شماره ۱ و ۳ الگوریتم دو حاصلضرب ماتریس در بردار Ax_0 و Ap_j وجود دارد. برای محاسبه ضرب‌هایی با این شکل، از دستور MPI_Allgatherv و زیربرنامه MatrixMulVec استفاده می‌شود و هر پردازشگر به ازای سطرهایی از ماتریس A که در اختیار خود دارد، حاصلضرب ماتریس در بردار را محاسبه می‌کند.

در خطوط ۴، ۷ و ۱۰ الگوریتم ضرب‌های داخلی وجود دارند. بردارهایی که در این ضرب‌های داخلی مشارکت دارند، بردارهای توزیعی می‌باشند یعنی بین پردازشگرهای متفاوت توزیع شده‌اند. برای محاسبه این ضرب‌های داخلی، از زیربرنامه موازی dot_product استفاده می‌شود که در آن هر پردازشگر ضرب داخلی بردار محلی مربوط به خود را محاسبه می‌کند و در ادامه با دستور MPI_Allreduce حاصل به دست آمده از ضرب داخلی توسط هر پردازشگر، جمع‌آوری خواهد شد.

در خطوط ۱، ۵، ۶ و ۱۱ الگوریتم، حاصلجمع موازی دو بردار توزیعی محاسبه می‌شود. برای محاسبه این‌گونه جمع‌های برداری در محیط MPI از زیربرنامه موازی daxpy استفاده می‌شود که در آن هر پردازشگر بخشی از دو بردار را که در اختیار خود دارد، با یکدیگر جمع می‌کند.

در ادامه این فصل به توضیح مفصل هریک از زیربرنامه‌های MatrixMulVec، dot_product و daxpy خواهیم پرداخت و جزئیات پیاده‌سازی آن‌ها و دستوراتی که در آن‌ها استفاده شده است را شرح خواهیم داد:

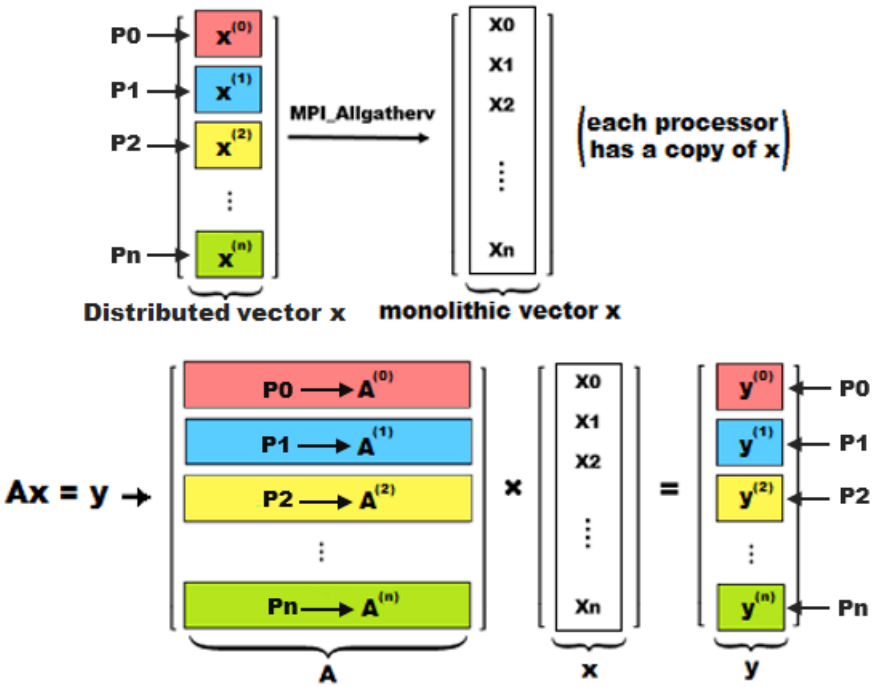


شکل ۳-۵: الگوریتم گرادیان مزدوج و بخش‌ها و زیربرنامه‌های موازی آن در محیط MPI

- **ضرب ماتریس در بردار (MatrixMulVec):** در خط شماره ۱ و ۳ الگوریتم شکل (۳-۵)، حاصلضرب

ماتریس A در بردار x_0 و p_j محاسبه می‌شود. در خط شماره ۳ حاصلضرب Ap_j در بردار w_j ذخیره خواهد شد. برای راحتی در نوشتار، فرض می‌کنیم که در این دو خط نیاز به محاسبه حاصلضرب $y = Ax$ خواهیم داشت که در خط شماره ۱ بردار x همان x_0 و در خط شماره ۳ همان بردار p_j می‌باشد.

نکته اساسی در این حاصلضرب این است که ماتریس A و بردار x ماتریس و بردار توزیعی هستند. ماتریس توزیعی ماتریسی است که سطرهای آن در بین پردازشگرهای متفاوت توزیع شده است. و بردار توزیعی یک بردار است که به بخش‌های متفاوت تقسیم شده و هر بخش آن در اختیار یک پردازشگر قرار دارد. به عنوان مثال برای پردازشگر شماره صفر زیرماتریس محلی $A^{(0)}$ را داریم که شامل بخشی از سطرها و تمام ستون‌های ماتریس A می‌باشد. زیرماتریس $A^{(0)}$ به فرمت فشرده سطری در آرایه‌های my_aa، my_ja و my_ia بر روی پردازشگر شماره صفر ذخیره شده است. اما از بردار x بر روی پردازشگر صفر تنها یک بخش به نام $x^{(0)}$ را داریم. پس انجام ضرب زیرماتریس $A^{(0)}$ در بردار $x^{(0)}$، عملا امکان‌پذیر نخواهد بود. لذا ابتدا باید بردارهای $x^{(i)}$ محلی از پردازشگرهای متفاوت با استفاده از دستور MPI_Allgatherv جمع‌آوری شده و به صورت بردار یکپارچه x دربیاید و در اختیار تمام پردازشگرها قرار بگیرد. آنگاه حاصلضرب $A^{(0)}$ در بردار x محاسبه شده و حاصل آن در بردار $y^{(0)}$ خواهد بود که $y^{(0)}$ بخشی از بردار y است که در اختیار پردازشگر شماره صفر قرار دارد.



شکل ۳-۶: ضرب موازی ماتریس توزیعی A در بردار یکپارچه x در محیط MPI

پیاده‌سازی حاصلضرب زیرماتریس $A^{(0)}$ در بردار x و محاسبه $y^{(0)}$ در برنامه (۳-۴) انجام می‌شود. هسته اصلی این برنامه در الگوریتم (۲-۱) در فصل دو توضیح داده شده است که نسخه سری حاصلضرب یک ماتریس با

فرمت فشرده سطـری را در یک بردار یکپارچه محاسبه می‌کند. در این برنامه نیز همین عملیات به ازای هر پردازشگـر پیاده‌سازی خواهد شد.

در خطوط ۲ تا ۱۰ این برنامه با استفاده از دستور MPI_Allgatherv بردار x که در بین پردازشگرها توزیع است، جمع‌آوری خواهد شد. از آنجایی که محیط اجرای برنامه MPI می‌باشد، این خطوط توسط تمام پردازشگرها انجام می‌شود و برداری که در نهایت تولید شده بردار یکپارچه xx است که همان بردار x در شکل (۳-۶) می‌باشد و در اختیار تمام پردازشگرها قرار دارد.

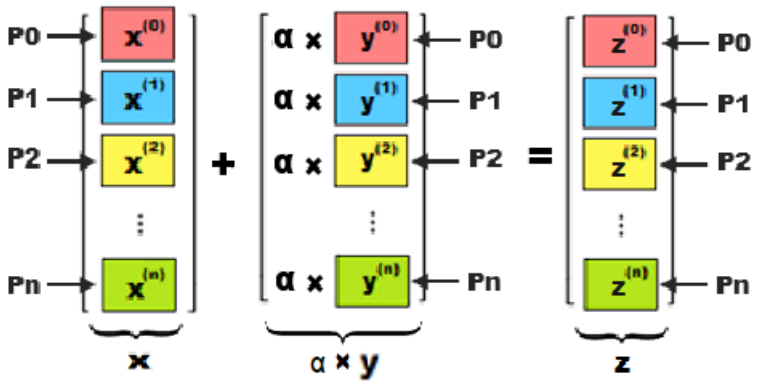
خطوط ۱۲ تا ۲۱ نیز توسط تمام پردازشگرها انجام می‌شود و هر پردازشگر سهم مربوط به خود از ماتریس A را که به صورت فشرده سطری ذخیره شده است، در بردار یکپارچه x ضرب می‌کنند. نکته قابل توجه این است که نتیجه به‌دست آمده از عملیات حاصلضرب، در بردار y ذخیره خواهد شد که یک بردار توزیعی می‌باشد.

برنامه ۳-۴: ضرب ماتریس تنک با فرمت CSR در یک بردار در محیط MPI

```
//! Allgather partial vector from each processor to convert      ۱
    monolithic vector
MPI_Allgatherv(                                                  ۲
  x,                                                             ۳
  myRowsSize,                                                    ۴
  MPI_DOUBLE,                                                    ۵
  xx,                                                            ۶
  rowCounts,                                                     ۷
  rowDispls,                                                     ۸
  MPI_DOUBLE,                                                    ۹
  MPI_COMM_WORLD);                                               ۱۰
                                                                 ۱۱
void MatrixMulVec(double *my_aa, int *my_ja, int *my_ia, double  ۱۲
    *x, double *y, int myRowsSize) {
                                                                 ۱۳
  //! Compute the matrix-vector product                          ۱۴
  for (int i = 0; i < myRowsSize; i++) {                         ۱۵
    y[i] = 0.0;                                                  ۱۶
    for (int j = my_ia[i]; j < my_ia[i+1]; j++) {                ۱۷
      y[i] += my_aa[j]*x[my_ja[j]];                              ۱۸
    }                                                            ۱۹
  }                                                              ۲۰
}                                                                ۲۱
```

- **به‌روزرسانی برداری (daxpy):** در خطوط ۱، ۵، ۶ و ۱۱ الگوریتم شکل (۳-۵) توجه کنید که همگی حاصل $z = x + \alpha y$ را محاسبه می‌کنند که x و y بردارهای توزیعی و α یک اسکالر می‌باشد. به عنوان مثال در خط شماره ۱ الگوریتم، بردار x همان بردار b، اسکالر $\alpha = -1$، بردار y همان Ax_0 و بردار z همان بردار r_0 می‌باشد. یا در خط شماره ۵ الگوریتم، بردار z معادل x_{j+1}، بردار x همان بردار x_j، اسکالر $\alpha = \alpha_j$ و در نهایت بردار y همان بردار p_j خواهد بود. به شکل (۳-۷) توجه کنید که در آن بردار $x^{(0)}$ سهم پردازشگر شماره صفر از بردار x و بردار $y^{(0)}$ سهم پردازشگر شماره صفر از بردار y می‌باشد.

نکته اساسی در این زیربرنامه این است که بردارهای x و y به صورت همسان بین پردازشگرها توزیع شده‌اند. یعنی اندیس درایه‌های بردار $x^{(\circ)}$ برابر با اندیس درایه‌های بردار $y^{(\circ)}$ می‌باشد.



شکل ۳-۷: اجرای موازی جمع برداری در محیط MPI

زیربرنامه‌ای که این عملیات را محاسبه می‌کند در برنامه (۳-۵) آورده شده است. خطوط ۴ تا ۶ از این زیربرنامه به ازای هر پردازشگر انجام می‌شود. به عنوان مثال در این برنامه پردازشگر شماره صفر بردار $x^{(\circ)}$ با α برابر بردار $y^{(\circ)}$ جمع شده و بردار حاصل $z^{(\circ)}$ محاسبه خواهد شد که سهم پردازشگر شماره صفر از بردار z می‌باشد.

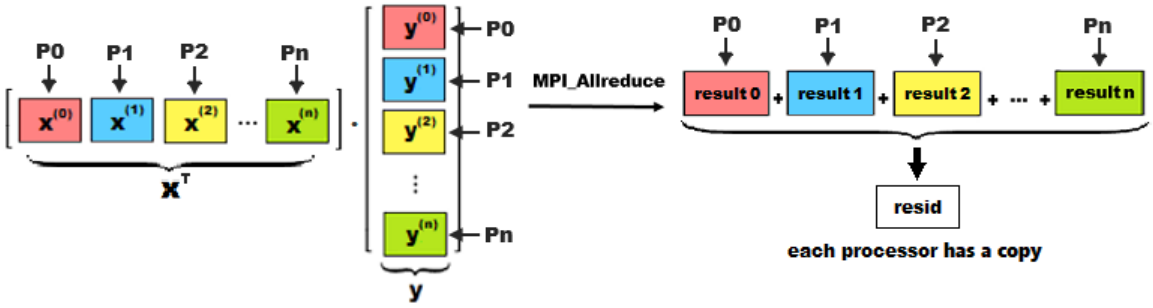
برنامه ۳-۵: انجام عملیات به‌روزرسانی برداری در محیط MPI

```
void daxpy(double *z, double alpha, double *x, double *y,          ۱
    int myRowsSize) {
                                                                   ۲
  //! Compute the addition of a vector with a vector times a      ۳
     constant
  for(int i=0 ; i<myRowsSize ;i++){                                ۴
    z[i] = x[i] + alpha*y[i];                                      ۵
  }                                                                ۶
}                                                                  ۷
```

- **ضرب داخلی دو بردار (dot_product):** در خطوط ۴، ۷ و ۱۰ الگوریتم گرادیان مزدوج نیاز به محاسبه ضرب داخلی دو بردار توزیعی خواهیم داشت. اگر این ضرب داخلی را به صورت $x^T.y$ نشان بدهیم، بردارهای x و y بردارهای توزیعی می‌باشند. به عنوان نمونه در خط ۴ الگوریتم (۳-۵)، در صورت کسر x و y همان بردار r_j می‌باشد و در مخرج آن بردار x معادل بردار w_j و بردار y همان p_j خواهد بود.

شکل (۳-۸) طرح کلی محاسبه ضرب داخلی بر روی پردازشگرهای متفاوت را نشان می‌دهد. پردازشگر شماره iام فقط بخش‌های $x^{(i)}$ و $y^{(i)}$ از بردارهای x و y را در اختیار خود دارد. همین پردازشگر حاصلضرب داخلی بردار $x^{(i)}$ در $y^{(i)}$ را محاسبه می‌کند و حاصل آن را در یک متغیر به نام $\text{result}^{(i)}$ ذخیره خواهد شد. در نهایت با توجه

به اینکه هریک از پردازشگرها باید به مقدار ضرب داخلی دسترسی داشته باشد، تمام ضرب‌های داخلی محاسبه شده توسط پردازشگرهای مختلف با دستور MPI_Allreduce و استفاده از عملوند جمع با یکدیگر جمع شده و در یک مقدار واحد ذخیره خواهند شد. این مقدار همان حاصلضرب بردارهای توزیعی x و y می‌باشد که هریک از پردازشگرها به آن دسترسی دارند.



شکل ۳-۸: اجرای موازی ضرب داخلی دو بردار در محیط MPI

در ادامه به برنامه (۳-۶) دقت کنید. خطوط ۴ تا ۶ برنامه توسط هریک از پردازشگرها اجرا خواهد شد. اجرای این خطوط برای پردازشگر شماره i، به این صورت می‌باشد که بردارهای محلی $x^{(i)}$ و $y^{(i)}$ در یکدیگر ضرب شده و حاصل در پارامتری به نام $\text{result}^{(i)}$ ذخیره می‌شود که فقط برای پردازشگر شماره i قابل دسترسی می‌باشد.

در خطوط ۱۱ تا ۱۷ برنامه، استفاده از دستور MPI_Allreduce باعث می‌شود که تمام $\text{result}^{(i)}$ محلی با یکدیگر جمع شده و متغیر resid حاصل می‌شود که در اختیار تمام پردازشگرها قرار دارد.

برنامه ۳-۶: انجام عملیات ضرب داخلی در محیط MPI

```
double dot_product(double *x, double *y, int myRowsSize) {
  //! Compute the dot product of the provided vectors
  double result = 0.0;
  for(int i=0 ; i<myRowsSize ;i++){
    result += x[i]*y[i];
  }
  return result;
}

MPI_Allreduce(
  &result,
  &resid,
  1,
  MPI_DOUBLE,
  MPI_SUM,
  MPI_COMM_WORLD);
```

فصل ۴

پیاده‌سازی‌های عددی و نتایج آن

۴-۱ مقدمه

در این فصل نتیجه عملکرد الگوریتم گرادیان مزدوج در هریک از محیط‌های برنامه‌نویسی موازی MPI و OpenMP ارزیابی می‌گردد که جزئیات هریک از این محیط‌های موازی در فصل‌های گذشته توضیح داده شد. برنامه‌ای که در پیوست (آ-۱) موجود می‌باشد به زبان برنامه‌نویسی پایتون بوده و با استفاده از آن دستگاه‌های مصنوعی را تولید می‌کنیم. این دستگاه‌های مصنوعی به شکل $Ax = b$ هستند که در آن بردار جواب x برداری است که تمام درایه‌های آن ۱ می‌باشد و ماتریس ضرایب A در این دستگاه‌ها ماتریسی متقارن، معین مثبت و تنک است که به فرمت فشرده سطری ذخیره شده است.

الگوریتم گرادیان مزدوج در گام $j+1$ ام، بردار x_{j+1} را تولید می‌کند و با فرض اینکه $r_{j+1} = b - Ax_{j+1}$ باشد. چنانچه شرط (۴-۱) برقرار باشد آنگاه الگوریتم گرادیان مزدوج متوقف خواهد شد. در پیاده‌سازی عددی این فصل پارامتر ε برابر با 10^{-6} در نظر گرفته شده است.

$$\|r_{j+1}\|_2 \leq \varepsilon \qquad (۴-۱)$$

در برنامه موجود در پیوست (آ-۲) پیاده‌سازی الگوریتم گرادیان مزدوج در محیط OpenMP و برنامه موجود در پیوست (آ-۳) پیاده‌سازی الگوریتم گرادیان مزدوج در محیط MPI می‌باشند. در ادامه این فصل، این دو برنامه را بر روی پردازنده‌های متفاوت اجرا کرده و به ارزیابی تعداد تکرارها و زمان اجرای نسخه موازی الگوریتم گرادیان مزدوج در مقایسه با نسخه غیرموازی(سری) الگوریتم می‌پردازیم.

۴-۲ نتایج آزمایش‌های عددی در محیط OpenMP

در این بخش دستگاه‌های مصنوعی که تولید کرده‌ایم را با استفاده از الگوریتم گرادیان مزدوج در محیط OpenMP اجرا می‌کنیم. تمام آزمایشات عددی در این فصل بر روی یک رایانه با پردازنده Intel(R) Core(TM) i3-1005G1 و ۴ هسته در سیستم‌عامل ویندوز انجام شده است. الگوریتم گرادیان مزدوج در محیط OpenMP با استفاده از کامپایلر GNU کامپایل شده است.

در جدول (۴-۱) ابعاد ماتریس‌های مختلف، تعداد درایه‌های غیرصفر در هریک از این ماتریس‌ها و تعداد رشته‌هایی که برای اجرای موازی روش گرادیان مزدوج مورد استفاده قرار گرفته، عنوان گردیده است. تمامی زمان‌های مطرح شده در جداول و تصاویر این فصل بر حسب ثانیه می‌باشد. هریک از اعدادی که در این جدول آورده شده میانگین ۵ بار اجرای الگوریتم گرادیان مزدوج در محیط OpenMP است.

در جدول (۴-۱) زمان الگوریتم گرادیان مزدوج و تعداد تکرارهای لازم برای برقراری شرط همگرایی (۴-۱) آورده شده است. در این جدول اجرای الگوریتم گرادیان مزدوج با ۱، ۲ و ۴ رشته بررسی گردیده است. دلیل اینکه برای اجرای الگوریتم از تعداد رشته‌های بیشتری استفاده نشده، خصوصیات سخت‌افزاری رایانه می‌باشد که حداکثر ۴ هسته را پشتیبانی می‌کند. از آنجایی که به صورت پیش‌فرض هر هسته به عنوان یک رشته در نظر گرفته می‌شود، بنابراین نمی‌توان بیشتر از ۴ رشته به صورت فیزیکی استفاده نمود.

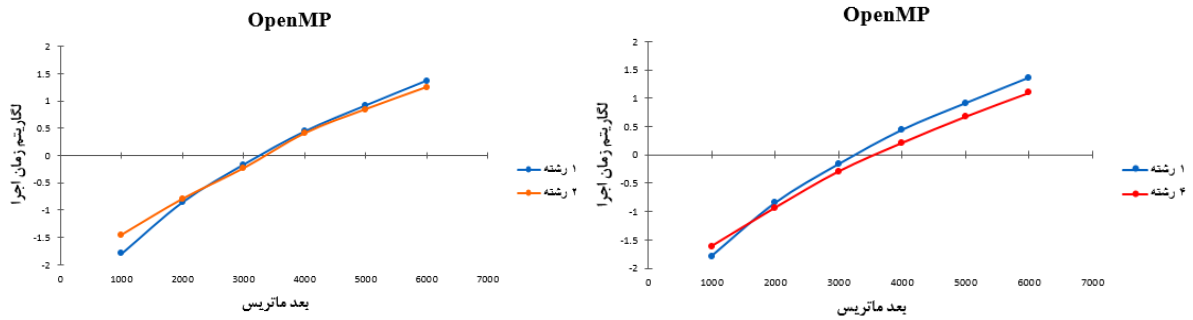
جدول ۴-۱: زمان اجرا و تعداد تکرارهای الگوریتم گرادیان مزدوج در محیط OpenMP

بعد ماتریس	تعداد عناصر غیرصفر	۱ رشته		۲ رشته		۴ رشته	
		زمان اجرا (ثانیه)	تعداد تکرار	زمان اجرا (ثانیه)	تعداد تکرار	زمان اجرا (ثانیه)	تعداد تکرار
۵۰۰	۶۸۹۳	۰.۰۰۰۵	۳۳	۰.۰۱۸۰	۳۳	۰.۰۲۳۲	۳۳
۱۰۰۰	۴۳۱۱۸	۰.۰۱۶۲	۵۷	۰.۰۳۴۸	۵۷	۰.۰۲۴۲	۵۷
۲۰۰۰	۲۹۵۷۸۰	۰.۱۴۰۶	۱۲۲	۰.۱۵۹۰	۱۲۳	۰.۱۱۶۴	۱۲۳
۳۰۰۰	۹۱۷۴۱۹	۰.۶۸۱۲	۲۲۲	۰.۵۷۸۴	۲۲۲	۰.۵۱۱۰	۲۲۱
۴۰۰۰	۲۰۸۶۵۴۷	۲.۷۳۶۶	۳۹۴	۲.۵۲۴۰	۳۹۴	۱.۶۴۱۴	۳۹۳
۵۰۰۰	۳۹۱۸۰۰۱	۸.۱۲۲۴	۶۳۵	۶.۹۸۹۶	۶۳۴	۴.۷۷۰۸	۶۳۳
۶۰۰۰	۶۵۳۲۳۵۳	۲۳.۰۱۰۶	۱۰۶۱	۱۷.۸۰۲۰	۱۰۶۰	۱۲.۶۳۴۰	۱۰۵۷

در جدول (۴-۱) اعدادی که در ستون "۱ **رشته**" آورده شده‌اند، نشان دهنده اجرای سری الگوریتم گرادیان مزدوج هستند. اعداد موجود در دو ستون "۲ **رشته**" و "۴ **رشته**" حاصل اجرای موازی الگوریتم گرادیان مزدوج با ۲ و ۴ رشته می‌باشد. اگر بخواهیم زمان‌های اجرا الگوریتم گرادیان مزدوج را در ستون‌های "۱ **رشته**"، "۲ **رشته**" و "۴ **رشته**" مقایسه کنیم. این‌گونه به نظر می‌رسد که برای ابعاد ۵۰۰ و ۱۰۰۰ زمان اجرای سری الگوریتم از زمان اجرای موازی الگوریتم بهتر است. در بعد ۲۰۰۰ زمان اجرای سری از زمان اجرای موازی الگوریتم با ۲ رشته بهتر است، در حالی که از زمان اجرای موازی الگوریتم با ۴ رشته بدتر می‌باشد. اما برای ابعاد بالای ۳۰۰۰، عملا اجراهای موازی زمان کمتری را دربر خواهند گرفت. همچنین به صورت کلی زمان اجرای الگوریتم گرادیان مزدوج با ۴ رشته از زمان اجرای الگوریتم با ۲ رشته کمتر می‌باشد. نکته قابل

ذکر در این جدول این است که تعداد تکرارها برای تمام ابعاد و تمام اجراهای سری و موازی تفاوت چندانی ندارند. به طور مثال برای بعد ۶۰۰۰، تعداد تکرار برای اجرای سری ۱۰۶۱ و تعداد تکرار برای اجراهای موازی ۱۰۶۰ و ۱۰۵۷ می‌باشند که اختلاف زیادی ندارند.

در ادامه مقایسه زمان اجرای الگوریتم گرادیان مزدوج در اجراهای سری و موازی در محیط OpenMP در نمودار شکل (۴-۱) آورده شده است. این مقایسه براساس اعداد موجود در جدول (۴-۱) انجام گرفته است.



شکل ۴-۱: مقایسه زمان اجرای الگوریتم سری با الگوریتم موازی گرادیان مزدوج در محیط OpenMP

با توجه به تصویر (۴-۱)، نمودار آبی رنگ به زمان اجرای الگوریتم سری گرادیان مزدوج اشاره دارد. همچنین نمودار قرمز زمان اجرای الگوریتم با ۴ رشته و نمودار نارنجی زمان اجرای الگوریتم با ۲ رشته می‌باشند. همان‌طور که مشاهده می‌کنید محور x ها نشان دهنده بعد ماتریس ضرایب و محور y ها نشان دهنده لگاریتم زمان اجرا می‌باشد که بر اساس واحد ثانیه در نظر گرفته شده است.

با توجه به تصویر سمت چپ قبل از بعد ۲۰۰۰، نمودار آبی رنگ در زیر نمودار نارنجی قرار دارد. یعنی زمان اجرای نسخه سری از زمان اجرای نسخه موازی کمتر می‌باشد. پس از بعد ۲۰۰۰ دو نمودار نزدیک به هم هستند اما تا حدودی نمودار نارنجی رنگ زیر نمودار آبی رنگ قرار می‌گیرد. به این معنا که به طور میانگین می‌توان گفت نسخه موازی با ۲ رشته زمان اجرای کمتری را نسبت به نسخه سری به خود اختصاص می‌دهد.

با توجه به اینکه زمان اجرای سری الگوریتم گرادیان مزدوج با نسخه موازی الگوریتم با ۲ رشته نزدیک به هم هستند. از ۴ رشته برای اجرای الگوریتم گرادیان مزدوج استفاده شده است که نمودار زمان اجرای آن در شکل سمت راست آورده شده است. در ابعاد کمتر از ۲۰۰۰ نمودار آبی رنگ تقریبا زیر نمودار قرمز قرار دارد و زمان سری اجرای الگوریتم از زمان موازی آن کمتر است. برای ابعاد ۲۰۰۰ به بالا نمودار آبی رنگ بر روی نمودار قرمز رنگ قرار خواهد گرفت، یعنی نسخه موازی با ۴ رشته زمان کمتری را نسبت به نسخه سری به خود اختصاص می‌دهد. با توجه به تصاویر می‌توان این‌گونه استدلال کرد که اگر امکانات سخت‌افزاری و فیزیکی فراهم باشد، با افزایش تعداد رشته‌ها برای ابعاد بالا نسخه موازی زمان کمتری را برای اجرا به خود اختصاص خواهد داد.

۴-۳ نتایج آزمایش‌های عددی در محیط MPI

در این بخش، دستگاه‌های مصنوعی تولید شده با ابعاد مختلف را با استفاده از الگوریتم گرادیان مزدوج در محیط MPI اجرا می‌کنیم. نتایج آزمایشات عددی در جدول (۴-۲) آورده شده است. مشابه با جدول (۴-۱) اعداد موجود در جدول، میانگین ۵ بار اجرای الگوریتم گرادیان مزدوج بر روی هرکدام از دستگاه‌های مصنوعی می‌باشد. در این جدول بعد ماتریس، تعداد عناصر غیرصفر، زمان اجرای الگوریتم برحسب ثانیه و تعداد تکرارها آورده شده است. الگوریتم گرادیان مزدوج در محیط MPI با استفاده از کامپایلر IntelMPI بر روی ۱، ۲ و ۴ هسته اجرا شده است. شرط همگرایی الگوریتم گرادیان مزدوج رابطه (۴-۱) می‌باشد.

با مقایسه ستون‌های "**۱ هسته**"، "**۲ هسته**" و "**۴ هسته**" در جدول (۴-۲) درمی‌یابیم که برای بعد ۵۰۰ زمان اجرای سری از زمان اجراهای موازی الگوریتم گرادیان مزدوج کمتر است. در بعد ۱۰۰۰ زمان اجرای سری از زمان اجرای الگوریتم موازی با ۲ هسته بهتر و از زمان اجرای موازی الگوریتم با ۴ هسته بدتر خواهد بود.

برای ابعاد بالای ۲۰۰۰، زمان اجراهای موازی الگوریتم گرادیان مزدوج از زمان اجرای سری الگوریتم کمتر خواهد بود. لازم به ذکر است که زمان اجرای موازی الگوریتم با ۴ هسته از زمان اجرای موازی الگوریتم با ۲ هسته کمتر می‌باشد. در ضمن تعداد تکرارها در این جدول برای تمام ابعاد و تمام اجراهای سری و موازی تفاوت چندانی ندارند. به طور مثال برای بعد ۵۰۰۰، تعداد تکرارهای اجرای سری ۶۴۲، تعداد تکرار اجرای موازی با ۲ هسته ۶۴۲ و تعداد تکرار اجرای موازی با ۴ هسته ۶۳۹ می‌باشند که اختلاف زیادی ندارند و به صورت مساوی درنظر گرفته می‌شوند.

جدول ۴-۲: زمان اجرا و تعداد تکرارهای الگوریتم گرادیان مزدوج در محیط MPI

بعد ماتریس	تعداد عناصر غیرصفر	۱ هسته		۲ هسته		۴ هسته	
		تعداد تکرار	زمان اجرا (ثانیه)	تعداد تکرار	زمان اجرا (ثانیه)	تعداد تکرار	زمان اجرا (ثانیه)
۵۰۰	۶۹۴۷	۳۴	۰.۰۰۱۲	۳۴	۰.۰۰۴۴	۳۴	۰.۰۰۲۶
۱۰۰۰	۴۳۷۶۷	۵۷	۰.۰۱۰۴	۵۷	۰.۰۱۱۶	۵۷	۰.۰۰۸۲
۲۰۰۰	۲۹۴۵۳۷	۱۲۰	۰.۱۲۶۶	۱۲۰	۰.۱۱۴۶	۱۲۰	۰.۰۹۲۶
۳۰۰۰	۹۱۴۲۸۳	۲۲۴	۰.۷۱۱۶	۲۲۴	۰.۵۹۳۸	۲۲۳	۰.۴۱۷۶
۴۰۰۰	۲۰۶۶۸۴۵	۳۸۷	۲.۷۶۱۴	۳۸۵	۲.۳۳۸۸	۳۸۴	۱.۵۲۵۰
۵۰۰۰	۳۹۱۱۰۲۴	۶۴۲	۸.۷۲۰۸	۶۴۲	۵.۲۳۰۸	۶۳۹	۴.۱۸۷۴
۶۰۰۰	۶۵۳۴۴۲۴	۱۰۴۶	۲۳.۹۶۶۸	۱۰۴۵	۱۳.۴۵۱۸	۱۰۴۲	۱۰.۹۴۳۸

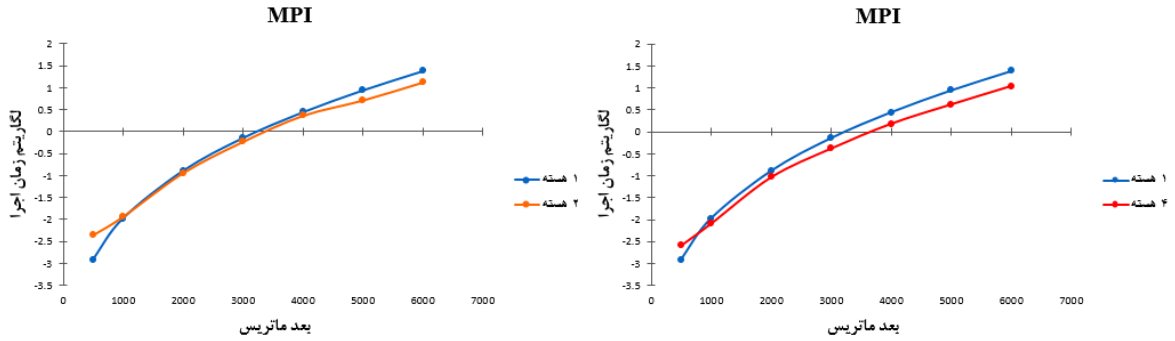
در ادامه مقایسه زمان اجرای الگوریتم گرادیان مزدوج در اجراهای سری و موازی در محیط MPI در نمودار شکل (۴-۲) آورده شده است. این نمودارها براساس اعداد موجود در جدول (۴-۲) رسم شده‌اند.

با تمرکز بر روی تصویر (۴-۲) مشاهده می‌کنیم که نمودارها براساس دو فاکتور لگاریتم زمان اجرا و بعد ماتریس ضرایب در دستگاه معادلات خطی ترسیم شده‌اند. در هر دو تصویر نمودار آبی زمان اجرای الگوریتم سری (۱ هسته)، نمودار نارنجی زمان اجرای الگوریتم با ۲ هسته و نمودار قرمز زمان اجرای الگوریتم با ۴ هسته هستند.

در شکل سمت چپ برای ابعاد کمتر از ۱۰۰۰ نمودار آبی زیر نمودار نارنجی قرار دارد. یعنی زمان اجرای نسخه سری

از زمان اجرای نسخه موازی با ۲ هسته کمتر می‌باشد. برای ابعاد بالاتر از ۱۰۰۰ دو نمودار به یکدیگر نزدیک می‌شوند، اما نمودار نارنجی تقریبا زیر نمودار آبی قرار دارد. به این معنا که زمان اجرای الگوریتم موازی با ۲ هسته از زمان اجرای سری کمتر می‌باشد.

با توجه به اینکه اجرای سری الگوریتم نزدیک به اجرای موازی الگوریتم با ۲ هسته می‌باشند، از ۴ هسته برای اجرای الگوریتم استفاده می‌کنیم که در شکل سمت راست آورده شده است. برای ابعاد کمتر از ۱۰۰۰ زمان اجرای سری بهتر می‌باشد و نموار آبی زیر نمودار قرمز قرار دارد. اما برای ابعاد بالاتر از ۱۰۰۰، نمودار قرمز رنگ در زیر نمودار آبی قرار می‌گیرد که نشان می‌دهد اجرای موازی با ۴ هسته بهتر از اجرای سری می‌باشد.



شکل ۴-۲: نمودار مقایسه زمان اجرا الگوریتم سری با الگوریتم موازی گرادیان مزدوج در محیط MPI

۴-۴ نتیجه‌گیری

در این پایان‌نامه نحوه پیاده‌سازی الگوریتم گرادیان مزدوج برای حل دستگاه معادلات خطی $Ax = b$ در هریک از محیط‌های برنامه‌نویسی موازی توزیعی (MPI) و اشتراکی (OpenMP) مورد بررسی قرار گرفت. برای مقایسه کیفیت پیاده‌سازی نسخه‌های موازی با نسخه‌های سری، دستگاه‌های معادلات خطی مصنوعی را تولید کرده و سپس نسخه‌های موازی و سری را بر روی این دستگاه‌های معادلات خطی مصنوعی اجرا نمودیم. جزئیات این پیاده‌سازی‌ها نشان می‌دهد که برای دستگاه‌های معادلات خطی که دارای ابعاد کوچکی هستند، زمان اجرای نسخه سری نسبت به اجرای نسخه‌های موازی کمتر خواهد بود. اما با افزایش بعد ماتریس، زمان اجرای نسخه‌های موازی نسبت به زمان اجرای نسخه سری کاهش پیدا می‌کند. این در حالی است که تعداد تکرارهای روش گرادیان مزدوج برای نسخه‌های سری و موازی تفاوت چندانی ندارد و تقریبا یکسان است.

پیاده‌سازی‌های موازی بر روی یک رایانه موازی با پردازنده Intel(R) Core(TM) i3-1005G1 و ۴ هسته در سیستم‌عامل ویندوز انجام شده است. نتایج پیاده‌سازی این مساله را بیان می‌کند که اگر امکانات سخت‌افزاری موجود به ما اجازه بدهد که از تعداد هسته‌های بیشتر در محیط MPI و تعداد رشته‌های بیشتری در محیط OpenMP استفاده کنیم، این‌گونه به نظر می‌رسد که با افزایش ابعاد ماتریس استفاده از نسخه‌های موازی اولویت بیشتری نسبت به استفاده از نسخه سری خواهد داشت.

فهرست منابع

[1] Datta, B.N. *Numerical Linear Algebra and Applications, Second Edition*. Other Titles in Applied Mathematics. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2010.

[2] Adve, S, Adve, Vikram S, Agha, Gul, Frank, Matthew I, Garzarán, M, Hart, J, Hwu, W-m, Johnson, R, Kale, L, Kumar, R, et al. Parallel computing research at illinois: The upcrc agenda. *Urbana, IL: Univ. Illinois Urbana-Champaign*, 2008.

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

[4] Flynn, M. J. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, 1972.

[5] Rünger, Gudula and Rauber, Thomas. *Parallel Programming - for Multicore and Cluster Systems; 2nd Edition.* Springer, 2013.

[6] Pacheco, Peter. *An Introduction to Parallel Programming*. Morgan Kaufmann, 2011.

[7] Chandra, R., Dagum, L., Kohr, D., Menon, R., Maydan, D., and McDonald, J. *Parallel Programming in OpenMP*. High performance computing. Elsevier Science, 2001.

[8] Saad, Y. *Iterative methods for sparse linear systems*. Computer Science Ser. PWS publishing, 1996.

پیوست آ

برنامه‌های استفاده شده در این پایان‌نامه

در بخش‌های عددی این پایان‌نامه یعنی در فصل دوم، الگوریتم گرادیان مزدوج را در محیط OpenMP و در فصل شماره ۳، الگوریتم گرادیان مزدوج را در محیط MPI پیاده‌سازی کرده‌ایم که این برنامه‌ها به زبان C++ نوشته شده‌اند. همچنین یک برنامه برای تولید دستگاه‌های مصنوعی نیز داریم که به زبان پایتون نوشته شده است. در ادامه این بخش هر سه برنامه آورده شده و به شرح هریک از آن‌ها می‌پردازیم.

آ-۱ برنامه ساخت دستگاه مصنوعی

این برنامه برای ساخت دستگاه معادلات خطی مصنوعی با زبان پایتون نوشته شده است. در این برنامه دستگاه مصنوعی $Ax = b$ ساخته می‌شود که A یک ماتریس معین مثبت و متقارن و تنک است، بردار x یک بردار تمام ۱ می‌باشد و بردار سمت راست b متناظر با بردار x پر خواهد شد. خروجی این برنامه یک فایل به نام matrix_csr می‌باشد که در آن ماتریس ضرایب با فرمت فشرده سطری و بردار سمت راست b به صورت کامل ذخیره شده است. در خطوط ۷۹ و ۴۸ برنامه‌های (آ-۲) و (آ-۳) از این فایل استفاده خواهد شد. برای درک بهتر نحوه کار این برنامه، خروجی این برنامه برای بعد ۵ در شکل (آ-۱) آورده شده است.

matrix_csr

```
5
5
4.0 4.0 4.0 4.0 4.0
0   1   2   3   4
0   1   2   3   4   5
4.0
4.0
4.0
4.0
4.0
```

شکل آ-۱: خروجی فایل matrix_csr برای بعد ۵

شکل کامل ماتریس ضرایب و بردار سمت راست برای تصویر (آ-۱) به صورت زیر خواهد بود.

$$A = \begin{bmatrix} ۴ & ۰ & ۰ & ۰ & ۰ \\ ۰ & ۴ & ۰ & ۰ & ۰ \\ ۰ & ۰ & ۴ & ۰ & ۰ \\ ۰ & ۰ & ۰ & ۴ & ۰ \\ ۰ & ۰ & ۰ & ۰ & ۴ \end{bmatrix}, b = \begin{bmatrix} ۴ \\ ۴ \\ ۴ \\ ۴ \\ ۴ \end{bmatrix}$$

برنامه آ-۱: برنامه ساخت دستگاه مصنوعی به زبان پایتون

```
"""                                                                    ۱
  Generate a sparse symmetric definite positive matrix.                ۲
  ----------                                                           ۳
  Parameters                                                           ۴
  dim : int,                                                           ۵
    The size of the random matrix to generate.                         ۶
  alpha : float, default=0.95                                          ۷
    The probability that a coefficient is zero (see notes).            ۸
       Larger values
    enforce more sparsity. The value should be in the range            ۹
       0 and 1.
  smallest_coef : float, default=0.1                                   ۱۰
    The value of the smallest coefficient between 0 and 1.             ۱۱
  largest_coef : float, default=0.9                                    ۱۲
    The value of the largest coefficient between 0 and 1.              ۱۳
  -------                                                              ۱۴
  Returns                                                              ۱۵
  prec : sparse matrix of shape (dim, dim)                             ۱۶
    The generated matrix.                                              ۱۷
  Notes                                                                ۱۸
  -----                                                                ۱۹
  The sparsity is actually imposed on the cholesky factor              ۲۰
     of the matrix.
  Thus alpha does not translate directly into the filling              ۲۱
     fraction of the matrix itself.
"""                                                                    ۲۲
                                                                       ۲۳
import numpy as np                                                     ۲۴
import sys                                                             ۲۵
                                                                       ۲۶
if (len(sys.argv) > 1):                                                ۲۷
                                                                       ۲۸
    dim = int(sys.argv[1])                                             ۲۹
    alpha=0.99                                                         ۳۰
    smallest_coef=.1                                                   ۳۱
    largest_coef=.9                                                    ۳۲
    random_state = np.random.mtrand._rand                              ۳۳
                                                                       ۳۴
    chol = 1*np.eye(dim)                                               ۳۵
    aux = random_state.rand(dim, dim)                                  ۳۶
```

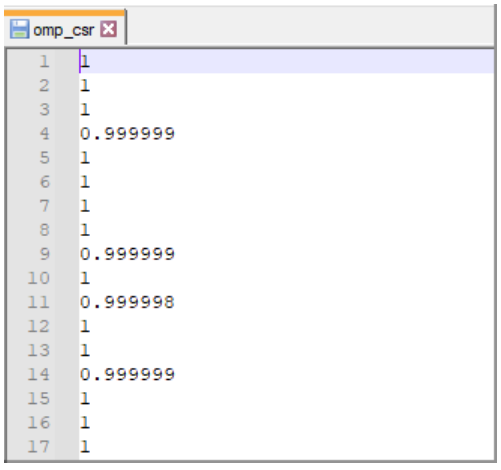
```
    aux[aux < alpha] = 0
    aux[aux > alpha] = (smallest_coef + (largest_coef -
        smallest_coef) * random_state.rand(np.sum(aux > alpha)))
    aux = np.tril(aux, k=-1)

    # Permute the lines
    permutation = random_state.permutation(dim)
    aux = aux[permutation].T[permutation]
    chol += aux
    prec = np.dot(chol.T, chol)


    rowptr = []
    val = []
    colidx = []
    ans = [1 for i in range(dim)]
    b = [0 for i in range(dim)]

    count = 0;
    rowptr.append(count)
    for i in range(0 , dim):
        for j in range (0, dim):
            if prec[i][j] != 0:
                val.append(prec[i][j])
                b[i] += prec[i][j]
                colidx.append(j)
                count=count+1
        rowptr.append(count)


    f = open('matrix_csr', 'w');
    f.write(str(dim))
    f.write('\n')
    f.write(str(rowptr[-1]))
    f.write('\n')

    for i in range(len(val)):
        f.write(str(round(val[i],6)))
        f.write('\t')
    f.write('\n')

    for i in range(len(colidx)):
        f.write(str(colidx[i]))
        f.write('\t')
    f.write('\n')

    for i in range(len(rowptr)):
        f.write(str(rowptr[i]))
        f.write('\t')
    f.write('\n')

    for i in range(dim):
        f.write(str(round(b[i],6)))
        f.write('\n')
    f.close()
```

```
    f = open('ans_csr', 'w')
    for i in range(dim):
        f.write(str(round(ans[i],6)))
        f.write('\n')
    f.close()

else:

    print ("please input the size of matrix (N)")
```

آ-۲ برنامه گرادیان مزدوج در محیط OpenMP

این برنامه به زبان C++ نوشته شده و الگوریتم گرادیان مزدوج را در محیط OpenMP پیاده‌سازی می‌کند. ورودی این برنامه شکل فشرده سطری ماتریس A و بردار سمت راست b از دستگاه $Ax = b$ می‌باشد که با استفاده از برنامه (آ-۱) ساخته شده و در فایل matrix_csr ذخیره شده‌اند. در خط شماره ۷۹ برنامه عملیات خواندن این فایل انجام شده و الگوریتم گرادیان مزدوج بر روی آن اجرا می‌شود. در خطوط ۱۶۵ تا ۱۶۷ برنامه بردار x_0 اولیه، برداری تمام صفر در نظر گرفته می‌شود.

خروجی این برنامه یک فایل به نام omp_csr می‌باشد که در آن بردار جواب حاصل از الگوریتم گرادیان مزدوج ذخیره خواهد شد. همچنین تعداد تکرارها و زمان اجرای الگوریتم چاپ می‌شود. به عنوان مثال پس از اینکه این برنامه را بر روی یک ماتریس معین مثبت متقارن مصنوعی با بعد ۱۰۰۰ اجرا کنیم در خروجی فایلی به نام omp_csr دریافت خواهد شد که در شکل زیر آورده شده است. همانطور که از تصویر مشهود است بردار جواب بدست آمده از برنامه گرادیان مزدوج نزدیک به جواب واقعی دستگاه مصنوعی می‌باشد که برداری تمام ۱ است.



شکل آ-۲: خروجی فایل omp_csr برای بعد ۱۰۰۰

برنامه آ-۲: برنامه گرادیان مزدوج با زبان C++ در محیط موازی OpenMP

```
#include <stdio.h>                                                    ۱
#include <stdlib.h>                                                   ۲
#include <fstream>                                                    ۳
#include <iostream>                                                   ۴
#include <cmath>                                                      ۵
#include <ctime>                                                      ۶
#include <string.h>                                                   ۷
                                                                      ۸
#define OPENMP "openmp_enable"                                        ۹
#ifdef OPENMP                                                         ۱۰
#include <omp.h>                                                      ۱۱
#endif                                                                ۱۲
                                                                      ۱۳
using namespace std;                                                  ۱۴
                                                                      ۱۵
/*****************************************************************/   ۱۶
/* Set the variable parameters for the simulation here.          */   ۱۷
Note that the number of threads for the OpenMP implementation         ۱۸
can be set at the command line. */                                    ۱۹
/*****************************************************************/   ۲۰
                                                                      ۲۱
//! Convergence criteria in terms of orders reduction in the L2      ۲۲
    norm
#define TOLERANCE 1e-6                                                ۲۳
                                                                      ۲۴
/*****************************************************************/   ۲۵
/* Function prototypes. All necessary functions are contained in     ۲۶
    this file. */
/*****************************************************************/   ۲۷
                                                                      ۲۸
//! Driver routine for the conjugate gradient method                  ۲۹
void conjugate_gradient(double *A, int *rowA, int *colA, double       ۳۰
    *x, double *b, int n_nodes, double tol, int freq);
                                                                      ۳۱
//! Subroutine for calculating a sparse matrix-vector product         ۳۲
void sp_mv_product(double *result, double *A, int *rowA, int          ۳۳
    *colA, double *x, int n);
                                                                      ۳۴
//! Subroutine for adding a vector to another multiplied by a         ۳۵
    constant
void daxpy(double *result, double alpha, double *x, double *y,       ۳۶
    int n);
                                                                      ۳۷
//! Subroutine for calculating the dot product of two length n        ۳۸
    vectors
double dot_product(double *x, double *y, int n);                      ۳۹
                                                                      ۴۰
//! Subroutine for taking the L-2 norm of a vector                    ۴۱
double vector_norm(double *vec, int n);                               ۴۲
                                                                      ۴۳
//! Subroutine for performing a deep copy of a vector                 ۴۴
void vector_copy(double *vec_in, double *vec_out, int n);             ۴۵
                                                                      ۴۶
//! Subroutine for deallocate all dynamic memory in the program       ۴۷
void deallocate_arrays(double *A, int *rowA, int *colA, double        ۴۸
```

```
    *x, double *b);
                                                                    ۴۹
/*************************************************************/    ۵۰
/* Main function driving the high-level solver execution.    */    ۵۱
/*************************************************************/    ۵۲
                                                                    ۵۳
int main(int argc, char* argv[]) {                                  ۵۴
                                                                    ۵۵
#ifdef OPENMP                                                       ۵۶
  cout<< "OpenMp Is Active\n";                                      ۵۷
#else                                                               ۵۸
  cout<< "OpenMp Is Deactivated\n";                                 ۵۹
#endif                                                              ۶۰
                                                                    ۶۱
//! Local variables and settings for the CG algorithm              ۶۲
  int freq, n, numberOfNonZero;                                     ۶۳
  double tol = TOLERANCE, StartTime, StopTime, UsedTime;            ۶۴
                                                                    ۶۵
//! Check for number of threads for the OpenMP version             ۶۶
#ifdef OPENMP                                                       ۶۷
  int n_threads;                                                    ۶۸
  printf("Give the number of threads: ");                           ۶۹
  scanf("%d", &n_threads);                                          ۷۰
  omp_set_num_threads(n_threads);                                   ۷۱
#endif                                                              ۷۲
                                                                    ۷۳
//! Pointers to arrays that we need throughout the solver          ۷۴
  double *A, *x, *b;                                                ۷۵
  *rowA, *colA;                                                     ۷۶
                                                                    ۷۷
//! Allocate memory                                                ۷۸
  ifstream matrixfile("matrix_csr");                                ۷۹
  if(!(matrixfile.is_open())){                                      ۸۰
    cout<< "Error: file not found"<<endl;                           ۸۱
    return 0;                                                       ۸۲
  }                                                                 ۸۳
  matrixfile >> n;                                                  ۸۴
  matrixfile >> numberOfNonZero;                                    ۸۵
                                                                    ۸۶
  A = new double [numberOfNonZero];                                 ۸۷
  colA = new int [numberOfNonZero];                                 ۸۸
  rowA = new int [n+1];                                             ۸۹
  x = new double [n];                                               ۹۰
  b = new double [n];                                               ۹۱
                                                                    ۹۲
  for(int i = 0; i < numberOfNonZero; i++){                         ۹۳
    matrixfile >> A[i];                                             ۹۴
  }                                                                 ۹۵
                                                                    ۹۶
  for(int i = 0; i < numberOfNonZero; i++){                         ۹۷
    matrixfile >> colA[i];                                          ۹۸
  }                                                                 ۹۹
                                                                    ۱۰۰
  for(int i = 0; i <= n ; i++){                                     ۱۰۱
    matrixfile >> rowA[i];                                          ۱۰۲
  }                                                                 ۱۰۳
```

```
                                                                    ۱۰۴
  for(int i = 0; i < n; i++){                                       ۱۰۵
    matrixfile >> b[i];                                             ۱۰۶
  }                                                                 ۱۰۷
  matrixfile.close();                                               ۱۰۸
                                                                    ۱۰۹
//! Start the timer for benchmarking                               ۱۱۰
  StartTime = double(clock())/double(CLOCKS_PER_SEC);               ۱۱۱
#ifdef OPENMP                                                       ۱۱۲
  StartTime = omp_get_wtime();                                      ۱۱۳
#endif                                                              ۱۱۴
                                                                    ۱۱۵
//! Solve the system using the Conjugate Gradient method           ۱۱۶
  conjugate_gradient(A, rowA, colA, x, b, n, tol, freq);            ۱۱۷
                                                                    ۱۱۸
//! Stop the timer                                                 ۱۱۹
#ifdef OPENMP                                                       ۱۲۰
  StopTime = omp_get_wtime();                                       ۱۲۱
  UsedTime = StopTime-StartTime;                                    ۱۲۲
#else                                                               ۱۲۳
  StopTime = double(clock())/double(CLOCKS_PER_SEC);                ۱۲۴
  UsedTime = StopTime-StartTime;                                    ۱۲۵
#endif                                                              ۱۲۶
                                                                    ۱۲۷
//Generate files for debug purpouse                                 ۱۲۸
  ofstream Af;                                                      ۱۲۹
  Af.open("omp_csr");                                               ۱۳۰
  for (int i = 0; i < n; i++) {                                     ۱۳۱
    Af<<x[i]<<endl;                                                 ۱۳۲
  }                                                                 ۱۳۳
  Af.close();                                                       ۱۳۴
                                                                    ۱۳۵
//! Free all memory used by the solver                             ۱۳۶
  deallocate_arrays(A, rowA, colA, x, b);                           ۱۳۷
                                                                    ۱۳۸
//! Print the total time for performance benchmarking.             ۱۳۹
  UsedTime=UsedTime*1000;                                           ۱۴۰
                                                                    ۱۴۱
#ifdef OPENMP                                                       ۱۴۲
  cout << "Completed in " << fixed << UsedTime << " ms with ";      ۱۴۳
  cout << n_threads << " threads." << endl;                         ۱۴۴
#else                                                               ۱۴۵
  cout << "\nCompleted in " << fixed << UsedTime;                   ۱۴۶
  cout << " ms in serial." << endl;                                 ۱۴۷
#endif                                                              ۱۴۸
                                                                    ۱۴۹
  cout<< "result saved on file omp_csr.\n";                         ۱۵۰
  return 0;                                                         ۱۵۱
}                                                                   ۱۵۲
                                                                    ۱۵۳
void conjugate_gradient(double *A, int *rowA, int *colA, double *x۱۵۴
    double *b, int n, double tol, int freq) {
                                                                    ۱۵۵
//! Initial variables.                                             ۱۵۶
  freq = n*n;                                                       ۱۵۷
  double resid, alpha, beta, sum;                                   ۱۵۸
```

```
                                                                    ۱۵۹
//! Allocate auxiliary vectors for the CG algorithm                  ۱۶۰
  double *r = new double[n];                                        ۱۶۱
  double *p = new double[n];                                        ۱۶۲
  double *mv_product = new double[n];                               ۱۶۳
                                                                    ۱۶۴
  for (int i = 0; i < n; i++) {                                     ۱۶۵
        x[i] = 0;                                                   ۱۶۶
  }                                                                 ۱۶۷
                                                                    ۱۶۸
//! Initialize the rOld and pOld                                    ۱۶۹
  sp_mv_product(mv_product, A, rowA, colA, x, n);                   ۱۷۰
  daxpy(r, -1.0, mv_product, b, n);                                 ۱۷۱
  vector_copy(r, p, n);                                             ۱۷۲
                                                                    ۱۷۳
  int iter = 0;                                                     ۱۷۴
  while ( true && iter < freq ) {                                   ۱۷۵
                                                                    ۱۷۶
    //! CG iteration                                                ۱۷۷
    iter ++ ;                                                       ۱۷۸
                                                                    ۱۷۹
    //! Sparse matrix-vector product (major portion of the work)    ۱۸۰
    sp_mv_product(mv_product, A, rowA, colA, p, n);                 ۱۸۱
                                                                    ۱۸۲
    //! Compute the alpha step                                      ۱۸۳
        resid = dot_product(r, r, n);                               ۱۸۴
    alpha  = resid/dot_product(p, mv_product, n);                   ۱۸۵
                                                                    ۱۸۶
    //! Update the value of the solution                           ۱۸۷
    daxpy(x, alpha, p, x, n);                                       ۱۸۸
                                                                    ۱۸۹
    //! Residual update (mv_product still holds A*p)                ۱۹۰
    daxpy(r, -1.0*alpha, mv_product, r, n);                         ۱۹۱
                                                                    ۱۹۲
    //! Check the convergence condition                             ۱۹۳
        sum = vector_norm(r,n);                                     ۱۹۴
        if(sum < tol)                                               ۱۹۵
                break;                                              ۱۹۶
                                                                    ۱۹۷
    //! Compute the beta value                                      ۱۹۸
    beta = dot_product(r, r, n)/resid;                              ۱۹۹
                                                                    ۲۰۰
    //! Update the p vector                                         ۲۰۱
    daxpy(p, beta, p, r, n);                                        ۲۰۲
  }                                                                 ۲۰۳
                                                                    ۲۰۴
  //! Free memory and exit                                          ۲۰۵
  delete [] r;                                                      ۲۰۶
  delete [] p;                                                      ۲۰۷
  delete [] mv_product;                                             ۲۰۸
}                                                                   ۲۰۹
                                                                    ۲۱۰
void sp_mv_product(double *result, double *A, int *rowA, int       ۲۱۱
    *colA, double *x, int n) {
                                                                    ۲۱۲
//! Compute the sparse matrix-vector product (CSR format)           ۲۱۳
```

```
int i,j;
#ifdef OPENMP
#pragma omp parallel for default(none) \
private(i,j) shared(result,A,rowA,colA,x,n)
#endif
  for (i = 0; i < n; i++) {
    result[i] = 0.0;
    for (j = rowA[i]; j < rowA[i+1]; j++) {
      result[i] += A[j]*x[colA[j]];
    }
  }
}

void daxpy(double *result, double alpha, double *x, double *y,
    int n) {

//! Compute the addition of a vector with a vector times a constant
int i;
#ifdef OPENMP
#pragma omp parallel for default(none) \
private(i) shared(result,alpha,x,y,n)
#endif
  for (int i = 0; i < n; i++) {
    result[i] = alpha*x[i] + y[i];
  }
}

double dot_product(double *x, double *y, int n) {

//! Compute the dot product of the provided vectors
double result = 0.0; int i;

#ifdef OPENMP
#pragma omp parallel for default(none) \
private(i) shared(x,y,n) ordered reduction(+:result)
#endif
  for (i = 0; i < n; i++) {
    result = result + x[i]*y[i];
  }
  return result;
}

double vector_norm(double *vec, int n) {

//! Compute the L-2 norm of the provided vector
double norm = 0.0; int i;

#ifdef OPENMP
#pragma omp parallel for default(none) \
private(i) shared(vec,n) ordered reduction(+:norm)
#endif
  for (i = 0; i < n; i++) {
    norm += vec[i]*vec[i];
  }
  norm = sqrt(norm);
  return norm;
```

```
}                                                                    ۲۶۹
                                                                     ۲۷۰
void vector_copy(double *vec_in, double *vec_out, int n) {           ۲۷۱
                                                                     ۲۷۲
//! Copy input vector into output vector                            ۲۷۳
int i;                                                               ۲۷۴
#ifdef OPENMP                                                        ۲۷۵
#pragma omp parallel for default(none) \                             ۲۷۶
private(i) shared(vec_out,vec_in,n)                                  ۲۷۷
#endif                                                               ۲۷۸
  for (i = 0; i < n; i++) {                                          ۲۷۹
    vec_out[i] = vec_in[i];                                          ۲۸۰
  }                                                                  ۲۸۱
}                                                                    ۲۸۲
                                                                     ۲۸۳
void deallocate_arrays(double *A, int *rowA, int *colA, double       ۲۸۴
    *x, double *b) {
                                                                     ۲۸۵
//! Deallocation of all dynamic memory in the program.              ۲۸۶
  delete [] A;                                                       ۲۸۷
  delete [] rowA;                                                    ۲۸۸
  delete [] colA;                                                    ۲۸۹
  delete [] x;                                                       ۲۹۰
  delete [] b;                                                       ۲۹۱
}                                                                    ۲۹۲
```

آ-۳ برنامه گرادیان مزدوج در محیط MPI

این برنامه به زبان C++ نوشته شده و الگوریتم گرادیان مزدوج را در محیط MPI پیاده‌سازی می‌کند. مشابه با برنامه (آ-۲) ورودی این برنامه نیز فرمت فشرده سطری ماتریس A و بردار b از دستگاه $Ax = b$ می‌باشد که با استفاده از برنامه (آ-۱) ساخته شده و در فایل matrix_csr ذخیره شده‌اند. در خط شماره ۴۸ برنامه عملیات خواندن این فایل توسط پردازشگر شماره صفر انجام می‌شود. همچنین خطوط ۲۵۳ تا ۲۵۵ برنامه توسط تمام پردازشگرها اجرا شده و هر پردازشگر بردار x_0 مربوط به خود را به صورت برداری تمام صفر مقداردهی می‌کند.

خروجی این برنامه یک فایل به نام mpi می‌باشد که در آن بردار جواب حاصل از الگوریتم گرادیان مزدوج ذخیره خواهد شد. همچنین تعداد تکرارها و زمان اجرای الگوریتم نمایش داده می‌شود.

برنامه آ-۳: برنامه گرادیان مزدوج با زبان C++ در محیط موازی MPI

```
#include <iostream>                                                  ۱
#include <cmath>                                                     ۲
#include <algorithm>                                                 ۳
#include <stdio.h>                                                   ۴
#include <stdlib.h>                                                  ۵
#include <fstream>                                                   ۶
#include <time.h>                                                    ۷
```

```
#include <mpi.h>
using namespace std;
/*************************************************************/
/* Function prototypes. All necessary functions are contained
   in this file. */
/*************************************************************/
//! Driver routine for the conjugate gradient method
int conjugate_gradient(double *my_aa, int *my_ja, int *my_ia,
   double *bPart, double *x, int *datacounts,int *datadispls,
   int myRowsSize, int M);
//! Subroutine for calculating a matrix-vector product
void MatrixMulVec(double *my_aa, int *my_ja, int *my_ia, double
   *x, double *result, int myRowsSize);
//! Subroutine for adding a vector to another multiplied by a
   constant
void daxpy(double *result, double alpha, double *x, double *y,
   int myRowsSize);
//! Subroutine for calculating the dot product of two vectors
double dot_product(double *x, double *y, int myRowsSize);


/*************************************************************/
/*                        Main function                      */
/*************************************************************/
int main (int argc, char* argv[]){

  int M, myid,numprocs,count,remainder,myRowsSize,NumofNonZero,
     num_e,iter;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);

  int *datacounts= NULL, *datadispls= NULL, *ecounts= NULL,
     *edispls= NULL, *ia= NULL, *ja= NULL;

  double  *aa=NULL, *b= NULL, *x , *res ;
  clock_t startTime, endTime;

  //read the matrix of file
  if(myid == 0){
    ifstream matrixfile("matrix_csr");
    if(!(matrixfile.is_open())){
      cout<< "Error: file not found"<<endl;
      return 0;
    }
    matrixfile >> M;
    matrixfile >> NumofNonZero;

```

```
    aa = new double [NumofNonZero];
    ja = new int [NumofNonZero];
    ia = new int [M+1];
    b = new double [M];

    for(int i = 0; i < NumofNonZero; i++){
      matrixfile >> aa[i];
    }

    for(int i = 0; i < NumofNonZero; i++){
      matrixfile >> ja[i];
    }

    for(int i = 0; i <= M ; i++){
      matrixfile >> ia[i];
    }

    for(int i = 0; i < M; i++){
      matrixfile >> b[i];
    }
    matrixfile.close();

    datacounts = new int[numprocs];
    datadispls = new int[numprocs];
    ecounts = new int[numprocs];
    edispls = new int[numprocs];

    count = M / numprocs;
    remainder = M - count * numprocs;

    int prefixSum = 0;
    for (int i = 0; i < numprocs; ++i) {
      int t1 = (i < remainder) ? count + 1 : count;
      datacounts[i] = t1;
      int t2 = prefixSum;
      datadispls[i] = t2;
      prefixSum += t1;
      ecounts[i] = ia[t2 + t1] - ia[t2];
      edispls[i] = ia[t2];
    }
  }
  MPI_Bcast(&M, 1, MPI_INT, 0, MPI_COMM_WORLD);

  if(myid != 0){
    count = M / numprocs;
    remainder = M - count * numprocs;

    datacounts = new int[numprocs];
    datadispls = new int[numprocs];
  }
  MPI_Bcast(datacounts, numprocs, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast(datadispls, numprocs, MPI_INT, 0, MPI_COMM_WORLD);

  MPI_Scatter(
    ecounts,
    1,
```

```
  MPI_INT,
  &num_e,
  1,
  MPI_INT,
  0,
  MPI_COMM_WORLD);

myRowsSize = myid < remainder ? count + 1 : count; // own size

double *bPart = new double[myRowsSize];
double *my_aa = new double[num_e];
int *my_ja = new int[num_e];
int *my_ia = new int[myRowsSize+1];

//! Scatter matrix entries to each processor
MPI_Scatterv(
  aa,
  ecounts,
  edispls,
  MPI_DOUBLE,
  my_aa,
  num_e,
  MPI_DOUBLE,
  0,
  MPI_COMM_WORLD);

//! Scatter partial Column Index
MPI_Scatterv(
  ja,
  ecounts,
  edispls,
  MPI_INT,
  my_ja,
  num_e,
  MPI_INT,
  0,
  MPI_COMM_WORLD);

//! Scatter partial Row pointers
MPI_Scatterv(
  ia,
  datacounts,
  datadispls,
  MPI_INT,
  my_ia,
  myRowsSize,
  MPI_INT,
  0,
  MPI_COMM_WORLD);

int Offset = my_ia[0];
for (int i = 0; i < myRowsSize; i++){
  my_ia[i] = my_ia[i] - Offset;
}
my_ia[myRowsSize] = num_e;
```

```
//! Scatter vector entries to each processor
MPI_Scatterv(
  b,
  datacounts, //t1
  datadispls, //t2
  MPI_DOUBLE,
  bPart,
  myRowsSize,
  MPI_DOUBLE,
  0, //root rank
  MPI_COMM_WORLD);

if(myid == 0){
  delete []aa;
  delete []ja;
  delete []ia;
  delete []b;
  delete []ecounts;
  delete []edispls;
}

x = new double [myRowsSize];
res = new double [M];

//! Solve the system using the Conjugate Gradient method
startTime = clock();
iter=conjugate_gradient(my_aa, my_ja, my_ia, bPart, x, datacounts,
    datadispls, myRowsSize, M);
endTime = clock();

//! concatenate solution
MPI_Allgatherv(
  x,
  myRowsSize,
  MPI_DOUBLE,
  res,
  datacounts,
  datadispls,
  MPI_DOUBLE,
  MPI_COMM_WORLD
);

//! Generate files for debug purpouse
if(myid == 0){
  ofstream Af;
  Af.open("mpi");
  for (int i = 0; i < M; i++) {
    Af<<res[i]<<endl;
  }
  Af.close();
  //! Print the total time
  double algorithmLen = double(endTime - startTime) /
      CLOCKS_PER_SEC;
  cout<< "\nAlgorithm Time "<< algorithmLen*1000.0<< " ms."<<
      endl;
  printf("result saved on file mpi.\n");
```

```
    printf("exit from iteration= %d\n",iter);
  }

  //! Deallocation of all dynamic memory in the program.
  delete []my_aa;
  delete []my_ja;
  delete []my_ia;
  delete []bPart;
  delete []x;
  delete []res;
  delete []datacounts;
  delete []datadispls;

  MPI_Finalize();
  return 0;
}

int conjugate_gradient(double *my_aa, int *my_ja, int *my_ia,
    double *bPart, double *x, int *datacounts, int *datadispls,
    int myRowsSize, int M) {

  //! Initial variables.
  int iter = 0;
  double *tmp, *r ,*p;
  double alpha, beta, result, resid=0.0, resid2=0.0;

  const double MaxIter = M*M;
  const double eps = 1e-6;

  //! Allocate auxiliary vectors for the CG algorithm
  r = new double [myRowsSize];
  p = new double [myRowsSize];
  tmp = new double [myRowsSize];

  for (int i = 0; i < myRowsSize; i++) {
    x[i] = 0;
  }

  //! Initialize the r0 and p0
  double *xx= new double[M];
  MPI_Allgatherv(
    x,
    myRowsSize,
    MPI_DOUBLE,
    xx,
    datacounts,
    datadispls,
    MPI_DOUBLE,
    MPI_COMM_WORLD);

  MatrixMulVec(my_aa, my_ja, my_ia, xx, tmp, myRowsSize);
  daxpy(r, -1.0, tmp, bPart, myRowsSize);

  memcpy(p,r,myRowsSize*sizeof(double));
  while(true && iter < MaxIter){
    //! CG iteration
```

```
    iter ++;                                                                    ۲۷۵
                                                                                ۲۷۶
    MPI_Allgatherv(                                                             ۲۷۷
      p,                                                                        ۲۷۸
      myRowsSize,                                                               ۲۷۹
      MPI_DOUBLE,                                                               ۲۸۰
      xx,                                                                       ۲۸۱
      datacounts,                                                               ۲۸۲
      datadispls,                                                               ۲۸۳
      MPI_DOUBLE,                                                               ۲۸۴
      MPI_COMM_WORLD);                                                          ۲۸۵
                                                                                ۲۸۶
    //! matrix-vector product                                                   ۲۸۷
    MatrixMulVec(my_aa, my_ja, my_ia, xx, tmp, myRowsSize);                     ۲۸۸
                                                                                ۲۸۹
    //! Compute the alpha step                                                  ۲۹۰
    result = dot_product(r,r,myRowsSize);                                       ۲۹۱
    MPI_Allreduce(                                                              ۲۹۲
      &result,                                                                  ۲۹۳
      &resid,                                                                   ۲۹۴
      1,                                                                        ۲۹۵
      MPI_DOUBLE,                                                               ۲۹۶
      MPI_SUM,                                                                  ۲۹۷
      MPI_COMM_WORLD);                                                          ۲۹۸
                                                                                ۲۹۹
    result = dot_product(p,tmp,myRowsSize);                                     ۳۰۰
    MPI_Allreduce(                                                              ۳۰۱
      &result,                                                                  ۳۰۲
      &resid2,                                                                  ۳۰۳
      1,                                                                        ۳۰۴
      MPI_DOUBLE,                                                               ۳۰۵
      MPI_SUM,                                                                  ۳۰۶
      MPI_COMM_WORLD);                                                          ۳۰۷
                                                                                ۳۰۸
    alpha = resid/resid2;                                                       ۳۰۹
                                                                                ۳۱۰
    //! Update the value of the solution                                        ۳۱۱
    daxpy(x,alpha,p,x,myRowsSize);                                              ۳۱۲
                                                                                ۳۱۳
    //! Residual update (tmp still holds A*p)                                   ۳۱۴
    daxpy(r,-1.0*alpha,tmp,r,myRowsSize);                                       ۳۱۵
                                                                                ۳۱۶
    //! Check the convergence criterion                                         ۳۱۷
    result = dot_product(r, r, myRowsSize);                                     ۳۱۸
    MPI_Allreduce(                                                              ۳۱۹
      &result,                                                                  ۳۲۰
      &resid2,                                                                  ۳۲۱
      1,                                                                        ۳۲۲
      MPI_DOUBLE,                                                               ۳۲۳
      MPI_SUM,                                                                  ۳۲۴
      MPI_COMM_WORLD);                                                          ۳۲۵
                                                                                ۳۲۶
    double norm = sqrt(resid2);                                                 ۳۲۷
    if (norm < eps) break;                                                      ۳۲۸
                                                                                ۳۲۹
    //! Compute the beta value                                                  ۳۳۰
```

```
    beta =  resid2/ resid;
    //! Update the p vector
    daxpy(p,beta,p,r,myRowsSize);
  }
  //! Free memory and exit
  delete []tmp;
  delete []p;
  delete []r;
  return iter;
}

void MatrixMulVec(double *my_aa, int *my_ja, int *my_ia, double
    *x, double *result, int myRowsSize) {

  //! Compute the matrix-vector product
  for (int i = 0; i < myRowsSize; i++) {
    result[i] = 0.0;
    for (int j = my_ia[i]; j < my_ia[i+1]; j++) {
      result[i] += my_aa[j]*x[my_ja[j]];
    }
  }
}

void daxpy(double *result, double alpha, double *x, double *y,
    int myRowsSize) {

  //! Compute the addition of a vector with a vector times a
      constant
  for(int i=0 ; i<myRowsSize ;i++){
    result[i] = alpha*x[i] + y[i];
  }
}

double dot_product(double *x, double *y, int myRowsSize) {

  //! Compute the dot product of the provided vectors
  double result = 0.0;
  for(int i=0 ; i<myRowsSize ;i++){
    result += x[i]*y[i];
  }
  return result;
}
```

واژه‌نامه فارسی به انگلیسی

اشتراکی . . . . . shared

بردار . . . . . vector

بعد . . . . . dimesion

پایه . . . . . base

پردازنده . . . . . processor

تنک . . . . . sparse

توزیعی . . . . . distributed

حافظه . . . . . memory

دستگاه معادلات خطی . . . . . system of linear equations

رشته،نخ . . . . . thread

زیرفضا . . . . . subspace

سری . . . . . sequential

گرادیان مزدوج . . . . . conjugate gradient

ماتریس . . . . . matrix

متعامد . . . . . orthogonal

متقارن . . . . . symmetric

معین مثبت . . . . . positive difinite

موازی . . . . . parallel

واژه‌نامه انگلیسی به فارسی

پایه . . . . . base

گرادیان مزدوج . . . . . conjugate gradient

بعد . . . . . dimesion

توزیعی . . . . . distributed

ماتریس . . . . . matrix

حافظه . . . . . memory

متعامد . . . . . orthogonal

موازی . . . . . parallel

معین مثبت . . . . . positive difinite

پردازنده . . . . . processor

سری . . . . . sequential

اشتراکی . . . . . shared

تنک . . . . . sparse

زیرفضا . . . . . subspace

متقارن . . . . . symmetric

دستگاه معادلات خطی . . . . . system of linear equations

رشته،نخ . . . . . thread

بردار . . . . . vector

Hakim Sabzevari University

An Outline of MSc. Thesis

Surname:Asaadi	Name:Zahra	Student No.:9713185316

Supervisor: Dr. Amin Rafiei

Advisor: Dr. Mahmood Amintoosi

Faculty of Mathematics and Computer Science

Program: Computer Science Field:Decision Science and Knowledge

Title of thesis: Parallel version of the conjugate gradient method in distributed and shared environments

Keywords:conjugate gradient, parallel shared and distributed machines, Krylov subspace methods

Abstract:

Today one of the most important problems in scientific computation is solving linear equation systems $Ax = b$. Directive or iterative methods can be used to solve these systems. The conjugate gradient method is one of the iterative methods based on Krylov subspace where it can be used to approximate the solution of the system. In high dimension problems, we will incur a lot of time and money to solve this system. Therefore, reducing computational time to increase efficiency has always been of interest to researchers. With the advent of parallel architecture in computers, parallel programming has significantly reduced computing time. In this thesis, we discuss in detail the implementation of the conjugate gradient algorithm in each of the OpenMP and MPI parallel environments. In the following, using artificial systems, we analyze the execution time of the program in each of these environments, and compare the execution time with different number of cores.

Hakim Sabzevari University
Faculty of Mathematics and Computer Science

A Thesis Submitted in Partial Fulfilment of the Requirement for the Degree of Master of Science in Computer Science

Parallel version of the conjugate gradient method in distributed and shared environments

Supervisor:
Dr. Amin Rafiei

Advisor:
Dr. Mahmood Amintoosi

By:
Zahra Asaadi

October 2021